Pensare in C++, seconda ed. Volume 1

1: Introduzione agli Oggetti

La genesi della rivoluzione del computer fu in una macchina. La genesi dei linguaggi di programmazione tende quindi ad assomigliare a quella macchina.

Ma i computer non sono tanto macchine quanto strumenti di ampliamento della mente (biciclette per la mente, come ama dire Steve Jobs) ed un diverso tipo di mezzo espressivo. Di conseguenza, gli strumenti stanno cominciando ad assomigliare meno alle macchine e di più a parti della nostra mente e agli altri mezzi espressivi come la scrittura, la pittura, scultura, animazione e la produzione dei film. La programmazione orientata agli oggetti è parte di questo movimento che va verso l'utilizzo del computer come mezzo espressivo.

Questo capitolo introdurrà i concetti base della programmazione orientata agli oggetti (object-oriented programming OOP), inclusa una panoramica dei metodi di sviluppo della OOP. Questo capitolo, e questo libro, presuppone che si abbia esperienza in un linguaggio di programmazione procedurale, sebbene non necessariamente il C. Se si pensa di aver bisogno di maggiore preparazione della programmazione e della sintassi del C prima di affrontare questo libro, si dovrebbe cominciare da *Thinking in C: Foundations for C++ and Javatraining CD ROM*, allegato a questo libro e disponibile anche su *www.BruceEckel.com*.

Questo capitolo fornisce materiali di inquadramento generale e materiali complementari. Molte persone non si sentono a proprio agio nell'affrontare la programmazione orientata agli oggetti senza prima essersi fatta un'idea del quadro generale. Per questa ragione, qui si presentano molti concetti atti a dare una buona panoramica della OOP. Per contro, molti non riescono ad afferrare i concetti di carattere generale se prima non hanno visto qualche aspetto concreto dei meccanismi; questo genere di persone potrebbe impantanarsi e smarrirsi, se non gli si presenta un po' di codice sul quale mettere le mani. Se si appartiene al secondo gruppo e non si vede l'ora di passare alle specifiche del linguaggio, si salti pure la lettura di questo capitolo: ciò non impedirà di scrivere programmi né di imparare il linguaggio. Tuttavia, si farà bene a tornare qui, alla fine, per completare le proprie conoscenze e capire perché gli oggetti sono importanti e come si fa a progettare con essi.

Il progresso dell'astrazione

Tutti i linguaggi di programmazione forniscono astrazioni. Si può dire che la complessità dei problemi che si possono risolvere è direttamente correlata al tipo e qualità di astrazione. Per tipo intendiamo: *"Cos'è che stiamo astraendo?"*. Il linguaggio assembler è una piccola astrazione della macchina che è alla base. I molti cosidetti linguaggi imperativi che seguirono (come il Fortran, Basic ed il C) furono astrazioni del linguaggio assembler. Questi linguaggi sono un grosso miglioramento del linguaggio assembler, ma la loro astrazione primaria richiede ancora che si pensi in termini della struttura del computer piuttosto che la struttura del problema che si sta tentando di risolvere. Il programmatore deve stabilire l'associazione tra il modello della macchina (nello spazio delle soluzioni, che è lo spazio dove si sta modellando il problema) ed il modello del problema che si sta risolvendo al momento (nello spazio del problema, che è il posto dove esiste il problema). Lo sforzo richiesto per eseguire questa associazione ed il fatto che è estrinseco al linguaggio

di programmazione, produce programmi che sono difficili da scrivere e da manuntenere e come effetto collaterale l'intera industria dei "metodi di programmazione".

L'alternativa alla modellazione della macchina è modellare il problema che si sta tendando di risolvere. I primi linguaggi come il LISP e l'ASP sceglievano particolari visioni del mondo("Tutti i problemi sono alla fine liste" oppure "Tutti i problemi sono algoritmici"). Il PROLOG riconduce tutti i problemi a catene di decisioni. I linguaggi sono stati creati per la programmazione basata su vincoli e per programmare esclusivamente manipolando simboli grafici (l'ultima si è dimostrata essere troppo restrittiva). Ognuno di questi approcci è una buona soluzione ad una particolare classe di problemi che essi risolvono, ma quando si esce fuori dal loro dominio essi diventano goffi.

L'approccio orientato agli oggetti fa un passo più avanti per il programmatore nel rappresentare gli elementi nello spazio del problema. Questa rappresentazione è sufficientemente generale da non limitare il programmatore ad un particolare tipo di problema. Ci riferiferiamo agli elementi nello spazio del problema e le loro rappresentazioni nello spazio delle soluzioni come oggetti (naturalmente, si avrà bisogno di altri oggetti che non hanno analoghi spazi di problema). L'idea è che un programma può adattarsi al gergo del problema aggiungendo nuovi tipi di oggetti, così quando si legge il codice che descrive la soluzione, si leggono le parole che esprimono anche il problema. C'è un linguaggio di astrazione più flessibile e potente di ciò che abbiamo avuto prima. Così, la OOP permette di descrivere il problema in termini del problema, piuttosto che in termini del computer dove verra attuata la soluzione. C'è ancora un collegamento al computer, tuttavia. Ogni oggetto assomiglia un pò ad un piccolo computer; esso ha uno stato e delle operazioni che gli si possono chiedere di compiere. In fondo, non è impropria l’analogia con gli oggetti del mondo reale: anch’essi hanno caratteristiche e comportamenti.

Qualche progettista di linguaggi ha deciso che la programmazione orientata agli oggetti di per se non è adeguata a risolvere facilemente tutti i problemi di programmazione e difende la combinazione di vari approcci nei linguaggi di programmazione multiparadigma.[4]

Alan Kay ha ricapitolato cinque caratteristiche base dello Smalltalk, il primo linguaggio orientato agli oggetti che ha avuto successo ed uno dei linguaggi sul quale il C++ è basato. Queste caratteristiche rappresentano un approccio puro alla programmazione orientata agli oggetti:

1. **Tutto è un oggetto.** Si pensi ad un oggetto come una variabile particolare; memorizza dati, ma si possono fare richieste a quel oggetto, chiedendo di eseguire operazioni su se stesso. In teoria, si può prendere qualsiasi componente concettuale nel problema che si sta cercando di risolvere (cani, edifici, servizi, ecc..) e rappresentarlo come un oggetto nel nostro programma.
2. **Un programma è un gruppo di oggetti che si dicono cosa fare l'un altro scambiandosi messaggi. Per fare una richiesta ad un oggetto, si manda un messaggia a quell'oggetto.** Più concretamente, si può pensare al messaggio, come una richiesta ad una chiamata a funzione che appartiene ad un particolare oggetto. Ogni oggetto ha la sua memoria fatta di altri oggetti. Messo in altro modo, si creano nuovi tipi di oggetti utilizzando altro oggetti esistenti. Così, si può costruire un programma complesso usando la semplicità degli oggetti.

3. **Ogni oggetto ha un tipo.** Usando il gergo, ogni oggetto è una istanza di una classe, in cui classe è sinonimo di tipo. La caratteristica più importante distinguente di una classe è *"Quali messaggi le si possono mandare?"*.
4. **Tutti gli oggetti di un particolare tipo possono ricevere gli stessi messaggi.** Questa espressione verrà chiarita meglio in seguito. Poichè un oggetto di tipo cerchio è anche un oggetto di tipo forma, è garantito che un cerchio accetti i messaggi per la forma. Ciò significa che si può scrivere codice che parla alle forme e che gestisce automaticamente qualsiasi altra forma. Questa *sostituibilità* è uno dei concetti più potenti della OOP.

Un oggetto ha un'interfaccia

Aristotele fu probabilmente il primo a cominciare un attento studio del concetto di tipo;egli parlò della *"classe dei pesci e la classe degli uccelli"*. L'idea che tutti gli oggetti, pur essendo unici, sono anche parte di una classe di oggetti che hanno caratteristiche e comportamenti in comune fu usata direttamente nel primo linguaggio orientato agli oggetti, Simula-67, con la sua parola riservata fondamentale **class** che introduce un nuovo tipo in un programma.

Simula, come implica il suo nome, fu creato per sviluppare simulazioni come il classico problema dello sportello bancario[5].” In questo si hanno un gruppo di sportelli, clienti, conti, transazioni ed unità di monetà: molti oggetti. Gli oggetti, che sono identici eccetto per il loro stato durante l'esecuzione di un programma, sono raggruppati insieme in classi di oggetti ed ecco da dove viene fuori la parole riservata **class**. Creare tipi di dato astratti (classi) è un concetto fondamentale nella programmazione orientata agli oggetti. I tipi di dato astratti funzionano quasi come i tipi predefiniti: si possono creare variabili di un tipo (chiamati *oggetti* o *istanze* nel parlato orientati agli oggetti) e manipolare queste variabili (chiamate *invio di messaggi* o *richieste*; si manda un messaggio e l'oggetto capisce cosa farne di esso). I membri (elementi) di ogni classe condividono alcuni aspetti comuni: ogni conto ha un saldo, ogni sportello accetta un deposito, ecc.. Allo stesso tempo, ogni membro ha il suo proprio stato, ogni conto ha un suo diverso saldo, ogni sportello ha un nome. Così, gli sportelli, i clienti, i conti , le transazioni, ecc..., possono essere rappresentati con un unica entità nel programma del computer. Questa entità è un oggetto ed ogni oggetto appartiene ad una particolare classe che definisce le sue caratteristiche e comportamenti.

Quindi, sebbene ciò che si fa realmente nella programmazione orientata agli oggetti è creare nuovi tipi di dato, virtualmente tutti i linguaggi di programmazione orientata gli oggetti usano la parola riservata **class**. Quando si vede la parola "*type*" si pensi a "*class*" e viceversa[6].

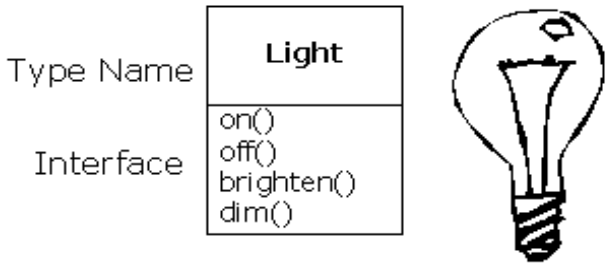
Poichè una classe descrive un insieme di oggetti che hanno caratteristiche identiche (elementi dato) e comportamenti (funzionalità), una classe è realmente un tipo di dato perchè un numero in virgola mobile, per esempio, ha anche un insieme di caratteristiche e comportamenti. La differenza è che un programmatore definisce una classe per adattarla ad un problema piuttosco che dover usare tipi di dato esistenti per i propri bisogni. Il sistema di programmazione dà il benvenuto a classi nuove e le dà tutte le attenzioni ed il controllo sul tipo che viene dato ai tipi predefiniti.

L'approccio orientato agli oggetti non è limitato alla costruzione di simulazioni. Se si è d'accordo o meno che qualsiasi programma è una simulazione del sistema che si sta

progettando, l'uso delle tecniche OOP può facilmente ridurre un grande insieme di problemi ad una semplice soluzione.

Una volta che si è terminata una classe, si possono fare quanti oggetti si vogliono e poi manipolarli come se essi fossero elementi che esistono nel problema che si sta tentando di risolvere. Infatti, una delle sfide della programmazione orientata agli oggetti è creare una corrispondenza uno a uno tra gli elementi dello spazio del problema e gli oggetti dello spazio della soluzione.

Ma come si ottiene un oggetto che sia utile per il nostro lavoro? Ci deve essere un modo di fare una richiesta all'oggetto così che esso faccia qualcosa, come completare una transazione, disegnare qualcosa sullo schermo o azionare un interruttore. Ed ogni oggetto può soddisfare solo alcune richieste. Le richieste che si possono fare ad un oggetto sono definite dalla sua *interfaccia* ed il tipo è ciò che determina la sua interfaccia. Un semplice esempio potrebbe essere una rappresentazione di una lampadina:




```
Lampadina lt;
lt.accendi();
```

L'interfaccia stabilisce *quali* richeste si possono fare ad un particolare oggetto. Tuttavia, ci deve essere codice da qualche parte per soddisfare quella richiesta. Questo, insieme con i dati nascosti, include l'*implementazione*. Da un punto di vista della programmazione procedurale non è poi tanto complicato. Un tipo ha una funzione associata con ogni possibile richiesta e quando si fa una particolare richiesta ad un oggetto quella funzione viene chiamata. Questo processo è di solito riassunto dicendo che si "manda un messaggio" (si fa un richiesta) ad un oggetto e l'oggetto capisce cosa fare con quel messaggio (esegue il codice).

Qui, il nome del tipo/classe è **Lampadina**, il nome di questa particolare oggetto **Lampadina** è **lt** e le richieste che si possono fare ad una **Lampadina** sono di accenderla, di spegnerla, è di aumentare o diminuire l'intensità della luce. Si crea un oggetto **Lampadina** dichiarando un nome (**lt**) per quell'oggetto. Per mandare un messaggio all'oggetto, si dichiara il nome dell'oggetto e lo si connette alla richiesta del messaggio con un punto. Dal punto di vista dell'utente di una classe predefinita, è tutto ciò che si deve programmare con gli oggetti.

Il diagramma mostrato sopra segue il formato dell' *Unified Modeling Language* (UML). Ogni classe è rappresentata da una scatola, con il nome del tipo in cima, i membri dato nella porzione di mezzo e le *funzioni membro* (le funzione che appartenfono a quest'oggetto, che ricevono tutti i messaggi che si mandano a quell'oggetto) nella parte bassa. Spesso, nei diagrammi di disegno UML sono mostrati solo il nome della classe e i membri funzione publici, mentre la parte centrale non compare. Se si è interessati solo al nome della classe, allora non c'è bisogno di indicare neanche la porzione inferiore.

L'implementazione nascosta

È utile separare i campi di gioco in *creatori di classi* (quelli che creano nuovi tipi di dato) e *programmatori client*[7] (gli utilizzatori di classi che usano i tipi di dato nelle loro applicazioni). Lo scopo del programmatore client è di raccogliere un insieme completo di classe da usare per un rapido sviluppo delle applicazioni. Lo scopo del creatore di classi è di costruire una classe che espone solo ciò che è necessario al programmatore client e mantiene tutto il resto nascosto. Perchè? Perchè se è nascosta, il programmatore client non può usarla, cioè il creatore della classe può cambiare la parte nascosta a suo desiderio senza preoccuparsi dell'impatto verso chiunque altro. La parte nascosta di solito rappresenta la parte più fragile di un oggetto e può essere facilmente alterata da un programmatore client sbadato o ignaro, perciò nascondere l'implementazione riduce i bug nei programmi. Il concetto di implementazione nascosta non va preso alla leggera.

In qualsiasi relazione è importante avere dei limiti che siano rispettati da tutte le parti coinvolte. Quando si crea una libreria, si stabilisce una relazione con il programmatore client, che è un programmatore, ma è anche uno che sta realizzando un' applicazione usando la nostra libreria oppure per costruire una libreria più grande.

Se i membri di una classe sono disponibili a tutti, allora il progammatore client può fare qualsiasi cosa con quella classe e non c'è modo di imporre regole. Anche se si preferirebbe che il programmatore client non manipolasse direttamente alcuni membri della nostra classe , senza il controllo del accesso non c'è modo di prevenirlo. È tutto alla luce del sole.

Quindi la prima ragione per il controllo dell'accesso è di fare in modo che i programmatori client non mettano le mani in parti che sono necessarie per il funzionamento interno dei tipi di dato, ma che non fanno parte dell'interfaccia di cui gli utenti hanno bisogno per risolvere i loro particolari problemi. Questo è un servizio reso agli utenti perchè essi possono facilmente vedere cosa è importante per loro e cosa possono ignorare.

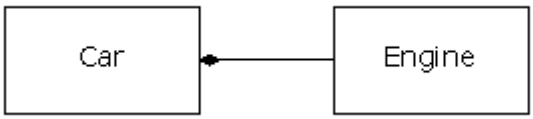
La seconda ragione per il controllo dell'accesso è di permettere al progettista della libreria di cambiare il funzionamento interno della classe senza preoccuparsi di come esso influirà sul programmatore client. Per esempio, si può implementare una particolare classe in una maniera semplice per facilitare lo sviluppo e scoprire più tardi che c'è bisogno di riscriverla per renderla più veloce. Se l'interfaccia e l'implementazione sono chiaramente separate e protette, si può realizzare ciò facilmente e richiedere un nuovo linkaggio dall'utente.

Il C++ usa tre esplicite parole riservate per impostare i limiti di una classe: **public**, **private** e **protected**. Il loro uso e significato sono abbastanza semplici. Questi *specificatori di accesso* determinano chi può usare le definizioni che seguono. **public** significa che le definizioni seguenti sono disponibili a tutti. La parola chiave **private** , dall'altro lato, significa che nessuno può accedere a quelle definizioni tranne noi, il creatore del tipo, dentro le funzioni membro di quel tipo. **private** è un muro tra noi ed il programmatore client. Se qualcuno cerca di accedere un membro private, otterrà un errore a tempo di compilazione. **protected** agisce proprio come private, con l'eccezione che una classe che eredità ha accesso ai membri protetti, ma non a quelli privati. L'ereditarietà sarà introdotta a breve.

Riutilizzo dell'implementazione

Una volta che una classe è stata creata e testata, dovrebbe idealmente rappresentare un' utile unità di codice. Ne consegue che questa riusabilità non è così facile da realizzare come molti spererebbero; ci vuole esperienza ed intuito per produrre un buon progetto, ma una volta che si ottiene ciò, esso ci implora di essere riusato. Il riuso del codice è una dei più grandi vantaggi che la programmazione orientata agli oggetti fornisce.

Il modo più semplice di riusare una classe è quello di usare un oggetto di quella classe direttamente, ma si può anche piazzare un oggetto di quella classe dentro una nuova classe. Ciò viene detto *"creazione di un oggetto membro"*. La nuova classe può essere composta da qualsiasi numero e tipo di oggetti, in qualsiasi combinazione di cui si ha bisogno per raggiungere la funzionalità desiderata nella nuova classe. Poichè si sta componendo una nuova classe da una esistente, questo concetto è chiamato *composizione* (o più generalemente, *aggregazione*). La composizione è spesso indicata come una relazione "*ha-un*", per esempio "*un' auto ha un motore*".



(Il diagramma UML di sopra indica la composizione con il rombo, che specifica che c'è una macchina. Si userà tipicamente una forma più semplice: solo una linea, senza il rombo, per indicare una associazione.[8])

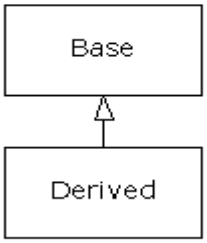
La composizione ha una grande flessibilità. Gli oggetti membro della nuova classe sono di solito privati, in modo da essere inaccessibili ai programmatori che useranno la classe. Ciò permette di cambiare quei membri senza disturbare il codice esistente scritto da chi ha utilizzato la classe. Si può anche cambiare gli oggetti membro a runtime, per cambiare dinamicamente il comportamento del programma. L'ereditarietà, che sarà descritta in seguito, non ha questa flessibilità poichè il compilatore deve porre restrizioni del tempo di compilazione sulle classi create con l'ereditarietà.

Poichè l'ereditarietà è molto importante nella programmazione orientata agli oggetti essa è spesso enfatizzata ed il programmatore novizio può avere l'impressione che essa debba essere usata dovunque. Il risultato potrebbe essere un progetto goffo e troppo complicato. Quando si creano nuove classi si dovrebbe, invece, prendere in considerazione in primo luogo la composizione, perché è più semplice e più flessibile. Se si segue questo approccio, si avranno progetti più puliti. Quando si avrà un po’ di esperienza, risulteranno evidenti i casi nei quali si avrà bisogno dell’ereditarietà.

Ereditarietà: riutilizzare l'interfaccia

Di per sè, l'idea di un oggetto è uno strumento pratico. Esso ci permette di impacchettare dati e funzionalità insieme con i concetti, così che si può descrivere un' appropriata idea dello spazio del problema piuttosto che dover usare gli idiomi che sottendono la macchina. Questi concetti sono espressi come unità fondamentali nel linguaggio di programmazione usando la parola chiave **class.**

È un peccato, tuttavia, andare incontro a tanti problemi per creare una classe e poi essere forzati a crearne una nuova che può avere delle funzionalità simili. È più carino se possiamo prendere una classe esistente, clonarla e poi aggiungere le modifiche al clone. Ciò effettivamente è quello che si ottiene con l'*ereditarietà*, con l'eccezione che se la classe originale (detta base o super o classe genitore) viene modificata, il clone modificato (chiamato la classe *derivata* o ereditata o *sub* o *figlia*) riflette anch' essa quei cambiamenti.

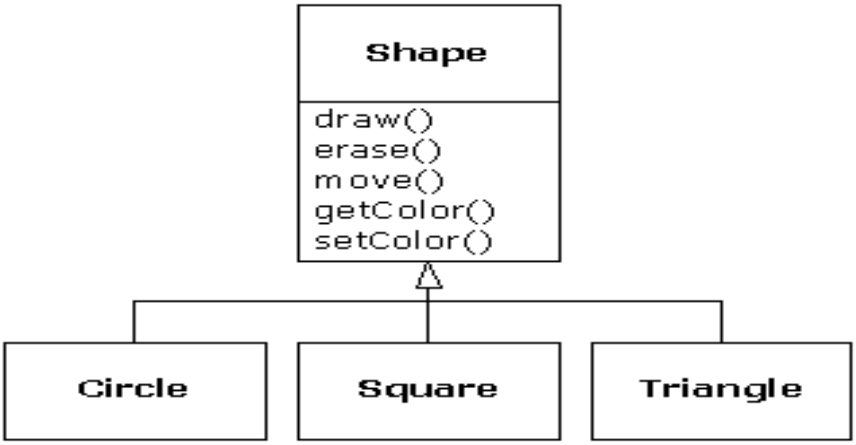


(La freccia del diagramma UML di sopra punta dalla classe derivata alla classe base. Come si vedrà, ci possono essere più di una classe derivata)

Un tipo fa più che descrivere i vincoli di un insieme di oggetti, esso ha anche una relazione con gli altri tipi. Due tipi possono avere caratteristiche e comportamenti in comune, ma un tipo può contenere più caratteristiche di un altro e può anche gestire più messaggi (o gestirne differenti). L'ereditarietà esprime questa similarità tra i tipi utilizzando il concetto di tipo base e tipo derivato. Un tipo base contiene tutte le caratteristiche ed i comportamenti che sono condivisi tra i tipi derivati da esso. Si crea un tipo base per rappresentare il nucleo delle idee di alcuni oggetti nel nostro sistema. Dal tipo base, si derivano altri tipi per esprimere diversi modi in cui questo nucleo può essere realizzato.

Per esempio, una macchina per il riciclo dei rifiuti mette in ordine pezzi di rifiuti. Il tipo base è il "rifiuto" ed ogni pezzo di rifiuto ha un peso, un valore e così via, e può essere sminuzzato, fuso o decomposto. Da ciò vengono derivati tipi più specifici di rifiuti che possono avere caratteristiche addizionali (una bottiglia ha un colore) o comportamenti (l'alluminio può essere schiacciato, una lattina di acciaio è magnetica). In più, alcuni comportamenti possono essere differenti (il valore della carta dipende dal suo tipo e condizione). Usando l'ereditarietà, si può costruire una gerarchia del tipo che esprime il problema che si sta cercando di risolvere in termini di tipi.

Un secondo esempio è il classico esempio della figura, forse usato in un sistema di disegno assististo al computer o in un gioco. Il tipo base è una figura ed ogni figura ha una misura, colore, posizione e così via. Ogni figura può disegnata, cancellata,mossa, colorata, ecc.. Da ciò, tipi specifici possono essere derivati (ereditati): cerchio, quadrati, triangoli e così via ciascuno dei quali può avere ulteriori caratteristiche e comportamenti. Alcuni comportamenti possono essere differenti, come quando si vuole calcolare l'area di una figura. La gerarchia del tipo incarna entrambe le similarità e differenze tra le figure.

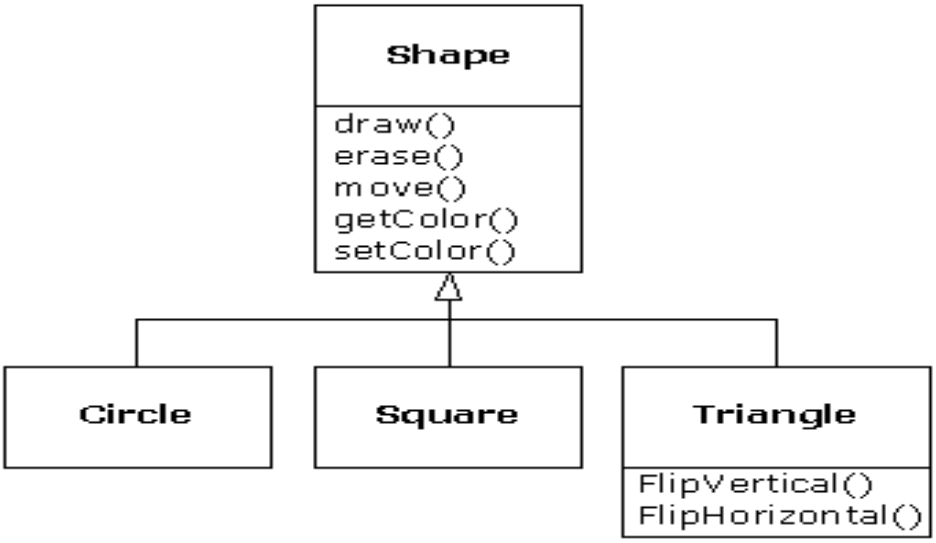


Proporre la soluzione negli stessi termini del problema ha un enorme beneficio perchè non c'è bisogno di modelli intermedi per passare da una descrizione del problema ad una descrizione della soluzione. Con gli oggetti, la gerarchia del tipo è un modello primario, quindi si va direttamente dalla descrizione del sistema nel mondo reale alla descrizione del sistema nel codice. Infatti una delle difficoltà che si hanno nel progetto orientato agli oggetti è che è troppo semplice andare dall'inizio alla fine. Una mente allenata ad affrontare soluzioni complesse rimane spesso perplesso da questa semplicità a principio.

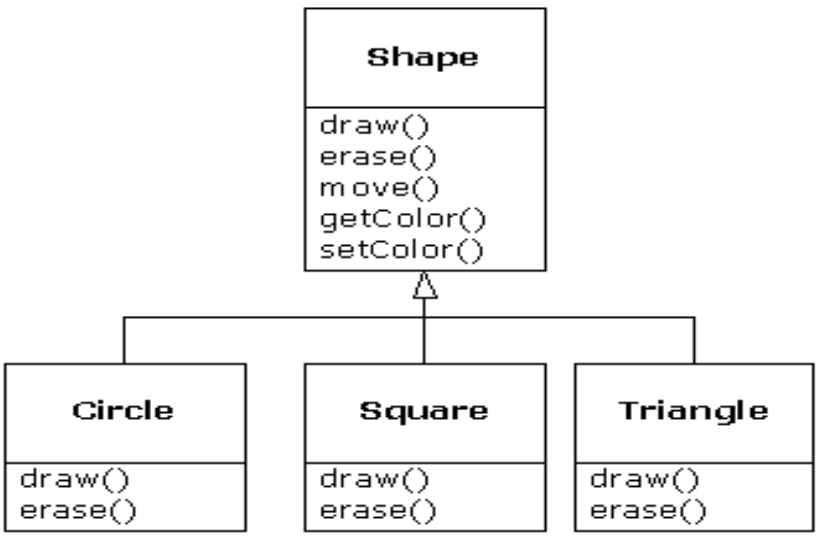
Quando si eredita da un tipo esistente, si crea un nuovo tipo. Il nuovo tipo contiene non solo tutti i membri del tipo esistente (sebbene quelli privati sono nascosti ed inaccessibili), ma cosa più importante esso duplica l'interfaccia delle classi base. Cioè, tutti i messaggi che si mandano agli oggetti della classe base vengono mandati anche alla classe derivata. Poichè conosciamo il tipo di una classe dai messaggi che le mandiamo, ciò significa che la classe derivata è dello *stesso tipo della classe base*. Nell'esempio precedente, un cerchio è un figura, questa equivalenza di tipo tramite l'ereditarietà è uno delle punti fondamentali per capire la programmazione orientata agli oggetti.

Poichè sia la classe base che la derivata hanno la stessa interfaccia, ci deve essere una qualche implementazione che accompagna l'interfaccia. Cioè, ci deve essere del codice da eseguire quando un oggetto riceve un particolare messaggio. Si semplicemente si eredita una classe e non si fa nient'altro, i metodi dell'interfaccia della classe base sono presenti nella classe derivata. Ciò significa che gli oggetti della classe derivata non hanno solo lo stesso tipo, ma anche lo stesso comportamento, che non è molto interessante.

Si hanno due modi per differenziare la propria nuova classe derivata dalla classe base originale. Il primo è molto diretto: aggiungere semplicemente funzioni nuove di zecca alla classe derivata. Queste nuove funzioni non fanno parte dell'interfaccia della classe base. Ciò vuol dire che la classe base non faceva tutto quello che si voleva che facesse e quindi sono state aggiunte più funzioni. Questo utilizzo semplice e primitivo dell'ereditarietà è, a volte, la soluzione perfetta per il proprio problema. Tuttavia, di dovrebbe verificare con cura la possibilità che anche la classe base possa aver bisogno di queste funzioni addizionali. Questo modo di procedere nella progettazione per scoperte e iterazioni accade regolarmente nella programmazione orientata agli oggetti.



Sebbene a volte l'ereditarietà possa implicare l'aggiunta di nuove funzioni all'interfaccia, ciò non è necessariamente vero. Il secondo e più importante modo di differenziare le nuove classi è *cambiare* il comportamento di una funzione di una classe base esistente. Ciò è indicato come *overriding* di quella funzione (ovvero forzare quella funzione).



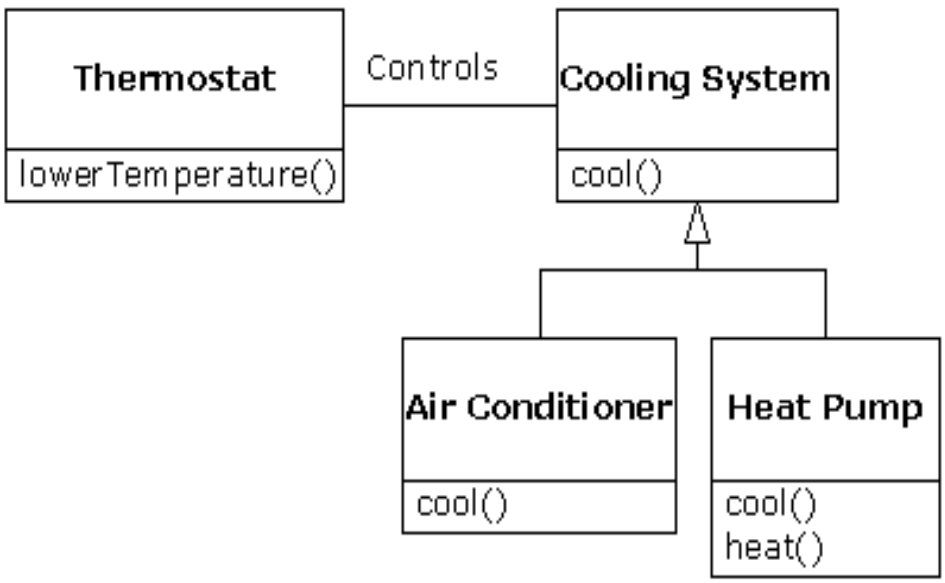
Per forzare una funzione, non si deve fare altro che creare una nuova definizione per quella funzione nella classe derivata. Ciò che si dice è "*Qui utilizzo la stessa funzione di interfaccia, però voglio fare qualcosa di diverso per il mio nuovo tipo*".

Confronto delle relazioni *"è-un"* ,*"è-come-un"*

C'è un certo dibattito sull'ereditarietà: dovrebbe l'ereditarietà ignorare solo le funzioni della classe base (e non aggiungere nuove funzioni membro che non sono nella classe base) ? Ciò significherebbe che il tipo derivato è esattamente lo stesso tipo della classe base poichè esso ha esattamente la stessa interfaccia. Come risultato si può sostituire esattamente un oggetto della classe derivata con un oggetto della classe base. Si può pensare a ciò come una *sostituzione pura* ed è spesso indicato come il *principio di*

sostituzione. In un senso, questo è un modo ideale di trattare l'ereditarietà. Spesso ci si riferisce alla relazione tra la classe base e le classi derivate in questo caso come una relazione *è-un*, perchè si può dire un cerchio è una figura. Un test per l'ereditarietà è determinare se si può specificare una relazione *è-un* per le classi ed essa ha senso.

Ci sono volte in cui si devono aggiungere nuovi elementi dell'interfaccia ad un tipo derivato, estendendo così l'interfaccia e creando un nuovo tipo. Il nuovo tipo può essere ancora sostituito con il tipo base, ma la sostituzione non è perfetta perchè le nuove funzioni non sono accessibili dal tipo base. Ciò può essere descritto come una relazione *è-come-un*; il nuovo tipo ha l'interfaccia del tipo vecchio ma contiene anche funzioni, quindi non si può dire veramente che è esattamente lo stesso. Per esempio, si consideri un condizionatore d'aria. Si supponga che la nostra casa sia cablata con i controlli per il raffreddamento, cioè ha un interfaccia che permette di controllare il raffreddamento. Si immagini che si rompa un condizionatore d'aria e che lo si rimpiazzi con la pompa di calore, che può sia riscaldare che raffreddare. La pompa di calore *è-come-un* condizionatore d'aria, ma può fare di più. Poichè il sistema di controlla della casa è stato progettato per controllare solo il raffreddamento, esso può solo comunicare con la parte del raffreddamento del nuovo oggetto. L'interfaccia del nuovo oggetto è stata estesa e il sistema esistente non conosce nient'altro che l'interfaccia originale.



Naturalmente, una volta che si nota questo progetto diventa chiaro che la classe base "sistema di raffreddamento"non è abbastanza generale e dovrebbe essere rinominata a "sistema di controllo della temperatura"in modo che si possa includere il riscaldamento al punto in cui valga il principio di sostituzione. Tuttavia, il diagramma di sopra è un esempio di cosa può succedere nel disegno e nel mondo reale.

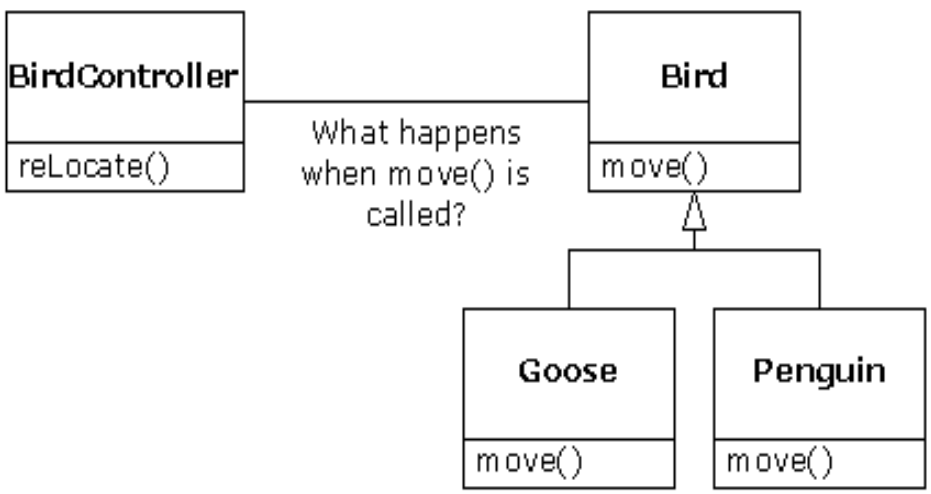
Quando si capisce il principio di sostituzione è facile sentire come questo approccio (sostituzione pura) sia l'unico modo di fare le cose ed infatti è carino se il nostro progetto è fatto così. Ma si scoprirà che ci sono volte che è ugualmente chiaro che si devono aggiungere nuove funzioni all'interfaccia della classe derivata. Con l'inspezione entrambi i casi dovrebbero essere ragionevolmente ovvi.

Oggetti intercambiabili con il polimorfismo

Quando si ha che fare con gerarchie di tipi, si vuole spesso trattare un oggetto non come il tipo specifico che è ma come il suo tipo base. Ciò permette di scrivere codice che non dipende da tipi specifici. Nell'esempio delle figure, le funzioni manipolano figure generiche senza preoccuparsi se esse sono cerchi, quadrati, triangoli e così via. Tutte le figure possono essere disegnate e mosse, così queste funzioni semplicemente mandano un messaggio all'oggetto della figura; esse non si preoccupano come gli oggetti fronteggiano il messaggio.

Tale codice non è affetto dall'aggiunta di nuovi tipi e l'aggiunta di nuovi tipi è il modo più comune di estendere un programma orientato agli oggetti per gestire nuove situazioni. Per esempio, si può derivare un nuovo sottotipo di figura chiamato pentagono senza modificare le funzioni che si occupano solo con figure generiche. Questa capacità di estendere facilmente un programma derivando nuovi sottotipi è importante perchè migliora enormemente il disegno e allo stesso tempo riduce il costo di manutenzione del software.

C'è un problema tuttavia, con il tentativo di trattare oggetti di tipi derivati come i loro generici tipi base (cerchi come figure, biciclette come veicoli, cormorani come uccelli, ecc..). Se una funzione dirà ad una generica funzione di disegnarsi o ad un generico veicolo di sterzare o ad un generico uccello di muoversi, il compilatore non può sapere a tempo di compilazione quale pezzo di codice sarà eseguito. Questo è il punto, quando un messaggio viene mandato, il programmatore non vuole sapere quale pezzo di codice sarà eseguito; la funzione di disegno sarà applicata ugualmente ad un cerchio, un quadrato o un triangolo e l'oggetto eseguirà l'esatto codice sul specifico tipo. Se non si deve conoscere quale pezzo di codice sarà eseguito, allora quando si aggiunge un nuovo sottotipo, il codice che esso esegue può essere diverso senza richiedere cambiamenti della chiamata della funzione. Perciò il compilatore non conosce precisamente quale pezzo di codice viene eseguito, quindi cosa fa? Per esempio, nel diagramma seguente l'oggetto **ControllorePennuto** funziona con l'oggetto generico **Pennuto** e non sa quale tipo esattamente sono. Ciò è conveniente dal punto di vista del **ControllorePennuto**, perchè non si deve scrivere codice speciale per determinare il tipo esatto di **Pennuto** con il quale si sta lavorando o il comportamento di quel **Pennuto**. Quindi ciò che accade, quando **sposta()** viene chiamato mentre si ignora il tipo specifico di **Pennuto**, si verificherà il giusto comportamento (un **Anitra** corre, vola, o nuota e un **Pinguino** corre o nuota?)



La risposta è il primo equivoco della programmazione orientata agli oggetti: il compilatore non può fare una chiamata a funzione nel senso tradizionale. La chiamata a funzione generata da un compilatore non OOP causa ciò che è chiamato *early binding*(concatenamento anticipato o statico), un termine che si può non aver sentito prima perchè non ci si è pensato mai in altro modo. Esso significa che il compilatore genera una chiamata ad un specifico nome di funzione ed il linker risolve questa chiamata ad un indirizzo assoluto del codice che deve essere eseguito. Nella OOP, il programma non può determinare l'indirizzo del codice fino al tempo di esecuzione, quindi qualche altro schema è necessario quando un messaggio viene mandato ad un generico oggetto.

Per risolvere il problema, i linguaggi orientati agli oggetti usano il concetto di *late binding* (concatenamento ritardato o dinamico). Quando si manda un messaggio ad un oggetto, il codice che viene chiamato non è determinato fino al tempo di esecuzione. Il compilatore non garantisce che la funzione esista ed esegue un controllo sul tipo sugli argomenti ed il valore di ritorno (un linguaggio in cui ciò non è vero è detto *weakly typed*, debolmente tipizzato), ma non conosce il codice esatto da eseguire.

Per eseguire il late binding, il compilatore C++ inserisce uno speciale bit di codice invece della chiamata assoluta. Questo codice calcola l'indirizzo del corpo della funzione, usando le informazioni memorizzate nell'oggetto (questo processo è spiegato in dettaglio nel Capitolo 15). Quindi, ogni oggetto può comportarsi diversamente a secondo del contenuto di quel bit speciale di codice. Quando si manda un messaggio ad un oggetto, l'oggetto non sa realmente cosa fare con quel messaggio.

Si dichiara che si vuole che una funzione abbia la flessibilità del late binding usando la parola chiave **virtual.** Non c'è bisogno di capire il meccanismo di **virtual** per usarlo, ma senza esso non si può programmare ad oggetti in C++. In C++, si deve ricordare di aggiungere **virtual**, perchè, per default, le funzioni membro non sono dinamicamente legate. Le funzioni virtuali ci permettono di esprimere una differenza di comportamento tra le classi di una stessa famiglia. Quelle differenze sono ciò che causano un comportamento polimorfico.

Si consideri l'esempio figura. La famiglia delle classi (tutte basate sulla stessa interfaccia uniforme) è stata diagrammata precedentemente nel capitolo. Per dimostrare il polimorfismo, vogliamo scrivere un singolo pezzo di codice che ignora i dettagli specifici del tipo e parla solo con la classe base. Quel codice è *disaccoppiato* dall'informazione specificata del tipo e quindi è più facile scrivere e più semplice capire. E se un nuovo tipo, un **Esagono**, per esempio viene aggiunto attraverso l'ereditarietà, il codice che si scrive funzionerà anche per il nuovo tipo di figura come faceva per i tipi esistenti. In questo modo il programma è *estendibile*.

Se si scrive una funzione in C++(come presto impareremo a fare):

```
void faiQualcosa(Figura& f) {
  f.elimina();
  // ...
  f.disegna();
}
```

Questa funzione parla ad ogni figura, quindi è indipendente dal tipo specifico dell'oggetto che viene disegnato e cancellato (la & significa "prendi l'indirizzo dell'oggetto che è passato a faiQualcosa()", ma non è importante che si capiscano questi dettagli ora). Se in qualche parte del programma usiamo la funzione **faiQualcosa():**

```
Cerchio c;
Triangolo t;
Linea l;
faQualcosa(c);
faQualcosa(t);
faQualcosa(l);
```

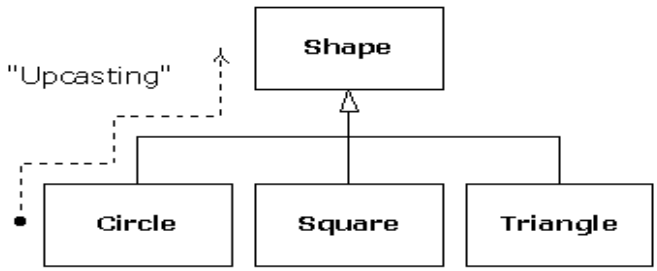
La chiamata a faQualcosa() funziona bene automaticamente, a dispetto del tipo esatto di oggetto.

È veramente un trucco fantastico. Si consideri la linea:

```
faiQualcosa(c);
```

Ciò che accade qui è che un Cerchio viene passato ad una funzione che si aspetta una Figura. Poichè un Cerchio è una Figura, esso può essere trattato come uno da faiQualcosa(). Cioè, qualsiasi messaggio che faQualcosa() può mandare ad una Figura, Cerchio lo può accettare. Quindi è una cosa completamente sicura e logica da fare.

Chiamiamo **upcasting** questo modo di trattare un tipo derivato come se fosse il suo tipo base. Nel nome cast è usato nel senso letterale inglese, fondere in uno stampo, e up viene dal modo in cui normalmente viene disposto il diagramma dell'ereditarietà, con il tipo base in testa e le classi derivate disposte a ventaglio verso il basso. Quindi, fare il casting su un tipo base significa risalire lungo diagramma dell'ereditarietà: "upcasting".



Un programma ad oggetti contiene qualche upcasting da qualche parte, perchè questo è il modo in cui ci si disaccoppia dal sapere qual è l'esatto tipo con cui si sta lavorando. Si guardi nel codice a **faiQualcosa()**:

```
f.elimina();
// ...
f.disegna();
```

Si noti che esso non dice: "Se sei un **Cerchio**, fai questo, se sei un **Quadrato**, fai quello, ecc..". Se si scrive questo tipo di codice, che controlla per tutti i possibili tipi di una **Figura** che si possono avere, è una grande confusione e si ha bisogno di cambiare il codice ogni volta che si aggiunge un nuovo tipo di **Figura**. Qui, diciamo solo: "Tu sei una figura, so che puoi eseguire **elimina()** , **disegna()**, fallo e fai attenzione ai dettagli correttamente".

Ciò che colpisce del codice in **faiQualcosa()** è che, in qualche modo, va tutto per il verso giusto. Chiamando **disegna()** per **Cerchio** si ottiene l'esecuzione di un codice diverso da quello eseguito quando si chiama **disegna()** per un **Quadrato** o **Triangolo**, ma quando il messaggio **disegna()** viene mandato ad una **Figura** anonima, il corretto comportamento avviene basandosi sul tipo effettivo di **Figura**. Ciò è sbalorditivo perchè,

come menzionato prima, quando il compilatore C++ sta compilando il codice per **faiQualcosa()**, non sa esattamente con che tipo sta trattando. Perciò di solito, ci si aspetterebbe che esso chiamasse le versioni di **elimina()** e **disegna()** per Figura e non in specifico per **Cerchio, Quadrato** o **Triangolo**. E ancora tutto va come deve andare grazie al polimorfismo. Il compilatore e il sistema a runtime gestiscono i dettagli; tutto ciò che si deve sapere è che funziona e, cosa più importante, come progettare. Se una funzione membro è **virtual**, allora quando si manda un messaggio ad un oggetto, questo farà la cosa giusta, anche quando è coinvolto l'upcasting.

Creazione e distruzione di oggetti

Tecnicamente, il dominio della OOP è la creazione di tipi di dato astratti, l'ereditarietà ed il polimorfismo, ma ci sono altri argomenti altrettanto importanti. Questa sezione dà una panoramica di questi argomenti.

Importante in particolare è il modo in cui vengono creati e distrutti gli oggetti. Dov'è il dato per un oggetto e come è controllato il tempo di vita di un oggetto? Linguaggi di programmazione diversi usano diverse filosofie. Il C++ segue l'approccio secondo il quale il controllo dell'efficienza è il problema più importante, perciò dà la scelta al programmatore. Per ottenere la massima velocità di esecuzione, la memorizzazione e il tempo di vita può essere determinato mentre il programma viene scritto, piazzando gli oggetto nello stack o nella memoria statica. Lo stack è un'area di memoria che è usata direttamente dal microprocessore per memorizzare i data durante l'esecuzione del programma. Le variabili nello stack sono a volte chiamate variabili *automatiche* o *delimitate*. La memoria statica è semplicemente una parte fissa della memoria che è allocata prima che il programma vada in esecuzione. L'utilizzo dello stack o di aree di memoria statica piazza una priorità sulla velocità di allocazione della memoria ed il rilascio, che può essere di gran valore in alcune situazioni.Tuttavia, si sacrifica la flessibilità perchè si deve conoscere esattamente la quantità, il tempo di vita ed il tipo di oggetti *mentre* si sta scrivendo il programma. Se si prova a risolvere un problema più generale, come il disegno assistito al computer, warehouse management o controllo del traffico aereo, ciò è troppo restrittivo.

Il secondo approccio è la creazione dinamica di oggetti in un area di memoria detta heap. Con questo approccio non si conosce quanti oggetti si ha bisogno fino a runtime, qual è il loro ciclo di vita e qual è il loro tipo esatto. Queste decisione vengono prese quando c'è la necessità durante l'esecuzione del programma. Se si ha bisogno di un nuovo oggetto, lo si crea semplicemente nella heap usando la parola chiave **new**. Quando non serve più, si deve rilasciare la memoria con la parola chiave **delete**.

Poichè la memoria libera è gestita dinamicamente a runtime, il tempo totale richiesto per allocare memoria nella heap è significativamento più lungo del tempo di allocazione nello stack (spesso serve una singola istruzione del microprocessore per spostare il puntatore in basso ed un'altra per muoverlo in alto). L'approccio dinamico fa generalmente l'assunzione logica che gli oggetti tendono ad essere complicati, quindi è necessario un overhead extra per trovare spazio e rilasciare quello spazio non avrà un impatto importante sulla creazione di un oggetto. In aggiunta, la maggiore flessibilità è essenziale per risolvere i generali problemi di programmazione.

C'è un altro problema, tuttavia , ed è il tempo di vita di un oggetto. Se si crea un oggetto nello stack o nella memoria statica, il compilatore determina dopo quanto tempo l'oggetto muore e può automaticamente distruggerlo. Tuttavia, se lo si crea nella heap, il

compilatore non ha conoscenza del suo tempo di vita. Nel C++, il programmatore deve determinare quando distruggere l'oggetto e poi eseguire la distruzione con la parola chiave **delete**. In alternativa, l'ambiente può fornire una funzionalità detta *garbage collector* che automaticamente scopre quando un oggetto non è più usato e lo distrugge. Naturalmente, scrivere programmi usando un *garbage collector* è molto più conveniente, ma richiede che tutte le applicazione debbano tollerare l'esistenza del *garbage collector* ed il suo overhead. Ciò non incontrava i requisiti di design del C++ e quindi non fu incluso, sebbene sono stati sviluppati per il C++ garbage collector di terze parti.

Gestione delle eccezioni: trattare gli errori

Fin dalla nascita dei linguaggi di programmazione, la gestione degli errori è stato uno dei problemi più difficili. Poichè è molto difficile progettare un buon meccanismo per la gestione degli errori, molti linguaggi semplicemente ignorano il problema, passandolo ai progettisti delle librerie che realizzano soluzioni a metà strada funzionanti in molte situazioni, ma che possono essere facilmente aggirate, generalmente soltanto ignorandole. Un problema principale con la maggior parte dei sistemi di gestione degli errori è che essi si affidano sulla vigilanza del programmatore nel seguire una convenzione convenuta che non è imposta dal linguaggio. Se i programmatori non sono attenti, cosa che spesso accade quando vanno di fretta, questi meccanismi possono venir facilmente dimenticati.

La *gestione delle eccezioni* integra la gestione degli errori direttamente nel linguaggio di programmazione e a volte persino nel sistema operativo. Un'eccezione è un oggetto che è "lanciato" dal punto dell'errore e può essere "preso" da un appropriato gestore delle eccezione progettato per gestire quel particolare tipo di errore. È come se la gestione delle eccezione sia un percorso diverso e parallelo di esecuzione che può essere preso quando le cose vanno per il verso sbagliato. E poichè usa un percorso di esecuzione diverso, non ha bisogno di interferire con il codice normalmente in esecuzione. Ciò rende il codice più semplice da scrivere poichè non si deve costantemente controllare se ci sono errori. Inoltre, un' eccezione lanciata è diversa da un valore di errore restituito da una funzione o da un flag impostato da una funzione per indicare una condizione di errore: queste segnalazioni possono essere ignorate. Una eccezione non può essere ignorata, quindi è garantito
che in qualche punto verrà affrontata. Infine, le eccezioni forniscono un modo per riprendersi in modo affidabile da una cattiva situazione. Invece di uscire e basta, spesso si può rimettere le cose a posto e ripristinare l'esecuzione di un programma, ottenendo per questa via programmi più robusti.

Vale la pena notare che la gestione delle eccezioni non è una caratteristica object-oriented, sebbene nei linguaggi orientati agli oggetti l'eccezione è normalmente rappresentata con un oggetto. La gestione delle eccezioni esisteva prima dei linguaggi orientati agli oggetti.

La gestione delle eccezioni è introdotta soltanto parzialmente in questo Volume; il Volume 2 (disponibile su *www.BruceEckel.com*) tratta esaustivamente questo argomento.

Analisi e progettazione

Il paradigma orientato agli oggetti è un nuovo e diverso modo di pensare alla programmazione e molte persone hanno problemi a come approcciare un progetto OOP. Una volta che si sà che tutto è supposto essere un oggetto e che si è imparato a pensare più

in un modo orientato agli oggetti, si può cominciare a creare buoni progetti ed approfittare di tutti i benefici che la OOP ha da offrire.

Un metodo (spesso detto una metodologia) è un insieme di processi e ausilii per dividere la complessità di un problema di programmazione. Molti metodi OOP sono stati formulati dall'alba della programmazione orientata agli oggetti. Questa sezione darà un assaggio di cosa si sta tentando di ottenere quando si usa un metodo.

Specialmente nella OOP, la metodologia è un campo in cui si fanno molti esperimenti, quindi è importante capire quale problema il metodo sta cercando di risolvere prima di adottarne uno. Ciò è particolarmente vero nel C++, in cui il linguaggio di programmazione è inteso a ridurre la complessità (a paragone con il C) riguardante l'espressione di un programma. Ciò può alleviare il bisogno di metodologie ancora più complesse. Invece, metodologie più semplici possono bastare in C++ per una classe di problemi più vasta che si può gestire usando metodologie semplici con linguaggi procedurali.

È importante capire che il termine "metodogia" è spesso esagerato e promette molto. Qualsiasi cosa si faccia quando si progetta e si scrive un programma è un metodo. Può essere il proprio metodo e si può non essere consci di farlo, ma è un processo che si segue mentre si crea. Se un processo è efficace, può aver bisogno solo di una piccola messa a punto per funzionare con il C++. Se non si è soddisfatti della propria produttività e del modo in cui riescono i propri programmi, si dovrebbe considerare di utilizzare un metodo formale o di sceglierne una parte dai molti disponibili.

Mentre si attraversa il processo di sviluppo, la cosa più importante è questa: non perdersi. È facile farlo. La maggior parte dei metodi di analisi e di design sono pensati per risolvere la maggior parte dei problemi. Si ricordi che la maggior parte dei progetti non appartiene a questa categoria, quindi di solito si può aver un' analisi e design vincente con un sottoinsieme relativamente piccolo di ciò che un metodo raccomanda[9]. Tuttavia un processo di un qualche tipo, non importa quanto limitato, in generale indicherà il giusto cammino molto meglio di quanto non si farebbe cominciando subito a codificare.

È anche facile restare impantanati, cadere nella “paralisi da analisi”, nella quale si ha la sensazione di non poter proseguire perché non si è ancora sistemato ogni più piccolo particolare dello stadio corrente. Si ricordi, per quanto approfondita possa essere la propria analisi, vi sono sempre cose di un sistema che non si lasciano rivelare fino al momento della progettazione e altre ancora, in maggior quantità, che non si manifestano fino a quando non passerà alla codificare o addirittura fino a quando il programma non è finito ed è in esecuzione. Per queste ragioni, è essenziale percorrere velocemente le fasi di analisi e progettazione ed implementare un collaudo del sistema in sviluppo.

C’è un punto che merita di essere sottolineato. Per via della storia che abbiamo vissuto con i linguaggi procedurali, è encomiabile una squadra che intenda procedere con cautela e capire ogni minimo particolare prima di passare alla progettazione e all’implementazione. Certo, quando si crea un DBMS c’è tutto da guadagnare a capire in modo esauriente i fabbisogni di un cliente. Ma un DBMS è una classe di problemi che sono ben formulati e ben capiti; in molti programmi di questo genere la struttura del database è il problema da affrontare. La classe di problemi di programmazione di cui ci si occupa in questo capitolo appartiene alla famiglia dei “jolly” (termine mio), nella quale la soluzione non può essere trovata semplicemente ricreando una soluzione ben conosciuta, ma coinvolge invece uno o più “fattori jolly”: elementi per i quali non esiste una soluzione precedente ben capita e per i quali è necessario effettuare delle ricerche[10]. Tentare di analizzare in modo

esauriente un problema jolly prima di passare alla progettazione e all'implementazione porta alla paralisi da analisi, perché non si hanno sufficienti informazioni per risolvere questo genere di problemi durante la fase di analisi. Per risolvere questo tipo di problemi occorre ripercorrere più volte l'intero ciclo e questo esige un comportamento incline all'assunzione di rischi (cosa che ha un suo senso, perché si cerca di fare qualcosa di nuovo e il compenso potenziale è più elevato). Potrebbe sembrare che il rischio aumenti "affrettando" una implementazione preliminare, ma ciò potrebbe invece ridurre il rischio in un progetto jolly, perché si scopre con anticipo se un determinato approccio al problema è plausibile. Sviluppare un prodotto significa gestire il rischio.

Spesso si propone di "costruirne uno da buttare". Con la OOP, ci si troverà certo a buttarne via una parte, ma siccome il codice è incapsulato nelle classi, durante la prima passata si produrrà inevitabilmente qualche schema di classe utile e si svilupperanno valide idee in merito al progetto del sistema che non saranno da buttare. Di conseguenza, una prima rapida passata sul problema non soltanto produce informazioni importanti per la successiva passata di analisi, progettazione e implementazione, ma crea anche una base di codice.
Detto questo, se si va in cerca di una metodologia che contenga enormi volumi di dettagli e imponga molti passi e documenti, è difficile stabilire dove fermarsi. Si tenga ben chiaro in mente quel che si sta cercando di scoprire:

1. Quali sono gli oggetti? (Come si scompone il proprio progetto in parti?)
2. Quali sono le loro interfacce? (Quali messaggi si devono mandare a ciascun oggetto?)

Se si viene fuori con nient'altro che gli oggetti e le loro interfacce, si può cominciare a scrivere un programma. Per varie ragioni si potrebbe aver bisogno di una maggior quantità di informazioni e di documenti, ma si può fare meno di questo. Il processo può essere articolato in cinque fasi, più una Fase 0 che è semplicemente l'impegno iniziale ad utilizzare un qualche tipo di struttura.

Fase 0: Fare un piano

Come prima cosa si deve decidere in quali passi si articolerà il proprio processo. Sembra semplice (e in effetti tutto questo sembra semplice) eppure la gente spesso non prende questa decisione prima di mettersi a codificare. Se il proprio piano è "diamoci sotto e cominciamo a scrivere il codice", bene (a volte è quello giusto, quando il problema è ben chiaro). Almeno si accetti l'idea che il piano è questo.

Si potrebbe anche decidere in questa fase che il processo va strutturato ancora un po', ma senza impegnarsi eccessivamente. Si può capire che a certi programmatori piaccia lavorare con "spirito vacanziero", senza una struttura che ingabbi il processo di sviluppo del loro lavoro:"Sarà fatto quando sarà fatto". La cosa può anche essere divertente, per un po', ma mi sono accorto che avere qualche obiettivo intermedio, le pietre miliari (o milestone come vengono chiamate nel gergo dei pianificatori), aiuta a focalizzare e a stimolare gli impegni riferendoli a quelle pietre miliari invece di ritrovarsi con l'unico obiettivo di "finire il progetto". Inoltre, così si suddivide il progetto in segmenti più agevoli da afferrare,

facendolo diventare meno minaccioso (e poi le tappe intermedie sono ottime occasioni per festeggiamenti).

Quando cominciai a studiare la struttura narrativa (così una volta o l'altra scriverò un romanzo) provavo una certa riluttanza nei confronti del concetto di struttura, perché mi sembrava di scrivere meglio quando buttavo giù le pagine direttamente. In seguito, però, mi resi conto che, quando scrivo di computer, la struttura mi è chiara al punto che non ci devo pensare più di tanto. Ma strutturo comunque mentalmente il mio lavoro, seppure in modo non del tutto consapevole. Anche se si è convinti che il proprio piano sia di mettersi subito a codificare, si finirà comunque col percorrere le fasi che seguono mentre ci si farà certe domande e ci i daranno le risposte.

Dichiarare la missione

Qualunque sistema si andrà a costruire, per quanto complicato, ha uno scopo fondamentale; il contesto nel quale si trova, il fabbisogno base che deve soddisfare. Se si riesce a guardare al di là dell'interfaccia utente, dei particolari specifici dell'hardware o del software, degli algoritmi di codifica e dei problemi di efficienza, si finirà per scoprire il nucleo essenziale del sistema: semplice e diretto. Come la cosiddetta idea base di un film di Hollywood, potrete descriverlo con una o due frasi. Questa descrizione pura è il punto di partenza.

L'idea base è molto importante perché dà il tono a tutto al proprio progetto; è la dichiarazione della missione. Non si riuscirà a coglierla con esattezza fin dalla prima volta (potrebbe essere trovata in una fase successiva del progetto prima che diventi del tutto chiara), però bisogno insistere finché non sembra quella giusta. Per esempio, in un sistema per il controllo del traffico aereo si potrebbe cominciare con un'idea base focalizzata sul sistema che si sta costruendo: *Il programma della torre tiene traccia degli aerei*. Si pensi, però, a quel che accade se si riduce il sistema alla dimensione di un aeroporto molto piccolo; magari c'è un solo controllore del traffico o addirittura nessuno. Un modello più utile, invece di riferirsi alla soluzione che si sta creando, descrive il problema: *arrivano aeroplani, scaricano, si riforniscono, ricaricano e ripartono.*

Qualsiasi sistema si costruisca, non importa quanto complicato, ha uno scopo fondamentale,il contesto nel quale si trova, le basi che esso soddisfa. Se si guarda oltre l'interfaccia utente, il dettaglio dell'hardware o del sistema specifico, gli algoritmi di codica e i problemi di efficienza, alla fine si trovera il nocciolo della sua essenza: semplice e lineare. Come la cosidetta idea base di un film di Hollywood, lo si può descrivere in una o due frasi. Questa pura descrizione è il punto di partenza.

Fase 1: Cosa stiamo facendo?

Nella *progettazione procedurale*, come veniva chiamato il metodo di progettazione dei programmi della generazione precedente, questa fase era dedicata a creare l'*analisi dei requisiti* e la *specifica del sistema*. Si trattava di attività nelle quali era facile smarrirsi; certi documenti, i cui soli nomi già mettevano soggezione, finivano per diventare a loro volta grossi progetti. Le intenzioni erano buone, però. L'analisi dei requisiti dice: "*Si faccia un elenco delle linee guida che si utilizzeranno per sapere quando il lavoro è concluso ed il cliente è soddisfatto*". La specifica del sistema dice: "*Questa è la descrizione di ciò che il*

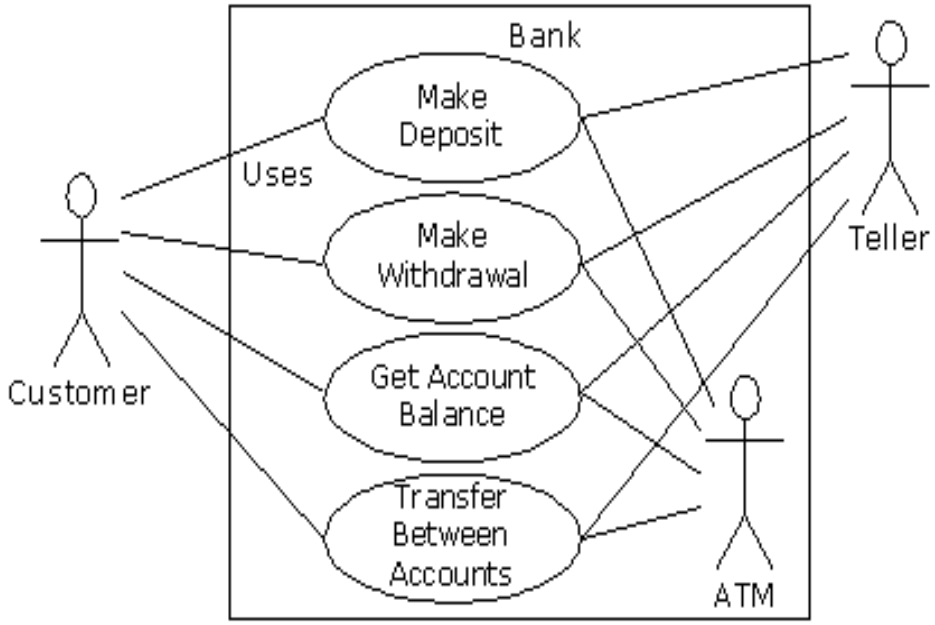
programma farà (non come) per soddisfare i requisiti". L'analisi dei requisiti è in realtà un contratto fra il programmatore ed il cliente (che può anche essere qualcuno che lavora nella vostra stessa azienda oppure qualche altro oggetto o sistema). La specifica del sistema è una panoramica generale del problema ed in un certo senso la verifica che si possa risolvere e quanto ci vorrà per risolverlo. Siccome richiedono entrambi un accordo fra persone (e siccome queste di solito cambiano col passare del tempo), sarebbe meglio produrre documenti brevi ed essenziali, idealmente puri elenchi e diagrammi schematici, per risparmiare tempo. Potrebbero esservi altri vincoli che costringono ad ampliarli in documenti più voluminosi, però se ci si impegna a dare al documento iniziale un contenuto breve e conciso, è possibile crearlo in poche sedute di discussione in gruppo, con un leader che crea dinamicamente la descrizione. In questo modo non soltanto si stimola il contributo di tutti, ma si favorisce l'adesione iniziale e l'accordo di tutti i componenti della squadra. Quel che è più importante, forse, è il fatto che così si può avviare un progetto con molto entusiasmo.

È necessario mantenersi concentrati sul cuore di ciò che si sta tentando di realizzare in questa fase: determinare cosa si suppone che il sistema faccia. Lo strumento più prezioso per ciò è una raccolta do ciò che sono detti "*Use case*" (casi d'utilizzo) . Gli Use case identificano le caratteristiche chiave del sistema che riveleranno alcune delle classi fondamentali che si useranno. Ci sono essenzialmente risposte descrittive alle domande tipo[11]:

- "Chi userà il questo sistema?"
- "Cosa possono fare gli attori con il sistema?"
- "Come fa ciò questo attore con il sistema?"
- "Come altrimenti potrebbe funzionare se qualcun altro fa questo o se lo stesso attore ha un obiettivo diverso?" (per rivelare variazioni)
- "Quali problemi possono accadere quando si fa ciò con il sistema?" (per far venir fuori le aspettative)

Se, per esempio, si sta progettando uno sportello bancario automatico, il caso di utilizzo di un aspetto particolare della funzionalità è in grado di descrivere quello che lo sportello automatico fa in ogni possibile situazione. Ciascuna di queste situazioni prende il nome di *scenario* e un caso di utilizzo può essere considerato una raccolta di scenari. Si può pensare a uno scenario come una domanda che comincia con: *"Che cosa fa il sistema se. . .?"*. Per esempio: *"Che cosa fa lo sportello automatico se un cliente ha versato un assegno nel corso delle ultime 24 ore e non c'è sufficiente disponibilità nel conto per soddisfare una richiesta di prelievo se l'assegno non è stato ancora accettato?"*

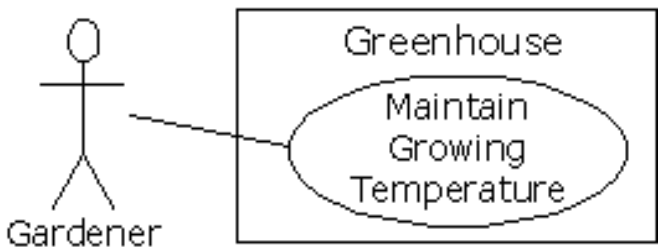
I diagrammi dei casi di utilizzo sono deliberatamente semplici, per evitarvi di restare impantanati prematuramente nei particolari dell'implementazione del sistema.



Ciascun simbolo di persona rappresenta un *"attore"*, che di solito è un essere umano o qualche altro tipo di agente libero (questi possono essere anche altri sistemi computerizzati, come nel caso del *Terminale*). Il riquadro rappresenta i confini del proprio sistema. Gli ovali rappresentano i casi di utilizzo, che sono descrizioni di lavori utili che si possono eseguire con il sistema. Le linee che collegano attori e casi di utilizzo rappresentano le interazioni.

Non importa come il sistema sia effettivamente implementato, basta che si presenti in questo modo all'utente.

Un caso di utilizzo non deve essere terribilmente complesso, anche se il sistema sottostante è complesso. Serve soltanto per far vedere il sistema nel modo in cui appare all'utente. Per esempio:



I casi di utilizzo producono le specifiche dei requisiti stabilendo le interazioni che l'utente può avere col sistema. Si tenti di scoprire un insieme completo di casi di utilizzo per il proprio sistema e, quando si riuscirà, si avrà il nucleo di quello che il sistema dovrebbe fare. Il bello dei casi di utilizzo sta nel fatto che riconducono sempre all'essenziale ed impediscono di sconfinare su questioni che non sono essenziali ai fini del risultato del lavoro. In altri termini, se si ha un insieme completo di casi di utilizzo, si è in grado di descrivere il proprio sistema e di passare alla fase successiva. Probabilmente non si avrà il quadro completo fin dal primo tentativo, ma va bene così. Tutto salterà fuori, col passare

del tempo e se si pretende di avere in questa fase una specifica perfetta del sistema, si finirà per impantanarsi.

Se ci si impantana per davvero, si può rilanciare questa fase ricorrendo a un grossolano strumento di approssimazione: si descriva il sistema con poche frasi e si cerchino nomi e verbi. I nomi possono suggerire attori, contesto del caso di utilizzo (per esempio *"magazzino"*) oppure manufatti manipolati nel caso di utilizzo. I verbi possono suggerire interazioni fra attori e casi di utilizzo e specificare passi entro il caso di utilizzo. Si scopriranno anche che nomi e verbi producono oggetti e messaggi durante la fase di progettazione (e si tenga presente che i casi di utilizzo descrivono interazioni fra sottosistemi, quindi la tecnica "nomi e verbi" può essere utilizzata soltanto come strumento per una seduta di brainstorming, perché non produce casi di utilizzo) [12].

La linea di demarcazione fra un caso di utilizzo ed un attore può segnalare l'esistenza di un'interfaccia utente, ma non la definisce. Per approfondire il processo col quale si definiscono e si creano interfacce utente, si consulti *Software for Use* di Larry Constantine e Lucy Lockwood (Addison-Wesley Longman, 1999) oppure si visti il sito *www.ForUse.com*.

Per quanto sia un pò un'arte magica, a questo punto è importante tracciare un piano di massima. Si ha a disposizione una panoramica di quello che si intende costruire, quindi si dovrebbe riuscire a farsi un'idea del tempo che ci vorrà. Qui entrano in gioco parecchi fattori.
Se dalla propria stima emerge un piano molto lungo, la società potrebbe decidere di non realizzarlo (e quindi utilizzare le risorse su qualcosa di più ragionevole: e questa è un'ottima cosa). Oppure un dirigente potrebbe aver già deciso per conto suo quanto tempo ci vorrà per realizzare il progetto e cercherà di influenzare in questo senso le stime. La cosa migliore, però, è definire onestamente un piano fin dall'inizio e affrontare quanto prima le decisioni più ardue. Sono stati fatti un sacco di tentativi per individuare tecniche di pianificazione accurate (non molto diverse da quelle impiegate per fare previsioni sul mercato azionario), però la cosa migliore è fidarsi della propria esperienza e del proprio intuito. Si faccia una stima a spanne di quanto ci si metterà per davvero, poi la si raddoppi e si sommi un dieci per cento. La propria stima a spanne sarà probabilmente corretta; si riuscirà ad ottenere qualcosa che funziona entro quel periodo di tempo. Raddoppiandola,
si trasformerà la stima in qualcosa di plausibile ed il dieci per cento servirà per i ritocchi finali ed i dettagli[13] . Comunque si voglia spiegarlo e indipendentemente dalle lamentele e dalle manovre che salteranno fuori quando si rivela un piano del genere, sembra che le cose vadano comunque in questo modo.

Fase 2: Come lo costruiremo?

In questa fase bisogna produrre un progetto che descriva l'aspetto delle classi e il modo in cui interagiscono. Una tecnica eccellente per determinare classi e interazioni è la scheda Classe-Responsabilità-Collaborazione (CRC). La validità di questo strumento deriva in parte dal fatto che si basa su un tecnologia molto povera: si comincia con un pacchetto di schede di cartoncino bianche in formato 7 × 12 e vi si scrive sopra. Ciascuna scheda rappresenta una sola classe e sulla scheda si scrive:

1. Il nome della classe. È importante che questo nome colga l'essenza di quello che la classe fa, in modo che ne esprima il senso a prima vista.
2. Le "responsabilità" della classe: che cosa dovrebbe fare. Di solito questo si può riepilogare enunciando semplicemente i nomi delle funzioni membro (perché questi nomi, in un progetto ben fatto, devono essere descrittivi), ma non sono escluse altre annotazioni.
   Se si ha bisogno di innescare il processo, si condideri il problema dal punto di vista di un programmatore pigro: quali oggetti ci piacerebbe che comparissero magicamente per risolvere il proprio problema?
3. Le "collaborazioni" della classe: con quali altre classi interagisce? "Interagire"è un termine deliberatamente molto ampio; potrebbe significare aggregazione o semplicemente che esiste qualche altro oggetto che eseguirà servizi per un oggetto della classe.
   Le collaborazioni devono anche considerare l'uditorio di questa classe. Per esempio, si crea una classe FuocoArtificio, chi la osserverà, un Chimico o uno Spettatore? Il primo vorrà sapere quali sostanze chimiche sono impiegate nella costruzione, mentre
   il secondo reagirà ai colori e alle forme che si producono quando il fuoco d'artificio esplode.

Si potrebbe pensare che le schede dovrebbero essere più grandi per contenere tutte le informazioni che piacerebbe scrivervi sopra, ma sono deliberatamente piccole, non soltanto per far sì che le classi siano piccole, ma anche per impedire di impegnarsi troppo presto
in troppi particolari. Se non si riesce a far stare in una piccola scheda tutto quello che si deve sapere su una classe, la classe è troppo complessa (o si sta scendendo troppo nei particolari, o si dovrebbe creare più di una sola classe). La classe ideale deve essere capita a prima vista. Le schede CRC sono concepite per aiutare a produrre una prima stesura del progetto, in modo da avere il quadro di massima per poi raffinarlo.

Le schede CRC si dimostrano particolarmente vantaggiose ai fini della comunicazione. È qualcosa che è meglio fare in tempo reale, in un gruppo, senza computer. Ogni persona si prende la responsabilità di svariate classi (che dapprima non hanno nomi né altre informazioni). Si esegue una simulazione dal vivo risolvendo un dato scenario per volta, stabilendo quali messaggi vengono inviati ai vari oggetti per soddisfare ciascun scenario. Mentre si percorre questo processo si scoprono le classi di cui si ha bisogno insieme con le loro responsabilità e collaborazioni, si riempiono le schede a mano a mano che si procede. Quando sono stati percorsi tutti i casi di utilizzo, dovrebbe essere pronto un primo schema generale del progetto, ragionevolmente completo.

Prima di cominciare a usare le schede CRC, la mia esperienza di consulente di maggior successo è stata quando mi sono trovato a dover lavorare sulla fase iniziale di progettazione disegnando oggetti su una lavagna con una squadra che non aveva mai costruito prima un
progetto OOP. Si parlava di come gli oggetti avrebbero dovuto comunicare fra loro, ne cancellavamo alcuni e li sostituivamo con altri oggetti. In pratica, gestivo tutte le "schede CRC" sulla lavagna. Era la squadra (che sapeva che cosa il progetto avrebbe dovuto fare) che creava materialmente il disegno generale; erano loro i "proprietari" del disegno, non era qualcosa che gli veniva dato da altri. Io non facevo altro che orientare il processo ponendo le domande giuste, mettendo alla prova le ipotesi e ricevendo le reazioni della

squadra
per modificare quelle ipotesi. Il bello del processo era nel fatto che la squadra imparava a fare progettazione orientata agli oggetti non studiando esempi astratti, ma lavorando sull'unico disegno che per loro era il più interessante in quel momento: il loro.

Quando si sarà messo insieme un pacchetto di schede CRC, potrebbe essere utile creare una descrizione più formale del progetto utilizzando l'UML [14] . Non è obbligatorio servirsene, però può essere d'aiuto, specialmente se si vuole appendere un diagramma sul muro, in modo che tutti possano pensarci sopra, che è un'ottima idea. Un'alternativa all'UML è una descrizione testuale degli oggetti e delle loro interfacce, oppure, a seconda del linguaggio di programmazione utilizzato, il codice stesso [15].

Con l'UML si dispone anche di una ulteriore notazione grafica per descrivere il modello dinamico di un sistema. Questo aiuta in situazioni nelle quali le transizioni di stato di un sistema o di un sottosistema sono dominanti al punto da aver bisogno di propri diagrammi (come nel caso di un sistema di controllo). Può anche essere necessario descrivere la struttura dei dati, per sistemi o sottosistemi nei quali i dati sono un fattore dominante (come nel caso di un database).

Si capirà di aver concluso la Fase 2 quando saranno descritti gli oggetti e le loro interfacce. Beh, non proprio tutti, la maggior parte: ce ne sono sempre alcuni che sfuggono e non si manifestano fino alla Fase 3. Ma non è un problema. L'unica cosa che interessa davvero è arrivare a scoprire tutti i propri oggetti, alla fine. Fa certo piacere scoprirli all'inizio del processo, ma la struttura dell'OOP fa sì che non sia grave scoprirli in tempi successivi. In effetti, la progettazione di un oggetto tende ad articolarsi in cinque stadi, nell'arco dello sviluppo del programma.

Cinque stadi di progettazione degli oggetti

Il ciclo di vita del design di un oggetto non è limitato al tempo in cui si scrive il programma. Invece, il progetto di un oggetto appare in una sequenza di fasi. È d'aiuto avere questa prospettiva perchè non ci illude di essere perfetti da subito; invece, si capisce che la comprensione di ciò che fa un oggetto e come deve apparire accade sempre. Questo punto di vista si applica anche al design di vari tipi di programma; il pattern di un particolare tipo affiora quando si lotta con quel problema (i Design Patterns sono trattati nel Volume 2). Gli oggetti, anche, hanno i loro pattern che emergono attraverso la comprensione, l'uso ed il riuso.

1. **Scoperta degli oggetti.** Questo stadio si manifesta durante l'analisi iniziale di un programma. Gli oggetti si possono scoprire cercando fattori esterni e linee di confine, duplicazione di elementi nel sistema e le più piccole unità concettuali. Alcuni oggetti
   sono ovvi se si ha già un insieme di librerie di classi. La comunanza fra classi, che fa pensare a classi base e all'ereditarietà, può comparire fin dal primo momento oppure più avanti nel corso della progettazione.

2. **Assemblaggio degli oggetti.** Nel costruire un oggetto si scoprirà la necessità di nuovi membri che non erano manifesti al momento della scoperta. Le necessità interne dell'oggetto possono richiedere altre classi per supportarlo.
3. **Costruzione del sistema.** Ancora una volta, ulteriori requisiti per un oggetto possono venir fuori in uno stadio successivo. Mentre si impara, si fanno evolvere i propri oggetti. Il fabbisogno di comunicazione e di interconnessione con altri oggetti nel sistema può modificare le necessità delle proprie classi o richiedere nuove classi. Per esempio, si può scoprire che occorrono classi facilitatrici o di guida, tipo liste concatenate, che contengono poche (o nessuna) informazioni di stato, ma che aiutano altre classi a funzionare.
4. **Estensione del sistema.** Nell'aggiungere nuove funzionalità a un sistema si può scoprire che il progetto precedente non supporta un'agevole estensione del sistema. Con queste nuove informazioni si può ristrutturare parti del sistema, eventualmente aggiungendo nuove classi o nuove gerarchie di classi.
5. **Riutilizzo degli oggetti.** Questo è il vero collaudo strutturale di una classe. Se qualcuno prova a riutilizzarla in una situazione interamente nuova, probabilmente ne scoprirà alcune limitazioni. Mentre si modifica una classe per adattarla a un maggior numero di nuovi programmi, i principi generali di quella classe diventeranno più chiari, fintanto che si arriva a un tipo davvero riutilizzabile. Non ci aspetti, però, che la maggior parte degli oggetti del progetto di un sistema sia riutilizzabile: è perfettamente accettabile che il grosso dei propri oggetti sia specifico del sistema. I tipi riutilizzabili tendono a essere meno comuni e devono risolvere problemi di carattere più generale per poter essere riutilizzabili.

Linee guida per lo sviluppo di oggetti

Queste fasi suggeriscono alcune linee guida quando si pensa allo sviluppo delle classi:

1. Si lasci che un problema specifico generi una classe, poi si lasci crescere e maturare la classe durante la soluzione di altri problemi.
2. Si ricordi che scoprire la classe di cui si ha bisogno (e le loro interfacce) è la maggior parte del progetto del sistema. Se si hanno già quelle classi, dovrebbe essere un progetto facile.
3. Non forzarsi di conoscere tutto all'inizio, si impari mentre si va avanti. Ciò accadrà comunque.
4. Si cominci a programmare, facendo funzionare qualcosa in modo che si può testare il progetto. Non si abbia paura di finire con codice a spaghetti di stile procedurale, le classi partizionano il problema ed aiutano a controllare anarchia ed entropia. Cattive classi non rompono le classi buone.
5. Preferire sempre la semplicità. Oggetti piccoli e semplici con ovvia utilità sono migliori di interfacce grosse e complicate. Quando bisogna prendere delle decisione, si usi l'approccio del rasoio di Occam: si consideri le scelte e si scelga la più semplice, poichè classi semplici sono quasi sempre le migliori. Iniziando con classi piccole e semplici si può espandere l'interfaccia delle classi quando la si comprende meglio, ma quanto più il tempo passa, è difficile eliminare elementi da una classe.

Fase 3: costruire il nucleo

Questa è la conversione iniziale dalla prima bozza in un insieme di codice che si compila e

si esegue e può essere collaudato, ma soprattutto che dimostra la validità o la non validità della propria architettura. Non si tratta di un processo da svolgere in una sola passata, ma è piuttosto l'inizio di una serie di passi che costruiranno iterativamente il sistema, come si vedrà nella Fase 4.

L'obiettivo è trovare il nucleo dell'architettura da implementare per generare un sistema funzionante, non importa quanto quel sistema sia incompleto in questa passata iniziale. Si sta creando uno schema di riferimento sul quale continuare a costruire con ulteriori iterazioni. Si sta anche eseguendo la prima di molte operazioni di integrazione di sistema e di collaudo, facendo sapere a tutti gli interessati come si presenterà il loro sistema e come sta progredendo. Idealmente, ci si sta anche esponendo a qualche rischio critico.
Si scoprirà anche, molto probabilmente, modifiche e miglioramenti che si potrebbero apportare all'architettura originale: cose che non si sarebbero potuto capire senza implementare il sistema.

Fa parte del processo di costruzione di sistema il confronto con la realtà che si ottengono collaudandolo a fronte dell'analisi dei requisiti e delle specifiche del sistema (in qualunque forma queste esistano). Ci si assicuri che i propri test verifichino i requisiti e i casi di utilizzo.
Quando il nucleo del sistema è stabile, si è pronti a procedere oltre e ad aggiungere nuove funzionalità.

Fase 4: iterare i casi di utilizzo

Una volta che il nucleo base gira, ciascuna funzionalità che si aggiunge è di per sé un piccolo progetto. Un insieme di funzionalità si aggiunge durante un'iterazione, un periodo di sviluppo relativamente breve.

Quanto dura un'iterazione? Idealmente, da una a tre settimane (può variare in funzione del linguaggio di implementazione). Alla fine di quel periodo, si avrà un sistema integrato e collaudato, con più funzionalità di quelle che si avevano prima. Quello, però, che è particolarmente interessante è la base dell'iterazione: un solo caso di utilizzo. Ciascun caso di utilizzo è un pacchetto di funzionalità correlate che vengono integrate nel sistema in un colpo solo, durante una sola iterazione. Non soltanto questo modo di operare vi dà un'idea migliore di quel che dovrebbe essere la portata di un caso di utilizzo, ma conferisce maggior validità all'idea di ricorrere a un caso di utilizzo, dal momento che il concetto non viene scartato dopo l'analisi e la progettazione, ma è invece un'unità di sviluppo fondamentale nel corso dell'intero processo di costruzione del software.

Le iterazioni finiscono quando si arriva alla funzionalità prefissata oppure matura una scadenza esterna ed il cliente può venir soddisfatto dalla versione corrente. (Ricordate, quello del software è un mercato per abbonamenti). Siccome il processo è iterativo, si avranno molte opportunità per consegnare un prodotto invece che un solo punto

terminale; i progetti di tipo open source funzionano esclusivamente in un ambiente iterativo, ad elevato feedback, ed è esattamente per questo che hanno tanto successo.

Sono molte le ragioni che giustificano un processo iterativo. Si può rilevare e risolvere con molto anticipo situazioni rischiose, i clienti hanno ampie possibilità di cambiare idea, la soddisfazione dei programmatori è più elevata ed il progetto può essere guidato con maggior precisione. Un ulteriore, importante vantaggio viene dal feedback verso gli interessati,che possono rilevare dallo stato attuale del prodotto a che punto si trovano tutti gli elementi. Ciò può ridurre o addirittura eliminare la necessità di condurre estenuanti riunioni di avanzamento, aumentando la fiducia ed il sostegno da parte degli interessati.

Fase 5: Evoluzione

Questo è quel punto nel ciclo dello sviluppo che si chiamava tradizionalmente "manutenzione", un termine ombrello che può significare qualunque cosa, da "farlo funzionare nel modo in cui doveva davvero funzionare fin dall'inizio" a "aggiungere funzionalità che il cliente si era dimenticato di indicare"ai più tradizionali "correggere gli errori che sono saltati fuori" e "aggiungere nuove funzionalità quando ce n'è bisogno". Al termine "manutenzione"sono stati attribuiti un tal numero di significati distorti che ha finito coll'assumere una qualità leggermente ingannevole, in parte perché suggerisce che si è in realtà costruito un programma immacolato, che basta lubrificare e tenere al riparo dalla ruggine. Forse esiste un termine migliore per descrivere come stanno le cose.

Verrà utilizzato il termine *evoluzione*[16]. Vale a dire: "Non vi verrà giusto la prima volta, quindi ci si conceda il respiro sufficiente per imparare, tornare sui propri passi e modificare". Si potrebbe aver bisogno di fare un mucchio di modifiche a mano a mano che si impara e si comprende più a fondo il problema. L'eleganza che si raggiunge evolvendo fino ad ottenere il risultato giusto ripagherà sia nel breve sia nel lungo periodo. Evoluzione è quando il proprio programma passa da buono a eccellente e quando diventano chiari certi aspetti che non erano stati capiti bene nella prima passata. E si manifesta anche quando le proprie classi da oggetti utilizzati per un solo progetto evolvono in risorse riutilizzabili.

Quando si dice "farlo giusto" non si intende soltanto che il programma funziona secondo i requisiti ed i casi di utilizzo. Significa anche che la struttura interna del codice ha per ognuno che la scritto un senso e dà la sensazione di essere ben integrata, senza contorcimenti sintattici, oggetti sovradimensionati o frammenti di codice esposti goffamente. Inoltre, si deve anche avere la sensazione che la struttura del programma sopravviverà alle modifiche alle quali sarà inevitabilmente soggetto durante la sua vita e che tali modifiche potranno essere effettuate agevolmente e in modo pulito. Non è cosa da poco. Non si deve soltanto capire quello che si sta costruendo, ma anche come il programma evolverà (quello che io chiamo il *vettore del cambiamento*[17]). Fortunatamente i linguaggi di programmazione orientati agli oggetti sono particolarmente adatti a supportare questo tipo di continue modifiche: i confini creati dagli oggetti sono quel che impedisce alla struttura di collassare. Questi linguaggi vi consentono anche di effettuare modifiche "che sembrerebbero drastiche in un programma procedurale" senza provocare terremoti in tutto il resto del codice. In effetti, il supporto dell'evoluzione potrebbe essere il vantaggio più importante dell'OOP.

Tramite l'evoluzione si arriva a creare qualcosa che almeno si avvicina a quello che si pensa di star costruendo, si può toccare con mano quel che si è ottenuto, paragonarlo con quel che si voleva ottenere e vedere se manca qualcosa. A questo punto si può tornare indietro e rimediare, riprogettando e implementando di nuovo le parti del programma che non funzionavano correttamente [18] . Si potrebbe davvero aver bisogno di risolvere il problema, o un suo aspetto, più di una volta prima di azzeccare la soluzione giusta. (Di solito in questi casi è utile studiare i *Design Patterns*. Si possono trovare informazioni in *Thinking in Patterns with Java*, scaricabile da *www.BruceEckel.com*)

Si ha evoluzione anche quando si costruisce un sistema, si vede che corrisponde con i propri requisiti, e poi si scopre che in realtà non era quello che si voleva. Quando si osserva il sistema in funzione, si scopre che in realtà si voleva risolvere un problema diverso. Se si ritiene che questo tipo di evoluzione possa manifestarsi, si ha il dovere verso di se stessi di costruire la vostra prima versione il più rapidamente possibile, in modo da poter scoprire se è proprio quella che si voleva.

Forse la cosa più importante da ricordare è che, per default , in realtà se si modifica una classe le sue super e sottoclassi continueranno a funzionare. Non si deve aver paura delle modifiche (specialmente se si possiede un insieme integrato di test unitari per verificare la correttezza delle proprie modifiche). Le modifiche non devastano necessariamente il programma e qualunque cambio nel risultato sarà circoscritto alle sottoclassi e/o agli specifici collaboratori della classe che sarà stata modificata.

Pianificare ricompensa

Naturalmente non ci si metterà mai a costruire una casa senza un bel pò di piani di costruzione accuratamente disegnati. Se si deve costruire una tettoia o un canile non serviranno disegni molto elaborati, ma anche in questi casi probabilmente si inizierà con qualche schizzo che servirà per orientarsi. Lo sviluppo del software è passato da un estremo all'altro. Per molto tempo, la gente non si curava affatto della struttura quando faceva sviluppo, ma poi i progetti di maggiori dimensioni hanno cominciato a fallire. Per reazione, ci siamo ritrovati a ricorrere a metodologie che avevano una quantità terrificante di struttura e di dettagli, concepite soprattutto per quei progetti di grandi dimensioni. Erano metodologie troppo spaventose da utilizzare: sembrava che uno dovesse passare tutto il suo tempo a scrivere documentazione, senza mai dedicare tempo alla programmazione (e spesso era proprio quello che accadeva). Spero che quello che vi ho presentato fin qui suggerisca una via di mezzo: una scala mobile. Si scelga l'approccio che meglio si adatta alle proprie necessità (e alla propria personalità). Anche se si deciderà di ridurlo a dimensioni minime, qualche tipo di piano rappresenterà un notevole miglioramento per il proprio progetto rispetto alla mancanza assoluta di un piano. Si ricordi che, secondo le stime più diffuse, più del 50 per cento dei progetti fallisce (alcune stime arrivano fino al 70 per cento!).

Seguendo un piano, meglio se è uno semplice e breve, arrivando a una struttura del progetto prima di iniziare la codifica, si scoprirà che le cose si mettono insieme molto più agevolmente di quando vi tuffate nella mischia e cominciate a menare fendenti. Otterrete anche parecchia soddisfazione. In base alla mia esperienza, arrivare ad una soluzione elegante è qualcosa che soddisfa profondamente un livello interamente diverso; ci si sente

più vicini all'arte che alla tecnologia. E l'eleganza rende sempre; non è un obiettivo frivolo da perseguire. Non soltanto dà un programma più facile da costruire e da correggere, ma sarà anche più facile da capire e da gestire ed è qui che si annida il suo valore economico.

Extreme programming

Ho studiato tecniche di analisi e progettazione, a più riprese, fin dai tempi delle superiori. Il concetto di *Extreme Programming* (XP) è la più radicale e piacevole che abbia mai visto. La si può trovare in *Extreme Programming Explained* di Kent Beck (Addison-Wesley 2000) e sul web su *www.xprogramming.com*.

XP è sia una filosofia sul lavoro della programmazione e un insieme di linea guida per farlo. Alcune di queste linee guida sono riflesse in altre recenti metodologie, ma i due contributi più importanti e notevoli, secondo me, sono "scrivere prima il test" e "programmazione in coppia". Sebbene parli energicamente dell'intero processo, Beck mette in evidenza che se si adottano solo queste due prassi si migliorerà enormemente la propria produttività e affidabilità.

Scrivere prima il test

Il test del software è stato relegato tradizionalmente alla fine di un progetto, dopo che "tutto funziona, ma proprio per essere sicuri". Implicitamente ha una bassa priorità e alle persone che si specializzano in materia non viene riconosciuta grande importanza e sono state spesso collocate negli scantinati, lontano dai "veri programmatori". I team di test hanno reagito vestendo abiti neri e sghignazzando ogni volta che rompono qualcosa (per essere onesti, ho avuto questa sensazione quando ho messo in crisi i compilatori C++).

XP rivoluziona completamente il concetto di test dandogli la stessa (o maggiore) priorità del codice. Infatti, si scrivono i test prima di scrivere il codice che deve essere testato e i test stanno con il codice per sempre. I test devono essere eseguiti con successo ogni volta che si fa un integrazione del progetto (il che avviene spesso e più di una volta al giorno).

Scrivere prima i test ha due effetti estremamente importanti.

Per prima cosa, forza una chiara definizione dell'interfaccia di una classe. Ho spesso suggerito alle persone: " si immagini la classe perfetta per risolvere un particolare problema" come un strumento quando si cerca di progettare un sistema. Il test con la strategia di XP va oltre ciò che esso specifica esattamente come la classe deve sembrare al consumatore di quella classe e esattamente come la classe si deve comportare. E questo senza ambiguità. Si può scrivere tutta la prosa, creare tutti i diagrammi che si vogliono descrivendo come una classe si dovrebbe comportare e sembrare, ma niente è reale come un insieme di test. La prima è una lista dei desideri, ma i test sono un contratto che è rafforzato dal compilatore e dal programma in esecuzione. È difficile immaginare una descrizione più concreta di una classe dei test.

Durante la creazione dei test, si deve avere in mente la classe e spesso si scopriranno le funzionalità di cui si ha bisogno che potrebbero mancare durante gli esperimenti con i diagrammi UML, le CRC card, gli use case, ecc...

Il secondo effetto importante che si ottiene nello scrivere prima i test deriva dalla loro esecuzione ogni volta che viene fatta una compilazione del proprio software. Questa attività fornisce l'altra metà del collaudo che viene eseguita dal compilatore. Se si osserva l'evoluzione dei linguaggi di programmazione da questa prospettiva, si noterà che gli autentici miglioramenti nella tecnologia sono avvenuti in realtà intorno ai collaudi. Il linguaggio assembler controllava soltanto la sintassi, ma il C ha imposto alcuni vincoli semantici, che impediscono di fare determinati tipi di errori. I linguaggi OOP impongono ulteriori vincoli semantici che, se ci si pensa, sono in realtà forme di collaudo. "*Questo tipo di dato viene utilizzato in modo appropriato?*"e "*Questa funzione viene chiamata in modo corretto?"* sono i tipi di test che vengono eseguiti dal compilatore o dal sistema a run time. Si è visto che cosa succede quando meccanismi di collaudo di questi tipo vengono incorporati nel linguaggio: la gente è in grado di scrivere programmi più complessi e di metterli in funzione in meno tempo e con minor fatica. Mi sono spesso domandato come mai le cose stiano in questo modo, ma ora mi rendo conto che sono i test: si fa qualcosa di sbagliato e la rete di sicurezza costituita dai test incorporati dice che c'è un problema ed indica dove si trova.

Tuttavia, le forme di collaudo intrinseco fornite dall'impostazione del linguaggio possono arrivare solo fino ad un certo punto. Arriva un momento in cui bisogna farsi avanti ed aggiungere i test rimanenti che producono un insieme completo (in cooperazione con il compilatore ed il sistema a run time) che verifica l'intero programma. E, così come si ha un compilatore che guarda le spalle del programmatore, non si vorrebbe forse che questi test aiutassero fin dall'inizio? Ecco perché si deve scrivere prima i test ed eseguirli automaticamente ad ogni nuova compilazione del proprio sistema. I propri test diventano un'estensione della rete di sicurezza fornita dal linguaggio.

Una delle cose che ho scoperto in merito all'utilizzo di linguaggi di programmazione sempre più potenti è il fatto che mi sento incoraggiato a tentare esperimenti sempre più azzardati, perché so che il linguaggio mi impedirà di sprecare tempo andando a caccia di bachi. La logica dei test di XP fa la stessa cosa per l'intero proprio progetto. Siccome si sa che i propri test intercetteranno sempre eventuali problemi che si creeranno (e che si aggiungeranno sistematicamente nuovi test quando si penserà a quei problemi), si possono fare modifiche importanti, quando serve, senza preoccuparsi di mandare in crisi l'intero progetto. Tutto questo è davvero molto potente.

Programmare in coppia

La programmazione in coppia è contraria al rude individualismo al quale si è indottrinati fin dai primi passi tramite la scuola (dove raggiungiamo gli obiettivi o li manchiamo da soli e lavorare insieme con i compagni è considerato “copiare”) e tramite i mezzi di comunicazione, specialmente i film di Hollywood, nei quali l'eroe di solito combatte contro il bieco conformismo [19]. Anche i programmatori sono considerati modelli di individualismo – “*cowboy della codifica*”, come ama dire Larry Constantine. Eppure, XP, che pure si batte contro il modo di pensare tradizionale, sostiene che il codice andrebbe scritto da due persone per stazioni di lavoro. E che si dovrebbe farlo in un'area che contenga un gruppo di stazioni di lavoro, senza le barriere che piacciono tanto agli arredatori degli uffici. In effetti Beck sostiene che il primo passo verso la conversione a XP consiste nell'arrivare al lavoro muniti di cacciaviti e di chiavi a tubo e smontare tutto ciò che ingombra [20] (per far questo ci vuole un dirigente capace di sviare le ire del reparto che gestisce gli ambienti di lavoro).

Il valore della programmazione in coppia sta nel fatto che una sola persona scrive materialmente il codice mentre l'altra ci pensa sopra. Il pensatore tiene presente il quadro generale, non soltanto il quadro del problema sul quale si lavora, ma le linee guida di XP. Se sono in due a lavorare, è meno probabile che uno di loro possa cavarsela dicendo: "*Non voglio scrivere prima i test*", per esempio. E se quello che codifica resta impantanato, il collega può dargli il cambio. Se restano impantanati entrambi, le loro riflessioni possono essere udite da qualcun altro nella stessa area di lavoro, che può dare una mano. Lavorare in coppia mantiene le cose in movimento e in riga. Cosa probabilmente più importante, rende la programmazione molto più sociale e divertente.

Ho cominciato a utilizzare la programmazione in coppia durante le esercitazioni in alcuni miei seminari e sembra che la cosa migliori in modo significativo l'esperienza di ognuno.

Perchè il C++ ha successo

Uno dei motivi per cui il C++ ha avuto così tanto successo è che lo scopo non era solo trasformare il C in un linguaggio OOP (sebbene si era partiti in quel modo), ma anche di risolvere molti altri problemi che oggi gli sviluppatori fronteggiano, specialmente quelli che hanno grossi investimenti nel C. Tradizionalmente, per i linguaggi OOP si è detto che si dovrebbe abbandonare tutto ciò che si conosce e partire da zero con un nuovo insieme di concetti e nuove sintassi, motivando che alla lunga è meglio perdere tutto il bagaglio di vecchie conoscenze dei linguaggi procedurali. Ciò potrebbe essere vero, alla lunga. Ma nei primi tempi tale bagaglio è prezioso. Gli elementi più utili possono non essere la base di codice esistente (che, dati adeguati strumenti, possono essere tradotti), ma invece la base mentale esistente. Se si è un buon programmatore C e si deve buttar via tutto ciò che si conosce del C per adottare il nuovo linguaggio, si diventa immediatamente molto meno produttivi per molti mesi, fino a che la propria mente non si adatta al nuovo paradigma. Mentre se si può far leva sulle conoscenze acquisite del C ed espanderle, si può continuare ad essere produttivi con ciò che si conosce già mentre si passa nel mondo della OOP. Poichè ognuno ha un proprio modello mentale di programmazione, questo passaggio è abbastanza disordinato poichè esso è privo dello sforzo aggiunto di una partenza con un nuovo linguaggio da uno noto. Quindi le ragioni del successo del C++ sono economiche: costa ancora passare alla OOP, ma il C++ può costare meno[21].

Lo scopo del C++ è migliorare la produttività. Questa produttività si realizza in diversi modi, ma il linguaggio è progettato per aiutare il programmatore quanto più è possibile, impedendo allo stesso tempo per quanto sia possibile, con regole arbitrarie che si usi un particolare insieme di caratteristiche. Il C++ è progettato per essere pratici; le decisioni prese per il linguaggio C++ furono dettate per fornire massimi benefici per il programmatore (almeno, dal punto di vista del C).

Un C migliore

Si può avere un beneficio istantaneo persino se si continua a scrivere in C perchè il C++ ha chiuso molti buchi del C e fornisce un controllo del tipo migliore ed analisi a tempo di compilazione. Si è forzati a dichiarare le funzioni in modo che il compilatore può controllare il loro uso. La necessità del preprocessore è stata virtualmente eliminata per la sostituzione di valore e le macro, che rimuovono un un insieme di bachi difficili da trovare. Il C++ ha una caratteristica chiamata *riferimento* che permette una gestione degli indirizzi

più conveniente per gli argomenti delle funzioni e i valori di ritorno. Una caratteristica detta *namespace* che migliora anche il controllo dei nomi. La gestione dei nomi viene migliorata attraverso una caratteristica detta *function overloading* (sovraccaricamento della funzione), che permette di usare lo stesso nome per funzioni diverse. Ci sono numerose funzionalità più piccole che migliorano la sicurezza del C.

Si sta già imparando

Quando si impara un nuovo linguaggio ci sono problemi di produttività. Nessuna azienda può permettersi di perdere improvvisamente un software engineer produttivo perchè egli o ella sta imparando un nuovo linguaggio. Il C++ è un estensione del C, non una sintassi e un modello di programmazione completamente nuovo. Esso permette di continuare a scrivere codice, applicando le funzionalità gradualmente mentre le si imparano e capiscono. Questa può essere una delle ragioni più importanti del successo del C++.

In aggiunta, tutto il codice C esistente è ancora vitale in C++, ma poichè il compilatore C++ è più esigente, si troveranno spesso errori del C nascosti quando si ricompila il codice in C++.

Efficienza

A volte bisogna trovare un compromesso tra la velocita di esecuzione e la produttività del programmatore. Un modello finanziario, per esempio, può essere utile solo per un breve periodo di tempo, quindi è più importante creare il modello rapidamente che eseguirlo rapidamente. Tuttavia, la maggior parte delle applicazioni richiede gradi di efficienza, quindi il C++ pecca sempre sul lato di un"efficienza maggiore. Poichè i programmatori C tendono ad essere molto consci dell'efficienza, questo è anche un modo di assicurare che essi non potranno dire che il linguaggio è troppo grasso e lento. Molte caratteristiche in C++ sono intese per permettere di migliorare le prestazione quando il codice generato non è abbastanza efficiente.

Non solo non bisogna fare gli stessi controlli a basso livello del C (e scrivere direttamente linguaggio assembler in un programma C++), ma l'evidenza suggerisce che la velocità del programma di un programma C++ ad oggetti tende ad essere tra il ±10% di un programma scritto in C e spesso molto di più[22]. Il progetto prodotto per un programma OOP può essere molto più efficiente della controparte in C.

I sistemi sono più semplici da esprimere e da capire

La classi concepite per adattarsi al problema tendono a esprimerlo meglio. Questo vuol dire che, quando si scrive il codice, si descrive la propria soluzione nei termini dello spazio del problema (“Metti il manico al secchio”) invece che nei termini del computer, che è lo spazio della soluzione (“Imposta nel circuito il bit significa che il relè verrà chiuso”). Si lavori con concetti di livello superiore e si può fare molto di più con una sola riga di codice. L’altro vantaggio che deriva da questa facilità di espressione sta nella manutenzione che (se possiamo credere alle statistiche) assorbe un’enorme quota dei costi durante il ciclo di vita di un programma. Se un programma è più facile da capire, è anche più facile farne la manutenzione. Il che può anche ridurre i costi della creazione e della manutenzione della documentazione.

Massimo potere con le librerie

Il modo più veloce di creare un programma è di usare il codice che è già stato scritto: una libreria. Uno dei maggiori scopi del C++ è di rendere più facile la realizzazione delle librerie. Questo viene realizzato fondendo le librerie in nuovi tipi di dato (classi), in modo che utilizzare una libreria significa aggiungere nuovi tipi al linguaggio. Poichè il compilatore C++ fa attenzione a come viene usata la libreria, garantendo una corretta inizializzazione e pulizia ed assicurando che le funzioni siano chiamate correttamente, ci si può focalizzare su ciò che si vuole la libreria faccia, non su come utilizzarla.

Poichè i nomi possono essere isolati dalle parti del nostro programma usando i namespace del C++, si possono usare quante librerie si vogliono senza gli scontri tra i tipi di nome che avvengono nel C.

Riuso dei sorgenti con i template

C'è una significativa classe di tipi che richiede modifiche ai sorgenti se si vuole riutilizzarli efficacemente. La funzionalità template in C++ esegue le modifiche ai sorgenti automaticamente, fornendo un potente strumento per il riuso del codice di libreria. Un tipo che si progetta usando i template funzionerà con molti altri tipi. I template sono molto utili perchè nascondono le complessità di questo genere per il riuso del codice dal programmatore client.

Gestione degli errori

La gestione degli errori in C è un problema noto ed è spesso ignorato: di solito si incrociano le dita sperando che tutto vada bene. Se si sta costruendo un programma grosso e complesso, non c' è niente di peggio che avere un errore nascosto da qualche parte e non avere nessun indizio di dove può essere. La *gestione delle eccezioni* in C++ (introdotta in questo capitolo ed approfondita per intero nel Volume 2, scaricabile da *www.BruceEckel.com*) è un modo per garantire che un errore venga notato e che accada qualcosa come risultato.

Programmare senza limitazioni

Molti linguaggi tradizionali hanno limitazioni intrinseche per quanto riguarda dimensioni e complessità dei programmi. Il BASIC, per esempio, può essere ottimo per mettere assieme rapide soluzioni per determinate classi di problemi, però se il programma si allunga
su troppe pagine o si avventura al di fuori dal dominio normale dei problemi di quel linguaggio, ci si ritrova a tentare di nuotare in un liquido che diventa sempre più vischioso. Anche il C ha queste limitazioni. Per esempio, quando un programma va oltre le 50000 linee di codice, iniziano problemi di collisione di nomi, effettivamente si finiscono i nomi di funzioni e di variabili. Un altro problema sono i piccoli bachi nel linguaggio C, errori sepolti in un grosso programma che possono essere estremamente difficili da trovare.

Non c'è una chiara linea di demarcazione che segnali quando il linguaggio che si sta utilizzando non va più bene e anche se ci fosse la si ignorerebbe. Non si dice: "*Questo programma in BASIC è diventato davvero troppo lungo; dovrò riscriverlo in C!*". Invece, si tenta di ficcarci dentro ancora un po' di righe per aggiungere giusto una sola nuova funzionalità. E così i costi supplementari cominciano a prendere il sopravvento.

Il C++ è concepito per aiutare a *programmare senza limitazioni*: vale a dire, per cancellare quelle linee di demarcazione derivate dalla complessità che si collocano fra un programma piccolo e uno grande. Non servirà certo l'OOP per scrivere un programma di servizio nello stile "hello world", però le funzionalità sono a disposizione per quando servono. Ed il compilatore è molto aggressivo quando si tratta di stanare errori che generano bachi tanto nei programmi di piccole dimensioni quanto in quelli grandi.

Strategie per la transizione

Se si fa proprio il concetto di OOP, probabilmente la domanda che ci si pone subito dopo sarà: *" Come posso convincere il mio capo/i colleghi/il reparto/gli amici ad utilizzare gli oggetti?"*. Si rifletta su come si farebbe in proprio, da "programmatori indipendenti" a imparare a usare un nuovo linguaggio e un nuovo paradigma di programmazione. Lo si è già fatto in passato. Prima vengono la formazione e gli esempi, poi viene un progetto di prova per dare il senso dei concetti essenziali senza fare nulla che possa creare confusione. Infine viene un progetto del "mondo reale", che fa davvero qualcosa di utile. Nel corso dei primi progetti si continuerà nella propria formazione leggendo, facendo domande agli esperti e scambiando suggerimenti con gli amici. È questo l'approccio che molti programmatori esperti suggeriscono per passare dal C al C++. Il passaggio di un'intera azienda produrrà, naturalmente, alcune dinamiche di gruppo, ma ad ogni punto di svolta verrà utile ricordare come se l'è cavata una singola persona.

Linee guida

Queste che seguono sono alcune linee guida da considerare quando si passa all'OOP e a al C++.

1. Addestramento

Il primo passo da compiere è una qualche forma di addestramento. Si tengano presente gli investimenti che la propria società ha già fatto nel codice e si cerchi di non scardinare tutto per sei o nove mesi mentre tutti cercano di capire come funzionano le interfacce. Si scelga un piccolo gruppo da indottrinare, preferibilmente formato da persone dotate di curiosità, che lavorano bene assieme e che possono assistersi a vicenda mentre imparano il C++.

Talvolta si suggerisce un approccio alternativo, che consiste nel formare tutti i livelli della società in un colpo solo, tenendo corsi di orientamento generale per i dirigenti e corsi di progettazione/programmazione per i capi progetto. Questo è ottimo per imprese di piccole dimensioni che si accingono a dare una svolta radicale al loro modo di lavorare o per il livello divisionale di società di maggiori dimensioni. Siccome, però, i costi sono più elevati, si preferisce di solito iniziare con un addestramento a livello di progetto, eseguire un progetto pilota (magari con l'aiuto di un mentore esterno) e lasciare che le persone che hanno partecipato al progetto diventino i docenti per il resto della società.

2. Progetti a basso rischio

Si cominci con un progetto a basso rischio e si preveda che si sbaglierà qualcosa. Quando si avrà un pò di esperienza, si potranno avviare altri progetti affidandoli ai componenti del primo gruppo di lavoro oppure assegnare a queste persone il compito di dare assistenza tecnica per l'OOP. Il primo progetto potrebbe non funzionare bene la prima volta, quindi non dovrà essere un progetto di importanza critica per la società. Dovrebbe essere qualcosa

di semplice, a sè stante ed istruttivo; questo vuol dire che dovrebbe portare a creare classi che saranno significative per gli altri programmatori della società quando verrà il loro turno per imparare il C++.

3. Trarre ispirazione dai progetti ben riusciti

Si cerchino esempi di buona progettazione orientata agli oggetti prima di cominciare da zero. È molto probabile che qualcuno abbia già risolto il problema e, se proprio non lo hanno risolto esattamente come si presenta, si può probabilmente applicare quello che si è imparato sull'astrazione per modificare uno schema esistente e adattarlo alle proprie necessità. Questo è il principio ispiratore dei *design patterns* trattato nel Volume 2.

4. Utilizzare librerie di classi esistenti

La principale motivazione economica per passare all'OOP è la comodità con la quale si può utilizzare codice esistente sotto forma di librerie di classi (in particolare le librerie Standard C++, che sono illustrate in dettaglio nel Volume due di questo libro). Il più breve ciclo di sviluppo di un'applicazione lo si otterrà quando si dovrà scrivere solo **main()** e si potranno creare ed utilizzare oggetti ricavati da librerie belle e pronte. Tuttavia, certi programmatori alle prime armi non capiscono ciò, non sono a conoscenza di librerie di classi in circolazione oppure, affascinati dal linguaggio, hanno voglia di scrivere classi che potrebbero già esistere. Si otterranno migliori risultati con l'OOP ed il C++ se si fa lo sforzo di cercare e riusare il codice degli altri all'inizio del processo di transizione.

5. Non riscrivere il codice esistente in C++

Sebbene *compilare* il codice C con un compilatore C++ produce di solito (a volte tremendi) benefici per trovare i problemi del vecchio codice, di solito non conviene prendere codice esistente e funzionale per riscriverlo in C++ (bisogna trasformarlo in oggetti, si può inglobare il codice C in classi C++). Ci sono dei benefici, specialmente se il codice deve essere riusato. Ma ci sono possibilità che non si vedano i grossi incrementi di produttività che si speravano, fino a che il progetto non parta da capo. il C++ e la OOP brillano di più quando si porta un progetto dall'idea alla realtà.

Il Management ostacola

Se si è un capo, il proprio lavoro è quello di acquisire risorse per la propria squadra, superare le barriere che impediscono alla squadra di avere successo ed in generale creare l'ambiente più produttivo e gradevole che sia possibile affinché la propria squadra possa fare quei miracoli che di solito vengono chiesti. Passare al C++ ha riflessi in tutte e tre queste categorie e sarebbe davvero meraviglioso se non costasse qualcosa. Sebbene il passaggio al C++ possa essere più economico,in dipendenza da vincoli[23] che si hanno con altre alternative OOP per una squadra di programmatori in C (e probabilmente per programmatori in altri linguaggi procedurali), non è del tutto gratuito e vi sono ostacoli che si farebbe bene a conoscere prima di promuovere il passaggio al C++ all'interno della propria società imbarcandosi nel passaggio vero e proprio.

Costi di partenza

Il costo del passaggio al C++ è molto più che la semplice acquisizione di compilatori C++ (il compilatore GNU C++ è gratuito, quindi non è certo un ostacolo). I costi di medio e lungo periodo si ridurranno al minimo se si investirà in addestramento (e magari in una consulenza per il primo progetto) e se inoltre si individuerà e si acquisterà librerie di classi che risolvono il proprio problema, piuttosto di tentare di costruire quelle librerie. Questi sono costi in denaro contante, che vanno conteggiati in una proposta realistica. Inoltre, vi sono costi nascosti che derivano dalla perdita di produttività mentre si impara un nuovo linguaggio ed eventualmente un nuovo ambiente di programmazione. Addestramento e consulenza possono certamente ridurre al minimo questi costi, ma i componenti della squadra devono superare le loro difficoltà nel capire la nuova tecnologia. Durante questo processo faranno un maggior numero di sbagli (e questo è un vantaggio, perché gli sbagli riconosciuti sono la via più rapida per l'apprendimento) e saranno meno produttivi. Anche in questi casi, almeno per determinati tipi di problemi di programmazione, disponendo delle classi giuste e con un ambiente di sviluppo adeguato, è possibile essere più produttivi mentre si impara il C++ (pur tenendo conto che si fanno più sbagli e si scrivono meno righe di codice al giorno) di quanto non sarebbe se si restasse col C.

Problemi di prestazioni

Una domanda molto diffusa è: "*Non è che l'OOP renda automaticamente i miei programmi molto più voluminosi e più lenti?*"La risposta è: "*Dipende*". Le maggior parte dei linguaggi OOP tradizionali sono stati progettati per scopi di sperimentazione e prototipazione rapida. Quindi essi virtualmente garantiscono un significativo incremento in dimensioni ed una diminuzione di velocità. Il C++, tuttavia, è progettato per la produzione. Quando lo scopo è la prototipazione rapida, si possono mettere insieme componenti il più velocemente possibile ignorando i problemi di efficienza. Se si utilizzano librerie di terzi, di solito esse saranno già state ottimizzate dai loro fornitori; in tutti i casi, questo non è un problema quando si lavora in una prospettiva di sviluppo rapido. Quando si ha un sistema che piace, se è sufficientemente piccolo e veloce, non serve altro. Altrimenti, si comincia a ritoccarlo con uno strumento di profilatura, cercando in primo luogo possibili accelerazioni che si potrebbero ottenere riscrivendo piccole parti del codice. Se questo non basta, si cercano le modifiche che si possono apportare all'implementazione sottostante, in modo che non si debba cambiare il codice che utilizza una classe particolare. Soltanto se nient'altro risolve il problema si ha bisogno di modificare la progettazione. Il fatto che le prestazioni siano così critiche in quella parte della progettazione fa capire che devono entrare a far parte dei criteri primari della progettazione. Con lo sviluppo rapido si ha il vantaggio di accorgersi molto presto di questi problemi.

Come menzionato prima, il numero che è più spesso dato per la differenza in dimensioni e velocità tra C e C++ è ±10% e spesso molto prossimo alla parità. Si può persino ottenere un significativo miglioramento in dimensioni e velocità quando si usa il C++ piuttosto che il C perchè il progetto che si fa in C++ potrebbe essere molto diverso da quello che si farebbe in C.

L'evidenza nelle comparazioni di dimensioni e velocità tra C e C++ tende ad essere aneddotica e a rimanere probabilmente così. Malgrado il numero di persone che suggerisce ad un'azienda di provare a sviluppare lo stesso progetto usando C e C++, nessuna di esse spreca denaro così, a meno che non è molto grande ed interessata in tali progetti di ricerca. Persino in questo caso, sembra che ci sia un modo migliore di spendere il denaro. Quasi

universalmente, i programmatori che sono passati dal C (o qualche altro linguaggio procedurale) al C++ (o qualche altro linguaggio OOP) hanno avuto l'esperienza personale di una grande accelerazione nella loro produttività e questo è il miglior argomento che si possa trovare.

Errori comuni di progettazione

Quando un team comincia a lavorare con l'OOP ed il C++, i programmatori di solito commettono una serie di comuni errori di progettazione. Ciò accade spesso a causa del limitato contributo degli esperti durante il progetto e l'implementazione dei primi progetti, poichè nessun esperto ha sviluppato nell'azienda e ci può essere resistenza ad acquisire consulenti. È facile avere troppo presto la sensazione di aver capito l'OOP e di andar via per la tangente. Qualche cosa che è ovvio per qualcuno che ha esperienza con il linguaggio può essere un soggetto di un grande dibattito interno per un novizio. Questo trauma può essere evitato usando l'esperienza di un esperto esterno per l'addestramento ed il mentoring.

Dall'altro lato, il fatto che è facile commettere questi errori di disegno indica i principali svantaggi del C++: la sua compatibilità all'indietro con il C (naturalmente, questa è anche la sua forza principale). Per realizzare l'impresa di compilare un codice C, il linguaggio ha dovuto fare qualche compromesso, che consiste in qualche punto oscuro: essi sono una realtà e constituiscono le maggiori difficoltà per l'apprendimento del linguaggio. In questo libro e nel volume seguente (ed gli altri libri, si veda l'Appendice C), si cercherà di rivelare la maggior parte delle insidie che si incontrano quando si lavora con il C++. Bisognerebbe essere sempre consapevoli che ci sono dei buchi nella rete di sicurezza.

Sommario

Questo capitolo cerca di dare un'idea dei vari argomenti della programmazione orientata agli oggetti, incluso il perchè la OOP è qualcosa di diverso e perchè il C++ in particolare è diverso, presentando concetti delle metodologie OOP ed infine i tipi di problematiche che si incontreranno quando la propria azienda comincerà ad usare la OOP ed il C++.

La OOP ed il C++ possono non essere per tutti. È importante valutare i propri bisogni e decidere se il C++ soddisferà in maniera ottimale quelle necessità o se è migliore un altro sistema di programmazione (incluso quello che si sta utilizzando correntemente). Se si sa che le proprie necessità saranno molto specializzate per il fututo e si hanno dei vincoli specifici che potrebbero non essere soddisfatti dal C++, allora bisogno investigare su possibili alternative[24]. Anche se alla fine si sceglie il C++ come linguaggio, si capiranno almeno quali sono le opzioni e si avrà una chiara visione del perchè si è presa quella direzione.

Si sa come appare un programma procedurale: definizioni di dati e chiamate a funzioni. Per cercare il significato di tale programma si deve lavorare un pò, guardando tra chiamate a funzioni e concetti a basso livello per creare un modello nella propria mente. Questa è la ragione per cui abbiamo bisogno di una rappresentazione intermedia quando si progettano programmi procedurali. Di per se, questi programmi tendono a confondere perchè i termini delle espressioni sono orientati più verso il computer che verso il problema che si sta risolvendo.

Poiche il C++ aggiunge molti concetti nuovi al linguaggio C, l'assunzione naturale può essere che il **main()** in un programma C++ sarà molto più complicato di un equivalente programma C. Qui si rimarrà piacevolmente sorpresi: un programma C++ ben scritto è generalmente molto più semplice e più facile da capire di un equivalente programma C. Ciò che si vedrà sono le definizioni degli oggetti che rappresentano i concetti nel nostro spazio del problema (invece che concetti relativi ad aspetti del computer) e i messaggi mandati a quegli oggetti per rappresentare le attività in quello spazio. Una delle delizie della programmazione orientata agli oggetti è che, con un programma ben progettato, è facile capire il codice leggendolo. Di solito c'è molto meno codice, poichè molti dei problemi saranno risolti riutilizzando il codice delle librerie esistenti.

---

[4] Si veda *Multiparadigm Programming in Leda* di Timothy Budd (Addison-Wesley 1995).

[5] Si può trovare un'implementazione interessante di questo problema nel Volume 2 di questo libro, disponibile su www.BruceEckel.com.

[6] qualcuno fa distinzione, indicando che tipo determina l'interfaccia mentre classe è una particolare implementazione di quella interfaccia.

[7] Sono in debito con il mio amico Scott Meyers per questo termine.

[8] Di solito la maggior parte dei diagrammi ha già abbastanza dettagli, quindi non si avrà bisogno di specificare se si utilizza l'aggregazione o la composizione.

[9] Un eccellente esempio di ciò è *UML Distilled*, by Martin Fowler (Addison-Wesley 2000), che riduce il processo UML, spesso opprimente, ad un maneggevole sottoinsieme.

[10] La mia regola del pollice per stimare tali progetti è: se c'è più di un fattore jolly, non provo neache a cercare di pianificare quanto tempo ci vorrà o quanto costerà fino a che non si ha un prototipo funzionante. Ci sono troppi gradi di libertà.

[11] Grazie per l'aiuto di James H Jarrett.

[12] Maggiori informazioni sugli use case possono essere trovate in *Applying Use Cases* di Schneider & Winters (Addison-Wesley 1998) e *Use Case Driven Object Modeling with UML* di Rosenberg (Addison-Wesley 1999).

[13] La mia personale opinione su ciò è cambiata in seguito. Duplicare ed aggiungere il 10 percento darà una accurata stima (assumento che non ci sono molti fattori jolly), ma si deve lavorare ancora abbastanza diligentemente per finire in quel tempo. Se si vuole tempo per renderlo elegante ed apprezzarsi del processo, il corretto moltipicatore è tre o quattro, io credo.

[14] Per chi inizia, raccomando il predetto *UML Distilled*.

[15] Python (www.Python.org) viene spesso usato come un "pseudo codice eseguibile ".

[16] Almeno un aspetto dell'evoluzione è discusso nel libro di Martin Fowler *Refactoring: improving the design of existing code* (Addison-Wesley 1999). Questi libro utilizza esclusivamente esempi in Java.

[17] Questo termine è esplorato nel capitolo *Design Patterns* del Volume 2.

[18] Ciò è tipo la "prototipazione rapida" dove si suppone di costruire una versione rapida e sporca in modo da imparare qualcosa del sistema e poi gettare via il prototipo e costruirlo esatto. Il problema della prototipazione rapida è che la gente non getta via il prototipo, ma ci costruisce sopra. Combinata con la mancanza di struttura della programmazione procedurale, ciò spesso produce sistemi disordinati che sono costosi da manuntenere.

[19] Sebbene ciò possa essere una prospettiva più Americana, le storie di Hollywood arrivano ovunque.

[20] Incluso (specialemente) il sistema PA. Una volta ho lavorato in una azienda che insisteva nel diffondere ogni telefonata che arrivava ad ogni executive ed essa interrompeva costantemente la nostra produttività (ma i manager non concepivano di soffocare una cosa così importante come il PA). Alla fine, quando nessuno guardava ho cominciato a tagliare i cavi delle casse.

[21] Dico posso perchè, a causa della complessità del C++, potrebbe convenire passare a Java. Ma la decisione di quale linguaggio scegliere ha molti fattori e si assume che si è scelto il C++.

[22] Tuttavia, si vedano gli articoli di Dan Saks nel *C/C++ User's Journal* per importanti inestigazioni sulle performance della libreria C++ .

[23]A causa di miglioramenti della produttività, Java dovrebbe essere preso in considerazione a questo punto.

[24] In particolare, si raccomanda di vedere Java (http://java.sun.com) e Python (http://www.Python.org).

2: Costruire & Usare gli Oggetti

Questo capitolo introdurrà la sintassi e i concetti necessari per la costruzione di programmi in C++ permettendo di scrivere ed eseguire qualche piccolo programma orientato agli oggetti. Nel capitolo seguente verranno illustrate in dettaglio le sintassi base del C e del C++.

Leggendo questo capitolo, si avrà un'infarinatura di cosa significa programmare ad oggetti in C++ e si scopriranno anche alcune delle ragioni dell'entusiasmo che circondano questo linguaggio. Ciò dovrebbe essere sufficiente per passare al Capitolo 3, che può essere più esaustivo poichè contiene la maggior parte dei dettagli del linguaggio C.

Il tipo di dato definito dall'utente o *classe*, è ciò che distingue il C++ dai linguaggi procedurali. Una classe è un nuovo tipo di dato che viene creato per risolvere un particolare tipo di problema. Una volta creato, chiunque può usarla senza sapere nello specifico come funziona o persino come sono fatte le classi. Questo capitolo tratta le classi come se esse fossero un altro tipo di dato predefinito disponibile all'utilizzo nei programmi.

Le classi che qualcun altro ha creato sono tipicamente impacchettate in una libreria. Questo capitolo usa diverse librerie di classi che sono disponibili in tutte le implementazioni del C++. Una libreria standard particolarmente importante è la iostreams, che (tra le altre cose) ci permette di leggere dai file e dalla tastiera e di scrivere su file e su schermo. Si vedrà anche la pratica classe **string** e il contenitore **vector** dalla libreria Standard C++. Alla fine del capitolo, si vedrà come sia facile usare le classi delle librerie predefinite.

Per creare il nostro primo programma si devono capire gli strumenti utilizzati per costruire le applicazioni.

Il processo di traduzione del linguaggio

Tutti i linguaggi del computer sono tradotti da qualcosa che tende ad essere facile da capirsi per un umano (*codice sorgente*) verso qualcosa che è eseguito su un computer(*istruzioni macchina*). Tradizionalmente i traduttori si divisono in due classi: *interpreti* e *compilatori*.

Interpreti

Un interprete traduce il codice sorgente in unità (che possono comprendere gruppi di istruzioni macchina) ed esegue immediatamente queste unità. Il BASIC, per esempio, è stato un linguaggio interpretato popolare. Gli interpreti BASIC tradizionali traducono ed eseguono una linea alla volta e poi dimenticano che quella linea è stata tradotta. Ciò li rende lenti, poichè essi devono ritradurre ogni codice ripetuto. Il BASIC viene anche compilato per essere più veloce. Gli interpreti più moderni, come quelli per il linguaggio Python, traducono l'intero programma in un linguaggio intermedio che viene poi eseguito da un interprete più veloce[25].

Gli interpreti hanno molti vantaggi. La transizione dal codice scritto a quello eseguito è quasi immediato ed il codice sorgente è sempre disponibile perciò l'interprete può essere più dettagliato quando si verifica un errore. Il beneficio spesso citato dagli interpreti è la faciltà di interazione e lo sviluppo rapido (ma non necessariamente l'esecuzione) dei programmi.

I linguaggi interpretati hanno spesso serie limitazioni quando si costruiscono grossi progetti (Python sembra essere un eccezione). L'interprete (o la versione ridotta) deve sempre essere in memoria per eseguire il codice e persino l'interprete più veloce può introdurre inaccettabili restrizioni alla velocità. La maggior parte degli interpreti richiede che il sorgente completo sia portato nell'interprete tutto insieme. Ciò introduce non solo una limitazione di spazio, ma può anche causare bug più difficili da trovare se il linguaggio non fornisce un strumento per localizzare l'effetto dei diversi pezzi di codice.

Compilatori

Un compilatore traduce codice sorgente direttamente in linguaggio assembler o in instruzioni macchina. Il prodotto finale è uno o più file contenenti il codice macchina. Questo è un processo complesso e di solito richiede diversi passi. La transizione dal codice scritto al codice eseguito è significamente più lunga con un compilatore.

In base all'acume di chi ha scritto il compilatore, i programmi generati dai compilatori tendono a richiedere molto meno spazio per essere eseguiti e sono eseguiti molto più rapidamente. Sebbene le dimensione e la velocità sono probabilmente le ragioni più spesso citate per l'uso di un compilatore, in molte situazioni non sono le ragioni più importanti. Alcuni linguaggi (come il C) sono progettati per permettere la compilazione indipendente di pezzi di programma. Questi pezzi sono alla fine combinati in un programma *eseguibile* finale da uno strumento detto *linker*. Questo processo è chiamato *compilazione separata*.

La compilazione separata ha molti benefici. Un programma che, preso tutto insieme eccederebbe il limite del compilatore o dell'ambiente di compilazione, può essere compilato in pezzi. I Programmi possono essere costruiti e testati un pezzo alla volta. Una volta che un pezzo funziona, può essere salvato e trattato come un blocco da costruzione. Le raccolte di pezzi testati e funzionanti possono essere combinati in *librerie* per essere usati da altri programmatori. Man mano che ogni pezzo viene creato, la complessità degli altri pezzi viene nascosta. Tutte queste caratteristiche aiutano la creazione di programmi di grosse dimensioni[26].

Le caratteristiche di debug dei compilatori sono andate migliorando significativamente. I primi compilatori generavano solo codice macchina ed i programmatori inserivano dei comandi di stampa per vedere cosa stava succedento. Ciò non era sempre efficace. I compilatori moderni possono inserire informazioni sul codice nel programma eseguibile. Questa informazione è usata dai potenti *debugger a livello di sorgente* per mostrare esattamente cosa sta succedento in un programma tracciando il suo avanzamento nel sorgente.

Qualche compilatore affronta il problema della velocità di compilazione usando la *compilazione in memoria*. La maggior parte dei compilatori funziona con file, leggendo e scrivendo in ogni passo del processo di compilazione. I compilatori in memoria mantegono il programma del compilatore nella RAM. La compilazione di programmi piccoli può sembrare veloce come un interprete.

Il processo di compilazione

Per programmare in C ed in C++, si ha bisogno di capire i passi e gli strumenti nel processo di compilazione. Alcuni linguaggi (C e C++ in particolare) cominciano la compilazione eseguendo un *preprocessore* sul sorgente. il preprocessore è un semplice programma che rimpiazza pattern nel codice sorgente con altri patter che il programmatore ha definito (usando le *direttive del preprocessore*). Le direttive del preprocessore sono utilizzate per risparmiare la scrittura ed aumentare la leggibilità del codice. (Più avanti nel libro, si imparerà come il design del C++ è inteso per scoraggiare un frequente uso del preprocessore, perchè può causare bug). Il codice pre-processato viene spesso scritto in un file intermedio.

I compilatori di solito eseguono il loro lavoro in due passi. Il primo passo parsifica il codice preprocessato. Il compilatore spezza il codice sorgente in piccole unità e le organizza in una struttura chiamata *albero*. Nell'espressione **A+B** gli elementi **A,+** e **B** sono foglie sull'albero di parsificazione.

Un *ottimizzatore globale* è usato a volte tra il primo ed il secondo passo per produrre un codice più piccolo e veloce.

Nel secondo passaggio, il generatore di codice utilizza l'albero di parsificazione e genera il linguaggio assembler o il codice macchina per i nodi dell'albero. Se il generatore di codice crea codice assembler, l'assembler deve essere eseguito. Il risultato finale in entrambi i casi è un modulo oggetto (un file che tipicamente ha estensione **.o** oppure **.obj**). Un *ottimizzatore peephole* è a volte utilizzato nel secondo passaggio per trovare pezzi di codice che contengono comandi ridondanti del linguaggio assembler.

L'uso della parola "oggetto" per descrivere pezzi di codice macchina è un artificio non fortunato. La parola è stata usata prima che la programmazione orientata agli oggetti fosse di uso generale. "Oggetto" viene usato nello stesso senso di "scopo" quando si discute di compilazione, mentre nella programmazione object-oriented significa "una cosa delimitata".

Il linker combina una lista di oggetti modulo in un programma eseguibile che può essere caricato ed eseguito dal sistema operativo. Quando una funzione in un modulo oggetto fa riferimento ad una funzione o variabile in un altro modulo oggetto, il linker risolve questi riferimenti; si assicura che tutte le funzioni esterne ed i dati di cui si reclama l'esistenza esistano durante la compilazione. Il linker aggiunge anche uno speciale modulo oggetto per eseguire attività di avviamento.

Il linker può cercare in file speciali chiamati librerie in modo da risolvere tutti i suoi riferimenti. Una libreria contiene un insieme di moduli oggetto in un singolo file. Una libreria viene creata e manutenuta da un programma chiamato *librarian*.

Controllo del tipo statico

Il compilatore esegue un *controllo del tipo* durante il primo passaggio. Il controllo del tipo testa l'uso opportuno degli argomenti nelle funzioni e previene molti tipi di errori di programmazione. Poichè il controllo del tipo avviene durante la compilazione invece che quando il programma è in esecuzione, esso è detto *controllo del tipo statico* .

Qualche linguaggio orientato agli oggetti (particolarmente Java) esegue il controllo del tipo statico a runtime (*controllo del tipo dinamico*). Se combinato con il controllo del tipo statico, il controllo del tipo dinamico è più potente del controllo statico da solo. Tuttavia, aggiunge un overhead all'esecuzione del programma.

Il C++ usa il controllo del tipo statico perchè il linguaggio non può eseguire nessuna particolare azione per operazioni cattive. Il controllo statico del tipo notifica all'utente circa gli usi errati dei tipi durante la compilazione e così massimizza la velocità di esecuzione. Man mano che si imparerà il C++, si vedrà che le scelte di design del linguaggio favoriscono la programmazione orientata alla produzione e l'altà velocità, per cui è famoso il C.

Si può disabilitare il controllo del tipo statico in C++. Si può anche fare il proprio controllo del tipo dinamico, si deve solo scrivere il codice.

Strumenti per la compilazione separata

La compilazione separata è particolarmente importante quando si costruiscono grossi progetti. In C e C++, un programma può essere creato in pezzi piccoli, maneggevoli e testati indipendentemente. Lo strumento fondamentale per dividere un programma in pezzi è la capacità di creare routine o sottoprogrammi. In C e C++, un sottoprogramma è chiamato *funzione* e le funzioni sono pezzi di codice che possono essere piazzati in file diversi, permettono la compilazione separata. Messo in altro modo, la funzione è un'unità atomica di codice, poichè non si può avere parte di una funzione in un unico file e un'altra parte in un file diverso; l'intera funzione deve essere messa in un singolo file (sebbene i file possono e devono contenere più di una funzione).

Quando si chiama una funzione, si passano tipicamente degli *argomenti*, che sono i valori con cui la funzione lavora durante la sua esecuzione. Quando la funzione è terminata, si ottiene un *valore di ritorno*, un valore che la funzione ci riporta come risultato. È anche possibile scrivere funzioni che non prendono nessun argomento e non restituiscono nessun valore.

Per creare un programma con file multipli, le funzioni in un unico file devono accedere a funzioni e dati in altri file. Quando si compila un file, il compilatore C o C++ deve conoscere le funzioni e i dati negli altri file, in particolare i loro nomi ed il corretto uso. Il compilatore assicura che le funzioni e i dati siano usati correttamente. Questo processo di dire al compilatore i nomi delle funzioni e dati esterni e come dovrebbero essere è detto *dichiarazione*. Una volta che si dichiara una funzione o una variabile, il compilatore sa come eseguire il controllo per essere sicuro che è stato usato correttamente.

Dichiarazioni e definizioni

È importante capire la differenza tra *dichiarazioni* e *definizioni* perchè questi termini saranno usati in maniera precisa nel libro. Essenzialmente tutti i programmi C e C++ richiedono dichiarazioni. Prima che si possa scrivere il primo programma, c'è bisogno di capire il modo corretto di scrivere una dichiarazione.

Una *dichiarazione* introduce un nome, un indentificativo, nel compilatore. Dice al compilatore "Questa funzione o variabile esiste da qualche parte e qui è come dovrebbe apparire. Una definizione dall'altra lato dice: "Crea questa variabile qui o Crea questa

funzione qui. ". Essa alloca memoria per il nome. Il significato funziona sia se si sta parlando di una variabile che di una funzione; in entrambi i casi, al punto della definizione il compilatore alloca memoria. Per una variabile, il compilatore determina quanto grande quella variabile sia e causa la generazione di spazio in memoria per mantenere i dati di quella variabile. Per una funzione, il compilatore genera codice, che termina con l'occupare spazio in memoria.

Si può dichiarare una variabile o una funzione in diversi posti, ma ci deve essere una sola definizione in C e C++ (questo è a volte detto ODR: *one-definition rule* una regola di defizione). Quando il linker sta unendo tutti i moduli oggetto, di solito protesta se trova più di una definizione della stessa funzione o variabile.

Una definizione può anche essere una dichiarazione. Se il compilatore non ha visto il nome **x** prima che si definisca **int x**; , il compilatore vede il nome come una dichiarazione e alloca spazio per esso tutto in una volta.

Sintassi di dichiarazione delle funzioni

Una dichiarazione di funzione in C e in C++ richiede il nome della funzione, i tipi degli argomenti passati alla funzione ed il valore di ritorno. Per esempio, qui c'è una dichiarazione per una funzione chiamata **func1()** che prende due argomenti interi (gli interi sono denotati in C/C++con la parola riservata **int**) e restituisce un intero:

```
int func1(int,int);
```

La prima parola riservata che si vede è il valore di ritorno: **int**. Gli argomenti sono racchiusi in parentesi dopo il nome della funzione nell'ordine in cui sono usati. Il punto e virgola indica la fine della dichiarazione; in questo caso, dice al compilatore "questo è tutto" non c'è definizione di funzione qui!

Le dichiarazioni C e C++ tentano di imitare la forma di uso dell'argomento. Per esempio, se **a** è un altro intero la funzione di sopra potrebbe essere usata in questo modo:

```
a = func1(2,3);
```

Poichè **func1()** restituisce un intero, il compilatore C o C++ controllerà l'uso di **func1()** per essere sicuro che **a** può accettare il valore di ritorno e che gli argomenti siano appropriati.

Gli argomenti nelle dichiarazioni della funzione possono avere dei nomi. Il compilatore li ignora ma essi possono essere utili come congegni mnemonici per l'utente. Per esempio, possiamo dichiarare **func1()** in un modo diverso che ha lo stesso significato:

```
int func1(int lunghezza, int larghezza);
```

Beccato!

C'è una significativa differenza tra il C ed il C++ per le funzioni con una lista degli argomenti vuota. In C, la dichiarazione:

```
int func2();
```

significa "una funzione con qualsiasi numero e tipo di argomento". Ciò evita il controllo del tipo, perciò nel C++ significa "una funzione con nessun argomento".

Definizione di funzioni.

Le definizioni delle funzioni sembrano dichiarazioni di funzioni tranne che esse hanno un corpo. Un corpo è un insieme di comandi racchiusi tra parentesi. Le parentesi denotano l'inizio e la fine di un blocco di codice. Per dare una definizione a **func1()** con un corpo vuoto (un corpo non contente codice), si può scrivere:

```
int func1(int lunghezza, int larghezza) { }
```

Si noti che nella definizione di funzione, le parentesi rimpiazzano il punto e virgola. Poichè le parentesi circondano un comando od un gruppo di comandi, non c'è bisogno del punto e virgola. Si noti anche che gli argomenti nella definizione della funzione devono avere i nome se si vuole usare gli argomenti nel corpo della funzione (poichè essi non sono mai usati qui, essi sono opzionali).

Sintassi di dichiarazione delle variabili

Il significato attribuito alla frase "dichiarazione di variabile" è stato storicamente confuso e contradditorio ed è importante che si capisca la corretta definizione per saper leggere correttamente il codice. Una dichiarazione di variabile dice al compilatore come una variabile appare. Essa dice: "So che non hai mai visto questo nome prima, ma ti prometto che esiste da qualche parte ed è di tipo X".

In una dichiarazione di funzione, si dà un tipo (il valore di ritorno), il nome della funzione, la lista degli argomenti ed il punto e virgola. Ciò è sufficiente per far capire al compilatore che è una dichiarazione e che come dovrebbe apparire la funzione. Per deduzione, una dichiarazione di variabile potrebbe essere un tipo seguito da un nome. Per esempio:

```
int a;
```

potrebbe dichiarare la variabile **a** come un intero, usando la logica di sopra. Qui c'è il conflitto: c'è abbastanza informazione nel codice di sopra perchè il compilatore crei spazio per un intero chiamato **a** e ciò è quello che accade. Per risolvere questo dilemma, una parola riservata è stata necessaria per il C ed il C++ per dire: "Questa è solo una dichiarazione, la sua definizione è altrove". La parola riservata è **extern**. Essa può significare che la definizione è esterna al file o che la definizione appare più avanti nel file.

Dichiarare una variabile senza definirla significa usare la parola chiave **extern** prima di una descrizione della variabile:

```
extern int a;
```

extern può anche essere applicato alle dichiarazioni di funzioni. Per **func1(),** ecco come appare:

```
extern int func1(int lunghezza, int larghezza);
```

Questa dichiarazione è equivalente alla precedenti dichiarazioni di **func1()**. Poichè non c'è il corpo della funzione, il compilatore deve trattarla come un dichiarazione di funzione

piuttosto che una definizione di funzione. La parola riservata **extern** è perciò superflua ed opzionale per le dichiarazione di funzione. È un peccato che i progettisti del C non abbiano richiesto l'uso di **extern** per la dichiarazione delle funzioni, sarebbe stato più consistente e avrebbe confuso meno (ma avrebbe richiesto più caratteri da scrivere, ciò probabilmente spiega la decisione).

Ecco altri esempi di dichiarazione:

```
//: C02:Declare.cpp
// esempi di dichiarazioni & definizioni
extern int i; // dichiarazione senza definzione
extern float f(float); // dichiarazione di funzione

float b;  // dichiarazione & definizione
float f(float a) {  // definizione
  return a + 1.0;
}

int i; // definizione
int h(int x) { // dichiarazione & definizione
  return x + 1;
}

int main() {
  b = 1.0;
  i = 2;
  f(b);
  h(i);
} ///:~
```

Nelle dichiarazioni delle funzione, gli identificatori degli argomenti sono opzionali. Nelle definizioni sono richiesti (gli identificatori sono richiesti solo in C non in C++).

Includere un header

Molte librerie contengono numeri significativi di funzioni e variabili. Per risparmiare lavoro ed assicurare coerenza quando si fanno dichiarazioni esterne per questi pezzi, il C ed il C++ usano un componente chiamato *file header*. Un file header è un file contentente le dichiarazioni esterne per una libreria; esso ha convenzionalmente una estenzione "h", per esempio **headerfile.h** (si può anche vedere qualche vecchio codice che utilizza altre estensioni come **.hxx** o **.hpp** ma è raro).

Il programmatore che crea la libreria fornisce il file header. Per dichiarare le funzioni e le variabili esterne nella libreria, l'utente semplicemente include il file header. Per includere il file header, si usa la direttiva del preprocessore **#include**. Questo dice al preprocessore per aprire il file header ed inserire il suo contenuto dove appare il comando **#include**. Un **#include** può menzionare un file in due modi: con < > oppure con doppie virgolette.

I nomi di file tra le < > come:

```
#include <header>
```

dicono al preprocessore di cercare un file in un modo che è una nostra implementazione, ma tipicamente ci sarà una specie di percorso di ricerca per l'include, che si specifica nell'ambiente di sviluppo o dando una linea di comando al compilatore. Il meccanismo per impostare il percorso di ricerca può variare tra macchine, sistemi operativi e implementazioni del C++ e può richiedere qualche indagine da parte propria.

I nomi dei file tra doppie virgolette:

```
#include "local.h"
```

dicono al preprocessore di cercare un file in un modo definito dall'implementazione (secondo la specifica). Questo tipicamente vuol dire ricercare il file nella cartella corrente. Se il file non viente trovato, allora la direttiva di include è riprocessata come se avesse le parentesi ad angolo invece delle virgolette.

Per l'include dell'header file della iostream si deve scrivere:

```
#include <iostream>
```

Il preprocessore cercherà l'header file della iostream (spesso in una sottocartella chiamata "include") e lo inserirà.

Formato include C++ Standard

Con l'evolvere del C++, i fornitori di compilatori scelsero estensioni diverse per i nomi dei file. In aggiunta, vari sistemi operativi avevano restrizioni diverse sui nomi dei file, in particolare sulla lunghezza del nome. Questi problemi causarono problemi di portabilità del codice. Per smussare questi angoli, lo standard usa un formato che permette nomi di file più lunghi dei noti otto caratteri ed elimina le estensioni. Per esempio, invece di includere alla vecchia maniera **iostream.h** :

```
#include <iostream.h>
```

si può ora scrivere:

```
#include <iostream>
```

Il traduttore può implementare il comando di include in un modo che soddisfa i bisogni di quel particolare compilatore e sistema operativo, se necessario troncando il nome ed aggiungendo una estensione. Naturalmente, si può anche copiare gli header file dati dal fornitore del compilatore ed usarli senza estensione se si vuole usare questo stile.

Le librerie che sono state ereditate dal C sono ancora disponibili con l'estensione tradizionale **.h** . Tuttavia, li si può anche usare con i C++ più moderni preapponendo una **"c"** prima del nome:

```
#include <stdio.h>
#include <stdlib.h>
```

diventa:

```
#include <cstdio>
#include <cstdlib>
```

E così via per tutti gli header C. Ciò fa distinguere al lettore se si sta usando una libreria C o C++.

L'effetto del nuovo modo di inclusione non è identico al vecchio: usando **.h** si ottiene la vecchia versione senza template, omettendo **.h** si usa la nuova versione con i template. Di solito si possono avere problemi se si cerca di mischiare le due forme in un singolo programma.

Linking

Il linker raccoglie moduli oggetto (che spesso hanno estensione **.o** oppure **.obj**), generati dal compiler, in un programma eseguibile che il sistema operativo può caricare ed eseguire. Questa è l'ultima fase del processo di compilazione.

Le caratteristiche del linker possono variare da sistema a sistema. In generale, si dicono al linker solo i nomi dei moduli oggetto e delle librerie che si vogliono linkare insieme ed il nome dell' eseguibile. Qualche sistema richiede che si invochi il linker. Con la maggior parte dei pacchetti C++ si invoca il linker dal compilatore. In molte situazioni, il linker viene invocato in un modo che a noi è trasparente.

Qualche vecchio linker non cercherà i file oggetto e le librerie più di una volta e cercheranno in una lista che gli è stata data da sinistra verso destra. Ciò significa che l'ordine dei file oggetto e delle librerie è importante. Se si ha un problema misterioso che si presenta a tempo di link, una possibilità è che l'ordine in cui i file sono dati al linker.

Utilizzo delle librerie

Ora che si conosce la terminologia di base, si può capire come usare una libreria. Per usare una libreria:

1. Includere l'header file della libreria.
2. Usare le funzioni e le variabili della libreria.
3. Linkare la libreria nel programma eseguibile.

Questi passi si applicano anche quando i moduli oggetto non sono combinati in una libreria. L'inclusione di un header file ed il link dei moduli degli oggetti sono i passi base per la compilazione separata sia in C che in C++.

Come il linker cerca una libreria

Quando si fa riferimento esterno ad una funzione o variabile in C o C++, il linker, incontrando questo riferimento, può fare due cose. Se non ha già incontrato la definizione della funzione o della variabile, aggiunge l'identificatore alla sua lista dei "riferimenti non risolti". Se il linker ha già incontrato la definizione, il riferimento viene risolto.

Se il linker non trova la definizione nella lista dei moduli oggetto, cerca le librerie. Le librerie hanno una specie di indice in modo che il linker non ha bisogno di cercare attraverso tutti gli moduli oggetto nella libreria, esso guarda solo nell'indice. Quando il linker trova una definizione in una libreria, l'intero modulo oggetto, non solo la definizione della funzione, viene linkata nel file eseguibile. Si noti che non viene linkata l'intera libreria l'intera libreria, ma solo il modulo oggetto della libreria che contiene la definizione che si

desidera (altrimenti i programmi sarebbero grandi senza bisogno). Se si vuole minimizzare la dimensione del programma eseguibile, si può considerare di mettere una singola funzione in ogni file del codice sorgente quando si costruiscono le proprie librerie. Ciò richiede maggiore lavoro[27], ma può essere utile all'utente.

Poichè il linker cerca i file nell'ordine in cui vengono dati, si può dare precedenza all'uso di una funzione di libreria inserendo un file con la propria funzione, usando lo stesso nome della funzione, nella lista prima che appaia il nome della libreria. Poichè il linker risolve qualsiasi riferimento a questa funzione usando la nostra funzione prima di cercare nella libreria, la nostra funzione viene usata al posto della funzione della libreria. Si noti che questo può anche essere un bug e i namespace del C++ prevengono ciò.

Aggiunte segrete

Quando viene creato un programma eseguibile C o C++, alcuni pezzi vengono linkati segretamente. Uno di questi è il modulo di startup, che contiene le routine di inizializzazione che devono essere eseguite in qualsiasi momento quando un programma C o C++ inizia. Queste routine impostano lo stack e inizializza alcune variabili del programma.

Il linker cerca sempre nella libreria standard le versioni compilate di qualsiasi funzione standard chiamata dal programma. Poichè la ricerca avviene sempre nella libreria standard, si può usare qualsiasi cosa della libreria semplicemente includendo l'appropriato file header nel programma; non si deve dire di cercare nella libreria standard. Le funzioni iostream, per esempio, sono nella libreria Standard C++. Per usarle si deve solo includere l'header file **<iostream>.**

Se si sta usanto un libreria aggiuntiva, si deve esplicitamente aggiungere il nome della libreria alla lista di file gestiti dal linker.

Usare librerie C

Proprio perchè si sta scrivendo codice in C++, nessuno ci impedisce di usare funzioni di una libreria. Infatti, l'intera libreria C è inclusa per default nel C++ Standard. È stato fatto un gran lavoro per noi in queste funzioni, quindi ci possono far risparmiare un sacco di tempo.

Questo libro userà le funzioni di libreria del C++ Standard (e quindi anche C Standard) quando servono, ma saranno usate solo funzioni di libreria standard, per assicurare la portabilità dei programmi. Nei pochi casi in cui le funzioni di libreria che devono essere usate non sono C++ Standard, saranno fatti tutti i tentativi per usare funzioni POSIX-compliant. POSIX è uno standard basato su uno sforzo di standardizzazione Unix che include funzioni che vanno oltre l'ambito della libreria C++. Ci si può generalmente aspettare di trovare funzioni POSIX su piattaforme UNIX (in particolare su Linux). Per esempio, se si sta usando il multithreading è meglio usare la libreria dei thread POSIX perchè il codice sarà più facile da capire, portare e manutenere (e la libreria thread POSIX sarà usata per i suoi vantaggi con il sistema operativo, se sono fornite).

Il tuo primo programma C++

Ora si conosce abbastanza per creare e compilare un programma. Il programma userà le classi Standard del C++ iostream. Esse leggono e scrivono su file e da "standard" input ed output (che normalmente è la console, ma si può redirezionare ai file o ai device). In questo semplice programma, un oggetto stream sarà usato per stampare un messaggio sullo schermo.

Usare le classi di iostream

Per dichiarare le funzioni ed i dati esterni nella classe di iostream, si deve include un file header con la dichiarazione

```
#include <iostream>
```

Il primo programma usa il concetto di standard output, che significa " un posto di scopo generico per mandare l'output". Si vedranno altri esempi che utilizzano lo standard output in modi diversi, ma qui esso utilizza solo la console. Il package iostream automaticamente definisce una variabile (un oggetto) chiamato **cout** che accetta tutti i dati da inviare allo standard output.

Per inviare dati allo standard output, si usa l'operatore <<. I programmatori C conoscono questo operatore come lo shift a sinistra di bit, che sarà descritto nel prossimo capitolo. Basta dire che uno shift a sinistra di bit non ha niente a che fare con l'output. Tuttavia, il C++ permette l'overloading degli operatori (sovraccaricamento degli operatori). Quando si usa l'overloading con un operatore, esso gli fornisce un nuovo significato quando viene usato con un oggetto di un particolare tipo. Con gli oggetti iostream, l'operatore << significa " manda a ". Per esempio:

```
cout << "salve!";
```

manda la stringa "salve! " all'oggetto chiamato **cout** (che è l'abbreviazione di "console input").

Quanto detto è sufficiente per quanto riguarda l'operatore di overload. Il capitolo 12 tratta l'overloading degli operatori in dettaglio.

Namespace

Come menzionato nel Capitolo 1, uno dei problemi incontrati nel linguaggio C è che si finiscono i nomi delle funzioni e degli identificatori quando i programmi raggiungono una certa dimensione. Naturalmente, non si finiscono veramente i nomi, tuttavia diventa più difficile pensarne di nuovi dopo un po'. La cosa più importante, quando un programma raggiunge una certa dimensione è tipicamente dividere in pezzi, ognuno dei quali è costruito e manuntenuto da una persona diversa del gruppo. Poichè il C ha una singola area dove tutti i nomi degli identificatori e delle funzioni esistono, ciò significa che tutti gli sviluppatori devono fare attenzione a non usare gli stessi nomi in situazione dove ci possono essere conflitti. Ciò diventa noioso, fa perdere tempo e risulta costoso.

Lo Standard C++ possiede un meccanismo per prevenire questa collisione: la parola riservata **namespace**. Ogni insieme di definizioni C++ in una libreria o programma è

wrappato - "inglobato" in un namespace e se qualche altra definizione ha un nome identico, ma si trova in un namespace differente, allora non c'è collisione.

I namespace sono uno strumento utile e pratico, ma la loro presenza significa che si deve essere consci di essi prima che si scriva qualsiasi programma. Se si include semplicemente un header file e si usano delle funzioni o oggetti da quel header, probabilmente si avranno strani errori quando si cercherà di compilare il programma, fino al punto che il compilatore con troverà nessuna dichiarazione per quell'oggetto che si è appena incluso nel file header! Dopo che si è visto questo messaggio un po' di volte si diventerà familiari con il suo significato (che è "si è incluso l'header file ma tutte le dichiarazione sono all'interno di un namespace e non si è detto al compilatore che si vuole usare le dichiarazione in quel namespace").

C'è una parola riservata che permette di dire che si vogliono usare le dichiarazione e/o le definizione di un namespace. La parola riservata, abbastanza appropriata, è **using**. Tutte le librerie standard del C++ sono wrappate in un singolo namespace, che è **std** (sta per standard). Poichè questo libro usa quasi esclusivamente librerie standard, si vedrà la direttiva *using* quasi in ogni programma:

```
using namespace std;
```

Ciò significa che si vuole esporre tutti gli elementi da un namespace chiamato **std**. Dopo questa dichiarazione, non ci si deve preoccupare che un particolare componente della libreria sia dentro un namespace, poichè la direttiva **using** rende quel namespace disponibile a tutto il file dove la direttiva **using** è stata scritta.

Esporre tutti gli elementi di un namespace dopo che qualcuno ha avuto qualche problema per nasconderli può sembrare un po' controproducente ed infatti si dovrebbe fare attenzione nel fare ciò senza riflettere (come si imparerà più avanti nel libro). Tuttavia, la direttiva using espone solo quei nomi per il file corrente, quindi non è così drastico come può sembrare (ma si pensi due volte nel fare ciò in un header file, è imprudente).

C'è una relazione tra i namespace ed il modo in cui i file header vengono inclusi. Prima che fossero standardizzate le moderne inclusioni dei file header (senza i '**.h**', come in <**iostream**>), il tipico modo di includere un file header era con '**.h**', come in <**iostream.h**>. A quel tempo i namespace non facevano parte neanche del linguaggio. Quindi per fornire un compatibilità all'indietro con il codice esistente, se si scriveva

```
#include <iostream.h>
```

significava

```
#include <iostream>
using namespace std;
```

Tuttavia, in questo libro verrà usato il formalismo standard dell'include (senza il '**.h**') e quindi la direttiva **using** deve essere esplicita.

Per ora, questo è tutto ciò che si deve conoscere sui namespace, ma nel capitolo 10 l'argomento è trattato in modo esauriente.

Fondamenti della struttura del programma

Un programma in C o C++ è un insieme di variabili, definizioni di funzione e chiamate a funzioni. Quando inizia il programma, esso esegue codice di inizializzazione e chiama una speciale funzione, **main()**. Qui si mette il codice principale del programma.

```
int funzione() {
  // il codice della funzione va qui(questo è un commento!)
}
```

La funzione di sopra ha una lista di argomenti vuota ed il corpo contiene solo un commento.

Ci possono essere molti insiemi di parentesi all'interno di una definizione di parentesi, ma ci deve sempre essere almeno una che circonda il corpo della funzione. Poichè **main()** è una funzione, segue le stesse regole. In C++, **main()** restituisce sempre un **int**.

Il C e C++ sono linguaggi liberi da forma. Con poche eccezioni, il compilatore ignora newline e spazi bianchi, quindi ci deve essere un modo per determinare la fine di una istruzione. I comandi sono delimitati da punto e virgola.

I commenti del C iniziano con /* e finiscono con */. Possono includere newline (a capo). Il C++ usa i commenti del C ed ha un tipo di commento in più: //, che incomincia un commento che termina con un newline. Conviene di più di /**/ per un commento di una linea ed è molto usato in questo libro.

"Ciao, mondo!"

Ed ora finalmente il primo programma:

```
//: C02:Hello.cpp
// dire Ciao con il C++
#include <iostream> // dichiarazioni Stream
using namespace std;

int main() {
  cout << "Ciao, Mondo! Ho "
       << 8 << " anni oggi!" << endl;
} ///:~
```

All'oggetto **cout** si passano una serie di argomenti con l'operatore "<<". Esso stampa questi argomenti nell'ordine da sinistra verso destra. La speciale funzione iostream **endl** visualizza la linea ed un fine linea. Con la iostream si possono unire insieme una serie di argomenti come questo, che rende la classe semplice da usare.

In C, il testo dentro le doppie virgolette è tradizionalmente detto una stringa. Tuttavia, la libreria standard C++ ha una potente classe detta **string** per manipolare il testo e perciò si userà il termine più preciso *array di caratteri* per il testo dentro le doppie virgolette.

Il compilatore crea lo spazio per gli array di caratteri e memorizza l'equivalente ASCII per ogni carattere. Il compilatore automaticamente accoda a quest' array di caratteri un pezzo extra di memoria contenente il valore 0 che indica la fine dell'array di caratteri.

Dentro un array di caratteri, si possono inserire caratteri speciali usando le sequenze di escape. Esse consistono in un backslash(\) seguito da un codice speciale. Per esempio **\n** significa newline. Il manuale del compilatore o la guida del C fornisce un insieme completo di sequenze di escape; altre sono **\t** (tab),\\(backslash) e **\b** (backslash).

Si noti che le istruzioni possono continuare su linee multiple e che l'intera istruzione termina con un punto e virgola.

Gli argomenti di array di caratteri e i numeri costanti sono miscelati insieme nella precedente istruzione **cout**. Poichè l'operatore << è sovraccaricato in molti modi quando viene usato con **cout**, se possono mandare a **cout** argomenti diversi ed esso capirà cosa fare con quel messaggio.

In tutto il libro si noterà che la prima linea di ogni file è un commento che inizia con i caratteri con cui inizia un commento (tipicamente //=, seguiti da due punti e l'ultima linea del listato finisce con un commento seguito da "**/:~**". Questa è una tecnica che uso per estrarre facilmente informazioni dai file di codice (il programma per fare ciò può essere trovato nel volume 2 di questo libro, su *www.BruceEckel.com*). La prima linea contiene anche il nome e la locazione del file, quindi ci si può riferire ad esse nel testo o in altri file e facilmente localizzarlo nel codice sorgente di questo libro.

Lanciare il compiler

Dopo aver scaricato e scompattato il codice sorgente del libro, si trovi il programma nella sottocartella **CO2**. Si invochi il compiler con **Hello.cpp** come argomento. Con programmi semplici di un file solo come questo, il compilatore compierà tutto il processo. Per esempio, per usare il compilatore C++ GNU (che è disponibile liberamente su Internet), si scriva:

```
g++ Hello.cpp
```

Altri compilatori avranno una sintassi simile, si consulti la documentazione del proprio compilatore per i dettagli.

Ancora riguardo le iostreams

Finora abbiamo visto solo l'aspetto più rudimentale della classe iostreams. La formattazione dell'output disponibile con iostreams include anche funzionalità come la formattazione numerica in decimale, ottale ed esadecimale. Qui c'è un altro esempio di uso di iostreams:

```
//: C02:Stream2.cpp
// altre caratteristiche di iostream
#include <iostream>
using namespace std;

int main() {
  // Specificare i formati con i manipulators:
  cout << "a numero in decimale: "
       << dec << 15 << endl;
  cout << "in ottale: " << oct << 15 << endl;
  cout << "in esadecimale: " << hex << 15 << endl;
```

```
  cout << "un numero in virgola mobile: "
       << 3.14159 << endl;
  cout << "carattere non stampabile (escape): "
       << char(27) << endl;
} ///:~
```

Questo esempio mostra la classe iostreams che stampa i numeri in decimale, ottale ed esadecimale usando i *manipulators* (manipolatori, che non stampano niente, ma cambiano lo stato dell' output stream). La formattazione dei numeri in virgola mobile è determinata automaticamente dal compilatore. In aggiunta, qualsiasi carattere può essere mandato ad un oggetto stream usando un cast ad un **char** (un **char** è un tipo di dato che memorizza un singolo carattere). Questo cast assomiglia ad una chiamata a funzione: **char()**, insieme al valore ASCII del carattere. Nel programma sopra, **char(27)** manda un "escape" a **cout**.

Concatenazione di array di caratteri

Un'importante caratteristica del preprocessore C è la *concatenazione degli array di caratteri*. Questa caratteristica è usata in qualche esempio di questo libro. Se due caratteri quotati sono adiacenti e non c'è punteggiature tra di loro, il compilatore incollerà gli array insieme in un singolo array di caratteri. Ciò è particolarmente utile quando i listati di codice hanno restrizioni sulla larghezza:

```
//: C02:Concat.cpp
// concatenzione di array di caratteri
#include <iostream>
using namespace std;

int main() {
  cout << " Questa è troppo lunga per essere messa su una "
                " singola linea ma può essere spezzata "
          " senza nessun problema\nfinchè non si usa "
          " punteggiatura per separare array di caratteri "
                " adiacenti.\n";
} ///:~
```

A prima vista, il codice di sopra può sembrare errato poichè non c'è il familiare punto e virgola alla fine di ogni linea. Si ricordi che il C ed il C++ sono linguaggi liberi da forma e sebbene si vede solitamente un punto e virgola alla fine di ogni linea, è richiesto un punto e virgola alla fine di ogni istruzione ed è possibile che un' istruzione continui su più linee.

Leggere l'input

Le classi iostreams forniscono la capacità di leggere l'input. L'oggetto usato per lo standard input è **cin** (che sta per console input). **cin** normalmente si aspetta un input dalla console, ma questo input può essere deviato su altre sorgenti. Un esempio di redirezione verrà mostrato più avanti in questo capitolo.

L'operatore delle iostreams usato con **cin** è >>. Questo operatore si aspetta lo stesso genere di input come suo argomento. Per esempio, se si dà un argomento intero, esso si aspetta un intero dalla console. Ecco un esempio:

```
//: C02:Numconv.cpp
// Converte un decimale a ottale e esadecimale
#include <iostream>
```

```
using namespace std;

int main() {
  int numero;
  cout << "Inserisci un numero decimale: ";
  cin >> numero;
  cout << "valore in ottale = 0"
       << oct << numero << endl;
  cout << "valore in esadecimale = 0x"
       << hex << numero << endl;
} ///:~
```

Questo programma converte un numero inserito dall'utente in rappresentazione ottale ed esadecimale.

Chiamare altri programmi

Mentre il tipico modo di usare un programma che legge dallo standard input e scrive sullo standard output è interno ad uno script shell Unix o un file batch Dos, qualsiasi programma può essere chiamato da un programma C o C++ usando la funzione standard C **system()**, che viene dichiarata nel file header **<cstdlib>**:

```
//: C02:CallHello.cpp
// Chiamare un altro programma
#include <cstdlib> // dichiara "system()"
using namespace std;

int main() {
  system("Ciao");
} ///:~
```

Per usare la funzione **system()**, si da un array di caratteri che normalmente si digiterebbe al prompt dei comandi del sistema operativo. Ciò può anche includere gli argomenti della linea di comando e l'array di caratteri può essere creato a run time (invece di usare solo un array statico come mostrato sopra). Il comando viene eseguito ed il controllo ritorna al programma.

Questo programma ci mostra come sia semplice usare una libreria di funzioni C in C++, si deve solo includere l'header file e chiamare la funzione. Questa compatibilità verso l'alto dal C al C++ è un grosso vantaggio se si sta imparando il linguaggio partendo dal C.

Introduzione a strings

Sebbene un array di caratteri può essere molto utile, è anche limitato. È semplicemente un gruppo di caratteri in memoria, ma se si vuole fare qualcosa con essi si devono gestire tutti i piccoli dettagli. Per esempio, la dimensione di un array di caratteri quotati è fissa a tempo di compilazione. Se si ha un array di caratteri e si vuole aggiungere qualche carattere ad esso, si deve diventare un po' esperti (inclusa la gestione della memoria dinamica, copia di array di caratteri e concatenazione) prima di ottenere ciò che si vuole. Ciò è esattamente il genere di cose che vorremmo un oggetto facesse per noi.

La classe **string** del C++ Standard è stata progettata per occuparsi (e nascondere) tutte le manipolazioni a basso livello degli array di caratteri che erano richieste precedentemente al programmatore C. Queste manipolazioni sono state una costante motivo di spreco di

tempo e sorgente di errori dall'inizio del linguaggio C. Quindi, sebbene un intero capitolo è dedicato alla classe **string** nel Volume 2 di questo libro, la **string** è così importante (rende la vita molto più facile) che verrà introdotta qui ed usata molto nella prima parte del libro.

Per usare string si deve includere l'header file **<string>** del C++. La classe **string** è nel namespace **std** quindi è necessaria una direttiva **using**. Grazie all' overloading dell'operatore, la sintassi per l'uso delle **string** è abbastanza intuitiva:

```
//: C02:HelloStrings.cpp
// Le basi della classe string del C++ Standard
#include <string>
#include <iostream>
using namespace std;

int main() {
  string s1, s2; // stringhe vuote
  string s3 = "Ciao, Mondo."; // inizializzazione
  string s4("Io sono "); // altra inizializzazione
  s2 = "Oggi"; // assegnamento
  s1 = s3 + " " + s4; // somma
  s1 += " 8 "; // concatenazione
  cout << s1 + s2 + "!" << endl;
} ///:~
```

Le prime due **string**, **s1** e **s2**, sono vuote, mentre **s3** e **s4** mostrano due modi equivalenti di inizializzare le stringhe con array di caratteri (si possono inizializzare **string** da altre stringhe).

Si può assegnare alla stringa usando "=". Ciò rimpiazza il contenuto precedente della stringa con qualunque cosa è sulla parte destra e non ci si deve preoccupare di ciò che accade al contenuto precedente, che è gestito automaticamente. Per unire stringhe si usa l'operatore "+", che permette anche di unire array di caratteri con stringhe. Se si vuole concatenare sia una stringa che un array di caratteri ad un'altra stringa, si può usare "+". Infine, si noti che iostreams già sa cosa fare con le stringhe, quindi si può mandare una stringa (o un'espressione che produce una stringa, che si ottiene da s1+s2+"!") direttamente a **cout** per stamparla.

Leggere e scrivere i file

In C, aprire e manipolare file richiede molta esperienza con il linguaggio data la complessità delle operazione. Tuttavia, la libreria del C++ iostream fornisce un modo semplice di manipolare file e quindi questa funzionalità può essere introdotta più semplicemente di come sarebbe in C.

Per aprire file in lettura e scrittua, si deve includere **<fstream>**. Sebbene ciò includerà automaticamente **<iostream>**, generalmente è più prudente includere esplicitamente **<iostream>** se si prevede di usare **cin**,**cout**, ecc...

Per aprire un file in lettura, si crea un oggetto **ifstream**, che si comporta come **cin**. Per aprire un file in scrittura, si crea un oggetto **ofstream**, che si comporta come **cout**. Una volta che si è aperto il file, si legge da esso o si scrive come si farebbe con qualsiasi altro oggetto iostream. È semplice.

Una delle funzioni più semplici della libreria iostream è **getline()**, che permette di leggere una linea (terminata da newline) in una stringa[28]. Il primo argomento è l'oggetto **ifstream** da cui si sta leggendo ed il secondo argomento è l'oggetto **string**. Quando la chiamata a funzione è terminata, l'oggetto **string** conterrà la linea.

Ecco un semplice esempio, che copia il contenuto di un file in un altro:

```
//: C02:Scopy.cpp
// Copia un file su un altro, una linea alla volta
#include <string>
#include <fstream>
using namespace std;

int main() {
  ifstream in("Scopy.cpp"); // Apre in lettura
  ofstream out("Scopy2.cpp"); // Apre in scrittura
  string s;
  while(getline(in, s)) // Elimina il carattere newline
    out << s << "\n"; // ... lo riaggiunge
} ///:~
```

Per aprire i file, si passano agli oggetti **ifstream** e **ofstream** i nomi dei file che si vogliono creare, come sopra.

Qui viene introdotto un nuovo concetto, che è il ciclo **while**. Sebbene questo verrà spiegato in dettaglio nel prossimo capitolo, l'idea base è che l'espressione tra parentesi che seguono il **while** controlla l'esecuzione di un istruzione susseguente (che possono anche essere istruzioni multiple, delimitate da parentesi graffe). Finchè l'espressione tra parentesi (in questo caso, **getline(in,s)**) vale true, le istruzioni controllate dal **while** continueranno ad essere eseguite. Ne segue che **getline()** restituirà un valore che può essere interpretato come true se un'altra linea è stata letta con successo e false se non ci sono più linee da leggere. Quindi, il ciclo **while** di sopra legge tutte le linee dal file di input e manda ogni linea al file di output.

getline() legge caratteri finchè non scopre un newline (un carattere di terminazione che può essere cambiato, ma questo argomento verrà trattato nel capitolo di iostream nel Volume 2). Tuttavia, elimina il newline e non lo memorizza nella stringa risultante. Quindi, se vogliamo che il file copiato sia identico al file sorgente, dobbiamo aggiungere un newline, come mostrato.

Un esempio interessante è copiare l'intero file in una singola stringa.

```
//: C02:FillString.cpp
// Legge un intero file e lo copia in una singola stringa
#include <string>
#include <iostream>
#include <fstream>
using namespace std;

int main() {
  ifstream in("FillString.cpp");
  string s, linea;
  while(getline(in, linea))
    s += linea + "\n";
  cout << s;
} ///:~
```

Grazie alla natura dinamica delle stringhe, non ci si deve preoccupare di quanta memoria serve per allocare una stringa, la stringa si espanderà automaticamente qualsiasi cosa ci si metta dentro.

Una delle cose utili mettendo un intero file in una stringa è che la classe **string** ha molte funzioni per cercare e manipolare che permettono di modificare il file come una singola stringa. Tuttavia ciò ha delle limitazioni. Per prima cosa, è spesso conveniente trattare un file come un insieme di linee invece di un enorme blocco di testo. Per esempio, se si vuole aggiungere la numerazione di linea è molto più facile se si ha ogni linea come un oggetto stringa separato. Per ottenere ciò, abbiamo bisogno di un altro approccio.

Introduzione a vector

Con le stringhe, si può espandere una stringa senza sapere di quanta memoria si avrà bisogno. Il problema della lettura delle linee da un file in oggetti stringa individuali è che non si conosce il numero di stringhe di cui si ha bisogno, lo si conosce solo dopo che si è letto l'intero file. Per risolvere questo problema, abbiamo bisogno di una specie di contenitore che si espanda automaticamente per contenere tante stringhe quante se ne hanno bisogno.

Infatti, perchè limitarci a gestire stringhe? Ne consegue che questo tipo di problema " non sapere quanto si ha di qualcosa mentre si scrive un programma" capita spesso. E questo oggetto "contenitore" sarebbe molto più utile se potesse gestire ogni tipo di oggetto! Fortunatamente, la libreria Standard C++ ha una soluzione bella e pronta : le classi container standard. Esse sono une delle vere potenzialità del C++ Standard.

C'è spesso un po' di confusione tra i container e algorithm nella libreria Standard C++ e l'entità nota come STL. Standard Template Library è il nome usato da Alex Stepanox (che a quel tempo lavorava alla Hewlett-Packard) quando presentò la sua libreria alla commissione per gli Standard del C++ in un meeting a San Diego, California nella primavera del 1994. Il nome colpì, specialmente dopo che HP decise di renderla pubblica. Nel frattempo, la commissione la integrò nella libreria Standard C++, facendo un gran numero di cambiamenti. Lo sviluppo della STL continua alla Silicon Graphics (SGI; see *http://www.sgi.com/Technology/STL*). La STL SGI diverge dalla libreria Standard C++ su molti punti sottili. Quindi sebbene sia un equivoco popolare, il C++ Standard non include la STL. Può confondere un po' poichè i container e algorithm nella libreria Standard C++ hanno la stessa root(e di solito gli stessi nomi) come la SGI STL. In questo libro, si scriverà "la libreria Standard C++" ed i "container della libreria standard" oppure qualcosa di simile e si eviterà il termine STL.

Anche se l'implementazione dei container e algorithm della libreria standard C++ utilizza concetti avanzati e la spiegazione completa si estende su due grossi capitoli del Volume 2 di questo libro, questa libreria può essere molto potente anche senza conoscerne molto di essa. È così utile che il più semplice dei container standard, il **vector**, viene presentato in questo capitolo ed utilizzato lungo tutto il libro. Si può fare tantissimo solo usando **vector** e non preoccupandosi della sua implementazione(di nuovo, una importante meta della OOP). Poichè si imparerà molto di più di esso e degli altri container quando si leggerano i relativi capitoli del Volume 2, mi perdonerete se i programmi che usano **vector** nella prima parte del libro non sono esattamente ciò che un programmatore C++ con esperienza farebbe. In molti casi l'uso mostrato qui sarà adeguato.

La classe vector è un *template*, che significa che può essere applicato efficientemente a tipi diversi. Cioè possiamo creare un **vector** di **Figure**, un **vector** di **Gatti**, un **vector** di **string**,ecc... Fondamentalmente , con un template si crea una "classe di niente". Per dire al compilatore con che classe lavorerà (in questo caso, cosa **vector** conterrà), si mette il nome del tipo desiderato tra < > . Quindi un **vector** di string sarà denotato con **vector<string>**. Quando si fa ciò, si otterrà un **vector** che contiene solo oggetti **string** e si avrà un messaggio di errore da compilatore se si cerca di metterci qualcos'altro.

Poichè **vector** esprime il concetto di un container, ci deve essere un modo di mettere cose nel container e un modo per riottenerle indietro. Per aggiungere un elemento alla fine di un **vector**, si usa la funzione membro **push_back()** (si ricordi che, poichè è una funzione membro si usa un "." per chiamarlo con un oggetto particolare). La ragione del nome di questa funzione membro può sembrare un po' verbosa, **push_back()** invece semplicemente come "metti", poichè ci sono altri container e altre funzioni membro per mettere nuovi elementi nei container. Per esempio, c'è un **insert()** per mettere qualcosa nel mezzo di un container, **vector** lo supporta ma il suo uso è più complicato e lo esploreremo nel Volume 2 del libro. C'è anche un **push_front()** (non fa parte di **vector**) per mettere cose all'inizio. Ci sono molte funzioni membro in **vector** e molti container nella libreria Standard C++, ma si resterà sorpresi di quanto si può fare con essi conoscendo solo poche caratteristiche.

Quindi si possono mettere nuovi elementi in un **vector** con **push_back()**, ma come si riottengono indietro? La soluzione è ancora più intelligente ed elegante, è utilizzato l'overloading dell'operatore per far sembrare **vector** come un array. L'array (che sarà descritto nel prossimo capitolo) è un tipo di dato che è disponibile virtualmente in ogni linguaggio di programmazione quindi si è familiari con esso. Gli array sono *aggregati*, che significa che essi consistono in un numero di elementi ammucchiati insieme. La caratteristica distintiva di un array è che questi elementi hanno la stessa dimensione e sono posizionati uno dopo l'altro. La cosa più importante è che questi elementi possono essere selezionati con un indice, che significa che si può dire "voglio l'elemento n-simo e quel elemento sarà ottenuto". Sebbene ci sono eccezioni nei linguaggi di programmazione, l'indicizzazione è realizzata usando le parentesi quadre, quindi se si ha un array **a** e si vuole il quinto elemento, si scrive **a[4]** (si noti che l'indice parte da zero).

Questa notazione molto compatta e potente è incorporata in **vector** usando l'overloading dell'operatore, proprio come "<<" e ">>" sono state incorporate nelle iostream. Di nuovo, non c'è bisogno di sapere come è stato implementato l'overloading, ma è utile sapere che c'è un po' di magia per far funzionare **[ ]** con **vector**.

Con quanto detto in mente, si può ora vedere un programma che usa **vector**. Per usare un **vector**, si include l'header file **<vector>**:

```
//: C02:Fillvector.cpp
// Copia un intero file in un vector di stringhe
#include <string>
#include <iostream>
#include <fstream>
#include <vector>
using namespace std;

int main() {
  vector<string> v;
  ifstream in("Fillvector.cpp");
  string linea;
```

```
  while(getline(in, linea))
    v.push_back(linea); // Aggiunge la linea alla fine
// Aggiunge i numeri di linea:
for(int i = 0; i < v.size(); i++)
    cout << i << ": " << v[i] << endl;
} ///:~
```

Molto di questo programma è simile al precedente; un file viene aperto e le linee vengono lette e messe in un oggetto **string** una alla volta. Tuttavia, questi oggetti stringa vengono inseriti in un **vector v**. Una volta che il ciclo **while** è completo, l'intero file risiede in memoria dentro **v**.

La prossima istruzione nel programma è detta un ciclo **for**. È simile ad un ciclo **while** tranne che aggiunge qualche simbolo in più. Dopo il **for**, c'è una "espressione di controllo" dentro le parentesi, proprio come il ciclo **while**. Tuttavia, questa espressione di controllo è composta di tre parti: una parte che inizializza, una che testa per vedere se si può uscire dal ciclo ed una che cambia qualcosa, tipicamente per spostarsi su una sequenza di elementi. Questo programma mostra il ciclo for nel modo in cui è più comunamente usato: la parte di inizializzazione **int i = 0** crea un intero **i** per usarlo come contatore del ciclo e gli assegna il valore iniziale zero. La parte di test dice che per rimanere nel ciclo, **i** deve essere minore del numero di elementi del **vector v** (si usa size(), che è stato buttato nel codice, ma si deve ammettere che ha un significato abbastanza chiaro). La porzione finale usa l'operatore auto-incremento per aggiungere uno al valore di **i**. **i++** significa: "prendi il valore di **i**, aggiungi uno ad esso e rimetti il risultato di nuovo in **i**". Quindi il risultato globale del ciclo **for** è di prendere una variabile **i** e passare dal valore zero ad un valore minore della dimensione di vector. Per ogni valore di **i**, l'istruzione **cout** viene eseguita e scrive una linea che consiste nel valore di **i** (magicamente convertito in un array di caratteri da **cout**), due punti ed uno spazio, la linea letta dal file ed un newline con **endl**. Quando si compila e si lancia, si vede che l'effetto è l'aggiunta dei numeri di linea al file.

Grazie al modo in cui funziona l'operatore ">>" con iostream, si può facilmente modificare il programma di sopra in modo che spezzi l'input in parole separate da spazi bianchi invece di linee:

```
//: C02:GetWords.cpp
// Spezza un file in parole separate da spazi bianchi
#include <string>
#include <iostream>
#include <fstream>
#include <vector>
using namespace std;

int main() {
  vector<string> parole;
  ifstream in("GetWords.cpp");
  string parola;
  while(in >> parola)
    parole.push_back(parola);
  for(int i = 0; i < parole.size(); i++)
    cout << parole[i] << endl;
} ///:~
```

L'espressione

```
while(in >> parola)
```

è ciò che ottiene in input una parola alla volta e quando questa espressione vale false significa che è stata raggiunta la fine del file. Naturalmente, delimitare le parole con spazi bianchi è abbastanza grezzo, ma è un esempio semplice. Più in avanti nel libro si vedranno esempi più sofisticati che permetteranno di spezzare l'input nel modo che si vuole.

Per dimostrare come sia facile usare un **vector** con qualsiasi tipo, ecco un esempio che crea un **vector<int>** :

```
//: C02:Intvector.cpp
// Creazione di un vector che contiene interi
#include <iostream>
#include <vector>
using namespace std;

int main() {
  vector<int> v;
  for(int i = 0; i < 10; i++)
    v.push_back(i);
  for(int i = 0; i < v.size(); i++)
    cout << v[i] << ", ";
  cout << endl;
  for(int i = 0; i < v.size(); i++)
    v[i] = v[i] * 10; // assegnazione
  for(int i = 0; i < v.size(); i++)
    cout << v[i] << ", ";
  cout << endl;
} ///:~
```

Per creare un **vector** che gestisce un tipo diverso, si usa quel tipo come un argomento di template (l'argomento nelle < >). I template e le librerie di template ben progettate sono intese per essere usate con facilità.

Quest'esempio continua per dimostrare un'altra caratteristica essenziale di **vector**. Nell'espressione

```
v[i] = v[i] * 10;
```

si può vedere che il **vector** non è usato solo per mettere cose in esso e riprenderle. Si può anche assegnare a qualsiasi elemento di un **vector**, usando anche l'operatore di indicizzazione delle parentesi quadre. Ciò significa che **vector** è progettato per scopi generali ed è molto flessibile per lavorare con insiemi di oggetti.

Sommario

L'intento di questo capitolo è di mostrare come possa essere semplice la programmazione orientata agli oggetti, *se* qualcun altro ha fatto il lavoro di definire gli oggetti per noi. In quel caso, si include un header file, si creano gli oggetti e si mandano messaggi ad essi. Se i tipi che si stanno usando sono potenti e ben progettati, allora non si avrà molto da lavorare ed il programma risultante sarà anch'esso potente.

Nel mostrare la facilità di uso delle classi di librerie con la OOP, questo capitolo presenta anche la famiglia delle **iostream** (in particolare, quelle che leggono e scrivono da e verso console e file), la classe **string** ed il template **vector**. Si è visto come sia semplice usarle e

si possono immaginare tante cose che ora si possono realizzare, ma c'è ancora molto di cui esse sono capaci[29]. Anche quando si utilizza un insieme limitato di funzionalità di questi strumenti nella prima parte del libro, esse forniscono comunque un grosso passo avanti rispetto all'apprendimento di un linguaggio come il C e sebbene apprendere gli aspetti a basso livello del C sia educativo, c'è anche un gran dispendio di tempo. Quindi si sarà molto più produttivi se si hanno oggetti che gestiscono gli aspetti dei livelli bassi. Dopo tutto, il punto cruciale della OOP è nascondere i dettagli in modo che si possa dipingere con un pennello più grande.

Tuttavia, per come l'OOP cerca di essere ad alto livello, ci sono aspetti fondamentali del C che non si possono non conoscere e questi saranno illustrati nel prossimo capitolo.

---

[25] I confini tra compilatori ed interpreti tendono a diventare sfumati, specialmente con Python, che ha molte caratteristiche e potenzialità dei linguaggi compilati ma l'immediatezza di un linguaggio interpretato.

[26] Python è di nuovo un eccezione, poichè fornisce compilazione separata.

[27] Raccomanderei di usare Perl o Python per automatizzare questo compito come parte del processo di preparazione delle librerie (si veda www.Perl.org oppure www.Python.org).

[28] Ci sono diverse varianti di **getline( )**, che saranno discusse nel capitolo iostream nel Volume 2.

[29] Se si è particolarmente ansiosi di vedere tutte le cose che si possono fare con questi ed altri componenti della libreria Standard, si veda il Volume 2 di questo libro su www.BruceEckel.com ed anche www.dinkumware.com.

3: Il C nel C++

Poichè il C++ è basato sul C, si deve conoscere la sintassi del C al fine di programmare in C++, come si deve essere ragionevolmente spedito in algebra al fine di affrontare calcoli.

Se non si è mai visto prima il C, questo capitolo darà un decente background dello stile del C nel C++. Se si conosce lo stile del C descritto nella prima edizione di Kernighan & Ritchie (spesso chiamato K&R C), si troveranno nuove e differenti caratteristiche nel C++ in aggiunta al C standard. Se si conosce il C standard, si dovrebbe sfogliare questo capitolo guardando le caratteristiche che sono peculiari del C++. Si noti che ci sono alcune fondamentali caratteristiche del C++ introdotte qui, le quali sono idee base che sono affini alle caratteristiche del C o spesso modifiche ai modi in cui il C fa le cose. Le caratteristiche più sofisticate del C++ saranno introdotte negli ultimi capitoli.
Questo capitolo è una veloce panoramica dei costrutti del C ed un'introduzione ad alcuni costrutti base del C++, presumendo che si è avuto qualche esperienza di programmazione in un altro linguaggio. Un'introduzione al C si trova nel CD ROM confezionato in fondo al libro, intitolato *Thinking in C: Foundations for Java & C++* di Chuck Allison (pubblicata da MindView, Inc., e anche disponibile al sito www.MindView.net). Questo e' un seminario in CD ROM con lo scopo di dare accuratamente tutti i fondamenti del linguaggio C. Si focalizza sulla conoscenza necessaria per poter destreggiarsi nei linguaggi C++ o Java piuttosto che provare a formare esperto del C (una delle ragioni dell'uso del linguaggio alto come il C++ o il Java è precisamente il fatto che possiamo evitare molti i loro angoli buii). Esso contiene inoltre esercizi e soluzioni guidate. Si tenga a mente ciò perchè va oltre il CD *Thinking in C,* il CD non e' una sostituzione di questo capitolo, ma dovrebbe essere usato invece come una preparazione per il libro.

Creare funzioni

Nel vecchio (pre-Standard) C, si poteva chiamare una funzione con qualsiasi numero o tipi di argomento ed il compilatore non avrebbe reclamato. Ogni cosa sembrava giusta fino a quando non si faceva girare il programma. Si ottenevano risultati misteriosi (o peggio, il programma si bloccava) con nessun accenno ad un perchè. La mancanza di aiuto con il passaggio dell'argomento e gli enigmatici bug risultavano essere probabilmente una ragione per cui il C fu nominato "*linguaggio assemblante di alto livello* ". I programmatori del C Pre-Standard si adattarono ad esso.
Il C Standard ed il C++ usano una caratteristica chiamata *function prototyping (prototipazione della funzione)*. Con il function prototyping, si deve usare una descrizione dei tipi di argomenti quando si sta dichiarando e definendo una funzione. Questa descrizione è un "prototipo". Quando la funzione viene chiamata , il compilatore usa il prototipo per assicurarsi che gli opportuni argomenti siano passati e che il valore restituito sia elaborato correttamente. Se il programmatore fa un errore quando chiama la funzione, il compilatore lo trova.
Essenzialmente, si è appreso della function prototyping (non è stata chiamata in tal modo) nel capitolo precedente, poichè la forma della dichiarazione di funzione nel C++ richiede opportuni prototipi. In un prototipo di funzione, la lista degli argomenti contiene i tipi degli argomenti che devono essere passati alla funzione e (a scelta per la dichiarazione) identificatori per gli argomenti. L'ordine ed il tipo di argomenti devono corrispondere nella dichiarazione, definizione e nella chiamata alla funzione. Ecco qui un esempio di un prototipo di funzione in una dichiarazione:

```
int traduci(float x, float y, float z);
```

Non si usa la stessa forma quando si dicharano variabili nei prototipi delle funzioni come si

fa nelle definizioni ordinarie di variabili. Ciòè, non si può dire: **float x, y, z**. Si deve indicare il tipo di ogni argomento. In una dichiazione di funzione, la forma seguente è anche accettabile:

```
int traduci(float, float, float);
```

Poichè il compilatore non fa altro che controllare i tipi quando la funzione viene chiamata, gli identificatori sono soltanto inclusi per chiarezza di chi legge il codice.
Nella definizione di funzione, i nomi sono richiesti perchè viene fatto riferimento agli argomenti dentro la funzione:

```
int traduci(float x, float y, float z) {
  x = y = z;
  // ...
}
```

Ne deriva che questa regola si applica solo nel C. Nel C++, un argomento può essere senza nome nella lista degli argomenti della definizione di funzione. Poiché è senza nome, non lo si può usare nel corpo della funzione naturalmente. Gli argomenti senza nome sono concessi per dare al programmatore un modo per “riservare spazio nella lista degli argomenti”. Chiunque usa la funzione deve chiamare ancora la funzione con i propri argomenti. Comunque, la persona che crea la funzione può dopo usare l’argomento in futuro senza forzare modifiche al codice che chiama la funzione. L’opzione di ignorare un argomento nella lista è anche possibile se si lascia il nome dentro, ma si avrà un irritante messaggio di allarme circa il valore che non è usato ogni volta che si compila la funzione. L’allarme viene eliminato se si rimuove il nome.
Il C ed il C++ hanno altri due modi di dichiarare una lista di argomenti. Se si ha una lista di argomenti vuota, si può dichiararla come **func( )** in C++, la quale dice al compilatore che ci sono esattamente zero argomenti. Si dovrebbe essere consapevoli che ciò significa solamente una lista di argomenti vuota in C++. In C ciò significa " un indeterminato numero di argomenti" (il quale è un " buco" nel C poichè disabilita il controllo del tipo in quel caso). Sia in C che C++, la dichiarazione **func(void);** intende una lista di argomenti vuota. La parola chiave **void** intende "nulla" in questo caso (e può anche significare "nessun tipo" nel caso dei puntatori, come si vedrà in seguito in questo capitolo).
L'altra opzione per le liste di argomenti si usa quando non si sa quanti argomenti o che tipo di argomenti si avranno; ciò è chiamata una *lista di argomenti variabile*. Questa " lista di argomenti inaccertata" è rappresentata da ellissi (**...**). Definere una funzione con una lista di argomenti variabile è significativamente più complicato che definire una funzione regolare. Si può usare una lista di argomenti variabile per una funzione che ha una serie fissata di argomenti se (per alcune ragioni) si vuole disabilitare il controllo di errore del prototipo di funzione. A causa di ciò, si dovrebbe restringere il nostro uso della lista di controllo variabile al C ed evitarli nel C++ (nella quale, come si imparerà, ci sono alternative migliori). L'uso della lista di argomenti variabile è descritta nella sezione della libreria della nostra guida locale C.

Le funzioni restituiscono valori

Un prototipo di funzione in C++ deve specificare il tipo del valore di ritorno della funzione (in C, se si omette il tipo del valore di ritorno esso è per default un **int**). La specificazione del tipo di ritorno precede il nome della funzione. Per specificare che nessun valore viene restituito, si usa la parola chiave **void**. Ciò genererà un errore se si prova far ritornare un valore dalla funzione. Ecco qui alcuni prototipi completi di funzioni :

```
int f1(void); // restituisce un  int, non da argomenti
int f2(); // Come f1() in C++ ma non in Standard C!
float f3(float, int, char, double); // Restituisce un float
void f4(void); //  Non da argomenti, non restituisce nulla
```

Per far restituire un valore da una funzione, si usa la dichiarazione **return. return** fa ritornare l'esecuzione del programma esattamente dopo la chiamata alla funzione. Se **return** ha un argomento, quel argomento diventa un valore di ritorno della funzione. Se

la funzione dice che ritornerà un tipo particolare, dopo ogni dichiarazione **return** deve ritornare quel tipo. Si può avere più di una dichiarazione **return** in una definizione di funzione:

```
//: C03:Return.cpp
//: uso del "return"
#include <iostream>
using namespace std;
char cfunc(int i) {
  if(i == 0)
    return 'a';
  if(i == 1)
    return 'g';
  if(i == 5)
    return 'z';
  return 'c';
}
int main() {
  cout << "scrivi un intero: ";
  int val;
  cin >> val;
  cout << cfunc(val) << endl;
} ///:~
```

In **cfunc( )**, la prima **if** che valuta il **true** esce dalla funzione tramite la dichiarazione **return**. Notare che la dichiarazione di funzione non è necessaria perché la definizione di funzione appare prima che essa sia usata nel **main( )**, quindi il compilatore conosce la funzione da quella dichiarazione di funzione.

Utilizzo della libreria di funzioni del C

Tutte le funzioni nella libreria locale di funzioni del C sono disponibili mentre si programma in C++ . Si dovrebbe guardare attentamente alla libreria di funzioni prima di definire la propria funzione – c'è una buona possibilità che qualcuno abbia già risolto il problema per noi e probabilmente abbia dato ad esso molti più pensieri e debug.
Tuttavia una parola di prudenza: molti compilatori includono molte funzioni extra che rendono la vita anche più facile e sono allettanti da usare, ma non sono parte della libreria Standard del C. Se si è certi che non si vorrà mai spostare le applicazioni su un'altra piattaforma (e chi è certo di ciò), si vada avanti e si usino queste funzioni che rendono la vita più facile. Se si vuole che la nostra applicazione sia portabile, si dovrebbe limitare alle funzioni della libreria standard. Se si devono realizzare attività specifiche della piattaforma, si provi ad isolare quel codice in un punto così che esso possa essere cambiato facilmente quando la si sposta su un'altra piattaforma. In C++ le attività specifiche della piattaforma sono spesso incapsulate in una classe, che è la soluzione ideale.
La formula per usare una libreria di funzioni è la seguente: primo, trovare la funzione nella propria guida di riferimento per la programmazione (molte guide elencano nell'indice la funzione per categorie e per ordine alfabetico). La descrizione della funzione dovrebbe includere una sezione che dimostri la sintassi del codice. La cima di questa sezione di solito ha almeno un **#include**, mostrandoci il file principale contenente il prototipo della funzione. Si duplichi questa linea **#include** nel nostro file, in modo che la funzione è propriamente dichiarata. Adesso si può chiamare una funzione nello stesso modo in cui appare nella sezione della sintassi. Se si fa un errore, il compilatore lo scoprirà comparandolo la nostra chiamata alla funzione con il prototipo della funzione presente in cima e ci darà informazioni circa il nostro errore. Il linker cerca la libreria Standard per default, quindi tutto ciò che si ha bisogna fare è: includere il file principale e chiamare la funzione.

Creare le proprie librerie con il librarian.

Si possono mettere le proprie funzioni in una libreria. Ogni ambiente di sviluppo ha un

librarian che amministra gruppi di moduli oggetto. Ogni librarian ha i propri comandi, ma l'idea generale è questa: se si vuole creare una libreria, bisogna fare un file principale contenente prototipi di funzione per tutte le funzioni nella nostra libreria. Si mette questo file principale da qualche parte nella libreria di ricerca del preprocessore, ciascun nella directory locale (così può essere trovata con **#include "header"**) o in una directory include (così può essere trovata con **#include <header>**). Poi si prendono tutti i moduli oggetto e si passano al librarian insieme con un nome per la libreria completata (la maggior parte dei librarian richiedono una comune estensione, come **.lib** o **.a**). Si pone la libreria completata dove risiedono le altre librerie così il linker può trovarle. Quando si usa una propria libreria, si dovrà aggiungere qualcosa alla riga di comando così il linker sa cercare la libreria per la funzione che si chiama. Si devono trovare tutti i dettagli nel proprio manuale locale, poiché variano da sitema a sistema.

Controllare l'esecuzione

Questa sezione copre le istruzioni per il controllo dell' esecuzione in C++. Si devono conoscere queste dichiarazioni prima di poter leggere e scrivere il codice in C e C++.
Il C++ usa tutte le istruzioni per il controllo dell' esecuzione del C. Esse includono **if-else**, **while**, **do-while**, **for** ed una selezione di dichiarazioni chiamati **switch**. Il C++ ammette anche l'infame **goto**, il quale sarà evitato in questo libro.

True e false

Tutte le dichiarazioni condizionali usano la verità o la falsità di una espressione condizionale per determinare la traiettoria di esecuzione. Un esempio di una espressione condizionale è **A == B**. Questo usa l'operatore condizionale == per vedere se la variabile **A** è equivalente alla variabile B**.** L'espressione produce un Booleano **true** o **false**(queste sono parole chiave solo in C++; nel C una espressione è "vera" se valuta un valuta che non vale zero). Altri operatori condizionali sono >, <, >=, etc. Le dichiarazioni condizionali sono più pienamente trattate in seguito in questo capitolo.

if-else

La dichiarazione **if-else** può esistere in due forme: con o senza l' **else.** Le due forme sono:

```
if(espressione)
    istruzione
```

oppure

```
if(espressione)
       istruzione
else
    istruzione
```

La "espressione" valuta **true** o **false**. L' "istruzione" significa sia una semplice istruzioni terminate da un punto e virgola oppure un' istruzione composta, la quale è un gruppo di semplici istruzioni racchiuse in parentesi. In qualsiasi momento la parola " istruzione" sia usata, implica sempre che la dichiarazione è semplice o composta. Si noti che questa istruzione può essere un altro **if**.

```
//: C03:Ifthen.cpp
// dimostrazione delle condizioni if e if-else

#include <iostream>
using namespace std;
int main() {
  int i;
  cout << "scrivi un numero e 'Enter'" << endl;
  cin >> i;
  if(i > 5)
    cout << "è maggiore di 5" << endl;
  else
```

```
    if(i < 5)
      cout << "è minore di 5 " << endl;
    else
      cout << "è uguale a 5 " << endl;
  cout << "scrivi un numero e 'Enter'" << endl;
  cin >> i;
  if(i < 10)
    if(i > 5)  // "if" è soltanto un'altra istruzione
      cout << "5 < i < 10" << endl;
    else
      cout << "i <= 5" << endl;
 else // "if(i < 10)"
    cout << "i >= 10" << endl;
} ///:~
```

E' convenzione indentare il corpo delle istruzioni del flusso di controllo così che il lettore può facilmente determinare dove esso inizia e dove finisce[30].

while

while, **do-while** e **for** servono per il controllo del ciclo. Una istruzione è ripetuta fino a quando l'espressione di controllo vale **false.** La forma del ciclo **while** è

```
while(espressione)
    istruzione
```

L'espressione è valutata solamente una volta all'inizio del ciclo e ancora prima di ogni ulteriore iterazione dell' istruzione.
Questo esempio sta nel corpo del ciclo **while** fino a quando si scrive il numero segreto o si preme control-C.

```
//: C03:Guess.cpp
// indovina un numero ( dimostra "while")
#include <iostream>
using namespace std;
int main() {
  int secret = 15;
  int guess = 0;
  // "!=" è il condizionale "non-uguale":
 while(guess != secret) { // dichiarazione complessa
    cout << "indovina il numero: ";
    cin >> guess;
  }
  cout << "Hai indovinato !" << endl;
} ///:~
```

L'espressione condizionale di **while** non è ristretta ad un semplice test come nell'esempio su menzionato; può essere complicato quanto piaccia sempre che produce un risultato **true** o **false.** Si vedra anche il codice dove il ciclo non ha corpo, solamente un semplicemente punto e virgola:

```
while(/* Do a lot here */)
 ;
```

In questi casi, il programmatore ha scritto l' espressione condizionale non solo per effettuare il test ma anche per fare il lavoro.

do-while

La forma del **do-while** è

```
do
    istruzione
while(espressione);
```

Il **do-while** è differente dal **while** perchè l'istruzione è eseguita sempre almeno una volta, anche se l'espressione vale false in primo luogo. In un nomrale **while**, se il condizionale è falso nella prima volta l'istruzione non viene mai eseguita.
Se il **do-while** è usato in **Guess.cpp** , la variabile **guess** non ha bisogno di un valore

iniziale, poiché è inizializzato dall' istruzione **cin** prima che viene esaminata:

```
//: C03:Guess2.cpp
// il programma guess con do-while

#include <iostream>
using namespace std;
int main() {
  int secret = 15;
int guess; // Qui non serve nessuna inizializzazione
  do {
    cout << "Indovina il numero: ";
 cin >> guess; // Avviene l'inizializzazione

  }   while(guess != secret);
  cout << "You got it!" << endl;
} ///:~
```

Per alcune ragioni, la maggior parte dei programmatori tendono a evitare **do-while** e a lavorare solo con **while.**

for

Un ciclo **for** effettua un' inizializzazione prima della prima ripetizione. Dopo effettua la prova condizionale e, alla fine di ogni ripetizione, alcune forme di "progresso". La forma del ciclo **for** è:

```
for(inizializzazione; condizione; passo)
 istruzione
```

Qualsiasi delle espressioni di *inizializzazione,condizione* e *passo* possono essere vuote. Il codice di inizializzazione è eseguito una volta soltanto all'inizio. La condizione è eseguita prima di ogni ripetizione (se vale falso all'inizio, l'istruzione non è mai eseguita). Alla fine di ogni ciclo, viene eseguito il *passo*.
I cicli **for** sono di solito usati per il compito di "conteggio":

```
//: C03:Charlist.cpp
//Visualizza tutti i caratteri ASCII
// dimostra il "for"

#include <iostream>
using namespace std;
int main() {
  for(int i = 0; i < 128; i = i + 1)
 if (i != 26)  // ANSI Terminal Clear screen
      cout << " valore: " << i
           << " carattere: "
           << char(i) // tipo di conversione
           << endl;
} ///:~
```

Si può notare che la variabile **i** è definita nel punto in cui è usata, invece che all'inizio del blocco denotato dall'apertura della parentesi graffa '**{**'. Ciò è in contrasto con le procedure di linguaggio tradizionali(incluso C) i quali richiedono che tutte le variabili siano definite all'inizio del blocco. Ciò verrà discusso in seguito in questo capitolo.

Le parole chiave break e continue

Dentro il corpo di qualsiasi construtto dei cicli **while**, **do-while,** o **for** si può controllare il fluire del ciclo usando **break** e **continue**. **break** fa uscire dal ciclo senza eseguire il resto delle istruzioni nel ciclo. **continue** fa finire l'esecuzione della corrente iterazione e ritorna all'inizio del ciclo per iniziare una nuova iterazione.
Come esempio di **break** e **continue,** ecco questo programma che è un semplice menu di

sistema:

```
//: C03:Menu.cpp
// dimostrazione di un semplice menu di programma
// l' uso di "break" e "continue"
#include <iostream>
using namespace std;
int main() {
  char c; // per memorizzare il responso
  while(true) {
    cout << "MAIN MENU:" << endl;
    cout << "l: sinistra, r: destra, q: quit -> ";
    cin >> c;
    if(c == 'q')
      break; // fuori dal "while(1)"

    if(c == 'l') {
      cout << " MENU A SINISTRA:" << endl;
      cout << "scegli a oppure b: ";
      cin >> c;
      if(c == 'a') {
        cout << "hai scelto 'a'" << endl;
  continue; // ritorna al menu principale
      }
      if(c == 'b') {
        cout << "hai scelto 'b'" << endl;
  continue; // ritorna al menu principale
      }
      else {
        cout << "non hai scelto nè a nè b!"
             << endl;
  continue; // ritorna al menu principale
      }
    }
    if(c == 'r') {
      cout << " MENU A DESTRA:" << endl;
      cout << "scegli c oppure d: ";
      cin >> c;
      if(c == 'c') {
        cout << "hai scelto 'c'" << endl;
 continue; // ritorna al menu principale
      }
      if(c == 'd') {
        cout << "hai scelto 'd'" << endl;
 continue; // ritorna al menu principale
      }
      else {
        cout << "non hai scelto nè c nè d!"
             << endl;
  continue; // ritorna al menu principale
      }
    }
    cout << "devi digitare l oppure r oppure q!" << endl;
  }
  cout << "uscita dal menu..." << endl;
} ///:~
```

Se l'utente seleziona 'q' nel menu principale, il **break** è usato per uscire, altrimenti il programma continua ad eseguire indefinitamente. Dopo ogni selezione del sub-menu, la parola chiave **continue** è usata per fare un salto indietro all'inizio del ciclo while. L'istruzione **while(true)** è l'equivalente di " fai questo ciclo per sempre ". La dichiarazione **break** ci permette di interrompere questo infinito ciclo while quando l'utente scrive una 'q'.

switch

Un' istruzione **switch** seleziona tra pezzi di codice basati sul valore dell'espressione intero. La sua forma è:

```
switch(selezione) {
    case valore-intero1 : istruzione; break;
    case valore-intero2 : istruzione; break;
    case valore-intero3 : istruzione; break;
    case valore-intero4 : istruzione; break;
    case valore-intero5 : istruzione; break;
    (...)
    default: istruzione;
}
```

selezione è un'espressione che produce un valore intero. **Switch** confronta il risultato di selezione con ogni *valore intero*. Se ne trova uno simile, la corrispondente dichiarazione (semplice o composta) viene eseguita. Se non si presenta nessun simile, viene eseguita l'istruzione **default**.

Si noterà nelle definizioni summenzionate che ogni **case** si conclude con un **break**, il quale sposta l'esecuzione alla fine del corpo dello **switch** (la chiusura della parentesi che completa lo **switch**). Questo è il modo convenzionale per costruire una dichiarazione **switch**, ma il **break** è facoltativo. Se manca vengono eseguite le istruzione del seguente **case** fino a che non si incontra il **break**. Sebbene non si vuole usualmente questo tipo di comportamento, può essere utile per un programmatore esperto.

L'istruzione **switch** è un modo pulito per attuare una selezione multipla (es. selezionare differenti percorsi di esecuzione), ma richiede un selettore che valuta un valore intero a tempo di compilazione. Se si vuole usare, ad esempio, un oggetto **string** come selettore, non funziona in una dichiarazione di **switch**. Per un selettore **string**, si deve usare invece una serie di dichiarazioni **if** e confrontare la **string** dentro il condizionale.

L'esempio di menu mostrato sotto fornisce un buon particolare esempio di **switch**:

```
//: C03:Menu2.cpp
// un menu che usa uno switch
#include <iostream>
using namespace std;
int main() {
  bool quit = false;  // un flag per uscire
  while(quit == false) {
    cout << "Scegli a, b, c oppure q per terminare: ";
    char response;
    cin >> response;
    switch(response) {
      case 'a' : cout << "hai scelto 'a'" << endl;
                 break;
      case 'b' : cout << "hai scelto 'b'" << endl;
                 break;
      case 'c' : cout << "hai scelto 'c'" << endl;
                 break;
      case 'q' : cout << "fine" << endl;
                 quit = true;
                 break;
      default  : cout << "Scegliere a,b,c oppure q!"
                 << endl;
    }
  }
} ///:~
```

Il flag **quit** è un **bool**, abbreviazione di "Boolean", che è un tipo che si troverà soltanto in C++. Esso può assumere solo i valori chiave **true** o **false**. Selezionare 'q' setta il flag **quit** su **true**. La prossima volta che il selettore sarà valutato, **quit == false** ritorna **false** così che il corpo del **while** non è eseguito.

Uso e abuso di goto

La parola chiave **goto** è supportata in C++, poichè esiste anche in C. L'uso di **goto** è spesso respinto perchè appartiene uno stile mediocre di programmazione e la maggior parte delle volte è vero. Ogni volta che si usa **goto**, si guardi il proprio codice per vedere se c'è un altro modo per scriverlo. In rare occasioni, si può scoprire che **goto** può risolvere un problema che non può essere risolto in altro modo, ma tuttavia, lo si consideri con cura. Ecco qui un esempio che ne fa forse un possibile candidato:

```
//: C03:gotoKeyword.cpp
// L'infame goto è supportato nel C++
#include <iostream>
using namespace std;
int main() {
  long val = 0;
  for(int i = 1; i < 1000; i++) {
    for(int j = 1; j < 100; j += 10) {
      val = i * j;
      if(val > 47000)
        goto bottom;
//Break andrebbe messo solamente esternamente al 'for'
    }
  }
  bottom: // un'etichetta
  cout << val << endl;
} ///:~
```

L'alternativa sarebbe settare un Booleano che è testato nella parte più esterna del ciclo **for**, e dopo fare un **break** all'interno del ciclo. Comunque, se si hanno svariati livelli di **for** o **while** questo potrebbe essere scomodo.

Ricorsione

La ricorsione è un'interessante e qualche volta utile tecnica di programmazione con cui si può chiamare la funzione in cui si è. Naturalmente se questo è tutto quello che si fa, si continuerà a chiamare la funzione dove si è fino a quando non si esaurisce la memoria, quindi ci deve essere qualche modo di "raggiungere il livello più basso " della chiamata ricorsiva. Nel esempio seguente, questo "raggiungimento del livello più basso " è compiuto dicendo semplicente che la ricorsione andrà soltano fino a che il **cat** supera 'Z': :[31]

```
//: C03:CatsInHats.cpp
// semplice dimostrazione di ricorsione
#include <iostream>
using namespace std;
void removeHat(char cat) {
  for(char c = 'A'; c < cat; c++)
    cout << "  ";
  if(cat <= 'Z') {
    cout << "cat " << cat << endl;
  removeHat(cat + 1); // chiamata ricorsiva
  } else
    cout << "VOOM!!!" << endl;
}
int main() {
  removeHat('A');
} ///:~
```

In **removeHat( )**, si può vedere che finchè **cat** è minore di 'Z', **removeHat( )** verrà chiamata dentro **removeHat( ),** effettuando così la ricorsione. Ogni volta che **removeHat( )** è chiamata, il proprio argomento è l'unico più grande del corrente **cat** quindi l'argomento continua ad incrementare.

La ricorsione è spesso usata quando si valuta qualche tipo di problema complesso, poichè non si è ristretti ad una particolare "taglia" per la soluzione – la funzione può richiamarsi fino a che non si è raggiunta la fine del problema.

Introduzione agli operatori

Si può pensare che gli operatori come tipi speciali di funzioni (si apprenderà che il sovraccaricamento degli operatori del C++ tratta gli operatori precisamente in quel modo). Un operatore prende uno o più argomenti e produce un nuovo valore. Gli argomenti hanno differente forma rispetto alle ordinarie chiamate di funzioni, ma l'effetto è lo stesso.
Dalla propria precedente esperienza di programmazione, si dovrebbe aver abbastanza confidenza con gli operatori che sono stati usati. I concetti di addizione (+), sottrazione ed il segno meno (-), moltiplicazione (*), divisione (/), e assegnamento(=) hanno tutti essenzialmente lo stesso significato in ogni linguaggio di programmazione. L'intera serie di operatori è elencata in seguito in questo capitolo.

Precedenza

L'operatore precedenza definisce l'ordine nel quale è valutata un'espressione quando sono presenti svariati operatori. Il C ed il C++ hanno specifiche regole per determinare l'ordine di valutazione. Il più facile da ricordare è che la moltiplicazione e la divisione vengono prima dell'addizione e della sottrazione. Dopo ciò, se un'espressione non ci è trasparente probabilmente non lo sarà per tutti quelli che leggono il codice, quindi si dovrebbero usare le parentesi per rendere esplicito l'ordine di valutazione. Per esempio:

```
A = X + Y - 2/2 + Z;
```

ha un differente significato dalla stessa dichiarazione con un gruppo particolare di parentesi:

```
A = X + (Y - 2)/(2 + Z);
```

(si provi a valutare il risultato con X = 1, Y = 2, e Z = 3).

Auto incremento e decremento

Il C, e di conseguenza il C++, è pieno di scorciatoie. Le scorciatoie possono rendere il codice molto più facile da scrivere e qualche volta molto più diffice da leggere. Forse i creatori del linguaggio C pensarono che sarebbe stato più facile capire un pezzo difficile di codice se i nostri occhi non avevano da esaminare una larga area di testo.
Una della migliori scorciatoie sono gli operatori auto-incremento e auto-decremento. Si usano spesso per cambiare le variabili del ciclo, le quali controllano il numero di volte che un ciclo viene eseguito.
L'operatore auto-decremento è '**--**' e significa " decrementa di una unità ". L'operatore auto-incremento è '**++**' e significa " incrementa di una unità ". Se **A** è un **int**, ad esempio, l'espressione **++A** è equivalente a (**A = A + 1**). Gli operatori auto-incremento e auto-decremento producono il valore della variabile come risultato. Se l'operatore appare prima della variabile (es. **++A**), l'operazione è prima eseguita ed il valore risultante viene prodotto. Se l'operatore appare dopo la variabile (es. **A++**), il valore è prodotto corrente e l'operazione è eseguita successivamente. Ad esempio :

```
//: C03:AutoIncrement.cpp
// mostra l'uso degli operatori auto-incremento
//e auto-decremento
#include <iostream>
using namespace std;
int main() {
  int i = 0;
  int j = 0;
cout << ++i << endl; // Pre-incremento
  cout << j++ << endl; // Post-incremento
  cout << --i << endl; // Pre-decremento
  cout << j-- << endl; // Post decremento
} ///:~
  cout << ++i << endl; // Pre-increment
  cout << j++ << endl; // Post-increment
  cout << --i << endl; // Pre-decrement
  cout << j-- << endl; // Post decrement
```

```
} ///:~
```

Se ci si è chiesto perchè il nome “C++”, ora si capisce cosa significa: “ un passo oltre il C “.

Introduzione ai tipi di dato

I *Data types* (tipi di dato) definiscono il modo in cui si usa lo spazio (memoria) nei programmi che si scrivono. Specificando un tipo di dato, si dice al compilatore come creare un particolare pezzo di memoria e anche come manipularla.
I tipi di dato possono essere predefiniti o astratti. Un tipo di dato predefinito è uno che il compilatore intrinsecamente conosce, uno che è fissato nel compilatore. I tipi di dato predefiniti sono quasi identici in C e C++. Un tipo di dato definito da un utente è uno che noi o un altro programmatore crea come una classe. Queste sono comunemente detti tipi di dato astratti. Il compilatore sa come gestire tipi predefiniti quandoparte; esso “impara” come gestire i tipi di dato astratti leggendo i file principali contenenti le dichiarazioni di classi (si imparerà ciò nel prossimo capitolo).

Tipi base predefiniti

La specificazione dello standard C per i tipi predefiniti (che il C++ eredita) non dice quanti bit ogni tipo predefinito deve contenere. Invece, esso stipula che il minimo ed il massimo valore che il tipo predefinito deve avere. Quando una macchina è basata su binari, questo valore massimo può essere direttamente tradotto in un numero minimo di bit necessari a rappresentare quel valore. Tuttavia, se una macchina usa, ad esempio, il codice binario decimale (BCD) per rappresentare numeri, allora l’ammontare dello spazio richiesto nella macchina per ottenere il numero massimo per ogni tipo di data sarà differente. I valori di minimo e massimo che possono essere immagazzinati in vari tipi di data sono definiti nei file del sistema principale **limits.h** e **float.h** (in C++ si scriverà invece generalmente **#include <climits>** e **<cfloat>**).
Il C ed il C++ hanno quattro data tipi base predefiniti, descritti qui per le macchine basate sul sistema binario. Una **char** memorizza caratteri ed usa un minimo di 8 bit(1 byte), sebbene può essere più grande. Un **int** immagazzina un numero intero ed usa un minimo di due byte. I tipi **float** e **double** memorizzano numeri a virgola mobile, di solito nel formato virgola mobile IEEE. **Float** è per singola precisione virgola-mobile e **double** per virgola mobile doppia precisione .
Come menzionato in precedenza, si possono definire variabili in qualsiasi parte in uno scope e le si possono definire ed inizzializzare allo stesso tempo. Ecco qui come definire variabili usano i quattro tipi di dato base:

```
//: C03:Basic.cpp
//Definizione dei quattro tipi base di dato
//in C e C++
int main() {
//definizione senza inizializzazione
  char protein;
  int carbohydrates;
  float fiber;
  double fat;
 //definizione ed inizializzazione simultanea
  char pizza = 'A', pop = 'Z';
  int dongdings = 100, twinkles = 150,
    heehos = 200;
  float chocolate = 3.14159;
//Notazione esponenziale:
  double fudge_ripple = 6e-4;
} ///:~
```

La prima parte del programma definisce variabili dei quattro tipi di dato base senza inizializzarli. Se non si inizializza una variabile, lo standard dice che essi sono indefiniti (di solito, significa che contiene spazzatura). La seconda parte del programma definisce e

inizializza variabili allo stesso tempo (è sempre meglio, se possibile, procurare un valore di inizializzazione al punto di definizione). Si noti che l'uso della notazione esponenziale è nella costante 6e-4.

bool, true, & false

Prima che **bool** diventasse parte dello standard C++, ognuno tendeva ad usare differenti tecniche per simulare un Booleano. Si ottenevano problemi di portabilità e potevano introdurre subdoli errori.

Il tipo **bool** dello standard C++ può avere due stati espressi dalle costanti **true** (che è convertita all'intero 1) e quella **false** (che è convertita all'intero zero). Tutti e tre i nomi sono parole chiave. In più, alcuni elementi del linguaggio sono stati adattati.

Elemento	Uso con bool
&& || !	prende argomenti bool e produce risultati **bool**.
< > <= >= == !=	produce risultati **bool**.
if, **for**, **while**, **do**	Espressione convezionale che converte ai valori **bool**.
? :	Primo operando converte al valore **bool**.

Poichè ci sono molti codici esistenti che usano un **int** per rappresentare un flag, il compilatore implicitamente convertirà un **int** in un **bool** (il valore non zero produrrà **true** mentre quello zero produrrà **false**). Idealmente, il compilatore ci darà un allarme come suggerimento per correggere la situazione.

Un idioma che è classificato come: " stile povero di programmazione" è l'uso del ++ per settare la bandiera a true. Ciò è ancora permesso, ma deprecato, cioè in futuro potrebbe non esserlo. Il problema è che se si sta facendo una conversione di tipo implicita da **bool** a **int**, incrementando il valore (forse oltre la portata del valori normali di **bool** zero e uno) , e dopo implicitamente lo si riconverte ancora.

I puntatori (i quali verranno introdotti in seguito in questo capitolo) saranno automaticamente convertiti in **bool** quando sarà necessario.

Specificatori

Gli specificatori modificano i significati dei tipi basi incorporati e li espandono ad una serie abbastanza più larga. Ci sono quattro specificatori: **long**, **short**, **signed**, e **unsigned**. **Long** e **short** modificano i valori di massimo e minimo che il tipo di dato gestirà. Un semplice **int** deve essere come minimo la grandezza di uno **short**. La gerarchia della grandezza per un tipo intero è: **short int**, **int**, **long int.** Tutte le grandezze potrebbero essere le stesse, sempre che soddisfano le richieste dei valori di minimo e massimo. In una macchina con una word di 64-bit, per esempio, tutti i tipi di dato potrebbero essere a 64 bit.

La gerarchia di grandezza per i numeri virgola-mobile è : **float**, **double**, e **long double.** "long float" non è un tipo legale. Non ci sono numeri in virgola mobile **short**.

Gli specificatori **signed** e **unsigned** dicono al compilatore come usare il bit segno con i tipi interi ed i caratteri (i numeri in virgola mobile contengono un segno) un numero **unsigned** non tiene conto del segno e dunque ha un bit extra disponibile, quindi può salvare numeri positivi grandi due volte i numeri positivi che possono essere salvati in un numero **signed**. **signed** è per default ed è solo necessario con **char**; **char** può essere o non essere per default **signed**. Specificando **signed char**, si utilizza il bit di segno.

Il seguente esempio mostra come la grandezza del tipo di dato in bytes usando l'operatore **sizeof**, introdotto più avanti in questo capitolo:

```
//: C03:Specify.cpp
// Dimostra l'uso dei specificatori
#include <iostream>
using namespace std;
int main() {
  char c;
  unsigned char cu;
  int i;
  unsigned int iu;
  short int is;
 short iis; // come uno short int
  unsigned short int isu;
  unsigned short iisu;
  long int il;
  long iil;  // come long int
  unsigned long int ilu;
  unsigned long iilu;
  float f;
  double d;
  long double ld;
  cout
    << "\n char= " << sizeof(c)
    << "\n unsigned char = " << sizeof(cu)
    << "\n int = " << sizeof(i)
    << "\n unsigned int = " << sizeof(iu)
    << "\n short = " << sizeof(is)
    << "\n unsigned short = " << sizeof(isu)
    << "\n long = " << sizeof(il)
    << "\n unsigned long = " << sizeof(ilu)
    << "\n float = " << sizeof(f)
    << "\n double = " << sizeof(d)
    << "\n long double = " << sizeof(ld)
    << endl;
} ///:~
```

il risultato che si ottiene facendo girare il programma probabilmente sarà differente da macchina/sistema operativo/ compilatore, poichè (come menzionato precedentemente) l'unica cosa che deve essere consistente è che ogni differente tipo memorizza i valori minimi e massimi specificati nello standard.
Quando si modifica un **int** con **short** o **long**, la parola chiave **int** è facoltativa, come mostrato sopra.

Introduzione ai puntatori

Ogni volta che si fa girare un programma, viene prima caricato (tipicamente dal disco) nella memoria del computer. Quindi, tutti gli elementi del nostro programma sono allocati da qualche parte nella memoria. La memoria è tipicamente disposta come una serie sequenziale di locazioni di memoria; ci riferiamo usualmente a queste locazioni come bytes di 8 bit ma realmente la grandezza di ogni spazio dipende dall'architettura della particolare macchina e di solito è chiamata grandezza della word della macchina. Ogni spazio può essere unicamente distinto dagli altri spazi dal proprio indirizzo. Per i propositi di questa discussione, si dirà solamente che tutte queste macchine usano byte che hanno indirizzi sequenziali che cominciano da zero e proseguono in su fino alla fine della memoria che si ha nel proprio computer.
Poichè il proprio programma vive in memoria mentre sta è eseguito, ogni elemento del programma ha un indirizzo. Supponiamo di iniziare con un semplice programma:

```
//: C03:YourPets1.cpp
#include <iostream>
using namespace std;
int cane, gatto, uccello, peshe;
```

```
void f(int animale) {
  cout << "numero id animale: " << animale<< endl;
}
int main() {
  int i, j, k;
} ///:~
```

Ciascuno degli elementi in questo programma ha una locazione in memoria quando il programma viene eseguito. Anche la funzione occupa memoria. Come si vedrà, l'area di memoria dove l'elemento è posto dipende dal modo in cui si definisce esso da e quale elemento è.
C'è un operatore in C e C++ che ci dirà l'indirizzo di un elemento. Questo è l'operatore '**&**'. Tutto quello che si fa è precedere il nome dell'identificatore con '**&**' ed esso produrrà l'indirizzo di quell' identificatore. **YourPets1.cpp** può essere modificato per scrivere gli indirizzi di tutti questi elementi:

```
//: C03:YourPets2.cpp
#include <iostream>
using namespace std;
int cane, gatto, uccello, pesce;
void f(int animale) {
  cout << "numero id animale: " << animale<< endl;
}
int main() {
  int i, j, k;
  cout << "f(): " << (long)&f << endl;
  cout << "cane: " << (long)&cane<< endl;
  cout << "gatto: " << (long)&gatto<< endl;
  cout << "uccello: " << (long)&uccello<< endl;
  cout << "pesce: " << (long)&pesce<< endl;
  cout << "i: " << (long)&i << endl;
  cout << "j: " << (long)&j << endl;
  cout << "k: " << (long)&k << endl;
} ///:~
```

(long) è un *cast*. Significa: " Non trattare questo come se fosse un tipo normale, invece trattalo come un **long**". Il cast non è essenziale, ma se non vi fosse, gli indirizzi sarebbero scritti invece in esadecimale, così assegnare un **long** rende le cose un po' più leggibili.
Il risultato di questo programma varierà dipendendo al tuo computer, OS, e tutte le sorti di altri fattori, ma darà sempre alcuni interessanti comprensioni. Per un singolo eseguibile sul nostro computer, il risultato assomiglia a questo:
The results of this program will vary depending on your computer, OS, and all sorts of other factors, but it will always give you some interesting insights. For a single run on my computer, the results looked like this:

```
f(): 4198736
dog: 4323632
cat: 4323636
bird: 4323640
fish: 4323644
i: 6684160
j: 6684156
k: 6684152
```

si può vedere come le variabili che sono definite dentro il **main( )** sono in differenti aree delle variabili definite fuori dal **main( ),** si capirà il perchè qunado si imparerà di più il linguaggio. **f()** appare nella sua area; il codice è tipicamente separato dal dato in memoria. Un'altra cosa interessante da notare è che le variabili definite una dopo l'altra sono poste in memoria in modo contiguo. Esse sono separate da un numero di byte che sono richiesti dai loro tipi di dato. Qui, l'unico tipo di dato usato è **int**, e **cat** è posto 4 byte da **dog**, e **bird** 4 byte da **cat**, ecc. Così apparirebbe che, in questa macchina, un **int** è lungo 4 byte.

Un'altra come questo interessante esperimento mostrante come la memoria è mappato, cosa si può fare con un indirizzo? La cosa principale importante da fare è salvarlo dentro un'altra variabile per un uso in seguito. C e C++ hanno un tipo speciale di variabile che detiene un indirizzo. Questa variabile è chiamata *puntatore*.
Other than this interesting experiment showing how memory is mapped out, what can you do with an address? The most important thing you can do is store it inside another variable for later use. C and C++ have a special type of variable that holds an address. This variable is called a *pointer*.
L'operatore che definisce un puntatore è lo stesso usato per la moltiplicazione '*'. Il compilatore sa che non è una moltiplicazione dal contesto in cui è usato, come si vedrà. Quando si definisce un puntatore, si deve specificare il tipo di variabile a cui punta. Si inizia dando un nome di un tipo, dopo invece di immediatamente dare un identificatore per la variabile, si dice "aspetta, è un puntatore " inserendo una stella tra il tipo e l'identificatore. Così il puntatore verso un **int** somiglia come questo:
When you define a pointer, you must specify the type of variable it points to. You start out by giving the type name, then instead of immediately giving an identifier for the variable, you say "Wait, it's a pointer" by inserting a star between the type and the identifier. So a pointer to an **int** looks like this:

```
int* ip; // ip punta ad una variabile int
int* ip; // ip points to an int variable
```

L'associazione di '*' con un tipo appare sensata a si legge facilmente, ma può realmente essere un po' ingannevole. La nostra inclinazione potrebbe essere dire " puntatore **int** " come se fosse un singolo discreto tipo. Comunque, con un **int** o un altro tipo data base, è possibile dire:
The association of the '*' with the type looks sensible and reads easily, but it can actually be a bit deceiving. Your inclination might be to say "intpointer" as if it is a single discrete type. However, with an **int** or other basic data type, it's possible to say:

```
int a, b, c;
```

mentre con un puntatore, ci piacerebbe dire:
whereas with a pointer, you'd *like* to say:

```
int* ipa, ipb, ipc;
```

la sintassi del C (e in eredità, quella del C++) non permette espressioni così sensate. Nella definizione sopracitata, solo **ipa** è un puntatore, ma **ipb** e **ipc** sono ordinari **int**(si può dire che "* lega più saldamente all' identificatore"). Conseguentemente, il miglior risultato può essere realizzato usando solo una definizione per riga; ancora si ottiene una sintassi sensata senza confusione:
C syntax (and by inheritance, C++ syntax) does not allow such sensible expressions. In the definitions above, only **ipa** is a pointer, but **ipb** and **ipc** are ordinary **int**s (you can say that "* binds more tightly to the identifier"). Consequently, the best results can be achieved by using only one definition per line; you still get the sensible syntax without the confusion:

```
int* ipa;
int* ipb;
int* ipc;
```

poichè una linea direttiva per la programmazione in C++ è che si dovrebbe sempre inizializzare una variabile al punto di definizione, questa forma effettivamente funziona meglio. Ad esempio, le variabili soprammenzionate non sono inizializzate ad alcun particolar valore; contengono spazzatura. E' molto meglio dire qualcosa come:
Since a general guideline for C++ programming is that you should always initialize a variable at the point of definition, this form actually works better. For example, the variables above are not initialized to any particular value; they hold garbage. It's much better to say something like:

```
int a = 47;
int* ipa = &a;
```

ora entrambi **a** e **ipa** sono stati inizializzati, e **ipa** detiene l'indirizzo di **a**.
Una volta che si ha inizializzato un puntatore, la cosa principale che si può fare è usarlo per modificare il valore a cui punta. Accedere ad una variabile attraverso un puntatore, si *deferenzia* il puntatore usando lo stesso operatore per definirlo, come questo:
Once you have an initialized pointer, the most basic thing you can do with it is to use it to modify the value it points to. To access a variable through a pointer, you *dereference* the pointer using the same operator that you used to define it, like this:

```
*ipa = 100;
```

ora **a** contiene il valore 100 invece di 47.
Queste sono le basi per i puntatori: si può detenere un indirizzo, e lo si può usare per modificare la variabile originale. Ma la domanda rimane ancora: perché non voler modificare una variabile usando un'altra variabile come una delega?
These are the basics of pointers: you can hold an address, and you can use that address to modify the original variable. But the question still remains: why do you want to modify one variable using another variable as a proxy?
Per questo introduttorio punto di vista, possiamo mettere la risposta in due larghe categorie:
1. cambiare " gli oggetti di fuori " dal di dentro di una funzione. Questo è forse la uso base principale dei puntatori, e sarà esaminato qui.
2. per realizzare molti altri intelligenti tecniche di programmazione, le quali ci impareremo un po' alla volta nel resto del libro.

For this introductory view of pointers, we can put the answer into two broad categories:
1. To change "outside objects" from within a function. This is perhaps the most basic use of pointers, and it will be examined here.
2. To achieve many other clever programming techniques, which you'll learn about in portions of the rest of the book.

Modificare l'oggetto esterno

Normalmente, quando si passa un argomento ad una funzione, una copia di quel argomento viene fatta dentro la funzione. Questo è detto *pass-by-value (passaggio per valore)*. Si può vedere l'effetto del passaggio per valore nel seguente programma:

```
//: C03:PassByValue.cpp
#include <iostream>
using namespace std;
void f(int a) {
  cout << "a = " << a << endl;
  a = 5;
  cout << "a = " << a << endl;
}
int main() {
  int x = 47;
  cout << "x = " << x << endl;
  f(x);
  cout << "x = " << x << endl;
} ///:~
```

In **f(), a** è una *variabile locale*, quindi esiste solo per la durata della chiamata di funzione a **f().** Poiché esso è un argomento di funzione , il valore di **a** è inizializzato dagli argomenti che sono passati quando la funzione è chiamata, in **main()** l'argomento è **x**, il quale ha un valore di 47, quindi questo valore è copiato in **a** quando **f()** è chiamata.

Quando si esegue questo programma si vedrà:

```
x = 47
a = 47
a = 5
x = 47
```

Inizialmente, naturalmente, x vale 47. Quando **f()** viene chiamata, viene creato spazio

temporaneo per tenere la variabile **a** per la durata della chiamata alla funzione ed **a** è inizializzata copiando il valore di **x**, che si verifica stampandola. Naturalmente, si può cambiare il valore di **a** e mostrare che questa è cambiata. Ma quando **f()** è termina, lo spazio temporaneo che è stato creato per **a** scompare e vediamo che l'unica connessione esistita tra **a** e **x** è avvienuta quando il valore di **x** è stato copiato in **a**.
Quando si è dentro **f(), x** è un *outside object* (oggetto esterno, una mia termonologia), e cambiare la variabile locale non influena l'oggetto esterno, naturalmente, poiché essi sono due separate locazioni di memoria. Ma cosa accade se si vuole modificare l'oggetto esterno? Ecco dove i puntatori tornano utili. In un certo senso, un puntatore è uno pseudonimo per un'altra variabile. Quindi se noi passiamo un puntatore in una funzione invece di un ordinario valore, stiamo realmente passando uno pseudonimo dell'oggetto esterno, permettendo alla funzione di modificare l'oggetto esterno:

```
//: C03:PassAddress.cpp
#include <iostream>
using namespace std;
void f(int* p) {
  cout << "p = " << p << endl;
  cout << "*p = " << *p << endl;
  *p = 5;
  cout << "p = " << p << endl;
}
int main() {
  int x = 47;
  cout << "x = " << x << endl;
  cout << "&x = " << &x << endl;
  f(&x);
  cout << "x = " << x << endl;
} ///:~
```

ora **f()** prende un puntatore come argomento e lo dereferenzia durante l'assegnamento e ciò causa la modifica dell'oggetto esterno **x**.Il risultato è:

```
x = 47
&x = 0065FE00
p = 0065FE00
*p = 47
p = 0065FE00
x = 5
```

si noti che il valore contenuto in **p** è lo stesso dell'indirizzo di **x** – il puntatore **p** punta veramente a **x**. Se ciò non è convincente abbastanza, quando **p** è dereferenziato per assegnare il valore 5, si vede che il valore di **x** è ora pure cambiato a 5.
Quindi, passare un puntatore in una funzione permetterà alla funzione di modificare l'oggeto esterno. In seguito si farà un grande uso dei puntatori, ma questo è l'uso più comune e semplice.

Introduzione ai riferimenti in C++

Approssimativamente i puntatori funzionano allo stesso modo in C e in C++, ma il C++ ha un modo aggiuntivo per passare un indirizzo in una funzione. Ciò è il *passaggio per riferimento* ed esiste in vari altri linguaggi di programmazione quindi non è una invenzione del C++.
Come prima impressione sui riferimenti si può pensare che non sono necessari e che si potrebbe scrivere tutti i propri programmi senza i riferimenti. In generale, ciò è vero, con l'eccezione di pochi importanti posti in cui ne apprenderemo in seguito nel libro. Si apprenderà anche di più sui riferimenti in seguito, ma l'idea base è la stessa della dimostrazione dell'uso dei puntatori sopracitato: si può passare l'indirizzo un argomento usando un riferimento. La differenza tra i puntatori e i riferimenti è che chiamare una funzione che accetta i riferimenti è più pulito, sintatticamente, che chiamare una funzione che accetta puntator i(ed è questa differenza sintattica che rende essenziali i riferimenti in

certe situazioni). Se **PassAddress.cpp** è modificato per usare i riferimenti, si può vedere la differenza nella chiamata alla funzione nel **main( )**:

```
//: C03:PassReference.cpp
#include <iostream>
using namespace std;
void f(int& r) {
  cout << "r = " << r << endl;
  cout << "&r = " << &r << endl;
  r = 5;
  cout << "r = " << r << endl;
}
int main() {
  int x = 47;
  cout << "x = " << x << endl;
  cout << "&x = " << &x << endl;
 f(x); // Assomiglia ad un passaggio di valore
           // è in realtà passata per riferimento
  cout << "x = " << x << endl;
} ///:~
```

nella lista di argomenti lista di **f()**, invece di dire **int*** per passare un puntatore, si dice **int&** per passare un riferimento. Dentro **f(),** se di dice solamente **'r'**(il quale produrrebbe l'indirizzo se **r** fosse un puntatore) si ottiene il valore della variabile a cui **r** si riferisce. Se si assegna **r**, si assegna realmente la variabile a cui **r** si riferisce. Infatti, l'unico modo per ottenere l'indirizzo che è stato tenuto in **r** è con l'operatore **'&'.**

In **main( )**, si può vedere l'effetto chiave del riferimento nella sintassi della chiamata a **f(),** la quale è proprio **f(x)**. Anche se assomiglia ad un ordinario passaggio di valore, l'effetto del riferimento è che realmente prende l'indirizzo e lo passa dentro, piuttosto che farne una copia del valore. L'output è :

```
x = 47
&x = 0065FE00
r = 47
&r = 0065FE00
r = 5
x = 5
```

Quindi si può vedere che il passaggio per riferimento permette ad una funzione di modificare l'oggetto esterno, proprio come si fa passando un puntatore (si può anche osservare che il riferimento oscura il fatto che un indirizzo sta per essere passato; ciò sarà esaminato in seguito nel libro). Dunque, per questa semplice introduzione si può ritenere che i riferimenti sono soltanto un modo differente sintatticamente (qualche volta detto " zucchero sintattico ") per ottenere la stessa cosa che il puntatore fa: permette alle funzioni di cambiare gli oggetti esterni.

I puntatori ed i riferimenti come modificatori

Si è visto che i tipi data base **char**, **int**, **float,** e **double,** insieme agli specificatori **signed**, **unsigned**, **short,** e **long**, i quali possono essere usati con i tipi data base quasi in ogni combinazione. Ora si sono aggiunti i puntatori ed i riferimenti che sono ortogonali ai tipi data base e agli specificatori, quindi le possibili combinazioni si sono triplicate:

//: C03:AllDefinitions.cpp

```
// Tutte le possibili combinazioni dei tipi di data base,
//specificatori, puntatori e riferimenti
#include <iostream>
using namespace std;
void f1(char c, int i, float f, double d);
void f2(short int si, long int li, long double ld);
void f3(unsigned char uc, unsigned int ui,
  unsigned short int usi, unsigned long int uli);
```

```
void f4(char* cp, int* ip, float* fp, double* dp);
void f5(short int* sip, long int* lip,
  long double* ldp);
void f6(unsigned char* ucp, unsigned int* uip,
  unsigned short int* usip,
  unsigned long int* ulip);
void f7(char& cr, int& ir, float& fr, double& dr);
void f8(short int& sir, long int& lir,
  long double& ldr);
void f9(unsigned char& ucr, unsigned int& uir,
  unsigned short int& usir,
  unsigned long int& ulir);
int main() {} ///:~
```

I puntatori e i riferimenti funzionano anche quando si passano oggetti dentro e fuori le funzioni; si apprenderà ciò in seguito nell' ultimo capitolo.
C'è solo un altro tipo che funziona con i puntatori: **void**. Se si dichiara che il puntatore è un **void***, significa che qualsiasi tipo di indirizzo può essere assegnato al puntatore (laddove se si ha un **int***, si può assegnare solo l'indirizzo di una variabile **int** a quel puntatore). Ad esempio:

```
//: C03:VoidPointer.cpp
int main() {
  void* vp;
  char c;
  int i;
  float f;
  double d;
 //L'indirizzo di QUALSIASI tipo può essere
//assegnato ad un puntatore void
  vp = &c;
  vp = &i;
  vp = &f;
  vp = &d;
} ///:~
```

ogni volta che si assegna ad un **void***, si perde ogni informazione sul tipo. Ciò significa che prima che si usa il puntatore, lo si può castare nel tipo corretto:

```
//: C03:CastFromVoidPointer.cpp
int main() {
  int i = 99;
  void* vp = &i;
// Non si può dereferenziare un puntatore void:
  // *vp = 3; // errore a Compile-time
  // Si deve fare un cast ad un int prima di dereferenziare:
  *((int*)vp) = 3;
} ///:~
```

il tipo **(int*)vp** prende il **void***e dice al compilatore di trattarlo come un **int***, così può essere dereferenziato con successo. Si può osservare che questa sintassi è brutta, ed è, ma il peggio è che il **void*** introduce un buco nel sistema del tipo di linguaggio. Cioè, permette, o anche incoraggia, il trattamento di un tipo come un altro tipo. Nell'esempio di sopra, si è trattato un **int** come un **int** assegnando **vp** ad un **int***, ma non c'è nulla che dice che noi non possiamo assegnarlo ad un **char*** o **double***,che modificherebbe un differente ammontare di memoria che è stata allocata per l'**int**, possibilmente interrompendo un programma. In generale, i puntatori **void** dovrebbero essere evitati, e usati solo in rari e speciali casi, i quali saremo pronti a considere significativamente in seguito nel libro.
Non si può avere un riferimento **void**, per ragioni che saranno spiegate nel capitolo 11.

Scoping (Visibilità)

Le regole di scope ci dicono dove una variabile è valida, dove viene creata e dove viene distrutta(es. va fuori dallo scope). Lo scope della variabile si estende dal punto dove è definita fino alla prima parentesi chiusa che confronta la più vicina parentesi aperta prima che la variabile fu definita. Cioè, uno scope è definito dalla propria serie di parentesi più "vicine". Per illustrarle:

```
//: C03:Scope.cpp
// quante variabili sono esaminate
// How variables are scoped
int main() {
  int scp1;
// scp1 visibile qui

  {
    // scp1 visibile qui ancora
    //.....
    int scp2;
   // scp2 visibile qui
    //.....
    {
      // scp1 & scp2 visibili ancora qui
      //..
      int scp3;
      //scp1 ,scp2 & scp3 visibili qui
      // ...
    }
// <-- scp3 distrutto qui
// scp3 non disponibile qui
// scp1 & scp2 visibili ancora qui
// ...
} // <-- scp2 distrutto qui
// spc2 & scp3 non disponibili qui
// ..
} // <-- scp1 distrutto qui
///:~
```

l'esempio di sopra mostra quando le variabili sono visibili e quando non sono disponibili (ciòè, quando vanno fuori dallo scope). Una variabile può essere usata solo quando è dentro il proprio scope. Gli scope possono essere nidificati. Nidificare significa che si può accedere ad una variabile in uno scope che racchiude lo scope in cui si è. Nell'esempio di sopra, la variabile **scp1** è disponibile dentro tutti gli altri scope, mentre **scp3** è disponibile solo nello scope più interno.

Definire le variabili al volo

Come visto poco fa in questo capitolo, c'è una significativa differenza tra C e C++ quando si definiscono le variabili. Entrambi i linguaggi richiedono che le variabili siano definite prima di essere usate, ma il C (e molti altri linguaggi tradizionali procedurali) obbligano a definire tutte le variabili all'inizio dello scope, cosicchè quando il compilatore crea un blocco può allocare spazio per queste variabili.

Mentre si legge il codice C, un blocco di definizioni di variabili è di solito la prima cosa da vedere quando si entra in uno scope. Dichiarare tutte le variabili all'inizio del blocco richiede che il programmatore scriva in un particolare modo a causa del compimento di dettagli del linguaggio. La maggior parte della gente non sa che tutte le variabili sono state usate prima che loro scrivono il codice, quindi devono cominciare a saltare indietro all'inizio del blocco per inserire nuove variabili, che è scomodo e causa errori. Queste definizioni di variabili di solito non significano molto per il lettore e realmente tendono a confondere perché appaiono fuori dal contesto nelle quali sono usati.

C++ (non il C) ci permette di definire variabili da qualsiasi pari in uno scope, quindi si può definire una variabile giusto prima di usarla.In più, si può inizializzarla una variabile al punto in cui la si definisce, che impedisce di compiere varie classi di errori. Definire variabili in questo modo rende il codice più facile da scrivere e riduce gli errori che si ottengono dal dover saltare avanti e dietro in uno scope. Rende il codice più facile da capire perché si vede una variabile definita nel contesto del proprio uso. Questo è molto importante quando si sta definendo e inizializzando una variabile allo stesso tempo – si può vedere il significato dell'inizializzazione del valore dal modo in cui la variabile è usata. Si possono anche definire variabili dentro le espressioni di controllo di cicli **for** e **while**, dentro la condizione di una istruzione **if**, e dentro l' istruzione di selezione di uno **switch**. Ecco qui un esempio mostrante le definizioni di variabili al volo:

```
//: C03:OnTheFly.cpp

// definizione di variabili al volo
#include <iostream>
using namespace std;

int main() {
  //..
  {
// Inizia un nuovo scope
    int q = 0; // il C richiede una definizione
    qui


    //..
```

```
// definizione nel punto di uso:

   for ( int i = 0; i < 100; i++) {



    q++; // q proviene da un grande
   scope


 // definizione alla fine dello scope:



     int p = 12;


   }

int p = 1;   // un p differente

 } // fine scope contenente q & p



 cout << "Digita caratteri:" << endl;
```

```
while ( char c = cin.get() != 'q') {


  cout << c << " non era

  lui!" << endl;


  if ( char x = c == 'a' || c == 'b')



    cout << "hai digitato

  a oppure b" << endl;


  else


    cout << "hai digitato"

  << x << endl;


}


cout << "Digita A, B, oppure C" <<

  endl;


switch ( int i = cin.get()) {
```

```
  case 'A': cout << "Snap" <<
  endl; break ;

  case 'B': cout << "Crackle" <<
  endl; break ;

  case 'C': cout << "Pop" <<
  endl; break ;

  default : cout << "Non A, B o
  C!" << endl;

 }

} ///:~
```

nello scopo più interno, **p** è definito proprio prima che lo scope finisca, quindi è realmente un gesto inutile (ma mostra che si puo' definire una variabile da qualche parte). Il **p** nello scope esterno è nella stessa situazione.

La definizione di **i** nell' espressione di controllo del ciclo **for** è un esempio di come si possa definire una variabile esattamente nel punto in cui si ha bisogno (lo si puo' fare solo col C++). Lo scope di **i** è lo scope dell' espressione controllata del ciclo **for** , quindi si puo' girare intorno e riusare **i** per il prossimo ciclo **for** . Questo idioma è semplice e comunemente usato in C++; **i** è il classico nome per un contatore di ciclo e non si devono inventarne di nuovi.

Benchè l' esempio mostra anche variabili definite senza le dichiarazioni **while** , **if,** e **switch,** questo tipo di definizione è abbastanza meno comune di queste nelle espressioni **for** , forse perchè la sintassi è molto vincolata. Ad esempio, non si possono usare parentesi. Cioè, non si puo' dire:

```
while (( char c = cin.get()) != 'q')
```

L' aggiunta di parentesi extra sembrerebbe una cosa innocente e utile da fare, e poichè non si puo' usarli, il risultato non è quello che potrebbe piacere. Il problema si presenta perchè' !=' ha una precedenza maggiore rispetto a' =' , quindi **char c** finisce contenendo un **bool** convertito in **char** . Quando è stampato, su molti terminali si vedrà una faccia sorridente come carattere.

In generale, si puo' considerare l' abilità del definire variabili senza le dichiarazioni **while** , **if,** e **switch** per completezza, ma l' unico posto dove si userà questo tipo di definizione di variabile è dentro un ciclo **for** (dove lo si userà abbastanza spesso).

Specificare un' allocazione di memoria

Quando si crea una variabile, si hanno diversi modi per specificare il tempo di vita di una variabile, come la memoria viene allocata per quella variabile e come la variabile è trattata dal compilatore.

Le variabili globali

Le variabili globali sono definite fuori da tutti i corpi delle funzioni e sono disponibili in tutte le parti del programma (anche codificati in altri file). Le variabili globali sono semplici da esaminare è sempre disponibili (es. il tempo di vita di una variabile globale finisce quando il programma termina). Se l' esistenza della variabile globale in un file è dichiarata usando la parola chiave **extern** in un altro file, il dato è disponibile per l' uso dal secondo file. Ecco qui un esempio di uso di variabili globali:

```
//: C03:Global.cpp

//{L} Global2

// dimostrazioni delle variabili globali

#include <iostream>

using namespace std;

int globale;

void func();

int main() {

  globale = 12;

  cout << globale << endl;



func(); // Modifica globale
```

```
  cout << globale << endl;

} ///:~
```

ecco qui un file che accede a **globale** come un **extern** :

```
//: C03:Global2.cpp {O}

// accesso alle variabili globali esterne

extern int globale;

// ( il linker risolve il riferimento )

void func() {

  globale = 47;

} ///:~
```

la memoria per la variabile **globale** è creata dalla definizione di **Global.cpp** e quella stessa variabile è acceduta dal codice in **Global2.cpp.** Poichè il codice in **Global2.cpp** è compilato separatamente dal codice nel **Global.cpp,** il compilatore deve essere informato che la variabile esiste altrove dalla dichiarazione.

```
extern int globale;
```

quando si esegue il programma, si vedrà che la chiamata a **func()** veramente influisce sulla singola globale istanza di **globale** .

In **Global.cpp** si puo' vedere uno speciale commento (i quali sono un mio progetto):

```
//{L} Global2
```

cio' significa che per creare il programma finale, il file oggetto col nome **Global2** deve essere linkato (non c' è nessuna estenzione perchè i nomi di estenzioni dei file oggetto differiscono da un sistema all' altro). In **Global2.cpp** la prima riga ha un altro speciale commento in una serie di caratteri **{O},** il quale dice: " non provare a creare un eseguibile da questo file, esso va compilato in modo che possa essere linkato a qualche altro eseguibile". Il programma **ExtractCode.cpp nel** Volume 2 di questo libro (scaricabile dal sito *www.BruceEckel.com*) legge questa serie di caratteri e crea l'appropriato **makefile** e ogni cosa è compilata correttamente (si apprenderanno i **makefile** alla fine di questo capitolo).

Variabili locali

Le variabili locali vivono dentro uno scope, sono " locali " alla funzione. Esse spesso sono chiamate variabili automatiche perchè vengono automaticamente create quando si entra nello scope e automaticamente muiono quando lo scope finisce. La parola chiave **auto** la rende esplicita, ma le variabili locali per default sono' **auto** e quindi non è mai necessario dichiarare qualcosa come un **auto** .

Variabili register

Una variabile **register** è un tipo di variabile locale. La parola chiave **register** dice al compilatore: " rendi l'accesso a questa variabile il più veloce possibile " . Incrementare la velocità di accesso è un compito che dipende dall'implementazione, ma come, il nome suggerisce, è spesso fatto piazzando la variabile in un registro. Non c' è garanzia che la variabile sarà posta in un registro o anche che la velocità di accesso sarà incrementata. Esso è un suggerimento al compilatore.

Ci sono delle restrizioni per l' uso delle variabili **register** . Non si puo' prendere o calcolare l' indirizzo della variabile **register** . Una variabile **register** puo' essere dichiarata solo dentro un blocco (non si hanno variabili globali o **static register**). Si puo', comunque, usare un **register** come un argomento formale in una funzione (es. nella lista degli argomenti).

In generale, le ottimizzazioni del compilatore saranno migliori delle nostre, quindi è meglio evitare la parola chiave **register** .

static

La parola chiave **static** ha diversi significati distinti. Normalmente, le variabili definite locali nella funzione scompaiono alla fine dello scope della funzione. Quando si chiama una funzione di novo, la memoria per le variabili è creata di nuovo e i valori sono rinizializzati. Se si vuole che un valore sia esistente in tutta la vita del programma, si puo' definire un variabile locale della funzione per essere **static** e darle un valore iniziale. L' inizializzazione è prodotta solo la prima volta che la funzione è chiamata ed il dato conserva il proprio valore tra le chiamate alla funzione. In questo modo, una funzione puo' " ricordarsi " alcuni pezzi di informazioni tra le chiamate alla funzione.

Si puo' restare forse meravigliati del perchè una variabile globale invece non è usata. La bellezza di una variabile **static** è che non è disponibile fuori lo scope della funzione, quindi non puo' essere inavvertitamente cambiata. Essa localizza l' errore.

Ecco qui un esempio di uso di variabili **static** :

```
//: C03:Static.cpp

// usare una variabile statica in una funzione

#include <iostream>

using namespace std;

void func() {

  static int i = 0;

  cout << "i = " << ++i << endl;

}
```

```
int main() {

  for ( int x = 0; x < 10; x++)

    func();

} ///:~
```

ogni volta che **func()** vienechiamata nel ciclo **for** , esso stampa un valore differente. Se la parola chiave **static** non è usata, il valore stampato sarà sempre ' 1' .

Il secondo significato di **static** è riferito al primo nel senso " non disponibile fuori un certo scope " . Quando **static** è applicato ad un nome di funzione o ad una variabile che è fuori da tutte le funzioni, significa " questo nome non è disponibile fuori da questo file. " Il nome di funzione o la variabile è locale al file, si dice che ha un *file scope*. Come dimostrazione, compilare e collegare i seguenti due file causerà un errore di collegamento:

```
//: C03:FileStatic.cpp

// dimostrazione dello scope del file. Compilando e

// linkando questo file con FileStatic2.cpp

// si avrà un errore dal linker

// Scope di file significa che è solo disponibile in questo file:

static int fs;

int main() {

  fs = 1;

} ///:~
```

anche se la variabile **fs** esiste come un **extern** nel seguente file, il linker non la troverà perchè è stata dichiarata **static** in **FileStatic.cpp** .

```
//: C03:FileStatic2.cpp {O}

provando a riferisi fs

// Trying to reference fs

extern int fs;

void func() {

  fs = 100;

} ///:~
```

un specificatore **static** puo' essere usato anche dentro una **class** . Questa spiegazione sarà rimandata fino a quando non si imparerà a creare classi, più avanti nel libro.

extern

la parola chiave **extern** è già stata descritta e dimostrata in breve. Essa dice al compilatore che una variabile o una funzione esiste, anche se il compilatore non l' ha ancora vista nel file corrente che sta per essere compilato. Questa variabile o funzione puo' essere definita in un altro file o più avanti nel file corrente. Come esempio dell' ultimo:

```
//: C03:Forward.cpp

// Forward function & dichiarazione di dati

// Forward function & data declarations

#include <iostream>

using namespace std;

// questo non è realmente esterno, ma bisogna dire al compilatore

//che esiste da qualche parte:

extern int i;

extern void func();

int main() {

  i = 0;

  func();

}

int i; // definizione di dato

void func() {

  i++;

  cout << i;

} ///:~
```

quando il compilatore incontra la dichiarazione ' **extern int i'** , sa che la definizione per **i** deve esistere da qualche parte come una variabile locale. Quando il compilatore raggiunge la definizione di **i** , nessun altra definizione è visibile, quindi esso sa che ha trovato la stessa **i** dichiarata poco fa nel file. Se si avesse definito **i** come **static** , si sarebbe detto al

compilatore che la **i** è definita globalmente (passando per **extern**), ma che ha anche un file scope(passando per **static**), quindi il compilatore genererà un errore.

Linkage

Per capire il comportamento dei programmi in C e C++, si ha bisogno di conoscere il linkage (il concatenamento). In un programma eseguibile, un identificatore è rappresentato dallo spazio in memoria che detiene una variabile o un corpo di funzione compilato. Il collegamento descrive questa memoria come se fosse vista dal linker. Ci sono due tipi di collegamento: *internal linkage* e *external linkage*.

Concatenamento interno (internal linkage) significa che la memoria è creata per rappresentare l' identificatore solo per il file che sta per essere compilato, gli altri file possono usare gli stessi nomi di identificatori con il concatenamento interno, o per la variabile globale, e nessun controversia sarà trovata dal linker, viene creata memoria per ogni identificatore. Il concatenamento interno è specificato dalla parola chiave **static** in C e C++.

concatenamento esterno (external linkage) significa che un singolo pezzo di memoria è creata per rappresentare l' identificatore per tutti i file che stanno per essere compilati. La memoria è creata una volta ed il linker deve risolvere tutti gli altri riferimenti a quella memoria. Le variabili globali ed i nomi di funzione hanno un collegamento esterno. Queste sono accessibili dagli altri file dichiarandole con la parola chiave **extern** . Le variabili definite fuori da tutte le funzioni (con l' eccezione di **const** in C++) e le definizioni di funzione sono per default concatenamento esterno. Si puo' specificamente obbligarli ad avere un concatenamento interno usando la parola chiave **static** . Si puo' esplicitamente dichiarare che un identificatore ha un concatenamento esterno definendolo con la parola chiave **extern**. Definire una variabile o una funzione con **extern** non è necessario in C, ma lo è qualche volta per **const** in C++.

Le variabili automatiche (locali) esistono solo temporaneamente, nello stack, mentre una funzione sta per essere chiamata. Il linker non conosce le variabili automatiche e quindi queste *non hanno linkage* .

Le costanti

Nel vecchio (pre-Standard) C, se si voleva una costante, si doveva usare il preprocessore:

#define PI 3.14159

dovunque si usava **PI** , il valore 3.14159 veniva sostituito dal preprocessore(si puo' ancora usare questo metodo in C ed in C++).

quando si usa il preprocessore per creare costanti, si piazza il controllo di queste costanti, fuori dalla portata del compilatore. Nessun controllo del tipo è prodotto sul nome di **PI** e non si puo' prendere l' indirizzo di **PI** (così non si puo' passare il puntatore o un riferimento a **PI**). **PI** non puo' essere una variabile di un tipo definito dall' utente. Il significato di **PI** dura dal punto in cui è definito fino alla fine del file; il preprocessore non riconosce lo scoping.

Il C++ introduce il concetto di costanti con nome che sono proprio come le variabili, eccetto che il loro valore non puo' essere cambiato. Il modificatore **const** dice al compilatore che il nome rappresenta una costante. Ogni tipo di dato, predefinito o definito dall' utente, puo' essere definito come **const** . Se si definisce qualcosa come **const** e dopo si cerca di modificarlo, il compilatore genererà un errore.

Si deve specificare il tipo di **const** :

const int x = 10;

nello standard C e C++, si puo' usare una costante con nome in un lista di argomenti, anche se l' argomento è un puntatore o un riferimento (es. si puo' prendere l' indirizzo di un **const**) un **const** ha uno scope, proprio come una variabile locale, quindi si puo' " nascondere " un**const** dentro una funzione ed essere sicuri che il nome non influenzerà il resto del programma.

Il **const** fu preso dal C++ e incorporato nello standard C, sebbene abbastanza diffentemente. In C, il compilatore tratta un **const** proprio come una variabile che ha una speciale etichetta che dice " non cambiarmi " . Quando si definisce una **const** in C, il compilatore crea memoria per essa, così se si definisce più di una **const** con lo stesso nome in due differenti file (o si pone la definizione in un file principale), il linker genererà un messaggio di errore circa la controversia. L' uso inteso del **const** in C è abbastanza differente di quello inC++ (in breve, è migliore in C++) .

Valori costanti

In C++, una **const** deve sempre avere un valore di inizializzazione (in C ciò non è vero). I valori costanti per i tipi predefiniti come decimali, ottali, esadecimali, o numeri in virgola mobile (tristemente, i numeri binari non sono considerati importanti), o come caratteri.

In assenza di ogni altro indizio, il compilatore assume un valore costante come numero decimale. Il numero 47, 0, e 1101 sono tutti trattati come numeri decimali.

Un valore costante con un primo numero 0 è trattato come un numero ottale (base 8). I numeri a base 8 possono contenere solo digitazioni 0-7; il compilatore richiama le digitazioni oltre il 7 come errore. Un leggittimo numero ottale è 017 (15 in base 10).

Un valore costante con un primo numero 0x è trattato come un numero esadecimale (base 16). I numeri in base 16 contengono digitazioni 0-9 e a-f o A-F. Un legittimo numero esadecimale è 0x1fe (510 in base 10).

I numeri in virgola mobile possono contenere punti decimali ed potenze esponenziali (rappresentata da e, che significa " 10 alla potenza di "): il punto decimale e la **e** sono opzionali. Se si assegna una costante da una variabile in virgola mobile, il compilatore prenderà il valore costante e lo convertirà in un numero in virgola mobile (questo processo è una forma di ciò che è chiamato *conversione di tipo implicito*). Comunque è una buona idea usare il punto deciamale o una **e** per ricordare al lettore che si sta usando un numero in virgola mobile; anche alcuni vecchi compilatori hanno bisogno del suggerimento.

Leggittimi valori costanti in virgola mobile sono: 1e4, 1.0001, 47.0, 0.0, e -1.159e-77. Si puo' aggiungere il suffisso per forzare il tipo di numero in virgola mobile. **f** o **F** forza un **float** , **L** o **l** forza un **long double** ; altrimenti il numero sarà un **double** .

Le costanti carattere sono caratteri racchiuse tra singole apicette come: ' **A**' ,' **o**' ," . Si noti che c' è una grossa differenza tra il carattere' **o**' (ASCII 96) e il valore 0. i caratteri speciali sono rappresentati con il " backslash escape " :' **\n**' (nuova riga),' **\t**' (tab),' ****' (backslash),' **\r**' (carriage return),' **\"**' (doppia apicetta),' **\"** (singola apicetta), etc. si puo' snche esprimere le costanti char in ottale:' **\17**' o in esadecimale:' **\xff**' .

volatile

laddove il qualificatore **const** dice al compilatore che " questo non cambia mai " (il quale permette al compilatore di produrre una ottimizzazione extra), il qualificatore **volatile** dice al compilatore " non si sa mai quando questo cambierà " , e previene il compilatore dal produrre qualsiasi ottimizzazione basata sulla stabilità di quella variabile. Si usi questa parola chiave quando si leggono valori fuori dal controllo del proprio codice, come un registro in un pezzo di hardware per comunicazioni. Una variabile **volatile** è sempre letta ogni volta che è richiesta, anche se è già stata letta la riga prima.

Un caso speciale di un pò di memoria che è " fuori dal controllo del nostro codice " è un programma multithreaded. Se si sta osservando un particolare flag che è modificato da un' altra thread o processo, quel flag dovrebbe essere **volatile** così il compilatore non crea la assunzione che puo' ottimizzare letture multiple del flag.

Si noti che **volatile** puo' non avere effetto quando un compilatore non è ottimizzato, ma puo' prevenire bug critici quando si inizia ad ottimizzare il codice(che accade quando il compilatore comincerà a guardare le letture ridondanti).

Le parole chiave **const** e **volatile** saranno illustrate nel capitolo in seguito.

Gli operatori e il loro uso

Questa sezione illutstra tutti gli operatori del C e del C++.

Tutti gli operatori producono un valore dai loro operandi. Questo valore è prodotto senza modificare gli operandi, eccetto con gli operatori assegnamento, incremento e decremento. Modificare un operando è chiamato un *side effect(effetto secondario)*. Il più comune uso per gli operatori che modificano i loro operandi è generare il side effect, ma si dovrebbe tenere a mente che il valore prodotto è disponibile solo per il nostro uso proprio come nell' operatore senza gli effetti secondari .

L'assegnamento

L' assegnamento è prodotto con l' operatore =. Esso significa " prendi la parte di destra (spesso chiamata *rvalue*) e copiala dentro la parte di sinistra (spesso chiamato *lvalue*). " Un rvalue è una qualsiasi costante, variabile o una espressione che produce un valore, maun lvalue deve essere una distinta, variabile chiamata(cioè, deve essere uno spazio fisico nella quale salvare il dato). Ad esempio, si puo' assegnare un valore costante alla variabile (**A = 4;**), ma non si puo' assegnare qualcosa al valore costante- non puo' essere un lvalue (non si puo' dire **4 = A;**).

Gli operatori matematici

Gli operatori basici matematici sono gli stessi di quelli disponibili nella maggior parte dei linguaggi di programmazione: addizione (**+**), sottrazione (**-**), divisione (**/**), moltiplicazione (*****), e modulo (**%** ; ciò produce il resto delle divisioni con interi). Le divisioni con interi troncano il risultato (non arrotonda). L' operatore modulo non puo' essere usato con un numero a virgola mobile.

Il C ed il C++ usano anche una notazione stenografica per produrre una operazione ed un assegnamento allo stesso tempo. Questo è denotato da un operatore seguito da un segno uguale, ed è consistente con tutti gli operatori nel linguaggio (ogni volta che ha senso). Ad esempio, aggiungere 4 alla variabile **x** e assegnare **x** al risultato, si dice: **x += 4;** .

Questo esempio mostra l' uso degli operatori matematici:

```
//: C03:Mathops.cpp

// operatori matematici

#include <iostream>

using namespace std;

// una macro per visualizzare una stringa ed un valore.

#define PRINT(STR, VAR) \

  cout << STR " = " << VAR << endl

int main() {

  int i, j, k;

float u, v, w;   //   si applica anche ai double

  cout << "inserisci un intero: " ;

  cin >> j;

  cout << "inserisci un altro intero: " ;

  cin >> k;

  PRINT( "j" ,j);   PRINT( "k" ,k);

  i = j + k; PRINT( "j + k" ,i);

  i = j - k; PRINT( "j - k" ,i);

  i = k / j; PRINT( "k / j" ,i);
```

```
  i = k * j; PRINT( "k * j" ,i);
  i = k % j; PRINT( "k % j" ,i);
  // il seguente funziona solo con gli interi:
  j %= k; PRINT( "j %= k" , j);
  cout << "Inserisci un numero a virgola mobile: " ;
  cin >> v;
  cout << "Inserisci un altro numero a virgola mobile:" ;
  cin >> w;
  PRINT( "v" ,v); PRINT( "w" ,w);
  u = v + w; PRINT( "v + w" , u);
  u = v - w; PRINT( "v - w" , u);
  u = v * w; PRINT( "v * w" , u);
  u = v / w; PRINT( "v / w" , u);
    // il seguente funziona con int,char
// e anche i double:

  PRINT( "u" , u); PRINT( "v" , v);
  u += v; PRINT( "u += v" , u);
  u -= v; PRINT( "u -= v" , u);
  u *= v; PRINT( "u *= v" , u);
  u /= v; PRINT( "u /= v" , u);
} ///:~
```

gli rvalue di tutti gli assegnamenti possono, naturalmente, essere ben più complessi.

Introduzione alle macro del preprocessore

Si noti che l' uso della macro **PRINT( )** per salvare il testo (ed i suoi errori!). Le macro del preprocessore sono tradizionalmente chiamati con tutte lettere maiuscole quindi risaltano, si apprenderà in seguito chele macro possono velocemente diventare pericolose (e possono essere anche molto utili).

Gli argomenti nella lista nella parentesi successiva al nome della macro sono sostituiti in tutto il codice che segue la parentesi chiusa. Il preprocessore rimuove il nome **PRINT** e sostituisce il codice dovunque la macro è chiamata, quindi il compilatore non puo' generare nessun messaggio di errore usando il nome della macro, e non esegue alcun tipo di controllo del tipo sugli argomenti (l' ultimo puo' essere giovevole, come mostrato nei comandi di debug alla fine del capitolo).

Gli operatori relazionali

Gli operatori relazionali stabilicono una relazione tra il valori degli operandi. Essi producono un booleano (specificato con la parola chiave **bool** in C++) **true** se la relazione è vera e **false** se la relazione è falsa. Gli operatori relazionali sono: più piccolo di (<), più grande di (>), più piccolo o uguale di (<=), più grande o uguale di (>=), equivalente (= =), e diverso da (!=). Essi possono essere usati con tutti i tipi di dato predefiniti in C e C++. In C++ possono dare ad essi speciali definizioni per i tipi data definiti dagli utenti(li si apprenderà nel capitolo12, il quale riguarda l' operatore sovraccaricato).

Gli operatori logici

Gli operatori logici *and* (**&&**) e *or* (||) producono un **true** o **false** basato sulla relazione logica dei suoi argomenti. Si ricordi che in C e in C++, una dichiarazione è **vera** se essa ha il valore non-zero, è **falsa** se ha un valore di zero. Se si stampa un **bool** , tipicamente si vedrà un **' 1'** per **true** e **' 0'** per **false** .

Questo esempio usa gli operatori relazionali e logici:

```
//: C03:Boolean.cpp

// Operatori relazionali e logici.


#include <iostream>

using namespace \std;

int main() {

  int i,j;
```

```
  cout << "Inserisci un intero: " ;
  cin >> i;
  cout << "Inserisci un altro intero: " ;
  cin >> j;
  cout << "i > j is " << (i > j) << endl;
  cout << "i < j is " << (i < j) << endl;
  cout << "i >= j is " << (i >= j) << endl;
  cout << "i <= j is " << (i <= j) << endl;
  cout << "i == j is " << (i == j) << endl;
  cout << "i != j is " << (i != j) << endl;
  cout << "i && j is " << (i && j) << endl;
  cout << "i || j is " << (i || j) << endl;
  cout << " (i < 10) && (j < 10) is "
      << ((i < 10) && (j < 10))   << endl;
} ///:~
```

si puo' rimpiazzare la definizione per **int** con **float** o **double** nel programma sopracitato. Bisogna stare attenti, comunque, che il confronto di un numero a virgola mobile con il valore zero è preciso; un numero che è di frazione differente da un altro numero è ancora " non-uguale " . Un numero a virgola mobile che il più piccolo bit sopra zero è ancora vero.

operatori su bit (Bitwise)

Gli operatori su bit permettono di manipolare bit individuali in un numero (poichè i valori a virgola mobile usano un formato interno, gli operatori bit a bit funzionano solo con tipi integrali : **char** , **int** e **long**). Gli operatori su bit effettuano l' algebra Booleana nei corrispondenti bit negli argomenti per produrre il risultato.

L' operatore bit a bit *and* **(&)** produce un uno nel bit dell' output se entrambi i bit dell' input sono alzati; altrimenti esso produce zero. Il bitwise *or* oppure l' operatore (|) produce un uno nel bit dell' output se uno dei due bit in input è uno e produce uno zero se entrambi i bit in input sono zero. Il bitwise *exclusive or*, o *xor* (^) produce di primo ordine nel bit dell' output se il primo o l' altro bit del input, ma non entrambi, è alzato. Il bitwise *not* (**~,** chiamato anche l'operatore complemento ad uno) è un operatore unario, prende solo un argomento (tutti gli altri operatori bit a bit sono operatori binari). I bitwise non producono l' opposto del bit di input, un uno se il bit del input è zero, uno zero se il bit del input è uno.

Gli operatori su bit possono essere combinati col segno = per unire l' operazione e l' assegnamento: **&=** , **|=,** e ^= sono tutte operazioni leggittimw (poichè ~ è un operatore unario che non puo' essere combinato col segno =).

operatori Shift

Anche gli operatori shift manipolano bit. L' operatore left-shift (<<) shifta a sinistra l' operando a sinistra dell' operatore del numero di bit specificati dopo l' operatore. L' operatore right-shift (>>) shifta l' operando a sinistra dell' operatore a destra del numero di bit specificati dopo l' operatore. Se il valore dopo l' operatore shift è più grande del numero di bit nell' operatore di sinistra, il risultato è indefinito. Se l' operatore di sinistra è senza segno, il right shift è un logico shift quindi il bit alto sarà riempito con zero. Se l' operatore di sinistra ha segno, il right shift puo' o non essere un logico shift (cioè, il comportamento è indefinito).

Shift puo' essere combinato con un segno uguale (<<= e >>=). Il lvalue è rimpiazzato dal lvalue shiftato di rvalue.

Quello che segue è un esempio che dimostra l' uso di tutti gli operatori riguardanti i bit. Per primo, ecco una funzione di scopo generale che stampa un byte in formato binario, creato separatamente così che puo' essere facilmente riusato. L'header file dichiara la funzione:

```
//: C03:printBinary.h

// visualizzare un byte in binario

void printBinary( const unsigned char val);

///:~
```

ecco qui un'implementazione della funzione:

```
//: C03:printBinary.cpp {O}

#include <iostream>

void printBinary( const unsigned char val) {

  for ( int i = 7; i >= 0; i--)

    if (val & (1 << i))

      std::cout << "1" ;

    else

      std::cout << "0" ;

} ///:~
```

la funzione **printBinary( )** prende un singolo byte e lo visualizza bit per bit. L' espressione

(1 << i)

produce un uno in ogni successiva posizione del bit; in binario: 00000001, 00000010, etc. se questo bit è un bitwise and con **val** e il valore non è 0, significa che c' era uno in quella posizione in **val** .

Finalmente, la funzione usata nell' esempio che mostra gli operatori che manipola il bit:

```
//: C03:Bitwise.cpp

//{L} printBinary

// dimostrazione della manipolazione dei bit

#include "printBinary.h"

#include <iostream>

using namespace std;

// una macro per scrivere meno:

#define PR(STR, EXPR) \

  cout << STR; printBinary(EXPR); cout << endl;

int main() {

  unsigned int getval;

  unsigned char a, b;

cout << "digita un numero compreso tra 0 e 255: " ;

  cin >> getval; a = getval;

  PR( "a in binario: " , a);

  cout << "digita un numero compreso tra 0 e 255: " ;

  cin >> getval; b = getval;

  PR( "b in binario: " , a);

  PR( "a | b = " , a | b);

  PR( "a & b = " , a & b);
```

```
  PR( "a ^ b = " , a ^ b);

  PR( "~a = " , ~a);

  PR( "~b = " , ~b);

// un interessante pattern di bit :

  unsigned char c = 0x5A;

  PR( "c in binario: " , c);

  a |= c;

  PR( "a |= c; a = " , a);

  b &= c;

  PR( "b &= c; b = " , b);

  b ^= a;

  PR( "b ^= a; b = " , b);

} ///:~
```

ancora una volta, una macro del preprocessore è utilizzato per scrivere di meno. Essa stampa la stringa della nostra scelta, poi la rappresentazione binaria di un' espressione, dopo un newline.

In **main( )** le variabili sono **unsigned** . Questo perchè, in generale, non si vuole il segno quando si sta lavorando con i byte. Un **int** deve essere usato invece di una **char** per **getval** perchè la dichiarazione " **cin >>** " alla prima digitazione sarà trattata altrimenti come carattere. Assegnando **getval** ad **a** e **b** , il valore è convertito in un singolo byte (troncandolo).

Il << ed il >> esegue lo shift dei bit, ma quando i bit vengono shiftati alla fine del numero, questi bit sono persi (è comune dire che loro cadono dentro il mitico *bit bucket*, un posto dove i bit scartati terminano, presumibilmente così possono essere riusati ...). Quando si manipolano i bit si puo' anche produrre la rotazione (*rotation)* , che significa che i bit che arrivano alla fine ricompaiono all' altro capo, come se stessero ruotando in giro. Anche se la maggior parte dei processori dei computer forniscono un comando di rotazione a livello di macchina (lo si vedrà nel linguaggio assembler per quel processore), non c' è un supporto diretto per la "rotazione" in C o in C++. Presumibilmente i creatori del C si sentirono giusificati nel tralasciare la " ruotazione" (aspirando, come essi dissero, ad linguaggio minimale) perchè si puo' costruire il proprio comando di rotazione. Ad esempio, ecco qui funzioni per eseguire rotazioni a destra e a sinistra:

```
//: C03:Rotation.cpp {O}

// produce rotazioni a destra e a sinistra
```

```
unsigned char rol( unsigned char val) {
int highbit;
  if (val & 0x80) // 0x80 è il bit alto solo
    highbit = 1;
  else
    highbit = 0;
// left shift( l'ultimo bit diventa 0) :
  val <<= 1;
// ruota il bit alto all'ultimo:
  val |= highbit;
  return val;
}
unsigned char ror( unsigned char val) {
  int lowbit;
  if (val & 1) // controlla il bit basso
    lowbit = 1;
  else
    lowbit = 0;

  val >>= 1; //spostamento a destra di una posizione
  // ruota il bit basso all'inizio:
  val |= (lowbit << 7);
  return val;
} ///:~
```

si provi ad usare queste funzioni in **Bitwise.cpp** . Si noti che le definizioni (o almeno le dichiarazioni) di **rol()** e **ror()** devono essere viste dal compilatore in **Bitwise.cpp** . prima che le funzioni vengano usate.

Le funzioni bit a bit sono estremamente efficienti da usare perchè si traducono direttamente in dichiarazioni del linguaggio assembler. Qualche volta una singola dichiarazione in C o C++ genererà una singola riga del codice assembler.

Gli operatori unari

Non ci sono solo operatori che prendono un singolo argomento. Il suo compagno, il *not logico* (**!**), prenderà il valore **true** e produrrà un valore **false** . L' unario meno (**-**) e l' unario più (+) sono gli stessi operatori del binario meno e più; il compilatore capisce a quale uso è diretto dal modo in cui si scrive l' espressione. Per esempio, l'istruzione

```
x = -a;
```

ha un significato ovvio. Il compilatore puo' capire:

```
x = a * -b;
```

ma il lettore puo' confondersi, quindi è più sicuro dire:

```
x = a * (-b);
```

L' unario meno produce il negativo del valore. L' unario più fornisce ad una simmetria con l' unario meno, sebbene non fa realmente niente.

Gli operatori decremento e incremento (**++** e **--**) sono stati introdotti poco fa in questo capitolo. Queste sono gli unici operatori oltre quelli che riguardanti l' assegnamento che hanno effetti secondari. Questi operatori incrementano o decrementano la variabile di una unità , possono avere differenti significati secondo il tipo di dato, questo è vero specialmente con i puntatori.

Gli ultimi operatori unari sono l' indirizzo di **(&),** dereferenza (* and **->**), e gli operatori di cast in C e in C++, e **new** e **delete** in C++. L' indirizzo e la dereferenza sono usati con i puntatori, descritti in questo capitolo. Il castingè descritto in seguito in questo capitolo, e **new** e **delete** sono introdotti nel capitolo 4.

L'operatore ternario

Il ternario **if-else** è unusuale perchè ha tre operandi. Esso è veramente un operatore perchè produce un valore, dissimile dall' ordinaria dichiarazione **if-else** . Consiste in tre espressioni: se la prima espressione (seguita da un **?**) vale il **true** , l' espressione successiva a **?** è valutata ed il risultato diventa il valore prodotto dall' operatore. Se la prima espressione è **false** , la terza (successiva a **:**) è eseguita ed il suo risultato diventa il valore prodotto dall' operatore.

L' operatore condizionale puo' essere usato per i suoi effetti secondari o per il valore che produce. Ecco qui un frammento di codice che li dimostra entrambi:

```
a = --b ? b : (b = -99);
```

qui, il condizionale produce un rvalue. **a** è assegnato al valore di **b** se il risultato del decremento di **b** non è zero. Se **b** diventa zero, **a** e **b** sono entrambi assegnati a -99. **b** è sempre decrementato, ma è assegnato a -99 solo se il decremento fa diventare **b** 0. Una dichiarazione simile puo' essere usata senza la " **a =** " solo per i suoi effetti secondari :

--b ? b : (b = -99);

qui la seconda b è superflua, poichè il valore prodotto dall' operatore è inutilizzato. Una espressione è richiesta tra **?** e **:.** In questo caso, l' espressione puo' essere semplicemente una costante che puo' rendere l' esecuzione del codice un po' più veloce.

L'operatore virgola

La virgola non è limitata a separare i nomi delle variabili in definizioni multiple, tipo:

int i, j, k;

naturalmente, esso è usato anche nelle liste degli argomenti di funzione. Comunque esso puo' essere usata per separare espressioni, in questo caso produce solo il valore dell' ultima espressione. Tutto il resto dell' espressione nella lista separata dalla virgola è valutata solo per i loro effetti secondari. Questo esempio incrementa una lista di variabili e usa l' ultimo come rvalue:

```
//: C03:CommaOperator.cpp

#include <iostream>

using namespace std;

int main() {

  int a = 0, b = 1, c = 2, d = 3, e = 4;

  a = (b++, c++, d++, e++);

  cout << "a = " << a << endl;

// le parentesi sono cruciali qui. Senza di

// loro, la dichiarazione valuterà :

  (a = b++), c++, d++, e++;

  cout << "a = " << a << endl;

} ///:~
```

in generale, è meglio evitare l' uso della virgola come nient'altro che un separatore, poichè non si è abituati a verderla come un operatore.

Tranelli comuni quando si usano gli operatori

Come illustrato sopra, uno dei tranelli quando si usano gli operatori è provare ad evitare di usare le parentesi quando si è incerti su come una espressione verrà valutata (si consulti il proprio manuale locale C per l' ordine della valutazione dell' espressione).

Ecco un altro errore estremamente comune che assomiglia a questo:

```
//: C03:Pitfall.cpp

// errori di operatore


int main() {

  int a = 1, b = 1;

  while (a = b) {

    // ....

  }

} ///:~
```

la dichiarazione **a = b** varra sempre vero quando **b** non è zero. La variabile **a** è assegnata al valore di **b** ed il valore di **b** è anche prodotto dall' operatore =. In generale, si usa l' operatore equivalenza = = dentro una dichiarazione condizionale, non nell' assegnamento. Questo confonde molti programmatori (comunque, alcuni compilatori richiameranno all' attenzione il problema, che è di aiuto).

Un problema affine è usare il bitwise *and* o *or* invece delle loro controparti logiche. I bitwise *and* o *or* usano uno dei caratteri (**&** o|), mentre il logico *and* o *or* ne usano due (**&&** e ||). Esattamente come con = e = =, è facile scrivere un carattere invece di due. Un utile stratagemma mnemonico è osservare che " i bit sono più piccoli, quindi non hanno bisogno di molti caratteri nei loro operatori " .

Casting degli operatori

La parola *cast* è usata nel senso di " fondere in uno stampo" . Il compilatore cambierà automaticamente un tipo di dato in un altro se ciò è sensato. Per esempio, se si assegna un valore intero ad una variabile in virgola mobile, il compilatore chiamerà segretamente una funzione (o più probabilmente, inserirà un codice) per convertire un **int** in un **float** . Il casting permette di fare questo tipo di conversione esplicita, o di forzarla quando non accadrebbe normalmente.

Per eseguire un cast, mettere il tipo di dato desiderato (includendo tutti i modificatori) dentro le parentesi a sinistra del valore. Questo valore puo' essere una variabile, una costante, un valore prodotto da una espressione o il valor di ritorno di una funzione. Ecco qui un esempio:

```
//: C03:SimpleCast.cpp

int main() {

  int b = 200;

  unsigned long a = ( unsigned long int )b;

} ///:~
```

il casting è potente, ma puo' causare mal di testa perchè in alcune situazioni forza il compilatore a trattare il dato come se fosse (ad esempio) più grande di quanto realmente è, quindi esso occuperà più spazio in memoria, questo puo' calpestare altri dati. Ciò usualmente accade quando si fa il casting di puntatori, non quando si fanno semplici cast come quello mostrato sopra.

Il C++ ha una sintassi di cast addizionale, la quale segue la sintassi della chiamata a funzione. Questa sintassi mette le parentesi intorno all' argomento, come una chiamata a funzione, piuttosto che intorno al tipo di dato :

```
//: C03:FunctionCallCast.cpp

int main() {

  float a = float (200);

// questo è l'equivalente di :

  float b = ( float )200;

} ///:~
```

naturalmente nel caso di sopra non si avrebbe bisogno di un cast, si potrebbe solo dire **200 . f** oppure **200.of** (in effetti, ciò è tipicamente cosa il compilatore farà nella espressione di sopra). I cast sono usualmente usati con le variabili, piuttosto che con le costanti.

Cast espliciti in C++

I cambi dovrebbero essere usati con cautela, poichè quello che realmente si sta facendo è dire al compilatore " dimentica il controllo del tipo , trattalo invece come qualche altro tipo. " . Cioè, si sta introducendo un buco nel sistem del tipo del C++ ed impedendo al compilatore di dire che si sta facendo qualcosa di sbagliato con un tipo. Cosa peggiore, il compilatore crede a noi implicitamente e non esegue alcun altro controllo per controllare gli errori. Una volta iniziato il casting , ci apriremo a tutti i tipi di problemi. Infatti, qualsiasi programma che usa molti cast dovrebbe essere visto con sospetto, non importa

quante volte viene detto che "deve" essere fatto in quel modo. In generale, i cast dovrebbero essere pochi e isolati alla soluzione di specifici problemi.

Una volta capito ciò e viene presentato un programma bacato, la prima tendenza potrebbe essere quella di cercare i cast come colpevoli, ma come si localizzano i cast dello stile C? Ci sono semplici nomi dentro le parentesi e se si inizia a cercare tali cose si scoprirà che è spesso difficile distinguerli dal resto del codice.

Lo standard C++ include un' esplicita sintassi di cast che puo' essere usata per riporre completamente il vechio stile di cast in C (naturalmente, lo stile del cast in C non puo' essere bandito senza un codice di interruzione, ma gli inventori del compilatore potrebbero facilmente richiamare l' attenzione sul vecchio stile di cast). L' esplicita sintassi del cast in C è tale che si puo' facimente trovarlo, come si puo' vedere dai loro nomi :

static_cast	Per i cast " ben funzionanti" e " ragionevolmente ben funzionanti" , incluse le cose che si possono fare senza un cast(come una conversione automatica di tipo).
const_cast	Per cast con **const** e/o **volatile** .
reinterpret_cast	casting in un altro significato completamente differente . Il punto chiave è che si avrà bisogno di fare un cast all' originale per usarlo con sicurezza. Il tipo a cui si fa il casting è tipicamente usato solo rigirare i bit o per altri misteriosi propositi. Questo è il più pericoloso tra i cast.
dynamic_cast	Per un downcasting sicuro (si veda il Capitolo15).

i primi tre espliciti cast saranno descritti più esaustivamente nelle sezioni seguenti, mentre l'ultimo puo' essere dimostrato solo quando si sarà appreso di più, nel capitolo 15.

static_cast

Uno **static_cast (cambio static)** è usato per tutte le conversioni che sono ben definite. Queste includono conversioni " sicure " che il compilatore permetterebbe a noi di programmare senza un cast e convesioni meno sicure che sono tuttavia ben definite. I tipi di conversione inclusi in **static_cast** includono tipiche convesioni senza cast , conversioni con narrowing(perdita di informazioni), forzando una conversione da un **void*** , conversioni del tipo implicite, ed una navigazione statica di gerarchie di classi (poichè non si sono viste ancora le classi e l' ereditarietà , quest' ultimo argomento sarà rimandato fino al capitolo 15) :

//: C03:static_cast.cpp

```
void func( int ) {}
int main() {
  int i = 0x7fff; // valore della posizione max = 32767
  long l;
  float f;
// (1) tipica conversione priva di cast:
  l = i;
  f = i;
// funziona anche:
  l = static_cast < long >(i);
  f = static_cast < float >(i);
//(2)   conversione con narrowing:
  i = l; // può perdere digit
  i = f; // può perdere informazioni
// dice "capisco," elimina i warnings :
  i = static_cast < int >(l);
  i = static_cast < int >(f);
  char c = static_cast < char >(i);
  // (3) forza una conversione dal void* :

  void * vp = &i;
// un vecchio modo produce una conversione pericolosa :
  float * fp = ( float *)vp;
// il nuovo modo   è ugualmente pericoloso :
  fp = static_cast < float *>(vp);
```

```
// (4) conversione implicita del tipo normalmente

// eseguita dal compilatore :

  double d = 0.0;

int x = d; // conversione del tipo automatica

  x = static_cast < int >(d); // più esplicita

  func(d); // conversione del tipo automatica

  func( static_cast < int >(d)); // più esplicita

} ///:~
```

nella sezione (1), si vedono i tipi di conversioni che utilizzati programmando in C, con o senza il cast. Promuovere da un **int** a un **long** o **float** non è un problema perchè l' ultimo puo' sempre contenere ogni valore che un **int** puo' contenere. Sebbene non è necessario, si puo' usare lo**static_cast(cambio static)** per mettere in rilievo queste promozioni.

L' altro modo di riconversione è mostrato in (2). Qui, si puo' perdere dati perchè un **int** non è grande come un **long** o un **float** ; non conterrebbe numeri della stessa grandezza. Perciò queste sono chiamate *conversioni con narrowing*. Il compilatore le eseguirà lo stesso, ma dimenticherà spesso di darci un warning. Si puo' eliminare questo warning ed indicare che realmente si intendeva usare un cast.

Assegnare da un **void*** non è permesso senza un cast in C++ (non come in C), come visto in (3). Questo è pericoloso e richiede che i programmatori sappiano ciò che stanno facendo. Lo **static_cast,** almeno, è più facile da localizzare che il vecchio cast standard quando si stanno cercando i bachi.

La sezione (4) del programma mostra i tipi di conversione di tipo implicito che sono normalmente eseguiti automaticamente dal compilatore. Questi sono automatici e non richiedono il casting, ma ancora lo **static_cast** mette in rilievo l' azione nel caso che si voglia rendere chiaro cosa succede o cercala in seguito.

const_cast

Se si vuole convertire da una **const** a una non **const** o da una **volatile** a una non **volatile** , si usa **const_cast** . Questo è l' unica conversione permessa con **const_cast;** se ogni altra conversione è utilizzara, essa deve essere fatto usando una espressione separata o si avrà un errore al tempo di compilazione.

```
//: C03:const_cast.cpp

int main() {

  const int i = 0;

  int * j = ( int *)&i; //   forma deprecata
```

```
 j   = const_cast < int *>(&i); // Preferita

 // non si puo' fare un cast simultaneo addizionale :

//! long* l = const_cast<long*>(&i); // Errore

 volatile int k = 0;

 int * u = const_cast < int *>(&k);

} ///:~
```

se si prende l' indirizzo di un oggetto **const** , si produce un puntatore ad un **const** , e ciò non puo' essere assegnato ad un puntatore di una non **const** senza un cast. Il vecchio stile di cast eseguirà ciò, ma il **const_cast** e l' unico appropriato da usare. Lo stesso vale per **volatile** .

reinterpret_cast

Questo è il meno sicuro dei meccanismi di casting, e l' unico che produce con più probabilità i bachi. Un **reinterpret_cast** pretende che un oggetto è un pattern di bit che puo' essere trattato (per alcuni propositi oscuri) come se fosse un tipo interamente differente. Questa è la manipolazione a basso livello dei bit per cui il C è noto. Virtualmente si avrà sempre bisogno di **reinterpret_cast** per tornare al tipo originale (o altrimenti si tratta la variabile come il tipo originale) prima di fare qualsiasi cosa con esso.

```
//: C03:reinterpret_cast.cpp

#include <iostream>

using namespace std;

const int sz = 100;



struct X { int a[sz]; };



void print(X* x) {

 for ( int i = 0; i < sz; i++)

   cout << x->a[i] << ' ';

 cout << endl << "--------------------" << endl;

}
```

```
int main() {

  X x;

  print(&x);

  int * xp = reinterpret_cast < int *>(&x);

  for ( int * i = xp; i < xp + sz; i++)

    *i = 0;

// non si puo' usare xp come un   X* a questo punto

// a meno che non si fa il cast indietro:

  print( reinterpret_cast <X*>(xp));

// in questo esempio, si puo' anche solo usare

// l'identificatore originale:

  print(&x);

} ///:~
```

in questo semplice esempio, **struct X** contiene un array di **int** , ma quando se ne crea uno nello stack come in **X x** , il valore di ogni **int** è spazzatura (ciò è mostrato usando la funzione **print( )** per visualizzare i contenuti di **struct**). Per inizializzarli, l' indirizzo di **X** è preso e cambiato in un puntatore **int** , il quale è poi agisce lungo l' array per settare ogni **int** a zero. Si noti come il più alto designato ad arrivare ad **i** è calcolato " aggiungendo " **sz** a **xp** ; il compilatore sa che si vuole realmente le locazioni di **sz** più grandi di **xp** e esso produce il corretto puntatore aritmetico per noi.

In this simple example, **struct X** just contains an array of **int** , but when you create one on the stack as in **X x** , the values of each of the **int** s are garbage (this is shown using the **print( )** function to display the contents of the **struct**). To initialize them, the address of the **X** is taken and cast to an **int** pointer, which is then walked through the array to set each **int** to zero. Notice how the upper bound for **i** is calculated by " adding " **sz** to **xp** ; the compiler knows that you actually want **sz** pointer locations greater than **xp** and it does the correct pointer arithmetic for you.

L' idea del **reinterpret_cast** è che quando lo si usa, quello che si ottiene non ha così relazione che non puo' essere usato per il proposito originale del tipo a emno che non si lo si cambi indietro. Qui, si vede il cambio in **X*** nella chiamata a stampare, ma naturalmente poichè si ha ancora l' originale identificatore si puo' anche usarlo. Ma l' xp è solo utile come un **int*** , il quale è realmente una " rinterpretazione " dell' originale **X** .

The idea of **reinterpret_cast** is that when you use it, what you get is so foreign that it cannot be used for the type' s original purpose unless you cast it back. Here, we see the cast back to an **X*** in the call to print, but of course since you still have the original identifier

you can also use that. But the **xp** is only useful as an **int*** , which is truly a " reinterpretation " of the original **X** .

Un **reinterpret_cast** spesso indica sconsigliato e non trasportabile per la programmazione, ma è disponibile quando si decide di dover usarla.

A **reinterpret_cast** often indicates inadvisable and/or nonportable programming, but it' s available when you decide you have to use it.

sizeof - un operatore a se

sizeof - an operator by itself

l' operatore **sizeof** sta da solo perchè soddisfa un bisogno unusuale. **Sizeof** ci da l' informazione circa l' ammontare della memoria allocata per gli oggetti data. Cme descritto poco fa in questo capitolo, **sizeof** dice a noi il numero di byte usati da ogni particolare variabile. Esso puo' anche dare la grandezza di un tipo data (con nessun nome di variabile):

The **sizeof** operator stands alone because it satisfies an unusual need. **sizeof** gives you information about the amount of memory allocated for data items. As described earlier in this chapter, **sizeof** tells you the number of bytes used by any particular variable. It can also give the size of a data type (with no variable name):

```
//: C03:sizeof.cpp

#include <iostream>

using namespace std;

int main() {

  cout << "sizeof(double) = " << sizeof ( double );

  cout << ", sizeof(char) = " << sizeof ( char );

} ///:~
```

per definizione, il **sizeof** di ogni tipo di char (**signed** , **unsigned** o pieno) è sempre uno, anche se la memoria utilizzata per un **char non** è realmente un byte. Per tutti gli altri tipi, il risultato è la grandezza in byte.

Si noti che **sizeof** è un operatore, non una funzione. Se lo si applica ad un tipo, deve essere usato con le parentesi nella forma mostrata sopra, ma se si applica ad una variabile si puo' usarla senza parentesi :

```
//: C03:sizeofOperator.cpp

int main() {

  int x;
```

```
  int i = sizeof x;

} ///:~
```

sizeof puo' anche darci la grandezza per i tipi di dato definiti dagli utenti. Ciò è usato in seguito nel libro.

La parola chiave asm

Questo è un meccanismo di fuga che ci permette di scrivere il codice assembler per il nostro hardware in un programma C++. Spesso si può far riferimento a variabili in C++ senza un codice assembly, che significa che si puo' comunicare facilmente con il proprio codice in C++ e limitare il codice assembly al necessario per l' efficienza o per l' uso di istruzioni speciali del processore. L' esatta sintassi che si deve usare quando si scrive in linguaggio assembly è compilatore-dipendente e puo' essere scoperto nella propria documentazione del compilatore.

operatori espliciti

Queste sono parole chiave per gli operatori logici o su bit. I programmatori non americani senza caratteri della tastiera come **&** , | , ^ , e così via, sono obbligati a usare l' orribile *trigraphs (trigrafo)* , il quale non solo è noiso da scrivere, ma oscura quando lo si legge. In C++ si rimedia con le parole chiave addizionali :

Parola chiave	Significato
and	**&&** (*and* logico)
or	|| (*or* logico)
not	**!** (NOT logico)
not_eq	**!**= (logico non-equivalente)
Bitand	**&** (bitwise *and*)
And_eq	**&**= (bitwise *and* -assegnamento)
Bitor	| (bitwise *or*)
Or_eq	|= (bitwise or-assegnamento)
Xor	^ (bitwise esclusivo-or)
xor_eq	^= (bitwise esclusivo-or-assegnamento)
Compl	~ (complemento)

Or

Not

not_eq

Bitand

& (bitwise *and*)

And_eq

&= (bitwise *and* -assegnamento)

Bitor

| (bitwise *or*)

Or_eq

|= (bitwise or-assegnamento)

Xor

^ (bitwise esclusivo-or)

xor_eq

^= (bitwise esclusivo-or-assegnamento)

Compl

Se il proprio compilatore è conforme allo Standard C++, supporta queste parole chiave.

Creazione di tipo composto

I tipi di dato fondamentali e le loro variazioni sono essenziali, ma piuttosto primitive. Il C ed il C++ forniscono di strumenti che ci permettono di comporre tipi di dato più sofisticati dai tipi di dato fondamentali. Come si vedrà , la più importante di queste è **struct** , che è la base per **class** in C++. Comunque, il modo più semplice per creare tipi più sofisticati è semplicemente dare uno pseudonimo ad un altro attraverso **typedef** .

Pseudonimi con typedef

Questa parola chiave promette più di quanto offre: **typedef** suggerisce " definizione di tipo " quando " pseudonimo" probabilmente sarebbe stata una descrizione più accurata , poichè questo è ciò che realmente fa. La sintassi è:

Typedef esistente-tipo-descrizione dello pseudonimo

si usa spesso **typedef** quando il tipo di dato diventa leggermente complicato, proprio per scrivere di meno. Ecco un **typedef** comunemente usato :

typedef unsigned long ulong;

ora se si dice **ulong** il compilatore sa che si intende **unsigned long** . Si puo' pensare che ciò potrebbe essere compiuto più facilmente usando la sostituzione del preprocessore, ma ci sono situazioni chiave nelle quali il compilatore deve essere consapevole che si sta trattando un nome come se fosse un tipo, quindi **typedef** è essenziale.

Un posto dove **typedef** torna utile è per i tipi di puntatori. Come precedentemente menzionato, se si dice :

int * x, y;

ciò realmente produce un **int*** il quale è **x** e un **int** (non un **int***) il quale è **y**. Cioè, il **' *'** si riferisce a destra non a sinistra. Comunque se si usa un **typedef** :

typedef int * IntPtr;

IntPtr x, y;

Dopo entrambi **x** e **y** sono tipi di **int*** .

Si puo' dire che è più esplicito, perciò più leggibile, evitare **typedef** per i tipi primitivi, ed effettivamente i programmi diventano rapidamente difficili da leggere quando vengono usati molti **typedef** . Tuttavia, i **typedef** diventano specialmente importanti in C quando sono usati con **struct** .

Combinare le variabili con struct

Una **struct** è un modo per raccogliere un gruppo di variabili in una struttura. Una volta creata una **struct** , dopo si possono fare molti esempi di questo " nuovo " tipo di variabile che si è inventati. Ad esempio:

```
//: C03:SimpleStruct.cpp

struct Structure1 {

  char c;

  int i;

  float f;

  double d;

};



int main() {

  struct Structure1 s1, s2;

  s1.c = 'a'; // Selezionare un elemento usando un '.'

  s1.i = 1;

  s1.f = 3.14;

  s1.d = 0.00093;

  s2.c = 'a';

  s2.i = 1;

  s2.f = 3.14;

  s2.d = 0.00093;

} ///:~
```

la dichiarazione **struct** deve terminare con un punto e virgola. In **main( )** , le due istanze di **Structure1** sono creati : **s1** e **s2.** ognuna di queste ha la loro propria versione separata di **c, i , f ,** e **d** . Quindi **s1** e **s2** rappresenta un gruppo di variabili completamente indipendenti. Per selezionare uno degli elementi senza **s1** o **s2** , si usa un **' .'** , sintassi che si è vista nel capitolo precedente quando si usano gli oggetti **class** in C++ , poichè **class** evolve dalle **struct** , questa è da dove quella sintassi è venuta fuori.

Una cosa che si noterà è l' incapacità di uso di **Structure1** (a cui è rivolto, questo è solamente richiesto dal C, non in C++). In C non si puo' dire solo **Structure1** quando si sta definendo le variabili, si deve dire **struct Structure1** . Questo è dove **typedef** diventa specialmente pratico in C:

```
//: C03:SimpleStruct2.cpp

// usare typedef con struct

typedef struct {

  char c;

  int i;

  float f;

  double d;

} Structure2;


int main() {

  Structure2 s1, s2;

  s1.c = 'a';

  s1.i = 1;

  s1.f = 3.14;

  s1.d = 0.00093;

  s2.c = 'a';

  s2.i = 1;

  s2.f = 3.14;

  s2.d = 0.00093;

} ///:~
```

usando **typedef** in questo modo, si puo' fingere (in C; si provi a rimuovere il **typedef** per il C++) che **Structure2** sia un tipo predefinito, come **int** o **float** , quando si definiscono **s1** e **s2** (ma si noti che esso ha solo la caratteristica del dato non il comportamento, che è quello che otteniamo con gli oggetti veri in C++). Si noterà che l' identificatore **struct** è stata persa via all' inizio, perchè lo scopo è creare il **typedef** . Comunque, ci sono delle volte in cui quando si puo' aver bisogno di riferirsi alla **struct** durante la sua definizione.

In questi casi, si puo' realmente ripetere il nome del **struct** come il nome del **struct** e come **typedef** :

```
//: C03:SelfReferential.cpp

// permettere ad una struct di riferirsi a se stessa


typedef struct SelfReferential {

  int i;

  SelfReferential* sr; // un giramento di testa ancora?

} SelfReferential;


int main() {

  SelfReferential sr1, sr2;

  sr1.sr = &sr2;

  sr2.sr = &sr1;

  sr1.i = 47;

  sr2.i = 1024;

} ///:~
```

se si osserva questo per un pò, si vedrà che **sr1** e **sr2** si puntano l'un l'altro ed ognuno ha un pezzo di dato.

Realmente, il nome **struct** non deve essere lo stesso del nome di **typedef** , ma è usualmente fatto in questo modo perchè tende a rendere le cose più semplici.

Puntatori e struct

Nell' esempio di sopra, tutte le **struct** sono manipolate come oggetti. Tuttavia, come qualsiasi pezzo di memoria, si puo' prendere l' indirizzo di un oggetto **struct** (come visto sopra in **SelfReferential.cpp**). Per scegliere gli elementi di un particolare oggetto **struct**, si usa un **' .'** , come visto sopra. Tuttavia, se si ha un puntatore ad un oggetto **struct** , si deve scegliere un elemento di quell' oggetto usando un differente operatore : il **' ->'** . Ecco qui un esempio :

```
//: C03:SimpleStruct3.cpp
```

```
//usare i puntatori alle struct

typedef struct Structure3 {

  char c;

  int i;

  float f;

  double d;

} Structure3;


int main() {

  Structure3 s1, s2;

  Structure3* sp = &s1;

  sp->c = 'a';

  sp->i = 1;

  sp->f = 3.14;

  sp->d = 0.00093;

  sp = &s2; // punta ad un differente oggetto struct

  sp->c = 'a';

  sp->i = 1;

  sp->f = 3.14;

  sp->d = 0.00093;

} ///:~
```

in **main( )** , il puntatore alla **struct sp** sta inizialmente puntando a **s1** ed i membri di **s1** sono inizializzati selezionandoli con ' ->' (si puo' usare lo stesso operatore per leggere questi membri). Ma poi **sp** punta a **s2** e queste variabili sono inizializzate allo stesso modo. Quindi si puo' vedere che un altro beneficio dei puntatori è che possono essere dinamicamente deviati a puntare a oggetti differenti; questo provoca più flessibilità nel proprio programma, come si apprenderà .

Per adesso, questo è tutto ciò che di cui si ha bisogno sulle **struct** , ma ci sentiremo maggiormente a nostro agio con loro (e specialmente i loro più potenti successori, le classi) più in avanti nel libro.

Chiarificare i programmi con enum

Un tipo data numerato è un modo di assegnare i nomi ai numeri, quindi dando più significato a chiunque legge il codice. La parola chiave **enum** (dal C) automaticamente enumera qualsiasi lista di identificatori che si danno assegnando i valori di 0,1,2. ecc.. Si possono dichiarare le variabili **enum** (le quali sono sempre rappresentate come valori inteir). La dichiarazione di un **enum** assomiglia ad una dichiarazione di **struct** .

Un tipo di dato enumerato è utile quando si vuole tenere conto di alcuni tipi di caratteristiche :

```
//: C03:Enum.cpp

// tenere traccia delle forme

enum ShapeType {

  circle,

  square,

  rectangle

};

// deve finire con un punto e virgola come una struct
```

```
int main() {

  ShapeType shape = circle;

// attività qui....

// adesso facciamo qualcosa basata su quale forma è
    :

  switch (shape) {

case circle:   /* cosa circolare */ break ;

    case square:   /* cosa quadrata
    */ break ;

    case rectangle:   /* cosa rettangolare
    */ break

  }
```

} ///:~

shape è una variabile di tipo dato enumerato **ShapeType** ed il suo valore è confrontato col valore nella enumerazione. Poichè **shape** è realmente solo un **int** , comunque, puo' essere qualsiasi valore che si può assegnare ad un **int** (incluso un numero negativo). Si puo' anche confrontare una variabile **int** col valore nella enumerazione.

Si deve essere consapevoli che l'esempio precedente il tipo muta ad essere un modo problematico al programma. C++ ha un modo migliore per codificare questa sorta di cosa,

la spiegazione del quale deve essere rimandata in seguito nel libro.
Se non piace il modo in cui il compilatore assegna valori, si può fare da sè:

```
enum ShapeType {
  circle = 10, square = 20, rectangle = 50
};
```

se si dà il valore ad alcuni nomi e non ad altri, il compilatore userà il prossimo valore intero. Ad esempio,

```
enum snap { crackle = 25, pop };
```

il compilatore da a **pop** il valore 26.
Si può vedere come sia più leggibile il codice quando si usa un tipo di dato enumerato. Comunque, in una certa misura ciò è ancora un tentativo (in C) per compiere le cose che noi facciamo con **class** in C++, quindi si vedrà poco **enum** in C++.

Controllo del tipo per le enumerazioni

Le enumerazioni sono abbastanza primitive, semplicemente associano valori interi con nomi, ma essi non provvedono ad un controllo del tipo. In C++ il concetto del tipo è fondamentale e ciò è vero con le enumerazioni. Quando si crea un' enumerazione con nome, effettivamente si crea un nuovo tipo proprio come si fa con una classe: il nome della nostra enumerazione diventa una parola riservata per la durata dell'unità di traduzione. In più, c'è un controllo di tipo per le enumerazioni in C++ più preciso del C. Si noterà ciò in particolare se si hanno esempi di enumerazione **color** chiamati **a**. In C si può dire **a++,** ma in C++ non si può. Cioè poichè incrementare una enumerazione è produrre due conversione di tipo, uno di loro legale in C++ e l'altro illegale. Primo, il valore dell' enumerazione è implicitamente cambiata da un **color** ad un **int**, dopo il valore è incrementato, poi l' **int** è cambiato in un **color**. In C++ ciò non è permesso, perché **color** è un tipo distinto e non equivalente ad **int**. Ciò ha senso, perché come si fa a sapere se l'incremento di **blue** sarà nella lista dei colori ? Se si vuole incrementare un **color**, esso dovrebbe essere una classe (con un'operazione di incremento) e non un **enum**, poiché la classe può essereabbastanza sicura. Ogni volta che si scrive codice che assume una conversione implicita in un tipo **enum**, il compilatore richiamerà all'attenzione su questa pericolosa attività.
Le union (descritte in seguito) hanno controlli sul tipo addizionali simili in C++.

Risparmiare la memoria con union

Qualche volta un programma gestirà differenti tipi di dato usando la stessa variabile. In questa situazione, si hanno due scelte : si può creare una **struct** contenente tutte i possibili tipi che potrebbero servire o si può usare un **union**. Un **union** accatasta tutti i tipi di dato in un singolo spazio, ne segue che la dimensione della **union** è data dal l'oggetto più grande che si è messo in essa**.** Si usi **union** per risparmiare la memoria.
Ogni volta che si piazza un valore in una **union**, il valore comincia sempre allo stesso posto all'inizio dell'**union**, ma usa solo lo spazio necessario. Quindi si crea una "super variabile " capace di detenere ogni variabile in **union**. Tutte gli indirizzi delle variabili di

union sono gli stessi (in una **class** o **struct**, gli indirizzi sono differenti).
Ecco un semplice uso di **union**. Si provi a rimuovere vari elementi e si veda che effetto esso ha sulla grandezza dell'**union**. Si noti che non ha senso dichiarare più di un esempio di un singolo tipo data in un **union** (tranne quando si sta usando un nome differente).

```
//: C03:Union.cpp
// la dimensione ed il semplice uso di una unione
#include <iostream>
using namespace std;
union Packed { // dichiarazione simile a class
  char i;
  short j;
  int k;
  long l;
  float f;
  double d;

// l'unione avrà la dimensione di un
//double, poiché quello è l'elemento più grande
};  // il punto e virgola termina una union, come una struct
int main() {
  cout << "sizeof(Packed) = "
       << sizeof(Packed) << endl;
  Packed x;
  x.i = 'c';
  cout << x.i << endl;
  x.d = 3.14159;
  cout << x.d << endl;
} ///:~
```

il compilatore esegue la giusta assegnazione secondo il membro **union** che si è scelto.
Ua volta si esegue un'assegnazione, il compilatore non si preoccupa di ciò che si fa con l'**union**. Nell'esempio sopracitato, si potrebbe assegnare un valore in virgola mobile a **x**:

```
x.f = 2.222;
```

e dopo mandarlo all'output come se fosse un **int** :

```
cout << x.i;
```

ciò produrrebbe spazzatura.

Gli array

I vettori sono un tipo composto perchè permettono di racchiudere molte variabili insieme, una dopo l'altra, sotto un singolo nome identificatore. Se si dice :

```
int a[10];
```

si crea memoria per 10 variabili **int** accatastate una su l'altra, ma senza un unico nome identificatore per ogni variabile. Invece, esse sono tutte raggruppate sotto il nome **a**.
Per accedere ad una di questi elementi del vettore (*array elements*), si usa la stessa sintassi di con parentesi quadre che si usa per definire un array:

```
a[5] = 47;
```

comunque, si deve ricordare che anche se la grandezza di **a** è 10, si scelgono gli elementi del vettore cominciando da 0 (questo è chiamato qualche volta *zero indexing*), quindi si possono selezionare solo elementi del vettore 0-9 :

```
//: C03:Arrays.cpp
#include <iostream>
using namespace std;
int main() {
  int a[10];
  for(int i = 0; i < 10; i++) {
    a[i] = i * 10;
    cout << "a[" << i << "] = " << a[i] << endl;
  }
} ///:~
```

l'accesso ad un array è estremamente veloce. Tuttavia, se l'indice supera la fine di un array,

non c'è nessuna rete di sicurezza – si andrà su altre variabili. L'altro svantaggio è che si deve definire la grandezza di un array al tempo di compilazione; se si vuole cambiare la grandezza al tempo di esecuzione non si può usare la sintassi sopracitata (il C ha un modo per creare un array dinamicamente, ma è significamente il più disordinato). Il **vector** del C++, introdotto nel capitolo precedente, produce un array simile ad un oggetto che automaticamente cambia da sè le dimensioni, quindi è solitamente la migliore soluzione se non si conosce la dimensione dell' array al tempo di compilazione.
Si possono programmare array di ogni tipo, anche di **struct** :

```
//: C03:StructArray.cpp
// un array di struct
typedef struct {
  int i, j, k;
} ThreeDpoint;
int main() {
  ThreeDpoint p[10];
  for(int i = 0; i < 10; i++) {
    p[i].i = i + 1;
    p[i].j = i + 2;
    p[i].k = i + 3;
  }
} ///:~
```

si noti come l'identificatore **i** di **struct** sia indipendente da **i** del ciclo **for**.
Per vedere che ogni elemento di un array è vicino al prossimo, si possono stampare gli indirizzi :

```
//: C03:ArrayAddresses.cpp
#include <iostream>
using namespace std;

int main() {
  int a[10];
  cout << "sizeof(int) = "<< sizeof(int) << endl;
  for(int i = 0; i < 10; i++)
    cout << "&a[" << i << "] = "
         << (long)&a[i] << endl;
} ///:~
```

quando si fa girare questo programma, si vedrà che ogni elemento è distante **int** da quello precedente. Cioè, loro sono ammassati l'uno su l'altro.

Puntatori ed array

L'identificatore di un array è diverso da quello per le variabili ordinarie. Per una cosa, un identificatore di array non è un lvalue; non si può assegnare ad esso. E' realmente solo un gancio nella sintassi con le parentesi quadre, quando si da' un nome di un array, senza parentesi quadre, quello che si ottiene è l'indirizzo iniziale di un array :

```
//: C03:ArrayIdentifier.cpp
#include <iostream>
using namespace std;
int main() {
  int a[10];
  cout << "a = " << a << endl;
  cout << "&a[0] =" << &a[0] << endl;
} ///:~
```

quando si fa partire il programma si vedrà che i due indirizzi (i quali saranno stampati in esadecimale, poichè non c'è il cast a **long**) sono gli stessi.
Quindi un modo per vedere l'identificatore dell'array è come un puntatore all'inizio dell'array di sola lettura. E sebbene non possiamo cambiare un identificatore dell'array per puntare da qualche altra parte, ma *possiamo* creare un altro puntatore e usarlo per muoverci nell'array. Infatti, la sintassi con le parentesi quadre funziona anche con

puntatori regolari :

```
//: C03:PointersAndBrackets.cpp
int main() {
  int a[10];
  int* ip = a;
  for(int i = 0; i < 10; i++)
    ip[i] = i * 10;
} ///:~
```

il fatto di dare un nome ad un array produce che il suo iniziale indirizzo tende ad essere abbastanza importante quando si vuole passare un array ad una funzione. Se si dichiara un array come un argomento della funzione, quello che realmente si dichiarerà è un puntatore. Quindi nell'esempio seguente, **func1( )** e **func2( )** effettivamente hanno le stesse liste di argomento :

```
//: C03:ArrayArguments.cpp
#include <iostream>
#include <string>
using namespace std;
void func1(int a[], int size) {
  for(int i = 0; i < size; i++)
    a[i] = i * i - i;
}
void func2(int* a, int size) {
  for(int i = 0; i < size; i++)
    a[i] = i * i + i;
}
void print(int a[], string name, int size) {
  for(int i = 0; i < size; i++)
    cout << name << "[" << i << "] = "
         << a[i] << endl;
}
int main() {
  int a[5], b[5];
// probabilmente valori spazzatura :

  print(a, "a", 5);
  print(b, "b", 5);
// inizializzazione degli array :
  func1(a, 5);
  func1(b, 5);
  print(a, "a", 5);
  print(b, "b", 5);
// si noti che gli array sono sempre modificati :
  func2(a, 5);
  func2(b, 5);
  print(a, "a", 5);
  print(b, "b", 5);
} ///:~
```

anche se **func1( )** e **func2( )** dichiarano i loro argomenti differentemente, l'uso è lo stesso dentro la funzione. Ci sono alcuni altri problemi che questo esempio rivela : gli array non possono essere passati per valore[32], cioè, non si ottiene mai automaticamente una copia locale di un array che si passa in una funzione. Quindi, quando si modifica un array, si sta modificando sempre l'oggetto esterno. Questo può essere un po' confusionario all'inizio, se ci si aspetta un passaggio per valore fornito di argomenti ordinari.
Si noti che **print( )** usa la sintassi con le parentesi quadre per gli argomenti dell'array. Anche se la sintassi del puntatore e la sintassi con le parentesi quadre sono effettivamente le stesse quando si passano gli array come argomenti, la sintassi con le parentesi quadre lo rende più chiaro al lettore che si intende questo argomento essere un array.
Si noti anche che l'argomento **size** è passato in ogni caso. Passare solamente l'indirizzo di

un array non è un'informazione sufficiente; si deve sempre essere in grado di sapere quanto grande è un array dentro la nostra funzione, così non si corre alla fine dell'array per saperlo.
Gli array possono essere di ogni tipo, incluso array di puntatori. Infatti, quando si vuole passare argomenti della riga di comando nel proprio programma, il C ed il C++ hanno una speciale lista di argomenti per **main( ),** che assomiglia a questa:

```
int main(int argc, char* argv[]) { // ...
```

il primo argomento è il numero di elementi nell'array, il quale è il secondo argomento. Il secondo argomento è sempre un array di **char***, perché gli argomenti sono passati da una riga di comando come array di caratteri (e si ricordi, un array può essere passato solo come puntatore). Ogni cluster di caratteri delimitati da spazi bianchi nella riga di comando è trasformata in un argomento dell'array separato. Il seguente programma stampa tutti gli argomenti della riga di comando procedendo lungo l'array :

```
//: C03:CommandLineArgs.cpp
#include <iostream>
using namespace std;
int main(int argc, char* argv[]) {
  cout << "argc = " << argc << endl;
  for(int i = 0; i < argc; i++)
    cout << "argv[" << i << "] = "
         << argv[i] << endl;
} ///:~
```

si noterà che **argv[0]** è il path ed il nome stesso del programma. Questo permette al programma di scoprire informazioni su di esso. Esso aggiunge anche all'array gli argomenti del programma, quindiun comune errore quando si va a prendere gli argomentti della riga di comando è afferrare **argv[0]** quando si vuole **argv[1].**
Non si è obbligati ad usare **argc** e **argv** come identificatori nel **main( )**, questi identificatori sono solo convenzioni (ma confonderanno la gente se non li si usano).
Ancora, c'è un modo alternativo per dichiarare **argv**:

```
int main(int argc, char** argv) { // ...
```

entrambi le forme sono equivalenti, ma io trovo che la versione usata in questo libro è la più intuitiva quando si legge il codice, poiché essa dice, direttamente , "questo è un array di puntatori di caratteri."
Tutto quello che si ottiene dalla riga di comando sono array di carattere; se si vuole trattare un argomento come un altro tipo, si è responsabili della conversione dentro il proprio programma. Per facilitare la conversione in numeri, ci sono alcune funzioni utili nella libreria dello Standard C, dichiarata in **<cstdlib>**. Le più semplici da usare sono **atoi( )**, **atol( ),** e **atof( )** per convertire un array di carattere ASCII in un **int**, **long,** ed un valore in virgola mobile **double**, rispettivamente. Ecco qui un esempio che usa **atoi( )** (le altre due funzioni sono chiamate allo stesso modo) :

```
//: C03:ArgsToInts.cpp
// convertire gli argomenti della riga di comando in int
#include <iostream>
#include <cstdlib>
using namespace std;
int main(int argc, char* argv[]) {
  for(int i = 1; i < argc; i++)
    cout << atoi(argv[i]) << endl;
} ///:~
```

in questo programma, si può mettere qualsiasi numero di argomenti nella riga di comando. Si noterà che il ciclo **for** comincia al valore **1** per saltare il nome del programma in **argv[0].** Ancora, se si pone un numero in virgola mobile contenente la virgola nella riga di

comando, **atoi( )** prende solo le digitazioni fino alla virgola. Se si mettono non-numeri nella riga di comando questi ritornano dietro da **atoi( )** come zero.

Esplorare il formato in virgola mobile

La funzione **printBinary( )** introdotta prima in questo capitolo è pratica per fare ricerche dentro la struttura interna di vari tipi di dato. La più interessante tra queste è il formato in virgola mobile che permette al C e al C++ di salvare numeri rappresentati valori molto grandi e molto piccoli in un limitato ammontare di spazio. Sebbene i dettagli non possono essere completamente esposti qui, i bit dentro i **float** e i **double** sono divisi dentro tre regioni: l'esponente, la mantissa ed il bit di segno;quindi esso memorizza i valori che usano una notazione scientifica. Il programma seguente permette di giocare qua e là stampando i pattern binari di vari numeri in virgola mobile in modo che si possa dedurre lo schema usato nel formato in virgola mobile del propriocompilatore (di solito questo è lo standard IEEE per i numeri in virgola mobile, ma il proprio compilatore potrebbe non seguirlo) :

```
//: C03:FloatingAsBinary.cpp
//{L} printBinary
//{T} 3.14159
#include "printBinary.h"
#include <cstdlib>
#include <iostream>
using namespace std;
int main(int argc, char* argv[]) {
  if(argc != 2) {
   cout << "si deve fornire un numero" << endl;
    exit(1);
  }
  double d = atof(argv[1]);
  unsigned char* cp =
    reinterpret_cast<unsigned char*>(&d);
  for(int i = sizeof(double)-1; i >= 0 ; i -= 2) {
    printBinary(cp[i-1]);
    printBinary(cp[i]);
  }
} ///:~
```

primo, il programma garantisce che si è dato un argomento controllando il valore di **argc**, il quale vale due se c'è un singolo argomento (vale uno se non ci sono argomenti, poiché il nome del programma è sempre il primo elemento di **argv**). Se fallisce, un messaggio vienestampato e la funzione **exit( )** della libreria dello standard C viene chiamata per terminare il programma.

Il programma prende l'argomento dalla riga di comando e converte i caratteri in **double** usando **atof( ).** Dopo il double è trattato come un array di byte prendendo l'indirizzo e cambiandolo in un **unsigned char***. Ognuno di questi byte è passato a **printBinary( )** per la visualizzazione.

Questo esempio è stato impostato per stampare i byte in un ordine tale che il bit del segno appare per primo – sulla mia macchina. Il proprio potrebbe essere differente, quindi si potrebbe ri-arrangiare il modo in cui le cose vengono stampate. Si dovrebbe essere consapevoli che i formati in virgola mobile sono non facili da comprendere; ad esempio, l'esponente e la mantissa non sono generalmente arrangiati sul byte, ma invece un numero di bit è riservato per ognuno e sono immagazzinati nella minor memoria possibile. Per vedere veramente cosa succede, si avrebbe bisogno di trovare la grandezza di ogni parte del numero (i bit di segno sono sempre un bit, ma gli esponenti e le mantisse sono di differenti grandezze) e stampare i bit in ogni parte separatamente.

Il puntatore aritmetico

Se tutto quello che si potesse fare con un puntatore che punta ad un array è trattarlo come se fosse uno pseudonimo dell'array, i puntatori ad array non sarebbero molto interessanti. Tuttavia, i puntatori sono molto più flessibili, poiché essi possono essere modificati per puntare altrove (ma si ricordi, l'identificatore dell'array non può essere modificato per puntare da qualche altra parte).
I puntatori aritmetici si riferiscono all'applicazione di alcuni degli operatori aritmetici ai puntatori. La ragione per cui il puntatore aritmetico è un soggetto separato da un ordinario aritmetico è che i puntatori devono conformarsi a speciali costrizioni in modo da funzionare correttamente Ad esempio, un comune operatore che si usa con i puntatori è ++, il quale siginifica "aggiungi uno al puntatore ". Quello che questo realmente significa è che il puntatore è cambiato per puntare"al prossimo valore ", qualunque cosa significhi. Ecco un esempio :

```
//: C03:PointerIncrement.cpp
#include <iostream>
using namespace std;
int main() {
  int i[10];
  double d[10];
  int* ip = i;
  double* dp = d;
  cout << "ip = " << (long)ip << endl;
  ip++;
  cout << "ip = " << (long)ip << endl;
  cout << "dp = " << (long)dp << endl;
  dp++;
  cout << "dp = " << (long)dp << endl;
} ///:~
```

per una esecuzione sulla mia macchina, l'output è :

```
ip = 6684124
ip = 6684128
dp = 6684044
dp = 6684052
```

quello che è interessante qui è che anche se l'operazione ++ appare essere la stessa operazione per gli entrambi **int*** e **double*,** si può vedere che il puntatore era cambiato solo di 4 byte per **int*** ma 8 byte per **double*.** Non allo stesso tempo, queste sono le grandezze **int** e **double** sulla mia macchina**.** E questoè il trucco del puntatore aritmetico : il compilatore calcola il corretto ammontare per cambiare il puntatore così esso punta al prossimo elemento nell'array(il puntatore aritmetico è solo significativo dentro gli array). Ciò funziona anche con array di **struct** :

```
//: C03:PointerIncrement2.cpp
#include <iostream>
using namespace std;
typedef struct {
  char c;
  short s;
  int i;
  long l;
  float f;
  double d;
  long double ld;
} Primitives;
int main() {
  Primitives p[10];
  Primitives* pp = p;
  cout << "sizeof(Primitives) = "
       << sizeof(Primitives) << endl;
```

```
  cout << "pp = " << (long)pp << endl;
  pp++;
  cout << "pp = " << (long)pp << endl;
} ///:~
```

L'output per una esecuzione sulla mia macchina era:

```
sizeof(Primitives) = 40
pp = 6683764
pp = 6683804
```

così si può vedere che il compilatore fa ancora la giusta cosa per puntatori a **struct** (e **class** e **union**).
Il puntatore aritmetico funziona anche con l'operatore **--**, **+,** e **-**, ma gli ultimi due operatori sono limitati : non si possono addizionare due puntatori e se si sottraggono i puntatori il risultato è il numero di elementi tra i due puntatori. Comunque, si può aggiungere o sottrarre un valore intero e un puntatore. Ecco un esempio per dimostrare l'uso dei puntatori aritmetici :

```
//: C03:PointerArithmetic.cpp
#include <iostream>
using namespace std;
#define P(EX) cout << #EX << ": " << EX << endl;
int main() {
  int a[10];
  for(int i = 0; i < 10; i++)
    a[i] = i; // Dare i valori indice
  int* ip = a;
  P(*ip);
  P(*++ip);
  P(*(ip + 5));
  int* ip2 = ip + 5;
  P(*ip2);
  P(*(ip2 - 4));
  P(*--ip2);
P(ip2 - ip); // Produce un numero di elementi
} ///:~
```

esso comincia con un altra macro, ma questa usa una caratteristica del preprocessore chiamata *stringizing* (implementata con il segno '**#**' prima di una espressione) che prende qualsiasi espressione e la cambia in un array di caratteri. Ciò è abbastanza utile, poiché permette all'espressione di essere stampata, seguita da due punti, seguita dal valore dell'espressione. In **main( )** si può vedere l'utile stenografia che è stata prodotta.
Sebbene le versioni di prefisso e di suffisso di ++ e - sono valide con i puntatori, solo le versioni di prefisso sono usati in questo esempio perchè sono applicate prima che i puntatori siano dereferenziati nell'espressione di sopra, così ci permettono di vedere gli effetti delle operazioni. Si noti che solo i valori interi erano stati aggiunti o sottratti; se due puntatori fossero combinati il compilatore non l'avrebbe permesso.
Ecco l'output per il programma di sopra :

```
*ip: 0
*++ip: 1
*(ip + 5): 6
*ip2: 6
*(ip2 - 4): 2
*--ip2: 5
```

in tutti i casi, i puntatori arimetici risultano nel puntatore che è adattato a puntare al "posto giusto ", basato sulla grandezza degli elementi a cui sta puntando.
Se il puntatore aritmetico sembra un pò opprimente all'inizio, non ci si deve preoccupare. La maggior parte delle volte che si avrà bisogno di creare array ed indici dentro essi con **[ ]**,

di solito si avrà bisogno al massimo di ++ e --. Il puntatore aritmetico è generalmente riservato per programmi più intelligenti e complessi, e molti dei contenitori nella libreria standard C++ nascondono la mggioranza di questi dettagli intelligenti quindi non ci si deve preoccupare di essi.

Suggerimenti per debugging

In uno ambiente ideale, si ha un eccellente debugger disponibile che rende facilmente trasparente il comportamento del proprio programma così si possonopresto scoprire gli errori. Tuttavia, molti debugger hanno punti deboli, e questi richiederanno di includere frammenti di codice nel proprio programma per aiutarci a capire cosa succede. In più, può capitare di sviluppare in un ambiente (come un sistema embedded, dove io ho speso i miei anni formativi) che non ha debugger disponibili e forse un feedback molto limitato (es. un display LED ad una riga). In questi casi si diventa creativi nei modi in cui si scoprono e si visualizzano informazioni sull'evoluzione del proprio programma. Questa sezione suggerisce alcune tecniche per fare ciò.

I flag di debugging

Se si cabla il codice di debugging nel programma, si può incorrere in alcuni problemi. Si inizia ad ottenere troppe informazioni, che rendono difficile isolare i bug. Quando si pensa di aver trovato il bug si inizia a strappare via il codice di debugging, per capire poi che c'è bisogno di rimetterlo di nuovo. Si possono risolvere questi problemi con due tipi di flag : i flag di debug del preprocessore e i flag di debug a runtime.

I flag di debugging del preprocessore

Usando le **#define** per defiire uno o più flag di debugging (preferibilmente in un header file) si può provare a testare un flag usando una dichiarazione **#ifdef** ed includere il codice di debugging condizionalmente. Quando si pensa che il nostro debugging è finito, si può semplicemente usare **#undef** per il/i flag ed il codice automaticamente sarà rimosso (e si ridurrà la grandezza e il tempo di esecuzione del proprio eseguibile)**.**

E' meglio decidere sui nomi per le flag di debug prima che si inizi a costruire il proprio progetto così i nomi saranno consistenti. I flag del preprocessore sono tradizionalmente distinguibili dalle variabili scrivendoli con sole lettere maiuscole. Una comune nome di flag è semplicemente **DEBUG** (ma sia faccia attenzione a non usare **NDEBUG**, il quale è riservato in C). La sequenza di istruzioni potrebbe essere:

```
#define DEBUG // Probabilmente in un header file
//...
#ifdef DEBUG // controllare per vedere se il flag è definito
/* qui il codice di debugging */
#endif // DEBUG
```

La maggior parte delle implementazioni del C e C++ permetteranno di utilizzare **#define** e **#undef** dalla riga di comando del compilatore, quindi si può ricompilare il codice ed inserire l'informazione di debugging con un singolo comando (preferibilmente con il makefile, uno strumento che sarà descritto tra breve). Si controlli la propria documentazione per dettagli.

Le flag di debugging a tempo di esecuzione

In alcune situazioni è più conveniente accendere o spegnere i flag del debug durante l'esecuzione del programma, settandoli quando il programma inizia usando la riga di comando. I programmi grandi sono noiosi da ricompilare solo per inserire il codice di debug.

Per accendere e spegnere il codice di debug dinamicamente, creare le flag **bool** :

```
//: C03:DynamicDebugFlags.cpp
#include <iostream>
#include <string>
using namespace std;
// i flag di debug non sono necessariamente globali :
bool debug = false;
int main(int argc, char* argv[]) {
  for(int i = 0; i < argc; i++)
    if(string(argv[i]) == "--debug=on")
      debug = true;
  bool go = true;
  while(go) {
    if(debug) {

 // codice di debug qui
      cout << "Debugger è on!" << endl;
    } else {
      cout << "Debugger è off." << endl;
    }
    cout << "Stato debugger [on/off/quit]: ";
    string reply;
    cin >> reply;
   if(reply == "on") debug = true; // accendere
    if(reply == "off") debug = false; // spegnere
    if(reply == "quit") break; // Fuori dal 'while'
  }
} ///:~
```

questo programma permette di accendere e spegnere i flag di debugging fino a che si scrive "quit" ("esci") per dire che se si vuole uscire. Si noti che esso richiede che siano scritte dentro le parole intere, non solo lettere (si può accorciarle ad una lettera se lo si desidera). Ancora, l'argomento della riga di comando può opzionalmente essere usata per accendere il debugging all'inizializzazione – questo argomento può apparire in ogni posto della riga di comando, poiché il codice di inizializzazione in **main( )** guarda tutti gli argomenti. La dimostrazione è abbastanza semplice :

```
string(argv[i])
```

questo prende l'array di carattere **argv[i]** e crea una **string**, la quale dopo può essere facilmente confrontata con la parte di destra di ==. Il programma di sopra cerca l'intera stringha **--debug=on** . Così si può anche cercare **--debug=** e dopo vedere cosa c'è dopo, per fornire più opzioni. Il Volume 2 (disponibile da *www.BruceEckel.com*) dedica un capitolo alla classe **string** dello Standard C++.
Sebbene un flag di debugging è una delle poche aree dove ha molto senso usare una variabile globale, non c'è niente che dice che esso deve essere in quel modo. Si noti che la variabile è scritta in lettere minuscole per ricordare al lettore che non è un flag del preprocessore.

Convertire le variabili e le espressioni in stringhe

Quando si scrive il codice di debugging è noioso scrivere le espressioni di stampa consistenti in array di caratteri contenenti il nome della variabile seguita dalla variabile. Fortunatamente, lo Standard C include l'operatore *stringize* '**#**', usato inizialmente in questo capitolo. Quando si pone una # prima di un argomento in una macro del preprocessore, il preprocessore cambia quel argomento in un array di caratteri. Questo, combinato con il fatto che gli array di caratteri con nessun intervento di punteggiatura, permette di rendere una macro più utile al fine di stampare i valori di variabili durante il debugging :

```
#define PR(x) cout << #x " = " << x << "\n";
```

se si stampa la variabile **a** chiamando la macro **PR(a)**, si avrà lo stesso effetto del codice :

```
cout << "a = " << a << "\n";
```

questo stesso processo funziona con intere espressioni. Il programma seguente usa una macro per creare una stenografia che stampa l'espressione stringized e dopo valuta l'espressione e ne stampa il risultato :

```
//: C03:StringizingExpressions.cpp
#include <iostream>
using namespace std;
#define P(A) cout << #A << ": " << (A) << endl;
int main() {
  int a = 1, b = 2, c = 3;
  P(a); P(b); P(c);
  P(a + b);
  P((c - a)/b);
} ///:~
```

si può vedere come una tecnica come questa diventa presto indispensabile, specialmente se si non hanno debugger (o si devono usare ambienti di sviluppomultipli): si può anche inserire un **#ifdef** per definire **P(A)** come "nulla" quando si vuole rimuovere il debugging.

La macro assert( ) del C

Nello standard file principale **<cassert>** si troverà **assert( )**, il quale è una utile macro di debugging. Quando si usa **assert( )**, si dà un argomento che è una espressione che si "asserisce essere vera ". Il preprocessore genera il codice che proverà l'asserzione. Se l'asserzione non è vera, il programma si fermerà dopo aver prodotto un messaggio di errore evidenziano quale asserzione è fallita. Ecco qui un esempio:

```
//: C03:Assert.cpp
// uso della macro di debugging assert()
#include <cassert>  // Contiene la macro
using namespace std;
int main() {
  int i = 100;
assert(i != 100); // Fallisce
} ///:~
```

il comando originato nello standard C, è così disponibile nel header file **assert.h**.
Quando è finito il debugging, si può rimuovere il codice generato dalla macro ponendo la riga :

```
#define NDEBUG
```

nel programma prima dell'inclusione di **<cassert>**, o definendo NDEBUG nella riga di comando del compilatore. NDEBUG è un flag usato in **<cassert>** per cambiare il modo in cui il codice è generato dalle macro.
In seguito in questo libro, si vedranno alcune alternative più sofisticate di **assert( )**.

Indirizzi della funzione

Una volta che una funzione viene compilata e caricata nel computer per essere eseguita, esso occupa un pezzo di memoria. Questa memoria, e quindi la funzione, ha un indirizzo. Il C non è mai stato un linguaggio per sbarrare l'entrata dove gli altri avevano paura di leggere. Si possono usare gli indirizzi della funzione con i puntatori proprio come si possono usare gli indirizzi delle variabili. La dichiarazione e l'uso dei puntatori a funzione appaiono un pò opachi inizialmente, ma seguono il formato del resto del linguaggio.

Definire un puntatore a funzione

Per definire un puntatore ad una funzione che non ha nessun argomento e nessun valore di ritorno, si scrive:

```
void (*funcPtr)();
```

quando si cerca una definizione complessa come questa, il modo migliore per attaccarla è iniziare nel mezzo e trovare l'uscita. “Cominciando dal mezzo “ significa iniziare dal nome della variabile, che è **funcPtr** . “Trovare l'uscita“ significa cercare a destra per l’oggetto più vicino (nessuno in questo caso; la parentesi destra ci ferma bruscamente), poi guardare a sinistra (un puntatore denotato con l’asterisco), poi guardare a destra(una lista di argomenti vuota indicante una funzione che non prende argomenti), poi guardare a sinistra (**void**, che indica la funzione che non ha valore di ritorno). Questo movimento a destra e a sinistra funziona con la maggio parte delle dichiarazioni.
Per esaminare, “cominciare in mezzo” (“**funcPtr** è un ...”), si vada a destra (niente la - ci si fermerà dalla parentesi destra), andare a sinistra e trovare "*" (“ … punta a … “), andare a destra e trovare lista di argomenti vuota (“ … la funzione che non prende argomenti …”), andare a sinistra e trovare il **void** (“**funcPtr** è un puntatore ad una funzione che non prende argomenti e restituisce un **void** “).
Ci si può chiedere perché ***funcPtr** richiede le parentesi. Se non le si usava, il compilatore avrebbe visto :

```
void *funcPtr();
```

si potrebbe dichiarare una funzione (che restituisce un **void***) piuttosto che definire una variabile. Il compilatore esegue lo stesso processo che facciamo noi quando capisce quale dichiarazione o definizione è supposta essere. Esso ha bisogno di queste parentesi per “sbatterci contro “ così esso torna a sinistra e trova la ‘*’, invece di continuare a destra e trovare una lista di argomenti vuota.

Definizioni e dichiarazioni complesse

Come digressione, una volta che si capito come funzionano le sintassi di dichiarazione in C e C++ si possono creare oggetti molto più complicati. Ad esempio :

```
//: C03:ComplicatedDefinitions.cpp
// definizioni complicate
/* 1. */      void * (*(*fp1)(int))[10];
/* 2. */      float (*(*fp2)(int,int,float))(int);
/* 3. */      typedef double (*(*(*fp3)())[10])();
              fp3 a;
/* 4. */      int (*(*f4())[10])();
int main() {} ///:~
```

si cammini attraverso ognuno e si usi la linea direttiva destra-sinistra. Il numero 1 dice “**fp1** è un puntatore ad una funzione che prende un argomento intero e restituisce un puntatore ad un array di 10 puntatori **void** .“
Il numero 2 dice “**fp2** è un puntatore ad una funzione che prende tre argomenti (**int**, **int,** e **float**) e restituisce un puntatore a funzione che prende un argomento intero e restituisce un **float** .”
Se si stanno creando molte definizioni complicate, si potrebbe voler usare un **typedef**. Il numero 3 mostra come un **typedef** risparmia di scrivere ndo la descrizione complicata ogni volta. Esso dice “**fp3** è un puntatore alla funzione che non prende argomenti e restituisce un puntatore ad un array di 10 puntatori a funzione che non prendono argomenti e restituisce **double**.” Esso dopo dice “**a** è uno di questi tipi **fp3**”. **Typedef** è generalmente utile per costruire descrizioni complicate da quelle semplici.
Il numero 4 è una dichiarazione di funzione invece di una definizione di variabile. Esso dice “**f4** è una funzione che restituisce un puntatore ad un array di 10 puntatori a funzione che restituisce interi.”
Raramente si avrà bisogno di dichiarazioni e definizioni complicate come queste. Comunque, se con l’esercizio non si sarà disturbati da espressione complicate che si potrebbero incontrare nella vita reale.

Usare un puntatore a funzione

Una volta definito un puntatore a funzione, si deve assegnarlo ad un indirizzo di una funzione prima di poterlo usare. Proprio come l'indirizzo di un array **arr[10]** è prodotto dal nome dell'array senza le parentesi (**arr**), l'indirizzo di una funzione **func()** è prodotto dal nome della funzione senza l'argomento lista (**func**). Si può anche usare una sintassi più esplicita **&func().** Per chiamare una funzione, si dereferenzia il puntatore allo stesso modo in cui lo si è dichiarato (si ricordi che il C ed il C++ provano sempre a rendere l'aspetto delle definizioni identico al modo in cui sono usate). Il seguente esempio mostra come un puntatore ad una funzione è definita ed usata :

```
//: C03:PointerToFunction.cpp
// definire e usare un puntatore ad una funzione
#include <iostream>
using namespace std;
void func() {
 cout << "func() chiamata..." << endl;
}
int main() {
 void (*fp)();  // definire un puntatore a funzione
  fp = func;  // Inizializzarlo
  (*fp)();    // dereferenziare chiama la funzione
  void (*fp2)() = func;  // Definire ed inizializzare
  (*fp2)();
} ///:~
```

dopo che il puntatore alla funzione **fp** è chiamato, è assegnato all'indirizzo della funzione **func()** usando **fp = func** (si noti che la lista degli argomenti manca nel nome della funzione). Il secondo caso mostra simultaneamente la definizione e l'inizializzazione.

Array di puntatori a funzioni

Uno dei più interessanti costrutti che si possono creare è un array di puntatori a funzioni. Per selezionare una funzione, si indicizza l'array e si dereferenzia il puntatore. Questo supporta il concetto di *codice a tabella* (*table-driven code*); invece di usare i condizionali e le dichiarazioni del caso, si selezionano le funzioni da eseguire basata su un variabile di stato (o una combinazione di variabili di stato). Questo tipo di approccio può essere utile se spesso si aggiungono o si eliminano funzioni dalla tabella (o se si vuole di creare o cambiare così una tabella dinamicamente).
Il seguente esempio crea alcune funzioni finte usando una macro del preprocessore, che crea un array di puntatori a queste funzioni che usano l'inizializzazione aggregata automatica. Come si può vedere, è facile aggiungere o rimuovere funzioni dalla tabella (e così, funzionalmente dal programma) cambiando poco del codice :

```
//: C03:FunctionTable.cpp
// usare un array di puntatori a funzioni
#include <iostream>
using namespace std;
// una macro per definire funzioni dummy :
#define DF(N) void N() { \

 cout << "funzione " #N " chiamata..." << endl; }
DF(a); DF(b); DF(c); DF(d); DF(e); DF(f); DF(g);
void (*func_table[])() = { a, b, c, d, e, f, g };
int main() {
  while(1) {
  cout << "premi un tasto da 'a' fino a 'g' "
       "oppure q per uscire" << endl;
    char c, cr;
    cin.get(c); cin.get(cr); // un secondo per CR
```

```
      if ( c == 'q' )
            break; // ... fuori dal while(1)
      if ( c < 'a' || c > 'g' )
        continue;
      (*func_table[c - 'a'])();
    }
  } ///:~
```

a questo punto, si potrebbe immaginare come questa tecnica possa essere utile quando si crea qualche tipo di interprete o programmi per processare liste.

Make: gestire compilazioni separate

Quando si usa una *compilazione separata* (spezzando il codice in un numero di unità di traduzione), si ha bisogno in qualche modo di compilare automaticamente ogni file e dire al linker di costruire tutti i pezzi- insieme con le librerie appropriate ed il codice di inizio in un file eseguibile. La maggior parte dei compilatori permettono di fare ciò con una singola instruzione della riga di comando. Per il compilatore C++ GNU, ad esempio, si potrebbe scrivere :

```
g++ SourceFile1.cpp SourceFile2.cpp
```

il problema con questo approccio è che il compilatore compilerà per primo ogni file individuale, nonostante che quel file ha bisogno di essere ricostruito o no. Con molti file in un progetto, può diventare proibitivo ricompilare tutto se è cambiato solo un singolo file. La soluzione a questo problema, sviluppato in Unix disponibile dovunque in alcune forme, è un programma chiamato **make**. L'utilità di **make** è che amministra tutti i file individuali in un progetto seguendo le istruzioni in un file testo chiamato **makefile**. Quando si editano alcuni dei file nel progetto e si scrive **make**, il programma **make** segue le linee direttive nel **makefile** per confrontare le data nei file del codice sorgente con le date dei file del target corrispondente, e se un file del codice sorgente è più recente del suo file target, **make** invoca il compilatore sul file del codice sorgente. **Make** ricompila solo i file del codice sorgente che erano cambiati, ed ogni altro file di codice sorgente che sono modificati. Usando **make**, non si deve ricompilare tutti i file nel proprio progetto ogni volta che si fa un cambiamento, ne si deve controllare che ogni cosa siacostruita propriamente. Il **makefile** contiene tutti i comandi per mettere il proprio progetto insieme. Imparare ad usare **make** farà risparmieremolto tempo e frustazioni. Si scoprirà anche che **make** è un tipico modo in cui si installa un nuovo software sulla macchina Linux/Unix (sebbene questi **makefile** tendono ad essere molto più complicati da quelli presenti in questo libro, e spesso automaticamente si genererà un **makefile** per la propria particolare macchina come parte del processo di istallazione).
Poichè **make** è disponibile in qualche forma virtualmente per tutti i compilatori C++ (anche se non lo è, si possono usare i **make** liberamente disponibili con qualsiasi compilatore), sarà lo strumento usato dappertutto in questo libro. Comunque, i venditori del compilatore hanno anche creato il loro tool per costruire progetti. Questi tool richiedono quali file sono nel proprio progetto e determinano tutte le relazioni tra gli stessi. Questi tool usano qualcosa di simile ad un **makefile**, generalmente chiamato file progetto (*project file)* , ma l'ambiente di programmazione mantiene questo file quindinon c'è da preoccuparsi di esso. La configurazione e l'uso dei file progetto variano da un ambiente di sviluppo all'altro, quindi si deve trovare una documentazione appropriata su come usarli (sebbene questi tool sono di solito così semplici da usare che si può apprendere giocandoci qua e la – la mia favorita forma di studio).
I **makefile** usati in questo libro dovrebbero funzionare anche se si sta usando un tool specifico.

Make activities

Quando si scrive **make** (o qualsiasi nome abbia il proprio programma "make"), il

programma **make** cerca nella corrente directory un file chiamato **makefile**, il quale si è creato se questo è il proprio progetto. Questo file elenca le dipendenze tra i file del codice sorgente. **make** guarda le date dei file. Se un file dipendente ha una data più vecchia del file da cui dipende, **make** esegue la regola data dopo la dipendenza.
Tutti i commenti nel **makefile** iniziano con # e continuano fino alla fine della riga.
Come semplice esempio, il **makefile** per un programma chiamato "hello" potrebbe contenere:

```
# un commento
hello.exe: hello.cpp
        mycompiler hello.cpp
```

ciò dice che **hello.exe**(il target) dipende da **hello.cpp** . Quando **hello.cpp** ha una data più nuova di **hello.exe** , **make** esegue la "regola" **mycompiler hello.cpp** . Potrebbero esserci dipendenze e regole multiple. Molti programmi **make** richiedono che tutte le regole comincino con una etichetta. Oltre ciò, lo spazio bianco è generalmente ignorato quindi si può formattare per la leggibilità.
Le regole non sono ristrette ad essere chiamate al compilatore; si può chiamare qualsiasi programma dentro **make**. Creando gruppi di insiemi di regole-dipendenze interdipendenti, si possono modificare i file del proprio codice sorgente, scrivere **make** ed essere certi che tutti i file interessati saranno ricostruiti correttamente.

Le macro

Un **makefile** contiene le *macro* (si noti che queste sono completamente differenti dalle macro del processore C/C++). Le macro permettono il rimpiazzo delle stringhe. I **makefile** in questo libro usano una macro per chiamare il compilatore del C++. Ad esempio :

```
CPP = mycompiler
hello.exe: hello.cpp
        $(CPP) hello.cpp
```

= è usato per identificare **CPP** come una macro, e **$** e le parentesi espandono la macro. In questo caso, l'espansione significa che la chiamata della macro **$(CPP)** sarà sostituita con la stringa **mycompiler** . Con la macro sopracitata, se si vuole cambiare in una differente chiamata al compilatore **cpp**, si cambia solo la macro in :

```
CPP = cpp
```

Si possono aggiungere alla macro le flag del compilatore, o usare le macro per aggiungere i flag del compilatore.

Le regole di suffisso

Diventa noioso dire a make di invocare il compilatore per ogni singolo file **cpp** del proprio progetto, quando si sa che è lo stesso progetto ogni volta. Poiché **make** è progettato per far risparmiare tempo, esso ha anche un modo per abbreviare le azioni, sempre che esse dipendano dai suffissi del nome del file. Queste abbreviazioni sono chiamate regole del suffisso. Una regola del suffisso è un modo per insegnare al **make** come convertire un file con un tipo di estensione (**.cpp** ad esempio) in un file con un altro tipo di estensione (**.obj** o **.exe**). Una volta che si insegna a **make** le regole per produrre un tipo di file da un altro, tutto quello che si deve fare è dire a **make** quali sono le dipendenze tra i file. Quando **make** trova un file con una data più recente di un file da cui dipende, esso usa una regola per creare un nuovo file.
La regola di suffisso dice a **make** che non ha bisogno di esplicitare le regole per costruire ogni cosa, ma invece può capire come costruire le cose basate sulla estensione del file. In questo caso esso dice "per costruire un file che termina in **exe** da uno che termina con **cpp**, invoca il seguente comando". Ecco qui a cosa assomiglia l'esempio sopracitato :

```
CPP = mycompiler
.SUFFIXES: .exe .cpp
```

```
.cpp.exe:
        $(CPP) $<
```

La direttiva **.SUFFIXES** dice a **make** che esso dovrebbe stare attento a qualsiasi delle seguenti estensioni del nome del file perché essi hanno un significato speciale per questo particolare makefile. In seguito si vede la regola di suffisso **.cpp.exe**, che dice "qui è come si converte qualsiasi file con una estensione **cpp** come in uno con un estensione **exe** " (quando il file **cpp** è più recente di quello **exe**). Come prima, la macro **$(CPP)** è usata, ma poi si vede qualcosa di nuovo. **$<**. Poichè inizia con una '**$**' , esso è una macro, ma questa è una delle macro speciali predefinite di **make**. La **$<** può essere usata solo nelle regole di suffisso, e significa " qualunque prerequisito innesca la regola " (qualche volta chiamata *dipendente*) ,che in questo caso traduce " il file **cpp** che ha bisogno di essere compilato. " Una volta che le regole di suffisso sono state impostate, si può semplicemente dire, ad esempio, "**make Union.exe**," e la regola di suffisso sarà immessa, anche se non c'è nessuna menzione di "Union" da qualsiasi parte nel **makefile**.

Target di Default

Dopo le macro e le regole di suffisso, **make** cerca il primo "target" in un file, e lo costruisce, a meno che si specifica differentemente. Quindi il seguente **makefile** :

```
CPP = mycompiler
.SUFFIXES: .exe .cpp
.cpp.exe:
        $(CPP) $<
target1.exe:
target2.exe:
```

se si scrive solo "make", allora **target1.exe** sarà costruito (usando la regola del suffisso di default) perché questo è il primo target che **make** incontra. Per costruire **target2.exe** si dovrebbe dire esplicitamente '**make target2.exe**'. Ciò diventa noioso, così normalmente si crea un target "finto" di default che dipende da tutto il resto dei target, come questo :

```
CPP = mycompiler
.SUFFIXES: .exe .cpp
.cpp.exe:
        $(CPP) $<
all: target1.exe target2.exe
```

qui, **'all'** non esiste e non c'è alcun file chiamato**'all'**, quindi ogni volta che si scrive **make**, il programma vede **'all'** come primo target nella lista (e quindi il target di default), dopo esso vede che **'all'** non esiste quindi deve controllare tutte le dipendenze. Quindi esso guarda al **target1.exe** e (usando la regola di suffisso) guarda sia che (1) **target1.exe** esiste e se (2) **target1.cpp** è più recente di **target1.exe,** e così esegue la regola di suffisso (se si fornisce una regola esplicita per un obiettivo particolare, quella regola è invece usata). Dopo esso passa al prossimo file nella lista dei target di default (tipicamente chiamata **'all'** per convenzione, ma si può chiamarla con qualunque nome) si può costruire ogni eseguibile del proprio progetto semplicemente scrivendo **'make'**. In più , si possono avere altre liste dei target non di default che fanno altre cose - ad esempio, si può settarla così scrivendo **'make debug'** che ricostruisce tutti i file che hanno all'interno il debugging.

I makefile di questo libro

Usando il programma **ExtractCode.cpp** del Volume 2 di questo libro, tutti i codici elencati in questo libro sono automaticamente estratti da lla versione di testo ASCII di questo libro e posti in subdirectory secondo i loro capitoli. In più, **ExtractCode.cpp** crea diversi **makefile** in ogni subdirectory (con differenti nomi) così ci si può semplicemente spostare dentro questa subdirectory e scrivere **make -f mycompiler.makefile** (sostituendo '**mycompiler**' con il nome del proprio compilatore, la flag **'-f'** dice "usa quello che segue come un **makefile**"). Infine, **ExtractCode.cpp** crea un **makefile** "master" nella directory principale dove i file del libro sono stati espansi, e questo **makefile** ha origine in ogni subdirectory chiama make con l'appropriato **makefile**.

Questo modo in cui si può compilare tutti i codici invocando un singolo comando **make**, e il processo si fermerà ogni volta che il nostro compilatore è incapace di gestire un file particolare (si noti che un compilatore conforme al C++ dovrebbe essere capace di compilatore tutti i file in questo libro). Poiché gli esempi di **make** variano da sistema a sistema, solo le caratteristiche base e più comuni sono usate nei **makefile** generati.

Un esempio di makefile

Come menzionato, lo strumento di estrazione del codice **ExtractCode.cpp** automaticamente genera i **makefile** per ogni capitolo. A causa di ciò, i makefile per ogni capitolo non saranno posti nel libro (tutti i makefile sono confezionati con il codice sorgente, il quale si può scaricare da *www.BruceEckel.com*). Comunque, è utile vedere un esempio di un **makefile**. Quello che segue è una versione ridotta di uno che era automaticamente generato per questo capitolo dallo strumento di estrazione del libro. Si troverà più di un **makefile** in ogni sotto-directory (essi hanno nomi differenti; si invoca un nome specifico con '**make-f**'). Questo è per il C++ GNU :

```
CPP = g++
OFLAG = -o
.SUFFIXES : .o .cpp .c
.cpp.o :
  $(CPP) $(CPPFLAGS) -c $<
.c.o :
  $(CPP) $(CPPFLAGS) -c $<
all: \
  Return \
  Declare \
  Ifthen \
  Guess \
  Guess2
# il resto dei file per questo capitolo non sono mostrati
Return: Return.o
  $(CPP) $(OFLAG)Return Return.o
Declare: Declare.o
  $(CPP) $(OFLAG)Declare Declare.o
Ifthen: Ifthen.o
  $(CPP) $(OFLAG)Ifthen Ifthen.o
Guess: Guess.o
  $(CPP) $(OFLAG)Guess Guess.o
Guess2: Guess2.o
  $(CPP) $(OFLAG)Guess2 Guess2.o
Return.o: Return.cpp
Declare.o: Declare.cpp
Ifthen.o: Ifthen.cpp
Guess.o: Guess.cpp
Guess2.o: Guess2.cpp
```

La macro CPP è settata al nome del compilatore. Per usare un differente compilatore, si può editare il **makefile** o cambiare il valore della macro nella riga di comando, come segue :

```
make CPP=cpp
```

si noti, comunque che **ExtractCode.cpp** ha uno schema automatico per costruire automaticamente i **makefile** per i compilatori addizionali.
La seconda macro **OFLAG** è un flag che è usato per indicare il nome del file di output. Sebbene molti compilatori assumono automaticamente come file di output quello che ha lo stesso nome di base del file in input, per altri questo non vale(come i compilatori in Linux/Unix, che per default crea un file chiamato **a.out**).
Si può vedere che ci sono due regole di suffisso qui, uno per i file **cpp** e uno per i file **.c** (nel caso qualsiasi codice sorgente di C abbia bisogno di essere compilato). Il target di default è **all** ed ogni riga per questo target è "continuato" usando il segno \, fino a **Guess2**,

il quale è l'ultimo nella lista e quindi non ha il segno\. Ci sono molti più file in questo capitolo, ma solo questi sono mostrati qui per ragione di brevità.
Le regole di suffisso si prendono cura di creare i file oggetti (con una estensione **.o**) dai file **cpp**, ma in generale si ha bisogno di specificare esplicitamente le regole per creare l'eseguibile, perché normalmente l'eseguibile è creato dal collegamento di molti file oggetti ed il **make** non può indovinare quali questi sono. Ancora, in questo caso (Linux/Unix) non c'è nessuna estensione standard per gli eseguibili quindiuna regola di suffisso non funziona per queste semplici situazioni. Dunque, si vedono tutte le regole per la costruzione dell'eseguibile esplicitamente specificate.
Questo **makefile** prende la via assolutamente più sicura di usare le caratteristiche minime di **make** ; esso usa i concettibase di **make** e i target e le dipendenze, come pure le macro. Questo modo virtualmente assicura il funzionamento come quanti più programmi **make** possibili. Esso tende a produrre un **makefile** più grande, ma ciò non è un male poiché è automaticamente generato da **ExtractCode.cpp**.
Ci sono molte caratteristiche di **make** che questo libro non userà, come pure le più nuove e più intelligenti versioni e variazioni di **make** con le scorciatoie avanzate che possono far risparmiare molto tempo. La propria documentazione locale forse descrive le ulteriori del proprio particolare **make**, e si può imparare di più sul **make** da *Managing Projects with Make* di Oram e Talbott (O'Reilly, 1993). Ancora, se il proprio fornitore di compilatore non fornisce un **make** o usa un **make** non standard, si può trovare il make di GNU per virtualmente ogni piattaforma esistente cercando su Internet gli archivi di GNU (dei quali ce ne sono molti).

Sommario

Questo capitolo è un viaggio abbastanza intenso attraverso tutti le fondamentali caratteristiche della sintassi del C++, la maggior parte dei quali sono ereditate dal C e sono in comune con esso(e risultano il vanto della retro compatibilità del C++ con il C). Sebbene alcune caratteristiche del C++ sono state qui introdotte, questo viaggio è per primo inteso per persone pratiche nella programmazione, e semplicemente che hanno bisogno di un' introduzione alla sintassi base del C e del C++. Se si è già un programmatore C, forse si è anche visto una o due cose sul C qui che non erano familiari, oltre alle caratteristiche del C++ che erano per la maggior parte nuove. Comunque, se questo capitolo sembra ancora un po' opprimente, si dovrebbe andare al corso su CD ROM *Thinking in C: Foundations for C++ and Java* (il quale contiene letture, esercizi, e soluzioni guidate), che è allegato al libro, e anche disponibile al sito *www.BruceEckel.com*

---

[30] si noti che tutte le convenzioni sembrano terminare dopo l'accordo che qualche tipo di indentazione prende posto. La contesa tra stili di formattazione è infinita. Vedere l'appendice A per la descrizione dello stile del codice di questo libro.
[31] grazie a Kris C. Matson per aver suggerito questo esercizio sull'argomento
[32] a meno che si prenda l'approccio veramente severo che "in C/C++ tutti gli argomento passati sono per valore, e il 'valore' di un array è quello che è prodotto da un identificatore di array: è un indirizzo." Ciò può essere visto come vero dal punto di vista del linguaggio assemblatore, ma io non credo che aiuti quando si prova a lavorare con concetti di alto livello. L'addizione dei riferimenti in C++ rende gli argomenti " tutti i passaggi per valore" più confusionari, al punto dove io credo è più di aiuto pensare in termini di " passaggio di valore" contro " passaggio di indirizzi ".

4: Astrazione dei dati

Il C++ è uno strumento di miglioramento della produttività. Perché mai si dovrebbe fare lo sforzo (ed è uno sforzo, al di là di quanto facilmente si tenti di effettuare la transizione

per cambiare un certo linguaggio che già si conosce e che è produttivo, con un nuovo linguaggio con la prospettiva immediata che per un certo tempo sia meno produttivo, fino a che non ci si è fatta la mano? Perché si è giunti al convincimento che si avranno grandi vantaggi usando questo nuovo strumento.

In termini di programmazione, maggiore produttività significa che meno persone possono produrre programmi più grandi, più complessi e in meno tempo. Ci sono certamente altri problemi quando si va verso la scelta di un nuovo linguaggio, come l'efficienza (la natura di questo linguaggio causerà ritardi e un codice "gonfio"?), la sicurezza (il linguaggio ci aiuterà ad essere sicuri che il nostro programma faccia sempre quello che abbiamo progettato e gestirà gli errori in modo decente?) e la manutenzione (il linguaggio ci aiuterà a creare codice facile da capire, modificare ed estendere?). Questi sono dei fattori certamente importanti e che verranno esaminati in questo libro.

Ma produttività grezza significa anche che per scrivere un programma, prima si dovevano occupare tre persone per una settimana, e ora se ne occupa una per un giorno o due. Questo aspetto tocca diversi livelli di economie: noi programmatori siamo contenti perché ci arriva una botta di energia quando si costruisce qualcosa, il nostro cliente (o il nostro capo) è contento perché il prodotto è fatto più velocemente e con meno persone, e l'utente finale è contento perché il prodotto è più economico. La sola via per ottenere un massiccio incremento di produttività è sfruttare il codice di altri programmatori, cioè usare librerie.

Una libreria è semplicemente del codice che qualcun altro ha scritto e impacchettato assieme. Spesso, il pacchetto minimo è costituito da un singolo file con un'estensione **.lib** e uno o più file header che dicono al compilatore cosa c'è nella libreria (per esempio, nomi di funzioni). Il linker sa come cercare nel file di libreria e come estrarre l'appropriato codice. Ma questo è solo uno dei modi per consegnare una libreria. Su piattaforme che occupano diverse architetture, come Linux/Unix, spesso il solo modo sensato di consegnare una libreria è con il codice sorgente, cosicché esso possa essere riconfigurato e ricompilato sul nuovo target.

Perciò, le librerie sono probabilmente il modo più importante per migliorare la produttività, ed una delle mete primarie del C++ è quella di renderne più facile l'uso. Ciò sottointende che c'è qualcosa di abbastanza complicato nell'uso delle librerie scritte in C. La comprensione di questo fattore vi farà capire a fondo il modello del C++ e come usarlo.

Una piccola libreria in stile C

Normalmente, una libreria parte come un insieme di funzioni, ma chi ha già usato librerie C di terze parti, sa che vi si trova molto di più: oltre a comportamento, azioni e funzioni di un oggetto, ci sono anche le sue caratteristiche (peso, materiale,ecc..), che vengono rappresentate dai dati. E quando si deve gestire un insieme di caratteristiche in C, è molto conveniente raggrupparle in una struttura **struct**, specialmente se volete rappresentare

più oggetti simili fra loro nel vostro spazio del problema. Quindi, potete costruire una istanza di questa struttura per ognuna di tali oggetti.

La maggior parte delle librerie C si presenta, quindi, come un insieme di strutture e un insieme di funzioni che agiscono su quelle strutture. Come esempio di un tale sistema, consideriamo un tool di programmazione che agisce come un array, ma le cui dimensioni possono essere stabilite runtime, cioè quando viene creato. Chiameremo la struttura **Cstash**. Sebbene sia scritta in C++, ha lo stesso stile che avrebbe se fosse scritta in C.

```
//: C04:CLib.h
// Header file per una libreria in stile  C
// Un'entità tipo array tipo creata  a runtime

typedef struct CStashTag {
  int size;      // dimensione di ogni elemento
int quantity; // numero di elementi allocati
  int next;     // indice del primo elemento vuoto

 // array di byte allocato dinamicamente:

  unsigned char* storage;
} CStash;

void initialize(CStash* s, int size);
void cleanup(CStash* s);
int add(CStash* s, const void* element);
void* fetch(CStash* s, int index);
int count(CStash* s);
void inflate(CStash* s, int increase);
///:~
```

Un nome di variabile come **CstashTag**, generalmente è usato per una struttura nel caso in cui si abbia bisogno di riferirsi all'interno della struttura stessa. Per esempio, quando si crea una lista concatenata (ogni elemento nella lista contiene un puntatore all'elemento successivo), si necessita di un puntatore alla successiva struttura, così si ha bisogno di un modo per identificare il tipo di quel puntatore all'interno del corpo della struttura. Vedrete anche che, quasi universalmente, per ogni struttura in una libreria C, il simbolo **typedef** viene usato come sopra, in modo tale da poter trattare una struttura come se fosse un nuovo tipo e poter quindi definire istanze di quella struttura:

```
CStash A, B, C;
```

Il puntatore **storage** è un **unsigned char***. Un **unsigned char*** è il più piccolo pezzo di memoria che un compilatore C può gestire: la sua dimensione è dipendente dall'implementazione della macchina che state usando, spesso ha le dimensioni di un byte, ma può essere anche più grande. Si potrebbe pensare che, poiché **Cstash** è progettata per contenere ogni tipo di variabile, sarebbe più appropriato un **void***. Comunque, lo scopo non è quello di trattare questa memoria come un blocco di un qualche tipo sconosciuto, ma piuttosto come un blocco contiguo di byte.

Il codice sorgente contenuto nel file di implementazione (il quale, se comprate una libreria commerciale, normalmente non viene dato, – avreste solo un compilato **obj** o **lib** o **dll**) assomiglia a questo:

```
//: C04:CLib.cpp {O}
// Implementazione dell'esempio di libreria in stile C
// Dichiarazione della struttura e delle funzioni:
#include "CLib.h"
#include <iostream>
#include <cassert>
using namespace std;
// Quantità di elementi da aggiungere
// quando viene incrementata la allocazione:
const int increment = 100;

void initialize(CStash* s, int sz) {
  s->size = sz;
  s->quantity = 0;
  s->storage = 0;
  s->next = 0;
}

int add(CStash* s, const void* element) {
  if(s->next >= s->quantity) // E' rimasto spazio sufficiente?
inflate(s, increment);
  // Copia dell'elemento nell'allocazione,
// partendo dal primo elemento vuoto:
int startBytes = s->next * s->size;
  unsigned char* e = (unsigned char*)element;
  for(int i = 0; i < s->size; i++)
    s->storage[startBytes + i] = e[i];
  s->next++;
  return(s->next - 1);
// numero d'indice
}

void* fetch(CStash* s, int index) {
 // Controllo dei limiti dell'indice:
assert(0 <= index);
  if(index >= s->next)
    return 0; // per indicare la fine
 // produciamo un puntatore all'elemento desiderato:
return &(s->storage[index * s->size]);
}

int count(CStash* s) {
  return s->next; // Elementi in CStash
}

void inflate(CStash* s, int increase) {
  assert(increase > 0);
  int newQuantity = s->quantity + increase;
  int newBytes = newQuantity * s->size;
  int oldBytes = s->quantity * s->size;
  unsigned char* b = new unsigned char[newBytes];
  for(int i = 0; i < oldBytes; i++)
    b[i] = s->storage[i]; // Copia della vecchia allocazione nella nuova
delete [](s->storage); // Vecchia allocazione
s->storage = b; //Puntiamo alla nuova memoria
s->quantity = newQuantity;
}

void cleanup(CStash* s) {
  if(s->storage != 0) {
```

```
    cout << "freeing storage" << endl;
    delete []s->storage;
   }
} ///:~
```

La funzione **initialize()** compie le inizializzazioni necessarie per la struttura **CStash** impostando le variabili interne con valori appropriati. Inizialmente, il puntatore **storage** è impostato a zero, cioè: inizialmente, nessuno spazio di memoria allocato.

La funzione **add()** inserisce un elemento in **CStash** nella successiva locazione di memoria disponibile. Per prima cosa effettua un controllo per vedere se c'è dello spazio residuo utilizzabile, e se non c'è espande lo spazio di memoria utilizzando la funzione **inflate()** descritta più sotto.

Poiché il compilatore non conosce il tipo specifico della variabile che deve essere immagazzinata (la funzione tratta un **void***), non è possibile effettuare solo un assegnamento, che sarebbe certamente la cosa conveniente da fare, invece si deve copiare la variabile byte per byte. Il modo più diretto per fare la copia, è tramite un array indicizzato. Tipicamente, vi sono già dati di tipo byte nella memoria allocata **storage**, e il riempimento è indicato dal valore di **next**. Per partire con il giusto offset di byte, la variabile **next** viene moltiplicata per la dimensione di ogni elemento (in byte) per ottenere **startBytes**. Dopodiché, l'argomento **element** viene castato ad un **unsigned char*** in modo da poter essere indirizzato byte per byte e copiato nello spazio disponibile **storage**, e **next** viene quindi incrementato in modo da indicare sia la prossima zona di memoria utilizzabile, che il numero d'indice dove il valore è stato immagazzinato, così che possa essere recuperato usando il numero d'indice con la funzione **fetch()**.

La funzione **fetch()** controlla che l'indice non ecceda i limiti e restituisce l'indirizzo della variabile desiderata, calcolata usando l'argomento **index**. Poiché **index** indica il numero di elementi di offset in **CStash**, per ottenere l'offset numerico in byte, si deve moltiplicare **index** per il numero di byte occupati da ogni elemento. Quando questo offset viene usato per l'indicizzazione in **storage** usando array indicizzati, non si ottiene l'indirizzo, bensì il byte a quell'indirizzo. Per ottenere l'indirizzo, si deve usare operatore **&,** "indirizzo-di".

La funzione **count()**, a prima vista, può sembrare strana a un esperto programmatore C. Sembra, in effetti, un grosso pasticcio fatto per fare una cosa che è più facile fare a mano. Se, per esempio, si ha una struttura **CStash** chiamata **intStash**, parrebbe molto più diretto scoprire quanti elementi ha usando **intStash.next** invece di fare una chiamata di funzione (che è overhead) come **count(&intStash)**. Comunque, se in futuro si vorrà cambiare la rappresentazione interna di **CStash**, e quindi anche il modo con cui si calcola il conteggio, l'interfaccia della chiamata di funzione permette la necessaria flessibilità. Ma, purtroppo, la maggior parte dei programmatori non si sbatterà per scoprire il miglior design per utilizzare la vostra libreria; guarderanno com'è fatta la **struct** e preleveranno direttamente il valore di **next**, e possibilmente cambieranno anche **next** senza il vostro permesso. Ah, se solo ci fosse un modo per il progettista di librerie per avere un controllo migliore sopra cose come questa (Sì, è un'anticipazione!).

Allocazione dinamica della memoria.

Non si conosce mai la quantità massima di memoria di cui si ha bisogno per la struttura **CStash**, così la memoria puntata da **storage** è allocata prendendola dallo *heap*. Lo heap è un grande blocco di memoria che viene usato per allocare runtime quantità più piccole di

memoria. Si usa lo heap quando non si conoscono le dimensioni della memoria di cui si avrà bisogno mentre si sta scrivendo il programma (solo in fase di runtime si scopre, per esempio, che si ha bisogno di allocare spazio di memoria per 200 variabili **aeroplano** invece di 20). Nel C Standard, le funzioni di allocazione dinamica della memoria includono **malloc()**, **calloc()**, **realloc()** e **free()**. Invece di chiamate le funzioni di libreria, comunque, il C++ ha un accesso più sofisticato (sebbene più semplice da usare) alla memoria dinamica, il quale è integrato nel linguaggio stesso attraverso parole chiave come **new** e **delete**.

La funzione **inflate()** usa **new** per ottenere una quantità di spazio più grande per **CStash**. In questa situazione, la memoria si potrà solo espandere e non restringere, e la funzione **assert()** garantirà che alla **inflate()** non venga passato un numero negativo come valore del parametro **increase**. Il nuovo numero di elementi che può essere contenuto (dopo che è stata completata la chiamata a **inflate()**), è calcolato e memorizzato in **newQuantity**, e poi moltiplicato per il numero di byte per elemento per ottenere **newBytes**, che rappresenta il numero di byte allocati. In questo modo siamo in grado di sapere quanti byte devono essere copiati oltre la vecchia locazione; **oldBytes** è calcolata usando la vecchia **quantity**.

L'allocazione della memoria attuale, avviene attraverso un nuovo tipo di espressione, che implica la parola chiave **new**:

```
new unsigned char[newBytes];
```

L'uso generale di **new**, ha la seguente forma:

new Type;

dove **Type** descrive il tipo di variabile che si vuole allocare nello heap. Nel nostro caso, si vuole un array di **unsigned char** lungo **newBytes**, così che si presenti come **Type**. E' possibile allocare anche qualcosa di semplice come un **int** scrivendo:

```
new int;
```

e sebbene tale allocazione venga raramente fatta, essa è tuttavia formalmente corretta.

Una espressione new restituisce un puntatore ad un oggetto dell'esatto tipo richiesto. Così, se si scrive **new Type**, si ottiene un puntatore a **Type**. Se si scrive **new int**, viene restituito un puntatore ad un intero, e se si vuole un array di **unsigned char**, verrà restituito un puntatore al primo elemento dell'array. Il compilatore si assicurerà che venga assegnato il valore di ritorno dell'espressione **new** a un puntatore del tipo corretto.

Naturalmente, ogni volta che viene richiesta della memoria, è possibile che la richiesta fallisca se non c'è sufficiente disponibilità di memoria. Come si vedrà in seguito, il C++ possiede dei meccanismi che entrano in gioco se non ha successo l'operazione di allocazione della memoria.

Una volta che la nuova memoria è allocata, i dati nella vecchia allocazione devono essere copiati nella nuova; questa operazione è realizzata di nuovo attraverso l'indicizzazione di un array, copiando in un ciclo un byte alla volta. Dopo aver copiato i dati, la vecchia allocazione di memoria deve essere rilasciata per poter essere usufruibile per usi futuri da

altre parti del programma. La parola chiave **delete** è il complemento di **new**, e deve essere usata per rilasciare ogni blocco memoria allocato in precedenza con una **new** (se ci si dimentica di usare **delete**, la zona di memoria interessata rimane non disponibile, e se questa cosiddetto meccanismo di *memoria persa* (*memory leak*) si ripete un numero sufficiente di volte, il programma esaurirà l'intera memoria disponibile). In aggiunta, quando si cancella un array si usa una sintassi speciale, ed è come se si dovesse ricordare al compilatore che quel puntatore non punta solo a un oggetto, ma ad un gruppo di oggetti: si antepone al puntatore da eliminare una coppia vuota di parentesi quadre:

```
delete [ ] myArray;
```

Una volta che la vecchia allocazione è stata eliminata, il puntatore a quella nuova può essere assegnato al puntatore che si usa per l'allocazione; la quantità viene aggiornata al nuovo valore e la funzione **inflate()** ha così esaurito il suo compito.

Si noti che la gestione dello heap è abbastanza primitiva, grezza. Lo heap cede blocchi di memoria e li riprende quando viene invocato l'operatore **delete**. Non c'è alcun servizio inerente alla compattazione dello heap, che lo comprima per liberare blocchi di memoria più grandi (un servizio di deframmentazione). Se un programma alloca e libera memoria di heap per un certo periodo, si rischia di avere uno heap frammentato che ha sì pezzi di memoria liberi, ma nessuno dei quali sufficientemente grandi da permettere di allocare lo spazio di memoria richiesto in quel momento. Un compattatore di heap complica un programma, perché sposta pezzi di memoria in giro per lo heap, e i puntatori usati dal programma non conserveranno i loro propri valori! Alcuni ambienti operativi hanno al proprio interno la compattazione dello heap, ma richiedono l'uso di handle speciali per la memoria (i quali possono essere temporaneamente convertiti in puntatori, dopo aver bloccato la memoria, in modo che il compattatore di heap non possa muoverli) al posto di veri puntatori. Non è impossibile costruire lo schema di un proprio compattatore di heap, ma questo non è un compito da prendere alla leggera.

Quando si crea una variabile sullo stack, durante la compilazione del programma, l'allocazione di memoria per la variabile viene automaticamente creata e liberata dal compilatore. Il compilatore sa esattamente quanta memoria è necessaria, e conosce anche il tempo di vita di quella variabile attraverso il suo scope. Con l'allocazione dinamica della memoria, comunque, il compilatore non sa di quanta memoria avrà bisogno e, quindi, non conosce neppure il tempo di vita di quella allocazione. Quindi, l'allocazione di memoria non può essere pulita automaticamente, e perciò il programmatore è responsabile del rilascio della memoria con la procedura di **delete**, la quale dice al gestore dello heap che può essere nuovamente usata alla prossima chiamata a **new**. Il luogo più logico dove fare la pulizia della memoria nella libreria, è la funzione **cleanup()**, perché è lì che viene compiuta l'intera procedura di chiusura di tutte le operazioni ausiliarie.

Per provare la libreria, sono state create due strutture di tipo **CStash**. La prima contiene interi e la seconda un array di 80 caratteri.

```
//: C04:CLibTest.cpp
//{L} CLib
// Test della libreria in stile C
#include "CLib.h"
#include <fstream>
#include <iostream>
#include <string>
#include <cassert>
```

```
using namespace std;

int main() {
    // Definiamo le  variabili all'inizio
    // del blocco, come in C:
  CStash intStash, stringStash;
  int i;
  char* cp;
  ifstream in;
  string line;
  const int bufsize = 80;
   // Ora, ricordiamoci di inizializzare le variabili:
  initialize(&intStash, sizeof(int));
  for(i = 0; i < 100; i++)
    add(&intStash, &i);
  for(i = 0; i < count(&intStash); i++)
    cout << "fetch(&intStash, " << i << ") = "
         << *(int*)fetch(&intStash, i)
         << endl;
  // (Holds) Creiamo una stringa di 80 caratteri:
  initialize(&stringStash, sizeof(char)*bufsize);
  in.open("CLibTest.cpp");
  assert(in);
  while(getline(in, line))
    add(&stringStash, line.c_str());
  i = 0;
  while((cp = (char*)fetch(&stringStash,i++))!=0)
    cout << "fetch(&stringStash, " << i << ") = "
         << cp << endl;
  cleanup(&intStash);
  cleanup(&stringStash);
} ///:~
```

Seguendo il formalismo del C, tutte le variabili sono create all'inizio dello scope di **main()**. Naturalmente, successivamente ci si deve ricordare di inizializzare le variabili **CStash** nel blocco chiamando **initialize()**. Uno dei problemi con le librerie C, è che si deve indicare accuratamente a chi userà la libreria, l'importanza delle funzioni di inizializzazione e di pulizia. Se queste funzioni non vengono chiamate, si va incontro a grossi problemi. Sfortunatamente, l'utente non si chiede sempre se l'inizializzazione e la pulizia siano obbligatori e quale sia il modo corretto di eseguirli; inoltre, anche il linguaggio C non prevede meccanismi per prevenire cattivi usi delle librerie.

La struttura **intStash** viene riempita con interi, mentre la **stringStash** con array di caratteri; questi array di caratteri sono ottenuti aprendo **ClibTest.cpp**, un file di codice sorgente: le linee di codice vengono lette e scritte dentro un'istanza di **string** chiamata **line**, poi usando la funzione membro **c_str()** di **string,** si produce un puntatore alla rappresentazione a caratteri di **line**.

Dopo che ogni **Stash** è stata caricata, viene anche visualizzata. La **intStash** viene stampata usando un ciclo **for**, che termina quando la **fetch()** restituisce zero per indicare che è uscito dai limiti.

Si noti anche il cast aggiuntivo in:

```
cp = (char*)fetch(&stringStash, i++)
```

a causa del controllo rigoroso dei tipi che viene effettuato in C++, il quale non permette di assegnare semplicemente **void*** a ogni altro tipo (mentre in C è permesso).

Cattive congetture.

C'è però un argomento più importante che deve essere capito prima di affrontare in generale i problemi nella creazione di una libreria C. Si noti che il file header **CLib.h** *deve* essere incluso in ogni file che fa riferimento a **CStash**, perché il compilatore non può anche indovinare a cosa assomiglia quella struttura. Comunque il compilatore *può* indovinare a cosa assomiglia una funzione; questo suona come una qualità del C, ma poi si rivela per esserne uno dei principali tranelli.

Sebbene si dovrebbero dichiarare sempre le funzioni, includendo un file header, la dichiarazione di funzione non è obbligatoria in C. In C (ma non in C++) è possibile chiamare una funzione che non è stata dichiarata. Un buon compilatore avvisa suggerendo di dichiarare prima la funzione, ma il C Standard non obbliga a farlo. Questa è una pratica pericolosa, perché il compilatore C può assumere che una funzione che è stata chiamata con un argomento di tipo **int** abbia una lista di argomenti contente interi, anche se essa può in realtà contenere un **float**. Come si vedrà, questa pratica può produrre degli errori difficili da trovare in fase di debugging.

Ogni file separato di implementazione C (cioè con estensione **.c**), è una ***unità di traslazione***, cioè il compilatore elabora separatamente ognuna di tali unità, e mentre la sta elaborando è conosce questa sola unità e nient'altro. Così, ogni informazione che viene fornita attraverso l'inclusione dei file header, risulta di grande importanza, poiché determina la conoscenza del compilatore sul resto del programma. Le dichiarazioni nei file header risultano essere particolarmente importanti, perché in qualsiasi posto essi vengano inclusi, faranno in modo che lì il compilatore sappia cosa fare. Se, per esempio, si ha una dichiarazione in un file header che dice **void func(float)**, il compilatore sa che se si chiama la funzione **func** con un argomento **int**, dovrà convertire **int** in **float** e passarlo come argomento (questa operazione è detta ***promotion***). Senza la dichiarazione di funzione, il compilatore C assumerebbe semplicemente l'esistenza di una funzione **func(int)**, non verrebbe fatta la promotion e il dato errato verrebbe tranquillamente passato a **func()**.

Per ogni unità di traslazione, il compilatore crea un file oggetto, con estensione **.o** oppure **.obj** o qualcosa di simile. I file oggetto, insieme al codice necessario per l'avvio, devono essere riuniti dal linker in un file programma eseguibile, e tutti i riferimenti esterni dovranno essere risolti durante l'operazione di linking. Per esempio, nel file **CLibTest.cpp** sono state dichiarate (cioè al compilatore è stato detto a cosa assomigliano) e usate funzioni come **initialize()** e **fetch()**, ma queste funzioni non sono state definite, lo sono altrove, in **CLib.cpp**. Quindi le chiamate in **CLib.cpp** sono riferimenti esterni. Il linker, quando mette insieme tutti i file oggetto, prendere i riferimenti esterni non risolti e li risolve trovando gli indirizzi a cui essi effettivamente si riferiscono, e gli poi indirizzi vengono messi nel programma eseguibile per sostituire i riferimenti esterni.

E' importante rendersi conto che, in C, i riferimenti esterni che il linker cerca, sono semplicemente nomi di funzioni, generalmente con anteposto il carattere underscore ('_'). Quindi, tutto quello che il linker deve fare è associare il nome di una funzione dove è chiamata, con il corpo della funzione stessa nel file oggetto. Se, accidentalmente, un programmatore fa una chiamata che il compilatore interpreta come **func(int)**, e in

qualche altro file oggetto c'è un corpo di funzione per **func(float)**, il linker vedrà **_func** in un posto e **_func** nell'altro, e riterrà essere tutto OK. La funzione **func(),** nel luogo dove viene invocata, spingerà un **int** sullo **stack**, ma lì il corpo di funzione di **func()** si aspetta che ci sia un **float** sullo **stack**. Se la funzione compie solo una lettura del valore e non tenta di fare un assegnamento, allora lo **stack** non scoppierà. In tal caso, infatti, il valore **float** che legge dallo **stack** potrebbe anche essere sensato, il che è addirittura peggio, perché così è ancora più difficile trovare il baco.

Cosa c'è di sbagliato?

Noi siamo straordinariamente adattabili, anche in situazioni nelle quali, forse, non vorremmo adattarci. Lo stile della libreria **CStash** è stata un fattore di base per i programmatori C, ma se la si guarda per un momento, si può notare come sia piuttosto, come dire?... goffa. Quando la si usa, si deve passare l'indirizzo della struttura ad ogni singola funzione della libreria. Quando si legge il codice, il meccanismo della libreria si mescola con il significato delle chiamate di funzione, provocando confusione quando si prova a capire cosa sta succedendo.

Comunque, uno dei più grandi ostacoli nell'uso delle librerie in C, è il problema del ***conflitto di nomi*** (***name clashes***). Il C ha un singolo spazio di nomi per la funzioni, cioè, quando il linker cerca un nome di funzione, lo cerca in una singola lista master. In più, quando il compilatore sta lavorando su una unità di traslazione, lo fa lavorando solo con una singola funzione con un dato nome.

Ora, supponiamo di aver comprato due librerie da due differenti aziende, e che ogni libreria abbia una struttura che deve essere inizializzata e pulita, ed entrambe le ditte costruttrici decidono che **initialize()** e **cleanup()** sono ottimi nomi per le relative funzioni. Se si includono entrambi i file header in un singolo file di traslazione, cosa fa il compilatore C? Fortunatamente, il C dà un errore, dicendo che c'è un errato accoppiamento di tipi (***type mismatch***) nelle due differenti liste di argomenti delle funzioni dichiarate. Ma anche se i file header non vengono inclusi negli stessi file di traslazione, lo stesso si avranno problemi nell'attività del linker. Un buon linker rileverà la presenza di un conflitto di nomi, ma altri linker prendono il primo nome di funzione che trovano, cercando lungo la lista di file oggetto nell'ordine a loro dato nella lista di link.

In entrambi i casi, non è possibile usare due librerie C che contengano funzioni con nomi identici. Per risolvere il problema, i venditori spesso aggiungono una sequenza di caratteri specifica all'inizio di tutti i nomi di funzione. Così, **initialize()** diventa **CStash_initialize ()** e così via, che è una pensata sensata.

Adesso, è venuto il momento per fare il primo passo verso la creazione di classi in C++.

I nomi delle variabili all'interno di una struttura non sono in conflitto con i nomi delle variabili globali; e così, perché non trarne vantaggio per i nomi di funzione, quando quelle funzioni operano su una particolare struttura? Cioè, perché non costruire funzioni membro di una struttura?

L'oggetto base.

Il primo passo è proprio questo: le funzioni C++ possono essere messe all'interno di strutture come *"funzioni membro"*. Di seguito viene mostrato il codice per una struttura **CStash** di tipo C convertita a una **CStash** di tipo C++:

```
// C04:CppLib.h
// La libreria in stile C convertita in C++

struct Stash {
  int size;      // Dimensione di ogni elemento
  intquantity; // Numero di elementi allocati
  int next;      // Indice del primo elemento vuoto
// Array di byte allocati dinamicamente:
unsigned char* storage;
 // Le funzioni membro!
void initialize(int size);
  void cleanup();
  int add(const void* element);
  void* fetch(int index);
  int count();
  void inflate(int increase);
}; ///:~
```

Per prima cosa, si noti che non ci sono **typedef**. Invece di usare **typedef**, il compilatore C++ trasforma il nome della struttura in un nuovo nome di tipo valido all'interno del programma (proprio come **int**, **char**, **float** e **double** sono nomi di tipo).

Tutti i dati membro sono esattamente come prima, ma ora le funzioni sono all'interno del corpo della struttura. In più, si noti che è stato rimosso il primo argomento presente in tutte le funzioni della versione C della libreria, cioè il puntatore a CStash. In C++, invece di costringere il programmatore a passare l'indirizzo della struttura come primo argomento di tutte le funzioni che operano sulla struttura, l'operazione viene fatta dal compilatore in modo del tutto trasparente. Ora, i soli argomenti passati alle funzioni, sono relativi a quello che la funzione fa, non al meccanismo interno della funzione.

E' importante rendersi conto che il codice della funzione è effettivamente lo stesso della precedente versione C della libreria. Il numero di argomenti è lo stesso (anche se non è più esplicitato, il passaggio dell'indirizzo della struttura c'è ancora), e c'è un solo corpo di funzione per ogni funzione. Cioè, solo perché si dice :

Stash A, B, C;

ciò non significa avere differenti funzioni **add()** per ogni variabile.

Così, il codice che è stato generato è quasi identico a quello che si avrebbe scritto per la versione C della libreria. Curiosamente, questo include i suffissi che si sarebbero fatti per produrre **Stash_initialize()**, **Stash_cleanup()**, e così via.

Quando il nome di funzione è all'interno della struttura, il compilatore fa effettivamente la stessa cosa. Perciò, **initialize()** all'interno della struttura **Stash** non collide con una funzione chiamata **initialize()** all'interno di qualsiasi altra struttura, oppure con una funzione globale con lo stesso nome. Sebbene la maggior parte delle volte non serva, qualche volta si ha la necessità di poter precisare che questa specifica **initialize()**

appartiene alla struttura **Stash**, e non a qualche altra struttura, e in particolare questo serve quando si deve definire la funzione, perché in tal caso si necessita di specificare pienamente cosa essa sia. Per ottenere questa specificazione completa, il C++ ha un operatore (**::**) chiamato ***operatore di risoluzione della visibilità*** (scope resolution operator), così chiamato perché adesso i nomi possono avere differenti visibilità: visibilità globale o all'interno di una struttura. Per esempio, per specificare la funzione membro **initialize()**, che appartiene a **Stash**, si dice:

Stash::initialize(int size).

Vediamo come si usa l'operatore di risoluzione della visibilità nelle definizioni di funzione:

```
//: C04:CppLib.cpp {O}
// Libreria C convertita in C++
// Dichiariamo strutture e funzioni:
#include "CppLib.h"
#include <iostream>
#include <cassert>
using namespace std;
// Quantità di elementi da aggiungere
// quando viene incrementata la allocazione:
const int increment = 100;

void Stash::initialize(int sz) {
  size = sz;
  quantity = 0;
  storage = 0;
  next = 0;
}

int Stash::add(const void* element) {
  if(next >= quantity)// E' rimasto spazio sufficiente?
    inflate(increment);
 // Copia dell'elemento nell'allocazione,
 // partendo dal primo spazio vuoto:
int startBytes = next * size;
  unsigned char* e = (unsigned char*)element;
  for(int i = 0; i < size; i++)
    storage[startBytes + i] = e[i];
  next++;
  return(next - 1); // numero d'indice
}

void* Stash::fetch(int index) {
  // Controllo dei limiti dell'indice:
  assert(0 <= index);
  if(index >= next)
    return 0; // Per indicare la fine
 // produciamo un puntatore all'elemento desiderato:
  return &(storage[index * size]);
}

int Stash::count() {
  return next; // Numero di elementi in CStash
}

void Stash::inflate(int increase) {
  assert(increase > 0);
  int newQuantity = quantity + increase;
```

```
  int newBytes = newQuantity * size;
  int oldBytes = quantity * size;
  unsigned char* b = new unsigned char[newBytes];
  for(int i = 0; i < oldBytes; i++)
    b[i] = storage[i]; // Copia della vecchia allocazione nella nuova
  delete []storage; // Vecchia allocazione
  storage = b; // Puntiamo alla nuova memoria
  quantity = newQuantity;
}

void Stash::cleanup() {
  if(storage != 0) {
    cout << "freeing storage" << endl;
    delete []storage;
  }
} ///:~
```

Ci sono diverse altre differenze tra il C e il C++. Primo, le dichiarazioni nei file header sono esplicitamente *richieste* dal compilatore. In C++ non si può chiamare una funzione senza prima dichiararla, e in caso contrario il compilatore emetterà un messaggio di errore. Questo è un modo importante per essere sicuri che le chiamate di funzione siano consistenti tra il punto dove sono chiamate e il punto dove sono definite. Rendendo obbligatoria la dichiarazione della funzione prima di poterla chiamare, il compilatore C++ si assicura virtualmente che l'utente lo faccia includendo il file header. Se lo stesso file header viene incluso anche nel file dove le funzioni sono definite, il compilatore controlla che la dichiarazione nel file header e la definizione di funzione combacino. Questo significa che il file header diventa un luogo ottimale per le dichiarazioni di funzione e assicura che le funzioni siano usate in modo consistente in tutte le unità di traslazione del progetto.

Naturalmente, le funzioni globali possono sempre essere dichiarate a mano in ogni luogo dove sono definite e usate (cosa così tediosa da diventare molto spiacevole.) Comunque, le strutture devono essere sempre dichiarate prima che siano definite e usate, e il luogo più conveniente dove mettere la definizione di una struttura è in un file header, tranne quelle che sono intenzionalmente nascoste in un file.

Si noti che tutte le funzioni membro sembrano uguali a quando erano funzioni C, tranne per la risoluzione della visibilità e per il fatto che il primo argomento che si trovava nella versione C della libreria, non è più esplicito (c'è ancora, naturalmente, perché la funzione deve essere in grado di lavorare su una particolare variabile struct). Notiamo, però, all'interno della funzione membro, che anche la selezione del membro se n'è andata! E così, invece di dire **s->size = sz;** si dice **size = sz;** e si elimina la tediosa scrittura **s->**, che in ogni caso non aggiunge in realtà nulla al significato di quello che si sta facendo. Il compilatore C++ lo fa chiaramente per noi. Infatti, prende il primo argomento "segreto" (l'indirizzo della struttura che in precedenza gli veniva passata a mano) e applica il selettore di membro ogniqualvolta ci si riferisce ad uno dei dati membro (incluse anche le altre funzioni membro) dando semplicemente il suo nome. Il compilatore cercherà prima tra i nomi della struttura locale e poi, se non lo trova, nella versione globale di quel nome. Questa caratteristica significa non solo codice più facile da scrivere, ma anche parecchio più facile da leggere.

Ma cosa avverrebbe se, per qualche ragione, si volesse essere in grado di mettere le mani sull'indirizzo della struttura? Nella versione C della libreria sarebbe stato piuttosto facile, perché ciascun primo argomento della funzione era un puntatore **s** a **CStash** (**CStash***). In C++ le cose sono ancora più conformi: c'è una parola chiave (parola riservata) speciale,

chiamata **this**, la quale produce l'indirizzo di **struct** ed è l'equivalente di **s** nella versione C della libreria. Così si può tornare al solito comportamento in stile C, dicendo:

```
this->size = Size;
```

Il codice generato dal compilatore, è esattamente lo stesso, senza che si abbia più bisogno di fare questo uso di **this**; occasionalmente, si può vedere del codice dove la gente usa esplicitamente **this->** dovunque, ma ciò non aggiunge nulla al significato del codice e indica solo un programmatore inesperto. Normalmente, non si usa spesso **this**, ma quando se ne ha bisogno, è presente (come si vedrà in alcuni degli esempi illustrati in questo libro).

C'è un'ultima cosa da dire: in C, si potrebbe assegnare un **void*** ad ogni altro puntatore in questo modo:

```
int i = 10;
void* vp = &i; // corretto sia in C che in C++
int* ip = vp; // Accettabile solo in C
```

senza che il compilatore protesti. Ma, in C++, quest'affermazione non è permessa. Perché? Perché il C non è molto rigoroso circa le informazioni sul tipo, e permette quindi di assegnare un puntatore di un tipo non specificato, ad un puntatore di tipo specificato. Non così il C++. Il tipo è un aspetto critico del C++, e il compilatore pesta i piedi quando c'è una qualsiasi violazione nelle informazioni sui tipi. Questa è sempre stata una questione importante, ma lo è specialmente in C++ perché nelle **struct**s ci sono funzioni membro. Se in C++ si potessero passare impunemente puntatori a **struct**s, allora si finirebbe per fare chiamate a funzioni membro per una struttura che non esiste logicamente per quella **struct**! Un'ottima ricetta per compiere disastri! Perciò, mentre il C++ permette l'assegnamento di ogni tipo di puntatore a un **void*** (questo era lo scopo originale di **void***, che si voleva grande a sufficienza per contenere un puntatore di qualsiasi tipo), non permetterà di assegnare un puntatore **void** ad altri tipi di puntatori. In C++ viene sempre richiesto un cast per dire sia al lettore, che al compilatore, che si vuole veramente trattare l'oggetto come il tipo di destinazione specificato.

E qui si solleva un'interessante questione. Una delle mete importanti del C++, è compilare la quantità maggiore di codice C esistente, per permettere una facile transizione al nuovo linguaggio. Questo comunque non significa che qualsiasi frammento di codice permesso in C lo sia anche in C++. Un compilatore C lascia passare parecchie cose pericolose che possono portare a errori. Per queste situazioni, invece, il compilatore C++ genera avvertimenti e messaggi di errore, e questo è molto spesso più un vantaggio che un impiccio; infatti, vi sono molte situazioni nelle quali in C si tenta inutilmente di rintracciare un errore, e non appena si ricompia il codice in C++, il compilatore identifica subito il problema! In C capita di compilare apparentemente in modo corretto un programma che poi non funziona, mentre in C++, quando un programma è stato compilato correttamente, è molto probabile che funzioni! e ciò è dovuto al fatto che è un linguaggio più restrittivo nell'uso dei tipi.

Nel seguente programma di test, è possibile vedere come, nella versione C++ di Cstash, ci siano parecchie cose nuove:

```
//: C04:CppLibTest.cpp
//{L} CppLib
// Test della libreria C++<
#include "CppLib.h"
#include "../require.h"
#include <fstream>
#include <iostream>
#include <string>
using namespace std;

int main() {
  Stash intStash;
  intStash.initialize(sizeof(int));
  for(int i = 0; i < 100; i++)
    intStash.add(&i);
  for(int j = 0; j < intStash.count(); j++)
    cout << "intStash.fetch(" << j << ") = "
    << *(int*)intStash.fetch(j)
         << endl;
  // Creiamo una stringa di 80 caratteri:
  Stash stringStash;
  const int bufsize = 80;
  stringStash.initialize(sizeof(char) * bufsize);
  ifstream in("CppLibTest.cpp");
  assure(in, "CppLibTest.cpp");
  string line;
  while(getline(in, line))
    stringStash.add(line.c_str());
  int k = 0;
  char* cp;
  while((cp =(char*)stringStash.fetch(k++)) != 0)
    cout << "stringStash.fetch("<< k << ") = "
    << cp << endl;
  intStash.cleanup();
  stringStash.cleanup();
} ///:~
```

Una delle cose che si notano, è che le variabili sono tutte definite “al volo” (vedi capitolo precedente). Cioè, sono definite in un punto qualsiasi permesso alla loro visibilità (scope), piuttosto che con la restrizione del C di metterle all’inizio dello scope.

Il codice è del tutto simile a **CLibTest.cpp**, ma quando viene chiamata una funzione membro, la chiamata avviene usando l’operatore ‘.’ di selezione del membro, preceduto dal nome della variabile. Questa sintassi risulta conveniente poiché imita la selezione di un membro dati della struttura; la differenza è che questa è una funzione membro, e quindi ha una lista di argomenti.

Naturalmente, la chiamata che il compilatore effettivamente genera, assomiglia molto di più alla funzione della libreria C originale, così, considerando i suffissi del nome (name decoration) e il passaggio di **this**, la chiamata di funzione C++ **intStash.initialize(sizeof(int), 100)** diventa qualcosa di simile a **Stash_initialize(&intStash, sizeof(int), 100)**. Se vi siete chiesti cosa mai ci sia sotto lo strato superficiale, ricordate che il compilatore C++ originale (“cfront” della AT&T) , produceva come suo output codice C, il quale poi veniva compilato dal compilatore C sottostante. Questo approccio significava che cfront poteva essere velocemente portato su ogni macchina che aveva un compilatore C: un grande contributo a diffondere rapidamente

la tecnologia dei compilatori C++. Ma, poiché il compilatore C++ doveva generare codice C, è chiaro che si doveva in qualche modo rappresentare la sintassi C++ in C (alcuni compilatori C++ permettono tuttora di produrre codice C).

C'è un'altra differenza da **ClibTest.cpp**, che è l'introduzione del file header **require.h**, un file header creato appositamente per questo libro per effettuare un controllo degli errori più sofisticato di quello dato dalla funzione **assert()**. Questo file contiene diverse funzioni, incluse quella che qui è usata, **assure()**, usata per i file: controlla se un file è stato aperto con successo, e se non lo è, comunica allo standard error che il file potrebbe non essere aperto (quindi è necessario il nome del file come secondo argomento) ed esce dal programma. Le funzioni di **require.h** rimpiazzano il codice di controllo degli errori di tipo distracting e repetitive, e in più forniscono utili messaggi di errore. Queste funzioni saranno completamente spiegate più tardi nel libro.

Cos'è un oggetto?

Ora che si è visto un esempio iniziale, è tempo di fare un passo indietro e dare un'occhiata a un po' di terminologia.

Il portare funzioni all'interno di una struttura, è il fondamento strutturale che il C++ aggiunge al C; introduce un nuovo modo di pensare le strutture: come concetti. In C, una struttura è un agglomerato di dati, un modo per impacchettare dati così che possano essere trattati in un blocco. Ma è difficile pensarlo come un qualcosa che non sia altro che una conveniente modalità di programmazione. Le funzioni che operano su quella struttura possono essere ovunque. Con le funzioni all'interno del pacchetto, la struttura diventa una nuova creatura, capace di descrivere sia caratteristiche (coma fa una struttura C) che comportamento. Il concetto di un oggetto, una entità a sé, chiusa, che può *ricordare* e *agire*, suggerisce, propone, sé stesso.

In C++, un oggetto è appunto una variabile, e la più pura definizione è: "una zona di memoria" (che è un modo più specifico di dire, "un oggetto deve avere un identificatore univoco", il quale, nel caso del C++, è un indirizzo di memoria univoco). E' un luogo dove è possibile memorizzare dati, ed è sottointeso che lì ci siano anche operazioni che possono agire su questi dati.

Sfortunatamente, quando si giunge a questo livello di terminologia, non c'è un completo accordo tra i linguaggi, anche se sono accettati abbastanza bene. Si possono anche trovare posizioni discordanti su cosa sia un linguaggio orientato agli oggetti. Ci sono alcuni linguaggi che sono *basati* sugli oggetti (object-based), cioè hanno oggetti come le "strutture con funzioni" che si sono viste fin qui, ma questa è solo una parte della faccenda che investe un linguaggio orientato agli oggetti; i linguaggi che si fermano all'impacchettamento delle funzioni all'interno delle strutture dati, sono solo linguaggi basati sugli oggetti, e non orientati agli oggetti.

Tipi di dati astratti

La capacità di impacchettare dati insieme a funzioni, permette la creazione di un nuovo tipo di dati. Questa capacità è spesso chiamata ***incapsulamento*** [33]. Un tipo di dato già esistente, può avere diversi blocchi di dati impacchettati assieme. Per esempio, un **float** è costituito da un esponente, una mantissa e un bit di segno, e gli si può dire di compiere

delle cose: aggiungere un altro **float** o un **int**, e così via: risulta quindi avere già un insieme sia di caratteristiche che di comportamenti.

La definizione di **Stash** crea un nuovo tipo di dato, su cui è possibile effettuare operazioni tramite **add()**, **fetch()**, e **inflate ()**. Per creare un'istanza di **Stash**, si scrive **Stash s**, proprio come per creare un **float** si scrive **float f**. Un tipo **Stash** possiede sia delle caratteristiche che dei comportamenti. Anche **Stash** si comporta come un vero tipo di dato built-in, ci si riferisce ad esso come ad un tipo di dato astratto, forse perché ci permette di astrarre un concetto dallo spazio del problema allo spazio della soluzione. In più, il compilatore C++ lo tratta come un nuovo tipo di dato, e se si dice che una funzione si aspetta un dato di tipo **Stash**, il compilatore si assicura che si passi uno **Stash** a quella funzione. Così, come per i tipi built-in definiti dall'utente, anche con i tipi di dati astratti c'è lo stesso livello di controllo di tipo. Qui si vede immediatamente una differenza nel modo in cui si compiono le operazioni sugli oggetti. Si scrive: **object.memberFunction(arglist)**, e significa "richiedere una funzione membro di un oggetto". Ma nel gergo object-oriented, significa anche ***"spedire un messaggio a un oggetto"***. Così. Per un **s** di tipo **Stash**, l'affermazione **s.add(&i) "spedisci un messaggio a s"** dice, "**add()** (aggiungi) questo a te stesso". E infatti, la programmazione orientata agli oggetti può essere riassunta in una singola frase: ***la programmazione orientata agli oggetti è lo spedire messaggi a oggetti***. In effetti, questo è tutto ciò che si fa: creare un gruppo di oggetti e spedire loro dei messaggi. Il trucco, naturalmente, è riuscire a capire cosa sono quegli oggetti e quei messaggi, ma una volta fatto questo lavoro, l'implementazione in C++ è sorprendentemente semplice.

Gli oggetti in dettaglio

Una domanda che spesso viene fatta nei seminari, è : "Quanto è grande un oggetto, e a cosa assomiglia?". La risposta è : "circa quello che ci si aspetta da una struttura di tipo C". Infatti, il codice che viene prodotto dal codice C per una struttura C (senza parti C++), sembra *esattamente* lo stesso del codice prodotto da un compilatore C++. Questo fatto risulta rassicurante per quei programmatori per i quali è importante il dettaglio della dimensione e della implementazione nel proprio codice, e per qualche ragione accedono direttamente alla struttura di bytes invece di usare gli identificatori (il dipendere da particolari dimensioni e implementazioni di una struttura, è una attività che non è portabile su altre piattaforme).

La dimensione di una struttura è la somma delle dimensioni di tutti i suoi membri. Talvolta, quando il compilatore implementa una **struct**, aggiunge dei bytes extra per fare in modo che i confini fra le zone di memoria assegnate ai dati siano netti, aumentando l'efficienza dell'esecuzione del programma. Nel cap. 15 si vedrà come, in alcuni casi, vengano aggiunti alla struttura dei puntatori "segreti", ma ora non è il caso di preoccuparsene.

Si può determinare la dimensione di una struttura usando l'operatore **sizeof**.Un semplice esempio:

```
//: C04:Sizeof.cpp
// Dimensioni di  struct
#include "CLib.h"
#include "CppLib.h"
#include <iostream>
using namespace std;
```

```
struct A {
  int i[100];
};

struct B {
  void f();
};

void B::f() {}

int main() {
  cout << "sizeof struct A = " << sizeof(A)
       << " bytes" << endl;
  cout << "sizeof struct B = " << sizeof(B)
       << " bytes" << endl;
  cout << "sizeof CStash in C = "
       << sizeof(CStash) << " bytes" << endl;
  cout << "sizeof Stash in C++ = "
       << sizeof(Stash) << " bytes" << endl;
} ///:~
```

Tenendo presente che il risultato dipende dalla macchina su cui gira il programma, la prima istruzione di print produce 200, poiché ogni **int** occupa due bytes. La **struct** B è qualcosa di anomalo, perché è una **struct** senza membri dati; in C è un'operazione illegale, ma in C++ è permesso perché si ha la necessità di creare una struttura il cui unico compito sia la visibilità dei nomi delle funzioni. Ancora, il risultato prodotto dal secondo print, è un qualcosa che sorprendentemente ha un valore diverso da zero. Nelle prime versioni del linguaggio, la dimensione era zero, ma quando si creano tali oggetti si crea una situazione imbarazzante: hanno lo stesso indirizzo dell'oggetto creato direttamente dopo di loro, e quindi non sono distinti. Una delle regole fondamentali nella trattazione degli oggetti, è che ogni oggetto deve avere un indirizzo univoco; quindi le strutture senza membri dati avranno sempre una qualche dimensione minima diversa da zero.

Gli ultimi due **sizeof**, mostrano che la dimensione della struttura in C++ è la stessa della dimensione della versione equivalente in C. Il C++ cerca di non aggiungere nessun sovrappiù che non sia strettamente necessario.

Etichetta di comportamento per i file header

Quando si crea una struttura che contiene funzioni membro, si sta creando un nuovo tipo di dato e, in generale, si vuole che questo tipo sia facilmente accessibile a chi scrive il programma e anche agli altri. Inoltre, si vuole separare l'interfaccia (la dichiarazione) dall'implementazione (la definizione delle funzioni membro), così che l'implementazione possa essere cambiata senza essere obbligati a una ricompilazione dell'intero sistema. Si raggiunge questo fine mettendo la dichiarazione di un nuovo tipo in un file header.

Quando ho imparato a programmare in C, i file header erano per me un mistero, e in molti libri dedicati al C non c'era molto interesse al riguardo; inoltre, il compilatore C non imponeva le dichiarazioni di funzione, cosicché per molto tempo i file header sono stati considerati un'utilità opzionale, tranne nel caso in cui venissero dichiarate delle strutture. In C++, invece, l'uso dei file header diventa chiaro in modo cristallino; sono virtualmente obbligatori per un sviluppo comodo di un programma; lì, è possibile inserire informazioni molto specifiche: le dichiarazioni. Il file header dice al compilatore cosa è disponibile nella libreria. Si può usare la libreria anche se si possiede solo il file header insieme al file

oggetto o al file libreria; non si ha bisogno del file **.cpp** del codice sorgente. Il file header è il luogo dove sono contenute le specifiche di interfaccia.

Sebbene non si sia obbligati dal compilatore, il miglior modo per costruire grandi progetti in C è di usare le librerie: collezioni di funzioni tra loro associate, poste all'interno dello stesso modulo oggetto o libreria, e usare poi un file header per contenere tutte le dichiarazioni per le funzioni. Questo metodo è *di rigore* in C++. E' possibile mettere qualsiasi funzione in una libreria C, ma in C++, il tipo di dato astratto determina le funzioni che vengono associate in forza del loro comune accesso ai dati in una **struct**. Ogni funzione membro deve essere dichiarata nella dichiarazione della **struct**: non è possibile metterla altrove.

L'uso delle librerie di funzioni era incoraggiato in C, ma è istituzionalizzato in C++.

L'importanza dei file header.

Quando si usa una funzione di una libreria, il C permette di ignorare il file header e di dichiarare semplicemente la funzione a mano. Nel passato, la gente qualche volta lo faceva per accelerare un pelo il compilatore ed evitandogli così il compito di aprire e includere il file (cosa non più redditizia con i compilatori moderni). Come esempio, di seguito riporto una dichiarazione estremamente "*lazzarona*" per la funzione **printf()** del C (da **<stdio.h>**):

```
printf(…);
```

L'omissione "…" indica una lista variabile di argomenti [34], che dice: **printf()** ha un qualche argomento, ognuno dei quali ha un tipo, ma si ignora quale. Soltanto prendendo un determinato argomento è possibile vederlo e accettarlo. Usando questo tipo di dichiarazione, vengono sospesi tutti i controlli di errore sugli argomenti della funzione.

Questa pratica può causare problemi oscuri, subdoli. Se si dichiara una funzione a mano, in un file si può fare un errore. Siccome il compilatore vede solo la dichiarazione fatta a mano e fatta solo in quel file, può essere in grado di adattarsi all'errore. Il programma verrà quindi linkato correttamente, ma l'uso della funzione in quel singolo file fallirà. Questo tipo di errore è molto difficile da trovare, ma si può facilmente evitare usando un file header.

Se si mettono tutte le dichiarazioni di funzione in un file header e si include quel file header sia dove viene definita la funzione che ovunque la si usi, ci si assicura una dichiarazione consistente attraverso l'intero sistema. Inoltre, ci si può assicurare che dichiarazione e definizione combacino, inserendo il file header nel file di definizione.

Se in un file C++ si dichiara una **struct**, è obbligatorio includere il file header ovunque si usi una **struct** e dove vengono definiti i membri funzione della **struct**. Se si prova a chiamare una funzione regolare, o a chiamare o definire una funzione membro senza che essa sia stata prima dichiarata, il compilatore C++ darà un messaggio di errore. Costringendo il programmatore ad un uso corretto dei file header, il linguaggio si assicura la consistenza nelle librerie, inoltre riduce gli errori con l'uso della stessa interfaccia ovunque sia necessaria.

Il file header è un contratto fra l'utente di una libreria e il suo programmatore, che descrive la sua struttura dati, specifica gli argomenti e i valori restituiti dalle funzioni chiamate.

Dice: “Questo è ciò che la mia libreria fa”. L’utente necessita di alcune di queste informazioni per sviluppare la propria applicazione, mentre il compilatore ne ha bisogno per generare il codice appropriato. L’utente che usa **struct**, include semplicemente il file header, crea le istanze di **struct** e linka il tutto nel modulo oggetto o in libreria (cioè: compila il codice).

Il compilatore fa osservare il contratto richiedendo di dichiarare tutte le strutture e le funzioni prima che siano usate e, nel caso di funzioni membro, prima che esse siano definite. Ciò implica essere costretti a mettere le dichiarazioni nel file header e a includerlo nel file dove le funzioni membro sono definite, e nei file dove sono usate. Siccome un singolo file header che descrive la libreria è incluso lungo tutto il sistema, il compilatore è in grado di assicurare la consistenza e prevenire gli errori.

Ci sono certe questioni di cui si deve essere al corrente al fine di organizzare il codice in modo appropriato e scrivere dei file header efficaci. La prima questione concerne quello che si deve mettere all’interno di un file header. La regola base è: “solo dichiarazioni”, cioè, solo informazioni al compilatore ma niente che allochi memoria attraverso la generazione di codice o la creazione di variabili. Questo perché, tipicamente, in un progetto il file header sarà incluso in diverse unità di traslazione, e se la memoria per un identificatore viene allocata in più di un posto, il linker emetterà un errore di definizione multipla (questa è, per il C++, la “regola dell’uno per le definizioni”: E’ possibile dichiarare delle cose quante volte si vuole, ma ci può essere una sola definizione per ogni cosa dichiarata).

Questa regola non è del tutto rigida: se si definisce una variabile che è “**file static**” (ha visibilità solo dentro un file) all’interno di un file header, ci saranno istanze multiple di quel dato nei moduli del progetto, ma il linker non riscontrerà collisioni [35]. Fondamentalmente, non si desidera fare nulla nel file header che possa causare una ambiguità nella fase di link.

Il problema della dichiarazione multipla.

La seconda questione legata ai file header è questa: quando si mette una dichiarazione di **struct** in un file header, è possibile che quel file header sia incluso più di una volta se il programma è complicato. Gli **Iostream**s rappresentano un buon esempio. Ogni volta che una **struct** fa un un’operazione di I/O, può includere uno o più degli header di **iostream**. Se il file **cpp** su cui si sta lavorando usa più di un tipo di **struct** (includendo tipicamente un file header per ognuna di esse), si corre il rischio di includere più di una volta lo header **<iostream>** e di ridichiarare gli **iostream**s.
Il compilatore considera la ridichiarazione di una struttura (si parla sia di **struct**s che di **class**es) come un errore, poiché se così non fosse permetterebbe l’uso dello stesso nome per tipi differenti. Per prevenire questo errore quando si includono file header multipli, si ha la necessità di costruire un qualcosa di intelligente all’interno dei file header usando il preprocessore (i file header dello Standard C++, come **<iostream>**, hanno già questo qualcosa di intelligente).
Sia il C che il C++ permettono di ridichiarare una funzione, purché le due dichiarazioni combacino, ma né uno né l’altro permettono la ridichiarazione di una struttura. Questa regola è particolarmente importante in C++, perché se il compilatore permettesse di ridichiarare una struttura e poi le due dichiarazioni differissero, quale userebbe?
Il problema della ridichiarazione emerge un po’ più in C++ che in C, perché in C++ ogni tipo di dato (strutture con funzioni) ha generalmente il suo proprio file header, e si deve includere un header in un altro se si vuole creare un altro tipo di dati che usa il primo. In ogni file **cpp** di un progetto, è probabile che vengano inclusi diversi file che includono a loro volta lo stesso file header. Durante una singola compilazione, il compilatore può

vedere lo stesso file header diverse volte. A meno che non si faccia qualcosa a riguardo, il compilatore vedrà la ridichiarazione della struttura e registrerà un errore di compilazione. Per risolvere il problema, si ha bisogno di conoscere un po' di più il preprocessore.

Le direttive del preprocessore: #define, #ifdef, #endif

Si può usare la direttiva di preprocessore **#define,** per creare dei flag nel momento della compilazione del programma. Si hanno due scelte: si può semplicemente dire al preprocessore che il flag è definito, senza specificarne un valore:

#define FLAG

oppure si può dargli un valore (che è il tipico modo di definire delle costanti in C):

#define PI 3.14159

In entrambi i casi, la label può essere testata dal preprocessore per vedere se è stata definita:

#ifdef FLAG

Questo produrrà come risultato "**true**", e il codice che segue la direttiva **#ifdef** sarà incluso nel pacchetto spedito al compilatore. Questa inclusione cessa quando il preprocessore incontra la direttiva

#endif

o

#endif // FLAG

Qualsiasi cosa che non sia un commento, messo sulla stessa linea dopo la direttiva **#endif**, è illegale, anche se qualche compilatore può accettarlo. La coppia **#ifdef** / **#endif** si può annidare una nell'altra.

Il complemento di **#define** è **#undef**, il quale farà un **#ifdef** usando la stessa variabile e producendo "**false**" come risultato. La direttiva **#undef** causerà anche lo stop dell'uso delle macro da parte del preprocessore.

Il complemento di **#ifdef** è **#ifndef**, che dà "**true**" se la label non è stata definita (questo è quello che si userà nei file header).

Nel preprocessore C ci sono altre caratteristiche utili, che si possono trovare nella documentazione allegata al particolare preprocessore.

Uno standard per i file header

Quando un file header contiene una struttura, prima di includerlo in un file **cpp**, si dovrebbe controllare se è già stato in qualche modo incluso in questo **cpp**. Questo viene fatto testando un flag del preprocessore. Se il flag non è stato messo in precedenza, significa che il tipo non è già stato dichiarato; si può allora mettere il flag (così che la struttura non possa venire ridichiarata) e dichiarare la struttura, cioè includere il file. Se invece il flag è già stato messo, allora il tipo è già stato dichiarato e non lo si deve più dichiarare. Ecco come dovrebbe presentarsi un file header:

```
#ifndef HEADER_FLAG
#define HEADER_FLAG
// La dichiarazione di Tipo deve essere inserita qui...
#endif // HEADER_FLAG
```

Come si può vedere, la prima volta che il file header è incluso, il contenuto del file header (includendo le dichiarazioni di tipo) sarà incluso dal processore. Tutte le volte seguenti che sarà incluso – in una singola unità di compilazione – la dichiarazione di tipo sarà ignorata. Il nome HEADER_FLAG può essere qualsiasi nome univoco, ma uno standard attendibile da seguire è quello di scrivere il nome del file header con caratteri maiuscoli e sostituire i punti con underscore (gli underscore iniziali sono riservati per i nomi di sistema). Un esempio:

```
//: C04:Simple.h
// Un semplice header che previene la re-definizione
#ifndef SIMPLE_H
#define SIMPLE_H

struct Simple {
  int i,j,k;
  initialize() { i = j = k = 0; }
};
#endif // SIMPLE_H ///:~
```

SIMPLE_H dopo #endif è commentato e quindi ignorato dal preprocessore, ma risulta utile per la documentazione.
Le dichiarazioni di preprocessore che prevengono la inclusioni multiple, sono spesso nominate come "***include guards***" (guardie degli include).

Namespaces negli header

Si sarà notato che in quasi tutti i file cpp di questo libro, sono presenti **using *direttive*** , normalmente nella forma:

```
using namespace std;
```

Poiché **std** è il **namespace** che circonda l'intera libreria Standard del C++, questa particolare direttiva d'uso permette ai nomi nella libreria Standard del C++ di essere usati senza qualificazione. Comunque, non si vedranno virtualmente mai direttive d'uso in un file header (almeno, non al di fuori dello scope). La ragione è che la direttiva d'uso elimina la protezione di quel particolare namespace, e l'effetto dura fino alla fine dell'unità di compilazione. Se si mette una direttiva d'uso (fuori dallo scope) in un file header, significa che questa perdita di "namespace protection" avverrà per ogni file che include questo header, il che significa, spesso, altri file header. Così, se si parte mettendo direttive d'uso nei file header, risulta molto facile finire per disattivare le direttive d'uso praticamente ovunque, neutralizzando, perciò, gli effetti benefici dei namespace.
In breve: non mettere le direttive using nei file header.

Uso degli header nei progetti

Quando si costruisce un progetto in C++, normalmente lo si crea mettendo assieme un po' di tipi differenti (strutture dati con funzioni associate). Normalmente, le dichiarazioni per ogni tipo per ciascun gruppo di tipi associati, si mettono in file header separati, e poi si definiscono le funzioni relative in una unità di traslazione. Quando poi si usa quel tipo, si deve includere il file header per eseguire correttamente le dichiarazioni.
In questo libro si seguirà talvolta questo schema, ma il più delle volte gli esempi saranno molto piccoli, e quindi ogni cosa - dichiarazioni di struttura, definizioni di funzioni, e la funzione **main()** – possono comparire in un singolo file. Comunque, si tenga a mente che nella pratica si dovranno usare sia file che file header separati.

Strutture annidate

Il vantaggio di estrarre nomi di dati e di funzioni dal namespace globale, si estende alle strutture. E' possibile annidare una struttura all'interno di un'altra struttura e pertanto tenere assieme gli elementi associati. La sintassi di dichiarazione è quella che ci si aspetterebbe , come è possibile vedere nella struttura seguente, che implementa uno stack push-down come una semplice lista concatenata, così che non esaurisca mai la memoria:

```
//: C04:Stack.h
// Struttura annidata in una lista concatenata
#ifndef STACK_H
#define STACK_H
```

```
struct Stack {
  struct Link {
    void* data;
    Link* next;
    void initialize(void* dat, Link* nxt);
  }* head;
  void initialize();
  void push(void* dat);
  void* peek();
  void* pop();
  void cleanup();
};
#endif // STACK_H ///:~
```

La **struct** annidata è chiamata **Link** e contiene un puntatore al successivo elemento **Link** della lista, e un puntatore ai dati immagazzinati in **Link**. Se il puntatore **next** è zero, significa che si è arrivati alla fine della lista.

Si noti che il puntatore **head** è definito giusto dopo la dichiarazione della **struct Link**, invece di avere una definizione separata come **Link* head**. Questa è una sintassi che viene dal C, ma qui sottolinea l'importanza del punto e virgola dopo la dichiarazione di struttura. Il punto e virgola indica la fine della lista di definizioni separate da virgola di quel tipo di struttura. (Normalmente la lista è vuota).
La struttura annidata ha la sua propria funzione **initialize()**, come tutte le strutture presentate fin'ora, per assicurare una corretta inizializzazione. **Stack** ha sia una funzione **initialize ()** che una funzione **cleanup()**, come pure una funzione **push()**, la quale ha come argomento un puntatore ai dati che si vogliono memorizzare (assume che i dati siano allocati nello **heap**), e **pop()**, la quale restituisce il puntatore **data** dalla cima di **Stack** e rimuove l'elemento in cima. Quando si usa la **pop()** su un elemento, si è responsabili della distruzione dell'oggetto puntato da **data**. Inoltre, la funzione **peek()** restituisce il puntatore data dall' elemento che è in cima, ma lascia l'elemento nello **Stack**.

Ecco le definizioni per le funzioni membro:

```
//: C04:Stack.cpp {O}
// Lista concatenata con annidamento
#include "Stack.h"
#include "../require.h"
using namespacestd;

void
Stack::Link::initialize(void* dat, Link* nxt) {
  data = dat;
  next = nxt;
}

void Stack::initialize() { head = 0; }

void Stack::push(void* dat) {
  Link* newLink = new Link;
  newLink->initialize(dat, head);
  head = newLink;
}

void* Stack::peek() {
  require(head != 0, "Stack empty");
  return head->data;
}
```

```
void* Stack::pop() {
  if(head == 0) return 0;
  void* result = head->data;
  Link* oldHead = head;
  head = head->next;
  delete oldHead;
  return result;
}

void Stack::cleanup() {
  require(head == 0, "Stack not empty");
} ///:~
```

La prima definizione è particolarmente interessante perché mostra come definire un membro in una struttura annidata: semplicemente, si usa un livello addizionale di risoluzione di scope, per specificare il nome della struttura racchiusa.
Stack::Link::initialize( ) prende gli argomenti e li assegna ai suoi membri.
Stack::initialize() imposta head a zero, così l'oggetto sa che ha una lista vuota.
Stack::push() prende l'argomento, il quale è un puntatore alla variabile di cui si vuole tenere traccia, e lo spinge su **Stack**. Per prima cosa, viene usato new per allocare memoria per **Link**, che sarà inserito in cima; poi viene chiamata la funzione **initialize()** di **Link** per assegnare valori appropriati ai membri di **Link**. Si noti che il puntatore **next** viene assegnato al head corrente; quindi head viene assegnato al nuovo puntatore **Link**. Questo in realtà mette **Link** in cima alla lista.
Stack::pop() cattura il puntatore **data** che in quel momento è in cima a **Stack**, poi porta giù il puntatore **head** e elimina quello vecchio in cima allo **Stack**, e in fine restituisce il puntatore prelevato. Quando **pop()** rimuove l'ultimo elemento, allora head diventa ancora zero, e ciò significa che lo **Stack** è vuoto.
Stack::cleanup() in realtà non fa alcuna ripulita, ma stabilisce una linea di condotta risoluta: "il programmatore client che userà l'oggetto **Stack**, è responsabile per espellere tutti gli elementi di **Stack** e per la loro eliminazione". La funzione **require()**, è usata per indicare se è avvenuto un errore di programmazione con lo **Stack** non vuoto.
Ma perché il distruttore di **Stack** non potrebbe occuparsi di distruggere tutti gli oggetti che il cliente programmatore non ha eliminato con **pop()**? Il problema è che lo **Stack** tratta puntatori **void** e, come si vedrà nel cap. 13, chiamando la **delete** per **void***, non si effettua correttamente la pulizia.

Anche il soggetto di "chi è il responsabile della gestione della memoria" non è qualcosa di semplice definizione, come si vedrà negli ultimi capitoli.

Un esempio per testare **Stack**:

```
//: C04:StackTest.cpp
//{L} Stack
//{T} StackTest.cpp
// Test di una lista concatenata annidata
#include "Stack.h"
#include "../require.h"
#include <fstream>
#include <iostream>
#include <string>
using namespace std;
```

```
int main(int argc, char* argv[]) {
  requireArgs(argc, 1);// Il nome del file è un argomento
  ifstream in(argv[1]);
  assure(in, argv[1]);
  Stack textlines;
  textlines.initialize();
  string line;
   // Leggiamo il file e immagazziniamo le linee di testo in Stack:
while(getline(in, line))
    textlines.push(new string(line));
 // Facciamo emergere le linee di testo da Stack e poi le stampiamo:
  string* s;
  while((s = (string*)textlines.pop()) != 0) {
    cout << *s << endl;
    delete s;
  }
  textlines.cleanup();
} ///:~
```

Questo esempio è simile all'esempio precedente, ma qui spingono linee da un file (come puntatore a **string**) sullo **Stack**, per poi espellerle, il che risulta nel fatto che il file viene stampato in ordine inverso. Si noti, che la funzione membro **pop()** restituisce un **void*** e questo deve essere castato a **string*** prima di essere usato. Per stampare la stringa, il puntatore deve essere dereferenziato.

Non appena **textlines** viene riempito, il contenuto di line è "clonato" per ogni **push()** con il codice **new string(line)**. Il valore restituito dall'espressione **new**, è un puntatore alla nuova stringa **string** che è stata creata e che ha copiato le informazioni da **line**. Se si era semplicemente passato l'indirizzo di **line** a **push()**, si finirà con uno **Stack** riempito con indirizzi identici, tutti che puntano a **line**. Nel prossimo libro, si imparerà un po' di più riguardo questo processo di "clonazione".

Il nome del file è preso dalla riga di comando. Per garantire che vi siano sufficienti argomenti sulla riga di comando, si veda una seconda funzione, usata dal file header **requie.h**: **requireArgs()**, la quale confronta **argc** con il numero di argomenti desiderati, e stampa un appropriato messaggio di errore ed esce dal programma se non vi sono argomenti sufficienti.

Risoluzione dello scope globale

L'operatore di risoluzione dello scope interviene nelle situazioni in cui il nome che il compilatore sceglie per default (il nome più "prossimo"), non è quello che si vuole. Per esempio, supponiamo di avere una struttura con un identificatore locale **a**, e che si voglia selezionare un identificatore globale **a** dall'interno di una funzione membro. Il compilatore sceglierebbe per default quello locale, così siamo costretti a dirgli di fare altrimenti. Quando si vuole specificare un nome globale usando la risoluzione di scope, si usa l'operatore con niente di fronte ad esso. L'esempio seguente illustra la risoluzione di scope globale sia per una variabile, che per una funzione:

```
//: C04:Scoperes.cpp
// Risoluzione di scope globale
int a;
void f() {}

struct S {
```

```
  int a;
  void f();
};

void S::f() {
  ::f();  // Altrimenti sarebbe  ricorsiva
  ::a++; // Selezioniamo la 'a' globale
  a--;       // La 'a' nello scope della struttura
}
int main() { S s; f(); }
 ///:~
```

Senza risoluzione di scope in **S::f()**, il compilatore selezionerebbe per default la versione membro di **f()** e **a**.

Sommario

In questo capitolo abbiamo trattato della svolta fondamentale del C++, e cioè: è possibile inserire funzioni nelle strutture. Questo nuovo tipo di struttura è chiamato tipo di dato astratto, e le variabili che si creano usando questa struttura, sono chiamate oggetti, o istanze, di quel tipo. Il chiamare una funzione membro di un oggetto, viene detto spedire un messaggio a quell'oggetto. L'azione primaria nella programmazione orientata agli oggetti, è lo spedire massaggi agli oggetti.

Sebbene l'impacchettare insieme dati e funzioni sia un significativo vantaggio per l'organizzazione del codice e renda le librerie più facili da usare (previene il conflitto sui nomi nascondendo il nome), c'è ancora parecchio che si può fare per ottenere una programmazione C++ più sicura. Nel prossimo capitolo si imparerà come proteggere alcuni membri di una struttura così che solo l'autore possa manipolarli. Questo stabilisce un chiaro confine tra cosa l'utente della struttura può modificare e cosa solo il programmatore può variare.

---

[33] Questo termine può essere fonte di discussione: alcuni lo usano nel senso definito in questo libro, altri per descrivere il *controllo di accesso*, che sarà discusso nel capitolo seguente.

[34] Per scrivere una definizione di funzione per una funzione che prende una vera lista di argomenti di variabili, si deve usare *varargs*, sebbene queste ultime dovrebbero essere vietate in C++. I dettagli sull'uso di varargs si trovano nei manuali di C.

[35] Comunque, nel C++ Standard, l'uso di *file static* è deprecabile.

5: Nascondere l'implementazione

Una tipica libreria C contiene una **struct** ed alcune funzioni associate per agire su essa. Si è visto come il C++ prenda funzioni che sono associate concettualmente e le renda associate letteralmente

mettendo le dichiarazioni della funzione dentro lo scope della **struct**, cambiando il modo in cui le funzioni vengono chiamate per una **struct**, eliminando il passaggio dell'indirizzo di struttura come primo argomento ed aggiungendo un nuovo tipo di nome al programma (quindi non si deve creare un **typedef** per la **struct**).

Tutto ciò aiuta ad organizzare il proprio codice e lo rende più facile da scrivere e leggere. Tuttavia ci sono altri problemi quando si creano librerie in C++, specialmente problematiche riguardanti il controllo e la sicurezza. Questo capitolo prende in esame il problema dei limiti nelle strutture.

Fissare i limiti

In qualsiasi relazione è importante avere dei limiti che sono rispettati da tutte le parti coinvolte. Quando si crea una libreria, si stabilisce una relazione con il *programmatore client* che usa quella libreria per costruire un'applicazione o un'altra libreria.

In una **struct** del C, come per la maggior parte delle cose in C, non ci sono regole. I programmatori client possono fare qualsiasi cosa vogliono e non c'è modo di forzare nessun particolare comportamento. Per esempio, nell'ultimo capitolo anche se si capisce l'importanza delle funzioni chiamate **inizialize()** e **cleanup()**, il programmatore client ha l'opzione di non chiamare quelle funzioni (osserveremo un miglior approccio nel prossimo capitolo). E anche se davvero si preferirebbe che il programmatore client non manipolasse direttamente alcuni dei membri della nostra **struct**, in C non c'è modo di prevenirlo. Ogni cosa è palese al mondo.

Ci sono due ragioni del perchè si deve controllare l'accesso ai membri. La prima serve a tenere lontane le mani del programmatore client dalle cose che non devono essere toccate, parti che sono necessarie al funzionamento interno dei tipi di dato, ma non parti dell'interfaccia di cui il programmatore client ha bisogno per risolvere i propri problemi. Questo è davvero un servizio ai programmatori client perchè essi possono facilmente capire cos'è importante per loro e cosa possono ignorare.

La seconda ragione per il controllo d'accesso è permettere al progettista della libreria di cambiare la struttura interna senza preoccuparsi di come influenzerà il programmatore client. Nell'esempio dello **Stack** nell'ultimo capitolo, si può volere allocare la memoria in grandi blocchi, per rapidità, invece di creare spazio ogni volta che un elemento viene aggiunto. Se l'interfaccia e l'implementazione sono chiaramente separate e protette, si può ottenere ciò richiedendo solo un nuovo link dal programmatore client.

Il controllo d'accesso del C++

Il C++ introduce tre nuove parole riservate per fissare i limiti in una struttura: **public, private** e **protected.** Il loro uso e il loro significato sono molto semplici. Questi *access specifiers (specificificatori di accesso)* sono usati solo in una dichiarazione di struttura e cambiano i limiti per tutte le dichiarazioni che li seguono. In qualsiasi momento si usa un specificatore d'accesso, esso deve essere seguito dai due punti.

Per **public** s'intende che tutte le dichiarazioni dei membri sono disponibili a tutti. I membri **public** sono come quelli dello **struct**. Per esempio, le seguenti dichiarazioni **struct** sono identiche:

```
//: C05:Public.cpp
// Public è come la struct del C

struct A {
  int i;
  char j;
  float f;
  void func();
};

void A::func() {}

struct B {
public:
  int i;
  char j;
  float f;
  void func();
};

void B::func() {}

int main() {
  A a; B b;
  a.i = b.i = 1;
  a.j = b.j = 'c';
  a.f = b.f = 3.14159;
  a.func();
  b.func();
} ///:~
```

La parola chiave **private**, invece, significa che nessuno eccetto noi può accedere a quel membro, il creatore del tipo, dentro i membri funzione di quel tipo. **private** è un muro di mattoni tra noi e il programmatore client, se qualcuno prova ad accedere al membro **private**, avrà un errore di compilazione. In **struct B** nell'esempio sopra, si potrebbe voler nascondere porzioni della rappresentazione (cioè il membro data) , accessibile solo per noi:

```
//: C05:Private.cpp
// Fissare i limiti

struct B {
private:
  char j;
  float f;
public:
```

```
  int i;
  void func();
};

void B::func() {
  i = 0;
  j = '0';
  f = 0.0;
};

int main() {
  B b;
  b.i = 1;    // OK, public
//!  b.j = '1';  // vietato, private
//!  b.f = 1.0;  // vietato, private
} ///:~
```

Sebbene **func()** può accedere a qualsiasi membro di **B** (poichè **func()** è un membro di **B**, in questo modo ha automaticamente il permesso), un' ordinaria funzione globale come **main()** non può. Naturalmente, neanche le funzioni membro delle altre strutture. Solamente le funzioni che sono chiaramente dichiarate nella dichiarazione della struttura (il "contratto") possono avere accesso ai membri **private**.

Non c'è nessun ordine richiesto per gli specificatori d'accesso e possono apparire pù di una volta. Essi influenzano tutti i membri dichiarati dopo di loro e prima del prossimo specificatore d'accesso.

protected

L'ultimo specificatore d'accesso è **protected**. Esso funziona come **private**, con un' eccezione che in realtà non può essere spiegata ora: le strutture "ereditate" (le quali non possono accedere a membri **private**) hanno il permesso di accedere ai membri **protected**. Questo diverrà più chiaro nel capitolo 14 quando l'ereditarietà verrà introdotta. Per il momento si consideri **protected** come **private**.

Friends

Cosa succede se si dà permesso di accesso ad una funzione che non è un membro della struttura corrente? Ciò si ottiene dichiarando quella funzione **friend** *dentro* la dichiarazione della struttura. È importante che la dichiarazione avvenga dentro la dichiarazione della struttura perchè si deve poter (e anche il compilatore) leggere la dichiarazione della struttura e vedere tutte le dimensioni ed il comportamento dei tipi di dato. Una regola molto importante in tutte le relazione è: " *Chi può accedere alla mia implementazione privata?*".

La classe controlla quale codice ha accesso ai suoi membri. Non c'è un modo magico di intrufolarsi dal di fuori se non si è un **friend**; non si può dichiarare una nuova classe e dire: "*Ciao, io sono un amico di **Bob**!*" ed aspettarsi di vedere i membri **private** e **protected** di **Bob**.

Si può dichiarare una funzione globale come un **friend** e si può dichiarare anche una funzione membro di un'altra struttura o perfino una struttura intera come un **friend**. Ecco qui un esempio:

```
//: C05:Friend.cpp
// Friend permette un accesso speciale
// Dichiarazione (specificazione di tipo incompleta)
struct X;

struct Y {
  void f(X*);
};

struct X { // Definizione
private:
  int i;
public:
  void inizializza();
  friend void g(X*, int); // friend globale
  friend void Y::f(X*);  // Struct membro friend
  friend struct Z; // L'intera struct è un friend
  friend void h();
};

void X::inizializza() {
  i = 0;
}

void g(X* x, int i) {
  x->i = i;
}

void Y::f(X* x) {
  x->i = 47;
}

struct Z {
private:
  int j;
public:
  void inizializza();
  void g(X* x);
};

void Z::inizializza() {
  j = 99;
}

void Z::g(X* x) {
  x->i += j;
}

void h() {
  X x;
  x.i = 100; // manipulazione diretta del dato
}

int main() {
  X x;
  Z z;
  z.g(&x);
} ///:~
```

struct Y ha una funzione membro **f()** che modificherà un oggetto del tipo **X**. Questo è un po' un rompicapo perchè il compilatore del C++ richiede di dichiarare ogni cosa prima di riferirsi a ciò, quindi la **struct Y** deve essere dichiarata prima degli stessi membri come un

friend nel **struct X**. Ma per essere dichiarata **Y::f(X*)**, deve essere prima dichiarata la **struct X**!

Ecco qui la soluzione. Si noti che **Y::f(X*)** prende l'*indirizzo* di un oggetto **X**. Questo è critico perchè il compilatore sa sempre come passare un indirizzo, il quale è di lunghezza fissa indifferente all'oggetto che è passato, anche se non ha tutte le informazioni circa la lunghezza del tipo. Se si prova a passare l'intero oggetto, comunque, il compilatore deve vedere l'intera struttura definizione di **X** per conoscere la lunghezza e come passarla, prima di permettere di far dichiarare una funzione come **Y::g(X)**.

Passando un'indirizzo di un **X**, il compilatore permette di fare una *specificazione di tipo incompleta* di **X** prima di dichiarare **Y::f(X*)**. Ciò avviene nella dichiarazione:

```
struct X;
```

Questa semplice dichiarazione dice al compilatore che c'è una **struct** con quel nome, quindi è giusto riferirsi ad essa se non c'è bisogno di conoscere altro che il nome.

Ora, in **struct X**, la funzione **Y::f(X*)** può essere dichiarata come un **friend** senza nessun problema. Se si provava a dichiararla prima il compilatore avrebbe visto la piena specificazione per **Y** e avrebbe segnalato un errore. Questo è una caratteristica per assicurare consistenza ed eliminare i bachi.

Notare le altre due funzioni **friend**. La prima dichiara una funzione ordinaria globale **g()** come un **friend**. Ma **g()** non è stata precedentemente dichiarata globale! Risulta che **friend** può essere usato in questo modo per dichiarare simultaneamente la funzione e darle uno stato **friend**. Ciò si estende alle intere strutture:

```
friend struct Z;
```

è una specificazione di tipo incompleta per **Z** e dà all'intera struttura lo stato **friend**.

Friends nidificati

Usare una struttura nidificata non dà automaticamente l'accesso ai membri **private**. Per compiere ciò, si deve seguire una particolare forma: primo, dichiarare(senza definire) la struttura nidificata, dopo la si dichiara come un **friend**, ed infine si definisce la struttura. La definizione della struttura deve essere separata dalla dichiarazione di **friend**, altrimenti sarebbe vista dal compilatore come un non membro. Ecco qui un esempio:

```
//: C05:NestFriend.cpp
// friend nidificati
#include <iostream>
#include <cstring> // memset()
using namespace std;
const int sz = 20;
```

```
struct Contenitore {
private:
  int a[sz];
public:
  void inizializza();
  struct Puntatore;
  friend struct Puntatore;
  struct Puntatore {
  private:
    Contenitore* h;
    int* p;
  public:
    void inizializza(Holder* h);
     // per muoversi nel vettore
    void prossimo();
    void precedente();
    void primo();
    void ultimo();
    // accedere ai valori:
    int leggi();
    void imposta(int i);
  };
};

void Contenitore::inizializza() {
  memset(a, 0, sz * sizeof(int));
}

void Contenitore::Puntatore::inizializza(Holder* rv) {
  h = rv;
  p = rv->a;
}

void Contenitore::Puntatore::prossimo() {
  if(p < &(h->a[sz - 1])) p++;
}

void Contenitore::Puntatore::precedente() {
  if(p > &(h->a[0])) p--;
}

void Contenitore::Puntatore::primo() {
  p = &(h->a[0]);
}

void Contenitore::Puntatore::ultimo() {
  p = &(h->a[sz - 1]);
}

int Contenitore::Puntatore::leggi() {
  return *p;
}

void Contenitore::Puntatore::imposta(int i) {
  *p = i;
}

int main() {
  Contenitore h;
  Contenitore::Puntatore hp, hp2;
  int i;

  h.inizializza();
```

```
  hp.inizializza(&h);
  hp2.inizializza(&h);
  for(i = 0; i < sz; i++) {
    hp.imposta(i);
    hp.prossimo();
  }
  hp.primo();
  hp2.ultimo();
  for(i = 0; i < sz; i++) {
    cout << "hp = " << hp.leggi()
         << ", hp2 = " << hp2.leggi() << endl;
    hp.prossimo();
    hp2.precedente();
  }
} ///:~
```

Una volta dichiarato **Puntatore** è concesso l'accesso ai membri private di **Contenitore** scrivendo:

```
friend struct Puntatore;
```

La **struct Contenitore** contiene un vettore di **int** e di **Puntatore** che permette loro l'accesso. Poichè **Puntatore** è fortemente associato con **Contenitore**, è ragionevole farne un membro della struttura del **Contenitore**. Ma poichè **Puntatore** è una classe separata dal **Contenitore**, si può farne più di uno in **main()** ed usarlo per scegliere differenti parti del vettore. **Puntatore** è una struttura invece di puntatori C grezzi, quindi si può garantire che punteranno sempre dentro **Contenitore**.

La funzione della libreria Standard C **memset()** (in **<cstring>**) è usata per convenienza nel programma di sopra. Essa imposta tutta la memoria ad un indirizzo di partenza (il primo argomento) un particolare valore (il secondo argomento) per **n** byte dopo l'indirizzo di partenza (**n** è il terzo argomento). Naturalmente, si potrebbe semplicemente usare un ciclo, ma **memset()** è disponibile, testato con successo (così è meno probabile che si introduca un errore) e probabilmente più efficiente di quando lo si codifica a mano.

E' puro?

La definizione di classe dà una segno di verifica, quindi si può vedere guardando la classe quali funzioni hanno il permesso di modificare le parti **private** della classe. Se una funzione è un **friend,** significa che non è un membro, ma si vuole dare un permesso di modificare comunque dati **private** e deve essere elencata nella definizione della classe così che chiunque possa vedere che è una delle funzioni privilegiate.

Il C++ è un linguaggio orientato a oggetti ibrido, non un puro, e **friend** fu aggiunto per aggirare i problemi pratici che sorgevano.

Layout dell'oggetto

Il capitolo 4 afferma che una **struct** scritta per un compilatore C e poi compilata col C++ rimarrebbe immutata. Ciò è riferito al layout dell'oggetto della **struct,** cioè, dove lo spazio per le variabili individuali è posizionato nella memoria allocata per l'oggetto. Se il compilatore del C++ cambia il layout della **struct** del C, allora verrebbe corrotto qualsiasi codice C scritto in base alla conoscenza delle posizioni delle variabili nello **struct**.

Quando si iniziano ad usare gli specificatori d'accesso, tuttavia, ci si sposta completamente nel regno del C++ e le cose cambiano un po'. Con un particolare "blocco di accesso" (un gruppo di dichiarazioni delimitate dagli specificatori d'accesso) è garantito che le variabili siano posizionate in modo contiguo, come in C. Tuttavia i blocchi d'accesso possono non apparire nell'oggetto nell'ordine in cui si dichiarano. Sebbene il compilatore posizionerà di solito i blocchi esattamente come si vedono, non c'è alcuna regola su ciò, perchè una particolare architettura di macchina e/o ambiente operativo forse può avere un esplicito supporto per il **private** e **protected** che potrebbe richiedere che questi blocchi siano posti in locazioni speciali di memoria. La specifica del linguaggio non vuole restringere questa possibilità.

Gli specificatori d'accesso sono parte della struttura e non influenzano gli oggetti creati dalla struttura. Tutte le informazioni degli specificatori d'accesso scompaiono prima che il programma giri; generalmente durante la compilazione. In un programma funzionante, gli oggetti diventano " regioni di memoria" e niente più. Se veramente lo si vuole, si possono violare tutte le regole ed accedere alla memoria direttamente, come si può farlo in C. Il C++ non è progettato per preventivarci dal fare cose poco saggie. Esso ci fornisce solamente di una alternativa più facile e molto desiderata.

In generale, non è una buona idea dipendere da qualcosa che è un' implementazione specifica quando si sta scrivendo un programma. Quando si devono avere dipendenze di specifiche di implementazione, le si racchiudano dentro una struttura cosicchè qualsiasi modifica per la portabilità è concentrata in un solo posto.

La classe

Il controllo d'accesso è spesso detto *occultamento dell'implementazione*. Includere funzioni dentro le strutture (ciò è spesso detto incapsulazione[36]), produce un tipo di dato con caratteristiche e comportamenti, ma l'accesso di controllo pone limiti con quel tipo di dato, per due importanti ragioni. La prima è stabilire cosa il programmatore client può e non può usare. Si può costruire un proprio meccanismo interno nella struttura senza preoccuparsi che il programmatore client penserà che questi meccanismi sono parte dell'interfaccia che dovrebbero essere usati.

Questa problematica porta direttamente alla seconda ragione, che riguarda la separazione dell'interfaccia dall'implementazione. Se la struttura viene usata in un insieme di programmi, ma i programmatori client non possono fare nient'altro che mandare messagi all'interfaccia pubblica, allora si può cambiare tutto ciò che è **private** senza richiedere modifiche al codice.

L'incapsulamento ed il controllo d'accesso, presi insieme, sono qualcosa di più che una **struct** del C. Ora siamo nel mondo della programmazione orientata agli oggetti, dove una

struttura descrive una classe di oggetti come se si descriverebbe una classe di pesci o una classe di uccelli: ogni oggetto appartenente a questa classe condividerà le stesse caratteristiche e comportamenti. Ecco cosa è diventata una dichiarazione di struttura, una descrizione del modo in cui tutti gli oggetti di questo tipo appariranno e si comporteranno.

Nell'originale linguaggio OOP, Simula-67, la parola chiave **class** fu usata per descrivere un nuovo tipo di dato. Ciò apparentemente ispirò Stroustrup a scegliere la stessa parola chiave per il C++, per enfatizzare che questo era il punto focale dell'intero linguaggio: la creazione di nuovi tipi di dato che sono qualcosa in più che le **struct** del C con funzioni. Ciò certamente sembra una adeguata giustificazione per una nuova parola chiave.

Tuttavia l'uso di una **class** nel C++ si avvicina ad essere una parola chiave non necessaria. E' identica alla parola chiave **struct** assolutamente in ogni aspetto eccetto uno: **class** è per default **private**, mentre struct è **public**. Ecco qui due strutture che producono lo stesso risultato:

```
//: C05:Class.cpp
// Similitudini tra struct e class

struct A {
private:
  int i, j, k;
public:
  int f();
  void g();
};

int A::f() {
  return i + j + k;
}

void A::g() {
  i = j = k = 0;
}

// Identici risultati sono prodotti con:

class B {
  int i, j, k;
public:
  int f();
  void g();
};

int B::f() {
  return i + j + k;
}

void B::g() {
  i = j = k = 0;
}

int main() {
  A a;
  B b;
  a.f(); a.g();
  b.f(); b.g();
```

```
} ///:~
```

La classe è il concetto fondamentale OOP in C++. È una delle parole chiave che non sarà indicata in grassetto in questo libro, diventa noiso vederla ripetuta. Il passaggio alle classi è così importante che sospetto che Stroustrup avrebbe preferito eliminare **struct**, ma il bisogno di compatibilità con il codice esistente non lo ha permesso.

Molti preferiscono uno stile di creazione di classi che è più simile alla **struct** che alla classe, perchè si può non usare il comportamento **private** della classe per default iniziando con **public**:

```
class X {
public:
  void funzione_di_interfaccia();
private:
  void funzione_privata();
  int rappresentazione_interna;
};
```

La logica dietro ciò sta nel fatto che il lettore è interessato a vedere prima i membri più importanti, poi può ignorare tutto ciò che è **private**. Infatti, le sole ragioni per cui tutti gli altri membri devono essere dichiarati nella classe sono dovute al fatto che così il compilatore conosce la grandezza degli oggetti e li può allocare correttamente e quindi garantire consistenza.

Gli esempi di questo libro, comunque, porrà i membri **private** per prima :

```
class X {
  void private_function();
  int internal_representation;
public:
  void interface_function();
};
```

qualcuno persino arricchisce i propri nomi nomi privati:

```
class Y {
public:
  void f();
private:
  int mX;  // nome "Self-decorated"
};
```

Poichè **mX** è già nascosto nello scope di Y, la **m** (sta per "membro") non è necessaria.Tuttavia, nei progetti con molte variabili globali (cosa da evitare, ma a volte è inevitabile nei progetti esistenti), è di aiuto poter distinguere all'interno di una definizione di funzione membro quale dato è globale e quale è un membro.

Modificare Stash per usare il controllo d'accesso

Ha senso riprendere l'esempio del Capitolo 4 e modificarlo per usare le classi ed il controllo d'accesso. Si noti la porzione d'interfaccia del programmatore client è ora chiaramente distinguibile, quindi non c'è possibilità da parte del programmatore client di manipolare una parte della classe che non dovrebbe.

```
//: C05:Stash.h
// Convertita per usare il controllo d'accesso
#ifndef STASH_H
#define STASH_H

class Stash {
  int size;      // Dimensione di ogni spazio
  int quantity;  // Numero dello spazio libero
  int next;      // prossimo spazio libero
 // array di byte allocato dinamicamente:
  unsigned char* storage;
  void inflate(int increase);
public:
  void initialize(int size);
  void cleanup();
  int add(void* element);
  void* fetch(int index);
  int count();
};
#endif // STASH_H ///:~
```

La funzione **inflate()** è stata resa **private** perchè è usata solo dalla funzione **add()** ed è in questo modo parte della implementazione sottostante, non dell' interfaccia. Ciò significa che, in seguito, si può cambiare l'implementazione sottostante per usare un differente sistema per la gestione della memoria.

Tranne l'include del file, l'header di sopra è l'uncia cosa che è stata cambiata per questo esempio. Il file di implementanzione ed il file di test sono gli stessi.

Modificare Stack per usare il controllo d'accesso

Come secondo esempio, ecco qui **Stack** trasformato in una classe. Ora la struttura nidificata data è **private**, cosa non male perchè assicura che il programmatore client non dovrà mai guardarla e non dipenderà dalla rappresentazione interna di **Stack**:

```
//: C05:Stack2.h
// struct nidificate tramite linked list
#ifndef STACK2_H
#define STACK2_H

class Stack {
  struct Link {
    void* data;
    Link* next;
    void initialize(void* dat, Link* nxt);
  }* head;
public:
  void initialize();
```

```
  void push(void* dat);
  void* peek();
  void* pop();
  void cleanup();
};
#endif // STACK2_H ///:~
```

Come prima, l'implementazione non cambia e quindi non è ripetuta qui. Anche il test è identico. L'unica cosa che è stata cambiata è la robustezza dell'interfaccia della classe. Il valore reale del controllo d'accesso è impedire di oltrepassare i limiti durante lo sviluppo. Infatti, il compilatore è l'unico che conosce il livello di protezione dei membri della classe. Nessuna informazione del controllo di accesso arriva al linker, tutti i controlli di protezione sono fatti dal compilatore.

Si noti che l'interfaccia presentata al programmatore client è adesso veramente quella di uno stack di tipo push-down. È implementato come una linked list, ma la si può cambiare senza influenzare il programmatore client che interagisce con essa.

Gestire le classi

Il controllo d'accesso in C++ permette di separare l'interfaccia dall'implementazione, ma l'occultamento dell'implementazione è solamente parziale. Il compilatore deve vedere ancora le dichiarazioni di tutte le parti di un oggetto per crearlo e manipolarlo correttamente. Si potrebbe immaginare un linguaggio di programmazione che richiede solo un'interfaccia pubblica di un oggetto e permette di nascondere l'implementazione privata, ma il C++ esegue la maggior parte dei controlli sul tipo staticamente (a tempo di compilazione) . Ciò significa che si saprà subito se c'è un errore. Significa anche che il proprio programma è più efficiente. Tuttavia, includere l'implementazione privata ha due effetti: l'implementazione è visibile anche se non è accessibile e può richiedere un inutile ricompilazione.

Nascondere l'implementazione

Alcuni progetti non possono permettersi di avere le loro implementazioni visibili al programmatore client. Essa può mostrare informazioni strategiche in un file header di libreria che un'azienda non vuole mostrare alla concorrenza. Si può star lavorando su un sistema dove la sicurezza è un requisito, un algoritmo di criptazione per esempio, e non si vuole esporre indizzi in un file header che potrebbero aiutare a crackare il codice. Oppure si può volere mettere il codice in un ambiente ostile, dove i programmatori accedono direttamente ai componenti privati, usando puntatori e casting. In tutte queste situazioni, serve avere la struttura compilata dentro un file d'implementazione piuttosto che esposta in un file header.

Ridurre la ricompilazione

Il project manager del proprio ambiente di programmazione ricompilerà un file se esso è toccato (cioè, modificato) oppure se un altro file dipende da esso, cioè se è modificato un file header. Questo significa che ogni volta che si fa un cambiamento ad una classe, non importa se alla sua interfaccia pubblica o ai membri privati, si forzerà una ricompilazione di qualsiasi cosa che include quel header file. Ciò è spesso detto *fragile base-class problem*

(problema di classe base fragile). Per un progetto grosso nelle prime fasi ciò può essere ingombrante perchè l'implementazione sottostante può cambiare spesso; se il progetto è molto grande, il tempo di compilazione può proibire cambiamenti rapidi.

La tecnica per risolvere ciò è spesso detta *handle classes* oppure "Cheshire cat"[37], ogni cosa tranne l'implementazione scompare fatta eccezione di un singolo puntatore, lo "smile". Il puntatore si riferisce ad una struttura la cui definizione è nel file di implementazione con tutte le definizioni delle funzioni membro. Quindi, finchè l'interfaccia non viene toccata, l'header file non è modificato. L'implementazione può cambiare e solo il file di imlementazione deve essere ricompilato e relinkato con il progetto.

Ecco qui un semplice esempio dimostrante la tecnica. Il file principale contiene solo l'interfaccia pubblica e un singolo puntatore di un una incompleta classe specificata :

```
//: C05:Handle.h
// handle classes

#ifndef HANDLE_H
#define HANDLE_H

class Handle {
  struct Cheshire; // solo la dichiarazione della classe
  Cheshire* smile;
public:
  void inizializza();
  void pulisci();
  int leggi();
  void cambia(int);
};
#endif // HANDLE_H ///:~
```

Questo è tutto quello che il programmatore client può vedere. La linea

```
struct Cheshire;
```

è una *incomplete type specification* (specificazione incompleta di tipo) o una *dichiarazione di classe* (una definizione di classe include il corpo di una classe). Esso dice al compilatore che il **Cheshire** è un nome di struttura, ma non fornisce dettagli circa la struct. Questa è un' informazione sufficiente per creare un puntatore alla **struct**, non si può creare un oggetto prima che sia stata fornito il corpo della struttura. In questa tecnica il corpo della struttura è nascosto nel file di implementazione:

```
//: C05:Handle.cpp {O}
// implementazione del Handle
#include "Handle.h"
#include "../require.h"

// Definizione dell'implementazione del Handle:
struct Handle::Cheshire {
  int i;
};

void Handle::inizializza() {
  smile = new Cheshire;
  smile->i = 0;
}
```

```
void Handle::pulisci() {
  delete smile;
}

int Handle::leggi() {
  return smile->i;
}

void Handle::cambia(int x) {
  smile->i = x;
} ///:~
```

Cheshire è una struttura nidificata, quindi deve essere definita con la risoluzione dello scope:

```
struct Handle::Cheshire {
```

Nel **Handle::inizializza( ),** il salvataggio è localizzato per una struttura Cheshire, e nel **Handle::pulisci( )** questo salvataggio è rilasciato. Questo salvataggio è usato al posto di tutti gli elementi dato che normalmente si metterebbero nella sezione **private** della classe. Quando si compila **Handle.cpp**, questa definizione di struttura è celata in un oggetto dove nessuno può vederla. Se si cambiano gli elementi di **Cheshire**, l'unico file che deve essere ricompilato è **Handle.cpp** perchè il file principale non viene modificato.

L'uso di **Handle** è identico all'uso di qualsiasi classe: si include l'header, si creano oggetti e si mandano messaggi.

```
//: C05:UseHandle.cpp
//{L} Handle
// Usare  la  Handle class
#include "Handle.h"

int main() {
  Handle u;
  u.inizializza();
  u.leggi();
  u.cambia(1);
  u.pulisci();
} ///:~
```

L'unica cosa a cui il programmatore client può accedere è l'interfaccia pubblica, quindi finchè l'implementazione è l'unica cosa che cambia, il file sopracitato non ha bisogno mai di essere ricompilato. Dunque sebbene questo non è un perfetto occultamento dell'implementazione, esso è una gran miglioramento.

Sommario

Il controllo d'accesso in C++ dà un prezioso controllo al creatore di una classe. Gli utenti della classe possono chiaramente vedere cosa esattamente possono usare e cosa ignorare. Ancor più importante è l' abilità di assicurare che nessun programmatore client diventi dipendente da qualche parte della implementazione sottostante della classe. Se si conosce ciò come creatore di una classe , si può cambiare l'implementazione sottostante con la consapevolezza che nessun programmatore client sarà penalizzato dai cambiamenti, perchè essi non possono accedere a quella parte della classe.

Quando si ha la competenza di cambiare l' implementazione sottostante, non solo si può migliorare il proprio progetto dopo un po', ma si ha anche la libertà di commettere errori. Non è importante come accuratamente si pianifichi e progetti, si faranno errori. Sapendo ciò è relativamente sicuro fare questi errori e ciò significa che si sarà più sperimentatori, si imparerà più velocemente e si finira il proprio progetto più presto.

L'interfaccia pubblica di una classe è ciò che il programmatore client *vede*, quindi è la parte più importante della classe per fare bene durante l'analisi e il disegno. Ma anche qui c'è margine per cambiare. Se non si ottiene la giusta interfaccia la prima volta, si possono aggiungere più funzioni mantendo quelle che i programmatori client hanno già usato nel loro codice.

---

[36] Come notato prima, a volte il controllo di accesso è indicato come incapsulamento.

[37] Questo nome è attribuito a John Carolan, uno dei primi pionieri del C++, e naturalmente, Lewis Carroll. Questa tecnica può anche essere vista come un tipo di design pattern "bridge", descritto nel Volume 2.

6: Inizializzazione & Pulizia

Nel Capitolo 4 si è migliorato significativamente l'uso delle librerie, riunendo tutti i componenti sparsi di una tipica libreria C e incapsulandoli in una struttura (un tipo dati astratto, che da ora in poi sarà chiamato *classe*).

In tal modo, non solo si fornisce un unico punto di accesso a un dato componente di una libreria, ma si nascondono anche i nomi delle funzioni all'interno del nome della classe. Il capitolo 5 ha introdotto il concetto di accesso controllato (protezione delle informazioni). Grazie ad esso il progettista della classe ha un modo di stabilire dei chiari limiti nel determinare cosa può essere manipolato dal programmatore client e cosa invece è oltre il limite. Ciò significa che i meccanismi interni delle operazioni eseguibili dal nuovo tipo dati avvengono a discrezione del progettista della classe e sono sotto il suo controllo, mentre viene reso chiaro ai programmatori client quali sono i membri a cui possono e dovrebbero prestare attenzione.

Insieme, incapsulamento e accesso controllato, semplificano significativamente l'uso di una libreria. Essi forniscono un concetto di nuovo tipo dati che è migliore, sotto diversi aspetti, di quello che esiste nel linguaggio C con i suoi tipi predefiniti. Il compilatore C++ può ora garantire il controllo di coerenza sul nuovo tipo dati, assicurando in tal modo un certo livello di sicurezza quando esso viene utilizzato.

Quando si tratta di sicurezza, comunque, il compilatore può fare molto di più di quanto non faccia il C. In questo e nei prossimi capitoli, saranno trattate ulteriori caratteristiche che sono state ingegnerizzate nel C++, le quali fanno sì che i bugs del vostro programma saltino fuori e vi afferrino, a volte perfino prima di compilare il programma, o più solitamente sotto forma di errori o warnings del compilatore. Pertanto, vi abituerete presto all'insolito scenario di un programma C++ che, se compilato, spesso gira correttamente al primo colpo.

Due importanti fattori legati alla sicurezza del codice sono l'inizializzazione e la pulizia delle variabili. Una grossa fetta di bugs in "C" derivano dal fatto che il programmatore dimentica di inizializzare o ripulire debitamente una variabile. Questo è specialmente vero nel caso di librerie "C", in cui i programmatori client non sanno come inizializzare una *struct*, o addirittura non sanno neppure che devono farlo. (Le librerie spesso non includono una funzione di inizializzazione, obbligando in tal modo il programmatore client a inizializzare le strutture "manualmente"). La pulizia è un genere di problema particolare, dato che i programmatori C non fanno fatica a scordarsi delle loro variabili una volta che hanno finito di usarle, così che spesso manca qualunque operazione di pulizia si renda necessaria per le strutture delle librerie.

In C++, il concetto di inizializzazione e pulizia è essenziale per un semplice utilizzo delle librerie e per eliminare molti bugs subdoli che vengono inseriti quando il programmatore client dimentica di eseguire tali operazioni. Questo capitolo prende in esame le proprietà del C++ che aiutano a garantire le appropriate operazioni di inizializzazione e pulizia di una variabile.

Inizializzazione garantita dal costruttore

Sia la classe **Stash** che **Stack** definite in precedenza hanno una funzione denominata **initialize( )**, il cui nome stesso suggerisce che dovrebbe essere richiamata prima di usare l'oggetto in qualsiasi altro modo. Purtroppo, ciò implica che sia il programmatore client ad assicurare la dovuta inizializzazione. I programmatori client tendono a trascurare dettagli come l'inizializzazione mentre caricano a testa bassa con la fretta di risolvere il loro problema grazie alla vostra fantastica libreria. In C++ l'inizializzazione è troppo importante per essere lasciata al programmatore client. Il progettista della classe può garantire che ogni oggetto venga inizializzato fornendo una funzione speciale chiamata *costruttore*. Se una classe ha un costruttore, il compilatore richiamerà automaticamente il costruttore nel punto in cui un oggetto viene creato, prima che il programmatore client possa mettergli le mani addosso. Il costruttore non è un'opzione per il programmatore client. Esso viene eseguito dal compilatore al momento in cui l'oggetto è definito.

La prossima sfida è il nome da assegnare a questa funzione speciale (il costruttore). Sorgono due problemi. Il primo è che qualunque nome si scelga può in teoria collidere con il nome che vi piacerebbe assegnare a un'altra funzione membro della classe. Il secondo è che il compilatore, avendo la responsabilità di invocare tale funzione autonomamente, deve sempre sapere quale funzione richiamare. La soluzione scelta da Stroustrup sembra la più semplice e anche la più logica: il costruttore ha lo stesso nome della classe. Ed è logico che tale funzione sia richiamata automaticamente all'inizializzazione.

Ecco una semplice classe con un costruttore:

```
class X {
  int i;
public:
  X();  // Costruttore
};
```

Adesso, quando si definisce un oggetto:

```
void f() {
  X a;
  // ...
}
```

quello che accade è lo stesso che accadrebbe se **a** fosse un **int**: viene allocato spazio in memoria per contenere l'oggetto. Ma quando il programma raggiunge la riga di programma in cui **a** viene definito, il costruttore viene richiamato automaticamente. Cioé, il compilatore inserisce silenziosamente la chiamata a **X::X()** per l'oggetto **a** nel punto in cui esso viene definito. Come per qualunque altra funzione membro della classe, il primo argomento (segreto) passato al costruttore è il puntatore **this** - l'indirizzo dell'oggetto per cui viene richiamato. Nel caso del costruttore, comunque, **this** sta puntando a un blocco di memoria non inizializzato, e sarà appunto compito del costruttore inizializzarlo dovutamente.

Come qualsiasi funzione, il costruttore può avere degli argomenti, permettendo di specificare come creare un oggetto, dargli dei valori iniziali, e così via. Tali argomenti danno modo di garantire che tutte le parti dell'oggetto siano inizializzate con valori appropriati. Ad esempio, se la classe **Tree** (albero) ha un costruttore con un solo

argomento di tipo intero per stabilire l'altezza dell'albero, si potrà creare un oggetto di tipo Tree in questo modo:

```
Tree t(12);  // albero alto 12 piedi
```

Se **Tree(int)** è l'unico costruttore, il compilatore non permetterà di creare oggetti in alcun altro modo. (Il prossimo capitolo parlerà di costruttori multipli e modi diversi di chiamare i costruttori).

Questo è tutto sul costruttore. E' una funzione dal nome speciale, che viene invocata automaticamente dal compilatore nel momento in cui ciascun oggetto viene creato. Nonostante la sua semplicità, ha un grande valore, in quanto elimina una vasta gamma di problemi e rende più semplice leggere e scrivere il codice. Nel frammento di codice precedente, ad esempio, non si vede una chiamata esplicita a qualche funzione **initialize()**, concettualmente separata dalla definizione dell'oggetto. In C++, definizione e inizializzazione sono concetti unificati - non si può avere l'una senza l'altra.

Sia il costruttore che il distruttore sono un genere di funzioni molto insolito: non hanno un valore di ritorno. Ciò è completamente diverso da un valore di ritorno **void**, in cui la funziona non ritorna niente, ma si ha sempre la possibilità di cambiarla perché ritorni qualcosa. I costruttori e i distruttori non ritornano niente, e non c'è alcun'altra possibilità. Portare un oggetto dentro e fuori dal programma sono azioni speciali, come la nascita e la morte, e il compilatore invoca le funzioni autonomamente, per essere certo che avvengano sempre. Se ci fosse un valore di ritorno, e si potesse scegliere il proprio, il compilatore dovrebbe in qualche modo sapere cosa farne, oppure il programmatore client dovrebbe richiamare esplicitamente i costruttori e i distruttori, il che eliminerebbe la loro protezione implicita.

Pulizia garantita dal distruttore

Un programmatore C si sofferma spesso sull'importanza dell'inizializzazione, ma più raramente si preoccupa della pulizia. Dopo tutto, cosa è necessario fare per ripulire un **int**? Basta scordarsene. Con le librerie comunque, semplicemente "lasciar perdere" un oggetto una volta che si è adoperato non è così sicuro. Che dire se l'oggetto modifica qualche dispositivo hardware, o visualizza qualcosa sullo schermo, o alloca spazio in memoria sullo heap? Se viene semplicemente abbandonato, l'oggetto non raggiungerà mai il suo fine uscendo da questo mondo. In C++, la pulizia è importante quanto l'inizializzazione e viene quindi garantita dal distruttore.

La sintassi per il distruttore è simile a quella del costruttore: la funzione ha lo stesso nome della classe. Tuttavia, il distruttore si distingue dal costruttore per il prefisso ~ (carattere tilde). Inoltre, il distruttore non ha mai argomenti, dato che la distruzione non necessita mai di alcuna opzione. Ecco la dichiarazione per il distruttore:

```
class Y {
public:
  ~Y();
};
```

Il distruttore viene richiamato automaticamente dal compilatore quando un oggetto esce dal suo campo di visibilità (*scope*). E' possibile capire dove viene richiamato il costruttore dal punto di definizione di un oggetto, ma l'unica evidenza della chiamata al distruttore è la parentesi graffa chiusa del blocco che contiene l'oggetto. E il distruttore viene sempre chiamato, perfino quando si usa **goto** per saltare fuori dal blocco. (**goto** esiste anche nel C++ per compatibilità retroattiva con il C, e per quelle volte in cui fa comodo.) Si dovrebbe notare che un *goto non-locale*, implementato tramite le funzioni di libreria C Standard **setjmp()** e **longjmp()**, non fa sì che i distruttori vengano richiamati. (Questa è la specifica, anche se il compilatore usato si comporta diversamente. Affidarsi a una caratteristica non riportata nella specifica significa che il codice generato non sarà portabile).

Ecco un esempio che dimostra le proprietà dei costruttori e dei distruttori viste finora:

```
//: C06:Constructor1.cpp
// Costruttori & distruttori
#include <iostream>
using namespace std;

class Tree {
  int height;
public:
  Tree(int initialHeight);  // Costruttore
  ~Tree();      // Distruttore
  void grow(int years);
  void printsize();
};

Tree::Tree(int initialHeight) {
  height = initialHeight;
}

Tree::~Tree() {
  cout << "all'interno del distruttore di Tree" << endl;
  printsize();
}

void Tree::grow(int years) {
  height += years;
}

void Tree::printsize() {
  cout << "L'altezza dell'albero è " << height << endl;
}

int main() {
  cout << "prima della parentesi graffa aperta" << endl;
  {
    Tree t(12);
    cout << "dopo la creazione dell'albero" << endl;
    t.printsize();
    t.grow(4);
    cout << "prima della parentesi graffa chiusa" << endl;
  }
  cout << "dopo la parentesi graffa chiusa" << endl;
} ///:~
```

Ecco il risultato sul video:

```
prima della parentesi graffa aperta
dopo la creazione dell'albero
L'altezza dell'albero è 12
dopo la parentesi graffa chiusa
all'interno del distruttore di Tree
L'altezza dell'albero è 16
dopo la parentesi graffa chiusa
```

Si può vedere che il distruttore viene richiamato automaticamente alla chiusura del blocco che racchiude l'oggetto (parentesi graffa chiusa).

Eliminazione del blocco di definizione

In C, le variabili devono sempre essere definite all'inizio di un blocco, dopo la parentesi graffa aperta. Tale requisito non è insolito nei linguaggi di programmazione, e la ragione spesso addotta è che si tratta di "buon stile di programmazione". Su questo punto ho i miei sospetti. E' sempre sembrato scomodo ai programmatori, saltare avanti e indietro all'inizio del blocco ogni volta che serve una nuova variabile. Anche il codice risulta più leggibile quando la variabile è definita vicino al punto in cui viene usata.

Puo' darsi che questi siano argomenti di carattere stilistico. In C++, comunque, sorge un problema significativo nell'essere obbligati a definire tutti gli oggetti all'inizio di un blocco. Se esiste un costruttore, esso deve essere richiamato quando l'oggetto viene creato. Ma se il costruttore vuole uno o più argomenti di inizializzazione, come si può sapere se si avranno tali informazioni all'inizio di un blocco? Nella tipica situazione di programmazione, non si avranno. Poiché il C non ha alcun concetto di "privato", tale separazione fra definizione e inizializzazione non è un problema. Il C++ invece, garantisce che quando si crea un oggetto, esso venga anche inizializzato simultaneamente. Questo assicura di non avere oggetti non inizializzati a spasso per il sistema. Il C non se ne dà pensiero; anzi, *incoraggia* tale pratica richiedendo di definire tutte le variabili all'inizio di un blocco, quando non necessariamente si hanno tutte le informazioni per l'inizializzazione [38].

In genere il C++ non permette di creare un oggetto prima che si abbiano le informazioni di inizializzazione per il costruttore. Pertanto, il linguaggio non sarebbe attuabile se si dovessero definire tutte le variabili all'inizio di un blocco. Invece, lo stile del linguaggio sembra incoraggiare la definizione di un oggetto il più vicino possibile al punto in cui esso viene usato. In C++, qualunque regola che si applica a un "oggetto", si riferisce automaticamente anche ad un oggetto di un tipo predefinito. Ciò implica che qualunque oggetto di una classe o variabile di un tipo predefinito possano essere definiti ovunque all'interno di un blocco di codice. Implica anche che si può attendere di avere tutte le informazioni necessarie per una variabile prima di definirla, in modo da potere sempre definire e inizializzare allo stesso tempo:

```
//: C06:DefineInitialize.cpp
// Definire le variabili ovunque
#include "../require.h"
#include <iostream>
#include <string>
using namespace std;

class G {
  int i;
```

```
public:
  G(int ii);
};

G::G(int ii) { i = ii; }

int main() {
  cout << "valore di inizializzazione? ";
  int retval = 0;
  cin >> retval;
  require(retval != 0);
  int y = retval + 3;
  G g(y);
} ///:~
```

come si può notare, vengono eseguite delle istruzioni, dopodiché **retval** viene definita, inizializzata, e utilizzata per acquisire l'input dell'utente. Infine, vengono definite **y** e **g**. Il C invece, non consente di definire una variabile se non all'inizio di un blocco.

Generalmente le variabili dovrebbero essere definite il più vicino possibile al punto in cui vengono utilizzate, ed essere sempre inizializzate al tempo stesso della definizione. (Per i tipi predefiniti, questo diventa un suggerimento stilistico, dato che per essi l'inizializzazione è facoltativa). E' una questione di protezione del codice. Riducendo la durata in cui una variabile è disponibile all'interno di un blocco di codice, si riduce anche l'eventualità che se ne abusi in qualche altro punto del blocco stesso. Questo inoltre, migliora la leggibilità del codice, dato che il lettore non deve saltare continuamente all'inizio del blocco per scoprire il tipo di ogni variabile.

cicli "for"

In C++, si vedrà spesso il contatore per un ciclo **for** definito all'interno dell'espressione **for** stessa:

```
for(int j = 0; j < 100; j++) {
    cout << "j = " << j << endl;
}
for(int i = 0; i < 100; i++)
 cout << "i = " << i << endl;
```

Le istruzioni qui sopra sono importanti casi speciali, che confondono i nuovi programmatori C++.

Le variabili **i** e **j** sono definite direttamente dentro l'espressione for (non consentito in C). Sono quindi disponibili per l'uso nel ciclo for. È una sintassi molto comoda poiché il contesto elimina qualsiasi dubbio sullo scopo di i e j, così da non essere costretti ad utilizzare nomi macchinosi come **i_contatore_cicli** per chiarezza.

Comunque, può risultare un po' di confusione se ci si aspetta che la durata delle variabili **i** e **j** si estenda oltre il blocco del ciclo for. Non è così [39].

Il capitolo 3 fa notare che anche le istruzioni **while** e **switch** permettono di definire oggetti nelle proprie espressioni di controllo, nonostante tale utilizzo sembri molto meno importante rispetto a quello col ciclo **for**.

Si deve fare attenzione a variabili locali che nascondono variabili del blocco più esterno. Generalmente, utilizzare per una variabile annidata lo stesso nome di una variabile globale al blocco genera confusione e induce a errori [40] .

Trovo che blocchi di dimensioni ridotte siano un indice di buona progettazione. Se si hanno diverse pagine per un'unica funzione, probabilmente si sta tentando di fare troppe cose con tale funzione. Funzioni più frazionate sono non solo più utili, ma rendono anche più facile scovare i bugs.

Allocazione di memoria

Ora che le variabili possono essere definite ovunque all'interno di un blocco di codice, potrebbe sembrare che lo spazio in memoria per una variabile non possa essere allocato fino al momento in cui essa viene definita. In realtà è più probabile che il compilatore segua la pratica del C di allocare tutto lo spazio necessario alle variabili di un dato blocco, nel punto in cui esso inizia con la parentesi graffa aperta. Ma ciò è irrilevante, dato che il programmatore non potrà accedere allo spazio di memoria (che poi è l'oggetto) finché esso non sia stato definito[41]. Nonostante la memoria venga allocata all'inizio del blocco di codice, la chiamata al costruttore non avviene fino alla riga di programma in cui l'oggetto è definito, dal momento che l'identificatore dell'oggetto non è disponibile fino ad allora. Il compilatore si assicura perfino che la definizione dell'oggetto (e quindi la chiamata al costruttore) non avvenga in un punto di esecuzione condizionale, come all'interno di un'istruzione **switch** o in un punto che possa essere eluso da un **goto**. Attivando le istruzioni commentate nel seguente frammento di codice si genererà un errore o uno warning:

```
//: C06:Nojump.cpp
// Non ammesso saltare oltre i costruttori

class X {
public:
  X();
};

X::X() {}

void f(int i) {
  if(i < 10) {
   //! goto jump1; // Errore: il goto elude l'inizializzazione
  }
  X x1;  // Costruttore richiamato qui
 jump1:
  switch(i) {
    case 1 :
      X x2;  // Costruttore richiamato qui
      break;
  //! case 2 : // Errore: case elude l'inizializzazione
      X x3;  // Costruttore richiamato qui
      break;
  }
}

int main() {
  f(9);
```

```
  f(11);
}///:~
```

In questa sequenza di istruzioni, sia **goto** che **switch** eluderebbero potenzialmente una riga di programma in cui viene richiamato un costruttore. In tal caso, l'oggetto risulterebbe accessibile benché il suo costruttore non sia stato invocato. Pertanto il compilatore genera un messaggio di errore. Ancora una volta questo garantisce che un oggetto non possa essere creato a meno che non venga anche inizializzato.

Le allocazioni di memoria di cui si è parlato qui avvengono, naturalmente, sullo *stack*. La memoria viene allocata dal compilatore spostando lo stack pointer verso il "basso" (termine relativo, che può indicare incremento o decremento dell'effettivo valore dello stack pointer, a seconda della macchina utilizzata). Gli oggetti possono anche essere allocati sullo *heap* tramite l'istruzione **new**, che verrà esplorata più a fondo nel Capitolo 13.

Stash con costruttori e distruttori

Gli esempi dei capitoli precedenti hanno funzioni ovvie che corrispondono ai costruttori e distruttori: **initialize( )** e **cleanup( )**. Ecco l'header file per la classe **Stash** che fa uso di costruttori e distruttori:

```
//: C06:Stash2.h
// Con costruttori e distruttori
#ifndef STASH2_H
#define STASH2_H

class Stash {
  int size;      // Dimensione di ogni blocco di memorizzazione
  int quantity;  // Numero di blocchi di memorizzazione
  int next;      // Prossimo blocco libero
  // Array di bytes allocato dinamicamente:
  unsigned char* storage;
  void inflate(int increase);
public:
  Stash(int size);
  ~Stash();
  int add(void* element);
  void* fetch(int index);
  int count();
};
#endif // STASH2_H ///:~
```

Le uniche funzioni membro ad essere cambiate sono **initialize( )** e **cleanup( )**, rimpiazzate con un costruttore e un distruttore:

```
//: C06:Stash2.cpp {O}
// Costruttori e distruttori
#include "Stash2.h"
#include "../require.h"
#include <iostream>
#include <cassert>
using namespace std;
const int increment = 100;
```

```
Stash::Stash(int sz) {
  size = sz;
  quantity = 0;
  storage = 0;
  next = 0;
}

int Stash::add(void* element) {
  if(next >= quantity) // Abbastanza spazio rimasto?
    inflate(increment);
  // Copia elemento nello spazio di memorizzazione,
  // cominciando dal prossimo blocco libero:
  int startBytes = next * size;
  unsigned char* e = (unsigned char*)element;
  for(int i = 0; i < size; i++)
    storage[startBytes + i] = e[i];
  next++;
  return(next - 1); // Numero indice
}

void* Stash::fetch(int index) {
  require(0 <= index, "Stash::fetch (-)index");
  if(index >= next)
    return 0; // Per indicare la fine
  // Restituisce il puntatore all'elemento desiderato:
  return &(storage[index * size]);
}

int Stash::count() {
  return next; // Numero di elementi in CStash
}

void Stash::inflate(int increase) {
  require(increase > 0,
    "Stash::inflate zero or negative increase");
  int newQuantity = quantity + increase;
  int newBytes = newQuantity * size;
  int oldBytes = quantity * size;
  unsigned char* b = new unsigned char[newBytes];
  for(int i = 0; i < oldBytes; i++)
    b[i] = storage[i]; // Copia il vecchio nel nuovo
  delete [](storage); // Vecchio spazio di memorizzazione
  storage = b; // Punta alla nuova zona di memoria
  quantity = newQuantity;
}

Stash::~Stash() {
  if(storage != 0) {
   cout << "freeing storage" << endl;
   delete []storage;
  }
} ///:~
```

Come si può notare, per difendersi da errori di programmazione si fa uso di funzioni definite in **require.h**, invece di **assert( )**. Le informazioni fornite da un **assert()** fallito infatti, non sono utili quanto quelle delle funzioni di **require.h** (come sarà mostrato più avanti nel libro).

Poiché **inflate()** è privata, l'unico modo in cui **require()** può fallire è se una delle altre funzioni membro passa accidentalmente un valore non corretto a **inflate()**. Se si è certi

che questo non possa accadere, si potrebbe considerare di rimuovere la **require()**, ma occorre tenere in mente che, finché la classe non è stabile, c'è sempre la possibilità di aggiungere del nuovo codice che introduca degli errori. Il costo di **require()** è basso (e si potrebbe rimuovere automaticamente usando il preprocessore) mentre il valore della robustezza del codice è alto.

Si noti come nel seguente programma di test la definizione di oggetti **Stash** appaia appena prima che essi servano, e come l'inizializzazione sia parte della definizione stessa, nella lista di argomenti del costruttore:

```
//: C06:Stash2Test.cpp
//{L} Stash2
// Costruttori e distruttori
#include "Stash2.h"
#include "../require.h"
#include <fstream>
#include <iostream>
#include <string>
using namespace std;

int main() {
  Stash intStash(sizeof(int));
  for(int i = 0; i < 100; i++)
    intStash.add(&i);
  for(int j = 0; j < intStash.count(); j++)
    cout << "intStash.fetch(" << j << ") = "
         << *(int*)intStash.fetch(j)
         << endl;
  const int bufsize = 80;
  Stash stringStash(sizeof(char) * bufsize);
  ifstream in("Stash2Test.cpp");
  assure(in, " Stash2Test.cpp");
  string line;
  while(getline(in, line))
    stringStash.add((char*)line.c_str());
  int k = 0;
  char* cp;
  while((cp = (char*)stringStash.fetch(k++))!=0)
    cout << "stringStash.fetch(" << k << ") = "
         << cp << endl;
} ///:~
```

Si noti anche come le chiamate a **cleanup()** siano state eliminate, ma i distruttori vengano sempre richiamati automaticamente quando **intStash** e **stringStash** escono dal campo di visibilità.

Una cosa a cui prestare attenzione negli esempi con **Stash**: sono stato molto attento a far uso soltanto di tipi predefiniti; cioé quelli senza distruttori. Se si tentasse di copiare oggetti di una classe dentro **Stash**, ci si imbatterebbe in ogni genere di problema e non funzionerebbe correttamente. La Libreria Standard del C++ in effetti può realizzare copie corrette di oggetti nei suoi contenitori (*containers*), ma si tratta di un processo piuttosto complesso e ingarbugliato. Il seguente esempio, con **Stack**, farà uso dei puntatori per aggirare questo ostacolo, e in un capitolo successivo anche la classe **Stash** sarà modificata per usare i puntatori.

Stack con costruttori e distruttori

Reimplementando la lista concatenata (contenuta in **Stack**) facendo uso di costruttori e distruttori, si mostra come questi lavorino elegantemente con le istruzioni **new** e **delete**. Ecco l'header file modificato:

```
//: C06:Stack3.h
// Con costruttori/distruttori
#ifndef STACK3_H
#define STACK3_H

class Stack {
  struct Link {
    void* data;
    Link* next;
    Link(void* dat, Link* nxt);
    ~Link();
  }* head;
public:
  Stack();
  ~Stack();
  void push(void* dat);
  void* peek();
  void* pop();
};
#endif // STACK3_H ///:~
```

Non solo **Stack** ha un costruttore e il distruttore, ma ce l'ha anche la classe annidata **Link**:

```
//: C06:Stack3.cpp {O}
// Costruttori/Distruttori
#include "Stack3.h"
#include "../require.h"
using namespace std;

Stack::Link::Link(void* dat, Link* nxt) {
  data = dat;
  next = nxt;
}

Stack::Link::~Link() { }

Stack::Stack() { head = 0; }

void Stack::push(void* dat) {
  head = new Link(dat,head);
}

void* Stack::peek() {
  require(head != 0, "Stack vuoto");
  return head->data;
}

void* Stack::pop() {
  if(head == 0) return 0;
  void* result = head->data;
  Link* oldHead = head;
```

```
  head = head->next;
  delete oldHead;
  return result;
}

Stack::~Stack() {
  require(head == 0, "Stack non vuoto");
} ///:~
```

Il costruttore **Link::Link( )** inizializza semplicemente i puntatori **data** e **next**, per cui nella funzione **Stack::push( )** la linea di codice:

```
head = new Link(dat,head);
```

non solo alloca un nuovo concatenamento (utilizzando la creazione dinamica di oggetti tramite la parola chiave **new**, introdotta nel Capitolo 4), ma inizializza anche i suoi puntatori correttamente.

Ci si potrebbe chiedere come mai il distruttore di **Link** non fa nulla – in particolare, perché non chiama **delete** sul puntatore **data**? Ci sono due problemi. Nel Capitolo 4, in cui è stata introdotta la classe **Stack**, si faceva notare che non si può usare propriamente **delete** con un puntatore **void** se esso punta ad un oggetto (asserzione che sarà dimostrata nel Capitolo 13). Ma oltre a questo, se il distruttore di **Link** eseguisse **delete** sul puntatore **data**, la funzione **pop()** finirebbe col restituire un puntatore ad un oggetto non più esistente, il che sarebbe certamente un bug. Questa viene a volte definita come la questione dell'*appartenenza* (*ownership*): **Link** e quindi **Stack** detengono solo i puntatori, ma non sono responsabili della loro pulizia. Ciò significa che si deve stare molto attenti a sapere chi ne *è* il responsabile. Ad esempio, se non si richiama **pop()** e **delete** per tutti i puntatori contenuti in **Stack**, essi non verranno ripuliti automaticamente dal distruttore di **Stack**. La faccenda può diventare spinosa e portare ai memory leaks (*falla di memoria*). Pertanto, sapere chi è responsabile della pulizia di un oggetto può fare la differenza fra un programma di successo ed uno pieno di bugs. Ecco perché **Stack::~Stack( )** visualizza un messaggio d'errore se al momento della distruzione l'oggetto **Stack** non è vuoto.

Dal momento che l'allocazione e la pulizia degli oggetti **Link** sono nascoste all'interno di **Stack** – essendo parte dell'implementazione interna – tali operazioni non risultano visibili nel programma di test. Tuttavia voi siete responsabili di ripulire i puntatori che vengono restituiti da **pop( )**:

```
//: C06:Stack3Test.cpp
//{L} Stack3
//{T} Stack3Test.cpp
// Costruttori/Distruttori
#include "Stack3.h"
#include "../require.h"
#include <fstream>
#include <iostream>
#include <string>
using namespace std;

int main(int argc, char* argv[]) {
  requireArgs(argc, 1); // l'argomento è il nome del file
  ifstream in(argv[1]);
  assure(in, argv[1]);
  Stack textlines;
  string line;
```

```
  // Legge il file e memorizza le righe nell'oggetto stack:
  while(getline(in, line))
    textlines.push(new string(line));
  // Ripesca le righe dall'oggetto stack e le visualizza
  string* s;
  while((s = (string*)textlines.pop()) != 0) {
    cout << *s << endl;
    delete s;
  }
} ///:~
```

In questo caso, tutte le righe contenute in **textlines** vengono ripulite tramite delete, ma se non lo fossero, **require()** genererebbe un messaggio d'errore indicante *memory leak*.

Inizializzazione di aggregati

Un *aggregato* è proprio quello che suggerisce il nome: un insieme di cose raggruppate insieme. Questa definizione include aggregati di tipi misti, come strutture e classi. Un array invece è un aggregato di elementi di un solo tipo.

Inizializzare aggregati può essere tedioso e indurre facilmente ad errori. L'inizializzazione di aggregati nel C++ è molto più sicura. Quando si crea un oggetto che è un aggregato, tutto quello che si deve fare è un'assegnazione, dopodiché sarà il compilatore ad occuparsi dell'inizializzazione. Tale inizializzazione si presenta in diversi gusti, a seconda del tipo di aggregato con cui si ha a che fare, ma in tutti i casi gli elementi per l'assegnazione devono essere racchiusi fra parentesi graffe. Per un array di tipi predefiniti questo è piuttosto semplice:

```
int a[5] = { 1, 2, 3, 4, 5 };
```

Se si tenta di fornire più inizializzatori di quanti siano gli elementi dell'array, il compilatore genera un messaggio d'errore. Ma cosa succede se si forniscono *meno* inizializzatori? Ad esempio:

```
int b[6] = {0};
```

In questo caso il compilatore userà il primo inizializzatore per il primo elemento dell'array, dopodiché userà zero per tutti gli altri elementi. Si noti che tale funzionamento non avviene se si definisce un array senza fornire una lista di inizializzatori. L'espressione di cui sopra dunque è un modo conciso di inizializzare un array a zero, senza usare un ciclo **for**, e senza quindi alcuna possibilità di commettere un errore di superamento dell'indice dell'array (off-by-one error) (a seconda del compilatore può anche essere più efficiente del ciclo **for**.)

Un'altra scorciatoia per gli array è il *conteggio automatico*, con cui si lascia che sia il compilatore a determinare la dimensione dell'array in base al numero di inizializzatori forniti:

```
int c[] = { 1, 2, 3, 4 };
```

Se ora si decide di aggiungere un altro elemento all'array, basta aggiungere un inizializzatore. Se si riesce a impostare il proprio codice in modo tale da doverlo modificare in un solo punto, si riducono le probabilità di commettere errori durante le modifiche. Ma come si determina la dimensione dell'array? L'espressione **sizeof c / sizeof * c** (dimensione dell'intero array diviso per la dimensione del primo elemento) assolve il compito senza bisogno di essere modificata anche se dovesse cambiare la dimensione dell'array [42]:

```
for(int i = 0; i < sizeof c / sizeof *c; i++)
 c[i]++;
```

Poiché le strutture sono anche degli aggregati, esse possono essere inizializzate in maniera simile. Dato che in una **struct** stile C tutti i membri sono pubblici, essi possono essere assegnati direttamente:

```
struct X {
  int i;
  float f;
  char c;
};

X x1 = { 1, 2.2, 'c' };
```

In caso si abbia un array di tali oggetti, questi possono essere inizializzati utilizzando una coppia di parentesi graffe annidata per ciascun oggetto:

```
X x2[3] = { {1, 1.1, 'a'}, {2, 2.2, 'b'} };
```

In questo caso, il terzo oggetto viene inizializzato a zero.

Se qualche membro è privato (tipico per una classe C++ ben progettata), o anche se tutto è pubblico, ma c'è un costruttore, le cose sono diverse. Negli esempi di cui sopra, gli inizializzatori vengono assegnati direttamente agli elementi dell'aggregato, ma i costruttori sono un mezzo per forzare l'inizializzazione attraverso un'interfaccia formale. In tal caso è necessario richiamare i costruttori per effettuare l'inizializzazione. Quindi se si ha una **struct** che assomiglia alla seguente:

```
struct Y {
  float f;
  int i;
  Y(int a);
};
```

E' necessario indicare le chiamate ai costruttori. L'approccio migliore è quello esplicito, come segue:

```
Y y1[] = { Y(1), Y(2), Y(3) };
```

Si ottengono tre oggetti e tre chiamate al costruttore. Ogni qualvolta si abbia un costruttore, sia che si tratti di una **struct** con tutti i membri **public** o di una **class** con dei membri **private**, tutte le inizializzazioni devono avvenire attraverso il costruttore, anche se si sta usando la forma per inizializzare aggregati.

Ecco un secondo esempio che mostra un costruttore con più argomenti:

```
//: C06:Multiarg.cpp
// Costruttore con più argomenti
// in caso di inizializzazione di aggregati
#include <iostream>
using namespace std;

class Z {
  int i, j;
public:
  Z(int ii, int jj);
  void print();
};

Z::Z(int ii, int jj) {
  i = ii;
  j = jj;
}

void Z::print() {
  cout << "i = " << i << ", j = " << j << endl;
}

int main() {
  Z zz[] = { Z(1,2), Z(3,4), Z(5,6), Z(7,8) };
  for(int i = 0; i < sizeof zz / sizeof *zz; i++)
    zz[i].print();
} ///:~
```

Sembra proprio che per ogni oggetto nell'array si abbia una chiamata esplicita al costruttore.

Costruttori di default

Un *costruttore di default* è un costruttore che può essere richiamato senza argomenti (o parametri). Viene usato per creare un "oggetto standard", ma è importante anche quando si chiede al compilatore di creare un oggetto senza fornire ulteriori dettagli. Per esempio, se si usa la **struct Y** definita in precedenza in una definizione come questa:

```
Y y2[2] = { Y(1) };
```

il compilatore lamenterà di non poter trovare il costruttore di default. Il secondo oggetto nell'array vuole essere creato senza argomenti, ed è qui che il compilatore cerca il costruttore di default. Infatti, se si definisce semplicemente un array di oggetti di tipo **Y**,

```
Y y3[7];
```

il compilatore si lamenterà perché ha bisogno di un costruttore di default per inizializzare ogni oggetto dell'array.

Lo stesso problema occorre se si crea un singolo oggetto in questo modo:

```
Y y4;
```

Ricordate, se c'è un costruttore, il compilatore assicura che la costruzione avvenga sempre, a prescindere dalla situazione.

Il costruttore di default è così importante che *se* - e solo se - non si è definito alcun costruttore per una struttura (**struct** o **class**), il compilatore ne creerà uno automaticamente. Pertanto il seguente codice funziona:

```
//: C06:AutoDefaultConstructor.cpp
// Costruttore di default generato automaticamente

class V {
  int i;  // privato
}; // Nessun costruttore

int main() {
  V v, v2[10];
} ///:~
```

Se almeno un costruttore è stato definito, comunque, ma non il costruttore di default, le istanze di **V** nell'esempio sopra genereranno errori di compilazione. Si potrebbe pensare che il costruttore sintetizzato dal compilatore debba eseguire qualche forma di inizializzazione intelligente, come azzerare la zona di memoria dell'oggetto. Ma non lo fa. Se lo facesse appesantirebbe il programma, senza alcun controllo da parte del programmatore. Se si vuole che la memoria venga azzerata, bisogna farlo esplicitamente, scrivendo il proprio costruttore di default.

Nonostante il compilatore sintetizzi un costruttore di default autonomamente, difficilmente il suo comportamento sarà quello desiderato. Si dovrebbe considerare tale caratteristica come una rete di sicurezza, da usarsi raramente. In generale, si dovrebbero definire i propri costruttori esplicitamente e non lasciarlo fare al compilatore.

Sommario

I meccanismi apparentemente elaborati forniti dal C++ dovrebbero suggerire chiaramente l'estrema importanza che il linguaggio pone sull'inizializzazione e la pulizia. Nel progettare il C++, una delle prime osservazioni fatte da Stroustrup sulla produttività del linguaggio C, fu che una grossa fetta di problemi nella programmazione deriva dall'impropria inizializzazione di variabili. Tali bugs sono difficili da scovare, e considerazioni simili valgono anche per la pulizia impropria. Poiché i costruttori e i distruttori consentono di *garantire* l'appropriata inizializzazione e pulizia (il compilatore non permette che un oggetto venga creato e distrutto senza le dovute chiamate al costruttore e al distruttore), si assume un controllo completo in piena sicurezza.

L'inizializzazione di aggregati è stata inclusa con la stessa filosofia – impedisce di commettere i tipici errori di inizializzazione con aggregati di tipo predefinito e rende il codice più conciso.

La protezione del codice è una questione primaria in C++. L'inizializzazione e la pulizia ne costituiscono una parte importante, ma progredendo nella lettura del libro se ne incontreranno altre.

[38] C99, la versione aggiornata del C Standard, permette di definire le variabili in qualsiasi punto di un blocco, come il C++.

[39] Una precedente revisione della bozza standard del C++ diceva che la durata della variabile si estendeva fino alla fine del blocco che racchiudeva il ciclo **for**. Alcuni compilatori la implementano ancora in questo modo, ma non è corretto, pertanto il codice sarà portabile solo se si limita la visibilità della variabile al ciclo **for**.

[40] Il linguaggio Java lo considera talmente una cattiva idea da segnalare tale codice come errato.

[41] OK, probabilmente potreste pasticciare con i puntatori, ma sareste molto, molto cattivi.

[42] Nel Volume 2 di questo libro (disponibile gratuitamente in lingua Inglese su www.BruceEckel.com), si vedrà un modo più conciso di calcolare la dimensione di un array utilizzando i *template*.

7: Overloading di funzioni e argomenti di default

Una delle più importanti caratteristiche in ogni linguaggio di programmazione è l'uso pratico dei nomi

Quando si crea un oggetto (una variabile), si assegna un nome ad una regione di memoria. Una funzione è il nome di un'azione. Nella ricerca di nomi per descrivere il sistema in uso si crea un programma che è più facile da capire e da cambiare. È un po' come scrivere in prosa - lo scopo è di comunicare con i lettori. Quando però si cerca di rappresentare il concetto di sfumatura del linguaggio umano in un linguaggio di programmazione si solleva un problema. Spesso la stessa parola esprime molteplici significati, dipendenti dal contesto. Quando una singola parola ha più di un significato possiamo parlare di *overloading* di tale parola. Questo è molto utile, specialmente quando le differenze sono molto piccole. Diciamo "lava la maglia, lava l'auto". Sarebbe sciocco dover dire forzatamente "lava_maglia la maglia e lava_auto l'auto", visto che l'ascoltatore non deve fare alcuna distinzione circa l'azione svolta. I linguaggi umani hanno una ridondanza intrinseca, quindi anche se si dimenticano alcune parole si può ancora capire il significato. Noi non abbiamo bisogno di identificatori univoci - possiamo dedurre il significato dal contesto.

La maggior parte dei linguaggi di programmazione, tuttavia, obbliga ad avere un identificatore unico per ogni funzione. Se si hanno tre tipi di dati differenti da stampare, **int, char,** e **float** , normalmente bisogna creare tre funzioni con nomi differenti, ad esempio **print_int(), print_char()** e **print_float()** . Questo comporta uno sforzo maggiore per chi deve scrivere il programma e per chi tenta di capirlo.

In C++ c'è un altro fattore che forza ad adottare l'overloading dei nomi delle funzioni: i costruttori. Poiché il nome del costruttore è predeterminato dal nome della classe, potrebbe sembrare che si possa essere un solo costruttore. Ma cosa fare se si vuole creare un oggetto in più di un modo? Ad esempio, immaginiamo di scrivere una classe che può essere inizializzata in modo standard e anche leggendo le informazioni da un file. Abbiamo bisogno di due costruttori, uno senza argomenti (costruttore di default) e uno che accetta un argomento di tipo **string** , nel quale è memorizzato il nome del file con il quale inizializzare l'oggetto. Entrambi sono costruttori, quindi devono avere lo stesso nome, cioè quello della classe. Perciò l'overloading delle funzioni è essenziale per permettere di usare lo stesso nome di funzione – il costruttore in questo case – di essere usato con parametri di tipo differente.

Sebbene l'overloading delle funzioni sia una necessità per i costruttori esso è anche di utilità generale e può essere usato con qualunque funzione, non solo membri di classi. Oltre a ciò, questo significa che se si hanno due librerie che contengono funzioni con lo stesso nome esse non entreranno in conflitto se hanno insiemi di parametri diversi. Tratteremo tutti questi fattori in dettaglio nel proseguo del capitolo.

L'argomento di questo capitolo è l'uso pratico dei nomi di funzione. L'overloading delle funzioni permette di usare lo stesso nome per funzioni differenti, ma c'è un altro modo di rendere più conveniente la chiamata ad una funzione. Cosa dire se si vuole chiamare la stessa funzione in modi diversi? Quando le funzioni hanno lunghe liste di parametri può diventare noioso (e difficile da leggere) scrivere le chiamate alla funzione quando la maggior parte degli argomenti ha lo stesso valore per tutte le chiamate. Una caratteristica molto usata del C++ è quella degli *argomenti di default* . Un argomento di default è quello

che il compilatore inserisce automaticamente se non è specificato nella chiamata alla funzione. Quindi le chiamate **f("salve"), f("ciao", 1)** e **f("buongiorno", 2, 'c')** possono tutte riferirsi alla stessa funzione. Potrebbero anche essere tre funzioni che hanno subito l'overloading, ma quando le liste di parametri sono simili si preferisce di solito un tale comportamento che chiama una singola funzione.

In realtà l'overloading delle funzioni e gli argomenti di default non sono molto complicati. Entro la fine del capitolo si vedrà quando usarli e i meccanismi sottostanti che li implementano durante la compilazione ed il linkaggio.

Ancora sul name mangling

Nel capitolo 4 è stato introdotto il concetto di *name mangling* . Nel codice

```
void f();
class X { void f(); };
```

la funzione **f()** nel campo di visibilità di **class X** non collide con la versione globale di **f** . Il compilatore riesce a fare questo usando internamente nomi diversi per la versione globale di **f()** e **X::f()** . Nel capitolo 4 era stato suggerito che i nomi interni sono semplicemenți realizzati a partire dal nome della funzione e "decorati" con il nome della classe; quindi i nomi usati internamente dal compilatore potrebbero essere **_f** e **_X_f** . Tuttavia è evidente che la "decorazione" del nome della funzione non coinvolge solamente il nome della classe.

Ecco il perché. Si supponga di voler effettuare l'overloading di due funzioni

```
void print(char);
void print(float);
```

Non importa se sono entrambe nello scope di una classe o in quello globale. Il compilatore non può generare due identificatori interni unici usando solo il nome dello scope delle funzioni. Dovrebbe utilizzare **_print** in entrambi i casi. L'idea alla base dell'overloading è di usare lo stesso nome per funzioni con parametri diversi. Quindi per supportare l'overloading il compilatore deve decorare il nome della funzione con i nomi dei tipi degli argomenti. Le funzioni suddette, definite nello scope globale, potrebbero avere nomi interni del tipo **_print_char** e **_print_float** . È importante notare che non è definito alcuno standard per il modo in cui il compilatore deve generare i nomi interni, quindi si vedranno risultati molto differenti a seconda dei compilator. (È possibile vedere come vengono generati chiedendo al compilatore di generare codice assembly come output). Questo, naturalmente, crea problemi se si vuole comprare librerie compilate per compilatori e linker specifici - ma anche se il name mangling fosse standardizzato ci sarebbero altri ostacoli da superare, a causa dei diversi modi in cui i compilatori generano il codice.

Questo è in definitiva tutto quello che bisogna sapere per poter usare l'overloading: si può usare lo stesso nome per funzioni diverse, basta che esse abbiano le liste di parametri differenti. Il compilatore genera i nomi interni delle funzioni, usati da sè stesso e dal linker, a partire dal nome della funzione, dall'insieme dei parametri e dallo scope.

Overloading dei valori di ritorno

È normale meravigliarsi: "Perché solo lo scope e la lista di parametri? perché non i valori di ritorno?". All'inizio potrebbe sembrare sensato usare anche i valori di ritorno per generare il nome interno. In questo modo si potrebbe fare un cosa del genere:

```
void f();
int f();
```

Questo codice sarebbe eseguito correttamente quando il compilatore potrebbe determinare automaticamente il significato dal contesto, come ad esempio

int x = f();

Come può il compilatore determinare qual'è il significato della chiamata in questo caso? Forse anche maggiore è la difficoltà che il lettore incontra nel capire quale funzione è chiamata. L'overloading dei soli tipi di ritorno è troppo debole, quindi il C++ *effetti collaterali (side effects)* non la permette.

Linkage type-safe

C'è anche un altro vantaggio portato dal name mangling. Il C si presenta un problema particolarmente fastidioso quando il programmatore client sbaglia nel dichiarare una funzione o, peggio, una funzione è chiamata senza averla dichiarata e il compilatore deduce la sua dichiarazione dal modo in cui è chiamata. Alcune volte la dichiarazione è corretta, ma quando non lo è può essere un bug difficile da scovare.

Siccome in C++ tutte le funzioni *devono* essere dichiarate prima di essere usate, le possibilità che si presenti questo problema sono diminuite sensibilmente. Il compilatore C++ rifiuta di dichiarare una funzione automaticamente, quindi è conveniente includere l'header appropriato. Con tutto ciò, se per qualche ragione si riuscisse a sbagliare la dichiarazione di una funzione, sia perché dichiarata a mano sia perché è stato incluso l'header errato (ad esempio una versione superata), il name mangling fornisce un'ancora di salvataggio che è spesso chiamata *linkage type-safe* .

Si consideri la seguente situazione. In un file è definita una funzione:

```
//: C07:Def.cpp {O}
// Definizione di funzione
void f(int) { }
///:~
```

Nel secondo file la funzione è dichiarata in maniera errata e in seguito chiamata:

```
//: C07:Use.cpp
//{L} Def
// Dichiarazione errata di funzione
void f(char);

int main() {
//!  f(1); // Causa un errore di link
} ///:~
```

Anche se è possibile vedere che la funzione è in realtà **f(int)** il compilatore non lo può sapere poiché gli è stato detto - attraverso una dichiarazione esplicita - che la funzione è **f(char)** .In questo modo la compilazione avviene correttamente. In C anche la fase di link avrebbe successo, ma *non* in C++. poiché il compilatore effettua il name mangling dei nomi, la definizione diventa qualcosa come **f_int** , mentre in realtà è usata **f_char** . Quando il linker tenta di risolvere la chiamata a **f_char** , può solo trovare **f_int** e restituisce un messagio di errore. Questo è il linkage type-safe. benché il problema non intervenga troppo spesso, quando succede può essere incredibilmente difficoltoso da trovare, specialmente nei grossi progetti. Questo è uno dei casi nei quali potete facilmente trovare un errore difficoltoso in un programma C semplicemente compilandolo con un compilatore C++.

Esempi di overloading

Possiamo ora modificare gli esempi precedenti per usare l'overloading delle funzioni. Come stabilito prima, una prima applicazione molto utile dell'overloading è nei costruttori. Si può vederlo nella seguente versione della classe **Stash** :

```
//: C07:Stash3.h
// Overloading di funzioni
#ifndef STASH3_H
#define STASH3_H

class Stash {
  int size;      // Dimensione di ogni spazio
  int quantity;  // Numero degli spazi di memorizzazione
  int next;      // Prossimo spazio vuoto
  // Array di byte allocato dinamicamente
  unsigned char* storage;
  void inflate(int increase);
public:
  Stash(int size); // Quantità iniziale zero
  Stash(int size, int initQuantity);
  ~Stash();
  int add(void* element);
  void* fetch(int index);
  int count();
};
#endif // STASH3_H ///:~
```

Il primo costruttore **Stash()** è lo stesso di prima, ma il secondo ha un argomento **quantity** che indica il numero iniziale di spazi di memorizzazione da allocare. Nella definizione, è possibile vedere che il valore interno di **quantity** è posto uguale a zero, insieme al puntatore **storage** . Nel secondo costruttore, la chiamata a **inflate(initQuantity)** aumenta **quantity** fino alla dimensione allocata:

```
//: C07:Stash3.cpp {O}
// Overloading di funzioni
#include "Stash3.h"
#include "../require.h"
#include #include using namespace std;
const int increment = 100;

Stash::Stash(int sz) {
  size = sz;
  quantity = 0;
  next = 0;
```

```
  storage = 0;
}

Stash::Stash(int sz, int initQuantity) {
  size = sz;
  quantity = 0;
  next = 0;
  storage = 0;
  inflate(initQuantity);
}

Stash::~Stash() {
  if(storage != 0) {
    cout << "deallocazione di storage" << endl;
    delete []storage;
  }
}

int Stash::add(void* element) {
  if(next >= quantity) // Abbastanza spazio a disposizione?
    inflate(increment);
  // Copia un elemento in storage,
  // iniziando dal prossimo spazio vuoto:
  int startBytes = next * size;
  unsigned char* e = (unsigned char*)element;
  for(int i = 0; i < size; i++)
    storage[startBytes + i] = e[i];
  next++;
  return(next - 1); // Indice
}

void* Stash::fetch(int index) {
  require(0 <= index, "Stash::fetch (-)index");
  if(index >= next)
    return 0; // Indica la fine
  // Trova il puntatore all'elemento desiderato:
  return &(storage[index * size]);
}

int Stash::count() {
  return next; // Numero di elementi in CStash
}

void Stash::inflate(int increase) {
  assert(increase >= 0);
  if(increase == 0) return;
  int newQuantity = quantity + increase;
  int newBytes = newQuantity * size;
  int oldBytes = quantity * size;
  unsigned char* b = new unsigned char[newBytes];
  for(int i = 0; i < oldBytes; i++)
    b[i] = storage[i]; // Copy old to new
  delete [](storage); // Release old storage
  storage = b; // Point to new memory
  quantity = newQuantity; // Adjust the size
} ///:~
```

Quando viene usato il primo costruttore nessuna memoria è allocata per **storage** . L'allocazione avviene la prima volta che si tenta di aggiungere con **add()** un oggetto e il blocco corrente di memoria è troppo piccolo.

Entrambi i costruttori vengono testati nel seguente programma:

```
//: C07:Stash3Test.cpp
//{L} Stash3
// Overloading di funzioni
#include "Stash3.h"
#include "../require.h"
#include #include #include using namespace std;

int main() {
  Stash intStash(sizeof(int));
  for(int i = 0; i < 100; i++)
    intStash.add(&i);
  for(int j = 0; j < intStash.count(); j++)
    cout << "intStash.fetch(" << j << ") = "
         << *(int*)intStash.fetch(j)
         << endl;
  const int bufsize = 80;
  Stash stringStash(sizeof(char) * bufsize, 100);
  ifstream in("Stash3Test.cpp");
  assure(in, "Stash3Test.cpp");
  string line;
  while(getline(in, line))
    stringStash.add((char*)line.c_str());
  int k = 0;
  char* cp;
  while((cp = (char*)stringStash.fetch(k++))!=0)
    cout << "stringStash.fetch(" << k << ") = "
         << cp << endl;
} ///:~
```

La chiamata al costruttore per **stringStash** usa un secondo argomento; presumibilmente le caratteristiche dello specifico problema che si sta affrontando saranno note, permettendo di scegliere una dimensioni inziziale per **Stash** .

Unioni

Come abbiamo visto, la sola differenza tra **struct** e **class** in C++ è che i membri di una **struct** sono di default **public** e quelli di una **class** di default **private** . Una **struct** può anche avere costruttori e distruttori, come ci si potrebbe aspettare. Ma anche una **union** può avere costruttori, distruttori, funzioni membro e anche controllo di accesso. Si possono vedere ancora una volta gli usi e i vantaggi dell'overloading nel seguente esempio:

```
//: C07:UnionClass.cpp
// Unioni con costruttori e funzioni membro
#includeusing namespace std;

union U {
private: // Anche il controllo di accesso!
  int i;
  float f;
public:
  U(int a);
  U(float b);
  ~U();
  int read_int();
  float read_float();
};
```

```
U::U(int a) { i = a; }

U::U(float b) { f = b;}

U::~U() { cout << "U::~U()\n"; }

int U::read_int() { return i; }

float U::read_float() { return f; }

int main() {
  U X(12), Y(1.9F);
  cout << X.read_int() << endl;
  cout << Y.read_float() << endl;
} ///:~
```

Si potrebbe pensare che nel codice sopra la sola differenza tra una **union** e una **class** sia il modo in cui le variabili sono memorizzate (infatti le variabili **int** e **float** sono sovrapposte nella stessa zona di memoria). Tuttavia, una **union** non può essere usata come classe base di una classe derivata, e ciò è a dir poco limitante dal punto di vista dell'object oriented design (l'ereditarietà sarò trattata nel capitolo 14). Nonostante le funzioni membro rendano più ordinato l'accesso alla **union** , non c'è alcun modo di prevenire il programmatore client dal selezionare l'elemento sbagliato dopo che la **union** è stata inizializzata. Nell'esempio sopra, si potrebbe chiamare **X.read_float()** , anche se è inappropriato. Tuttavia una **union** "sicura" può esser incapsulata in una classe. Nel seguente esempio, si noti come la **enum** chiarifichi il codice e come l'overloading sia utile con i costruttori:

```
//: C07:SuperVar.cpp
// Una super-variable
#include using namespace std;

class SuperVar {
  enum {
    character,
    integer,
    floating_point
  } vartype;  // Definisce una variabile
  union {  // Unione anonima
    char c;
    int i;
    float f;
  };
public:
  SuperVar(char ch);
  SuperVar(int ii);
  SuperVar(float ff);
  void print();
};

SuperVar::SuperVar(char ch) {
  vartype = character;
  c = ch;
}

SuperVar::SuperVar(int ii) {
  vartype = integer;
  i = ii;
}
```

```
SuperVar::SuperVar(float ff) {
  vartype = floating_point;
  f = ff;
}

void SuperVar::print() {
  switch (vartype) {
    case character:
      cout << "character: " << c << endl;
      break;
    case integer:
      cout << "integer: " << i << endl;
      break;
    case floating_point:
      cout << "float: " << f << endl;
      break;
  }
}

int main() {
  SuperVar A('c'), B(12), C(1.44F);
  A.print();
  B.print();
  C.print();
} ///:~
```

Nel codice sopra, la **enum** non ha nessun nome di tipo (enumerazione "senza etichetta"). Questo è accettabile se si sta per definire immediatamente una istanza della **enum** , come in questo esempio. Non c'è nessun bisogno di riferirsi al nome del tipo della **enum** nel futuro, quindi esso è opzionale.

La **union** non ha nè nome di tipo nè nome di variabile. Essa è chiamata *unione anonima* , e alloca la memoria per la **union** senza richiedere l'accesso ai suoi elementi attravarso il nome di una variabile e l'operatore punto. Per esempio, se la una **union** anonima è:

```
//: C07:AnonymousUnion.cpp
int main() {
  union {
    int i;
    float f;
  };
  // Accede ai membri senza usare qualificatori:
  i = 12;
  f = 1.22;
} ///:~
```

Notare che si accede ai membri di un unione anonima come se fossero variabili ordinarie. La sola differenza è che entrambe le variabili occupano la stessa regione di memoria. Se un'unione anonima ha "file scope" (fuori da tutte le funzioni e classi) essa deve essere dichiarata **static** , in moto da avere linkage interno.

Nonostante **SuperVar** è ora sicura, la sua utilità è dubbia, in quanto la ragione per usare una **union** nel primo esempio è per risparmiare memoria e l'aggiunta di **vartype** occupa alcuni bit relativi ai dati nella **union** , quindi il risparmio è in effetti eliminato. Ci sono un paio di alternative per rendere questo schema funzionante. Se **vartpye** controllasse più di una istanza dell'unione - se fossero tutte dello stesso tipo - allora si avrebbe bisogno solo di una per tutto il gruppo e ciò non occuperebbe più memoria. un approccio più utile è di porre delle **#ifdef** attorno al codice che usa **vartype** , che potrebbero garantire un

corretto uso durante la fase di sviluppo e testing. Per la versione definitiva l'overhead in termini di tempo e memoria potrebbe essere eliminato.

Argomenti di default

Si esaminino i due costruttori di **Stash()** in **Stash3.h** . Non sembrano molto differenti, vero? Infatti, il primo costruttore sembra essere un caso particolare del secondo con **size** iniziale uguale a zero. Creare e mantenere due differenti versioni di una funione simile è uno spreco di energie.

Per rimediare a questo il C++ offre gli *argomenti di default* . Un argomento di default è un valore posto nella dichiarazione di una funzione che il compilatore inserisce automaticamente se non si specifica un'altro valore nella chiamata. Nell'esempio **Stash** , possiamo sostituire le due funzioni:

```
Stash(int size); // Quantità iniziale zero
Stash(int size, int initQuantity);
```

con la singola funzione:

```
Stash(int size, int initQuantity = 0);
```

La definizione di **Stash(int)** è semplicemente rimossa - solo la definizione dell'unica **Stash(int,int)** è necessaria.

A questo punto le due definizioni di oggetti

```
Stash A(100), B(100, 0);
```

produrranno esattamente lo stesso risultato. In entrambi i casi sarà chiamato lo stesso costruttore, ma per **A** il secondo argomento è inserito automaticamente dal compilatore quando vede che il primo argomento è un **int** e non c'è alcun secondo argomento. Il compilatore ha visto l'argomento di default, quindi sa di poter ancora effettuare la chiamata se inserisce il secondo argomento, e questo è ciò che gli è stato detto di fare rendendolo un argomento di default.

Sia gli argomenti di default che l'overloading di funzioni sono convenienti. Entrambe queste caratteristiche permettono di usare un singolo nome di funzione in situazioni differenti. La differenza è che usando gli argomenti di default il compilatore li sostituisce automaticamente quando non si vuole inserirli a mano. L'esempio precedente è una buona dimostrazione dell'uso degli argomenti di default invece dell'overloading; altrimenti ci si sarebbe ritrovati con due o più funzioni con dichiarazioni e comportamenti simili. Se le funzioni hanno comporamenti molto differenti, solitamente non ha senso usare gli argomenti di default (in questo caso, ci si dovrebbe chiedere se due funzioni molto diverse debbano avere lo stesso nome).

Quando si usano gli argomenti di default bisogna conoscere due regole. La prima è che solo gli argomenti finali possono essere resi di default. Qundi non si può porre un argomento di default seguito da uno non di default. La seconda è che, una volta che si è usato un argomento di default in una particolare chiamata a funzione, tutti gli argomenti seguenti devono essere lasciati di default (questo regola deriva dalla prima).

Gli argomenti di default sono posti solo nella dichiarazione di una funzione (solitamente in un header). Il compilatore deve infatti sapere quale valore usare. A volte i programmatori inseriscono i valori di default, in forma di commenti, anche nella definizione della funzione, a scopo documentativo

```
void fn(int x /* = 0 */) { // ...
```

Argomenti segnaposto

Gli argomenti di una funzione possono essere dichiarati anche senza identificatori. Quando sono usati con gli argomenti di default questo può apparire un po' strambo. Si può scrivere

```
void f(int x, int = 0, float = 1.1);
```

In C++ non è obbligatorio specificare gli identificatori nella definizione:

```
void f(int x, int, float flt) { /* ... */ }
```

Nel corpo della funzione possono essere usati **x** e **flt** ma non il secondo argomento, visto che non ha nome. Le chiamate alla funzione devono tuttavia specificare un valore per il segnaposto: **f(1)** o **f(1,2,3.0)** . Questa sintassi permette di inserire l'argomento come segnaposto senza usarlo. L'idea è che si potrebbe in seguito voler cambiare la definione della funzione in modo da usare il segnaposto, senza il bisogno di cambiare tutto il codice in cui si richiama la funzione. Naturalmente si può raggiungere lo stesso risultato assegnando un nome agli argomenti non usati, ma, definendo un argomento per il corpo di una funzione senza poi usarlo, la maggior parte dei compilatori genererà un warning, credendo che ci sia un errore di concetto. Lasciando consapevolmente l'argomento senza nome si impedisce che venga generato il warning.

Ancora più importante, se si scrive una funzione che usa un argomento e in seguito si decide che non serve, si può realmente rimuoverlo senza generare warning e senza disturbare alcun codice client che chiama la versione precedente.

Scegliere tra l'overloading e gli argomenti di default

Sia l'overloading gli argomenti di defualt forniscono degli strumenti vantaggiosi per le chiamate a funzione. Tuttavia ci possono essere dei casi in cui non sapere scegliere quale tecnica usare. Ad esempio, si consideri il seguente strumento ideato per gestire i blocchi di memoria automaticamente:

```
//: C07:Mem.h
#ifndef MEM_H
#define MEM_H
typedef unsigned char byte;

class Mem {
  byte* mem;
  int size;
  void ensureMinSize(int minSize);
public:
  Mem();
  Mem(int sz);
```

```
  ~Mem();
  int msize();
  byte* pointer();
  byte* pointer(int minSize);
};
#endif // MEM_H ///:~
```

Un oggetto **Mem** possiede un blocco di **byte** e assicura di avere abbastanza memoria. Il costruttore di default non alloca alcuna memoria e il secondo costruttore assicura che siano disponibili **sz** byte di memoria nell'oggetto **Mem** . Il distruttore dealloca la memoria, **msize()** restituisce il numero di byte allocati correntemente in **Mem** e **pointer()** restituisce un puntatore all'indirizzo iniziale del blocco di memoria (**Mem** è uno strumento abbastanza a basso livello). C'è anche un overloading di **pointer()** con il quale il programmatore client può impostare la dimensione minima **minSize** del blocco di byte che vuole, e la funzione membro assicura questo.

Sia il costruttore che la funzione membro **pointer()** usano la funzione membro **private ensureMinSize()** per incrementare la dimensione del blocco di memoria (notare che non è sicuro memoriazzare il risultato di **pointer()** se la memoria è ridimensionata).

Ecco l'implementazione della classe

```
//: C07:Mem.cpp {O}
#include "Mem.h"
#include using namespace std;

Mem::Mem() { mem = 0; size = 0; }

Mem::Mem(int sz) {
  mem = 0;
  size = 0;
  ensureMinSize(sz);
}

Mem::~Mem() { delete []mem; }

int Mem::msize() { return size; }

void Mem::ensureMinSize(int minSize) {
  if(size < minSize) {
    byte* newmem = new byte[minSize];
    memset(newmem + size, 0, minSize - size);
    memcpy(newmem, mem, size);
    delete []mem;
    mem = newmem;
    size = minSize;
  }
}

byte* Mem::pointer() { return mem; }

byte* Mem::pointer(int minSize) {
  ensureMinSize(minSize);
  return mem;
} ///:~
```

Si può vedere come **ensureMinSize()** è la sola funzione addetta ad allocare memoria e che essa è usata dal secondo costruttore e dal secondo overloading di **pointer()** .

All'interno di **ensureMinSize()** non è necessario fare nulla se la dimensione **size** è grande abbastanza. Se è necessario allocare nuova memoria per rendere il blocco più grande (che è anche il caso in cui la dimensione del blocco è zero dopo aver richiamato il costruttore di default), la nuova porzione "extra" è posta a zero usando la funzione della libreria Standard C **memset()** , che è stata introdotta nel capitolo 5. Dopo di questo è chiamata la funzione della libreria Standard C **memcpy()** , che in questo caso copia i dati esistenti da **mem** a newmem (solitamente in modo molto efficiente). Infine la vecchia memoria è deallocata, mentre la nuova memoria e le dimensioni vengono assegnata ai membri appropriati.

La classe **Mem** è ideata per essere usata come strumento all'interno di altre classi per semplificare la gestione della memoria (potrebbe anche essere usata per nascondere un più sofisticato sistema di gestione della memoria fornito, ad esempio, dal sistema operativo). Essa è testata in modo appriopriato in questo esempio creando una semplice classe "string":

```
//: C07:MemTest.cpp
// Test della classe Mem
//{L} Mem
#include "Mem.h"
#include #include using namespace std;

class MyString {
  Mem* buf;
public:
  MyString();
  MyString(char* str);
  ~MyString();
  void concat(char* str);
  void print(ostream& os);
};

MyString::MyString() {  buf = 0; }

MyString::MyString(char* str) {
  buf = new Mem(strlen(str) + 1);
  strcpy((char*)buf->pointer(), str);
}

void MyString::concat(char* str) {
  if(!buf) buf = new Mem;
  strcat((char*)buf->pointer(
    buf->msize() + strlen(str) + 1), str);
}

void MyString::print(ostream& os) {
  if(!buf) return;
  os << buf->pointer() << endl;
}

MyString::~MyString() { delete buf; }

int main() {
  MyString s("My test string");
  s.print(cout);
  s.concat(" some additional stuff");
  s.print(cout);
  MyString s2;
  s2.concat("Using default constructor");
```

```
  s2.print(cout);
} ///:~
```

Tutto quello che si può fare con questa classe è di creare una **MyString** , concatenare il testo e stamparlo su un **ostream** . La classe contiene solo un puntatore a un oggetto **Mem** , ma si noti la distinzione tra il costruttore di default, che imposta il puntatore a zero, e il secondo costruttore, che crea un **Mem** e nel quale copia i dati. Il vantaggio del costruttore di default è che si può creare, ad esempio, un grosso array di **MyString** vuoti occupando pochissima memoria, visto che la dimensione di ciascun oggetto è solo quella di un puntatore e l'unico overhead del costruttore di default è un assegnamento a zero. Il peso di **MyString** si inizia a far sentire solo quando si concatenano due stringhe; a quel punto se necessario si crea l'oggetto **Mem** . Tuttavia, se si usa il costruttore di default e non si concatenano mai stringhe, il distruttore è ancora sicuro perché la chiamata a **delete** su zero i definita in modo da non rilasciare alcun blocco di memoria e non causa problemi.

Guardando questi due costruttori potrebbe sembrare inizialmente che siano candidati per usare gli argomenti di default. Tuttavia, cancellando il costruttore di default e scrivendo il costruttore rimanente con un argomento di default:

```
MyString(char* str = "");
```

funziona tutto correttamente, ma si perdono tutti i benefici precedenti, siccome un oggetto **Mem** è creato sempre e comunque. Per riavere l'efficienza precedente, bisogna modificare il costruttore:

```
MyString::MyString(char* str) {
  if(!*str) { // Puntatore a stringa vuota
    buf = 0;
    return;
  }
  buf = new Mem(strlen(str) + 1);
  strcpy((char*)buf->pointer(), str);
}
```

Questo significa, in effetti, che il valore di default diventa un flag che esegue un blocco di codice specifico se non è usato il valore di default. Anche se sembra abbastanza fattibile con un piccolo costruttore con questo, generalmente questa usanza può causare problemi. Se è necessario *verificare* il valore di default piuttosto che trattarlo come un valore ordinario, questo è un indizio che state scrivendo due funzioni differenti all'interno di uno stesso corpo di funzione: una versione per i casi normali e una di default. Si potrebbe anche dividerle in due corpi di funzione distinti e lasciare che sia il compilatore a scegliere la versione giusta. Questo comporta un piccolissimo (e solitamente invisibile) aumento di efficienza, visto che l'argomento extra non è passato alla funzione e il codice per il confronto non è eseguito. più importante ancora, si mantiene il codice di due funzioni separate *in* due funzioni separate, piuttosto che combinarle in una sola usando gli argomenti di default; questo atteggiamento porta ad una manutenzione più facile, specialmente se le funzioni sono lunghe.

D'altra parte, si consideri la classe **Mem** . Se si esaminano le definizioni dei due costruttori e delle due funzioni **pointer()** è possibile vedere come in entrambi i casi l'uso degli argomenti di default non causa minimamente il cambiamento delle definizioni. Quindi, la classe potrebbe essere facilmente:

```
//: C07:Mem2.h
```

```
#ifndef MEM2_H
#define MEM2_H
typedef unsigned char byte;

class Mem {
  byte* mem;
  int size;
  void ensureMinSize(int minSize);
public:
  Mem(int sz = 0);
  ~Mem();
  int msize();
  byte* pointer(int minSize = 0);
};
#endif // MEM2_H ///:~
```

Notare che una chiamata a **ensureMinSize(0)** sarà sempre abbastanza efficiente

benché in entrambi i casi abbia basato alcune delle mie scelte di design sull'efficenza, bisogna stare molto attenti a non cadere nella trappola di pensare solo in termini di efficenza (anche se affascinante). Lo scopo principale del design è l'interfaccia della classe (i suoi membri **public** , utilizzabili dal programmatore client). Se verrà prodotta una classe facile da usare e da riusare, essa sarà un successo; si può sempre concentrarsi sull'efficienza, ma gli effetti di una classe progettata male perché il programmatore è troppo preso dalle problematiche di efficienza possono essere terribili. Notare che in **MemTest.cpp** l'uso di **MyString** non cambia se è usato un costruttore di default o se l'efficienza è alta o bassa.

Sommario

In generale si cerchi di non usare un argomento di default come un flag in base al quale eseguire codice. Piuttosto, se possibile, si dovrebbe dividere la funzione in due o più funzioni usando l'overloading. Un argomento di default dovrebbe avere un valore con cui la funzione lavora nel modo consueto. È un valore usato molto più spesso degli altri, quindi il programmatore client può generalmente ignorarlo o usarlo solo se vuole cambiarlo rispetto al valore di default.

Si usa un argomento di default per rendere le chiamate alla funzione più facili, specialmente nei casi in cui si hanno diversi valori usati molto spesso. Non solo è più facile scrivere la chiamata, ma anche leggerla, specialmente se il creatore della classe ordina gli argomenti in modo che quello modificato meno di frequente appaia in fondo alla lista.

Un uso particolarmente importatante degli argomenti di default si ha quando si inizia a scrivere una funzione con un insieme di argomenti, e dopo averla usata per un po' si scopre di avere bisogno di ulteriori argomenti. Rendendoli di default si assicura che il codice cliente che usa l'interfaccia precedente non è disturbato.

8: Costanti

Il concetto di costante (espresso dalla parola chiave **const**) è stato introdotto per dare la possibilità ai programmatori di tracciare una linea di demarcazione tra ciò che può cambiare e ciò che deve rimanere costante. Questo strumento aumenta il grado di sicurezza e controllo in un progetto C++.

Fin dalle origini, la parola chiave **const** è stata utilizzata per scopi differenti. Con il passare degli anni, il particolare utilizzo di **const** in C++ ha portato ad un cambiamento di significato della parola chiave anche per il linguaggio C. Inizialmente, tutto questo potrebbe confondere, ma in questo capitolo si imparerà quando, come e perché utilizzare **const**. Alla fine del capitolo verrà introdotta la parola chiave **volatile**, che è una parente stretta di **const** (entrambe riguardano la modalità con la quale un oggetto "varia") con la quale condivide la stessa sintassi.

La prima ragione che ha portato all'introduzione di **const** è stata la necessità di eliminare l'uso della direttiva **#define** per la sostituzione di valori. Il suo impiego, in seguito, si è esteso ai puntatori, agli argomenti e a tipi restituiti delle funzioni, agli oggetti e alle funzioni membro. Ognuno di questi utilizzi differisce leggermente dagli altri anche se, concettualmente, hanno tutti significati compatibili che verranno trattati nelle varie sezioni di questo capitolo.

Sostituzione di valori

Quando si programma in C, il preprocessore viene spesso utilizzato per creare macro e per sostituire valori. Dato che il preprocessore effettua semplicemente una sostituzione testuale e non possiede strumenti per controllare ciò che è stato digitato, la sostituzione di valori effettuata in questo modo può introdurre comportamenti errati, errori che possono essere facilmente evitati in C++ mediante l'utilizzo di valori **const**.

In C, l'utilizzo tipico del preprocessore per sostituire dei valori al posto di nomi è di questo tipo:

```
#define BUFSIZE 100
```

BUFSIZE è un nome che esiste solo prima dell'intervento del preprocessore, perciò non occupa memoria e può essere messo in un file di header per poter fornire un unico valore a tutti gli altri sorgenti che ne fanno utilizzo. Ai fini della manutenzione del software, è molto importante utilizzare questo tipo di sostituzione al post dei cosiddetti "numeri magici". Se si utilizzano numeri magici nel codice, chi leggerà il codice non avrà idea della provenienza di questi numeri e di ciò che essi cosa rappresentano; inoltre, qualora si decidesse di cambiare un valore, bisognerà modificare "a mano" tutti i sorgenti, e non esiste nessun metodo per assicurare che nessun valore sia stato dimenticato (o che sia stato cambiato accidentalmente un valore che doveva rimanere invariato).

Per la maggior parte del tempo, **BUFSIZE** si comporterà come una variabile ordinaria, ma non per sempre. In aggiunta, in questo caso non si hanno informazioni sul tipo. Questa caratteristica può introdurre errori molto difficili da trovare. Il C++ usa **const** per

eliminare questo tipo di problemi portando sotto l'egida del compilatore la sostituzione dei valori. In questo caso si potrà utilizzare

```
const int bufsize = 100;
```

Si può usare **bufsize** ovunque il compilatore conosca il suo valore a tempo di compilazione. Il compilatore può utilizzare **bufsize** per fare quello che viene chiamato *constant folding*, cioè per ridurre una espressione costante complicata in una semplificata effettuando i conti necessari a tempo di compilazione. Questo è importante specialmente nelle definizioni degli array:

```
char buf[bufsize];
```

Si può utilizzare **const** per tutti i tipi built-in (**char**, **int**, **float**, e **double**) e le loro varianti (così come per gli oggetti, come si vedrà più avanti in questo capitolo). A causa degli errori subdoli che il preprocessore potrebbe introdurre, si dovrebbe usare sempre **const** piuttosto che **#define** per la sostituzione di valori.

const nei file di header

Per utilizzare **const** al posto di **#define**, si devono mettere le definizioni dei **const** all'interno dei file di header, così come viene fatto per i **#define**. In questo modo, si può inserire la definizione di un **const** in una sola posizione e utilizzarla negli altri sorgenti includendo l'header. Il comportamento standard di **const** in C++ è quello conosciuto come *internal linkage*, cioè un **const** è visibile solo all'interno del file dove viene definito e non può essere "visto" a tempo di link da altri file. È obbligatorio assegnare un valore ad un **const** quando questo viene definito, *tranne* nel caso in cui venga fatta una dichiarazione esplicita utilizzando **extern**:

```
extern const int bufsize;
```

Normalmente, il compilatore C++ cerca di evitare l'allocazione di memoria per un **const**, e ne conserva la definizione nella tabella dei simboli. Quando si utilizzano **extern** e **const** insieme, però, l'allocazione viene forzata (questo accade anche in altri casi, per esempio quando si utilizza l'indirizzo di un **const**). In questo caso deve essere allocato spazio in memoria, perché utilizzare **extern** è come comunicare al compilatore "usa il link esterno", vale a dire che più sorgenti possono far riferimento a questa grandezza, che, quindi, dovrà essere allocata in memoria.

Nel caso ordinario, quando **extern** non fa parte della definizione, l'allocazione di memoria non viene fatta. Quando **const** viene utilizzato, il suo valore viene semplicemente sostituito a tempo di compilazione.

L'obiettivo di non allocare memoria per un **const** non viene raggiunto anche nel caso di strutture complicate. Ogni qual volta il compilatore deve allocare memoria, la sostituzione del valore costante a tempo di compilazione non è possibile (visto che non v'è modo per il compilatore di conoscere con sicurezza il valore da memorizzare – perché se così fosse la memorizzazione stessa non sarebbe necessaria).

Proprio perché il compilatore non può impedire l'allocazione di memoria in ogni situazione, le definizioni dei **const** *devono* utilizzare l'internal link come default, che implica il link solo all'*interno* del file in cui è presente la definizione. Se fosse altrimenti, si

otterrebbero degli error di link quando si utilizzano **const** complicato, perché questo implicherebbe l'allocazione di memoria per file **cpp** diversi. Il linker troverebbe la stessa definizione in diversi file oggetto, e genererebbe errore. Grazie all'internal linkage di default, il linker non prova a linkare le definizione dei **const** tra file diversi, e quindi non si hanno collisioni. Con i tipi built-in, che sono quelli maggiormente utilizzati nelle espressioni costanti, il compilatore può sempre effettuare il constant folding.

Const e sicurezza

L'uso di **const** non è limitato alla sostituzione del **#define** nelle espressioni costanti. Se si inizializza una variabile con un valore prodotto a runtime e si è sicuri che questo valore non cambierà durante tutta la vita della variabile, è una buona abitudine dichiarare questa grandezza come **const**, in maniera da ricevere un errore dal compilatore qualora si provi accidentalmente a cambiarne il valore. Ecco un esempio:

```
//: C08:Safecons.cpp
// Utilizzo di const per sicurezza
#include <iostream>
using namespace std;

const int i = 100;  // Costante tipica
const int j = i + 10; // Valore derivante da un'espressione costante
long address = (long)&j; // Forza l'allocazione di memoria
char buf[j + 10]; // Ancora un'espressione costante

int main() {
  cout << "type a character & CR:";
  const char c = cin.get(); // Non può cambiare
  const char c2 = c + 'a';
  cout << c2;
  // ...
} ///:~
```

Si può notare che **i** è una costante a tempo di compilazione, mentre **j** viene calcolata a partire da **i**. Ciononostante, siccome **i** è una costante, il valore calcolato per **j** proviene da un'espressione costante e quindi è anch'esso una costante a tempo di compilazione. La riga successiva utilizza l'indirizzo di **j** e quindi forza l'allocazione di memoria da parte del compilatore. Questo, però, non impedisce l'utilizzo di **j** per determinare la dimensione di **buf**, perché il compilatore sa che **j** è **const** e quindi il suo valore è valido anche se, ad un certo punto del programma, è stata allocata memoria per contenerlo.

Nel **main()**, si può osservare un altro utilizzo di **const** con l'identificatore **c**, in questo caso il valore non può essere conosciuto a tempo di compilazione. Questo significa che ne verrà richiesta la memorizzazione e il compilatore non manterrà nessun riferimento nella tabella dei simboli (questo comportamento è lo stesso che si ha nel linguaggio C). L'inizializzazione deve essere fatta nel punto in cui viene fatta la definizione, ed una volta fatta il valore non può essere più cambiato. Si può vedere come **c2** viene calcolato a partire da **c** e come **const**, in questo caso, funzioni come in quelli precedenti – un vantaggio in più rispetto all'uso di **#define**.

Come suggerimento pratico, se si pensa che un valore non dovrebbe cambiare, lo si può dichiarare **const**. Questo non solo protegge da variazioni inavvertite, ma permette al compilatore di generare codice più efficiente eliminando allocazione di memoria e letture dalla memoria stessa.

Aggregati

È possibile utilizzare **const** con gli aggregati; in questo caso è molto probabile che il compilatore non sia abbastanza raffinato da poter mettere un aggregato nella sua tabella dei simboli, e quindi verrà allocata memoria. In questi casi, **const** ha il significato di "porzione di memoria che non può cambiare." Il suo valore, però, non può essere utilizzato a tempo di compilazione, perché il compilatore non conosce il contenuto della memoria in questa fase. Nel codice che segue, si possono vedere delle istruzioni illegali:

```
//: C08:Constag.cpp
// Costanti e aggregati
const int i[] = { 1, 2, 3, 4 };
//! float f[i[3]]; // Illegale
struct S { int i, j; };
const S s[] = { { 1, 2 }, { 3, 4 } };
//! double d[s[1].j]; // Illegale
int main() {} ///:~
```

In una definizione di array, il compilatore deve essere in grado di generare il codice che muove il puntatore allo stack per inserire l'array. In entrambe le definizioni illegali proposte sopra, il compilatore segnala errore, perché non può trovare un'espressione costante nella definizione degli array.

Differenze con il C

Le costanti vennero introdotte nelle prime versioni di C++ quando le specifiche del C Standard erano state appena completate. Ciononostante il C commitee decise di includere **const** nel C, prendendo il significato di "variabile ordinaria che non può cambiare". In C, un **const**, occupa sempre memoria ed il suo nome è globale. Il compilatore C non può trattare un **const** come una costante a tempo di compilazione. In C, se si scrive:

```
const int bufsize = 100;
char buf[bufsize];
```

verrà restituito un errore, anche se sembra una cosa ragionevole da fare. Questo avviene perché siccome **bufsize** occupa memoria da qualche parte, il compilatore C non può conoscerne il valore a tempo di compilazione. In C si può anche scrivere

```
const int bufsize;
```

mentre in C++ questo non è consentito, e il compilatore C accetta questa come una dichiarazione che indica che c'è della memoria allocata da qualche parte. Siccome il C utilizza come default l'external linkage per i **const**, questa dichiarazione ha senso. Il default C++ è quello di utilizzare l'internal linkage per i **const**, quindi se si vuole ottenere lo stesso comportamento, bisogna esplicitamente passare all'external linkage utilizzando **extern**.

```
extern const int bufsize; // Solo dichiarazione senza
inizializzazione
```

Questa dichiarazione è corretta anche in C.

In C++, un **const** non alloca necessariamente memoria. In C un **const** alloca sempre memoria. L'allocazione o meno di memoria per un **const** in C++ dipende dall'uso che ne viene fatto. In generale, se un const viene utilizzato semplicemente per sostituire un nome con un valore (come se si volesse utilizzare un **#define**), allora non viene allocata memoria. Se non viene allocata memoria (questo dipende dalla complessità del tipo di dato e dalla raffinatezza del compilatore), i valori possono essere inseriti all'interno del codice per aumentarne l'efficienza dopo il controllo di tipo, non prima, come avviene con **#define**. Se, invece, si utilizza, un indirizzo di un **const** (anche non intenzionalmente, passando il **const** ad una funzione come argomento passato per riferimento) oppure si definisce come **extern**, la memoria verrà allocata.

In C++, un **const** che si trova all'esterno di tutte le funzioni ha una visibilità estesa a tutto il file (i.e., non è visibile fuori dal file). Il comportamento di default, quindi, comporta l'utilizzo dell'internal linkage. Questo comportamento è molto diverso rispetto a quanto avviene in C++ per tutti gli altri identificatori (e a ciò che avviene per un **const** in C!) per i quali il default è l'external linkage. Se, in due file differenti, si dichiarano due **const** con lo stesso nome ma non **extern**, e non se ne utilizza mai l'indirizzo, il compilatore C++ ideale non allocherà memoria per i **const** e sostituirà semplicemente i valori nel codice. Dato che **const** è implicitamente visibile a livello di file, può essere inserito nei file di header in C++, senza conflitti durante il link.

Per forzare l'external linakage di un **const**, in maniera tale da poterlo utilizzare in un altro file, si deve esplicitamente definire come **extern**:

```
extern const int x = 1;
```

Si noti che, avendo fatto l'inizializzazione e avendo utilizzato **extern**, si forza l'allocazione di memoria per il **const** (anche se il compilatore potrebbe avere l'opzione di sostituirne il valore). L'inizializzazione caratterizza l'istruzione precedente come una definizione piuttosto che una dichiarazione. La dichiarazione:

```
extern const int x;
```

in C++ indica che la definizione esiste da un'altra parte (anche in questo caso, questo non è necessariamente vero in C). Si può capire, adesso, perché il C++ richiede che la definizione di un **const** ne inizializzi anche il valore; l'inizializzazione distingue una dichiarazione da una definizione (in C è sempre una definizione, quindi l'inizializzazione non è necessaria). Con la **extern const**, il compilatore non conosce il valore della costante e quindi non può farne la sostituzione all'interno del codice.

L'approccio del C alle costanti non è molto utile, e se si vuole usare un valore simbolico all'interno di un'espressione costante (valore che deve essere valutato durante la compilazione), in C si è quasi obbligati all'utilizzo della direttiva **#define**.

Puntatori

Anche i puntatori possono essere **const**. Il compilatore cercherà di prevenire l'allocazione di memoria e di fare la sostituzione del valore anche per i puntatori **const**, ma, in questo caso, questa proprietà sembra essere meno utile. La cosa importante è che il compilatore segnalerà ogni tentativo di cambiare il valore di un puntatore **const**, la qual cosa aumento il grado di sicurezza del codice.

Quando si usa **const** con i puntatori, si hanno due opzioni: **const** può essere applicato a quello che viene puntato dal puntatore, oppure può essere applicato all'indirizzo contenuto nel puntatore stesso. La sintassi per questi due casi può sembrare poco chiara in principio, ma diventa semplice dopo aver fatto un po' di pratica.

Puntare ad un const

Il trucco per comprendere la definizione di un puntatore, valido per ogni definizione complicata, è di leggerla partendo dall'identificatore procedendo verso l'"esterno". Lo specificatore **const** viene messo vicino alla cosa alla quale si fa riferimento. Quindi, se si vuole impedire che l'elemento al quale si sta puntando venga cambiato, bisogna scrivere una definizione di questo tipo:

```
const int* u;
```

Partendo dall'identificatore, si legge "**u** è un puntatore, che punta ad un **const int**". In questo caso non è richiesta alcuna inizializzazione, perché **u** può puntare ovunque (infatti non è un **const**), ed è la cosa alla quale punta che non può essere cambiata.

Ed ora arriva la parte che porta a confondersi. Si potrebbe pensare che per creare un puntatore costante, cioè per impedire ogni cambiamento dell'indirizzo contenuto in u, si debba semplicemente mettere **const** alla destra di **int**, in questo modo:

```
int const* v;
```

Non è del tutto assurdo pensare che l'istruzione precedente vada letta come "**v** è un puntatore **const** ad un **int**". Ciononostante, il modo in cui va letta è "**v** è un puntatore ordinario ad un **int** che si comporta come **const**". Anche in questo caso, quindi, **const** è riferito ad **int**, e l'effetto è lo stesso della definizione precedente. Il fatto che queste due definizioni coincidano porta confusione; per prevenire ciò, probabilmente si dovrebbe utilizzare solo la prima forma.

Puntatore const

Per rendere **const** il puntatore stesso, si deve mettere lo specificatore **const** alla destra del simbolo *, in questo modo:

```
int d = 1;
int* const w = &d;
```

In questo caso si legge: "**w** è un puntatore, **const**, che punta ad un **int**". Siccome ora è il puntatore stesso ad essere **const**, il compilatore richiede che questo abbia un valore iniziale che non potrà essere cambiato per tutta la "vita" del puntatore. È permesso, però, cambiare ciò a cui si punta scrivendo:

```
*w = 2;
```

Si può anche creare un puntatore **const** ad un oggetto **const** usando entrambe le forme corrette:

```
int d = 1;
const int* const x = &d;  // (1)
```

```
int const* const x2 = &d; // (2)
```

In questo caso ne il puntatore ne l'oggetto puntato possono cambiare valore.

Qualcuno asserisce che la seconda forma è più coerente, perché **const** viene sempre messo alla destra di ciò che modifica. Ognuno deciderà quale delle due forme permesse risulti più chiara per il proprio stile di programmazione.

Di seguito vengono riproposte le righe precedenti in un file compilabile:

```
//: C08:ConstPointers.cpp
const int* u;
int const* v;
int d = 1;
int* const w = &d;
const int* const x = &d;  // (1)
int const* const x2 = &d; // (2)
int main() {} ///:~
```

Formattazione

In questo libro si è deciso di mettere ogni definizione di puntatore su una linea diversa, e inizializzare ogni puntatore nel punto in cui viene definito se è possibile. Per questo motivo, la scelta di "attaccare" il simbolo '*' al tipo di dato è possibile:

```
int* u = &i;
```

come se **int*** fosse un tipo a parte. Questo rende il codice più semplice da comprendere, ma, sfortunatamente, questo non il modo reale in cui le cose funzionano. Il simbolo '*' si riferisce all'identificatore e non al tipo. Può essere messo ovunque tra il nome del tipo e l'identificatore. Quindi si può fare:

```
int *u = &i, v = 0;
```

per creare un **int* u**, come per l'istruzione precedente, e un **int v**. Dato che questo potrebbe confondere il lettore, è meglio seguire la forma utilizzata in questo libro.

Assegnazione e type checking

Il C++ è molto particolare per quanto riguarda il type checking, e questa particolarità si applica anche alle assegnazioni dei puntatori. Si può assegnare un oggetto non-**const** ad un puntatore **const**, perché si è semplicemente promesso di non cambiare qualcosa che in verità potrebbe cambiare. Non si può, invece, assegnare l'indirizzo di un oggetto **const** ad un puntatore non-**const**, perché si potrebbe cambiare l'oggetto attraverso il puntatore. Si può usare il cast per forzare questo tipo di assegnazione; questa, però, è pessima tecnica di programmazione, perché si sta violando la proprietà di invariabilità dell'oggetto, insieme al meccanismo di sicurezza implicito nell'uso di **const**. Per esempio:

```
//: C08:PointerAssignment.cpp
int d = 1;
const int e = 2;
int* u = &d; // OK -- d non è const
//! int* v = &e; // Illegale -- e è const
int* w = (int*)&e; // Legale, ma pericoloso
```

```
int main() {} ///:~
```

Anche se il C++ aiuta a prevenire gli errori, non protegge da se stessi quando si viola esplicitamente un meccanismo di sicurezza.

Array di caratteri letterali

Gli array di caratteri letterali rappresentano il caso in cui la stretta invariabilità non viene forzata. Si può scrivere:

```
char* cp = "howdy";
```

ed il compilatore accetterà questa istruzione senza segnalazioni di errore. Questo è tecnicamente un errore, perché un array di caratteri letterali (**"howdy"** in questo caso) viene creato dal compilatore come un array di caratteri costanti, e **cp** ne rappresenta l'indirizzo di partenza in memoria. Modificare uno qualsiasi dei caratteri nell'array comporterà un errore a runtime, anche se non tutti i compilatori si segnalano l'errore durante la compilazione.

Quindi gli array di caratteri letterali sono, di fatto, degli array di caratteri costanti. Ciononostante il compilatore permette che vengano trattati come non-**const**, perché c'è molto codice C che sfrutta questo comportamento. Quindi se si provano a cambiare i valori in un array di caratteri letterali, il comportamento non è definito, anche se probabilmente funzionerà su molte macchine.

Se si vuole modificare la stringa, basta metterla in un array:

```
char cp[] = "howdy";
```

Dato che i compilatori spesso si comportano nello stesso modo nei due casi, ci si dimentica di usare quest'ultima forma e spesso si ottengono strani funzionamenti.

Argomenti di funzioni e valori restituiti da funzioni

L'uso di **const** per specificare gli argomenti delle funzioni e i valori restituiti da queste, è un altro caso dove il concetto di costante può essere confuso. Se si stanno passando oggetti *per valore*, **const** non ha alcun significato per il chiamante (perché in questo caso si sta specificando che la funzione chiamante non può modificare l'argomento che gli è stato passato). Se si restituisce per valore un oggetto di tipo definito dall'utente come **const**, vuol dire che il valore restituito non può essere modificato. Se si stanno passando oppure restituendo *indirizzi*, l'utilizzo di **const** è un "promessa" che ciò che è contenuto a quel determinato indirizzo non verrà cambiato.

Passaggio di un parametro come valore const

Quando si passano degli argomenti per valore, questi possono essere specificati come **const** in questo modo:

```
void f1(const int i) {
  i++; // Illegale - errore durante la compilazione
```

```
}
```

ma cosa vuol dire? Si sta promettendo che il valore originale della variabile non verrà cambiato dalla funzione **f1().** Dato che l'argomento viene passato per valore, viene fatta immediatamente una copia della variabile originale, quindi la promessa viene implicitamente mantenuta dalla funzione chiamata.

Ma all'interno della funzione, **const** ha un altro significato: l'argomento non può essere cambiato. Quindi è uno strumento messo a disposizione del creatore della funzione, e non ha alcun significato per il chiamante.

Per non creare confusione in chi utilizza la funzione, si può trasformare l'argomento in **const** *all'interno* della funzione, piuttosto che nella lista degli argomenti. Questo può essere fatto utilizzando un puntatore, ma con un *riferimento*, un argomento che verrà trattato nel capitolo 11: in questo modo si ottiene una sintassi più leggibile. Brevemente, un riferimento è come un puntatore costante al quale viene applicato automaticamente l'operatore '*', quindi si comporta come un alias dell'oggetto. Per creare un riferimento, si deve utilizzare **&** nella definizione. Quindi, per non creare confusione, la funzione precedente diventa:

```
void f2(int ic) {
  const int& i = ic;
  i++;  // Illegale - errore durante la compilazione
}
```

Anche in questo caso si avrà un messaggio d'errore, ma questa volta la costanza dell'oggetto non è esposta nell'interfaccia della funzione; la costanza viene nascosta al chiamante avendo significato solo per chi sviluppa la funzione.

Restituire un valore const

Un comportamento simile a quello visto sopra si ha per il valore restituito dalla funzione. Se si crea una funzione che restituisce un valore **const**:

```
const int g();
```

si sta promettendo che la variabile originale (quella all'interno della funzione) non verrà cambiata. Ma anche in questo caso, restituendo la variabile per valore, questa viene copiata e quindi il valore originale non potrebbe essere modificato in nessun modo attraverso il valore restituito.

In un primo momento può sembrare che questo comportamento faccia si che **const** non abbia alcun significato. Si può vedere l'apparente perdita di effetto che si ha restituendo un valore **const** in questo esempio:

```
//: C08:Constval.cpp
// Restituire const per valore
// no ha significato per tipi built-in

int f3() { return 1; }
const int f4() { return 1; }

int main() {
  const int j = f3(); // Funziona
```

```
  int k = f4(); // Ma funziona anche questo!
} ///:~
```

Per i tipi built-in, non cambia nulla quando si restituisce un **const** per valore, quindi, in questi casi, si dovrebbe evitare di confondere l'utente ed eliminare **const**.

Restituire un valore come **const** diventa importante quando si lavora con tipi definiti dall'utente. Se una funzione restituisce un oggetto per valore come **const**, il valore restituito dalla funzione non può essere un lvalue (cioè, non si può comparire a sinistra di un'assegnazione o modificato in altro modo). Per esempio:

```
//: C08:ConstReturnValues.cpp
//  const restituito per valore
// Il risultato non può essere utilizzato come lvalue

class X {
  int i;
public:
  X(int ii = 0);
  void modify();
};

X::X(int ii) { i = ii; }

void X::modify() { i++; }

X f5() {
  return X();
}

const X f6() {
  return X();
}

void f7(X& x) { // Passaggio di un riferimento non-const
  x.modify();
}

int main() {
  f5() = X(1); // OK - valore restituito non-const
  f5().modify(); // OK
//!  f7(f5()); // Genera un warning oppure un errore
// Cause un errore a tempo di compilazione:
//!  f7(f5());
//!  f6() = X(1);
//!  f6().modify();
//!  f7(f6());
} ///:~
```

f5( ) restituisce un oggetto **X** non-**const**, mentre **f6()** restituisce un oggetto **X const**. Solo il valore restituito non-**const** può essere utilizzato come lvalue. È importante usare **const** quando si restituisce un oggetto per valore quando se ne vuole impedire l'utilizzo come lvalue.

La ragione per cui **const** non ha significato quando si restituisce per valore un tipo built-in è che il compilatore impedisce comunque che questo venga utilizzato come lvalue (perché è sempre un valore e non una variabile). Solo quando si restituisce per valore un oggetto definito dall'utente l'uso di **const** è significativo.

La funzione **f7()** accetta come argomento un *riferimento* non-**const** (i *riferimenti* sono un altro modo di gestire gli indirizzi in C++; questo argomento verrà trattato nel Capitolo 11). Lo stesso comportamento si ottiene utilizzando un puntatore non-**const**; la differenza risiede solo nella sintassi. La ragione per cui in C++ non si riesce a compilare è dovuta alla creazione di un oggetto temporaneo.

Oggetti temporanei

A volte, durante la valutazione di un'espressione, il compilatore deve creare degli *oggetti temporanei*. Questi oggetti come tutti gli altri: hanno bisogno di spazio in memoria e devono essere costruiti e distrutti. La differenza è che non si vedono – è il compilatore che decide quando servono e quali debbano essere le loro proprietà. Gli oggetti temporanei hanno la caratteristica di essere creati automaticamente come **const**. Questo viene fatto perché tipicamente non si vuole permettere la modifica di un oggetto temporaneo; forzare il cambiamento di un oggetto temporaneo rappresenta quasi sempre un errore. Rendendo automaticamente tutti gli oggetti temporanei **const**, il compilatore genera un avviso quando viene commesso questo errore.

Nell'esempio precedente, **f5()** restituisce un oggetto **X** non-**const**. Ma nell'espressione:

```
f7(f5());
```

il compilatore deve creare un oggetto temporaneo per conservare il valore restituito da **f5()**, in modo da poterlo passare a **f7()**. Questo andrebbe bene se **f7()** accettasse l'argomento per valore; in questo caso l'oggetto temporaneo verrebbe copiato all'interno di **f7()** e le modifiche fatte sulla copia non interesserebbero l'**X** temporaneo. **f7()**, invece, accetta un argomento passato come *riferimento*, vale a dire che in questo esempio gli viene passato l'indirizzo dell'oggetto temporaneo. Dato che **f7()** non accetta il suo argomento come riferimento **const**, può modificare l'oggetto temporaneo. Il compilatore, però, sa che l'oggetto temporaneo scomparirà appena la valutazione dell'espressione sarà terminata, quindi ogni modifica fatta all'oggetto **X** temporaneo verrà persa. Creando tutti gli oggetti temporanei automaticamente come **const**, questa situazione viene segnalata da un messaggio durante la compilazione, evitando quella che potrebbe essere una ricerca d'errore molto difficile.

Si notino, ora, le seguenti espressioni corrette:

```
f5() = X(1);
f5().modify();
```

Sebbene queste superino l'ispezione del compilatore, creano comunque problemi. **f5()** restituisce un oggetto **X** e il compilatore, per eseguire le due espressioni, deve creare un oggetto temporaneo e conservare il valore restituito. In entrambe le espressioni, quindi, l'oggetto temporaneo viene modificato, ed una volta che l'ultima riga è stata eseguita, viene cancellato. Il risultato è che le modifiche vengono perse, quindi questo codice contiene probabilmente degli errori – ma il compilatore non genera ne errori ne warning. Ambiguità di questo tipo sono abbastanza semplici da individuare; quando le cose sono più complesse gli errori potrebbero annidarsi in queste falle del compilatore.

Il modo in cui la costanza degli oggetti viene preservata verrà illustrato più avanti in questo capitolo.

Passare e restituire indirizzi

Se si passa o restituisce un indirizzo (che sia un puntatore oppure un riferimento), il programmatore utente può modificare il valore originale dell'oggetto puntato. Se si crea il puntatore o il riferimento come **const** si impedisce che questo avvenga, la qual cosa potrebbe far risparmiare molte preoccupazioni. Di fatto, ogni volta che si passa un indirizzo ad una funzione andrebbe fatto utilizzando **const**, qualora ciò fosse possibile. In caso contrario, si sta escludendo la possibilità di utilizzare la funzione con qualsiasi cosa sia **const**.

La scelta di restituire come **const** un puntatore piuttosto che un riferimento dipende da cosa si vuole che il programmatore utente possa fare con il valore restituito. Di seguito è riportato un esempio che mostra l'uso di puntatori **const** come argomenti di funzioni e valori restituiti:

```
//: C08:ConstPointer.cpp
// Puntatori costanti come argomenti/valori restituiti

void t(int*) {}

void u(const int* cip) {
//!  *cip = 2; // Illegale - valore modificato
  int i = *cip; // OK - copia il valore
//!  int* ip2 = cip; // Illegale: non-const
}

const char* v() {
  // Restituisce l'indirizzo di un array statico di caratteri:
  return "result of function v()";
}

const int* const w() {
  static int i;
  return &i;
}

int main() {
  int x = 0;
  int* ip = &x;
  const int* cip = &x;
  t(ip);  // OK
//!  t(cip); // Non OK
  u(ip);  // OK
  u(cip); // OK
//!  char* cp = v(); // Non OK
  const char* ccp = v(); // OK
//!  int* ip2 = w(); // Non OK
  const int* const ccip = w(); // OK
  const int* cip2 = w(); // OK
//!  *w() = 1; // Non OK
} ///:~
```

La funzione **t()** accetta un puntatore non-**const** come argomento, mentre **u()** accetta un puntatore **const**. All'interno della funzione **u()** non è consentito modificare la destinazione di un puntatore **const**, mentre è consentito copiare l'informazione puntata in una variabile non-**const**. Il compilatore impedisce anche di creare un puntatore non-**const** che utilizzi l'indirizzo memorizzato dal puntatore **const**.

Le funzioni **v( )** e **w( )** controllano la semantica del valore restituito. **v()** restituisce un **const char*** creato attraverso un array di caratteri letterali. Questa istruzione, di fatto, genera l'indirizzo dell'array di caratteri letterali, dopo averlo creato e memorizzato nella memoria statica. Come visto precedentemente, questo array di caratteri è tecnicamente una costante, quindi è corretto utilizzarlo come valore restituito da **v()**.

Il valore restituito da **w()** richiede che entrambi il puntatore e l'elemento puntato siano dei **const**. Come per **v()**, il valore restituito da **w()** è valido solo perché è **static**. È sbagliato restituire puntatori a variabili allocate nello stack, perché questi non saranno più validi dopo l'uscita dalla funzione e la "pulizia" dello stack. (Un altro puntatore che può essere restituito è l'indirizzo di una zona di memoria allocata nell'heap, che rimane valido anche dopo l'uscita dalla funzione.)

Nel **main()**, le funzioni vengono testate con argomenti diversi. Si può vedere come **t()** accetti un puntatore non-**const**, mentre quando si prova a passare come argomento un puntatore ad un **const**, dato che non c'è nessuna sicurezza che **t()** lasci l'oggetto puntato invariato, il compilatore restituirà un messaggio d'errore. L'argomento di **u()** è un puntatore **const**, quindi **u()** accetterà entrambi i tipi di puntatore. Una funzione che accetti un puntatore **const**, quindi, è più generale rispetto ad una che accetti un semplice puntatore.

Prevedibilmente, il valore restituito da **v()** può essere assegnato solo ad un puntatore a **const**. Ci si dovrebbe anche aspettare che il compilatore rifiuti di assegnare il valore restituito da **w()** ad un puntatore non-**const**, mentre accetti di utilizzare un **const int* const**. Può sorprendere il fatto che venga ritenuto valido anche un **const int***, anche se questo non corrisponde esattamente al tipo restituito. Anche in questo caso, siccome il valore (cioè l'indirizzo contenuto nel puntatore) viene copiato, la promessa di lasciare invariato il valore originale viene automaticamente rispettata. Il secondo **const** nell'espressione **const int* const**, in questo caso, è significativo solo quando si cerca di usare questo valore come lvalue, nel qual caso il compilatore si opporrà.

Passaggio di argomenti standard

In C è molto comune passare argomenti per valore, e quando si vuole passare un indirizzo non ci sono alternative all'uso del puntatore[43]. In C++ viene utilizzato un altro approccio. Quando si vuole passare un argomento lo si fa per riferimento, e in particolare utilizzando un riferimento **const**. Per il programmatore utente, la sintassi apparirà identica ad un passaggio per valore, non dovendo usare la sintassi confusa dei puntatori – in questo modo non bisogna far nessun ragionamento che riguardi i puntatori. Per il creatore della funzione, il passaggio di un indirizzo e praticamente sempre più conveniente rispetto al passaggio di un intero oggetto, e se viene passato come riferimenti **const**, la funzione scritta non potrà cambiare l'oggetto a cui si fa riferimento, quindi dal punto di vista dell'utente è come se il parametro venga passato per valore (ma in maniera più efficiente).

Sfruttando la sintassi dei riferimenti (dal punto di vista del chiamante è come se si trattasse di un passaggio per valore) è possibile passare un oggetto temporaneo ad una funzione che accetta un riferimento **const**, mentre non è possibile fare lo stesso per una funzione che accetti un puntatore , perché in questo caso l'indirizzo dovrebbe essere utilizzato in modo esplicito. Passare un parametro per riferimento, quindi, da luogo ad una comportamento che non si può ottenere in C: si può passare ad una funzione l'*indirizzo* di un oggetto temporaneo, che è sempre **const**. Questo è il motivo per cui, per passare ad una

funzione oggetti temporanei per riferimento, l'argomento deve essere un riferimento **const**. L'esempio seguente ne è una dimostrazione:

```
//: C08:ConstTemporary.cpp
// Gli oggetti temporanei sono const

class X {};

X f() { return X(); } // Restituito per valore

void g1(X&) {} // Passato per riferimento non-const
void g2(const X&) {} // Passato per riferimento const

int main() {
  // Errore: temporaneo const creato da f():
//!  g1(f());
  // OK: g2 accetta un riferimento const:
  g2(f());
} ///:~
```

f() restituisce un oggetto della **classe X** *per valore*. Questo significa che, quando si passa immediatamente il valore restituito da **f()** ad un'altra funzione, come nelle chiamate a **g1( )** e **g2( )**, viene creato un oggetto temporaneo **const**. La chiamata all'interno di **g1()** è un errore, perché **g1()** non accetta un riferimento **const**, mentre nel caso di **g2()** la chiamata è corretta.

Classi

In questa sezione viene mostrato come utilizzare **const** con le classi. Si potrebbe voler creare un **const** locale all'interno di una classe per poterlo utilizzare nella valutazione di espressioni constanti, le quali verrebbero valutate a tempo di compilazione. In ogni caso, il significato di **const** all'interno delle classi può essere diverso, quindi si devono comprendere le varie opzioni che ci sono per creare membri **const** di una classe.

Si può anche definire un intero oggetto come **const** (e, come è stato visto, il compilatore crea sempre gli oggetti temporanei come **const**). Preservare la costanza di un oggetto, però, è più difficile. Il compilatore può assicurare la costanza di un tipo built-in, ma non può controllare una classe nel suo complesso. Per garantire la costanza di un oggetto, vengono introdotte le funzioni membro **const**: solo una funzione membro **const** può essere chiamata da un oggetto **const**.

const nelle classi

Si potrebbe voler utilizzare **const** per valutare espressioni costanti all'interno delle classi. L'esempio tipico è quando si crea un array all'interno di una classe e si vuole utilizzare un **const** al posto di un **#define** per stabilire la dimensione dell'array e per utilizzarlo nelle operazioni che coinvolgono l'array stesso. Si vuole tenere nascosto l'array all'interno della classe, così se si usa un nome come **size**, per esempio, si potrebbe utilizzare lo stesso nome in un'altra classe senza avere conflitti. Il preprocessore tratta tutti i **#define** come globali a partire dal punto in cui sono stati definiti, quindi non possono essere utilizzati per raggiungere lo stesso scopo.

Si potrebbe pensare che la scelta più logica dove mettere un **const** sia all'interno della classe. Questa scelta, però, non produce il risultato desiderato. All'interno di una classe, **const** riassume, anche se parzialmente, il significato che ha in C: permette di allocare memoria all'interno di un oggetto e rappresenta un valore che, una volta inizializzato, non può più essere cambiato. L'uso di **const** all'interno di una classe significa "Questo è costante per tutta la vita dell'oggetto." Ogni oggetto, però, può avere un valore diverso per questa costante.

Questo è il motivo per cui, quando si crea un **const** ordinario (non-**static**) all'interno di una classe, non gli si può assegnare un valore iniziale. L'inizializzazione deve avvenire sicuramente nel costruttore, ma in un punto particolare. Dato che un **const** deve essere inizializzato nel punto in cui viene creato, all'interno del corpo del costruttore il **const** deve *già* essere stato inizializzato. In caso contrario si avrebbe la libertà di effettuare l'inizializzazione in punto qualsiasi del costruttore, il che significa che il **const** potrebbe essere non inizializzato per un periodo. Inoltre, non ci sarebbe nessuno che impedirebbe di cambiare il valore del **const** in vari punti del costruttore.

La lista di inizializzazione del costruttore

Il punto speciale in cui effettuare l'inizializzazione viene chiamato *lista di inizializzazione del costruttore*, ed è stato sviluppato originalmente per essere utilizzato con l'ereditarietà (questo argomento è trattato nel Capitolo 14). La lista di inizializzazione del costruttore – che, come il nome indica, è presente solo nella definizione del costruttore – è una lista di "chiamate a costruttori" che viene messa dopo la lista degli argomenti della funzione separata da un ":", e prima della parentesi che indica l'inizio del corpo del costruttore. Questo per ricordare che l'inizializzazione nella lista avviene prima che venga eseguito il codice del costruttore principale. È questo il punto dove inizializzare tutti i **const**. L'uso corretto di **const** all'interno di una classe è mostrato di seguito:

```
//: C08:ConstInitialization.cpp
// Inizializzare const all'interno di una classe
#include <iostream>
using namespace std;

class Fred {
  const int size;
public:
  Fred(int sz);
  void print();
};

Fred::Fred(int sz) : size(sz) {}
void Fred::print() { cout << size << endl; }

int main() {
  Fred a(1), b(2), c(3);
  a.print(), b.print(), c.print();
} ///:~
```

La forma della lista di inizializzazione del costruttore vista sopra può confondere,in un primo momento, perché non si è abituati a trattare un tipo built-in come se avesse un costruttore.

"Costruttori" per tipi built-in

Durante lo sviluppo del linguaggio, quando si aumentò lo sforzo fatto affinché i tipi definiti dall'utente si comportassero come dei tipi built-in, si notò che poteva essere utile anche che i tipi built-in si comportassero come tipi definiti dall'utente. Per questo motivo, nella lista di inizializzazione del costruttore, si può trattare un tipo built-in come se avesse un costruttore, in questo modo:

```
//: C08:BuiltInTypeConstructors.cpp
#include <iostream>
using namespace std;

class B {
  int i;
public:
  B(int ii);
  void print();
};

B::B(int ii) : i(ii) {}
void B::print() { cout << i << endl; }

int main() {
  B a(1), b(2);
  float pi(3.14159);
  a.print(); b.print();
  cout << pi << endl;
} ///:~
```

Questa caratteristica diventa essenziale quando si devono inizializzare membri **const**, perché questi devono essere inizializzati prima del corpo del costruttore.

Per generalizzare il caso, quindi, ha senso estendere il concetto di "costruttore" anche ai tipi built-in (nel qual caso equivale ad una semplice assegnazione) ed è questo il motivo per cui, nel codice precedente, la definizione **pi(3.14159)** è corretta.

Spesso è utile incapsulare un tipo built-in all'interno di una classe per garantirne l'inizializzazione attraverso il costruttore. Come esempio, di seguito è riportata la classe **Integer**:

```
//: C08:EncapsulatingTypes.cpp
#include <iostream>
using namespace std;

class Integer {
  int i;
public:
  Integer(int ii = 0);
  void print();
};

Integer::Integer(int ii) : i(ii) {}
void Integer::print() { cout << i << ' '; }

int main() {
  Integer i[100];
  for(int j = 0; j < 100; j++)
    i[j].print();
```

```
} ///:~
```

Gli array di **Integer** nel **main( )** vengono inizializzati automaticamente a zero. Questa modo di effettuare l'inizializzazione non è necessariamente meno efficiente di un ciclo **for** oppure di un **memset()**. Molti compilatori riescono ad ottimizzare questo codice ottenendo un eseguibile molto veloce.

Costanti a tempo di compilazione nelle classi

L'uso precedente di **const** è interessante ed utile in alcuni casi, ma non risolve il problema originale, vale a dire: "come si può ottenere una costante a tempo di compilazione all'interno di una classe?" La risposta richiede l'uso di un'ulteriore parola chiave, la quale verrà presentata in maniera completa nel Capitolo 10: **static**. La parola chiave **static**, in questo caso, prende il seguente significato: "questa è l'unica istanza di questo membro, a prescindere dal numero di oggetti creati per questa classe," che è proprio ciò che è necessario in questo caso: un membro della classe che sia costante, e che non possa cambiare da un oggetto ad un altro della stessa classe. Un **static const** di un tipo built-in, quindi, può essere trattato come una costante a tempo di compilazione.

Uno **static const**, quando viene utilizzato all'interno di una classe, si comporta in maniera strana: deve essere inizializzato nel punto in cui viene definito. Questo deve essere fatto solo per un **static const**; la stessa tecnica non funzionerebbe con un altro elemento, perché tutti gli altri membri devono essere inizializzati nel costruttore, oppure in altre funzioni membro.

L'esempio seguente mostra la creazione e l'utilizzo di un **static const** chiamato **size** all'interno di una classe che rappresenta uno stack di puntatori a stringa[44]:

```
//: C08:StringStack.cpp
// Utilizzo di static const per creare una
// costante a tempo di compilazione all'interno di una classe
#include <string>
#include <iostream>
using namespace std;

class StringStack {
  static const int size = 100;
  const string* stack[size];
  int index;
public:
  StringStack();
  void push(const string* s);
  const string* pop();
};

StringStack::StringStack() : index(0) {
  memset(stack, 0, size * sizeof(string*));
}

void StringStack::push(const string* s) {
  if(index < size)
    stack[index++] = s;
}

const string* StringStack::pop() {
  if(index > 0) {
    const string* rv = stack[--index];
```

```
    stack[index] = 0;
    return rv;
  }
  return 0;
}

string iceCream[] = {
  "pralines & cream",
  "fudge ripple",
  "jamocha almond fudge",
  "wild mountain blackberry",
  "raspberry sorbet",
  "lemon swirl",
  "rocky road",
  "deep chocolate fudge"
};

const int iCsz =
  sizeof iceCream / sizeof *iceCream;

int main() {
  StringStack ss;
  for(int i = 0; i < iCsz; i++)
    ss.push(&iceCream[i]);
  const string* cp;
  while((cp = ss.pop()) != 0)
    cout << *cp << endl;
} ///:~
```

Dato che **size** viene utilizzato per determinare la dimensione dell'array **stack**, di fatto è una costante a tempo di compilazione, ma viene nascosta all'interno della classe.

Si noti che: **push( )** accetta un **const string*** come argomento, **pop( )** restituisce un **const string***, e **StringStack** contiene **const string***. Se questo non fosse vero, non si potrebbe utilizzare un **StringStack** per contenere i puntatori di **iceCream**. Inoltre, in questo modo si impedisce che gli oggetti contenuti in **StringStack** vengano modificati. Ovviamente non tutti i contenitori vengono progettati con queste restrizioni.

"Enum hack" nel codice old-style

Nelle vecchie versioni del C++, non era permesso l'utilizzo di **static const** all'interno delle classi. Questo significava che **const** non poteva essere utilizzato per valutare un'espressione costante all'interno delle classi. Ciononostante, si voleva ottenere questo comportamento e una soluzione tipica (solitamente chiamata "enum hack") era quella di utilizzare un **enum** senza etichetta e senza istanze. Tutti i valori di un tipo **enum** devono essere stabiliti a tempo di compilazione, hanno validità solo all'interno della classe, e sono disponibili per valutare espressioni costanti. Quindi, si può comunemente incontrare:

```
//: C08:EnumHack.cpp
#include <iostream>
using namespace std;

class Bunch {
  enum { size = 1000 };
  int i[size];
};

int main() {
```

```
  cout << "sizeof(Bunch) = " << sizeof(Bunch)
       << ", sizeof(i[1000]) = "
       << sizeof(int[1000]) << endl;
} ///:~
```

In questo caso l'uso di **enum** non occupa memoria nell'oggetto, e i valori enumerati vengono tutti valutati a tempo di compilazione. I valori possono essere esplicitati anche in questo modo:

```
enum { one = 1, two = 2, three };
```

Con i tipi **enum** interi, il compilatore continuerà la numerazione a partire dall'ultimo valore, quindi **three** varrà 3.

Nell'esempio **StringStack.cpp** precedente, la linea:

```
static const int size = 100;
```

diverrebbe:

```
enum { size = 100 };
```

anche se si potrà incontrare spesso nel codice la tecnica che utilizza **enum** nel codice, **static const** è lo strumento previsto dal linguaggio per risolvere questo problema. Non c'è nessun obbligo che impone l'uso di **static const** al posto dell'"**enum** hack", ed in questo libro viene utilizzato l'"**enum** hack", perché, nel momento in cui il libro è stato scritto, questa tecnica è supportata da un numero maggiore di compilatori.

Oggetti const e funzioni membro

Le funzioni membro di una classe possono essere **const**. Cosa vuol dire? Per capirlo, bisogna aver assimilato bene il concetto di oggetti **const**.

Un oggetto **const** è definito allo stesso modo sia per un tipo definito dall'utente che per un tipo built-in. Per esempio:

```
const int i = 1;
const blob b(2);
```

In questo caso, **b** è un oggetto **const** di tipo **blob**. Il suo costruttore viene chiamato con un argomento pari a due. Affinché il compilatore possa forzare la costanza, deve assicurare che nessun dato membro dell'oggetto venga cambiato durante la vita dell'oggetto stesso. Può assicurare facilmente che nessun dato pubblico venga modificato, ma come può conoscere quale funzione membro cambierà i dati membro e quale, invece, sarà "sicura" per un oggetto **const**?

Se si dichiara una funzione membro **const**, si sta dicendo al compilatore che la funzione può essere chiamata per un oggetto **const**. Una funzione membro che non viene dichiarata specificatamente **const**, viene trattata come una che potenzialmente potrebbe modificare i dati membro di un oggetto, e il compilatore non permetterà che venga utilizzata con oggetti **const**.

Richiedere che una funzione membro sia **const** non garantisce che questa si comporterà nel modo esatto, quindi il compilatore obbliga a ripetere la specificazione **const** anche quando la funzione viene definita. (**const** diventa un tratto caratteristico della funzione, e sia il compilatore che il linker effettueranno dei controlli sulla costanza.) La costanza di un funzione viene forzata durante le definizione generando un messaggio d'errore quando si prova a cambiare un membro qualsiasi dell'oggetto, *oppure* quando si chiama un funzione membro non-**const**. Questo è il modo in cui viene garantito, all'interno della definizione, che ogni funzione dichiarata come **const** si comporti nel modo corretto.

Per comprendere qual è la sintassi da utilizzare per dichiarare una funzione membro **const**, si noti che la dichiarazione con **const** significa che il valore restituito è **const**, quindi questo non è il modo per ottenere il risultato desiderato. Lo specificatore **const** va messo *dopo* la lista degli argomenti. Per esempio,

```
//: C08:ConstMember.cpp
class X {
  int i;
public:
  X(int ii);
  int f() const;
};

X::X(int ii) : i(ii) {}
int X::f() const { return i; }

int main() {
  X x1(10);
  const X x2(20);
  x1.f();
  x2.f();
} ///:~
```

Si noti che la parola chiave **const** deve essere ripetuta nella definizione altrimenti il compilatore la considererà come una funzione differente. Dato che **f()** è una funzione membro **const**, se si cercherà di cambiare **i** oppure si chiamerà una funzione non-**const**, il compilatore segnalerà errore.

La chiamata ad una funzione membro **const** è sicura sia utilizzando oggetti **const** che oggetti non-**const**. Può essere considerata, quindi, come la forma più generale di funzione membro (ed è una sfortuna che questo non sia il default per una funzione membro). Ogni funzione che non modifichi i dati membro dovrebbe essere dichiarata come **const**, in modo tale da poter essere utilizzata con oggetti **const**.

Di seguito vengono messe a confronto funzioni membro **const** e non-**const**:

```
//: C08:Quoter.cpp
// Selezione di frasi casuali
#include <iostream>
#include <cstdlib> // Generatore di numeri casuali
#include <ctime> // Utile per fornire il seme al generatore di
numeri casuali
using namespace std;

class Quoter {
  int lastquote;
public:
  Quoter();
```

```
  int lastQuote() const;
  const char* quote();
};

Quoter::Quoter(){
  lastquote = -1;
  srand(time(0)); // Generatore casuale del seme
}

int Quoter::lastQuote() const {
  return lastquote;
}

const char* Quoter::quote() {
  static const char* quotes[] = {
    "Are we having fun yet?",
    "Doctors always know best",
    "Is it ... Atomic?",
    "Fear is obscene",
    "There is no scientific evidence "
    "to support the idea "
    "that life is serious",
    "Things that make us happy, make us wise",
  };
  const int qsize = sizeof quotes/sizeof *quotes;
  int qnum = rand() % qsize;
  while(lastquote >= 0 && qnum == lastquote)
    qnum = rand() % qsize;
  return quotes[lastquote = qnum];
}

int main() {
  Quoter q;
  const Quoter cq;
  cq.lastQuote(); // OK
//!  cq.quote(); // Non OK; funzione non const
  for(int i = 0; i < 20; i++)
    cout << q.quote() << endl;
} ///:~
```

Ne il costruttore ne il distruttore possono essere funzioni membro **const**, perché virtualmente effettuano sempre delle modifiche sull'oggetto durante l'inizializzazione ed il clean-up. Anche la funzione membro **quote()** non può essere **const**, perché modifica il dato membro **lastquote** (si confronti l'istruzione **return**). **lastQuote()**, invece, non effettua modifiche, quindi può essere **const** e può essere utilizzata in modo sicuro per l'oggetto **const cq**.

mutable: const bitwise contro const logiche

Cosa bisognerebbe fare se si volesse creare una funzione membro **const**, ma si volesse anche cambiare qualche dato all'interno dell'oggetto? Questa è quella che, a volte, viene chiamata differenza tra un **const** bitwise ed un **const** logico (o **const** *memberwise*). In un **const** bitwise ogni bit dell'oggetto è costante, quindi l'immagine bit a bit dell'oggetto non cambierà mai. In un **const** logico, l'intero oggetto è concettualmente costante, e si possono tollerare cambiamenti tra membri (cioè fatti da funzioni membro su dati membro dello stesso oggetto). Se si comunica al compilatore che un oggetto è **const**, il compilatore garantirà la costanza nell'accezione di bitwise. Per realizzare la costanza logica, esistono due tecniche per cambiare un dato membro all'interno di una funzione membro **const**.

Il primo approccio è quello storico e consiste nel *"castare via" la costanza*. Questo viene fatto in modo particolare. Si utilizza **this** (la parola chiave che restituisce l'indirizzo dell'oggetto corrente) e lo si "casta" ad un puntatore al tipo di oggetto considerato. Potrebbe sembrare che **this** sia *già* di questo tipo. All'interno di una funzione membro **const**, però, **this** è un puntatore **const**: effettuando il cast ad un puntatore ordinario, si può rimuovere la costanza. Ecco un esempio:

```
//: C08:Castaway.cpp
// "Castare via" la costanza

class Y {
  int i;
public:
  Y();
  void f() const;
};

Y::Y() { i = 0; }

void Y::f() const {
//!  i++; // Errore - funzione membro const
  ((Y*)this)->i++; // OK: costanza "castata via"
  // Migliore: utilizzo della sintassi esplicita di cast del C++:
  (const_cast<Y*>(this))->i++;
}

int main() {
  const Y yy;
  yy.f(); // Effettua un cambiamento!
} ///:~
```

Sebbene questa tecnica funzioni e venga utilizzata all'interno di codice corretto, non rappresenta la soluzione migliore. Il problema è che la perdita di costanza è nascosta nella definizione della funzione membro, l'interfaccia della classe non mostrerà esplicitamente che l'oggetto viene modificato, quindi l'unico modo per venire a conoscenza di questo comportamento è accedere al codice sorgente (e sospettare che la costanza sia stata "castata via"). Per esporre all'esterno queste informazioni, si dovrebbe utilizzare la parola chiave **mutable** nella dichiarazione della classe, per specificare che un particolare dato membro potrebbe cambiare all'interno di un oggetto **const**:

```
//: C08:Mutable.cpp
// La parola chiave "mutable"

class Z {
  int i;
  mutable int j;
public:
  Z();
  void f() const;
};

Z::Z() : i(0), j(0) {}

void Z::f() const {
//! i++; // Errore - funzione membro const
    j++; // OK: mutable
}

int main() {
```

```
  const Z zz;
  zz.f(); // Effettua un cambiamento!
} ///:~
```

In questo modo, l'utilizzatore di una classe può capire quali membri possono essere modificati in una funzione membro **const** guardando la dichiarazione della classe.

ROMability

Se un oggetto è definito come **const**, si candida ad essere memorizzato nella memoria a sola lettura (ROM, da cui ROMability); spesso questa considerazione è molto importante nella programmazione di sistemi embedded. Creare un oggetto **const** non è sufficiente – le caratteristiche necessarie per la ROMability sono più strette. L'oggetto deve essere un **const** bitwise, e non un **const** logico. Constatare se un oggetto è un **const** logico può essere fatto facilmente utilizzando la parola chiave **mutable**; altrimenti il compilatore probabilmente sarà in grado di determinare il tipo di **const**, controllando se la costanza viene "castata via" all'interno di una funzione membro. Bisogna anche considerare che:

1. La classe o la struttura (**struct**) non devono avere costruttori o distruttori definiti dall'utente.
2. Non ci possono essere classi base (argomento trattato nel Capitolo 14) o oggetti membro con costruttori o distruttori definiti dall'utente.

L'effetto di un'operazione di scrittura su una qualsiasi parte di un oggetto **const** ROMable non è definito. Sebbene un oggetto creato appositamente possa essere memorizzato nella ROM, nessun oggetto richiede di essere memorizzato in questa parte della memoria.

volatile

La sintassi di **volatile** è identica a quella utilizzata per **const**, ma il significato di **volatile** è: "Questo dato potrebbe cambiare in maniera non controllata dal compilatore." In qualche modo è l'"ambiente" esterno che cambia il dato (per esempio attraverso multitasking, multithreading o interrupt), e **volatile** comunica al compilatore di non fare nessuna assunzione sul dato, specialmente durante l'ottimizzazione.

Se il compilatore ha spostato un dato in un registro per leggerlo e non ha mai modificato il valore nel registro, quando viene richiesto lo stesso dato verrebbe utilizzato il valore nel registro senza accedere di nuovo alla memoria. Se il dato è **volatile**, il compilatore non può fare questa supposizione perché il dato potrebbe essere stato cambiato da un altro processo; quindi dovrà rileggere il dato piuttosto che ottimizzare il codice e rimuovere quella che, normalmente, sarebbe un lettura ridondante.

Gli oggetti **volatile** si creano utilizzando la stessa sintassi per la creazione di oggetti **const**. Si possono anche creare oggetti **const volatile**, i quali non possono essere cambiati da un programmatore che utilizza gli oggetti, ma vengono cambiati da un agente esterno. L'esempio riportato di seguito potrebbe rappresentare una classe associata con dell'hardware per la comunicazione:

```
//: C08:Volatile.cpp
// La parola chiave volatile

class Comm {
```

```
  const volatile unsigned char byte;
  volatile unsigned char flag;
  enum { bufsize = 100 };
  unsigned char buf[bufsize];
  int index;
public:
  Comm();
  void isr() volatile;
  char read(int index) const;
};

Comm::Comm() : index(0), byte(0), flag(0) {}

// è solo una demo; non funziona
// come interrupt service routine:
void Comm::isr() volatile {
  flag = 0;
  buf[index++] = byte;
  // Ritorna all'inizio del buffer:
  if(index >= bufsize) index = 0;
}

char Comm::read(int index) const {
  if(index < 0 || index >= bufsize)
    return 0;
  return buf[index];
}

int main() {
  volatile Comm Port;
  Port.isr(); // OK
//!  Port.read(0); // Errore, read() non è volatile
} ///:~
```

Come per **const**, si può utilizzare **volatile** per dati membro, funzioni membro, e oggetti. Per oggetti **volatile** possono essere chiamate solo funzioni membro **volatile**.

La ragione per la quale **isr()** non può essere utilizzata come interrupt service routine è che all'interno di una funzione membro, deve essere passato l'indirizzo all'oggetto corrente (**this**), e generalmente un ISR non accetta nessun argomento. Per risolvere questo problema, si può creare **isr()** come funzione membro **static**: questo argomento verrà trattato nel Capitolo 10.

La sintassi di **volatile** è identica a quella di **const**, per questo motivo le due parole chiave vengono presentate insieme. Si fa riferimento ad entrambe parlando di *qualificatori c-v*.

Sommario

La parola chiave **const** consente di definire oggetti, argomenti di funzione, valori restituiti e funzioni membro come costanti, ed eliminare l'utilizzo del preprocessore per la sostituzione di valori senza perdere nessuno dei vantaggi derivanti dal suo uso. In questo modo viene fornito un altro strumento per la sicurezza ed il controllo del codice. L'utilizzo della cosiddetta *const correctness* (cioè dell'uso di **const** ovunque sia possibile) può essere di grande aiuto nello sviluppo dei progetti.

Sebbene si possa ignorare l'uso di **const** e continuare ad utilizzare le tecniche di programmazione C, questo strumento viene messo a disposizione per aiutare il

programmatore. Dal Capitolo 11 in avanti si in inizierà a fare un grosso uso dei riferimenti, e si potrà constatare ancora di più quanto sia critico l'uso di **const** con gli argomenti delle funzioni.

---

[43] Qualcuno afferma che, in C, *tutto* viene passato per valore, dato che anche quando si passa un puntatore ne viene fatta una copia (quindi si sta passando il puntatore per valore). Anche se questo è corretto, penso che, in questo momento, potrebbe confondere le idee.

[44] Al momento della pubblicazione del libro non tutti i compilatori supportano questa funzionalità.

9: Funzioni Inline

Una delle caratteristiche importanti di C++ ereditata dal C è l'efficienza. Se l'efficienza del C++ è drasticamente minore del C, ci può essere una folta rappresentanza di programmatori che non può giustificare il suo uso.

In C, uno dei modi di preservare l'efficienza è attraverso l'uso di *macro*, queste permettono di fare ciò che appare come una chiamata ad una funzione ma senza l'overhead tipico della chiamata. La macro è implementata con il pre-processore invece che dalla stesso processore, e il pre-processore sostituisce tutte le chiamate alle macro direttamente con il codice delle macro, quindi non c'è costo derivante dal pushing degli argomenti, esecuzione di una CALL del linguaggio assembler, il ritorno di argomenti e l'esecuzione di un RETURN di linguaggio assembler. Tutto il lavoro è eseguito dal pre-processore, così si ha la convenienza e la leggibilità di una chiamata a funzione senza costi aggiuntivi.

Ci sono due problemi con l'uso delle macro in C++. Il primo già cruciale in C: una macro assomiglia a una chiamata di funzione, ma non sempre agisce come tale. Questo può "nascondere" errori difficili da trovare. Il secondo problema è specifico di C++: il pre-processore non ha il permesso di accedere a dati delle classi membro. Questo significa che le macro non possono essere usate come funzioni di classi membro.

Per mantenere l'efficienza delle macro, ma aggiungere la sicurezza e lo scope di classe di una vera funzione,il C++ offre le *funzioni inline*. In questo capitolo, tratteremo dei problemi delle macro in C++, di come questi siano stati risolti con le funzioni inline, delle linee guida e delle intuizioni del modo di lavorare *inline*.

Le insidie del pre-processore

Il punto chiave dei problemi delle macro è che ci si può ingannare nel pensare che il comportamento del pre-processore è lo stesso del compilatore. Naturalmente è stato spiegato che una macro è simile e lavora come una chiamata a funzione, per cui è facile cadere in quest' equivoco. Le difficoltà iniziano quando si affrontano le sottili differenze.

Come semplice esempio, si consideri il seguente:

```
#define F (x) (x + 1)
```

Ora, se viene fatta una chiamata **F** come questa

```
F(1)
```

il preprocessore la sviluppa, piuttosto inaspettatamente, nella seguente:

```
(x) (x + 1)(1)
```

Il problema sorge a causa della distanza tra **F** e la sua parentesi aperta nella definizione della macro. Quando questo spazio viene rimosso, si può effettivamente *chiamare* la macro con lo spazio

```
F (1)
```

ed essa sarà sviluppata correttamente in

```
(1 + 1)
```

L'esempio precedente è abbastanza insignificante e il problema si renderà evidente subito. Le difficoltà reali nasceranno quando saranno usate espressioni come argomenti nelle chiamate a macro.

Ci sono due problemi. Il primo e che le espressioni possono essere sviluppate all'interno delle macro per cui l'ordine di valutazione è differente da ciò che normalmente ci si aspetta. Per esempio,

```
#define FLOOR(x,b) x>=b?0:1
```

Ora, se vengono usate espressioni per gli argomenti

```
if(FLOOR(a&0x0f,0x07)) // ...
```

la macro sarà sviluppata come

```
if(a&0x0f>=0x07?0:1)
```

La precedenza di **&** è minore rispetto a quella di **>=**, quindi la valutazione della macro ci sorprenderà. Una volta scoperto il problema, lo si può risolvere mettendo parentesi intorno ad ogni cosa nella definizione della macro (questa è una buona regola da usare quando si creano macro). Quindi,

```
#define FLOOR(x,b) ((x)>=(b)?0:1)
```

Scoprire il problema può essere difficile, comunque, e si può non trovarlo finché non si è dato per scontato l'esatto comportamento della macro. Nella versione senza parentesi della precedente macro, molte espressioni lavoreranno correttamente perché la precedenza di >= è minore di molti operatori come +, /, –, e anche degli operatori di shift su bit. Quindi si può facilmente iniziare a pensare che essa lavori con tutte le espressioni, incluse quelle che usano gli operatori logici su bit.

Il problema precedente può essere risolto con un'attenta pratica di programmazione: mettere le parentesi a tutto nelle macro. Tuttavia, la seconda difficoltà è più insidiosa. Diversamente da una funzione normale, ogni volta che si usa un argomento in una macro, questo viene valutato. Finché la macro è chiamata solo con variabili ordinarie, questa valutazione non avra effetti negativi, ma se la valutazione di un argomento ha effetti collaterali, allora i risultati possono essere sorprendenti e il comportamento non seguirà affatto quello di una funzione.

Per esempio, questa macro determina se il suo argomento cade all'interno di un certo intervallo:

```
#define BAND(x) (((x)>5 && (x)<10) ? (x) : 0)
```

Finché si usa un argomento "ordinario", la macro lavorerà praticamente come una funzione reale. Ma non appena si diventa meno attenti pensando che sia una funzione reale, cominciano i problemi. Quindi:

```
//: C09:MacroSideEffects.cpp
#include "../require.h"
#include <fstream>
using namespace std;

#define BAND(x) (((x)>5 && (x)<10) ? (x) : 0)

int main() {
  ofstream out("macro.out");
  assure(out, "macro.out");
  for(int i = 4; i < 11; i++) {
    int a = i;
    out << "a = " << a << endl << '\t';
    out << "BAND(++a)=" << BAND(++a) << endl;
    out << "\t a = " << a << endl;
  }
} ///:~
```

Da notare l'uso di caratteri maiuscoli per il nome della macro. Questa è una regola utile perché dice al lettore che essa è una macro e non una funzione, così se ci sono problemi o dubbi ciò funziona da piccolo promemoria.

Ecco l'output prodotto dal programma, che non è niente affatto ciò che ci si sarebbe aspettato da una vera funzione:

```
a = 4
  BAND(++a)=0
   a = 5
a = 5
  BAND(++a)=8
   a = 8
a = 6
  BAND(++a)=9
   a = 9
a = 7
  BAND(++a)=10
   a = 10
a = 8
  BAND(++a)=0
   a = 10
a = 9
  BAND(++a)=0
   a = 11
a = 10
  BAND(++a)=0
   a = 12
```

Quando **a** vale quattro, solo la prima parte della condizione basta (perché già non rispettata), quindi l'espressione viene valutata una volta e l'effetto collaterale della chiamata a macro è che **a** diventa cinque, il che è ciò che ci si aspetta da una normale chiamata a funzione nella stessa situazione. Tuttavia, quando il numero cade all'interno del range, verranno testate entrambe le condizioni dell'if il che comporta due incrementi. Il risultato viene prodotto valutando di nuovo l'argomento, per cui si passa a un terzo incremento. Una volta che il numero esce fuori dal range, entrambe le condizioni sono

ancora testate avendo ancora due incrementi. Gli effetti collaterali sono differenti, dipendono dall'argomento.

Questo non è chiaramente il tipo di comportamento che si vuole da una macro che assomigli a una funzione. In questo caso, la soluzione ovvia è renderla una vera funzione, il che naturalmente aggiunge un overhead extra e può ridurre l'efficienza se si chiama spesso la funzione. Sfortunatamente, il problema può non essere sempre così ovvio, si può, ad esempio, avere una libreria che a nostra insaputa, contenga funzioni e macro mischiate insieme, quindi un problema come questo può nascondere bug molto difficili da trovare. Per esempio, la macro **putc( )** nella libreria **cstdio** può valutare il suo secondo argomento due volte. Questo è specificato nello Standard C. Inoltre anche le implementazioni poco attente di **toupper( )** come macro possono valutare l'argomento più di una volta, il che può dare risultati inaspettati con **toupper(*p++)**.[45]

Macro e accessi

Naturalmente, in C è richiesta una codifica accurata e l'uso di macro, potremmo certamente ottenere la stessa cosa in C++ se non ci fosse un problema: una macro non ha il concetto di *scope* richiesto con le funzioni membro. Il preprocessore semplicemente esegue una sostituzione di testo, quindi non si può dire qualcosa tipo

```
class X {
  int i;
public:
#define VAL(X::i) // Errore
```

o qualcosa simile. Inoltre , può non esserci indicazione a quale oggetto ci si stia riferendo. Semplicemente non c'è modo di esprimere lo scope di classe in una macro. Senza alcuna alternativa per le macro, i programmatori sarebbero tentati di rendere alcuni dati **public** per amore dell'efficienza, svelando così l'implementazione sottostante e impedendo cambiamenti in quell'implementazione, così come l'eliminazione della protezione fornita dalla direttiva **private**.

Funzioni Inline

Risolvendo in C++ il problema delle macro con accesso a classi membro di tipo **private**, verranno eliminati *tutti* i problemi associati alle macro. Ciò è stato fatto portando il concetto di macro sotto il controllo del compilatore dove esse appartengono. Il C++ implementa le macro come *funzioni inline*, che è una vera e propria funzione in ogni senso. Qualsiasi comportamento ci si aspetti da una funzione ordinaria, lo si otterrà da una funzione inline. La sola differenza e che una f.i. è sviluppata sul posto, come una macro, così da eliminare l'overhead della chiamata a funzione. Pertanto, si possono (quasi) mai usare le macro, ma solo le funzioni inline.

Ogni funzione definita all'interno del corpo di una classe è automaticamente inline, ma è possibile rendere una funzione inline facendo precedere la definizione di funzione dalla parola riservata **inline**. Comunque, affinché si abbia un qualche effetto si deve includere il corpo della funzione con la dichiarazione, altrimenti il compilatore la tratterà come una funzione ordinaria. Pertanto,

```
inline int plusOne(int x);
```

non ha effetto su tutto ma solo sulla dichiarazione della funzione (che in seguito può essere o meno una funzione inline). L'approccio giusto prevede invece:

```
inline int plusOne(int x) { return ++x; }
```

Da notare che il compilatore verificherà (come sempre) l'esattezza della lista degli argomenti della funzione e del valore di ritorno (eseguendo eventuali conversioni), cose che il preprocessore è incapace di fare. Inoltre provando a scrivere il codice precedente come una macro, si otterrà un effetto indesiderato.

Quasi sempre le funzioni inline saranno definite in un file header. Quando il compilatore vede una tale definizione, mette il tipo di funzione (il nome insieme al valore di ritorno) e il corpo della funzione stessa nella sua tabella dei simboli. Quando la funzione sarà richiamata, il compilatore verificherà l'esattezza della chiamata e l'uso corretto del valore di ritorno, e sostituisce il corpo della funzione con la chiamata stessa, eliminando così l'overhead. Il codice inline occupa spazio, ma se la funzione è piccola, ciò effettivamente prende meno spazio del codice generato per fare una chiamata a una funzione ordinaria (con il pushing degli argomenti nello stack e l'esecuzione della CALL).

Una funzione inline in un file header ha uno stato speciale, si deve includere il file header contenente la funzione e la sua definizione in ogni file dove la funzione è usata, ma non si finisce il discorso con la definizione multipla di errori (comunque, la definizione deve essere identica in tutti i posti dove la funzione inline è inclusa).

Inline all'interno delle classi

Per definire una funzione inline, si deve normalmente precedere la definizione di funzione con la parola riservata **inline**. Comunque ciò non è necessario all'interno delle definizioni di classi. Ogni funzione definita all'interno di una classe è automaticamente inline. Per esempio:

```
//: C09:Inline.cpp
// L'inline all'interno delle classi
#include <iostream>
#include <string>
using namespace std;

class Point {
  int i, j, k;
public:
  Point(): i(0), j(0), k(0) {}
  Point(int ii, int jj, int kk)
    : i(ii), j(jj), k(kk) {}
  void print(const string& msg = "") const {
    if(msg.size() != 0) cout << msg << endl;
    cout << "i = " << i << ", "
         << "j = " << j << ", "
         << "k = " << k << endl;
  }
};

int main() {
  Point p, q(1,2,3);
  p.print("value of p");
  q.print("value of q");
} ///:~
```

Qui, i due costruttori e la funzione **print( )** sono tutte inline per default. Da notare che nel **main( )** il fatto che si stiano usando funzioni inline è del tutto trasparente, come è giusto che sia. Il comportamento logico di una funzione deve essere identico al di là del fatto che essa sia o meno inline (altrimenti il compilatore è rotto). La sola differenza si noterà nelle prestazioni.

Naturalmente, la tentazione è di usare le funzioni inline ovunque all'interno delle dichiarazioni di classe perchè si risparmia lo step extra di creare una definizione di funzione esterna. Ricordarsi che la tecnica inline serve per fornire buone opportunità per l'ottimizzazione del compilatore . Ma rendere inline una funzione grande causerà una duplicazione di codice ovunque la funzione venga chiamata, gonfiando il codice e diminuendo la velocità (il solo modo sicuro per scoprirlo è sperimentare agli effetti di rendere inline un programma sul proprio compilatore).

Access functions (funzioni d'accesso)

Uno degli usi più importanti della tecnica inline all'interno delle classi è l'*access function*. Questa è una piccola funzione che permette di leggere o cambiare parte dello stato di un'oggetto - cioè variabili o una variabile interna. Il motivo per cui la tecnica inline è importante per le access functions può essere visto nel seguente esempio:

```
//: C09:Access.cpp
// Inline  e access functions

class Access {
  int i;
public:
  int read() const { return i; }
  void set(int ii) { i = ii; }
};

int main() {
  Access A;
  A.set(100);
  int x = A.read();
} ///:~
```

Qui, l'utente della classe non ha mai un contatto diretto con lo stato della variabile all'interno della classe ed esse possono essere mantenute **private**, sotto il controllo del progettista della classe. Tutti gli accessi a dati di tipo **private** può essere controllato attraverso la funzione interfaccia. In più, l'accesso è notevolmente efficiente. Si consideri il **read( )**, per esempio. Senza l'inline, il codice generato per la chiamata a **read( )** dovrà tipicamente includere il pushing (inserimento) nello stack e fare una chiamata assembler CALL. Con molte macchine, la dimensione di questo codice potrà essere più grande della dimensione del codice creato dalla tecnica inline, e il tempo d''esecuzione potrà essere certamente più lungo.

Senza le funzioni inline, un progettista che privilegia l'efficienza sarà tentato di dichiarare semplicemente **i** di tipo public, eliminando l'overhead permettendo all'utente di accedere direttamente ad **i**. Da un punto di vista progettuale, ciò è disastroso perchè **i** in tal caso diverrebbe parte dell'interfaccia pubblica, il che significa che chi ha scritto la classe non può più cambiarla. Si rimane bloccati con un **int** chiamato **i**. Questo è un problema perchè ci si potrebbe accorgere prima o poi che può essere più utile rappresentare quell'informazione come **float** piuttosto che con un **int**, ma perchè **int i** è parte di un

interfaccia pubblica, non la si può più cambiare. Magari si potrebbe voler eseguire calcoli supplementari oltre a leggere o settare **i**, ma non si può se questo è **public**. Se, d'altronde, si sono sempre usate funzioni per leggere o cambiare lo stato di un'oggetto, si può modificare la rappresentazione sottostante dell'oggetto (per la gioia del cuore..).

Inoltre, l'uso di funzioni per controllare dati permette di aggiungere codice alla funzione per capire quando il valore del dato viene cambiato, il che può essere molto utile durante il debugging. Se una dato è **public,** chiunque può cambiarlo in qualunque momento.

Accessors e mutators

Qualcuno preferisce dividere il concetto delle access functions in *accessors* (leggere lo stato delle informazioni da un oggetto, *che accede*) e *mutators* (cambiare lo stato di un oggetto, *che muta*). Inoltre l'overloading delle funzioni può essere usato per fornire lo stesso nome di funzione sia per l'accessor che per il mutator; il modo in cui si chiama la funzione determina se si sta leggendo o modificando lo stato dell'informazione. Ad esempio,

```
//: C09:Rectangle.cpp
// Accessors e mutators

class Rectangle {
  int wide, high;
public:
  Rectangle(int w = 0, int h = 0)
    : wide(w), high(h) {}
  int width() const { return wide; } // Legge
  void width(int w) { wide = w; } // Setta
  int height() const { return high; } // Legge
  void height(int h) { high = h; } // Setta
};

int main() {
  Rectangle r(19, 47);
  // Cambia width & height:
  r.height(2 * r.width());
  r.width(2 * r.height());
} ///:~
```

Il costruttore usa la lista di inizializzazione (introdotta nel capitolo 8 e trattata completamente nel capitolo 14) per inizializzare i valori di **wide** e **high** (usando la forma dello pseudo costruttore per i tipi predefiniti).

Non si possono avere nomi di funzioni e usare lo stesso identificatore come dato membro, per cui si sarebbe tentati di distinguere i dati con un trattino di sottolineatura. Comunque, gli identificatori con il trattino di sottolineatura sono riservati e non si possono usare.

Si può scegliere invece di usare -get- e -set- per indicare accessor e mutator:

```
//: C09:Rectangle2.cpp
// Accessors e mutators con "get" e "set"

class Rectangle {
  int width, height;
public:
  Rectangle(int w = 0, int h = 0)
    : width(w), height(h) {}
  int getWidth() const { return width; }
```

```
  void setWidth(int w) { width = w; }
  int getHeight() const { return height; }
  void setHeight(int h) { height = h; }
};

int main() {
  Rectangle r(19, 47);
  // Cambia  width & height:
  r.setHeight(2 * r.getWidth());
  r.setWidth(2 * r.getHeight());
} ///:~
```

Naturalmente, accessors e mutators non devono essere semplici pipeline per una variabile interna. Qualche volta possono eseguire calcoli più complicati. L'esempio seguente usa le funzioni della libreria Standard C per produrre una semplice classe**Time** :

```
//: C09:Cpptime.h
// Una semplice classe time
#ifndef CPPTIME_H
#define CPPTIME_H
#include <ctime>
#include <cstring>

class Time {
  std::time_t t;
  std::tm local;
  char asciiRep[26];
  unsigned char lflag, aflag;
  void updateLocal() {
    if(!lflag) {
      local = *std::localtime(&t);
      lflag++;
    }
  }
  void updateAscii() {
    if(!aflag) {
      updateLocal();
      std::strcpy(asciiRep,std::asctime(&local));
      aflag++;
    }
  }
public:
  Time() { mark(); }
  void mark() {
    lflag = aflag = 0;
    std::time(&t);
  }
  const char* ascii() {
    updateAscii();
    return asciiRep;
  }
  // Differenza in secondi:
  int delta(Time* dt) const {
    return int(std::difftime(t, dt->t));
  }
  int daylightSavings() {
    updateLocal();
    return local.tm_isdst;
  }
  int dayOfYear() { // Dal 1° Gennaio
    updateLocal();
```

```
    return local.tm_yday;
  }
  int dayOfWeek() { // Da Domenica
    updateLocal();
    return local.tm_wday;
  }
  int since1900() { // Anni dal 1900
    updateLocal();
    return local.tm_year;
  }
  int month() { // Da Gennaio
    updateLocal();
    return local.tm_mon;
  }
  int dayOfMonth() {
    updateLocal();
    return local.tm_mday;
  }
  int hour() { // Dalla mezzanotte, orario 24-ore
    updateLocal();
    return local.tm_hour;
  }
  int minute() {
    updateLocal();
    return local.tm_min;
  }
  int second() {
    updateLocal();
    return local.tm_sec;
  }
};
#endif // CPPTIME_H ///:~
```

Le funzioni della libreria Standard di C hanno diverse rappresentazioni per il tempo, e queste fanno tutte parte della classe **Time**. Comunque, non è necessario aggiornarle tutte, per tanto **time_t t** è usata come rappresentazione base, **tm local** e **asciiRep** (rappresentazione dei caratteri ASCII) hanno ciascuno dei flag per indicare se devono essere aggiornate al valore corrente di **time_t**. Le due funzioni **private**, **updateLocal( )** e **updateAscii( )** verificano i flags e di conseguenza eseguono l'aggiornamento.

Il costruttore chiama la funzione **mark( )** (che può essere anche chiamata dall'utente per forzare l'oggetto a rappresentare il tempo corrente) e questo azzera i due flags per indicare che l'ora locale e la rappresentazione ASCII non sono più valide. La funzione **ascii( )** chiama **updateAscii( )**, la quale copia il risultato della funzione della libreria standard **asctime( )** nel buffer locale perchè **asctime( )** usa un'area dati statica che viene sovrascritta ogni volta che si chiama. Il valore di ritorno della funzione **ascii( )** è l'indirizzo di questo buffer locale.

Tutte le funzioni che iniziano con **daylightSavings( )** usano la funzione **updateLocal( )**, la quale provoca come conseguenza per la struttura inline di essere abbastanza pesante. Questo non deve sembrare utile, specialmente considerando che probabilmente non si vorrà chiamare la funzione molte volte. Comunque, questo non deve significare che tutte le funzione devono essere fatte non-inline. Se si vogliono tutte le altre funzioni non-inline, conviene almeno mantenere **updateLocal( )** inline, in tal modo il suo codice sarà duplicato nelle funzioni non-inline, eliminando l'overehead extra.

Ecco un piccolo programma test:

```
//: C09:Cpptime.cpp
// Test di una semplice classe time
#include "Cpptime.h"
#include <iostream>
using namespace std;

int main() {
  Time start;
  for(int i = 1; i < 1000; i++) {
    cout << i << ' ';
    if(i%10 == 0) cout << endl;
  }
  Time end;
  cout << endl;
  cout << "start = " << start.ascii();
  cout << "end = " << end.ascii();
  cout << "delta = " << end.delta(&start);
} ///:~
```

Un oggetto **Time** viene creato, poi vengono eseguite alcune attività mangia-tempo e dopo viene creato un secondo oggetto **Time** per segnare il tempo finale. Questi vengono usati per mostrare il tempo iniziale, finale e trascorso.

Stash e Stack con l'inline

Armati della tecnica inline, si possono adesso convertire le classi **Stash** e **Stack** per una maggiore efficienza:

```
//: C09:Stash4.h
// Inline functions
#ifndef STASH4_H
#define STASH4_H
#include "../require.h"

class Stash {
  int size;      // Dimensione di ogni spazio
  int quantity;  // Numero di spazi per lo storage
  int next;      // Prossimo spazio libero
  // Array di bytes allocati dinamicamente:
  unsigned char* storage;
  void inflate(int increase);
public:
  Stash(int sz) : size(sz), quantity(0),
    next(0), storage(0) {}
  Stash(int sz, int initQuantity) : size(sz),
    quantity(0), next(0), storage(0) {
    inflate(initQuantity);
  }
  Stash::~Stash() {
    if(storage != 0)
      delete []storage;
  }
  int add(void* element);
  void* fetch(int index) const {
    require(0 <= index, "Stash::fetch (-)index");
    if(index >= next)
      return 0; // Per indicare la fine
   // Produce un puntatore all'elemento desiderato:
    return &(storage[index * size]);
  }
```

```
  int count() const { return next; }
};
#endif // STASH4_H ///:~
```

Le funzioni piccole ovviamente lavorano bene con la tecnica inline, ma da notare che le due funzioni grandi sono ancora lasciate come non-inline, usando anche per loro la tecnica inline non si avrebbe un guadagno nelle performance:

```
//: C09:Stash4.cpp {O}
#include "Stash4.h"
#include <iostream>
#include <cassert>
using namespace std;
const int increment = 100;

int Stash::add(void* element) {
  if(next >= quantity) // Abbastanza spazio rimasto?
    inflate(increment);
  // Copia l'elemento nello storage,
  // parte dal prossimo spazio libero:
  int startBytes = next * size;
  unsigned char* e = (unsigned char*)element;
  for(int i = 0; i < size; i++)
    storage[startBytes + i] = e[i];
  next++;
  return(next - 1); // Indice
}

void Stash::inflate(int increase) {
  assert(increase >= 0);
  if(increase == 0) return;
  int newQuantity = quantity + increase;
  int newBytes = newQuantity * size;
  int oldBytes = quantity * size;
  unsigned char* b = new unsigned char[newBytes];
  for(int i = 0; i < oldBytes; i++)
    b[i] = storage[i]; // Copia il vecchio sul nuovo
  delete [](storage); // Rilascia lo storage vecchio
  storage = b; // Punta alla nuova memoria
  quantity = newQuantity; // Aggiusta la dimensione
} ///:~
```

Ancora una volta, il programma verifica che tutto funzioni correttamente:

```
//: C09:Stash4Test.cpp
//{L} Stash4
#include "Stash4.h"
#include "../require.h"
#include <fstream>
#include <iostream>
#include <string>
using namespace std;

int main() {
  Stash intStash(sizeof(int));
  for(int i = 0; i < 100; i++)
    intStash.add(&i);
  for(int j = 0; j < intStash.count(); j++)
    cout << "intStash.fetch(" << j << ") = "
         << *(int*)intStash.fetch(j)
         << endl;
```

```
  const int bufsize = 80;
  Stash stringStash(sizeof(char) * bufsize, 100);
  ifstream in("Stash4Test.cpp");
  assure(in, "Stash4Test.cpp");
  string line;
  while(getline(in, line))
    stringStash.add((char*)line.c_str());
  int k = 0;
  char* cp;
  while((cp = (char*)stringStash.fetch(k++))!=0)
    cout << "stringStash.fetch(" << k << ") = "
         << cp << endl;
} ///:~
```

Questo è lo stesso programma-test usato prima, per cui l'output deve essere sostanzialmente lo stesso.

La classe **Stack** fa perfino miglior uso della tecnica inline:

```
//: C09:Stack4.h
// Con l'inline
#ifndef STACK4_H
#define STACK4_H
#include "../require.h"

class Stack {
  struct Link {
    void* data;
    Link* next;
    Link(void* dat, Link* nxt):
      data(dat), next(nxt) {}
  }* head;
public:
  Stack() : head(0) {}
  ~Stack() {
    require(head == 0, "Stack not empty");
  }
  void push(void* dat) {
    head = new Link(dat, head);
  }
  void* peek() const {
    return head ? head->data : 0;
  }
  void* pop() {
    if(head == 0) return 0;
    void* result = head->data;
    Link* oldHead = head;
    head = head->next;
    delete oldHead;
    return result;
  }
};
#endif // STACK4_H ///:~
```

Da notare che il distruttore **Link**, che era presente ma vuoto nella precedente versione di **Stack** è stato rimosso. In **pop( )**, l'espressione **delete oldHead** semplicemente libera la memoria usata da **Link** (ciò non distrugge il dato puntato da**Link**).

La maggior parte delle funzioni diventa inline piuttosto esattamente e ovviamente, specialmente per **Link**. Perfino **pop( )** sembra lecito, sebbene qualche volta si possono

avere condizioni o variabili locali per le quali non è chiaro che la tecnica inline sia la più utile. Qui la funzione è piccola abbastanza tanto che probabilmente non danneggia alcunché.

Se tutte le funzioni sono rese inline, l'uso della libreria diventa abbastanza semplice perchè non c'e necessità del linking, come si può vedere nell'esempio (da notare che non c'è **Stack4.cpp**):

```
//: C09:Stack4Test.cpp
//{T} Stack4Test.cpp
#include "Stack4.h"
#include "../require.h"
#include <fstream>
#include <iostream>
#include <string>
using namespace std;

int main(int argc, char* argv[]) {
  requireArgs(argc, 1); // L'argomento è il nome del file
  ifstream in(argv[1]);
  assure(in, argv[1]);
  Stack textlines;
  string line;
  // Legge il file e memorizza le linee nello stack:
  while(getline(in, line))
    textlines.push(new string(line));
  // Estrae le linee dallo stack e le stampa:
  string* s;
  while((s = (string*)textlines.pop()) != 0) {
    cout << *s << endl;
    delete s;
  }
} ///:~
```

C'è chi scriverà, in qualche caso, classi con tutte funzioni inline, così che l'intera classe sarà in un file header. Durante lo sviluppo di un programma ciò è probabilmente innocuo, sebbene in qualche caso può rendere più lunga la compilazione. Ma una volta che il programma si stabilizza, si può tornare indietro e scrivere funzioni non-inline dove è possibile.

L'inline e il compilatore

Per capire dove la tecnica inline è efficace, è utile conoscere cosa fa il compilatore quando incontra una funzione inline. Come per ogni funzione, il compilatore, memorizza il *tipo* della funzione (cioè, il prototipo della funzione includendo il nome e i tipi degli argomenti, insieme al valore di ritorno della funzione) nella sua tabella dei simboli. In più, quando il compilatore vede che il tipo della funzione inline e il corpo della funzione sono analizzabili senza errori, anche il codice per la funzione viene tirato dentro la tabella dei simboli. In qualsiasi forma sia memorizzato il codice sorgente, istruzioni assembler compilate o altre rapresentazioni spetta al compilatore deciderlo.

Quando si fa una chiamata ad una funzione inline, il compilatore per prima cosa si assicura che la chiamata può essere fatta in modo corretto. Ovvero tutti i tipi degli argomenti devono o essere giusti nella lista degli argomenti o il compilatore deve essere in grado di fare una conversione di tipo verso i tipi esatti ed inoltre il valore di ritorno deve essere di

tipo corretto (o convertibile) nell'espressione di destinazione. Questo, naturalmente, è esattamente ciò che il compilatore fa per ogni funzione ed è notevolmente diverso da ciò che fa il preprocessore, che non può far verifiche sui tipi o eseguire conversioni.

Se tutte le informazioni sul tipo di funzione si adattano con il contesto della chiamata, il codice inline viene sostituito direttamente alla chiamata di funzione, eliminando l'overhead di chiamata e permettendo ulteriori ottimizzazioni al compilatore. Inoltre, se il codice inline è una funzione membro, l'indirizzo dell'oggetto (**this**) viene inserito al posto giusto, il che, ovviamente, è un'altra azione che il preprocessore non è in grado di fare.

Limitazioni

Ci sono due situazioni nelle quali il compilatore non può eseguire l'inlining. In questi casi, semplicemente, ritorna alla forma ordinaria di una funzione prendendo la definizione inline e memorizzandola proprio come una funzione non-inline. Se deve fare questo in unità di conversioni multiple (le quali normalmente causano un errore di definizione multipla), il linker è in grado di ignorare le definizioni multiple.

Il compilatore non può eseguire l'inlining se la funzione è troppo complicata. Questo dipende dal particolare compilatore, ma su questo punto molti compilatori rinunciano, la tecnica inline quindi non apporterà probabilmente un aumento di efficienza. In generale ogni sorta di loop è considerata troppo complicata da espandere con la tecnica inline e, pensandoci sopra, un ciclo probabilmente comporta molto più tempo all'interno della funzione che non quello richiesto dall'overhead di chiamata. Se la funzione è solo un insieme di semplici istruzioni, il compilatore non avrà probabilmente difficoltà nell'applicare l'inlining, ma se ci sono molte istruzioni, l'overhead della chiamata sarà minore del costo di esecuzione del corpo della funzione. Si ricordi che ogni volta che si chiama una grossa funzione inline, l'intero corpo della funzione viene inserito al posto della chiamata, per cui facilmente si ottiene un "rigonfiamento" del codice senza apprezzabili miglioramenti delle prestazioni (da notare che alcuni esempi di questo libro possono eccedere le dimensioni ragionevoli per la tecnica inline in favore della salvaguardia delle proprietà dello schermo).

Il compilatore inoltre non può eseguire l'inlining se l'indirizzo della funzione è preso implicito o esplicito. Se il compilatore deve produrre un indirizzo, allora esso allocherà memoria per il codice della funzione e userà l'indirizzo derivato. Comunque, quando un indirizzo non è richiesto, il compilatore probabilmente applicherà la tecnica inline al codice.

E' importante capire che la tecnica inline è solo una proposta al compilatore; quest'ultimo non è forzato a fare niente inline. Un buon compilatore applicherà la tecnica inline con funzioni piccole e semplici mentre intelligentemente ignorerà tale tecnica per quelle troppo complicate. Ciò darà i risultati sperati: l'esatta semantica della chiamata a funzione con l'efficienza di una macro.

Riferimenti in avanti

Se si immagina cosa faccia il compilatore per implementare la tecnica inline, ci si può confondere nel pensare che ci siano più limitazioni di quante ne esistano effettivamente. In particolare, se una funzione inline fa un riferimento in avanti ad un'altra funzione che non

è stata ancora dichiarata nella classe (al di là del fatto che sia inline o meno), può sembrare che il compilatore non sia in grado di maneggiarlo:

```
//: C09:EvaluationOrder.cpp
// Ordine di valutazione dell'inline

class Forward {
  int i;
public:
  Forward() : i(0) {}
  // Chiamata a funzioni non dichiarate:
  int f() const { return g() + 1; }
  int g() const { return i; }
};

int main() {
  Forward frwd;
  frwd.f();
} ///:~
```

In **f( )**, viene fatta una chiamata a **g( )**, sebbene **g( )** non è stata ancora dichiarata. Ciò funziona perchè il linguaggio stabilisce che le funzioni non-inline in una classe saranno valutate fino alla parentesi graffa di chiusura della dichiarazione di classe.

Naturalmente, se **g( )** a sua volta chiama **f( )**, si avrebbero una serie di chiamate ricorsive che sarebbero troppo complicate per il compilatore da gestire con l'inline (inoltre, si dovrebbero eseguire alcune prove in **f( )** or **g( )** per forzare una di esse a "raggiungere il livello più basso", altrimenti la ricorsione sarebbe infinita).

Attività nascoste nei costruttori e distruttori

Costruttori e distruttori sono due situazioni in cui si può pensare che l'inline è più efficiente di quanto non lo sia realmente. Costruttori e distruttori possono avere attività nascoste, perchè la classe può contenere suboggetti dai quali costruttori e distruttori devono essere chiamati. Questi suboggetti possono essere normali oggetti o possono esistere a causa dell'ereditarietà (trattata nel Capitolo 14). Come esempio di classe con oggetti membro:

```
//: C09:Hidden.cpp
// Attività nascoste nell'inline
#include <iostream>
using namespace std;

class Member {
  int i, j, k;
public:
  Member(int x = 0) : i(x), j(x), k(x) {}
  ~Member() { cout << "~Member" << endl; }
};

class WithMembers {
  Member q, r, s; // costruttori
  int i;
public:
  WithMembers(int ii) : i(ii) {} // Insignificante?
  ~WithMembers() {
    cout << "~WithMembers" << endl;
  }
```

```
};

int main() {
  WithMembers wm(1);
} ///:~
```

Il costruttore per **Member** è abbastanza semplice da trattare con la tecnica inline poichè non c'e niente di speciale da fare - nessuna eredità o oggetti membro che causano attività nascoste. Ma nella classe **WithMembers** c'e molto di più di cui occuparsi di quanto salta all'occhio. I costruttori e distruttori per gli oggetti **q**, **r**, e **s** sono stati chiamati automaticamente, e *quei* costruttori e distruttori sono pure inline, per cui la differenza da una normale funzione è significativa. Questo non deve necessariamente significare che si dovrebbero sempre fare definizioni di costruttori e distruttori non-inline; ci sono casi in cui ciò ha senso. Inoltre, quando si sta facendo una "bozza" iniziale di un programma per scrivere velocemente il codice, è spesso molto conveniente usare la tecnica inline. Ma se si ricerca l'efficienza, è una situazione da osservare .

Ridurre la confusione

In un libro come questo, la semplicità e concisione di mettere le definizioni inline all'interno delle classi è molto utile perché ben si adattano su una pagina o su un video (come in un seminario). Comunque, Dan Saks[46] ha posto in rilievo che in un progetto reale ciò ha l'effetto di ingombrare inutilmente l'interfaccia della classe e quindi rendere la classe molto pesante da usare. Egli ricorre a funzioni membro definite dentro le classi, usando il latino *in situ* (sul posto), e sostiene che tutte le definizioni dovrebbero essere piazzate fuori dalla classe per mantenere l'interfaccia pulita. L'ottimizzazione che egli intende dimostrare è una questione separata. Se si vuole ottimizzare il codice, si usi la parola riservata **inline.**Usando questo approccio, l'esempio di prima, **Rectangle.cpp** diventa:

```
//: C09:Noinsitu.cpp
// Rimuovere le funzioni in situ

class Rectangle {
  int width, height;
public:
  Rectangle(int w = 0, int h = 0);
  int getWidth() const;
  void setWidth(int w);
  int getHeight() const;
  void setHeight(int h);
};

inline Rectangle::Rectangle(int w, int h)
  : width(w), height(h) {}

inline int Rectangle::getWidth() const {
  return width;
}

inline void Rectangle::setWidth(int w) {
  width = w;
}

inline int Rectangle::getHeight() const {
  return height;
}
```

```
inline void Rectangle::setHeight(int h) {
  height = h;
}

int main() {
  Rectangle r(19, 47);
  // Transpone Width e Height:
  int iHeight = r.getHeight();
  r.setHeight(r.getWidth());
  r.setWidth(iHeight);
} ///:~
```

Ora se si vuole paragonare l'effetto delle funzioni inline con quelle non-inline, semplicemente si può rimuovere la keyword **inline** (le funzioni inline dovrebbero trovarsi normalmente nei file header, per quanto possibile, mentre le funzioni non-inline devono trovarsi nella propria unità di conversione). Se si vogliono mettere le funzioni nella documentazione, lo si può fare con una semplice operazione di taglia-e-incolla. Le funzioni *in situ* necessitano di maggior lavoro e potenzialmente possono presentare più errori. Un'altra controversia per questo approccio è che si può sempre produrre uno stile di formattazione coerente per le definizioni di funzioni, qualcosa che non sempre è necessario con le funzioni *in situ*.

Ulteriori caratteristiche del preprocessore

In precedenza, ho detto che *quasi* sempre si vogliono usare le funzioni **inline** invece delle macro. Le eccezioni sorgono quando si ha bisogno di usare tre caratteristiche speciali del preprocessore C (che poi è anche il preprocessore C++): stringizing (convertire in stringhe *ndt*), concatenazione di stringhe e token pasting (incollatura di identificatori). Stringizing, introdotta in precedenza nel libro, viene eseguita con la direttiva **#** e permette di prendere un identificatore e convertirlo in una stringa. La concatenazione di stringhe si ha quando due stringhe adiacenti non hanno caratteri di punteggiatura tra di loro, nel qual caso esse vengono unite. Queste due caratteristiche sono particolarmente utili nello scrivere codice di debug. Cioè,

```
#define DEBUG(x) cout << #x " = " << x << endl
```

Questo stampa il valore di ogni variabile. Si può anche ottenere una traccia che stampi le istruzioni eseguite:

```
#define TRACE(s) cerr << #s << endl; s
```

La direttiva **#s** converte in stringa le istruzioni per l'output e la seconda **s** reitera l'istruzione così esso viene eseguito. Naturalmente, questo tipo di cosa può causare problemi, specialmente con un ciclo **for** di una sola linea:

```
for(int i = 0; i < 100; i++)
 TRACE(f(i));
```

Poichè ci sono esattamente due istruzioni nella macro **TRACE( )**, il ciclo **for** di una linea, esegue solo la prima. La soluzione è di sostituire il punto e virgola con una virgola nella macro.

Token pasting

Token pasting, implementato con la direttiva **##**, è molto utile quando si sta scrivendo il codice. Essa permette di prendere due identificatori e incollarli insieme per creare automaticamente un nuovo identificatore. Per esempio,

```
#define FIELD(a) char* a##_string; int a##_size
class Record {
  FIELD(one);
  FIELD(two);
  FIELD(three);
  // ...
};
```

Ogni chiamata alla macro **FIELD( )** crea un identificatore per memorizzare una stringa e un altro per memorizzare la lunghezza di questa. Non solo è di facile lettura, ma può eliminare errori di codice e rendere la manutenzione più facile.

Miglioramenti nell'error checking

Le funzioni **require.h** sono state usate fino a questo punto senza definirle (sebbene **assert( )** è già stata usata per aiutare a trovare gli errori di programmazione dove appropriato). E' il momento di definire questo file header. Le funzioni inline qui sono convenienti perchè permettono ad ogni cosa di essere posizionata in un file header, il che semplifica il processo di utilizzo dei package. Basta includere il file header senza il bisogno di preoccuparsi del linking di file.

Si dovrebbe notare che le eccezioni (presentate in dettaglio nel Volume 2 di questo libro) forniscono un modo molto più efficace di maneggiare molte specie di errori – specialmente quelli che si vogliono riparare– invece di fermare il programma. Le condizioni che tratta **require.h**, comunque, sono quelle che impediscono la continuazione del programma, in modo simile a quando l'utente non fornisce sufficienti argomenti alla riga di comando o quando un file non può essere aperto. Perciò, è accettabile la chiamata alla funzione **exit()** della libreria Standard C.

Il seguente file header si trova nella root directory del libro, per cui facilmente accessibile da tutti i capitoli.

```
//: :require.h
// Test per le condizioni di errore nei programmi
// Per i primi compilatori inserire "using namespace std"
#ifndef REQUIRE_H
#define REQUIRE_H
#include <cstdio>
#include <cstdlib>
#include <fstream>
#include <string>

inline void require(bool requirement,
  const std::string& msg = "Richiesta fallita"){
  using namespace std;
  if (!requirement) {
    fputs(msg.c_str(), stderr);
    fputs("\n", stderr);
    exit(1);
```

```
  }
}

inline void requireArgs(int argc, int args,
  const std::string& msg =
    "Devi usare %d argomenti") {
  using namespace std;
   if (argc != args + 1) {
     fprintf(stderr, msg.c_str(), args);
     fputs("\n", stderr);
     exit(1);
   }
}

inline void requireMinArgs(int argc, int minArgs,
  const std::string& msg =
    "Devi usare almeno  %d argomenti") {
  using namespace std;
  if(argc < minArgs + 1) {
    fprintf(stderr, msg.c_str(), minArgs);
    fputs("\n", stderr);
    exit(1);
  }
}

inline void assure(std::ifstream& in,
  const std::string& filename = "") {
  using namespace std;
  if(!in) {
    fprintf(stderr, "Impossibile aprire il file %s\n",
      filename.c_str());
    exit(1);
  }
}

inline void assure(std::ofstream& out,
  const std::string& filename = "") {
  using namespace std;
  if(!out) {
    fprintf(stderr, "Impossibile aprire il file %s\n",
      filename.c_str());
    exit(1);
  }
}
#endif // REQUIRE_H ///:~
```

I valori di default forniscono messaggi ragionevoli che possono essere cambiati se necessario.

Si noterà che invece di usare argomenti **char***, vengono usati **const string&**. Ciò permette per queste funzioni argomenti sia **char*** che **string**, e quindi in generale molto più utile (si può voler seguire questo modello nel proprio codice).

Nelle definizioni di **requireArgs( )** e **requireMinArgs( )**, viene aggiunto 1 al numero di argomenti necessari sulla linea di comando perchè **argc** già include il nome del programma che viene eseguito come argomento 0, e quindi già ha un valore che è uno in più del numero degli argomenti presenti sulla linea di comando.

Si noti l'uso delle dichiarazioni locali "**using namespace std**" dentro ogni funzione. Ciò perchè alcuni compilatori nel momento della scrittura di questo libro non includevano erroneamente le funzioni standard della libreria C in **namespace std**, per cui un uso esplicito potrebbe causare un errore a compile-time. La dichiarazione locale permette a **require.h** di lavorare sia con librerie corrette che con quelle incomplete evitando la creazione di namespace **std** per chiunque includa questo file header.

Ecco un semplice programma per testare **require.h**:

```
//: C09:ErrTest.cpp
//{T} ErrTest.cpp
// Test di  require.h
#include "../require.h"
#include <fstream>
using namespace std;

int main(int argc, char* argv[]) {
  int i = 1;
  require(i, "value must be nonzero");
  requireArgs(argc, 1);
  requireMinArgs(argc, 1);
  ifstream in(argv[1]);
  assure(in, argv[1]); // Use il nome del file
  ifstream nofile("nofile.xxx");
  // Fallimento:
//!  assure(nofile); // L'argomento di default
  ofstream out("tmp.txt");
  assure(out);
} ///:~
```

Si potrebbe essere tentati di fare un passo ulteriore per aprire file e aggiungere macro a **require.h**:

```
#define IFOPEN(VAR, NAME) \
  ifstream VAR(NAME); \
  assure(VAR, NAME);
```

Che potrebbero essere usate così:

```
IFOPEN(in, argv[1])
```

Dapprima, questo potrebbe sembrare interessante poichè sembra ci sia da digitare di meno. Non è terribilmente insicuro, ma è una strada che è meglio evitare. Si noti come, ancora una volta, una macro appare come una funzione ma si comporta diversamente; essa effettivamente crea un oggetto (**in**) il cui scope dura al di là della la macro. Si può capire questo, ma per i nuovi programmatori e i manutentori di codice è solo una cosa in più da decifrare. Il C++ è già complicato abbastanza di per sè, per cui è bene evitare di usare le macro ogni qualvolta si può.

Sommario

E' di importanza cruciale essere abili a nascondere l'implementazione sottostante di una classe perchè si può voler cambiare questa in seguito. Si faranno questi cambiamenti per aumentare l'efficienza, o perchè si arriva a una migliore comprensione del problema, o perchè si rendono disponibili nuove classi che si vogliono usare nell'implementazione.

Qualsiasi cosa che metta in pericolo la privacy dell'implementazione riduce la flessibilità del linguaggio. Per questo, la funzione inline è molto importante perchè essa virtualmente elimina il bisogno delle macro e i loro problemi correlati. Con la tecnica inline, le funzioni possono essere efficienti come macro.

Le funzioni inline possono essere usate nelle definizioni di classi, naturalmente. Il programmatore è tentato di fare così perchè è più facile, e così avviene. Comunque, non è che un punto di discussione, infatti più tardi, cercando un'ottimizzazione delle dimensioni, si possono sempre cambiare le funzioni in funzioni non-inline senza nessuno effetto sulla loro funzionalità. La linea guida dello sviluppo del codice dovrebbe essere “Prima rendilo funzionante, poi ottimizzalo.

---

[45]Andrew Koenig entra in maggiori dettagli nel suo libro *C Traps & Pitfalls* (Addison-Wesley, 1989).

[46] Co-autore con Tom Plum di *C++ Programming Guidelines*, Plum Hall, 1991.

10: Controllo dei Nomi

Inventare nomi è un'attività fondamentale nella programmazione, e quando un progetto diventa molto grande il numero di nomi puo' diventare opprimente.

Il C++ fornisce una gran quantità di controlli sulla creazione e visibiltà dei nomi, sulla loro posizione in memoria e sul linkage. La parola chiave **static** è stata sovraccaricata (di significato) in C prima ancora che la gente conoscesse il significato del termine "overload" e il C++ ha aggiunto anche un altro significato. Il concetto base per tutti gli usi che si fanno della parola **static** sembra essere "qualcosa che ricorda la sua posizione" (come l'elettricità statica), che si tratti della posizione fisica in memoria o della visibilità all'interno di un file. In questo capitolo impareremo come la parola chiave **static** controlla l'allocazione e la visibilità, vedremo un modo migliore per controllare l'accesso ai nomi, attraverso il concetto di *namespace* (spazio dei nomi) proprio del C++. Scopriremo anche come usare le funzioni scritte e compilate in C.

Elementi statici dal C

Sia in C che in C++ la parola chiave **static** assume due significati di base, che sfortunatamente spesso si intrecciano:

1. Allocato una volta per tutte ad un indirizzo fisso; cioè l'oggetto viene creato in una speciale *area di dati statica* piuttosto che nello stack ogni volta che viene chiamata una funzione. Questo è il concetto di *allocazione statica*.
2. Locale ad una particolare unità di compilazione (e locale allo scope di una classe in C++, come vedremo). Qui la parola **static** controlla la *visibilità* del nome, cosicchè il nome non puo' essere visto al di fuori dell'unità di compilazione o della classe. Questo descrive anche il concetto di *linkage*, che determina quali nomi sono visibili al linker.

Questa sezione spiega i due precedenti significati della parola **static** così come sono stati ereditati dal C.

variabili statiche all'interno di funzioni

Quando creiamo una variabile locale all'interno di una funzione, il compilatore le alloca memoria sullo stack ogni volta che la funzione viene chiamata, spostando opportunamente in avanti lo stack pointer. Se c'è una sequenza di inizializzazione per la variabile, questa viene eseguita ad ogni chiamata della funzione. A volte, tuttavia, si vuole mantenere il valore di una variabile tra una chiamata e l'altra di una funzione. Questo lo si puo' ottenere utilizzando una variabile globale, ma questa sarebbe visibile a tutti e non solo alla funzione in esame. Il C e il C++ permettono di creare oggetti **static** all'interno di funzioni; l'allocazione di memoria per questi oggetti non avviene sullo stack, ma all'interno dell'area dati statica del programma. L'oggetto viene inizializzato una sola volta, la prima volta che la funzione viene chiamata, e poi mantiene il suo valore tra una chiamata e l'altra della funzione. Per esempio, la seguente funzione restituisce il carattere successivo nell'array ogni volta che la funzione viene chiamata:

```
//: C10:VariabiliStaticheInFunzioni.cpp
#include "../require.h"
#include <iostream>
using namespace std;

char unChar(const char* charArray = 0) {
  static const char* s;
  if(charArray) {
    s = charArray;
    return *s;
  }
  else
    require(s, "s non-inizializzata");
  if(*s == '\0')
    return 0;
  return *s++;
}

char* a = "abcdefghijklmnopqrstuvwxyz";

int main() {
  // unChar(); // require() fallisce
  unChar(a); // Inizializza s con il valore di a
  char c;
  while((c = unChar()) != 0)
    cout << c << endl;
} ///:~
```

La variabile **static char* s** mantiene il suo valore tra le chiamate a **unChar( )** perchè la sua locazione di memoria non è parte dello spazio di stack della funzione, ma è dentro l'area statica del programma. Quando viene chiamata la funzione **unChar( )** con un argomemto di tipo **char ***, all'argomento viene assegnato **s** e viene restituito il primo carattere dell'array. Tutte le chiamate successive alla funzione **unChar()**, fatte senza argomento, producono il valore di default zero per **charArray**, il che indica alla funzione che si stanno ancora estraendo caratteri dal valore precedentemente inizializzato di **s**. La funzione continua a produrre caratteri fino a quando non incontra il terminatore null dell'array di caratteri, momento in cui smette di incrementare il puntatore in modo da non sfondare il limite dell'array. Ma cosa succede se chiamiamo la funzione **unChar()** senza argomenti e senza preventivamente inizializzare il valore di **s**? Nella definizione di **s** potremmo fornire un valore iniziale,

```
static char* s = 0;
```

ma anche se non effettuiamo l'inizializzazione di una variabile statica di un tipo predefinito, il compilatore garantisce che questa variabile sarà inizializzata a zero (convertita ad un tipo appropriato) allo start-up del programma. Cosicchè, la prima volta che viene chiamata la funzione **unChar()**, **s** è zero. In questo caso, l'istruzione condizionale **if(!s)** lo proverebbe. L'inizializzazione di **s** fatta sopra è molto semplice, ma in generale l'inizializzazione di oggetti statici (come per tutti gli altri) puo' essere fatta con espressioni che coinvolgono costanti, variabili dichiarate precedentemente e funzioni. Bisogna essere consapevoli che la funzione di sopra è molto vulnerabile ai problemi del multithreading; ogni qualvolta si introducono funzioni con variabili statiche bisogna stare attenti al multithreading.

oggetti di classi statiche all'interno di funzioni

Le regole sono le stesse di quelle per oggetti statici di tipi definiti dall'utente, compreso il fatto che l'oggetto richiede una qualche inizializzazione. Tuttavia l'assegnamento del valore zero è significativo solo per tipi predefiniti; i tipi definiti dall'utente necessitano di un costruttore per essere inizializzati. Percio', se non si specificano gli argomenti per il costruttore in fase di definizione di un oggetto statico, viene usato un costruttore di default, che ogni classe deve fornire. Per esempio,

```
//: C10:OggettiStaticiInFunzioni.cpp
#include <iostream>
using namespace std;

class X {
  int i;
public:
  X(int ii = 0) : i(ii) {} // Default
  ~X() { cout << "X::~X()" << endl; }
};

void f() {
  static X x1(47);
  static X x2; // Richiesto il costruttore di Default
}

int main() {
  f();
} ///:~
```

Gli oggetti statici di tipo **X** all'interno di **f()** possono essere inizializzati sia con una lista di argomenti passati al costruttore, sia con il costruttore di default. Questa costruzione avviene la prima volta che il controllo passa attraverso la definizione, e solo la prima volta.

Distruttori di oggetti statici

I distruttori per oggetti statici (cioè tutti quelli con locazione statica, non solo quelli statici locali come nell'esempio sopra) vengono chiamati quando si esce dal **main()** o quando viene chiamata esplicitamente la funzione **exit()** della libreria standard del C. In molte implementazioni la funzione **exit()** viene chiamata all'uscita del **main()**. Questo significa che è dannoso chiamare la funzione **exit()** all'interno di un distruttore, in quanto si potrebbe cadere in un loop infinito. I distruttori di oggetti statici non vengono chiamati se si esce dal programma usando la funzione di libreria **abort()**. Si possono specificare le azioni che il programma deve svolgere all'uscita del **main()** (o alla chiamata di **exit()**) usando la funzione della libreria standard del C **atexit()**. In questo caso le funzioni registrate con **atexit()** possono essere chiamate prima dei distruttori per tutti gli oggetti costruiti, prima di uscire dal **main()** (o della chiamata ad **exit()**).

La distruzione di oggetti statici, come per quelli ordinari, avviene nell'ordine inverso rispetto all'inizializzazione. Tuttavia, solo gli oggetti costruiti vengono distrutti. Fortunatamente i tools di sviluppo del C++ tengono traccia dell'ordine di inizializzazione e degli oggetti che sono stati costruiti (per i quali, cioè, è stato chiamato un costruttore). Gli oggetti globali vengono sempre costruiti prima di entrare nel **main**() e distrutti all'uscita del **main**(), ma se una funzione che contiene un oggetto statico locale non viene mai chiamata, il costruttore di tale oggetto non viene mai eseguito e quindi neanche il distruttore sarà eseguito. Per esempio,

```
//: C10:DistruttoriStatici.cpp
```

```
// Distruttori di oggetti statici
#include <fstream>
using namespace std;
ofstream out("statdest.out"); // file di tracciamento

class Obj {
  char c; // Identificatore
public:
  Obj(char cc) : c(cc) {
    out << "Obj::Obj() per " << c << endl;
  }
  ~Obj() {
    out << "Obj::~Obj() per " << c << endl;
  }
};

Obj a('a'); // Globale (memorizzazione statica)
// Costruttore & distruttore sempre chiamati

void f() {
  static Obj b('b');
}

void g() {
  static Obj c('c');
}

int main() {
  out << "dentro il main()" << endl;
  f(); // Chiama il costruttore statico per b
  // g() non chiamata
  out << "all'uscita del main()" << endl;
} ///:~
```

In **Obj** la variabile **char c** funge da identificatore, in modo tale che il costruttore e il distruttore possono stampare informazioni riguardo all'oggetto su cui stanno agendo. La variabile **Obj a** è un oggetto globale, per cui il suo costruttore viene sempre chiamato prima di entrare nel **main()**, ma i costruttori di **static Obj b** dentro **f()** e di **static Obj c** dentro **g()** vengono eseguiti solo se queste funzioni vengono chiamate. Per dimostrare quali costruttori e distruttori vengono eseguiti, è stata chiamata solo la funzione **f()**. L'output del programma è

```
Obj::Obj() per a
dentro il main()
Obj::Obj() per b
all'uscita del main()
Obj::~Obj() per b
Obj::~Obj() per a
```

Il costruttore per **a** viene chiamato prima di entrare nel **main()**, mentre il costruttore per **b** viene chiamato solo perchè viene chiamata la funzione **f()**. Quando si esce dal **main()**, i distruttori degli oggetti per i quali è stato eseguito il costruttore vengono eseguiti nell'ordine inverso rispetto alla costruzione. Questo significa che se viene chiamata anche **g()**, l'ordine di distruzione di **b** e **c** dipende dall'ordine di chiamata di **f()** e **g()**. Notare che anche l'oggetto **out**, di tipo **ofstream**, è statico, in quanto definito all'esterno di qualsiasi funzione, e vive nell'area di memoria statica. E' importante che la sua definizione (al contrario di una dichiarazione **extern**) appaia all'inizio del file, prima che **out** possa essere usata. Altrimenti si potrebbe usare un oggetto prima che questo venga opportunamente inizializzato. In C++ il costruttore di un oggetto statico globale viene chiamato prima di entrare nel **main()**, cosicchè abbiamo un modo molto semplice e portabile per eseguire del codice prima di entrare nel **main()** e di eseguire codice con il

distruttore all'uscita dal **main()**. In C questo richiede di mettere le mani al codice di start-up in linguaggio assembler, fornito insieme al compilatore.

Il controllo del linkage

Ordinariamente, qualsiasi nome con scope a livello di file (cioè non annidato all'interno di una classe o di una funzione) è visibile attraverso tutte le unità di compilazione in un programma. Questo spesso è chiamato *linkage esterno*, in quanto durante il link il nome è visibile al linker dovunque, esternamente a tale unità di compilazione. Le variabili globali e le funzioni ordinarie hanno questo tipo di linkage. Ci sono casi in cui si vuole limitare la visibilità di un nome. Si potrebbe volere che una variabile sia visibile a livello di file, in modo che tutte le funzioni definite in quel file possano accedervi, ma si vuole impedire l'accesso alle funzioni al di fuori del file o evitare una collisione di nomi con identificatori definiti in altri file.

Un nome di oggetto o di funzione con visibilità a livello di file esplicitamente dichiarata con **static** è locale alla sua unità di compilazione (nell'accezione di questo libro vuol dire il file **.cpp** dove avviene la dichiarazione). Questo nome ha un *linkage interno*. Questo significa che si puo' usare lo stesso nome in altre unità di compilazione senza creare conflitto. Un vantaggio del linkage interno è il fatto che i nomi possono essere piazzati in un file di intestazione senza temere conflitti durante il link. I nomi che vengono comunemente messi nei file di intestazione, come le definizioni **const** e le funzioni **inline**, hanno per default un linkage interno (tuttavia, **const** ha un linkage interno solo in C++, mentre per il C il default è esterno). Notare, inoltre, che il concetto di linkage si applica solo a quegli elementi per i quali l'indirizzo viene calcolato a tempo di link o di load; ad esempio alle dichiarazioni di classi e alle variabili locali non si applica il concetto di linkage.

Confusione

Qui c'è un esempio di come le due accezioni del termine >**static** possano intrecciarsi l'un l'altra. Tutti gli oggetti globali hanno implicitamente una classe di memorizzazione statica, così, se scriviamo (a livello di file),

```
int a = 0;
```

l'allocazione di **a** avviene nell'area dati statica del programma, e l'inizializzazione di **a** avviene una sola volta, prima di entrare nel **main()**. In più, la visibilità di **a** è globale rispetto a tutte le unità di compilazione. In termini di visibilità l'opposto di **static** (visibile solo in questa unità di compilazione) è **extern**, che asserisce esplicitamente che la visibiltà è trasversale a tutte le unità di compilazione. Percio' la definizione precedente è equivalente alla seguente:

```
extern int a = 0;
```

Ma se scriviamo,

```
static int a = 0;
```

tutto quello che abbiamo fatto è di alterare la visibilità di **a**, così da conferirle un linkage interno. La classe di memorizzazione non è cambiata, in quanto gli oggetti risiedono nell'area dati statica sia che la visibilità sia **static** che **extern**. Quando passiamo alle variabili locali, la parola **static** non agisce più sulla visibilità, che è già limitata, ma altera la classe di memorizzazione. Se dichiariamo come **extern** una variabile che altrimenti sarebbe locale, significa che questa è già stata definita da qualche parte (quindi la variabile è di fatto globale alla funzione). Per esempio:

```
//: C10:LocaleExtern.cpp
//{L} LocaleExtern2
```

```
#include <iostream>

int main() {
  extern int i;
  std::cout << i;
} ///:~

//: C10:LocaleExtern2.cpp {O}
int i = 5;
///:~
```

Con i nomi di funzione (non funzioni membro di classi) **static** ed **extern** possono alterare solo la visibilità, così la dichiarazione

```
extern void f();
```

è equivalente a

```
void f();
```

e la dichiarazione

```
static void f();
```

significa che **f()** è visibile solo all'interno di questa unità di compilazione - detta a volte *file statico*.

Altri specificatori di classi di memorizzazione

Gli specificatori **static** ed **extern** si usano abitualmente. Ma ci sono altri due specificatori di classi di memorizzazione che si vedono meno frequentemente. Lo specificatore **auto** non è quasi mai usato, in quanto esso istruisce il compilatore che una variabile è locale. **auto**, infatti, è una forma abbreviata che sta per "automatic" e si riferisce al modo in cui il compilatore automaticamente alloca memoria per una variabile. Il compilatore è sempre in grado di determinare questo fatto dal contesto, per cui l'uso di **auto** è ridontante. L'altro specificatore è **register**. Una variabile **register** è di tipo locale (**auto**). Con **register** si avvisa il compilatore che questa variabile potrebbe essere usata pesantemente e che quindi se puo' la deve memorizzare in un registro. In questo senso è un aiuto all'ottimizzazione. Compilatori diversi rispondono in maniera differente a questo suggerimento; essi hanno anche la possibilità di ignorarlo. Se ad esempio si prende l'indirizzo di una variabile, lo specificatore **register** è del tutto inutile e quindi quasi certamente viene ignorato. Si dovrebbe evitare l'uso eccessivo di **register**, in quanto è probabile che il compilatore effettui l'ottimizzazione meglio di noi.

Spazio dei nomi (namespaces)

Anche se i nomi possono essere annidati all'interno di classi, i nomi di funzioni globali, di variabili globali e di classi sono comunque in un unico spazio di nomi. La parola chiave **static** ci permette un certo controllo su questo, permettendoci di conferire alle variabili e alle funzioni un linkage interno (cioè rendendoli file statici). Ma in un progetto vasto, la perdita di controllo sullo spazio dei nomi globale puo' causare problemi. Per risolvere questo problema per le classi, spesso i fornitori creano lunghi nomi complicati che difficilmente possono confliggere, ma ci costringono poi a digitare questi nomi (Si puo' usare **typedef** per semplificarli). Non è una soluzione elegante, supportata dal linguaggio. Possiamo suddividere lo spazio dei nomi globale in tanti pezzi più maneggevoli, usando una caratteristica propria del C++, ottenuta con la parola chiave *namespace*. La parola chiave **namespace**, come **class**, **struct**, **enum** ed **union**, pone i nomi dei suoi membri in uno spazio distinto.

Creazione di uno spazio dei nomi

La creazione di uno spazio dei nomi è simile a quella di una **class**:

```
//: C10:MiaLib.cpp
namespace MiaLib {
  // Dichiarazioni
}
int main() {} ///:~
```

Questa definizione crea un nuovo spazio dei nomi che contiene le dichiarazioni racchiuse tra parentesi. Ma ci sono differenze significative rispetto a **class**, **struct**, **union** ed **enum:**

- Una definizione namespace puo' apparire solo a livello globale o nidificata in un altro namespace.
- Non è necessario il punto e virgola dopo le parentesi che racchiudono la definizione.
- La definizione di un namespace puo' continuare attraverso file di intestazione multipli, usando una sintassi che per le classi provocherebbe delle ridefinizioni:

```
//: C10:Header1.h
#ifndef HEADER1_H
#define HEADER1_H
namespace MiaLib {
  extern int x;
  void f();
  // ...
}
#endif // HEADER1_H ///:~
//: C10:Header2.h
#ifndef HEADER2_H
#define HEADER2_H
#include "Header1.h"
// Aggiungere più nomi a MiaLib
namespace MiaLib { // NON è una ridefinizione!
  extern int y;
  void g();
  // ...
}
#endif // HEADER2_H ///:~
//: C10:Continuazione.cpp
#include "Header2.h"
int main() {} ///:~
```

- Un nome di namespace puo' essere sostituito con uno *pseudonimo*, così non siamo costretti a digitare un ingombrante nome creato da un fornitore di librerie.

```
//: C10:BobsSuperDuperLibreria.cpp
namespace BobsSuperDuperLibreria {
  class Widget { /* ... */ };
  class Poppit { /* ... */ };
  // ...
}
// Troppo lungo da digitare! Usiamo un alias (pseudonimo):
namespace Bob = BobsSuperDuperLibreria;
int main() {} ///:~
```

- Non si possono creare istanze di un namespace come per una classe.

Namespace senza nome

Ogni unità di compilazione contiene un *namespace* senza nome che si puo' aggiungere utilizzando la parola chiave "**namespace**" senza identificatore:

```
//: C10:NamespaceSenzaNome.cpp
namespace {
  class Braccio { /* ... */ };
  class Gamba   { /* ... */ };
  class Testa   { /* ... */ };
  class Robot   {
    Braccio braccio[4];
    Gamba   gamba[16];
    Testa   testa[3];
    // ...
  } xanthan;
  int i, j, k;
}
int main() {} ///:~
```

I nomi in questo spazio sono automaticamente disponibili in questa unità di compilazione senza qualificazione. E' garantito che uno spazio senza nome è unico per ogni unità di compilazione. Ponendo nomi locali in namespace senza nome si evita la necessità di usare la parola **static** per conferirgli un linkage interno. Il C++ depreca l'uso dei file statici a favore dei namespace senza nome.

Friends

Si puo' *iniettare* una dichiarazione **friend** in un namespace inserendola dentro una classe:

```
//: C10:IniezioneFriend.cpp
namespace Me {
  class Noi {
    //...
    friend void tu();
  };
}
int main() {} ///:~
```

In questo modo la funzione **tu()** è un membro del namespace **Me**. Se si introduce un **friend** all'interno di una classe in un *namespace* globale, il **friend** viene inserito globalmente.

Usare uno spazio dei nomi

Ci si puo' riferire ad un nome all'interno di un namespace in tre modi: specificando il nome usando l'operatore di risoluzione di scope, con una direttiva **using** per introdurre tutti i nomi nel namespace, oppure una dichiarazione **using** per introdurre nomi uno alla volta.

Risoluzione di scope

Ogni nome in un namespace puo' essere esplicitamente specificato usando l'operatore di risoluzione di scope allo stesso modo in cui ci si puo' riferire ad un nome all'interno di una classe:

```
//: C10:RisoluzioneDiScope.cpp
namespace X {
  class Y {
    static int i;
  public:
    void f();
```

```
  };
  class Z;
  void funz();
}
int X::Y::i = 9;
class X::Z {
  int u, v, w;
public:
  Z(int i);
  int g();
};
X::Z::Z(int i) { u = v = w = i; }
int X::Z::g() { return u = v = w = 0; }
void X::funz() {
  X::Z a(1);
  a.g();
}
int main(){} ///:~
```

Notare che la definizione **X::Y::i** potrebbe essere tranquillamente riferita a un dato membro della classe **Y** annidata nella classe **X** invece che nel namespace **X**. Quindi, i namespace si presentano molto simili alle classi.

La direttiva using

Siccome digitare l'intero qualificatore per un nome in un namespace potrebbe presto diventare tedioso, la parola chiave **using** ci permette di importare un intero namespace in un colpo solo. Quando questa viene usata insieme alla parola chiave **namespace** si ottiene quella che si chiama *direttiva using* . La direttiva **using** fa si che i nomi appaiano come se appartenessero al namespace più vicino che la racchiude, cosicchè possiamo usare convenientemente nomi senza qualificatori. Consideriamo un semplice namespace:

```
//: C10:NamespaceInt.h
#ifndef NAMESPACEINT_H
#define NAMESPACEINT_H
namespace Int {
  enum segno { positivo, negativo };
  class Intero {
    int i;
    segno s;
  public:
    Intero(int ii = 0)
      : i(ii),
        s(i >= 0 ? positivo : negativo)
    {}
    segno getSegno() const { return s; }
    void setSegno(segno sgn) { s = sgn; }
    // ...
  };
}
#endif // NAMESPACEINT_H ///:~
```

Un uso della direttiva **using** è quello di includere tutti i nomi definiti in **Int** dentro un altro namespace, lasciando che questi nomi siano annidati dentro questo secondo namespace:

```
//: C10:NamespaceMat.h
#ifndef NAMESPACEMAT_H
#define NAMESPACEMAT_H
#include "NamespaceInt.h"
namespace Mat {
  using namespace Int;
  Intero a, b;
```

```
  Intero divide(Intero, Intero);
  // ...
}
#endif // NAMESPACEMAT_H ///:~
```

Si possono anche includere tutti i nomi definiti in **Int** dentro una funzione, ma in questo modo i nomi sono annidati nella funzione:

```
//: C10:Aritmetica.cpp
#include "NamespaceInt.h"
void aritmetica() {
  using namespace Int;
  Intero x;
  x.setSegno(positivo);
}
int main(){} ///:~
```

Senza la direttiva **using**, tutti i nomi di un namespace hanno bisogno di essere completamente qualificati. Un aspetto della direttiva **using** potrebbe sembrare controintuitiva all'inizio. La visibilità dei nomi introdotti con una direttiva **using** corrisponde allo scope in cui la direttiva è posizionata. Ma si possono nascondere i nomi provenienti da una direttiva **using**, come se fossero dichiarati a livello globale!

```
//: C10:SovrapposizioneNamespace1.cpp
#include "NamespaceMat.h"
int main() {
  using namespace Mat;
  Intero a; // Nasconde Mat::a;
  a.setSegno(negativo);
  // Adesso è necessaria la risoluzione di scope
  // per selezionare Mat::a :
  Mat::a.setSegno(positivo);
} ///:~
```

Supponiamo di avere un secondo namespace che contiene alcuni dei nomi definiti nel **namespace Mat**:

```
//: C10:SovrapposizioneNamespace2.h
#ifndef SOVRAPPOSIZIONENAMESPACE2_H
#define SOVRAPPOSIZIONENAMESPACE2_H
#include "NamespaceInt.h"
namespace Calcolo {
  using namespace Int;
  Intero divide(Intero, Intero);
  // ...
}
#endif // SOVRAPPOSIZIONENAMESPACE2_H ///:~
```

Siccome con la direttiva **using** viene introdotto anche questo namespace, c'è la possibilità di una collisione. Tuttavia l'ambiguità si presenta soltanto nel punto in cui si *usa* il nome e non a livello di direttiva **using**:

```
//: C10:AmbiguitaDaSovrapposizione.cpp
#include "NamespaceMat.h"
#include "SovrapposizioneNamespace2.h"
void s() {
  using namespace Mat;
  using namespace Calcolo;
  // Tutto ok finchè:
  //! divide(1, 2); // Ambiguità
}
int main() {} ///:~
```

Così, è possibile scrivere direttive **using** per introdurre tanti namaspace con nomi che confliggono senza provocare mai ambiguità.

La dichiarazione using

Si possono inserire uno alla volta dei nomi dentro lo scope corrente con una *dichiarazione using*. A differenza della direttiva **using**, che tratta i nomi come se fossero dichiarati a livello globale, una dichiarazione **using** è una dichiarazione interna allo scope corrente. Questo significa che puo' sovrapporre nomi provenienti da una direttiva **using**:

```
//: C10:DichiarazioniUsing.h
#ifndef DICHIARAZIONIUSING_H
#define DICHIARAZIONIUSING_H
namespace U {
  inline void f() {}
  inline void g() {}
}
namespace V {
  inline void f() {}
  inline void g() {}
}
#endif // DICHIARAZIONIUSING_H ///:~
//: C10:DichiarazioneUsing1.cpp
#include "DichiarazioniUsing.h"
void h() {
  using namespace U; // Direttiva using
  using V::f; // Dichiarazione using
  f(); // Chiama V::f();
  U::f(); // Bisogna qualificarla completamente per chiamarla
}
int main() {} ///:~
```

La dichiarazione **using** fornisce il nome di un identificatore completamente specificato, ma non da informazioni sul suo tipo. Questo vuol dire che se il namespace contiene un set di funzioni sovraccaricate con lo stesso nome, la dichiarazione **using** dichiara tutte le funzioni del set sovraccaricato. Si puo' mettere una dichiarazione **using** dovunque è possibile mettere una normale dichiarazione. Una dichiarazione **using** agisce come una normale dichiarazione eccetto per un aspetto: siccome non gli forniamo una lista di argomenti, è possibile che la dichiarazione **using** causi un sovraccaricamento di funzioni con lo stesso tipo di argomenti (cosa che non è permessa con un normale sovraccaricamento). Questa ambiguità, tuttavia, non si evidenzia nel momento della dichiarazione, bensì nel momento dell'uso. Una dichiarazione **using** puo' apparire anche all'interno di un namespace, e ha lo stesso effetto della dichiarazione dei nomi all'interno del namespace:

```
//: C10:DichiarazioneUsing2.cpp
#include "DichiarazioniUsing.h"
namespace Q {
  using U::f;
  using V::g;
  // ...
}
void m() {
  using namespace Q;
  f(); // Chiama U::f();
  g(); // Chiama V::g();
}
int main() {} ///:~
```

Una dichiarazione **using** è un alias, e permette di dichiarare le stesse funzioni in namespace diversi. Se si finisce per ridichiarare la stessa funzione importando namespace diversi, va bene, non ci saranno ambiguità o duplicazioni.

L' uso dei namespace

Alcune delle regole di sopra possono sembrare scoraggianti all'inizio, specialmente se si pensa di doverle usare continuamente. In generale, invece, si puo' fare un uso molto semplice dei namespace, una volta capito come funzionano. Il concetto chiave da ricordare è che introducendo una direttiva **using** globale (attraverso una "**using namespace**" esterna a qualsiasi scope) si apre il namespace per il file in cui è inserita. Questo in generale va bene per i file di implementazione (un file "**cpp**") perchè la direttiva **using** ha effetto solo fino alla fine della compilazione del file. Cioè non ha effetto su nessun altro file, percio' si puo' effettuare il controllo dei namespace un file di implementazione alla volta. Ad esempio, se si scopre un conflitto di nomi a causa dell'uso di diverse direttive **using** in un particolare file di implementazione, è semplice modificare quel file in modo che usi qualificazioni esplicite o dichiarazioni **using** per eliminare il conflitto, senza modificare altri file di implementazione.

Per gli header file le cose sono diverse. Non si dovrebbe mai introdurre una direttiva **using** all'interno di un header file, perchè questo significherebbe che tutti i file che lo includono si ritrovano il namespace aperto (e un header file puo' includere altri header file).

Così, negli header file si potrebbero usare sia qualificazioni esplicite o direttive **using** a risoluzione di scope che dichiarazioni **using**. Questa è la pratica che si usa in questo libro e seguendola non si rischia di "inquinare" il namespace globale e cadere nel mondo C++ precedente all'introduzione dei namespace.

Membri Statici in C++

A volte si presenta la necessità per tutti i membri di una classe di usare un singolo blocco di memoria. In C si puo' usare una variabile globale, ma questo non è molto sicuro. I dati globali possono essere modificati da chiunque, e i loro nomi possono collidere con altri nomi identici in un progetto grande. L'ideale sarebbe che il dato venisse allocato come se fosse globale, ma fosse nascosto all'interno di una classe e chiaramente associato a tale classe. Questo è ottenuto con dati membro **static** all'interno di una classe. C'è un singolo blocco di memoria per un dato membro **static**, a prescindere da quanti oggetti di quella classe vengono creati. Tutti gli oggetti condividono lo stesso spazio di memorizzione **static** per quel dato membro, percio' questo è un modo per loro di "comunicare" l'un l'altro. Ma il dato **static** appartiene alla classe; il suo nome è visibile solo all'interno della classe e puo' essere **public**, **private**, o **protected**.

Definire lo spazio di memoria per i dati membri statici

Siccome i dati membri **static** hanno un solo spazio di memoria a prescindere da quanti oggetti sono stati creati, questo spazio di memoria deve essere definito in un solo punto. Il compilatore non alloca memoria. Il linker riporta un errore se un dato membro **static** viene dichiarato ma non definito. La definizione deve essere fatta al di fuori della classe (non sono permesse definizioni inline), ed è permessa solo una definizione. Percio' è usuale mettere la definizione nel file di implementazione della classe. La sintassi a volte crea dubbi, ma in effetti è molto logica. Per esempio, se si crea un dato membro statico dentro una classe, come questo:

```
class A {
  static int i;
```

```
public:
  //...
};
```

Bisogna definire spazio di memoria per questo dato membro statico nel file di definizione, così:

```
int A::i = 1;
```

Se vogliamo definire una variabile globale ordinaria, dobbiamo scrivere:

```
int i = 1;
```

ma qui per specificare **A::i** vengono usati l'operatore di risoluzione di scope e il nome della classe. Alcuni provano dubbi all'idea che **A::i** sia **private** e che qui c'è qualcosa che sembra manipolarlo portandolo allo scoperto. Non è che questo rompe il meccanismo di protezione? E' una pratica completamente sicura per due motivi. Primo, l'unico posto in cui l'inizializzazione è legale è nella definizione. In effetti se il dato **static** fosse stato un oggetto con un costruttore, avremmo potuto chiamare il costruttore invece di usare l'operatore = (uguale). Secondo, una volta che la definizione è stata effettuata, l'utente finale non puo' effettuarne una seconda, il linker riporterebbe un errore. E il creatore della classe è forzato a creare una definizione, altrimenti il codice non si linka durante il test. Questo assicura che la definizione avvenga una sola volta ed è gestita dal creatore della classe. L'intera espressione di inizializzazione per un membro statico ricade nello scope della classe. Per esempio,

```
//: C10:Statinit.cpp
// Scope di inizializzatore static
#include <iostream>
using namespace std;

int x = 100;

class ConStatic {
  static int x;
  static int y;
public:
  void print() const {
    cout << "ConStatic::x = " << x << endl;
    cout << "ConStatic::y = " << y << endl;
  }
};

int ConStatic::x = 1;
int ConStatic::y = x + 1;
// ConStatic::x NON ::x

int main() {
  ConStatic cs;
  cs.print();
} ///:~
```

Qui la qualificazione **ConStatic::** estende lo scope di **ConStatic** all'intera definizione.

Inizializzazione di array static

Il capitolo 8 ha introdotto la variabile **static const** che permette di definire un valore costante all'interno del corpo di una classe. E' anche possibile creare array di oggetti **static**, sia **const** che non-**const**. La sintassi è ragionevolmente consistente:

```
//: C10:ArrayStatico.cpp
// Inizializzazione di array statici nelle classi
class Valori {
  // static consts vengono inizializzati sul posto:
  static const int scSize = 100;
  static const long scLong = 100;
```

```
  // Il conteggio automatico funziona con gli array statici.
  // Gli Array statici, sia non-integrali che non-const,
  // vanno inizializzati esternamente:
  static const int scInteri[];
  static const long scLongs[];
  static const float scTabella[];
  static const char scLettere[];
  static int size;
  static const float scFloat;
  static float tabella[];
  static char lettere[];
};

int Valori::size = 100;
const float Valori::scFloat = 1.1;

const int Valori::scInteri[] = {
  99, 47, 33, 11, 7
};

const long Valori::scLongs[] = {
  99, 47, 33, 11, 7
};

const float Valori::scTabella[] = {
  1.1, 2.2, 3.3, 4.4
};

const char Valori::scLettere[] = {
  'a', 'b', 'c', 'd', 'e',
  'f', 'g', 'h', 'i', 'j'
};

float Valori::tabella[4] = {
  1.1, 2.2, 3.3, 4.4
};

char Valori::lettere[10] = {
  'a', 'b', 'c', 'd', 'e',
  'f', 'g', 'h', 'i', 'j'
};

int main() { Valori v; } ///:~
```

Con **static const**s di tipi integrali si possono fornire definizioni all'interno della classe, ma per qualsiasi altra cosa (incluso array di tipi integrali, anche se sono **static const**) bisogna fornire un'unica definizione esterna per il membro. Queste definizioni hanno un linkage interno, cosicchè essi possono essere messi negli header file. La sintassi per inizializzare gli array statici è la stessa di qualunque altro aggregato, incluso il conteggio automatico. Si possono creare anche oggetti **static const** di tipi di classi e array di questi oggetti. Tuttavia non si possono inizializzare con la "sintassi inline" permessa per i tipi integrali **static const**s predefiniti.

```
//: C10:ArrayDiOggettiStatici.cpp
// Array static di oggetti di classi
class X {
  int i;
public:
  X(int ii) : i(ii) {}
};

class Stat {
  // Questo non funziona:
```

```
//!  static const X x(100);
  // Gli oggetti di classi statiche, sia const che non-const,
  // vanno inizializzati esternamente:
  static X x2;
  static X xTabella2[];
  static const X x3;
  static const X xTabella3[];
};

X Stat::x2(100);

X Stat::xTabella2[] = {
  X(1), X(2), X(3), X(4)
};

const X Stat::x3(100);

const X Stat::xTabella3[] = {
  X(1), X(2), X(3), X(4)
};

int main() { Stat v; } ///:~
```

L'inizializzazione di array di oggetti di classi sia **const** che non-**const static** deve essere effettuata allo stesso modo, seguendo la tipica sintassi della definizione **static**.

Classi nidificate e locali

Si possono facilmente mettere membri dati statici dentro classi nidificate all'interno di altre classi. La definizione di tali membri è un'estensione ovvia ed intuitiva - semplicemente si usa un altro livello di risoluzione di scope. Tuttavia non si possono avere dati membri **static** dentro classi locali (una classe locale è una classe definita dentro una funzione). Così,

```
//: C10:Locale.cpp
// Membri statici & classi locali
#include <iostream>
using namespace std;

// Classi nidificate POSSONO avere dati membri statici:
class Esterna {
  class Interna {
    static int i; // OK
  };
};

int Esterna::Interna::i = 47;

// Classi Locali NON possono avere dati membri statici:
void f() {
  class Locale {
  public:
//! static int i;  // Errore
    // (Come si potrebbe definire i?)
  } x;
}

int main() { Esterna x; f(); } ///:~
```

Si puo' notare subito un problema con i membri **static** in una classe locale: Come descrivere un dato membro a livello di file allo scopo di definirlo? Nella pratica le classi locali sono usate raramente.

funzioni membro static

Si possono creare anche funzioni membro **static** che, come i dati membro **static**, funzionano per la classe nel suo insieme, piuttosto che per un particolare oggetto della classe. Invece di costruire una funzione globale che vive e "inquina" il namespace globale e locale, si pone la funzione dentro la classe. Quando si crea una funzione membro **static** si esprime un'associazione con una classe particolare. Si puo' chiamare una funzione membro **static** nel modo ordinario, con il punto o la freccia, in associazione con un oggetto. Tuttavia è più usuale chiamare una funzione membro **static** da sola, senza nessun oggetto specifico, utilizzando l'operatore di risoluzione di scope, come nell'esempio seguente:

```
//: C10:FunzioneSempliceMembroStatico.cpp
class X {
public:
  static void f(){};
};

int main() {
  X::f();
} ///:~
```

Quando si vedono funzioni membro statiche in una classe, bisogna ricordarsi che il progettista aveva in mente che la funzione fosse concettualmente associata alla classe nella sua interezza. Una funzione membro **static** non puo' accedere ai dati membro ordinari, ma solo i membri **static**. Puo' chiamare solo altre funzioni membro **static**. Normalmente, quando una funzione membro viene chiamata le viene tacitamente passato l'indirizzo dell'oggetto corrente (**this**), ma un membro **static** non ha un puntatore **this**, e questa è la ragione per cui una funzione membro **static** non puo' chiamare i membri ordinari. Si ottiene anche un leggero aumento nella velocità di esecuzione, come per le funzioni globali, per il fatto che le funzioni membro **static** non hanno l'extra overhead di passare il parametro **this**. Nello stesso tempo si sfrutta il beneficio di avere la funzione dentro la classe. Per i dati membro, **static** indica che esiste un solo spazio di memoria per i dati membri per tutti gli oggetti di una classe. Questo parallelizza l'uso di **static** per definire oggetti *dentro* una funzione, per indicare che viene usata una sola copia di una variabile locale per tutte le chiamate alla funzione. Qui c'è un esempio che mostra membri dati **static** e funzioni membro **static** usati insieme:

```
//: C10:FunzioniMembroStatiche.cpp
class X {
  int i;
  static int j;
public:
  X(int ii = 0) : i(ii) {
    // Funzioni membro non-static possono accedere a
    // funzioni membro o dati statici:
   j = i;
  }
  int val() const { return i; }
  static int incr() {
    //! i++; // Errore: le funzioni membro statiche
    // non possono accedere ai dati membro non-static
    return ++j;
  }
  static int f() {
    //! val(); // Errore: le funzioni membro static
    // non possono accedere alle funzioni membro non-static
    return incr(); // OK -- chiama una funzione static
  }
};
```

```
int X::j = 0;

int main() {
  X x;
  X* xp = &x;
  x.f();
  xp->f();
  X::f(); // Funziona solo con membri static
} ///:~
```

Per il fatto di non disporre del puntatore **this**, le funzioni membro **static** nè possono accedere ai dati membri non-**static** nè possono chiamare funzioni membro non-**static**. Notare in **main( )** che un membro **static** puo' essere selezionato utilizzando l'usuale sintassi con il punto o con la freccia, associando la funzione con un oggetto, ma anche senza nessun oggetto (perchè un membro **static** è associato alla classe e non ad un particolare oggetto), usando il nome della classe e l'operatore di risoluzione di scope. Qui c'è un'importante caratteristica: a causa del modo in cui avviene l'inizializzazione di un oggetto membro **static**, si puo' mettere un dato membro **static** della stessa classe *all'interno* della classe. Qui c'è un esempio che permette ad un solo oggetto di tipo **Uovo** di esistere rendendo il costruttore private. Si puo' accedere all'oggetto, ma non si puo' creare un nuovo oggetto di tipo **Uovo**:

```
//: C10:UnicoEsemplare.cpp
// Membri statici di un qualche tipo, assicurano che
// esiste solo un oggetto di questo tipo.
// Viene detto anche "unico" esemplare ("singleton" pattern).
#include <iostream>
using namespace std;

class Uovo {
  static Uovo u;
  int i;
  Uovo(int ii) : i(ii) {}
  Uovo(const Uovo&); // Previene la  copy-costruzione
public:
  static Uovo* istanza() { return &u; }
  int val() const { return i; }
};

Uovo Uovo::u(47);

int main() {
//!  Uovo x(1); // Errore -- non puo' creare un Uovo
  // Si puo' accedere alla singola istanza:
  cout << Uovo::istanza()->val() << endl;
} ///:~
```

L'inizializzazione di **u** avviene dopo che è stata completata la dichiarazione della classe, così il compilatore ha tutte le informazioni di cui ha bisogno per allocare memoria ed effettuare la chiamata al costruttore. Per prevenire del tutto la creazione di un qualsiasi altro oggetto, è stato aggiundo qualcos'altro: un secondo costruttore private, detto *copy-constructor (costruttore di copia)*. A questo punto del libro non possiamo sapere perchè questo è necessario, perchè il costruttore di copia sarà introdotto nel prossimo capitolo. Tuttavia, come piccola anticipazione, se rimuoviamo il costruttore di copia definito nell'esempio precedente, potremmo creare un oggetto Uovo come questo:

```
Uovo u = *Uovo::istanza();
Uovo u2(*Uovo::istanza());
```

Entrambe le istruzioni di sopra usano il costruttore di copia, così per escludere questa possibilità il costruttore di copia è dichiarato come private (nessuna definizione è necessaria, perchè esso non sarà mai chiamato). Una larga parte del prossimo capitolo è dedicata al costruttore di copia e così sarà molto più chiaro.

Dipendenza dall' inizializzazione di oggetti statici

All'interno di un'unità di compilazione l'ordine di inizializzazione di oggetti statici è garantito essere uguale all'ordine in cui le definizioni degli oggetti appaiono nel file. L'ordine di distruzione è garantito essere l'inverso di quello di inizializzazione. Tuttavia non c'è nessuna garanzia nell'ordine di inizializzazione di oggetti statici *trasversalmente* alle unità di compilazione, e il linguaggio non fornisce nessun modo per specificare tale ordine. Questo puo' causare problemi significativi. Come esempio disastroso (che potrebbe causare l'interruzione del sistema operativo e la morte dei processi), se un file contiene

```
//: C10:Out.cpp {O}
// Primo file
#include <fstream>
std::ofstream out("out.txt"); ///:~
```

e un altro file usa l'oggetto **out** in uno dei suoi inizializzatori

```
//: C10:Oof.cpp
// Secondo file
//{L} Out
#include <fstream>
extern std::ofstream out;
class Oof {
public:
  Oof() { std::out << "ahi"; }
} oof;
int main() {} ///:~
```

il programma potrebbe funzionare, ma potrebbe anche non funzionare. Se l'ambiente di programmazione costruisce il programma in modo che il primo file viene inizializzato prima del secondo, non ci sarà nessun problema. Tuttavia se il secondo file viene inizializzato prima del primo file, il costruttore di **Oof** fa affidamento sull'esistenza di **out**, il quale pero' non è stato ancora costruito e questo provoca un caos. Questo problema succede soltanto con gli inizializzatori di oggetti statici che dipendono l'un l'altro. Gli oggetti statici in un'unità di compilazione vengono inizializzati prima che venga invocata qualsiasi funzione in tale unità - ma potrebbe essere dopo il **main( )**. Non si puo' essere sicuri dell'ordine di inizializzazione di oggetti statici che risiedono su file diversi. Un esempio subdolo puo' essere trovato in ARM.[47] In un file si ha, a livello globale:

```
extern int y;
int x = y + 1;
```

e in un secondo file si ha, sempre a livello globale:

```
extern int x;
int y = x + 1;
```

Per tutti gli oggetti statici, il meccanismo di linking-loading garantisce l'inizializzazione a zero prima che abbia luogo l'inizializzazione dinamica del programmatore. Nell'esempio precedente, l'azzeramento della memoria occupata dall'oggetto **fstream out** non ha un significato particolare, percio' essa è semplicemente indefinita fino a quando non viene chiamato il costruttore. Tuttavia, per i tipi predefiniti l'inizializzazione a zero ha sempre senso e se i file sono inizializzazti nell'ordine mostrato sopra, **y** ha valore iniziale zero e così **x** diventa uno e **y** diventa dinamicamente due. Ma se i file vengono inizializzati nell'ordine inverso, **x** viene staticamente inizializzato a zero, **y** viene dinamicamente inizializzato a uno e quindi **x** diventa due. I programmatori devono essere consapevoli di questo, altrimenti potrebbero creare un programma con dipendenze da inizializzazioni statiche che

possono funzionare su una piattaforma, ma spostandoli su un'altra piattaforma potrebbero misteriosamente non funzionare.

Cosa fare

Ci sono tre approcci per gestire questo problema:

1. Non farlo. Evitare le dipendenze da inizializzazione statica è la soluzione migliore.
2. Se bisogna farlo, è bene mettere le definizioni critiche di oggetti statici in un solo file, così da controllare la loro inizializzazione in modo portabile, mettendoli nell'ordine giusto.
3. Se si è convinti che è inevitabile sparpagliare oggetti statici in varie unità di compilazione - come nel caso di una libreria, dove non si puo' controllare l'uso che il programmatore ne fa - ci sono due tecniche di programmazine per risolvere il problema.

Tecnica numero uno

Questa tecnica è stata introdotta da Jerry Schwarz mentre creava la libreria iostream (giacchè le definizioni di **cin**, **cout** e **cerr** sono **static** e vivono in file separati). Questa è attualmente meno usata della seconda, ma è stata usata per molto tempo e ci si potrebbe imbattere in codice che la usa; percio' è importante capire come funziona. Questa tecnica richiede una classe addizionale nell'header file della libreria. Questa classe è responsabile dell'inizializzazione dinamica degli oggetti statici della libreria. Questo è un semplice esempio:

```
//: C10:Inizializzatore.h
// Tecnica di inizializzazione di oggetti static
#ifndef INIZIALIZZATORE_H
#define INIZIALIZZATORE_H
#include <iostream>
extern int x; // Dichiarazioni, non definizioni
extern int y;

class Inizializzatore {
  static int initCount;
public:
  Inizializzatore() {
    std::cout << "Inizializzatore()" << std::endl;
    // Inizializza solo la prima volta
    if(initCount++ == 0) {
      std::cout << "effettua l'inizializzazione"
                << std::endl;
      x = 100;
      y = 200;
    }
  }
  ~Inizializzatore() {
    std::cout << "~Inizializzatore()" << std::endl;
    // Cancella solo l'ultima volta
    if(--initCount == 0) {
      std::cout << "effettua la cancellazione"
                << std::endl;
      // Qualsiasi altra pulizia qui
    }
  }
};
```

```
// L'istruzione seguente crea un oggetto in ciascun file
// in cui Inizializzatore.h viene incluso, ma questo oggetto è visibile solo in questo file:
static Inizializzatore init;
#endif // INIZIALIZZATORE_H ///:~
```

Le dichiarazioni di **x** e **y** annunciano solo che questi oggetti esistono, ma non allocano memoria per essi. Tuttavia, la definizione di **Inizializzatore init** alloca memoria per questo oggetto in tutti i file in cui l'header file è incluso. Ma siccome il nome è **static** (che controlla la visibilità, non il modo in cui viene allocata la memoria; la memorizzazione è a livello di file per default), esso è visibile solo all'interno dell'unità di compilazione e il linker non si "arrabbia" per questa definizione multipla. Qui c'è il file per le definizioni di **x**, **y**, e **initCount**:

```
//: C10:DefinInizializzatore.cpp {O}
// Definizioni per Inizializzatore.h
#include "Inizializzatore.h"
// L'inizializzazione Static forza
// tutti questi valori a zero:
int x;
int y;
int Inizializzatore::initCount;
///:~
```

(Naturalmente, un'istanza di **init** viene posta anche in questo file quando l'header viene incluso.) Supponiamo che l'utente della libreria crei altri due file:

```
//: C10:Inizializzatore.cpp {O}
// Inizializzazione Static
#include "Inizializzatore.h"
///:~
```

e

```
//: C10:Inizializzatore2.cpp
//{L} DefinInizializzatore
// Inizializzazione Static
#include "Inizializzatore.h"
using namespace std;

int main() {
  cout << "dentro il main()" << endl;
  cout << "all'uscita del main()" << endl;
} ///:~
```

Adesso non ha importanza quale unità di compilazione viene inizializzata per prima. La prima volta che una unità di compilazione contenente **Inizializzatore.h** viene inizializzata, **initCount** sarà posto a zero e così l'inizializzazione sarà effettuata (questo dipende pesantemente dal fatto che l'area di memoria statica viene inizializzata a zero prima che qualsiasi inizializzazione dinamica abbia luogo). Per tutte le altre unità di compilazione, **initCount** sarà diverso da zero e l'inizializzazione viene saltata. La cancellazione avviene nell'ordine inverso e **~Inizializzatore( )** assicura che cio' avvenga una volta sola. Questo esempio ha usato tipi predefiniti come oggetti statici globali. La tecnica funziona anche con le classi, ma questi oggetti devono essere inizializzati dinamicamente dalla classe **Inizializzatore**. Un modo per fare questo è di creare classi senza costruttori e distruttori, ma con funzioni membro per l'inizializzazione e la cancellazione che usano nomi diversi. Un approccio più comune, comunque, è quello di avere puntatori ad oggetti e di crearli usando **new** dentro **Inizializzatore( )**.

Tecnica numero due

Dopo tanto tempo che era in uso la prima tecnica, qualcuno (non so chi), ha introdotto la tecnica presentata in questa sezione, che è molto più semplice e chiara della prima. Il fatto che c'è voluto tanto tempo per scoprirla è il tributo da pagare alla complessità del C++. La

tecnica si basa sul fatto che gli oggetti statici all'interno delle funzioni vengono inizializzati (soltanto) la prima volta che la funzione viene chiamata. Bisogna ricordare che il problema che stiamo cercando di risolvere non è *quando* gli oggetti statici vengono inizializzati (cosa che possiamo controllare separatamente), ma piuttosto assicurare che l'inizializzazione avvenga nell'ordine appropriato. Questa tecnica è molto pulita e intelligente. Per ogni oggetto che ha una dipendenza dall'inizializzazione, poniamo un oggetto statico all'interno della funzione che restituisce un riferimento a quell'oggetto. Così, l'unico modo che abbiamo per accedere all'oggetto statico è chiamare la funzione, e se quest'oggetto ha la necessità di accedere ad altri oggetti statici da cui esso dipende, deve chiamare le *loro* funzioni. E la prima volta che la funzione viene chiamata, forza l'inizializzazione. La correttezza dell'ordine di inizializzazione statica è garantita dal modo in cui viene costruito il codice e non da un ordine arbitrario stabilito dal linker. Per fare un esempio, qui ci sono due classi che dipendono l'una dall'altra. La prima contiene un **bool** che è inizializzato soltanto dal costruttore, così possiamo verificare se il costruttore è stato chiamato per un'istanza statica della classe (l'area di memorizzazione statica viene inizializzata a zero allo startup del programma, il che da luogo ad un valore **false** per un **bool** se il costruttore non è stato chiamato):

```
//: C10:Dipendenza1.h
#ifndef DIPENDENZA1_H
#define DIPENDENZA1_H
#include <iostream>

class Dipendenza1 {
  bool init;
public:
  Dipendenza1() : init(true) {
    std::cout << "Costruzione di Dipendenza1"
              << std::endl;
  }
  void print() const {
    std::cout << "Init di Dipendenza1: "
              << init << std::endl;
  }
};
#endif // DIPENDENZA1_H ///:~
```

Il costruttore, inoltre, si annuncia quando viene chiamato e quindi possiamo stampare, con **print( )**, lo stato dell'oggetto per scoprire se è stato inizializzato. La seconda classe viene inizializzata da un oggetto della prima classe, che è quello che causa la dipendenza:

```
//: C10:Dipendenza2.h
#ifndef DIPENDENZA2_H
#define DIPENDENZA2_H
#include "Dipendenza1.h"

class Dipendenza2 {
  Dipendenza1 d1;
public:
  Dipendenza2(const Dipendenza1& dip1): d1(dip1){
    std::cout << "Costruzione di Dipendenza2";
    print();
  }
  void print() const { d1.print(); }
};
#endif // DIPENDENZA2_H ///:~
```

Il costruttore annuncia se stesso e stampa lo stato dell'oggetto **d1** cosicchè possiamo vedere se è stato inizializzato nel momento in cui il costruttore viene chiamato. Per dimostrare cosa puo' andare storto, il file seguente pone dapprima le definizioni degli oggetti statici nell'ordine sbagliato, come potrebbe succedere se il linker inizializzasse

dapprima l'oggetto **Dipendenza2** rispetto all'oggetto **Dipendenza1**. Successivamente l'ordine di definizione viene invertito per mostrare come funziona correttamente se l'ordine di inizializzazione è "giusto." Per ultimo, viene dimostrata la tecnica numero due. Per fornire un output più leggibile, viene creata la funzione **separatore( )**. Il trucco è quello di non permettere di chiamare una funzione globalmente a meno che la funzione non è usata per effettuare l'inizializzazione di una variabile, così la funzione **separatore( )** restituisce un valore fittizio che viene usato per inizializzare una coppia di variabili globali.

```
//: C10:Tecnica2.cpp
#include "Dipendenza2.h"
using namespace std;

// Restituisce un valore cosicchè puo' essere chiamata
// come inizializzatore globale:
int separatore() {
  cout << "----------------------" << endl;
  return 1;
}

// Simula il problema della dipendenza:
extern Dipendenza1 dip1;
Dipendenza2 dip2(dip1);
Dipendenza1 dip1;
int x1 = separatore();

// Ma se avviene in questo ordine funziona correttamente:
Dipendenza1 dip1b;
Dipendenza dip2b(dip1b);
int x2 = separatore();

// Seguono oggetti static inglobati all'interno di funzioni
Dipendenza1& d1() {
  static Dipendenza1 dip1;
  return dip1;
}

Dipendenza2& d2() {
  static Dipendenza2 dip2(d1());
  return dip2;
}

int main() {
  Dipendenza2& dip2 = d2();
} ///:~
```

Le funzioni **d1( )** e **d2( )** inglobano istance statiche degli oggetti **Dipendenza1** e **Dipendenza2**. A questo punto, l'unico modo che abbiamo per ottenere gli oggetti statici è quello di chiamare le funzioni e quindi forzare l'inizializzazione degli oggetti statici alla prima chiamata delle funzioni. Questo significa che l'inizializzazione è garantita essere corretta, come si puo' vedere facendo girare il programma e osservando l'output. Qui è mostrato come organizzare effettivamente il codice per usare la tecnica esposta. Ordinariamente gli oggetti statici dovrebbero essere definiti in file separati (perchè si è costretti per qualche motivo; ma va ricordato che la definizione degli oggetti in file separati è cio' che causa il problema), qui invece definiamo le funzioni contenitrici in file separati. Ma è necessario definirle in header file:

```
//: C10:Dipendenza1StatFun.h
#ifndef DIPENDENZA1STATFUN_H
```

```
#define DIPENDENZA1STATFUN_H
#include "Dipendenza1.h"
extern Dipendenza1& d1();
#endif // DIPENDENZA1STATFUN_H ///:~
```

In realtà lo specificatore "extern" è ridondante per la dichiarazione di una funzione. Qui c'è il secondo header file:

```
//: C10:Dipendenza2StatFun.h
#ifndef DIPENDENZA2STATFUN_H
#define DIPENDENZA2STATFUN_H
#include "Dipendenza2.h"
extern Dipendenza2& d2();
#endif // DIPENDENZA2STATFUN_H ///:~
```

A questo punto, nei file di implementazione dove prima avremmo messo la definizione degli oggetti statici, mettiamo la definizione delle funzioni contenitrici:

```
//: C10:Dipendenza1StatFun.cpp {O}
#include "Dipendenza1StatFun.h"
Dipendenza1& d1() {
  static Dipendenza1 dip1;
  return dip1;
} ///:~
```

Presumibilmente andrebbe messo altro codice in questo file. Segue l'altro file:

```
//: C10:Dipendenza2StatFun.cpp {O}
#include "Dipendenza1StatFun.h"
#include "Dipendenza2StatFun.h"
Dipendenza2& d2() {
  static Dipendenza2 dip2(d1());
  return dip2;
} ///:~
```

Così adesso ci sono due file che potrebbero essere linkati in qualsiasi ordine e se contenessero oggetti statici ordinari potrebbero dar luogo ad un ordine di inizializzazione qualsiasi. Ma siccome essi contengono funzioni che fanno da contenitore, non c'è nessun pericolo di inizializzazione errata:

```
//: C10:Tecnica2b.cpp
//{L} Dipendenza1StatFun Dipendenza2StatFun
#include "Dipendenza2StatFun.h"
int main() { d2(); } ///:~
```

Quando facciamo girare questo programma possiamo vedere che l'inizializzazione dell'oggetto statico **Dipendenza1** avviene sempre prima dell'inizializzazione dell'oggetto statico **Dipendenza2**. Possiamo anche notare che questo approccio è molto più semplice della tecnica numero uno. Si puo' essere tentati di scrivere **d1( )** e **d2( )** come funzioni inline dentro i rispettivi header file, ma questa è una cosa che dobbiamo evitare. Una funzione inline puo' essere duplicata in tutti i file in cui appare- e la duplicazione *include* la definizione dell' oggetto statico. Siccome le funzioni inline hanno per default un linkage interno, questo puo' significare avere oggetti statici multipli attraverso le varie unità di compilazione, che potrebbe sicuramente creare problemi. Percio' dobbiamo assicurare che ci sia una sola definizione di ciascuna funzione contenitrice, e questo significa non costruire tali funzioni inline.

Specificazione di linkage alternativi

Cosa succede se stiamo scrivendo un programma in C++ e vogliamo usare una libreria C? Se prendiamo la dichiarazione di funzione C,

```
float f(int a, char b);
```

il compilatore C++ decorerà questo nome con qualcosa tipo **_f_int_char** per supportare l' overloading di funzioni (e il linkage tipo-sicuro). Tuttavia il compilatore C che ha compilato la libreria certamente non ha decorato il nome della funzione, così il suo nome

interno sarà **_f**. Percio' il linker non sarà in grado di risolvere in C++ la chiamata a **f( )**. La scappatoia fornita dal C++ è la *specificazione di linkage alternativo*, che si ottiene con l'overloading della parola chiave **extern**. La parola **extern** viene fatta seguire da una stringa che specifica il linkage che si vuole per la dichiarazione, seguita dalla dichiarazione stessa:

```
extern "C" float f(int a, char b);
```

Questo dice al compilatore di conferire un linkage tipo C ad **f( )** così il compilatore non decora il nome. Gli unici due tipi di specificazioni di linkage supportati dallo standard sono **"C"** e **"C++,"** ma i fornitori di compilatori hanno l'opzione di supportare altri linguaggi allo stesso modo. Se si ha un gruppo di dichiarazioni con linkage alternativi, bisogna metterle tra parentesi graffe, come queste:

```
extern "C" {
  float f(int a, char b);
  double d(int a, char b);
}
```

Oppure, per un header file,

```
extern "C" {
#include "Mioheader.h"
}
```

Molti fornitori di compilatori C++ gestiscono le specificazioni di linkage alternativi all'interno dei loro header file che funzionano sia con il C che con il C++, in modo che non ci dobbiamo preoccupare al riguardo.

Sommario

La parola chiave **static** puo' generare confusione, in quanto in alcune situazioni controlla la posizione in memoria, mentre in altre controlla la visibilità e il linkage dei nomi. Con l'introduzione dei namespaces, caratteristica propria del C++, abbiamo un'alternativa migliore e molto più flessibile per controllare la proliferazione dei nomi in progetti grandi. L'uso di **static** all'interno delle classi è un modo ulteriore di controllare i nomi in un programma. I nomi (all'interno delle classi) non confliggono con quelli globali e la visibilità e l'accesso sono tenuti dentro i confini del programma, fornendo maggior controllo nella manutenzione del codice.

---

[47]Bjarne Stroustrup and Margaret Ellis, *The Annotated C++ Reference Manual*, Addison-Wesley, 1990, pp. 20-21.

11: I Riferimenti & il Costruttore di Copia

I Riferimenti sono come i puntatori costanti, che sono dereferenziati automaticamente dal compilatore.

Benchè i riferimenti esistano anche in Pascal, la versione del C++ è stata presa dal linguaggio Algol. È essenziale in C++ supportare la sintassi dell'overloading degli operatori (vedi Capitolo 12), ma c'è anche una generale convenienza nel controllare come gli argomenti vengono passati dentro e fuori le funzioni.

In questo capitolo daremo prima brevemente uno sguardo alle differenze tra i puntatoti in C e C++, per poi introdurre i riferimenti. Ma la parte più consistente del capitolo è rivolta a indagare su un aspetto abbastanza confuso per i nuovi programmatori in C++: il costruttore di copia, uno speciale costruttore (che richiede i riferimenti) che costruisce un nuovo oggetto a partire da un oggetto già esistente dello stesso tipo. Il costruttore di copia viene usato dal compilatore per passare alle funzioni e ritornare dalle funzioni oggetti *per valore*.

Infine viene fatta luce su una caratteristica oscura del C++, che è il *puntatore-a-membro*.

Puntatori in C++

La differenza principale che c'è tra i puntatori in C e quelli in C++ è che il C++ è un linguaggio fortemente tipizzato. Questo viene fuori, ad esempio, laddove abbiamo a che fare con **void***. Il C non permette di assegnare a caso un puntatore di un tipo ad un altro tipo, ma esso *permette* di fare ciò attraverso **void***. Ad esempio:

```
bird* b;
rock* r;
void* v;
v = r;
b = v;
```

Siccome questa "caratteristica" del C permette di trattare, di nascosto, qualunque tipo come qualunque altro, esso apre una voragine nel sistema dei tipi. Il C++ non permette ciò; il compilatore da un messaggio di errore, e se si vuole davvero trattare un tipo come un altro, bisogna farlo esplicitamente usando il cast, sia per il compilatore che per il lettore. (Il Capitolo 3 ha introdotto la sintassi migliorativa del casting "esplicito" del C++.)

Riferimenti in C++

Un *riferimento* (**&**) è come un puntatore costante, che viene automaticamente dereferenziato. Viene usato generalmente per le liste di argomenti e per i valori di ritorno delle funzioni. Ma si possono costruire anche riferimenti isolati. Per esempio,

```
//: C11:FreeStandingReferences.cpp
#include <iostream>
using namespace std;

// Riferimento ordinario isolato:
int y;
```

```
int& r = y;
// Quando viene creato un riferimento, deve essere
// inizializzato per riferirsi ad un oggetto vero.
// Tuttavia si può anche scrivere:
const int& q = 12;  // (1)
// I riferimenti sono legati all'indirizzo di memoria di qualcos'altro:
int x = 0;          // (2)
int& a = x;         // (3)
int main() {
  cout << "x = " << x << ", a = " << a << endl;
  a++;
  cout << "x = " << x << ", a = " << a << endl;
} ///:~
```

Nella linea (1), il compilatore alloca uno spazio di memoria, lo inizializza con il valore 12 e fissa un riferimento a questo spazio di memoria. Il punto è che un riferimento deve essere legato a *qualche altro* pezzo di memoria. Quando si accede ad un riferimento, di fatto si accede a questo pezzo di memoria. Così, se scriviamo linee come la (2) e la (3), allora incrementare **a** corrisponde di fatto a incrementare **x**, come è mostrato nel **main( )**. Diciamo che il modo più semplice di pensare ad un riferimento è come un puntatore elaborato. Un vantaggio di questo "puntatore" è che non bisogna preoccuparsi di inizializzarlo (il compilatore ne forza l'inizializzazione) e come dereferenziarlo (lo fa il compilatore).

Ci sono determinate regole quando si usano i riferimenti:

1. Un riferimento deve essere inizializzato quando viene creato. (I puntatori possono essere inizializzati in qualunque momento.)
2. Quando un riferimento viene inizializzato per riferirsi ad un oggetto, non può essere cambiato per riferirsi ad un altro oggetto. (I puntatori possono essere puntati verso un altro oggetto in qualunque momento.)
3. Non esiste il riferimento NULL . Deve essere sempre possibile assumere che un riferimento sia legato a un pezzo di memoria valido.

I Riferimenti nelle funzioni

Il posto più comune dove si possono trovare i riferimenti è negli argomenti di funzioni e nei valori di ritorno delle stesse. Quando un riferimento viene usato come argomento di una funzione, ogni modifica al riferimento *dentro* la funzione causa cambiamenti all'argomento *fuori* dalla funzione. Naturalmente si può ottenere lo stesso effetto con un puntatore, ma un riferimento ha una sintassi molto più pulita. (Si può pensare, se volete, ai riferimenti come una pura convenienza sintattica.)

Se si ritorna un riferimento da una funzione, bisogna usare la stessa accortezza che si userebbe se si ritornasse un puntatore. Qualunque sia l'oggetto a cui un riferimento è connesso, questo non deve essere buttato via al ritorno dalla funzione, altrimenti si avrebbe un riferimento a una zona di memoria sconosciuta.

Qui c'è un esempio:

```
//: C11:Reference.cpp
// Semplici riferimenti del C++

int* f(int* x) {
  (*x)++;
```

```
  return x; // Sicuro, x è fuori da questo scope
}

int& g(int& x) {
  x++; // Lo stesso effetto che in f()
  return x; // Sicuro, fuori da questo scope
}

int& h() {
  int q;
//!  return q;  // Errore
  static int x;
  return x; // Sicuro, x vive fuori da questo scope
}

int main() {
  int a = 0;
  f(&a); // Brutto (ma esplicito)
  g(a);  // Pulito (ma nascosto)
} ///:~
```

La chiamata alla funzione **f( )** non ha la stessa convenienza e chiarezza dell'uso del riferimento, ma è chiaro che viene passato un indirizzo. Nella chiamata alla funzione **g( )**, viene passato un indirizzo (attraverso un riferimento), ma non lo si vede.

riferimenti const

L'argomento passato per riferimento in **Reference.cpp** funziona solo quando l'argomento è un oggetto non-**const**. Se è un oggetto **const**, la funzione **g( )** non accetta l'argomento, il che in effetti è una buona cosa, perchè la funzione *effettua* modifiche all'argomento esterno. Se sappiamo che la funzione rispetterà la **const**anza di un oggetto, allora porre l'argomento come riferimento **const** permette di usare la funzione in qualunque situazione. Questo significa che, per i tipi predefiniti, la funzione non deve modificare gli argomenti, mentre per i tipi definiti dall'utente la funzione deve chiamare solo funzioni membro **const** e non deve modificare nessun dato membro **public**.

L'uso di riferimenti **const** negli argomenti di funzioni è importante soprattutto perchè la funzione può ricevere oggetti temporanei. Questo può essere stato creato come valore di ritorno da un'altra funzione o esplicitamente dall'utente della nostra funzione. Gli oggetti temporanei sono sempre **const**, per cui se non si usa un riferimento **const**, l'argomento non viene accettato dal compilatore. Come esempio molto semplice,

```
//: C11:ConstReferenceArguments.cpp
// Passaggio dei riferimenti come const

void f(int&) {}
void g(const int&) {}

int main() {
//!  f(1); // Errore
  g(1);
} ///:~
```

La chiamata a **f(1)** causa un errore di compilazione, perchè il compilatore deve prima creare un riferimento. E questo lo fa allocando memoria per un **int**, inizializzandola a 1 e producendo l'indirizzo da legare al riferimento. Il dato memorizzato *deve* essere **const** perchè cambiarlo non ha senso – non si possono mai mettere le mani su di esso. Bisogna

fare la stessa assunzione per tutti gli oggetti temporanei: cioè che sono inaccessibili. Il compilatore è in grado di valutare quando si sta cambiando un dato del genere, perchè il risultato sarebbe una perdita di informazione.

Riferimenti a puntatori

In C, se si vuole modificare il *contenuto* di un puntatore piuttosto che l'oggetto a cui punta, bisogna dichiarare una funzione come questa:

```
void f(int**);
```

e bisogna prendere l'indirizzo del puntatore quando lo si passa alla funzione:

```
int i = 47;
int* ip = &i;
f(&ip);
```

Con i riferimenti in C++, la sintassi è piu chiara. L'argomento della funzione diventa un riferimento a un puntatore, e non bisogna prendere l'indirizzo del puntatore.

```
//: C11:ReferenceToPointer.cpp
#include <iostream>
using namespace std;

void increment(int*& i) { i++; }

int main() {
  int* i = 0;
  cout << "i = " << i << endl;
  increment(i);
  cout << "i = " << i << endl;
} ///:~
```

Facendo girare questo programma si può provare che è il puntatore ad essere incrementato e non ciò a cui punta.

Linee guida sul passaggio degli argomenti

Dovrebbe essere normale abitudine passare gli argomenti ad una funzione come riferimento **const**. Anche se questo potrebbe sulle prime sembrare solo un problema di efficienza (e noi non vogliamo preoccuparci di affinare l'efficienza mentre progettiamo e implementiamo un programma), c'è molto di più in gioco: come vedremo nel resto del capitolo, per passare un oggetto per valore è necessario un costruttore di copia e questo non sempre è disponibile.

Il miglioramento dell'efficienza può essere sostanziale se si usa una tale accortezza: passare un argomento per valore richiede la chiamata a un costruttore e a un distruttore, ma se non si deve modificare l'argomento, il passaggio come riferimento **const** richiede soltanto un push di un indirizzo sullo stack.

Infatti, l'unica occasione in cui non è preferibile passare un indirizzo è quando si devono fare tali modifiche all'oggeto che il passaggio per valore è il solo approccio sicuro (piuttosto che modificare l'oggetto esterno, cosa che il chiamante in genere non si aspetta). Questo è l'argomento della prossima sezione.

Il costruttore di copia

Adesso che abbiamo capito i concetti base dei riferimenti in C++, siamo pronti per affrontare uno dei concetti più confusi del linguaggio: il costruttore di copia, spesso chiamato **X(X&)** ("rif X di X "). Questo costruttore è essenziale per controllare il passaggio e il ritorno di tipi definiti dall'utente durante le chiamate a funzioni. È così importante, di fatto, che il compilatore sintetizza automaticamente un costruttore di copia se non se ne fornisce uno, come vedremo.

Il passaggio & il ritorno per valore

Per comprendere la necessità del costruttore di copia, consideriamo il modo in cui il C gestisce il passaggio e il ritorno di variabili per valore durante le chiamate a funzioni. Se dichiariamo una funzione e la chiamiamo,

```
int f(int x, char c);
int g = f(a, b);
```

come fa il compilatore a sapere come passare e restituire queste variabili? È gia noto! Il range dei tipi con cui ha a che fare è così piccolo – **char**, **int**, **float**, **double**, e le loro varianti – che questa informazione è già presente nel compilatore.

Se facciamo generare il codice assembly al nostro compilatore e andiamo a vedere le istruzioni generate per la chiamata alla funzione **f( )**, vedremmo qualcosa equivalente a:

```
push  b
push  a
call  f()
add  sp,4
mov  g, register a
```

Questo codice è stato significativamente ripulito per renderlo generico; le espressioni per **b** ed **a** potrebbero essere diverse, a seconda che le variabili sono globali (in qual caso esse saranno **_b** e **_a**) o locali (il compilatore le indicizzerà a partire dallo stack pointer). Questo è vero anche per l'espressione di **g**. La forma della chiamata ad **f( )** dipenderà dallo schema di decorazione dei nomi, e "register a" dipende da come sono chiamati i registri della CPU all'interno dell'assembler della macchina. La logica che c'è dietro, comunque, rimane la stessa.

In C e C++, dapprima vengono messi sullo stack gli argomenti, da destra a sinistra, e poi viene effettuata la chiamata alla funzione. Il codice di chiamata è responsabile della cancellazione degli argomenti dallo stack (cosa che giustifica l'istruzione **add sp,4**). Ma bisogna notare che per passare gli argomenti per valore, il compilatore ne mette semplicemente una copia sullo stack – esso ne conosce le dimensioni e quindi il push di questi argomenti ne produce una copia accurata.

Il valore di ritorno di **f( )** è messo in un registro. Di nuovo, il compilatore conosce tutto ciò che c'è da conoscere riguardo al tipo di valore da restituire, in quanto questo tipo è incorporato nel linguaggio e il compilatore lo può restituire mettendolo in un registro. Con i tipi primitivi del C, il semplice atto di copiare i bit del valore è equivalente a copiare l'oggetto.

Passare & ritornare oggetti grandi

Ma adesso consideriamo i tipi definiti dall'utente. Se creiamo una classe e vogliamo passare un oggetto di questa classe per valore, come possiamo supporre che il compilatore sappia cosa fare? Questo non è un tipo incorporato nel compilatore, ma un tipo che abbiamo creato noi.

Per investigare sul problema, possiamo iniziare con una semplice struttura che è chiaramente troppo grande per essere restituita in un registro:

```
//: C11:PassingBigStructures.cpp
struct Big {
  char buf[100];
  int i;
  long d;
} B, B2;

Big bigfun(Big b) {
  b.i = 100; // Opera sull'argomento
  return b;
}

int main() {
  B2 = bigfun(B);
} ///:~
```

Decodificare l'output in assembly in questo caso è un pò più complicato perchè molti compilatori usano funzioni di "appoggio" invece di mettere tutte le funzionalità inline. Nel **main( )**, la chiamata a **bigfun ( )** parte come possiamo immaginare – l'intero contenuto di **B** è messo sullo stack. (Qui alcuni compilatori caricano dei registri con l'indirizzo di **Big** e la sua dimensione e poi chiamano una funzione di appoggio per mettere **Big** sullo stack.)

Nel frammento di codice precedente, mettere gli argomenti sullo stack era tutto ciò che era richiesto prima della chiamata alla funzione. In **PassingBigStructures.cpp**, tuttavia, possiamo vedere un'azione in più: prima di effettuare la chiamata viene fatto il push dell'indirizzo di **B2** anche se è ovvio che non è un argomento. Per capire cosa succede qui, è necessario capire quali sono i vincoli del compilatore quando effettua una chiamata ad una funzione.

Struttura di stack in una chiamata a funzione

Quando il compilatore genera codice per una chiamata a funzione, prima mette gli argomenti sullo stack e poi effettua la chiamata. All'interno della funzione viene generato del codice per far avanzare lo stack pointer anche più di quanto necessario per fornire spazio alle variabili locali della funzione. ("Avanzare" può significare spostare in giù o in su a seconda della macchina.) Ma durante una CALL in linguaggio assembly la CPU fa il push dell'indirizzo del codice di programma da cui la funzione è stata chiamata, così l'istruzione assembly RETURN può usare questo indirizzo per ritornare nel punto della chiamata. Questo indirizzo naturalmente è sacro, perchè senza di esso il programma viene completamente perso. Qui viene mostrato come potrebbe apparire la struttura di stack dopo una CALL e l'allocazione di variabili locali in una funzione:

Il codice generato per il resto della funzione si aspetta che la memoria sia srutturata esattamente in questo modo, in modo tale che esso può tranquillamente pescare tra gli argomenti della funzione e le variabili locali senza toccare l'indirizzo di ritorno. Chiameremo questo blocco di memoria, che contiene tutto ciò che viene usato dalla funzione durante il processo di chiamata, *struttura di funzione (function frame)*.

Possiamo pensare che è ragionevole tentare di restituire i valori attraverso lo stack. Il compilatore dovrebbe semplicemente metterli sullo stack, e la funzione dovrebbe restituire un offset per indicare la posizione nello stack da cui inizia il valore di ritorno.

Ri-entranza

Il problema sussiste perchè le funzioni in C e C++ supportano gli interrupt; cioè i linguaggi sono *ri-entranti*. Essi supportano anche le chiamate ricorsive alle funzioni. Questo significa che in qualsiasi punto dell'esecuzione del programma può arrivare un interrupt, senza interrompere il programma. Naturalmente la persona che scrive la routine di gestione dell'interrupt (interrupt service routine, ISR) è responsabile del salvataggio e del ripristino di tutti i registri usati nella ISR, ma se l'ISR ha bisogno di ulteriore memoria sullo stack, questo deve essere fatto in maniera sicura. (Si può pensare ad un ISR come ad una normale funzione senza argomenti e con un valore di ritorno **void** che salva e ripristina lo stato della CPU. La chiamata ad una ISR è scatenata da qualche evento hardware piuttosto che da una chiamata esplicita all'interno del programma.)

Adesso proviamo ad immaginare cosa potrebbe succedere se una funzione ordinaria provasse a restituire un valore attraverso lo stack. Non si può toccare nessuna parte dello stack che si trova al di sopra dell'indirizzo di ritorno, così la funzione dovrebbe mettere i valori al di sotto dell'indirizzo di ritorno. Ma quando viene eseguita l'istruzione assembly RETURN lo stack pointer deve puntare all'indirizzo di ritorno (o al di sotto di esso, dipende dalla macchina), così appena prima del RETURN la funzione deve spostare in su lo stack pointer, tagliando fuori in questo modo tutte le variabili locali. Se proviamo a ritornare dei valori sullo stack al di sotto dell'indirizzo di ritorno, è proprio in questo momento che diventiamo vulnerabili, perchè potrebbe arrivare un interrupt. L' ISR sposterà in avanti lo stack pointer per memorizzare il suo indirizzo di ritorno e le sue variabili locali e così sovrascrive il nostro valore di ritorno.

Per risolvere questo problema il chiamante *dovrebbe* essere responsabile dell'allocazione di uno spazio extra sullo stack per il valore di ritorno, prima di chiamare la funzione. Tuttavia, il C non è stato progettato per fare questo, e il C++ deve mantenere la

compatibilità. Come vedremo brevemente, il compilatore C++ usa uno schema molto più efficiente.

Un'altra idea potrebbe essere quella di restituire il valore in qualche area dati globale, ma anche questa non può funzionare. Rientranza vuol dire che qualunque funzione può essere una routine di interrupt per qualunque altra funzione, *inclusa la funzione corrente*. Così, se mettiamo il valore di ritorno in un'area globale, possiamo ritornare nella stessa funzione, che a questo punto sovrascrive il valore di ritorno. La stessa logica si applica alla ricorsione.

L'unico posto sicuro dove restituire i valori sono i registri, così siamo di nuovo al problema di cosa fare quando i registri non sono grandi abbastanza per memorizzare il valore di ritorno. La risposta è nel mettere sullo stack come un argomento della funzione l'indirizzo dell'area di destinazione del valore di ritorno, e far si che la funzione copi le informazioni di ritorno direttamente nell'area di destinazione. Questo non solo risolve tutti i problemi, ma è molto efficiente. Questo è anche il motivo per cui, in **PassingBigStructures.cpp**, il compilatore fa il push dell'indirizzo di **B2** prima di chiamare **bigfun( )** nel **main( )**. Se guardiamo all'output in assembly di **bigfun( )**, possiamo vedere che essa si aspetta questo argomento nascosto e ne effettua la copia nell'area di destinazione *all'interno* della funzione.

Copia di bit contro inizializzazione

Fin qui tutto bene. Abbiamo una strada percorribile per il passaggio e il ritorno di grosse, ma semplici, strutture. Ma va notato che tutto quello che abbiamo è un modo per copiare bit da una parte all'altra della memoria, cosa che funziona bene certamente per il modo primitivo in cui il C vede le variabili. Ma in C++ gli oggetti possono essere molto più sofisticati di un mucchio di bit; essi hanno un significato. Questo significato potrebbe non rispondere bene ad una copia di bit.

Consideriamo un semplice esempio: una classe che sa quanti oggetti del suo tipo ci sono in ogni istante. Dal Capitolo 10 sappiamo che per fare questo si include un membro dati **static**:

```
//: C11:HowMany.cpp
// Una classe che conta i suoi oggetti
#include <fstream>
#include <string>
using namespace std;
ofstream out("HowMany.out");

class HowMany {
  static int objectCount;
public:
  HowMany() { objectCount++; }
  static void print(const string& msg = "") {
    if(msg.size() != 0) out << msg << ": ";
    out << "objectCount = "
         << objectCount << endl;
  }
  ~HowMany() {
    objectCount--;
    print("~HowMany()");
  }
};
```

```
int HowMany::objectCount = 0;

// Passaggio e ritorno PER VALORE:
HowMany f(HowMany x) {
  x.print("x argument inside f()");
  return x;
}

int main() {
  HowMany h;
  HowMany::print("after construction of h");
  HowMany h2 = f(h);
  HowMany::print("after call to f()");
} ///:~
```

La classe **HowMany** contiene uno **static int objectCount** e una funzione membro **static print( ),** per riportare il valore di **objectCount**, con un argomento opzionale per stampare un messaggio. Il costruttore incrementa il contatore ogni volta che viene creato un oggetto, e il distruttore lo decrementa.

Il risultato, tuttavia, non è quello che ci si aspetta:

```
after construction of h: objectCount = 1
x argument inside f(): objectCount = 1
~HowMany(): objectCount = 0
after call to f(): objectCount = 0
~HowMany(): objectCount = -1
~HowMany(): objectCount = -2
```

Dopo che è stato creato **h**, il contatore di oggetti vale 1, il che è giusto. Ma dopo la chiamata ad **f( )** ci si aspetta di avere il contatore di oggetti a due, perchè viene creato anche l'oggetto **h2**. Invece il contatore vale zero, il che indica che qualcosa è andata terribilmente storta. Questo viene confermato dal fatto che i due distruttori alla fine portano il contatore a valori negativi, cosa che non dovrebbe mai succedere.

Guardiamo dentro **f( )** cosa succede dopo che è stato passato l'argomento per valore. Il passaggio per valore significa che l'oggetto **h** esiste al di fuori della struttura della funzione, e che c'è un oggetto addizionale *all'interno* della struttura della funzione, che è la copia che è stata passata per valore. Tuttavia l'argomento è stato passato usando il concetto primitivo del C della copia per bit, mentre la classe **HowMany** del C++ richiede una vera inizializzazione per mantenere l'integrità, perciò la semplice copia per bit fallisce nella produzione dell'effetto desiderato.

Quando la copia locale dell'oggeto esce fuori scope alla fine della chiamata ad **f( )**, viene chiamato il distruttore, che decrementa **objectCount**, per cui al di fuori della funzione **objectCount** vale zero. Anche la creazione di **h2** viene fatta usando la copia per bit, per cui anche qui il costruttore non viene chiamato e quando **h** ed **h2** escono fuori scope i loro distruttori producono valori negativi per **objectCount**.

Costruzione della copia

Il problema sussiste perchè il compilatore fa un'assunzione su come viene creato *un nuovo oggetto a partire da uno già esistente*. Quando si passa un oggetto per valore si crea un nuovo oggetto, l'oggetto passato all'interno della funzione, da un oggetto esistente, cioè

l'oggetto originale al di fuori della struttura della funzione. Questo spesso è vero anche quando si ritorna un oggetto all'uscita di una funzione. Nell'espressione

```
HowMany h2 = f(h);
```

h2, oggetto non preventivamente costruito, viene creato dal valore di ritorno della funzione **f( )**, quindi di nuovo un oggetto viene creato a partire da uno esistente.

L'assunzione del compilatore è che si vuole fare una creazione usando la copia per bit, e questo in molti casi funziona perfettamente, ma in **HowMany** non funziona, in quanto il significato dell'inizializzazione va al di là della semplice copia per bit. Un altro esempio comune è quello delle classi che contengono puntatori – a cosa devono puntare? Devono essere copiati? Devono essere associati a qualche altro pezzo di memoria?

Fortunatamente si può intervenire in questo processo ed evitare che il compilatore faccia una copia per bit. Questo si fa definendo la propria funzione da usare ogni volta che è necessario creare un nuovo oggetto a partire da uno esistente. Logicamente, siccome si costruisce un nuovo oggetto, questa funzione è un costruttore, e altrettanto logicamente, l'unico argomento di questo costruttore deve essere l'oggetto da cui si sta effettuando la costruzione. Ma questo oggetto non può essere passato al costruttore per valore, in quanto stiamo cercando di *definire* la funzione che gestisce il passaggio per valore, e sintatticamente non ha senso passare un puntatore, perchè, dopotutto, stiamo creando un nuovo oggetto da quello esistente. Qui i riferimenti ci vengono in soccorso, quindi prendiamo il riferimento all'oggetto sorgente. Questa funzione è chiamata *costruttore di copia* ed è spesso chiamata **X(X&)**, se la classe è **X**.

Se si crea un costruttore di copia, il compilatore non effettua una copia per bit quando si crea un oggetto a partire da uno esistente. Esso chiamerà sempre il costruttore di copia. Se invece non si fornisce il costruttore di copia, il compilatore effettua comunque una copia, ma con il costruttore di copia abbiamo la possibilità di avere un controllo completo sul processo.

Adesso è possibile risolvere il problema riscontrato in **HowMany.cpp:**

```
//: C11:HowMany2.cpp
// Il costruttore di copia
#include <fstream>
#include <string>
using namespace std;
ofstream out("HowMany2.out");

class HowMany2 {
  string name; // Identificatore dell'oggetto
  static int objectCount;
public:
  HowMany2(const string& id = "") : name(id) {
    ++objectCount;
    print("HowMany2()");
  }
  ~HowMany2() {
    --objectCount;
    print("~HowMany2()");
  }
  // Il costruttore di copia:
  HowMany2(const HowMany2& h) : name(h.name) {
    name += " copy";
```

```
    ++objectCount;
    print("HowMany2(const HowMany2&)");
  }
  void print(const string& msg = "") const {
    if(msg.size() != 0)
      out << msg << endl;
    out << '\t' << name << ": "
        << "objectCount = "
        << objectCount << endl;
  }
};

int HowMany2::objectCount = 0;

// Passaggio e ritorno PER VALORE:
HowMany2 f(HowMany2 x) {
  x.print("x argument inside f()");
  out << "Returning from f()" << endl;
  return x;
}

int main() {
  HowMany2 h("h");
  out << "Entering f()" << endl;
  HowMany2 h2 = f(h);
  h2.print("h2 after call to f()");
  out << "Call f(), no return value" << endl;
  f(h);
  out << "After call to f()" << endl;
} ///:~
```

Ci sono un pò di modifiche in questo codice , in modo che si possa avere un'idea più chiara di cosa succede. Prima di tutto , la **string**a **name** agisce da identificatore quando vengono stampate informazioni relative all'oggetto. Nel costruttore, si può mettere una stringa di identificazione (generalmente il nome dell'oggetto) copiato in **name** usando il costruttore di **string**. Il default = **""** crea una **string**a vuota. Il costruttore incrementa **ObjectCount** come prima, e il distruttore lo decrementa.

Poi c'è il costruttore di copia, **HowMany2(const HowMany2&)**. Il costruttore di copia può creare un oggetto solo da uno già esistente, così il nome dell'oggetto esistente viene copiato in **name**, seguito dalla parola "copia" così si può vedere da dove viene. Se si guarda meglio, si può vedere che la chiamata **name(h.name)** nella lista di inizializzazione del costruttore chiama il costruttore di copia di **string**.

Nel costruttore di copia, il contatore di oggetti viene incrementato come nel normale costruttore. In questo modo si può ottenere un conteggio accurato degli oggetti quando c'è un passaggio e un ritorno per valore.

La funzione **print( )** è stata modificata per stampare un messaggio, l'identificatore dell'oggetto e il contatore degli oggetti. Esso deve accedere ora al dato **name** di un particolare oggetto, perciò non può essere una funzione membro **static**.

All'interno del **main( )**, si può vedere che è stata aggiunta una seconda chiamata a **f( )**. Tuttavia, questa seconda chiamata usa il comune approccio del C di ignorare il valore di ritorno. Ma adesso che sappiamo come viene restituito il valore (cioè, il codice *all'interno* della funzione gestisce il processo di ritorno, mettendo il risultato in un'area di destinazione il cui indirizzo viene passato come argomento nascosto), ci si potrebbe

chiedere cosa succede quando il valore di ritorno viene ignorato. Il risultato del programma potrebbe fornire qualche delucidazione su questo.

Prima di mostrare il risultato, qui c'è un piccolo programma che usa iostreams per aggiungere il numero di linea ad un file:

```
//: C11:Linenum.cpp
//{T} Linenum.cpp
// Aggiunge i numeri di linea
#include "../require.h"
#include <vector>
#include <string>
#include <fstream>
#include <iostream>
#include <cmath>
using namespace std;

int main(int argc, char* argv[]) {
  requireArgs(argc, 1, "Usage: linenum file\n"
    "Adds line numbers to file");
  ifstream in(argv[1]);
  assure(in, argv[1]);
  string line;
  vector<string> lines;
  while(getline(in, line)) // Legge un intero file nella stringa line
    lines.push_back(line);
  if(lines.size() == 0) return 0;
  int num = 0;
  // Il numero di linee nel file determina la larghezza:
  const int width =
    int(log10((double)lines.size())) + 1;
  for(int i = 0; i < lines.size(); i++) {
    cout.setf(ios::right, ios::adjustfield);
    cout.width(width);
    cout << ++num << ") " << lines[i] << endl;
  }
} ///:~
```

L'intero file viene letto dentro **vector<string>**, usando lo stesso codice già visto precedentemente in questo libro. Quando si stampano i numeri di linea, si vorrebbero avere tutte le linee allineate l'una all'altra, e questo richiede di aggiustare i numeri di linea nel file in modo che la larghezza permessa per i numeri di linea sia coerente. Possiamo facilmente calcolare il numero di una linea usando **vector::size( )**, ma quello di cui abbiamo veramente bisogno è sapere se ci sono più di 10 linee, 100 linee, 1,000 linee, ecc. Se prendiamo il logaritmo, base 10, del numero delle linee nel file, lo tronchiamo ad un **int** ed aggiungiamo uno al valore, calcoliamo la larghezza massima che deve assumere il contatore di linee.

Possiamo notare una coppia di strane chiamate all'interno del ciclo **for**: **setf( )** e **width( )**. Queste sono chiamate **ostream** che permettono di controllare, in questo caso, la giustificazione e la larghezza dell'output. Ma queste devono essere chiamate ogni volta che viene stampata una linea ed è per questo che sono messe dentro un ciclo **for**. Il volume 2 di questo libro contiene un capitolo intero che spiega gli iostreams e dice molte più cose riguardo a queste chiamate, come pure su altri modi di controllare iostreams.

Quando **Linenum.cpp** viene applicato a **HowMany2.out**, il risultato è

```
 1) HowMany2()
 2)   h: objectCount = 1
 3) Entering f()
 4) HowMany2(const HowMany2&)
 5)   h copy: objectCount = 2
 6) x argument inside f()
 7)   h copy: objectCount = 2
 8) Returning from f()
 9) HowMany2(const HowMany2&)
10)   h copy copy: objectCount = 3
11) ~HowMany2()
12)   h copy: objectCount = 2
13) h2 after call to f()
14)   h copy copy: objectCount = 2
15) Call f(), no return value
16) HowMany2(const HowMany2&)
17)   h copy: objectCount = 3
18) x argument inside f()
19)   h copy: objectCount = 3
20) Returning from f()
21) HowMany2(const HowMany2&)
22)   h copy copy: objectCount = 4
23) ~HowMany2()
24)   h copy: objectCount = 3
25) ~HowMany2()
26)   h copy copy: objectCount = 2
27) After call to f()
28) ~HowMany2()
29)   h copy copy: objectCount = 1
30) ~HowMany2()
31)   h: objectCount = 0
```

Come ci si potrebbe aspettare, la prima cosa che succede è che viene chiamato il costruttore normale per **h**, che incrementa il contatore di oggetti a uno. Ma dopo, quando si entra in **f( )**, il costruttore di copia viene chiamato in modo trasparente dal compilatore per effettuare il passaggio per valore. Viene creato un nuovo oggetto, che è la copia di **h** (da cui il nome "h copy") all'interno della struttura di **f( )**, così il contatore di oggetti diventa due, grazie al costruttore di copia.

La linea otto indica l'inizio del ritorno da **f( )**. Ma prima che la variabile locale "h copy" possa essere distrutta (essa esce fuori scope alla fine della funzione), deve essere copiata nel valore di ritorno, che è **h2**. L'oggetto (**h2**), non ancora costruito, viene creato a partire da un oggetto già esistente (la variabile locale dentro **f( )**), perciò il costruttore di copia viene di nuovo usato alla linea nove. Adesso il nome dell'identificatore di **h2** diventa "h copy copy" , in quanto esso viene copiato dalla copia locale ad **f( )**. Dopo che l'oggetto è stato restituito, ma prima di uscire dalla funzione, il contatore di oggetti diventa temporaneamente tre, ma dopo l'oggetto locale "h copy" viene distrutto. Dopo che la chiamata ad **f( )** è stata completata, alla linea 13, ci sono solo due oggetti, **h** e **h2**, e si può vedere che **h2** finisce per essere la "copia della copia di h."

Oggetti temporanei

La linea 15 inizia la chiamata ad **f(h)**, ignorando questa volta il valore di ritorno. Si può vedere alla linea 16 che il costruttore di copia viene chiamato esattamente come prima per passare l'argomento. E, come prima, la linea 21 mostra come il costruttore di copia viene chiamato per il valore di ritorno. Ma il costruttore di copia deve avere un indirizzo su cui lavorare come sua destinazione (un puntatore **this**). Da dove viene questo indirizzo?

Succede che il compilatore può creare un oggetto temporaneo ogni qualvolta ne ha bisogno per valutare opportunamente un'espressione. In questo caso ne crea uno che noi non vediamo che funge da destinazione per il valore di ritorno, ignorato, di **f( )**. La durata di questo oggetto temporaneo è la più breve possibile in modo tale che non ci siano in giro troppi oggetti temporanei in attesa di essere distrutti e che impegnano risorse preziose. In alcuni casi l'oggetto temporaneo può essere passato immediatamente ad un'altra funzione, ma in questo caso esso non è necessario dopo la chiamata alla funzione e quindi non appena la funzione termina, con la chiamata al distruttore dell'oggetto locale (linee 23 e 24), l'oggetto temporaneo viene distrutto (linee 25 e 26).

Infine, alle linee 28-31, l'oggetto **h2** viene distrutto, seguito da **h**, e il contatore di oggetti torna correttamente a zero.

Costruttore di copia di default

Siccome il costruttore di copia implementa il passaggio e il ritorno per valore, è importante che il compilatore ne crei uno di default nel caso di semplici strutture – che poi è la stessa cosa che fa in C. Tuttavia, tutto quello che abbiamo visto finora è il comportamento di default primitivo: una copia per bit.

Quando sono coinvolti tipi più complessi, il compilatore C++ deve comunque creare automaticamente un costruttore di copia di default. Ma, di nuovo, un copia per bit non ha senso, perchè potrebbe non implementare il significato corretto.

Qui c'è un esempio che mostra un approccio più intelligente che può avere il compilatore. Supponiamo di creare una nuova classe composta di oggetti di diverse classi già esistenti. Questo viene chiamato, in modo abbastanza appropriato, *composizione*, ed è uno dei modi per costruire nuove classi a partire da classi già esistenti. Adesso mettiamoci nei panni di un utente inesperto che vuole provare a risolvere velocemente un problema creando una nuova classe in questo modo. Non sappiamo nulla riguardo al costruttore di copia, pertanto non lo creiamo. L'esempio mostra cosa fa il compilatore mentre crea un costruttore di copia di default per questa nostra nuova classe:

```
//: C11:DefaultCopyConstructor.cpp
// Creazione automatica del costruttore di copia
#include <iostream>
#include <string>
using namespace std;

class WithCC { // Con costruttore di copia
public:
  // Richiesto il costruttore di default esplicito:
  WithCC() {}
  WithCC(const WithCC&) {
    cout << "WithCC(WithCC&)" << endl;
  }
};

class WoCC { // Senza costruttore di copia
  string id;
public:
  WoCC(const string& ident = "") : id(ident) {}
  void print(const string& msg = "") const {
    if(msg.size() != 0) cout << msg << ": ";
    cout << id << endl;
```

```
  }
};

class Composite {
  WithCC withcc; // Oggetti incorporati
  WoCC wocc;
public:
  Composite() : wocc("Composite()") {}
  void print(const string& msg = "") const {
    wocc.print(msg);
  }
};

int main() {
  Composite c;
  c.print("Contents of c");
  cout << "Calling Composite copy-constructor"
       << endl;
  Composite c2 = c;  // Chiama il costruttore di copia
  c2.print("Contents of c2");
} ///:~
```

La classe **WithCC** contiene un costruttore di copia, che annuncia semplicemente che è stato chiamato, e questo solleva una questione interessante. Nella classe **Composite**, viene creato un oggetto di tipo **WithCC** usando un costruttore di default. Se non ci fosse stato per niente il costruttore nella classe **WithCC**, il compilatore ne avrebbe creato automaticamente uno di default, che in questo caso non avrebbe fatto nulla. Ma se aggiungiamo un costruttore di copia, diciamo al compilatore che ci accingiamo a gestire la creazione del costruttore ed esso non ne creerà uno per noi e darà errore, a meno che non gli diciamo esplicitamente di crearne uno di default, come è stato fatto per **WithCC**.

La classe **WoCC** non ha un costruttore di copia, ma memorizza un messaggio in una **string**a interna che può essere stampata con **print( )**. Questo costruttore viene esplicitamente chiamato nella lista di inizializzatori del costruttore di **Composite** (brevemente introdotta nel Capitolo 8 e completamente coperta nel Capitolo 14). La ragione di ciò sarà chiara più avanti.

La classe **Composite** ha oggetti membro sia di tipo **WithCC** che **WoCC** (notare che l'oggetto incorporato **WoCC** viene inizializzato nella lista di inizializzatori del costruttore, come deve essere), e non ha un costruttore di copia esplicitamente definito. Tuttavia, in **main( )** viene creato un oggetto usando il costruttore di copia nella definizione:

```
Composite c2 = c;
```

Il costruttore di copia di **Composite** viene creato automaticamente dal compilatore, e l'output del programma rivela il modo in cui ciò avviene:

```
Contents of c: Composite()
Calling Composite copy-constructor
WithCC(WithCC&)
Contents of c2: Composite()
```

Per creare un costruttore di copia una classe che usa la composizione (e l'ereditarietà, che è introdotta nel Capitolo 14), il compilatore chiama ricorsivamente i costruttori di copia per tutti gli oggetti membri e per le classi base. Cioè, se l'oggetto membro contiene un altro oggetto, anche il suo costruttore di copia viene chiamato. Così, in questo caso il

compilatore chiama il costruttore di copia per **WithCC**. L'output mostra che questo costruttore viene chiamato. Siccome **WoCC** non ha un costruttore di copia, il compilatore ne crea uno che effettua semplicemente la copia per bit, e lo chiama all'interno del costruttore di copia di **Composite**. La chiamata a **Composite::print( )** in main mostra che questo succede in quanto i contenuti di **c2.WoCC** sono identici ai contenuti di **c.WoCC**. Il processo che il compilatore mette in atto per sintetizzare un costruttore di copia è detto *inizializzazione per membro (memberwise initialization)* .

È sempre meglio creare il proprio costruttore di copia, invece di farlo fare al compilatore. Questo garantisce che starà sotto il nostro controllo.

Alternative alla costruzione della copia

A questo punto forse vi gira un pò la testa e vi state meravigliando di come sia stato possibile scrivere finora codice funzionante senza sapere nulla riguardo al costruttore di copia. Ma ricordate: abbiamo bisogno di un costruttore di copia solo se dobbiamo passare un oggetto *per valore*. Se questo non succede, non abbiamo bisogno di un costruttore di copia.

Prevenire il passaggio-per-valore

"Ma," potreste dire, "se non forniamo un costruttore di copia, il compilatore ne creerà uno per noi. E come facciamo a sapere che un oggetto non sarà mai passato per valore?"

C'è una tecnica molto semplice per prevenire il passaggio per valore: dichiarare un costruttore di copia **private**. Non c'è bisogno di fornire una definizione, a meno che una delle funzioni membro o una funzione **friend** non abbiano bisogno di effettuare un passaggio per valore. Se l'utente prova a passare o a ritornare un oggetto per valore, il compilatore da un messaggio di errore, in quanto il costruttore di copia è **private**. Inoltre esso non creerà un costruttore di copia di default, in quanto gli abbiamo detto esplicitamente che abbiamo noi il controlo su questo.

Qui c'è un esempio:

```
//: C11:NoCopyConstruction.cpp
// Prevenire la costruzione della copia

class NoCC {
  int i;
  NoCC(const NoCC&); // Nessuna definizione
public:
  NoCC(int ii = 0) : i(ii) {}
};

void f(NoCC);

int main() {
  NoCC n;
//! f(n); // Errore: chiamato il costruttore di copia
//! NoCC n2 = n; // Errore: chiamato c-c
//! NoCC n3(n); // Errore: chiamato c-c
} ///:~
```

Notare l'uso della forma molto più generale

```
NoCC(const NoCC&);
```

che fa uso di **const**.

Funzioni che modificano oggetti esterni

La sintassi dei riferimenti è più accurata di quella dei puntatori, ma nasconde il significato al lettore. Per esempio, nella libreria iostreams una versione overloaded della funzione **get( )** prende come argomento un **char&**, e lo scopo della funzione è proprio quello di modificare l'argomento per inserire il risultato della **get( )**. Ma quando si legge del codice che usa questa funzione, non è immediatamente ovvio che l'oggetto esterno viene modificato:

```
char c;
cin.get(c);
```

In effetti la chiamata alla funzione fa pensare ad un passaggio per valore e quindi suggerisce che l'oggetto esterno *non* viene modificato.

Per questo motivo, probabilmente è più sicuro da un punto di vista della manutenzione del codice usare i puntatori quando si deve passare l'indirizzo di un argomento che deve essere modificato. Se un indirizzo lo si passa *sempre* come riferimento **const** *eccetto* quando si intende modificare l'oggetto esterno attraverso l'indirizzo, caso in cui si usa un puntatore non-**const**, allora il codice diventa molto più facile da seguire per un lettore.

Puntatori a membri

Un puntatore è una variabile che memorizza l'indirizzo di qualche locazione di memoria. Si può cambiare a runtime quello che il puntatore seleziona e la destinazione di un puntatore può essere sia un dato sia una funzione. Il *puntatore-a-membro* del C++ segue lo stesso principio, solo che seleziona una locazione all'interno di una classe. Il dilemma qui è che il puntatore ha bisogno di un indirizzo, ma non ci sono "indirizzi" dentro una classe; selezionare un membro di una classe significa prenderne l'offset all'interno della classe. Non si può produrre un indirizzo prima di comporre tale offset con l'indirizzo di inizio di un particolare oggetto. La sintassi dei puntatori a membri richiede di selezionare un oggetto nel momento stesso in cui viene dereferenziato un puntatore a membro.

Per capire questa sintassi, consideriamo una semplice struttura, con un puntatore **sp** e un oggetto **so** per questa struttura. Si possono selezionare membri con la sintassi mostrata:

```
//: C11:SimpleStructure.cpp
struct Simple { int a; };
int main() {
  Simple so, *sp = &so;
  sp->a;
  so.a;
} ///:~
```

Adesso supponiamo di avere un puntatore ordinario a un intero, **ip**. Per accedere al valore puntato da **ip** si dereferenzia il puntatore con '*****':

```
*ip = 4;
```

Infine consideriamo cosa succede se abbiamo un puntatore che punta a qualcosa all'interno di un oggetto di una classe, anche se di fatto esso rappresenta un offset all'interno dell'oggetto. Per accedere a ciò a cui punta bisogna dereferenziarlo con *. Ma si tratta di un offset all'interno dell'oggetto, e quindi bisogna riferirsi anche a quel particolare oggetto. Così l' * viene combinato con la dereferenziazione dell'oggetto. Così la nuova sintassi diventa **->*** per un puntatore a un oggetto e **.*** per un oggetto o un riferimento, come questo:

```
objectPointer->*pointerToMember = 47;
object.*pointerToMember = 47;
```

Ora qual'è la sintassi per definire **pointerToMember**? Come qualsiasi puntatore, bisogna dire a quale tipo punta e usare un * nella definizione. L'unica differenza è che bisogna dire con quale classe di oggetti questo puntatore-a-membro deve essere usato. Naturalmente, questo si ottiene con il nome della classe e l'operatore di risoluzione di scope.

```
int ObjectClass::*pointerToMember;
```

definisce una variabile puntatore-a-membro chiamata **pointerToMember** che punta a un **int** all'interno di **ObjectClass**. Si può anche inizializzare un puntatore-a-membro quando lo si definisce (o in qualunque altro momento):

```
int ObjectClass::*pointerToMember = &ObjectClass::a;
```

Non c'è un "indirizzo" di **ObjectClass::a** perchè ci stiamo riferendo ad una classe e non ad un oggetto della classe. Così, **&ObjectClass::a** può essere usato solo come sintassi di un puntatore-a-membro.

Qui c'è un esempio che mostra come creare e usare i puntatori a dati membri:

```
//: C11:PointerToMemberData.cpp
#include <iostream>
using namespace std;

class Data {
public:
  int a, b, c;
  void print() const {
    cout << "a = " << a << ", b = " << b
         << ", c = " << c << endl;
  }
};

int main() {
  Data d, *dp = &d;
  int Data::*pmInt = &Data::a;
  dp->*pmInt = 47;
  pmInt = &Data::b;
  d.*pmInt = 48;
  pmInt = &Data::c;
  dp->*pmInt = 49;
  dp->print();
} ///:~
```

Ovviamente non è il caso di usare i puntatori-a-membri dappertutto, se non in casi particolari (che è il motivo esatto per cui sono stati introdotti).

Inoltre essi sono molto limitati: possono essere assegnati solo a locazioni specifiche all'interno di una classe. Non si possono, ad esempio, incrementare o confrontare come i normali puntatori.

Funzioni

In modo simile si ottiene la sintassi di un puntatore-a-membro per le funzioni membro. Un puntatore a funzione (introdotto nel Capitolo 3) è definito come questo:

```
int (*fp)(float);
```

Le parentesi intorno a **(*fp)** sono necessarie affinchè il compilatore valuti in modo appropriato la definizione. Senza di esse potrebbe sembrare la definizione di una funzione che restituisce un **int***.

Le parentesi giocano un ruolo importante anche nella definizione e nell'uso dei punatori a funzioni membro. Se abbiamo una funzione in una classe, possiamo definire un puntatore ad essa usando il nome della classe e l'operatore di risoluzione di scope all'interno di una definizione ordinaria di puntatore a funzione:

```
//: C11:PmemFunDefinition.cpp
class Simple2 {
public:
  int f(float) const { return 1; }
};
int (Simple2::*fp)(float) const;
int (Simple2::*fp2)(float) const = &Simple2::f;
int main() {
  fp = &Simple2::f;
} ///:~
```

Nella definizione di **fp2** si può notare che un puntatore ad una funzione membro può anche essere inizializzato quando viene creato, o in qualunque altro momento. A differenza delle funzioni non-membro, l'operatore **&** *non è* opzionale quando si prende l'indirizzo di una funzione membro. Tuttavia, si può fornire l'identificatore della funzione senza la lista degli argomenti, perchè la risoluzione dell'overload può essere effettuata sulla base del tipo del puntatore a membro.

Un esempio

L'importanza di un puntatore sta nel fatto che si può cambiare runtime l'oggetto a cui punta, il che conferisce molta flessibiltà alla programmazione, perchè attraverso un puntatore si può selezionare o cambiare *comportamento* a runtime. Un puntatore-a-membro non fa eccezione; esso permette di scegliere un membro a runtime. Tipicamente, le classi hanno solo funzioni membro pubblicamente visibili (I dati membro sono generalmente considerati parte dell'implementazione sottostante), così l'esempio seguente seleziona funzioni membro a runtime.

```
//: C11:PointerToMemberFunction.cpp
#include <iostream>
using namespace std;
```

```
class Widget {
public:
  void f(int) const { cout << "Widget::f()\n"; }
  void g(int) const { cout << "Widget::g()\n"; }
  void h(int) const { cout << "Widget::h()\n"; }
  void i(int) const { cout << "Widget::i()\n"; }
};

int main() {
  Widget w;
  Widget* wp = &w;
  void (Widget::*pmem)(int) const = &Widget::h;
  (w.*pmem)(1);
  (wp->*pmem)(2);
} ///:~
```

Naturalmente non è particolarmente ragionevole aspettarsi che un utente qualsiasi possa creare espressioni così complicate. Se un utente deve manipolare direttamente un puntatore-a-membro, allora un **typedef** è quello che ci vuole. Per rendere le cose veramente pulite si possono usare i puntatori-a-membri come parte del meccanismo di implementazione interna. Qui c'è l'esempio precedente che usa un puntatore-a-membro *all'interno* della classe. Tutto quello che l'utente deve fare è passare un numero per selezionare una funzione.[48]

```
//: C11:PointerToMemberFunction2.cpp
#include <iostream>
using namespace std;

class Widget {
  void f(int) const { cout << "Widget::f()\n"; }
  void g(int) const { cout << "Widget::g()\n"; }
  void h(int) const { cout << "Widget::h()\n"; }
  void i(int) const { cout << "Widget::i()\n"; }
  enum { cnt = 4 };
  void (Widget::*fptr[cnt])(int) const;
public:
  Widget() {
    fptr[0] = &Widget::f; // Richiesta specificazione completa
    fptr[1] = &Widget::g;
    fptr[2] = &Widget::h;
    fptr[3] = &Widget::i;
  }
  void select(int i, int j) {
    if(i < 0 || i >= cnt) return;
    (this->*fptr[i])(j);
  }
  int count() { return cnt; }
};

int main() {
  Widget w;
  for(int i = 0; i < w.count(); i++)
    w.select(i, 47);
} ///:~
```

Nell'interfaccia della classe e nel **main( )**, si può vedere che l'intera implementazione, comprese le funzioni, è stata nascosta. Il codice deve sempre invocare **count( )** per

selezionare una funzione. In questo modo l'implementatore della classe può cambiare il numero di funzioni nell'implementazione sottostante, senza influire sul codice in cui la classe viene usata.

L'inizializzazione dei puntatori-a-membri nel costruttore può sembrare sovraspecificato. Non si potrebbe semplicemente scrivere

```
fptr[1] = &g;
```

visto che il nome **g** appare nella funzione membro, che è automaticamente nello scope della classe? Il problema è che questo non è conforme alla sintassi del puntatore-a-membro, che invece è richiesta, così tutti, specialmente il compilatore, possono sapere cosa si sta facendo. Allo stesso modo, quando un puntatore-a-membro viene dereferenziato, assomiglia a

```
(this->*fptr[i])(j);
```

che pure è sovraspecificato; **this** sembra ridondante. Di nuovo, la sintassi richiede che un puntatore-a-membro sia sempre legato a un oggetto quando viene dereferenziato.

Sommario

I puntatori in C++ sono quasi identici ai puntatori in C, il che è buono. Altrimenti un sacco di codice scritto in C non si compilerebbe in C++. Gli unici errori a compile-time che vengono fuori sono legati ad assegnamenti pericolosi. Se proprio si vuole cambiare il tipo, bisogna usare esplicitamente il cast.

Il C++ aggiunge anche i *riferimenti,* presi dall'Algol e dal Pascal, che sono come dei puntatori costanti automaticamente dereferenziati dal compilatore. Un riferimento contiene un indirizzo, ma lo si può trattare come un oggetto. I riferimenti sono essenziali per una sintassi chiara con l'overload degli operatori (l'argomento del prossimo capitolo), ma essi aggiungono anche vantaggi sintattici nel passare e ritornare oggetti per funzioni ordinarie.

Il costruttore di copia prende un riferimento ad un oggetto esistente dello stesso tipo come argomento e viene usato per creare un nuovo oggetto da uno esistente. Il compilatore automaticamente chiama il costruttore di copia quando viene passato o ritornato un oggetto per valore. Anche se il compilatore è in grado di creare un costruttore di copia di default, se si pensa che ne serve uno per la nostra classe è meglio definirlo sempre in proprio per assicurare il corretto funzionamento. Se non si vuole che gli oggetti siano passati o ritornati per valore, si può creare un costruttore di copia private.

I puntatori-a-membri hanno le stesse funzionalità dei puntatori ordinari: si può scegliere una particolare zona di memoria (dati o funzioni) a runtime. I puntatori-a-membri semplicemente funzionano con i membri di classi invece che con dati globali o funzioni. Si ottiene la flessibilità di programmazione che permette di cambiare il comportamento a runtime.

---

[48] Grazie a Mortensen per questo esempio

12: Sovraccaricamento degli operatori

Il sovraccaricamento degli operatori è semplicemente "uno zuccherino sintattico," che vuol dire un altro modo di chiamare le funzioni.

La differenza è che gli argomenti di questa funzione non appaiono dentro le parentesi, ma piuttosto circondano o seguono caratteri che abbiamo sempre pensato come immutabili operatori.

Ci sono due differenze tra l'uso di un operatore e una chiamata ad una funzione ordinaria. La sintassi è differente; un operatore spesso è "chiamato" mettendolo tra gli argomenti o a volte dopo gli argomenti. La seconda differenza è che il compilatore stabilisce quale "funzione" chiamare. Per esempio, se usiamo l'operatore + con argomenti floating-point, il compilatore "chiama" la funzione per sommare due floating-point (questa "chiamata" è tipicamente l'atto di mettere del codice in-line, o un'istruzione di un processore floating-point). Se usiamo l'operatore + con un numero floating-point e un int, il compilatore "chiama" una funzione speciale per trasformare l' **int** in **float**, e poi "chiama" il codice per sommare due floating-point.

Ma in C++ è possibile definire nuovi operatori che lavorano con le classi. Questa definizione è proprio come la definizione di una funzione ordinaria, solo che il nome della funzione consiste della parola chiave **operator** seguita dall'operatore. Questa è la sola differenza e diventa una funzione come tutte le altre, che il compilatore chiama quando vede il tipo appropriato.

Avvertenze & rassicurazioni

Si è portati ad essere super entusiasti con il sovraccaricamento degli operatori. È un giocattolo divertente, sulle prime. Ma ricordiamoci che è *solo* uno zuccherino sintattico, un altro modo per chiamare una funzione. Guardando la cosa sotto questo aspetto, non abbiamo nessuna ragione di sovraccaricare un operatore, se non ci permette di rendere il codice che coinvolge la nostra classe più facile da scrivere e soprattutto più facile da *leggere* (notare che si legge molto più codice di quanto non se ne scriva). Se questo non è il caso, non ci dobbiamo preoccupare.

Un'altra reazione diffusa al sovraccaricamento degli operatori è il panico; improvvisamente gli operatori C non hanno più un significato familiare. "Ogni cosa è cambiata e il mio codice C farà cose diverse!" Questo non è vero. Tutti gli operatori che contengono solo dati di tipo predefinito non possono cambiare. Non potremo mai sovraccaricare operatori del tipo

```
1 << 4;
```

oppure

```
1.414 << 2;
```

Solo espressioni che contengono tipi definiti dall'utente possono avere un operatore sovraccaricato.

Sintassi

Definire un operatore sovraccaricato è come definire una funzione, ma il nome della funzione è **operator@**, dove @ rappresenta l'operatore da sovraccaricare. Il numero di argomenti nella lista degli argomenti dell'operatore sovraccaricato dipende da due fattori:

1. Se è un operatore unario (un solo argomento) o binario (due argomenti).
2. Se l'operatore è definito come funzione globale (un argomento se unario, due se binario) o come funzione membro (zero argomenti se unario, uno se binario – l'oggetto diventa l'argomento a sinistra dell'operatore).

Qui c'è una piccola classe che mostra la sintassi per il sovraccaricamento di un operatore :

```
//: C12:OperatorOverloadingSyntax.cpp
#include <iostream>
using namespace std;

class Integer {
  int i;
public:
  Integer(int ii) : i(ii) {}
  const Integer
  operator+(const Integer& rv) const {
    cout << "operator+" << endl;
    return Integer(i + rv.i);
  }
  Integer&
  operator+=(const Integer& rv) {
    cout << "operator+=" << endl;
    i += rv.i;
    return *this;
  }
};

int main() {
  cout << "built-in types:" << endl;
  int i = 1, j = 2, k = 3;
  k += i + j;
  cout << "user-defined types:" << endl;
  Integer ii(1), jj(2), kk(3);
  kk += ii + jj;
} ///:~
```

I due operatori sovraccaricati sono definiti come funzioni membro inline che si annunciano quando vengono chiamate. Il singolo argomento è quello che appare alla destra dell'operatore per operatori binari. Gli operatori unari non hanno argomenti quando definiti come funzioni membro. La funzione membro viene chiamata per l'oggetto alla sinistra dell'operatore.

Per operatori non-condizionali (gli operatori condizionali restituiscono usualmente valori Boolean), quasi sempre il valore di ritorno è un oggetto o un riferimento dello stesso tipo su cui si sta operando, se i due argomenti sono dello stesso tipo (se non sono dello stesso tipo, l'interpretazione di quello che potrebbe restituire sta a noi). In questo modo possono essere costruite espressioni complicate, come:

```
kk += ii + jj;
```

L' **operator+** produce un nuovo **Integer** (temporaneo) che viene usato come argomento **rv** per **operator+=**. Questo dato temporaneo viene distrutto quando non è più necessario.

Operatori sovraccaricabili

Benchè sia possibile sovraccaricare quasi tutti gli operatori disponibili in C, l'uso del sovraccaricamento degli operatori è abbastanza restrittivo. In particolare, non si possono combinare operatori che non hanno nessun significato in C (come ** per rappresentare l'elevamento a potenza), non si può cambiare la precedenza degli operatori, e non si può cambiare il numero di argomenti richiesti per un operatore. Questo ha senso – tutte queste azioni produrrebbero operatori che confondono il significato, invece di chiarirlo.

Le prossime due sottosezioni mostrano esempi di tutti gli operatori "regolari", sovraccaricati nella forma che più verosimilmente viene usata.

Operatori unari

L'esempio seguente mostra la sintassi per sovraccaricare tutti gli operatori unari, sia nella forma di funzioni globali (non **friend** di funzioni membro) che di funzioni membro. Questo (esempio) va dalla classe **Integer** mostrata precedentemente fino ad una nuova classe **byte**. Il significato degli operatori specifici dipende dall'uso che ne vogliamo fare, ma teniamo presente il cliente programmatore prima di fare cose inaspettate.

Qui c'è un catalogo di tutte le funzioni unarie:

```
//: C12:OverloadingUnaryOperators.cpp
#include <iostream>
using namespace std;

// Funzioni non-membro:
class Integer {
  long i;
  Integer* This() { return this; }
public:
  Integer(long ll = 0) : i(ll) {}
  //Operatori senza effetti collaterali accettano argomenti const&:
  friend const Integer&
    operator+(const Integer& a);
  friend const Integer
    operator-(const Integer& a);
  friend const Integer
    operator~(const Integer& a);
  friend Integer*
    operator&(Integer& a);
  friend int
    operator!(const Integer& a);
  // Operatori con effetti collaterali hanno argomenti non-const&:
  // Prefisso:
  friend const Integer&
    operator++(Integer& a);
  // Postfisso:
  friend const Integer
    operator++(Integer& a, int);
  // Prefisso:
  friend const Integer&
```

```
    operator--(Integer& a);
  // Postfisso:
  friend const Integer
    operator--(Integer& a, int);
};

// Operatori globali:
const Integer& operator+(const Integer& a) {
  cout << "+Integer\n";
  return a; //L'operatore + unario non ha effetto sull'argomento
}
const Integer operator-(const Integer& a) {
  cout << "-Integer\n";
  return Integer(-a.i);
}
const Integer operator~(const Integer& a) {
  cout << "~Integer\n";
  return Integer(~a.i);
}
Integer* operator&(Integer& a) {
  cout << "&Integer\n";
  return a.This(); // &a è ricorsivo!
}
int operator!(const Integer& a) {
  cout << "!Integer\n";
  return !a.i;
}
// Prefisso; restituisce il valore incrementato
const Integer& operator++(Integer& a) {
  cout << "++Integer\n";
  a.i++;
  return a;
}
// Postfisso; restituisce il valore prima di incrementarlo:
const Integer operator++(Integer& a, int) {
  cout << "Integer++\n";
  Integer before(a.i);
  a.i++;
  return before;
}
// Prefisso; restituisce il valore decrementato
const Integer& operator--(Integer& a) {
  cout << "--Integer\n";
  a.i--;
  return a;
}
// Postfisso; restituisce il valore prima di decrementarlo:
const Integer operator--(Integer& a, int) {
  cout << "Integer--\n";
  Integer before(a.i);
  a.i--;
  return before;
}

// Mostra che gli operatori sovraccaricati funzionano:
void f(Integer a) {
  +a;
  -a;
  ~a;
  Integer* ip = &a;
  !a;
  ++a;
  a++;
```

```
  --a;
  a--;
}

// Funzioni membro("this" implicito):
class Byte {
  unsigned char b;
public:
  Byte(unsigned char bb = 0) : b(bb) {}
  // Non ci sono effetti collaterali: funzione membro const:
  const Byte& operator+() const {
    cout << "+Byte\n";
    return *this;
  }
  const Byte operator-() const {
    cout << "-Byte\n";
    return Byte(-b);
  }
  const Byte operator~() const {
    cout << "~Byte\n";
    return Byte(~b);
  }
  Byte operator!() const {
    cout << "!Byte\n";
    return Byte(!b);
  }
  Byte* operator&() {
    cout << "&Byte\n";
    return this;
  }
  //Ci sono effetti collaterali: funzione membro non-const:
  const Byte& operator++() { // Prefisso
    cout << "++Byte\n";
    b++;
    return *this;
  }
  const Byte operator++(int) { // Postfisso
    cout << "Byte++\n";
    Byte before(b);
    b++;
    return before;
  }
  const Byte& operator--() { // Prefisso
    cout << "--Byte\n";
    --b;
    return *this;
  }
  const Byte operator--(int) { // Postfisso
    cout << "Byte--\n";
    Byte before(b);
    --b;
    return before;
  }
};

void g(Byte b) {
  +b;
  -b;
  ~b;
  Byte* bp = &b;
  !b;
  ++b;
  b++;
```

```
  --b;
  b--;
}

int main() {
  Integer a;
  f(a);
  Byte b;
  g(b);
} ///:~
```

Le funzioni sono raggruppate in base al modo in cui vengono passati i loro argomenti. Le linee guida su come passare e restituire argomenti saranno fornite in seguito. Le forme di sopra (e quelle che seguono nella prossima sezione) sono quelle tipicamente usate, così assumiamo queste come modello per il sovraccaricamento dei nostri operatori.

Incremento & decremento

Gli operatori sovraccaricati ++ e – – presentano un dilemma perchè vorremmo essere in grado di chiamare funzioni diverse a seconda che essi appaiano prima (prefisso) o dopo (postfisso) l'oggetto su cui operano. La soluzione è semplice, ma qualche volta la gente trova la cosa un pò fuorviante sulle prime. Quando il compilatore vede, per esempio, **++a** (pre-incremento), esso genera una chiamata a **operator++(a)**; ma quando vede **a++,** esso genera una chiamata a **operator++(a, int)**. Cioè il compilatore distingue le due forme effettuando chiamate a due diverse funzioni sovraccaricate. In **OverloadingUnaryOperators.cpp** per le versioni riferite alle funzioni membro, se il compilatore vede **++b**, genera una chiamata a **B::operator++( )**; se vede **b++** chiama **B::operator++(int)**.

Tutto quello che l'utente vede è che viene chiamata una funzione diversa per la versione con il prefisso e quella con il postfisso. In fondo le chiamate alle due funzioni hanno firme diverse, così esse si aggangiano a due corpi di funzione diversi. Il compilatore passa un valore costante fittizio per l'argomento **int** (il quale non ha mai un identificativo, in quanto il suo valore non è mai usato) per generare una firma diversa per la versione con postfisso.

Operatori binari

Il listato seguente ripete l'esempio di **OverloadingUnaryOperators.cpp** per gli operatori binari, così abbiamo un esempio per tutti gli operatori che potremmo voler sovraccaricare.

```
//: C12:Integer.h
// Operatori sovraccaricati non-membri
#ifndef INTEGER_H
#define INTEGER_H
#include <iostream>

// Funzioni non-membro:
class Integer {
  long i;
public:
  Integer(long ll = 0) : i(ll) {}
  // Operatori che creano valori nuovi, modificati:
  friend const Integer
    operator+(const Integer& left,
              const Integer& right);
```

```
friend const Integer
  operator-(const Integer& left,
            const Integer& right);
friend const Integer
  operator*(const Integer& left,
            const Integer& right);
friend const Integer
  operator/(const Integer& left,
            const Integer& right);
friend const Integer
  operator%(const Integer& left,
            const Integer& right);
friend const Integer
  operator^(const Integer& left,
            const Integer& right);
friend const Integer
  operator&(const Integer& left,
            const Integer& right);
friend const Integer
  operator|(const Integer& left,
            const Integer& right);
friend const Integer
  operator<<(const Integer& left,
             const Integer& right);
friend const Integer
  operator>>(const Integer& left,
             const Integer& right);
//Modifica per assegnamento & ritorno di lvalue:
friend Integer&
  operator+=(Integer& left,
             const Integer& right);
friend Integer&
  operator-=(Integer& left,
             const Integer& right);
friend Integer&
  operator*=(Integer& left,
             const Integer& right);
friend Integer&
  operator/=(Integer& left,
             const Integer& right);
friend Integer&
  operator%=(Integer& left,
             const Integer& right);
friend Integer&
  operator^=(Integer& left,
             const Integer& right);
friend Integer&
  operator&=(Integer& left,
             const Integer& right);
friend Integer&
  operator|=(Integer& left,
             const Integer& right);
friend Integer&
  operator>>=(Integer& left,
              const Integer& right);
friend Integer&
  operator<<=(Integer& left,
              const Integer& right);
// Gli operatori condizionali restituiscono true/false:
friend int
  operator==(const Integer& left,
             const Integer& right);
friend int
```

```
    operator!=(const Integer& left,
               const Integer& right);
  friend int
    operator<(const Integer& left,
              const Integer& right);
  friend int
    operator>(const Integer& left,
              const Integer& right);
  friend int
    operator<=(const Integer& left,
               const Integer& right);
  friend int
    operator>=(const Integer& left,
               const Integer& right);
  friend int
    operator&&(const Integer& left,
               const Integer& right);
  friend int
    operator||(const Integer& left,
               const Integer& right);
  // Scrive i contenuti su un ostream:
  void print(std::ostream& os) const { os << i; }
};
#endif // INTEGER_H ///:~
//: C12:Integer.cpp {O}
// Implementazione di operatori sovraccaricati
#include "Integer.h"
#include "../require.h"

const Integer
  operator+(const Integer& left,
            const Integer& right) {
  return Integer(left.i + right.i);
}
const Integer
  operator-(const Integer& left,
            const Integer& right) {
  return Integer(left.i - right.i);
}
const Integer
  operator*(const Integer& left,
            const Integer& right) {
  return Integer(left.i * right.i);
}
const Integer
  operator/(const Integer& left,
            const Integer& right) {
  require(right.i != 0, "divide by zero");
  return Integer(left.i / right.i);
}
const Integer
  operator%(const Integer& left,
            const Integer& right) {
  require(right.i != 0, "modulo by zero");
  return Integer(left.i % right.i);
}
const Integer
  operator^(const Integer& left,
            const Integer& right) {
  return Integer(left.i ^ right.i);
}
const Integer
  operator&(const Integer& left,
```

```
            const Integer& right) {
  return Integer(left.i & right.i);
}
const Integer
  operator|(const Integer& left,
            const Integer& right) {
  return Integer(left.i | right.i);
}
const Integer
  operator<<(const Integer& left,
             const Integer& right) {
  return Integer(left.i << right.i);
}
const Integer
  operator>>(const Integer& left,
             const Integer& right) {
  return Integer(left.i >> right.i);
}
// Modifica per assegnamento & ritorno di lvalue:
Integer& operator+=(Integer& left,
                    const Integer& right) {
   if(&left == &right) {/* auto-assegnamento */}
   left.i += right.i;
   return left;
}
Integer& operator-=(Integer& left,
                    const Integer& right) {
   if(&left == &right) {/* auto-assegnamento */}
   left.i -= right.i;
   return left;
}
Integer& operator*=(Integer& left,
                    const Integer& right) {
   if(&left == &right) {/* auto-assegnamento */}
   left.i *= right.i;
   return left;
}
Integer& operator/=(Integer& left,
                    const Integer& right) {
   require(right.i != 0, "divide by zero");
   if(&left == &right) {/* auto-assegnamento */}
   left.i /= right.i;
   return left;
}
Integer& operator%=(Integer& left,
                    const Integer& right) {
   require(right.i != 0, "modulo by zero");
   if(&left == &right) {/* auto-assegnamento */}
   left.i %= right.i;
   return left;
}
Integer& operator^=(Integer& left,
                    const Integer& right) {
   if(&left == &right) {/* auto-assegnamento */}
   left.i ^= right.i;
   return left;
}
Integer& operator&=(Integer& left,
                    const Integer& right) {
   if(&left == &right) {/* auto-assegnamento */}
   left.i &= right.i;
   return left;
}
```

```
Integer& operator|=(Integer& left,
                    const Integer& right) {
   if(&left == &right) {/* auto-assegnamento */}
   left.i |= right.i;
   return left;
}
Integer& operator>>=(Integer& left,
                     const Integer& right) {
   if(&left == &right) {/* auto-assegnamento */}
   left.i >>= right.i;
   return left;
}
Integer& operator<<=(Integer& left,
                     const Integer& right) {
   if(&left == &right) {/* auto-assegnamento */}
   left.i <<= right.i;
   return left;
}
// Gli operatori condizionali restituiscono true/false:
int operator==(const Integer& left,
               const Integer& right) {
    return left.i == right.i;
}
int operator!=(const Integer& left,
               const Integer& right) {
    return left.i != right.i;
}
int operator<(const Integer& left,
              const Integer& right) {
    return left.i < right.i;
}
int operator>(const Integer& left,
              const Integer& right) {
    return left.i > right.i;
}
int operator<=(const Integer& left,
               const Integer& right) {
    return left.i <= right.i;
}
int operator>=(const Integer& left,
               const Integer& right) {
    return left.i >= right.i;
}
int operator&&(const Integer& left,
               const Integer& right) {
    return left.i && right.i;
}
int operator||(const Integer& left,
               const Integer& right) {
    return left.i || right.i;
} ///:~
//: C12:IntegerTest.cpp
//{L} Integer
#include "Integer.h"
#include <fstream>
using namespace std;
ofstream out("IntegerTest.out");

void h(Integer& c1, Integer& c2) {
  // Un'espressione complessa:
  c1 += c1 * c2 + c2 % c1;
  #define TRY(OP) \
    out << "c1 = "; c1.print(out); \
```

```
    out << ", c2 = "; c2.print(out); \
    out << ";  c1 " #OP " c2 produces "; \
    (c1 OP c2).print(out); \
    out << endl;
  TRY(+) TRY(-) TRY(*) TRY(/)
  TRY(%) TRY(^) TRY(&) TRY(|)
  TRY(<<) TRY(>>) TRY(+=) TRY(-=)
  TRY(*=) TRY(/=) TRY(%=) TRY(^=)
  TRY(&=) TRY(|=) TRY(>>=) TRY(<<=)
  // Espressioni condizionali:
  #define TRYC(OP) \
    out << "c1 = "; c1.print(out); \
    out << ", c2 = "; c2.print(out); \
    out << ";  c1 " #OP " c2 produces "; \
    out << (c1 OP c2); \
    out << endl;
  TRYC(<) TRYC(>) TRYC(==) TRYC(!=) TRYC(<=)
  TRYC(>=) TRYC(&&) TRYC(||)
}

int main() {
  cout << "friend functions" << endl;
  Integer c1(47), c2(9);
  h(c1, c2);
} ///:~
//: C12:Byte.h
// Operatori sovraccaricati membri di classi
#ifndef BYTE_H
#define BYTE_H
#include "../require.h"
#include <iostream>
// Funzioni membro ("this" implicito):
class Byte {
  unsigned char b;
public:
  Byte(unsigned char bb = 0) : b(bb) {}
  // Senza effetti collaterali: funzioni membro const:
  const Byte
    operator+(const Byte& right) const {
    return Byte(b + right.b);
  }
  const Byte
    operator-(const Byte& right) const {
    return Byte(b - right.b);
  }
  const Byte
    operator*(const Byte& right) const {
    return Byte(b * right.b);
  }
  const Byte
    operator/(const Byte& right) const {
    require(right.b != 0, "divide by zero");
    return Byte(b / right.b);
  }
  const Byte
    operator%(const Byte& right) const {
    require(right.b != 0, "modulo by zero");
    return Byte(b % right.b);
  }
  const Byte
    operator^(const Byte& right) const {
    return Byte(b ^ right.b);
  }
```

```
const Byte
  operator&(const Byte& right) const {
  return Byte(b & right.b);
}
const Byte
  operator|(const Byte& right) const {
  return Byte(b | right.b);
}
const Byte
  operator<<(const Byte& right) const {
  return Byte(b << right.b);
}
const Byte
  operator>>(const Byte& right) const {
  return Byte(b >> right.b);
}
// Modifica per assegnamento & ritorno di lvalue.
// operator= può essere solo funzione membro:
Byte& operator=(const Byte& right) {
  // Gestisce l'auto-assegnamento:
  if(this == &right) return *this;
  b = right.b;
  return *this;
}
Byte& operator+=(const Byte& right) {
  if(this == &right) {/* auto-assegnamento */}
  b += right.b;
  return *this;
}
Byte& operator-=(const Byte& right) {
  if(this == &right) {/* auto-assegnamento */}
  b -= right.b;
  return *this;
}
Byte& operator*=(const Byte& right) {
  if(this == &right) {/* auto-assegnamento */}
  b *= right.b;
  return *this;
}
Byte& operator/=(const Byte& right) {
  require(right.b != 0, "divide by zero");
  if(this == &right) {/* auto-assegnamento */}
  b /= right.b;
  return *this;
}
Byte& operator%=(const Byte& right) {
  require(right.b != 0, "modulo by zero");
  if(this == &right) {/* auto-assegnamento */}
  b %= right.b;
  return *this;
}
Byte& operator^=(const Byte& right) {
  if(this == &right) {/* auto-assegnamento */}
  b ^= right.b;
  return *this;
}
Byte& operator&=(const Byte& right) {
  if(this == &right) {/* auto-assegnamento */}
  b &= right.b;
  return *this;
}
Byte& operator|=(const Byte& right) {
  if(this == &right) {/* auto-assegnamento */}
```

```
    b |= right.b;
    return *this;
  }
  Byte& operator>>=(const Byte& right) {
    if(this == &right) {/* auto-assegnamento */}
    b >>= right.b;
    return *this;
  }
  Byte& operator<<=(const Byte& right) {
    if(this == &right) {/* auto-assegnamento */}
    b <<= right.b;
    return *this;
  }
  // Operatori condizionali restituiscono true/false:
  int operator==(const Byte& right) const {
      return b == right.b;
  }
  int operator!=(const Byte& right) const {
      return b != right.b;
  }
  int operator<(const Byte& right) const {
      return b < right.b;
  }
  int operator>(const Byte& right) const {
      return b > right.b;
  }
  int operator<=(const Byte& right) const {
      return b <= right.b;
  }
  int operator>=(const Byte& right) const {
      return b >= right.b;
  }
  int operator&&(const Byte& right) const {
      return b && right.b;
  }
  int operator||(const Byte& right) const {
      return b || right.b;
  }
  // Scrive i contenuti su un ostream:
  void print(std::ostream& os) const {
    os << "0x" << std::hex << int(b) << std::dec;
  }
};
#endif // BYTE_H ///:~
//: C12:ByteTest.cpp
#include "Byte.h"
#include <fstream>
using namespace std;
ofstream out("ByteTest.out");

void k(Byte& b1, Byte& b2) {
  b1 = b1 * b2 + b2 % b1;

  #define TRY2(OP) \
    out << "b1 = "; b1.print(out); \
    out << ", b2 = "; b2.print(out); \
    out << ";  b1 " #OP " b2 produces "; \
    (b1 OP b2).print(out); \
    out << endl;

  b1 = 9; b2 = 47;
  TRY2(+) TRY2(-) TRY2(*) TRY2(/)
  TRY2(%) TRY2(^) TRY2(&) TRY2(|)
```

```
  TRY2(<<) TRY2(>>) TRY2(+=) TRY2(-=)
  TRY2(*=) TRY2(/=) TRY2(%=) TRY2(^=)
  TRY2(&=) TRY2(|=) TRY2(>>=) TRY2(<<=)
  TRY2(=) // Operatori di assegnamento

  // Espressioni condizionali:
  #define TRYC2(OP) \
    out << "b1 = "; b1.print(out); \
    out << ", b2 = "; b2.print(out); \
    out << ";  b1 " #OP " b2 produces "; \
    out << (b1 OP b2); \
    out << endl;

  b1 = 9; b2 = 47;
  TRYC2(<) TRYC2(>) TRYC2(==) TRYC2(!=) TRYC2(<=)
  TRYC2(>=) TRYC2(&&) TRYC2(||)

  // Assegnamenti concatenati:
  Byte b3 = 92;
  b1 = b2 = b3;
}

int main() {
  out << "member functions:" << endl;
  Byte b1(47), b2(9);
  k(b1, b2);
} ///:~
```

Si può vedere che all' **operator=** è consentito essere solo una funzione membro. Questo è spiegato in seguito.

Notare che tutti gli operatori di assegnamento hanno del codice per controllare l'auto-assegnamento; questo è una linea guida generale. In certi casi questo non è necessario; per esempio, con **operator+=** *possiamo* scrivere **A+=A** e ottenere la somma di **A** con se stesso. Dove è più importante controllare l'auto assegnamento è nell'**operator=** perchè con oggetti complicati si possono ottenere risultati disastrosi (in certi casi è OK, ma bisogna sempre stare attenti quando si scrive **operator=**).

Tutti gli operatori mostrati nei due esempi precedenti sono sovraccaricati per manipolare un singolo tipo. È anche possibile sovraccaricare operatori per manipolare tipi misti e sommare, per esempio, mele con arancie. Prima di cominciare ad usare in modo esaustivo il sovraccaricamento degli operatori, tuttavia, bisogna dare uno sguardo alla sezione sulla conversione automatica dei tipi più avanti in questo capitolo. Spesso una conversione di tipo alla destra dell'operatore può far risparmiare un sacco di operatori sovraccaricati.

Argomenti & valori di ritorno

Può sembrare leggermente fuorviante sulle prime quando si guarda dentro **OverloadingUnaryOperators.cpp**, **Integer.h** e **Byte.h** e si vedono tutti i modi diversi in cui vengono passati gli argomenti e restituiti i valori di ritorno. Benchè si *possano* passare e restituire gli argomenti in tutti i modi possibili, le scelte fatte in questi esempi non sono casuali. Esse seguono un modello logico, che è lo stesso che dovremmo usare nella maggior parte dei casi.

1. Come per una qualunque argomento di una funzione, se l'argomento serve solo in lettura e non bisogna fare cambiamenti su di esso è bene passarlo come riferimento

const. Le operazioni aritmetiche ordinarie (come + e –, ecc.) e i Booleani non cambiano i loro argomenti, e quindi il modo più diffuso di passare gli argomenti è come riferimento **const**. Quando la funzione è un membro di una classe, questo si traduce nell'avere funzioni membro **const**. Solo con gli operatori di assegnamento (come +=) e l' **operator=**, che cambiano l'argomento alla sinistra dell'operatore, l'argomento di sinistra *non* è una costante, ma viene passato come indirizzo perchè viene modificato.

2. Il tipo del valore di ritorno che dobbiamo scegliere dipende dal significato atteso dell'operatore (di nuovo, si può fare qualunque cosa con gli argomenti e i valori di ritorno.) Se l'effetto dell'operatore è di produrre un nuovo valore, c'è bisogno di generare un nuovo oggetto come valore di ritorno. Per esempio, **Integer::operator+** deve produrre un oggetto **Integer** che è la somma degli operandi. Questo oggetto viene restituito per valore come **const**, così il risultato non può essere modificato come un lvalue (valore di sinistra).
3. Tutti gli operatori di assegnamento modificano il valore di sinistra. Per consentire al risultato di un assegnamento di essere usato in espressioni concatenate, come **a=b=c**, ci si aspetta che il valore di ritorno sia un riferimento allo stesso valore di sinistra appena modificato. Ma questo riferimento deve essere **const** o non**const**? Benchè noi leggiamo l'espressione **a=b=c** da sinistra verso destra, il compilatore la analizza da destra verso sinistra, cosicchè non siamo obbligati a restituire un non**const** per supportare il concatenamento dell'assegnazione. Tuttavia la gente qualche volta si aspetta di poter effettuare operazioni sull'oggetto a cui è stata appena fatta l'assegnazione, come **(a=b).func( );** per chiamare **func( )** su **a** dopo avergli assegnato **b**. Perciò il valore di ritorno per tutti gli operatori di assegnamento potrebbe essere un riferimento non**const** al valore di sinistra.
4. Per gli operatori logici tutti si aspettano di avere come valore di ritorno nel peggiore dei casi un **int** e nel migliore dei casi un **bool** (le librerie sviluppate prima che molti compilatori C++ supportassero il tipo incorporato **bool,** usano un **int** o un **typedef** equivalente).

Gli operatori di incremento e decremento presentano il solito dilemma delle versioni pre- e postfissa. Entrambe le versioni modificano l'oggetto e quindi non possono trattare l'oggetto come **const**. La versione con prefisso restituisce il valore dell'oggetto dopo averlo modificato, quindi ci si aspetta come ritorno lo stesso oggetto su cui opera. In questo caso, quindi, si può semplicemente restituire un ***this** come riferimento. La versione con postfisso si suppone che restituisca il valore dell'oggetto *prima che* questo venga modificato, perciò siamo obbligati a creare un oggetto separato per rappresentare questo valore e restituirlo. In questo caso il ritorno deve essere fatto per valore se si vuole preservare il significato atteso (Notare che qualche volta si possono trovare gli operatori di incremento e decremento che restituiscono un **int** o un **bool** per indicare, per esempio, che un oggetto utilizzato per scorrere una lista ha raggiunto la fine di questa lista). Adesso la domanda è: il valore di ritorno deve essere **const** o non**const**? Se si consente all'oggetto di essere modificato e qualcuno scrive **(++a).func( )**, **func( )** opera su **a** stessa, ma con **(a++).func( )**, **func( )** opera su un oggetto temporaneo restituito da **operator++** in versione con postfisso. Gli oggetti temporanei sono automaticamente **const**, perciò questo sarà marcato dal compilatore, ma per coerenza ha molto più senso renderli entrambi **const**, come viene fatto qui. Oppure si può scegliere di rendere non-**const** la versione con prefisso e **const** la versione con postfisso. A causa della varietà di significati che si possono dare agli operatori di incremento e decremento, è necessario considerarli caso per caso.

Ritorno per valore come const

Il ritorno per valore come **const** potrebbe sembrare un pò subdolo sulle prime, perciò richiede un minimo di spiegazione. Consideriamo l'operatore binario **operator+**. Se lo usiamo in un'espressione come **f(a+b)**, il risultato di **a+b** diventa un oggetto temporaneo usato nella chiamata di **f( )**. Siccome è temporaneo, esso è automaticamente **const**, perciò non ha nessun effetto la dichiarazione esplicita **const** o non-**const**.

Tuttavia è anche possibile inviare un messaggio al valore di ritorno di **a+b**, piuttosto che passarlo semplicemente ad una funzione. Per esempio possiamo scrivere **(a+b).g( )**, in cui **g( )** è una qualche funzione membro della classe **Integer**, in questo caso. Rendendo **const** il valore di ritorno si impone che solo le funzioni membro **const** possano essere chiamate per questo valore di ritorno. Questo è un uso corretto di **const**, perchè previene il potenziale errore di memorizzare un'informazione preziosa in un oggetto che molto probabilmente verrebbe perso.

Ottimizzazione del valore di ritorno

Quando vengono creati nuovi oggetti da restituire per valore, guardiamo la forma usata. In **operator+**, per esempio:

```
return Integer(left.i + right.i);
```

Questa potrebbe sembrare sulle prime una "chiamata a funzione di un costruttore," ma non lo è. La sintassi è quella di un oggetto temporaneo; l'istruzione dice "costruisci un oggetto temporaneo di tipo **Integer** e restituiscilo." Per questo, si potrebbe pensare che il risultato è lo stesso che si ottiene creando un oggetto locale e restituendolo. Invece la cosa è alquanto diversa. Se invece scriviamo:

```
Integer tmp(left.i + right.i);
return tmp;
```

succedono tre cose. Primo, l'oggetto **tmp** viene creato includendo la chiamata al suo costruttore. Secondo, il costruttore di copia copia **tmp** nella locazione del valore di ritorno (esterna allo scope). Terzo, alla fine viene chiamato il distruttore di **tmp**.

In contrasto, l'approccio del "ritorno di un oggetto temporaneo" funziona in maniera completamente diversa. Quando il compilatore vede fare una cosa del genere, sa che non c'è nessun altro interesse sull'oggetto creato se non quello di usarlo come valore di ritorno. Il compilatore trae vantaggio da questa informazione, creando l'oggetto *direttamente* dentro la locazione del valore di ritorno, esterna allo scope. Questo richiede solo una chiamata al costruttore ordinario (non è necessario alcun costruttore di copia) e non c'è la chiamata al distruttore, in quanto di fatto non viene mai creato l'oggetto temporaneo. Così, se da una parte non costa nulla se non un pò di accortezza da parte del programmatore, dall'altra è significativamente molto più efficiente. Questo viene spesso chiamata *ottimizzazione del valore di ritorno*.

Operatori inusuali

Alcuni operatori aggiuntivi hanno una sintassi leggermente diversa per il sovraccaricamento.

L'operatore sottoscrizione, **operator[ ]**, deve essere una funzione membro e richiede un singolo argomento. Siccome l' **operator[ ]** implica che l'oggetto per il quale viene chiamato agisce come un array, il valore di ritorno per questo operatore deve essere un riferimento, così può essere usato convenientemente alla sinistra del segno di uguale. Questo operatore viene comunemente sovraccaricato; se ne potranno vedere esempi nel resto del libro.

Gli operatori **new** e **delete** controllano l'allocazione dinamica della memoria e possono essere sovraccaricati in molti modi diversi. Questo argomento è trattato nel Capitolo 13.

Operatore virgola

L'operatore virgola viene chiamato quando appare dopo un oggetto del tipo per cui l'operatore virgola è definito . Tuttavia, "**operator,**" (operatore virgola) *non* viene chiamato per le liste di argomenti di funzioni, ma solo per oggetti che appaiono separati da virgole. Non sembra che ci siano tanti usi pratici di questo operatore ed è stato messo nel linguaggio solo per coerenza. Qui c'è un esempio di come la funzione virgola può essere chiamata quando appare una virgola *prima o dopo* un oggetto:

```
//: C12:OverloadingOperatorComma.cpp
#include <iostream>
using namespace std;

class After {
public:
  const After& operator,(const After&) const {
    cout << "After::operator,()" << endl;
    return *this;
  }
};

class Before {};

Before& operator,(int, Before& b) {
  cout << "Before::operator,()" << endl;
  return b;
}

int main() {
  After a, b;
  a, b;  // Viene chiamato l'operatore virgola

  Before c;
  1, c;  // Viene chiamato l'operatore virgola
} ///:~
```

La funzione globale consente alla virgola di essere posta prima dell'oggetto in questione. L'uso mostrato è piuttosto oscuro e opinabile. Benchè si potrebbe usare un lista separata da virgole come parte di un'espressione più complessa, è molto subdolo nella maggior parte delle situazioni.

Operator->

L' **operator–>** (dereferenziazione di puntatore) è generalmente usato quando si vuole fare apparire un oggetto come puntatore. Siccome un oggetto di questo tipo incorpora molta più "intelligenza" di quanta ne abbia un puntatore normale, spesso viene chiamato

puntatore intelligente. Questo è utile soprattutto se si vuole "avvolgere" una classe intorno ad un puntatore per renderlo sicuro, o nell'uso comune di un *iteratore*, cioè un oggetto che scorre all'interno di una *collezione /contenitore* di altri oggetti e li seleziona uno alla volta, senza fornire l'accesso diretto all'implementazione del contenitore (I contenitori e gli iteratori si possono trovare spesso nelle librerie di classi, come nella libreria Standard del C++, descritta nel Volume 2 di questo libro).

Questo operatore deve essere una funzione membro. Esso ha dei vincoli aggiuntivi atipici: deve restituire un oggetto (o un riferimento ad un oggetto) che abbia anch'esso un operatore di dereferenziazione di puntatore, oppure deve restituire un puntatore che può essere usato per selezionare quello a cui punta l'operatore. Qui c'è un esempio:

```
//: C12:SmartPointer.cpp
#include <iostream>
#include <vector>
#include "../require.h"
using namespace std;

class Obj {
  static int i, j;
public:
  void f() const { cout << i++ << endl; }
  void g() const { cout << j++ << endl; }
};

// Definizioni di membri static:
int Obj::i = 47;
int Obj::j = 11;

// Container:
class ObjContainer {
  vector<Obj*> a;
public:
  void add(Obj* obj) { a.push_back(obj); }
  friend class SmartPointer;
};

class SmartPointer {
  ObjContainer& oc;
  int index;
public:
  SmartPointer(ObjContainer& objc) : oc(objc) {
    index = 0;
  }
  // Il valore di ritorno indica la fine della lista:
  bool operator++() { // Prefisso
    if(index >= oc.a.size()) return false;
    if(oc.a[++index] == 0) return false;
    return true;
  }
  bool operator++(int) { // Postfisso
    return operator++(); // Usa la versione con prefisso
  }
  Obj* operator->() const {
    require(oc.a[index] != 0, "Zero value "
      "returned by SmartPointer::operator->()");
    return oc.a[index];
  }
};
```

```
int main() {
  const int sz = 10;
  Obj o[sz];
  ObjContainer oc;
  for(int i = 0; i < sz; i++)
    oc.add(&o[i]); // Lo riempie
  SmartPointer sp(oc); // Crea un iteratore
  do {
    sp->f(); // Chiamata all'operatore di dereferenziazione di puntatore
    sp->g();
  } while(sp++);
} ///:~
```

La classe **Obj** definisce gli oggetti che vengono manipolati in questo programma. Le funzioni **f( )** e **g( )** semplicemente stampano valori di interesse usando i dati membri **static**. I puntatori a questi oggetti sono memorizzati all'interno dei contenitori di tipo **ObjContainer** usando la sua funzione **add( )**. **ObjContainer** si presenta come un array di puntatori, ma si può notare che non c'è nessun modo di tirar fuori questi puntatori. Tuttavia, **SmartPointer** è dichiarata come classe **friend**, perciò ha il permesso di guardare dentro al contenitore. La classe **SmartPointer** si presenta molto più come un puntatore intelligente – lo si può spostare in avanti usando l' **operator++** (si può anche definire un **operator– –**), non può oltrepassare la fine del contenitore a cui punta, e produce (attraverso l'operatore di dereferenziazione di puntatori) il valore a cui punta. Notare che **SmartPointer** è un adattamento personalizzato per il contenitore per cui è stato creato; a differenza di un puntatore ordinario, non c'è un puntatore intelligente "general purpose". Si può apprendere molto di più sui puntatori intelligenti detti "iteratori" nell'ultimo capitolo di questo libro e nel Volume 2 (scaricabile da *www.BruceEckel.com*).

In **main( )**, una volta che il contenitore **oc** è riempito con oggetti di tipo **Obj** , viene creato uno **SmartPointer sp**. Le chiamate al puntatore intelligente avvengono nelle espressioni:

```
sp->f(); // Chiamate a puntatori intelligenti
sp->g();
```

Qui, anche se **sp** di fatto non ha le funzioni membro **f( )** e **g( )**, l'operatore di dereferenziazione di puntatori automaticamente chiama queste funzioni per l'**Obj*** restituito da **SmartPointer::operator–>**. Il compilatore effettua tutti i controlli per assicurare che la chiamata alla funzione lavori correttamente.

Un iteratore nidificato

È molto più comune vedere una classe "puntatore intelligente" o "iteratore" nidificata all'interno della classe che essa serve. L'esempio precedente può essere riscritto per nidificare **SmartPointer** all'interno di **ObjContainer**, come segue:

```
//: C12:NestedSmartPointer.cpp
#include <iostream>
#include <vector>
#include "../require.h"
using namespace std;

class Obj {
  static int i, j;
public:
  void f() { cout << i++ << endl; }
  void g() { cout << j++ << endl; }
```

```
};

// Definizioni di membri Static:
int Obj::i = 47;
int Obj::j = 11;

// Container:
class ObjContainer {
  vector<Obj*> a;
public:
  void add(Obj* obj) { a.push_back(obj); }
  class SmartPointer;
  friend class SmartPointer;
  class SmartPointer {
    ObjContainer& oc;
    unsigned int index;
  public:
    SmartPointer(ObjContainer& objc) : oc(objc) {
      index = 0;
    }
    // Il valore di ritorno indica la fine della lista:
    bool operator++() { // Prefisso
      if(index >= oc.a.size()) return false;
      if(oc.a[++index] == 0) return false;
      return true;
    }
    bool operator++(int) { // Postfisso
      return operator++(); // Usa la versione con prefisso
    }
    Obj* operator->() const {
      require(oc.a[index] != 0, "Zero value "
        "returned by SmartPointer::operator->()");
      return oc.a[index];
    }
  };
  // Funzione per produrre un puntatore intelligente che
  // punta all'inizio di ObjContainer:
  SmartPointer begin() {
    return SmartPointer(*this);
  }
};

int main() {
  const int sz = 10;
  Obj o[sz];
  ObjContainer oc;
  for(int i = 0; i < sz; i++)
    oc.add(&o[i]); //Lo riempie
  ObjContainer::SmartPointer sp = oc.begin();
  do {
    sp->f(); // Chiamata all'operatore di dereferenziazione di puntatore
    sp->g();
  } while(++sp);
} ///:~
```

Oltre alla nidificazione delle classi, ci sono solo due differenze qui. La prima è nella dichiarazione della classe in modo che possa essere **friend**:

```
class SmartPointer;
friend SmartPointer;
```

Il compilatore deve innanzitutto sapere che la classe esiste prima che possa dire che è una **friend**.

La seconda differenza è nella funzione **begin( ),** membro di **ObjContainer**, che produce uno **SmartPointer** che punta all'inizio della sequenza **ObjContainer**. Benchè sia solo una questione di convenienza, è preziosa perchè segue parte della forma usata nella libreria Standard del C++.

Operator->*

L' **operator–>*** è un operatore binario che si comporta come tutti gli altri operatori binari. Viene fornito per quelle situazioni in cui si vuole simulare il comportamento di un *puntatore-a-membro*, descritto nel capitolo precedente.

Proprio come l' **operator->**, l'operatore di dereferenziazione di puntatore-a-membro viene generalmente usato con certi tipi di oggetti che rappresentano un "puntatore intelligente," anche se l'esempio mostrato qui è più semplice, in modo da risultare comprensibile. Il trucco quando si definisce l' **operator->*** è che deve ritornare un oggetto per il quale l' **operator( )** (chiamata a funzione) può essere chiamato con gli argomenti da passare alla funzione membro che si sta chiamando.

L'operatore *chiamata a funzione*, **operator( ),** deve essere una funzione membro, ed è unica per il fatto che accetta un numero qualsiasi di argomenti. Esso fa si che il nostro oggetto si presenti come se fosse una funzione. Anche se è possibile definire diverse funzioni con l' **operator( )** sovraccaricato, con diversi argomenti, questo viene spesso usato per tipi che hanno una sola operazione (funzione membro) o che hanno almeno un'operazione particolarmente importante rispetto alle altre. Possiamo vedere nel Volume 2 che la Libreria dello Standard C++ usa l'operatore chiamata a funzione per creare "oggetti funzione."

Per creare un **operator->*** bisogna prima creare una classe con un **operator( )** che rappresenta il tipo di oggetto che l' **operator->*** deve restituire. Questa classe deve in qualche modo catturare le informazioni necessarie affinchè quando l' **operator( )** viene chiamato (il che avviene automaticamente), il puntatore-a-membro viene dereferenziato per l'oggetto. Nell'esempio seguente, il costruttore **FunctionObject** cattura e memorizza sia il puntatore all'oggetto che il puntatore alla funzione membro, e quindi l' **operator( )** li usa per costruire la chiamata al puntatore-a-membro:

```
//: C12:PointerToMemberOperator.cpp
#include <iostream>
using namespace std;

class Dog {
public:
  int run(int i) const {
    cout << "run\n";
    return i;
  }
  int eat(int i) const {
     cout << "eat\n";
     return i;
  }
  int sleep(int i) const {
    cout << "ZZZ\n";
    return i;
```

```
  }
  typedef int (Dog::*PMF)(int) const;
  // operator->* deve restituire un oggetto
  // che ha un operator():
  class FunctionObject {
    Dog* ptr;
    PMF pmem;
  public:
    // Salva il puntatore all'oggetto e il puntatore al membro
    FunctionObject(Dog* wp, PMF pmf)
      : ptr(wp), pmem(pmf) {
      cout << "FunctionObject constructor\n";
    }
    // Effettua la chiamata usando il puntatore all'oggetto
    // e il puntatore al membro
    int operator()(int i) const {
      cout << "FunctionObject::operator()\n";
      return (ptr->*pmem)(i); // Effettua la chiamata
    }
  };
  FunctionObject operator->*(PMF pmf) {
    cout << "operator->*" << endl;
    return FunctionObject(this, pmf);
  }
};

int main() {
  Dog w;
  Dog::PMF pmf = &Dog::run;
  cout << (w->*pmf)(1) << endl;
  pmf = &Dog::sleep;
  cout << (w->*pmf)(2) << endl;
  pmf = &Dog::eat;
  cout << (w->*pmf)(3) << endl;
} ///:~
```

Dog ha tre funzioni membro, e tutte prendono un argomento **int** e restituiscono un **int**. **PMF** è un **typedef** per semplificare la definizione di un puntatore-a-membro per le funzioni membro di **Dog**.

Con **operator->*** viene creata e restituita una **FunctionObject**. Notare che **operator->*** conosce sia il puntatore-a-membro che l'oggetto per cui questo viene chiamato (**this**), e li passa al costruttore di **FunctionObject** che ne memorizza i valori. Quando viene chiamato **operator->***, il compilatore chiama immediatamente l' **operator( )** per calcolare il valore di ritorno di **operator->***, passandogli gli argomenti di **operator->***. L'operatore **FunctionObject::operator( )** prende gli argomenti e dereferenzia il "reale" puntatore-a-membro usando il puntatore all'oggetto e il puntatore-a-membro da esso memorizzati.

Notare che quello che stiamo facendo qui, proprio come con l' **operator->**, è di inserirci nel mezzo della chiamata ad **operator->***. Questo ci permette di fare delle operazioni extra, se ne abbiamo bisogno.

Il meccanismo dell'**operator->*** implementato qui funziona solo con funzioni membro che prendono come argomento un **int** e restituiscono un **int**. Questo è limitativo, ma creare un meccanismo di sovraccaricamento per ogni singola possibilità sembra un compito proibitivo. Fortunatamente il meccanismo dei **template** del C++ (descritto

nell'ultimo capitolo di questo libro e nel Volume 2) è stato pensato proprio per gestire problemi di questo tipo.

Operatori che non si possono sovraccaricare

Ci sono certi operatori tra quelli disponibili che non sono sovraccaricabili. Il motivo generico per questa restrizione è la sicurezza. Se questi operatori fossero sovraccaricabili, si potrebbero in qualche modo mettere a repentaglio o rompere i meccanismi di sicurezza, rendendo le cose difficili o confondendo la pratica esistente.

- L' **operator.** (operatore punto) di selezione dei membri. Correntemente, il punto ha un significato per qualunque membro di classe, ma se se ne permettesse il sovraccaricamento, allora non si potrebbe più accedere ai membri in maniera normale, ma solo con i puntatori o con l'operatore **operator->**.
- L' **operator .*** (operatore punto asterisco), dereferenziatore del puntatore a membro, per gli stessi motivi dell' **operator.** (operatore punto).
- Non c'è un operatore per l'elevamento a potenza. La scelta più popolare per questa operazione è stato l' **operator**** (operatore **) dal Fortran, ma questo solleva un difficile problema di parsing. Neanche il C ha un operatore per l'elevamento a potenza, per cui sembra che il C++ non ne abbia bisogno di uno, anche perchè si può sempre usare una funzione. Un operatore di elevamento a potenza potrebbe aggiungere una notazione comoda, ma questa nuova funzionalità nel linguaggio non compenserebbe la complessità aggiunta al compilatore.
- Non ci sono operatori definiti dall'utente. Cioè non si possono aggiungere (e sovraccaricare) operatori che non siano già tra quelli disponibili nel linguaggio. Il problema è in parte nella difficoltà di gestire le precedenze e in parte nel fatto che lo sforzo non sarebbe compensato dai vantaggi.
- Non si possono cambiare le regole di precedenza. Sono già abbastanza difficili da ricordare così come sono, senza lasciare che la gente ci giochi.

Operatori non-membro

In alcuni degli esempi precedenti, gli operatori possono essere membri o non-membri e non sembra esserci molta differenza. Questo in genere solleva una questione, "Quale dobbiamo scegliere?" In generale, se non c'è differenza è preferibile scegliere un membro di classe, per enfatizzare l'associazione tra l'operatore e la sua classe. Quando l'operando di sinistra è sempre un oggetto della classe corrente questo funziona benissimo.

Tuttavia qualche volta si vuole che l'operando di sinistra sia un oggetto di qualche altra classe. Un caso comune in cui si può vedere questo è quando gli operatori << e >> sono sovraccaricati per iostreams. Siccome iostreams è una libreria fondamentale del C++, probabilmente si vogliono sovraccaricare questi operatori per molte classi proprie, per cui il problema merita di essere affrontato:

```
//: C12:IostreamOperatorOverloading.cpp
// Esempi di operatori sovraccaricati non-membri
#include "../require.h"
#include <iostream>
#include <sstream> // "String streams"
#include <cstring>
using namespace std;
```

```
class IntArray {
  enum { sz = 5 };
  int i[sz];
public:
  IntArray() { memset(i, 0, sz* sizeof(*i)); }
  int& operator[](int x) {
    require(x >= 0 && x < sz,
      "IntArray::operator[] out of range");
    return i[x];
  }
  friend ostream&
    operator<<(ostream& os, const IntArray& ia);
  friend istream&
    operator>>(istream& is, IntArray& ia);
};

ostream&
operator<<(ostream& os, const IntArray& ia) {
  for(int j = 0; j < ia.sz; j++) {
    os << ia.i[j];
    if(j != ia.sz -1)
      os << ", ";
  }
  os << endl;
  return os;
}

istream& operator>>(istream& is, IntArray& ia){
  for(int j = 0; j < ia.sz; j++)
    is >> ia.i[j];
  return is;
}

int main() {
  stringstream input("47 34 56 92 103");
  IntArray I;
  input >> I;
  I[4] = -1; // Usa l'operatore sovraccaricato operator[]
  cout << I;
} ///:~
```

Questa classe contiene anche un **operator [ ]** sovraccaricato, che restituisce un riferimento al giusto valore nell'array. Siccome viene restituito un riferimento, l'espressione

```
I[4] = -1;
```

non solo si presenta in modo molto più umano di quanto non si sarebbe ottenuto con l'uso dei puntatori, ma sortisce anche l'effetto desiderato.

È importante che gli operatori di shift sovraccaricati ritornino *per riferimento*, in modo che l'azione si rifletta sugli oggetti esterni. Nelle definizioni di funzioni, un'espressione come

```
os << ia.i[j];
```

causa la chiamata alle funzioni *esistenti* dell'operatore sovraccaricato (cioè quelle definite in **<iostream>**). In questo caso, la funzione chiamata è **ostream& operator<<(ostream&, int)** perchè **ia.i[j]** si risolve in un **int**.

Una volta che sono state effettuate tutte le azioni su **istream** o su **ostream**, questo viene restituito in modo che possa essere usato in espressioni più complicate.

In **main( )** viene usato un nuovo tipo di **iostream**: **stringstream** (dichiarato in **<sstream>**). Questa è una classe che prende una **string** (che può costruire a partire da un array di **char**, come mostrato qui) e la traduce in un **iostream**. Nell'esempio di sopra, questo significa che gli operatori di shift possono essere testati senza aprire un file o digitando dati dalla linea di comando.

La forma mostrata in questo esempio per l'inserimento e l'estrazione dei dati è standard. Se si vogliono creare questi operatori per la propria classe si può copiare il prototipo delle funzioni e i tipi di ritorno e seguire la stessa forma per il corpo.

Linee guida di base

Murray[49] suggerisce queste linee guida per la scelta tra membri e non-membri:

Operatore	Uso raccomandato
Tutti gli operatori unari	membri
= ( ) [ ] –> –>*	*Devono essere* membri
+= –= /= *= ^= &= |= %= >>= <<=	membri
Tutti gli altri operatori binari	non-membri

Assegnamento con il sovraccaricamento

Una fonte comune di confusione per i nuovi programmatori in C++ è l'assegnamento. Questo perchè il segno di = (uguale) è un'operazione fondamentale nella programmazione, dal livello più alto fino alla copia di un registro a livello macchina. In più qualche volta viene invocato anche il costruttore di copia (descritto nel Capitolo 11) quando si usa il segno di =:

```
MyType b;
MyType a = b;
a = b;
```

Nella seconda linea viene *definito* l'oggetto **a**. Un nuovo oggetto viene creato laddove non ne esisteva uno prima. Siccome sappiamo come il compilatore C++ sia difensivo riguardo all'inizializzazione di oggetti, sappiamo anche che deve essere sempre chiamato un costruttore dove si definisce un oggetto. Ma quale costruttore? **a** viene creato da un oggetto di tipo **MyType** preesistente (**b**, sul lato destro del segno di =), per cui c'è una sola scelta: il costruttore di copia. Anche se è coinvolto un segno di uguale, viene di fatto chiamato il costruttore di copia.

Nella terza linea le cose sono diverse. Sul lato sinistro del segno di uguale c'è un oggetto precedentemente inizializzato. Chiaramente non viene chiamato un costruttore per un

oggetto che è stato già creato. In questo caso viene chiamato l'operatore **MyType::operator=** per **a**, prendendo come argomento tutto ciò che si trova sul lato destro del segno di uguale (si possono avere più funzioni **operator=** per trattare differenti titpi di argomenti).

Questo comportamento non è limitato al costruttore di copia. Ogni volta che si inizializza un oggetto usando un segno di = invece della forma ordinaria di chiamata a funzione del costruttore, il compilatore cerca un costruttore che accetta tutto ciò che sta sul lato destro dell'operatore:

```
//: C12:CopyingVsInitialization.cpp
class Fi {
public:
  Fi() {}
};

class Fee {
public:
  Fee(int) {}
  Fee(const Fi&) {}
};

int main() {
  Fee fee = 1; // Fee(int)
  Fi fi;
  Fee fum = fi; // Fee(Fi)
} ///:~
```

Quando si ha a che fare con il segno di = è importante avere in mente questa distinzione : se l'oggetto non è stato ancora creato, è richiesta un'inizializzazione; altrimenti viene usato l' **operator=**.

È comunque meglio evitare di scrivere codice che usa il segno di = per l'inizializzazione; usare, invece, sempre la forma esplicita del costruttore. Le due costruzioni con il segno di uguale diventano quindi:

```
Fee fee(1);
Fee fum(fi);
```

In questo modo si evita di confondere il lettore.

Il comportamento dell'operator=

In **Integer.h** e **Byte.h**, asseriamo che **operator=** può essere solo una funzione membro. Esso è intimamente legato all'oggetto sul lato sinistro di '='. Se fosse possibile definire l' **operator=** globalmente, potremmo tentare di ridefinire il segno '=' predefinito:

```
int operator=(int, MyType); // Global = non consentito!
```

Il compilatore evita questa possibilità forzando l' **operator=** ad essere una funzione membro.

Quando si crea un **operator=**, bisogna copiare tutte le informazioni necessarie dall'oggetto presente sul lato destro all'oggetto corrente (cioè l'oggetto per il quale l'

operator= viene chiamato) per effettuare quello che consideriamo un "assegnamento" per la nostra classe. Per oggetti semplici questo è ovvio:

```
//: C12:SimpleAssignment.cpp
// Semplice operator=()
#include <iostream>
using namespace std;

class Value {
  int a, b;
  float c;
public:
  Value(int aa = 0, int bb = 0, float cc = 0.0)
    : a(aa), b(bb), c(cc) {}
  Value& operator=(const Value& rv) {
    a = rv.a;
    b = rv.b;
    c = rv.c;
    return *this;
  }
  friend ostream&
  operator<<(ostream& os, const Value& rv) {
    return os << "a = " << rv.a << ", b = "
      << rv.b << ", c = " << rv.c;
  }
};

int main() {
  Value a, b(1, 2, 3.3);
  cout << "a: " << a << endl;
  cout << "b: " << b << endl;
  a = b;
  cout << "a after assignment: " << a << endl;
} ///:~
```

Qui l'oggetto sul lato sinistro del segno di = copia tutti gli elementi dell'oggetto sulla destra, e poi ritorna un riferimento a se stesso, il che permette di creare un'espressione molto più complessa.

Questo esempio contiene un errore comune. Quando si effettua un assegnamento tra due oggetti dello stesso tipo bisogna sempre fare il controllo sull'auto assegnamento: l'oggetto viene assegnato a se stesso? In certi casi, come questo, è innocuo effettuare l'operazione di assegnamento in qualunque modo, ma se vengono apportate modifiche all'implementazione della classe ci possono essere delle differenze, e se non si tiene conto di questo si possono introdurre errori molto difficili da scovare.

I puntatori nelle classi

Cosa succede se l'oggetto non è così semplice? Per esempio, cosa succede se l'oggetto contiene puntatori ad altri oggetti? Semplicemente, copiando un puntatore si finisce per avere due oggetti che puntano alla stessa locazione di memoria. In situazioni come queste, bisogna fare un pò di conti.

Ci sono due approcci comuni al problema. La tecnica più semplice è quella di copiare tutti i dati puntati dal puntatore quando si fa l'assegnamento o si invoca il costruttore di copia. Questo è chiaro:

```
//: C12:CopyingWithPointers.cpp
// Soluzione del problema della replica del puntatore
// con la duplicazione dell'oggetto che viene puntato
// durante l'assegnamento e la costruzione della copia.
#include "../require.h"
#include <string>
#include <iostream>
using namespace std;

class Dog {
  string nm;
public:
  Dog(const string& name) : nm(name) {
    cout << "Creating Dog: " << *this << endl;
  }
  // Il costruttore di copia & operator=
  // sono corretti.
  // Crea un oggetto Dog da un puntatore a Dog:
  Dog(const Dog* dp, const string& msg)
    : nm(dp->nm + msg) {
    cout << "Copied dog " << *this << " from "
         << *dp << endl;
  }
  ~Dog() {
    cout << "Deleting Dog: " << *this << endl;
  }
  void rename(const string& newName) {
    nm = newName;
    cout << "Dog renamed to: " << *this << endl;
  }
  friend ostream&
  operator<<(ostream& os, const Dog& d) {
    return os << "[" << d.nm << "]";
  }
};

class DogHouse {
  Dog* p;
  string houseName;
public:
  DogHouse(Dog* dog, const string& house)
   : p(dog), houseName(house) {}
  DogHouse(const DogHouse& dh)
    : p(new Dog(dh.p, " copy-constructed")),
      houseName(dh.houseName
        + " copy-constructed") {}
  DogHouse& operator=(const DogHouse& dh) {
    // Controlla l'auto-assegnamento:
    if(&dh != this) {
      p = new Dog(dh.p, " assigned");
      houseName = dh.houseName + " assigned";
    }
    return *this;
  }
  void renameHouse(const string& newName) {
    houseName = newName;
  }
  Dog* getDog() const { return p; }
  ~DogHouse() { delete p; }
  friend ostream&
  operator<<(ostream& os, const DogHouse& dh) {
    return os << "[" << dh.houseName
      << "] contains " << *dh.p;
```

```
  }
};

int main() {
  DogHouse fidos(new Dog("Fido"), "FidoHouse");
  cout << fidos << endl;
  DogHouse fidos2 = fidos; // Costruzione della copia
  cout << fidos2 << endl;
  fidos2.getDog()->rename("Spot");
  fidos2.renameHouse("SpotHouse");
  cout << fidos2 << endl;
  fidos = fidos2; // Assegnamento
  cout << fidos << endl;
  fidos.getDog()->rename("Max");
  fidos2.renameHouse("MaxHouse");
} ///:~
```

Dog è una classe semplice che contiene solo una **string** che memorizza il nome del cane. Tuttavia bisogna in genere sapere quando succede qualcosa a **Dog** perchè i costruttori e i distruttori stampano informazioni quando vengono chiamati. Notare che il secondo costruttore è come un costruttore di copia, solo che prende un puntatore a **Dog** invece di un riferimento, ed ha un secondo argomento che è un messaggio che viene concatenato al nome dell'argomento **Dog**. Questo viene usato per aiutare nel tracciamento del comportamento del programma.

Si può vedere che ogni volta che una funzione membro stampa delle informazioni, non accede direttamente a queste informazioni, ma invia ***this** a **cout**. Questo a sua volta chiama l' **ostream operator<<**. È importante farlo in questo modo, perchè se vogliamo riformattare il modo in cui le informazioni di **Dog** vengono visualizzate (come abbiamo fatto aggiungendo '[' e ']') bisogna agire in un posto solo.

Un **DogHouse** contiene un **Dog*** e mostra le quattro funzioni che bisogna sempre definire quando la classe contiene dei puntatori: tutti i costruttori ordinari necessari, il costruttore di copia, **operator=** (o lo si definisce o se ne impedisce l'uso), e il distruttore. L' **operator=** fa il controlo sull'auto-assegnamento, come deve essere, anche se qui non è strettamente necessario. Questo elimina virtualmente la possibilità di dimenticarsene se si fanno delle modifiche al codice che lo rendono necessario.

Il conteggio dei Riferimenti

Nell'esempio di sopra, il costruttore di copia e l' **operator=** effettuano una nuova copia dei dati puntati dal puntatore, e il distruttore la cancella. Tuttavia se il nostro oggetto richiede un sacco di memoria o un elevato overhead di inizializzazione, si vorrebbe evitare questa copia. Un approccio comune a questo problema è chiamato *conteggio dei riferimenti*. Si conferisce intelligenza all'oggetto che viene puntato in modo che esso sa quanti oggetti lo stanno puntando. La costruzione-copia o l'assegnamento comportano l'aggiunta di un altro puntatore ad un oggetto esistente e l'incremento di un contatore di riferimenti. La distruzione comporta il decremento del contatore dei riferimenti e la distruzione dell'oggetto se il contatore arriva a zero.

Ma cosa succede se vogliamo scrivere nell'oggetto (**Dog** nell'esempio di sopra)? Più di un oggetto può usare questo **Dog**, cosicchè potremmo andare a modificare il **Dog** di qualcun altro come il nostro, il che non sembra essere molto carino. Per risolvere questo problema di "duplicazione" viene usata una tecnica aggiuntiva chiamata *copia-su-scrittura*. Prima di

scrivere un blocco di memoria ci si assicura che nessun altro vi sta scrivendo. Se il contatore di riferimenti è maggiore di uno bisogna fare una copia personale del blocco di memoria prima di scriverci dentro, in modo da non disturbare il territorio di qualcun altro . Qui c'è un semplice esempio del contatore di riferimenti e della copia su scrittura:

```
//: C12:ReferenceCounting.cpp
// Contatore di riferimenti, copia-su-scrittura
#include "../require.h"
#include <string>
#include <iostream>
using namespace std;

class Dog {
  string nm;
  int refcount;
  Dog(const string& name)
    : nm(name), refcount(1) {
    cout << "Creating Dog: " << *this << endl;
  }
  // Previene l'assegnamento:
  Dog& operator=(const Dog& rv);
public:
  // Gli oggetti Dog possono essere creati solo sull'heap:
  static Dog* make(const string& name) {
    return new Dog(name);
  }
  Dog(const Dog& d)
    : nm(d.nm + " copy"), refcount(1) {
    cout << "Dog copy-constructor: "
         << *this << endl;
  }
  ~Dog() {
    cout << "Deleting Dog: " << *this << endl;
  }
  void attach() {
    ++refcount;
    cout << "Attached Dog: " << *this << endl;
  }
  void detach() {
    require(refcount != 0);
    cout << "Detaching Dog: " << *this << endl;
    // Distrugge l'oggetto se nessuno lo sta usando:
    if(--refcount == 0) delete this;
  }
  // Copia questo oggetto Dog in modo condizionale.
  // La chiama prima di modificare Dog, assegna
  // il puntatore di ritorno a Dog*.
  Dog* unalias() {
    cout << "Unaliasing Dog: " << *this << endl;
    // Non lo duplica se non replicato:
    if(refcount == 1) return this;
    --refcount;
    // Usa il costruttore di copia per duplicare:
    return new Dog(*this);
  }
  void rename(const string& newName) {
    nm = newName;
    cout << "Dog renamed to: " << *this << endl;
  }
  friend ostream&
  operator<<(ostream& os, const Dog& d) {
    return os << "[" << d.nm << "], rc = "
```

```
      << d.refcount;
  }
};

class DogHouse {
  Dog* p;
  string houseName;
public:
  DogHouse(Dog* dog, const string& house)
   : p(dog), houseName(house) {
    cout << "Created DogHouse: "<< *this << endl;
  }
  DogHouse(const DogHouse& dh)
    : p(dh.p),
      houseName("copy-constructed " +
        dh.houseName) {
    p->attach();
    cout << "DogHouse copy-constructor: "
         << *this << endl;
  }
  DogHouse& operator=(const DogHouse& dh) {
    // Controlla auto-assegnamento:
    if(&dh != this) {
      houseName = dh.houseName + " assigned";
      // Prima cancella quello che stiamo usando:
      p->detach();
      p = dh.p; // Simile al costruttore di copia
      p->attach();
    }
    cout << "DogHouse operator= : "
         << *this << endl;
    return *this;
  }
  // Decrementa refcount, e distrugge l'oggetto condizionatamente
  ~DogHouse() {
    cout << "DogHouse destructor: "
         << *this << endl;
    p->detach();
  }
  void renameHouse(const string& newName) {
    houseName = newName;
  }
  void unalias() { p = p->unalias(); }
  // Copia-su-scrittura. Ogni volta che modifichiamo
  // i contenuti del puntatore
  // bisogna prima chiamare unalias():
  void renameDog(const string& newName) {
    unalias();
    p->rename(newName);
  }
  // ... o quando permettiamo l'accesso a qualcun altro:
  Dog* getDog() {
    unalias();
    return p;
  }
  friend ostream&
  operator<<(ostream& os, const DogHouse& dh) {
    return os << "[" << dh.houseName
      << "] contains " << *dh.p;
  }
};

int main() {
```

```
  DogHouse
    fidos(Dog::make("Fido"), "FidoHouse"),
    spots(Dog::make("Spot"), "SpotHouse");
  cout << "Entering copy-construction" << endl;
  DogHouse bobs(fidos);
  cout << "After copy-constructing bobs" << endl;
  cout << "fidos:" << fidos << endl;
  cout << "spots:" << spots << endl;
  cout << "bobs:" << bobs << endl;
  cout << "Entering spots = fidos" << endl;
  spots = fidos;
  cout << "After spots = fidos" << endl;
  cout << "spots:" << spots << endl;
  cout << "Entering auto-assegnamento" << endl;
  bobs = bobs;
  cout << "After auto-assegnamento" << endl;
  cout << "bobs:" << bobs << endl;
  // Commentare le seguenti linee:
  cout << "Entering rename(\"Bob\")" << endl;
  bobs.getDog()->rename("Bob");
  cout << "After rename(\"Bob\")" << endl;
} ///:~
```

La classe **Dog** è l'oggetto puntato da **DogHouse**. Essa contiene un contatore di riferimenti e delle funzioni per controllare e leggere questo contatore. C'è un costruttore di copia in modo che possiamo costruire un nuovo oggetto **Dog** da uno esistente.

La funzione **attach( )** incrementa il contatore dei riferimennti di un oggetto **Dog** per indicare che c'è un altro oggetto che lo usa. **detach( )** decrementa il contatore dei riferimenti. Se il contatore arriva a zero allora nessuno lo sta usando più, così la funzione membro distrugge il suo oggetto chiamando **delete this**.

Prima di effettuare delle modifiche (come rinominare un **Dog**), bisogna assicurarsi che non si sta modificando un **Dog** che qualche altro oggetto sta usando. Si ottiene ciò chiamando **DogHouse::unalias( )**, che a sua volta chiama **Dog::unalias( )**. L'ultima funzione restituisce il puntatore al **Dog** esistente se il contatore dei riferimenti è uno (che significa che nessun altro sta puntando a questo **Dog**), ma duplicherà il **Dog** se il contatore dei riferimenti è maggiore di uno.

Il costruttore di copia, invece di creare la sua propria memoria, assegna **Dog** al **Dog** dell'oggetto sorgente. Quindi, siccome adesso c'è un oggetto in più che usa questo blocco di memoria, esso incrementa il contatore dei riferimenti chiamando **Dog::attach( )**.

L' **operator=** ha a che fare con un oggetto che è stato già creato sul lato sinistro del segno =, perciò deve prima pulire a fondo questo oggeto chiamando **detach( )** per l'oggetto **Dog**, che distrugge il vecchio **Dog** se nessun altro lo sta usando. Quindi l' **operator=** ripete il comportamento del costruttore di copia. Notare che esso controlla prima se stiamo assegnando l'oggetto a se stesso.

Il distruttore chiama **detach( )** per distruggere l'oggetto **Dog,** se ci sono le condizioni.

Per implementare la copia-su-scrittura, bisogna controllare tutte le azioni che scrivono il nostro blocco di memoria. Per esempio, la funzione membro **renameDog( )** ci permette di cambiare i valori nel blocco di memoria. Ma prima usa **unalias( )** per prevenire la modifica di un **Dog** "replicato"(un **Dog** con più di un oggetto **DogHouse** che punta ad

esso). E se abbiamo bisogno di produrre un puntatore a un **Dog** dall'interno di **DogHouse**, bisogna dapprima chiamare **unalias( )** per questo puntatore.

main( ) testa le varie funzioni che devono lavorare correttamente per implementare il conteggio dei riferimenti: il costruttore, il costruttore di copia, l'**operator=** e il distruttore. Esso controlla anche la copia-suscrittura chiamando **renameDog( )**.

Qui c'è l'output (dopo qualche piccola riformattazione):

```
Creating Dog: [Fido], rc = 1
Created DogHouse: [FidoHouse]
  contains [Fido], rc = 1
Creating Dog: [Spot], rc = 1
Created DogHouse: [SpotHouse]
  contains [Spot], rc = 1
Entering copy-construction
Attached Dog: [Fido], rc = 2
DogHouse copy-constructor:
  [copy-constructed FidoHouse]
    contains [Fido], rc = 2
After copy-constructing bobs
fidos:[FidoHouse] contains [Fido], rc = 2
spots:[SpotHouse] contains [Spot], rc = 1
bobs:[copy-constructed FidoHouse]
  contains [Fido], rc = 2
Entering spots = fidos
Detaching Dog: [Spot], rc = 1
Deleting Dog: [Spot], rc = 0
Attached Dog: [Fido], rc = 3
DogHouse operator= : [FidoHouse assigned]
  contains [Fido], rc = 3
After spots = fidos
spots:[FidoHouse assigned] contains [Fido],rc = 3
Entering auto-assegnamento
DogHouse operator= : [copy-constructed FidoHouse]
  contains [Fido], rc = 3
After auto-assegnamento
bobs:[copy-constructed FidoHouse]
  contains [Fido], rc = 3
Entering rename("Bob")
After rename("Bob")
DogHouse destructor: [copy-constructed FidoHouse]
  contains [Fido], rc = 3
Detaching Dog: [Fido], rc = 3
DogHouse destructor: [FidoHouse assigned]
  contains [Fido], rc = 2
Detaching Dog: [Fido], rc = 2
DogHouse destructor: [FidoHouse]
  contains [Fido], rc = 1
Detaching Dog: [Fido], rc = 1
Deleting Dog: [Fido], rc = 0
```

Studiando l' output, scorrendo il codice sorgente e facendo esperimenti con il programma, si può affinare la conoscenza di queste tecniche.

Creazione automatica dell' operator=

Siccome assegnare un oggetto ad un altro oggetto *dello stesso tipo* è un'attività che molti si aspettano che sia possibile, il compilatore crea automaticamente **type::operator=(type)**

se non se ne fornisce uno. Il comportamento di questo operatore imita quello del costruttore di copia creato automaticamente dal compilatore; se una classe contiene oggetti (o è ereditata da un'altra classe), l' **operator=** per questi oggetti viene chiamato ricorsivamente. Questo viene detto *assegnamento per membro*. Per esempio,

```
//: C12:AutomaticOperatorEquals.cpp
#include <iostream>
using namespace std;

class Cargo {
public:
  Cargo& operator=(const Cargo&) {
    cout << "inside Cargo::operator=()" << endl;
    return *this;
  }
};

class Truck {
  Cargo b;
};

int main() {
  Truck a, b;
  a = b; // Stampa: "inside Cargo::operator=()"
} ///:~
```

L' **operator=** generato automaticamente per **Truck** chiama **Cargo::operator=**.

In generale non vogliamo che il compilatore faccia questo per noi. Con classi di una certa complessità (specialmente se contengono puntatori!) è preferibile creare esplicitamente l' **operator=**. Se non vogliamo che gli utenti della nostra classe effettuino l'assegnamento, dichiariamo l' **operator=** come funzione **private** (non è necessario definirla a meno che non la stiamo usando nella nostra classe).

Conversione automatica di tipo

In C e C++, se il compilatore vede un'espressione o una chiamata a funzione che usano un tipo che non è proprio quello richiesto, spesso può effettuare una conversione di tipo. In C++, possiamo ottenere lo stesso effetto per i tipi definiti dall'utente, definendo delle funzioni di conversione automatica dei tipi. Queste funzioni si presentano in due forme: come tipo particolare di costruttore o come operatore sovraccaricato.

Conversione con costruttore

Se si definisce un costruttore che prende come unico argomento un oggetto (o riferimento) di un altro tipo, questo permette al compilatore di effettuare una conversione automatica di tipo. Per esempio,

```
//: C12:AutomaticTypeConversion.cpp
// Costruttore per la conversione di tipo
class One {
public:
  One() {}
};

class Two {
```

```
public:
  Two(const One&) {}
};

void f(Two) {}

int main() {
  One one;
  f(one); // Si aspetta un Two, riceve un One
} ///:~
```

Quando il compilatore vede la funzione **f( )** chiamata con l'oggetto **One**, guarda la dichiarazione di **f( )** e si accorge che il tipo richiesto è **Two**. Quindi vede se c'è un modo di ricavare **Two** da **One**, trova il costruttore **Two::Two(One)** e silenziosamente lo chiama. L'oggetto risultante di tipo **Two** viene passato ad **f( )**.

In questo caso la conversione automatica di tipo ci ha risparmiato l'onere di definire due versioni sovraccaricate di **f( )**. Tuttavia il costo è una chiamata nascosta a un costruttore di **Two**, che può importare se si è interessati all'efficienza delle chiamate ad **f( )**.

Prevenire la conversione con costruttore

Ci sono dei casi in cui la conversione automatica con costruttore può causare problemi. Per disattivare questa conversione si può modificare il costruttore, facendolo precedere dalla parola chiave **explicit** (che funziona solo con i costruttori). Usata per modificare il costruttore della classe **Two** nell'esempio di sopra, si ha:

```
//: C12:ExplicitKeyword.cpp
// Usando la parola chiave "explicit"
class One {
public:
  One() {}
};

class Two {
public:
  explicit Two(const One&) {}
};

void f(Two) {}

int main() {
  One one;
//!  f(one); // Nessuna autoconversione permessa
  f(Two(one)); // OK -- l'utente effettua la conversione
} ///:~
```

Rendendo **explicit** il costruttore di **Two**, si dice al compilatore di non effettuare nessuna conversione automatica usando questo particolare costruttore (altri costruttori non-**explicit** nella stessa classe possono comunque effetture la conversione automatica). Se l'utilizzatore vuole che la conversione avvenga, deve scrivere del codice fuori dalla classe Nel codice di sopra, **f(Two(one))** crea un oggetto temporaneo di tipo **Two** da **one**, proprio come ha fatto il compilatore nella versione precedente dell'esempio.

Conversione con operatore

Il secondo modo per ottenere la conversione automatica è attraverso un operatore sovraccaricato. Si può creare una funzione membro che prende il tipo corrente e lo converte in quello desiderato usando la parola chiave **operator** seguita dal tipo verso cui vogliamo fare la conversione. Questa forma di sovraccaricamento di operatore è unica, in quanto si vede specificato il valore di ritorno – il valore di ritorno è il *nome* dell'operatore che stiamo sovraccaricando. Qui c'è un esempio:

```
//: C12:OperatorOverloadingConversion.cpp
class Three {
  int i;
public:
  Three(int ii = 0, int = 0) : i(ii) {}
};

class Four {
  int x;
public:
  Four(int xx) : x(xx) {}
  operator Three() const { return Three(x); }
};

void g(Three) {}

int main() {
  Four four(1);
  g(four);
  g(1);  // Chiama Three(1,0)
} ///:~
```

Con la tecnica del costruttore è la classe di destinazione che effettua la conversione, mentre con la tecnica dell'operatore è la classe sorgente a fare la conversione. Il pregio della tecnica del costruttore è che si possono aggiungere nuovi percorsi di conversione ad un sistema esistente quando si crea una nuova classe. Tuttavia creando un costruttore con un solo argomento si definisce *sempre* una conversione automatica di tipo (anche nel caso di più argomenti, se agli altri vengono lasciati i valori di default), che potrebbe non essere la cosa che si vuole (in qual caso si può disattivare la conversione con **explicit**). In più, non c'è la possibilità di usare la conversione con costruttore da un tipo definito dall'utente ad un tipo predefinito; questo è possibile solo con il sovraccaricamento di operatore.

Riflessività

Uno dei principali motivi di convenienza nell'usare operatori globali sovraccaricati invece di operatori membri di classe è che nelle versioni globali la conversione automatica di tipo può essere applicata ad entrambi gli operandi, mentre con gli oggetti membro l'operando di sinistra deve essere sempre del tipo giusto. Se si vuole che entrambi gli operandi siano convertiti, la versione globale fa risparmiare un sacco di codice. Qui c'è un piccolo esempio:

```
//: C12:ReflexivityInOverloading.cpp
class Number {
  int i;
public:
  Number(int ii = 0) : i(ii) {}
  const Number
  operator+(const Number& n) const {
```

```
    return Number(i + n.i);
  }
  friend const Number
    operator-(const Number&, const Number&);
};

const Number
  operator-(const Number& n1,
            const Number& n2) {
    return Number(n1.i - n2.i);
}

int main() {
  Number a(47), b(11);
  a + b; // OK
  a + 1; // 2nd argomento convertito a Number
//! 1 + a; // Sbagliato! il primo argomento non è di tipo Number
  a - b; // OK
  a - 1; // 2nd argomento convertito a Number
  1 - a; // 1mo argomento convertito a Number
} ///:~
```

La classe **Number** ha sia un **operator+** come membro, sia un **friend operator–**. Siccome c'è un costruttore che prende come unico argomento un **int**, un **int** può essere automaticamente convertito a **Number**, ma solo alle giuste condizioni. In **main( )**, possiamo vedere che aggiungere un **Number** ad un altro **Number** funziona bene perchè c'è una corrispondenza esatta con l'operatore sovraccaricato. Quando il compilatore vede un **Number** seguito da un **+** e da un int **int**, può trovare la corrispondenza con la funzione membro **Number::operator+** e convertire l'argomento **int** a **Number** usando il costruttore. Ma quando vede un **int,** un **+** e un **Number**, non sa cosa fare perchè tutto quello che ha è **Number::operator+**, che richiede che l'operando di sinistra sia già un oggetto di tipo **Number**. Cosicchè il compilatore produce un errore.

Con il **friend operator–**, le cose sono diverse. Il compilatore ha bisogno di inserire entrambi gli argomenti, per quanto può; non è costretto ad avere un tipo **Number** come argomento sul lato sinistro. Così, se vede

```
1 - a
```

può convertire il primo argomento a **Number** usando il costruttore.

A volte si vuole limitare l'uso dei propri operatori, rendendoli membri di classe. Per esempio, quando si moltiplica una matrice per un vettore, il vettore deve stare sulla destra. Ma se si vuole che gli operatori convertano entrambi gli argomenti, bisogna definire l'operatore come funzione friend.

Fortunatamente il compilatore se vede **1 – 1** non converte gli argomenti a oggetti di tipo **Number** per poi chiamare l' **operator–**. Questo significherebbe che il codice C esistente si troverebbe improvvisamente a funzionare in maniera diversa. Il compilatore cerca dapprima il confronto più "semplice", che è l'operator predefinito nell'espressione **1 – 1**.

Esempio di conversione di tipo

Un esempio in cui la conversione automatica di tipo è estremamente utile accade con qualsiasi classe che incapsula stringhe di caratteri (in questo caso possiamo implementare la classe usando semplicemente la classe **string** del C++ Standard, perchè è semplice). Senza la conversione automatica di tipo, se si vogliono usare tutte le funzioni per la manipolazione delle stringhe presenti nella libreria Standard del C, bisogna creare una funzione membro per ognuna di esse, come questa:

```
//: C12:Strings1.cpp
// Nessuna autoconversione di tipo
#include "../require.h"
#include <cstring>
#include <cstdlib>
#include <string>
using namespace std;

class Stringc {
  string s;
public:
  Stringc(const string& str = "") : s(str) {}
  int strcmp(const Stringc& S) const {
    return ::strcmp(s.c_str(), S.s.c_str());
  }
  // ... ecc., per ogni funzione in string.h
};

int main() {
  Stringc s1("hello"), s2("there");
  s1.strcmp(s2);
} ///:~
```

Qui viene creata solo la funzione **strcmp( )**, ma si può creare la funzione corrispondente per ognuna di quelle presenti in **<cstring>,** se necessario. Fortunatamente possiamo fornire una conversione automatica di tipo permettendo l'accesso a tutte le funzioni in **<cstring>**:

```
//: C12:Strings2.cpp
// Con auto conversione di tipo
#include "../require.h"
#include <cstring>
#include <cstdlib>
#include <string>
using namespace std;

class Stringc {
  string s;
public:
  Stringc(const string& str = "") : s(str) {}
  operator const char*() const {
    return s.c_str();
  }
};

int main() {
  Stringc s1("hello"), s2("there");
  strcmp(s1, s2); // Funzione C Standard
  strspn(s1, s2); // Qualunque funzione su string!
} ///:~
```

Ora qualunque funzione che prende un argomento di tipo **char*** può prendere anche un argomento di tipo **Stringc** perchè il compilatore sa come costruire un **char*** a partire da una **Stringc**.

Trappole nella conversione automatica di tipo

Siccome il compilatore deve scegliere il modo in cui effettuare silenziosamente la conversione, può cadere in errore se non la progettiamo correttamente. Una situazione semplice ed ovvia si presenta quando una classe **X** può convertire se stessa in un oggetto di tipo **Y** con un **operator Y( )**. Se la classe **Y** ha un costruttore che prende un unico argomento di tipo **X**, questo rappresenta lo stesso tipo di conversione. Il compilatore ha ora due modi di effettuare la conversione da **X** a **Y**, e si può creare un'ambiguità :

```
//: C12:TypeConversionAmbiguity.cpp
class Orange; // Dichiarazione di Classe

class Apple {
public:
  operator Orange() const; // Converte Apple in Orange
};

class Orange {
public:
  Orange(Apple); // Converte Apple in Orange
};

void f(Orange) {}

int main() {
  Apple a;
//! f(a); // Errore: conversione ambigua
} ///:~
```

La soluzione ovvia al problema è di non fare ciò. Bisogna fornire una sola possibilità di conversione automatica da un tipo ad un altro.

Un problema molto più difficile da individuare accade quando si fornisce una conversione automatica verso più tipi. Questa viene detta *fan-out*:

```
//: C12:TypeConversionFanout.cpp
class Orange {};
class Pear {};

class Apple {
public:
  operator Orange() const;
  operator Pear() const;
};

//eat() sovraccaricata:
void eat(Orange);
void eat(Pear);

int main() {
  Apple c;
//! eat(c);
  // Errore: Apple -> Orange o Apple -> Pear ???
} ///:~
```

La classe **Apple** ha una conversione automatica sia verso **Orange** che verso **Pear**. La cosa insidiosa rispetto a questo è che non c'è nessun problema fino a quando qualcuno non crea due versioni sovraccaricate della funzione **eat( )** (con una sola versione il codice in **main( )** funziona bene).

Di nuovo, la soluzione – e la parola d'ordine generale con la conversione automatica dei tipi – è quella di fornire una singola conversione automatica da un tipo a un altro. Si possono avere conversioni verso altri tipi; ma queste possono semplicemente essere non *automatiche*. Si possono definire chiamate a funzioni esplicite con nomi come **makeA( )** e **makeB( )**.

Attività nascoste

La conversione automatica dei tipi può introdurre molte più attività sottostanti di quante ci si possa aspettare. Come piccolo esercizio mentale, osserviamo questa modifica al file **CopyingVsInitialization.cpp**:

```
//: C12:CopyingVsInitialization2.cpp
class Fi {};

class Fee {
public:
  Fee(int) {}
  Fee(const Fi&) {}
};

class Fo {
  int i;
public:
  Fo(int x = 0) : i(x) {}
  operator Fee() const { return Fee(i); }
};

int main() {
  Fo fo;
  Fee fee = fo;
} ///:~
```

Non c'è nessun costruttore per creare l'oggetto **Fee fee** dall'oggetto **Fo**. Tuttavia, **Fo** ha una funzione di conversione automatica verso **Fee**. Non c'è nessun costruttore di copia per creare un oggetto di tipo **Fee** da un altro oggetto di tipo **Fee**, ma questa è una delle funzioni speciali che il compilatore è in grado di creare per noi (il costruttore di default, il costruttore di copia, l'**operator=**, e il distruttore possono essere automaticamente sintetizzati dal compilatore). Cosicchè per l'istruzione relativamente innocua

```
Fee fee = fo;
```

viene chiamato l' operatore di conversione di tipo e viene creato un costruttore di copia.

Usare la conversione automatica di tipo con attenzione. Come per ogni sovraccaricamento di operatore, è eccellente quando riduce significativamente il codice da scrivere, ma non è meritevole di un uso gratuito.

Sommario

La ragione dell'esistenza del sovraccaricamento degli operatori è per quelle situazioni in cui questo rende la vita più facile. Non c'è niente di particolarmente magico riguardo a ciò; gli operatori sovraccaricati sono semplicemente delle funzioni con nomi simpatici, e le chiamate a queste funzioni vengono fatte dal compilatore quando questo riconosce la giusta corrispondenza. Ma se il sovraccaricamento non fornisce un significativo beneficio al creatore della classe o all'utente della stessa, è inutile complicare le cose con la sua aggiunta.

---

[49] Rob Murray, *C++ Strategies & Tactics*, Addison-Wesley, 1993, pagina 47.

13: Creazione dinamica di oggetti

A volte si conosce l'esatta quantità, tipo e tempo di vita degli oggetti del nostro programma. Ma non sempre.

Quanti aeroplani gestirà un sistema di traffico aereo? Quante figure userà un sistema CAD? Quanti nodi ci saranno in una rete?

Per risolvere il problema della programmazione generale, è essenziale che si possa creare e distruggere gli oggetti a tempo di esecuzione. Naturalmente, il C ha sempre fornito funzioni per l'*allocazione dinamica della memoria* **malloc()** e **free()** (insieme con le varianti di **malloc()**) che allocano memoria nella heap (anche detta memoria libera) a runtime.

Tuttavia, queste non funzioneranno in C++. Il costruttore non permette di maneggiare l'indirizzo della memoria da inizializzare e per una buona ragione. Se lo si potesse fare, si potrebbe:

1. Dimenticare. L'inizializzazione garantita degli oggetti in C++ non sarebbe tale.
2. Accidentalmente fare qualcosa all'oggetto prima di inizializzarlo, aspettandosi che accada la cosa giusta.
3. Maneggiare l'oggetto di dimensioni sbagliate.

E naturalmente, persino se si è fatto tutto correttamente, chiunque modificasse il nostro programma è incline agli stessi errori. L'impropria inizializzazione è responsabile di un gran numero di problemi della programmazione, perciò è molto importante garantire le chiamate ai costruttori per gli oggetti creati nella heap.

Perciò come fa il C++ a garantire una corretta inizializzazione e pulizia, ma permettere di creare oggetti dinamicamente nella heap?

La risposta è di portare la creazione dinamica nel nucleo del linguaggio **malloc( )** e **free()** sono funzioni di libreria e così fuori dal controllo del compilatore. Tuttavia, se si ha un operatore per eseguire l'atto combinato dell'allocazione dinamica della memoria e l'inizializzazione ed un altro operatore per eseguire l'atto combinato di pulizia e rilascio della memoria, il compilatore può ancora garantire che i costruttori e distruttori saranno chiamati per tutti gli oggetti.

In questo capitolo, si imparerà come le funzioni **new** e **delete** del C++ elegantemente risolvono questo problema, creando in modo sicuro gli oggetti nella heap.

Creazione dell'oggetto

Quando un oggetto C++ viene creato accadono due eventi:

1. La memoria è allocata per l'oggetto.
2. Il costruttore viene chiamato per inizializzare la memoria.

Per ora si dovrebbe credere che il passo due ci sia sempre. Il C++ lo impone perchè oggetti non inizializzati sono la maggior fonte di bachi. Non importa dove o come l'oggetto viene creato, il costruttore viene sempre chiamato.

Tuttavia, può succedere che il passo uno venga attuato in diversi modi o in tempi alterni:

1. La memoria può essere allocata nell'area di memoria statica prima che il programma cominci. Questa memoria esiste durante tutta la vita del programma.
2. Lo spazio può essere creato nello stack ogni volta che viene raggiunto un particolare punto di esecuzione (una parentesi aperta). Questo spazio viene rilasciato automaticamente in un punto complementare di esecuzione (la parentesi di chiusura). Queste operazioni di allocazione nello stack vengono eseguite con le istruzioni proprie del processore e sono molto efficienti. Tuttavia, si deve sapere esattamente di quante variabili si ha bisogno quando si scrive il programma in modo che il compilatore possa generare il giusto codice.
3. Lo spazio può essere allocato da una zona di memoria detta heap (anche conosciuta come memoria libera). Ciò viene detta allocazione dinamica della memoria. Per allocare questa memoria, viene chiamata una funzione a run time (tempo di esecuzione); ciò significa che si può decidere in qualsiasi momento che si vuole della memoria e quanta se ne vuole. Si è anche responsabili del rilascio della memoria, ciò significa che il tempo di vita della memoria è a piacere e non dipende dallo scope.

Spesso queste tre regioni sono piazzate in un singolo pezzo contiguo di memoria fisica: l'area statica, lo stack e la heap (in un ordine determinato dal progettista del compilatore). Tuttavia, non ci sono regole. Lo stack può essere in un posto speciale e la heap può essere implementato facendo chiamate a pezzi di memoria dal sistema operativo. Per un programmatore, queste cose sono normalmente nascoste, quindi tutto ciò che bisogna sapere è che la memoria è disponibile quando serve.

L'approccio del C alla heap

Per allocare memoria dinamicamente a runtime, il C fornisce funzioni nella sua libreria standard: **malloc()** e le sue varianti **calloc()** e **realloc()** per allocare memoria nella heap, **free()** per rilasciare la memoria alla heap. Queste funzioni sono pragmatiche ma primitive e richiedono comprensione ed attenzione da parte del programmatore. Per creare un'istanza di una classe nella heap usando le funzioni della memoria dinamica del C, si dovrebbe fare qualcosa del genere:

```
//: C13:MallocClass.cpp
// Malloc con oggetti di classi
// Cosa si deve fare se non      si vuole usare "new"
#include      "../require.h"
#include      <cstdlib> // malloc() & free()
#include      <cstring> // memset()
#include      <iostream>
using namespace std;
class Oggetto {
       int i, j, k;
       enum { sz = 100 };
       char buf[sz];
public:
  void inizializza() { //      non si può usare il costruttore
    cout      << "inizializzo Oggetto" << endl;
    i      = j = k = 0;
```

```
    memset(buf, 0, sz);
      }
  void distruggi() const      { // Non si può usare il distruttore
    cout      << "distruggo Oggetto" << endl;
  }
};
int main() {
  Oggetto*      oggetto = (Oggetto*)malloc(sizeof(Oggetto));
       require(oggetto != 0);
  oggetto->inizializza();
  // ... un pò dopo:
  oggetto->distruggi();
  free(oggetto);
} ///:~
```

Si può vedere l'uso di **malloc()** per creare spazio per l'oggetto nella linea:

```
Oggetto* oggetto = (Oggetto*)malloc(sizeof(Oggetto));
```

Qui, l'utente deve determinare le dimensioni dell'oggetto (un posto per un errore). **malloc()** restituisce un **void*** perchè produce solo una porzione di memoria, non un oggetto. Il C++ non permette di assegnare un puntatore **void*** ad un qualsiasi altro puntatore, perciò bisogna effettuare un cast.

Poichè **malloc()** può fallire nel ricercare memoria libera (in questo caso restituisce zero), si deve controllare il puntatore restituito per essere sicuri che tutto sia andato bene.

Ma il problema peggiore è questa linea:

```
oggetto->inizializza();
```

Gli utenti devono ricordasi di inizializzare l'oggetto prima di usarlo. Si noti che non è stato usato un costruttore perchè il costruttore non può essere chiamato esplicitamente[50] , esso viene chiamato per noi dal compilatore quando viene creato un oggetto . Il problema qui è che l'utente ora ha la possibilità di dimenticare di eseguire l'inizializzazione prima che l'oggetto venga utilizzato, introducendo così una grossa sorgente di bachi.

Molti programmatori trovano che le funzioni per l'allocazione dinamica della memoria in C sono molto complicate e fonte di confusione; è comune trovare programmatori C che usano macchine di memoria virtuale che allocano enormi array di variabili nell'area statica di memoria per evitare di pensare all'allocazione dinamica della memoria. Poichè il C++ sta cercando di rendere sicure e facili da usare le librerie per il programmatore inesperto, l'approccio del C alla memoria dinamica è inaccettabile.

operatore new

La soluzione del C++ è quella di combinare tutte le azioni necessarie per creare un oggetto in un singolo operatore chiamato **new**. Quando si crea un oggetto con **new**, esso alloca sufficiente memoria nella heap per contenere l'oggetto e chiama il costruttore per quella memoria. Cosi , se si scrive:

```
MioTipo *fp = new MioTipo(1,2);
```

a runtime, viene chiamato l'equivalente di **malloc(sizeof(MioTipo))** (spesso, è una chiamata a **malloc()**) ed il costruttore per **MioTipo** viene chiamato con l'indirizzo risultante come il puntatore **this**, usando **(1,2)** come lista argomenti. Il puntatore è poi assegnato a **fp**, esso è un oggetto istanziato ed inizializzato, non ci si può mettere le mani sopra prima di allora. Esso è anche il giusto tipo **MioTipo** perciò non è necessario il cast.

La **new** di default controlla se l'allocazione della memoria è avvenuta con successo prima di passare l'indirizzo al costruttore, perciò non si deve esplicitamente determinare se la chiamata ha avuto successo. Più avanti nel capitolo si scoprirà cosa accade se non c'è più memoria libera.

Si può creare una nuova espressione usando qualsiasi costruttore disponibile per la classe. Se il costruttore non ha argomenti, si scrive l'espressione new senza la lista di argomenti del costruttore:

```
MioTipo *fp = new MioTipo;
```

Si noti come il processo di creazione degli oggetti nella heap diventi semplicemente una singola espressione, con tutti i controlli sulla dimensione, conversione e sicurezza integrati dentro. È facile creare un oggetto nella heap come se lo si facesse nello stack.

operatore delete

Il complemento alla espressione new è l'espressione delete, che per prima chiama il distruttore e poi libera la memoria (spesso con una chiamata alla **free()**). Proprio come una espressione new ritorna un puntatore all'oggetto, un' espressione di delete richiede l'indirizzo di un oggetto.

```
delete fp;
```

Ciò distrugge e poi libera la memoria allocata per l'oggetto **MioTipo** creato dinamicamente precedentemente.

delete può essere chiamata solo per un oggetto creato con **new**. Se si usa **malloc( )** (o **calloc( )** o **realloc( )**) e poi **delete**, il comportamento non è definito. Poichè molte implementazioni di **new** e **delete** usano **malloc()** e **free()**, si terminerà con il rilascio della memoria senza chiamare il distruttore.

Se il puntatore che si sta cancellando è zero, non accadrà niente. Per questo motivo, spesso si raccomanda di impostare il puntatore a zero immediatamente dopo che lo si è cancellato, in modo da prevenire la cancellazione doppia. Cancellare un oggetto più di una volta è una cosa completamente sbagliata e causa problemi.

Un semplice esempio

Questo semplice esempio mostra come avviene l'inizializzazione:

```
//: C13:Tree.h
#ifndef      TREE_H
#define      TREE_H
#include      <iostream>
class Albero {
  int altezza;
public:
       Albero(int altezzaAlbero) : altezza(altezzaAlbero) {}
  ~Albero()      { std::cout << "*"; }
       friend std::ostream&
       operator<<(std::ostream& os, const Albero* t) {
         return os << "      altezza Albero:"
                   << t->altezza<< std::endl;
       }
};
#endif      // TREE_H ///:~
//: C13:NewAndDelete.cpp
// semplice demo di new &      delete
#include      "Tree.h"
using namespace std;
int main() {
  Albero* t      = new Albero(40);
       cout << t;
  delete t;
} ///:~
```

La prova che il costruttore viene chiamato è data dalla stampa del valore di **Albero**. Qui , viene fatto con l'overloading dell'operatore << per usarlo con un **ostream** ed un **Albero***. Si noti, tuttavia, che anche se la funzione è dichiarata come un **friend**, essa è definita come una inline! Ciò è per pura convenienza, definire una funzione **friend** come inline per una classe non cambia lo stato di **friend** o il fatto che essa è una funzione globale e non una funzione membro della classe. Si noti anche che il valore di ritorno è il risultato di un' intera espressione, che è un **ostream&** (così deve essere, per soddisfare il tipo del valore di ritorno della funzione).

Overhead del manager della memoria

Quando si creano oggetti automatici nello stack, la dimensione degli oggetti ed il loro tempo di vita è costruito direttamente nel codice generato, perchè il compilatore conosce esattamente tipo, quantità e visibilità. Creare oggetti nella heap implica maggior overhead, sia come tempo che come spazio. Ecco un tipico scenario. (Si può rimpiazzare **malloc()** con **calloc()** o **realloc ()**)

Si chiama **malloc()**, che richiede un blocco di memoria libera. (Questo codice potrebbe essere in realtà parte di **malloc()**)

Viene cercato un blocco di memoria libera grande abbastanza per soddisfare la richiesta. Ciò avviene controllando una mappa che mostra quali blocchi sono correntemente in uso e quali sono disponibili. È un processo rapido, ma può richiedere più di un tentativo quindi potrebbe non essere deterministico, cioè non si può pensare che **malloc()** richieda sempre la stessa quantità di tempo.

Prima che un puntatore al blocco venga restituito, la dimensione e la locazione del blocco deve essere memorizzato in modo che successive chiamate di **malloc()** non lo usino e che quando si chiama **free(),** il sistema sappia quanta memoria liberare.

Il modo in cui tutto ciò viene implementato può variare enormemente. Per esempio, non c'è niente che impedisce che le primitive per l'allocazione della memoria siano implementate nel processore. Se si è curiosi, si può scrivere dei programmi di test per provare a indovinare il modo in cui **malloc()** è implementata. Si può anche leggere la libreria del codice sorgente, se la si ha (I sorgenti del C GNU sono sempre disponibili).

I primi esempi ridisegnati

Utilizzando **new** e **delete**, l'esempio **Stash** introdotto precedentemente in questo libro può essere riscritto usando tutte le caratteristiche discusse nel libro finora. Esaminando il nuovo codice si avrà anche una utile rassegna degli argomenti.

A questo punto del libro, nè la classe **Stash** che la **Stack** possederanno gli oggetti a cui puntano; cioè, quando l'oggetto **Stash** o **Stack** perde visibilità, non chiamerà **delete** per tutti gli oggetti a cui punta. La ragione per cui ciò non è possibile è data dal fatto che, in un tentativo di essere generici, essi gestiscono puntatori **void**. Se si cancella un puntatore **void**, la sola cosa che accade è che la memoria viene rilasciata, perchè non c'è nessuna informazione sul tipo e non c'è modo per il compilatore di sapere quale distruttore chiamare.

delete void* è probabilmente un bug

Vale la pena di chiarire che se si chiama **delete** per un **void***, è quasi certo che si tratta di bug nel nostro programma a meno che la destinazione del puntatore sia molto semplice; in particolare, si dovrebbe non avere un distruttore. Qui c'è un esempio per mostrare cosa accade:

```
//:    C13:BadVoidPointerDeletion.cpp
//    Deleting void pointers can cause memory leaks
#include    <iostream>
 using          namespace    std;
class    Object {
void*    data; // un pò di memoria
const          int    size;
const          char    id;
public:    Object(int sz, char    c) : size(sz), id(c) {
data    = new  char[size];
cout    << "Costruzione oggetto"  <<    id << ", dimensioni = "  <<    size << endl;
}
~Object()    {
cout    << "Distruzione oggetto"  <<    id << endl;
delete    []data; // OK, rilascio solo la memoria
 //    non è necessaria la chiamata al distruttore
}
};
int    main() {
Object*    a = new Object(40, 'a');
delete    a;
void*    b = new Object(40, 'b');
delete    b;
} ///:~
```

La classe **Object** contiene un **void*** che è inizializzato da dati "grezzi" (non punta ad oggetti che hanno un distruttore). Nel distruttore di **Object**, delete viene chiamato per questo **void*** senza effetti indesiderati, poichè l'unica cosa di cui abbiamo bisogno che accada è che sia rilasciata la memoria.

Tuttavia, nel **main()** si può vedere che è indispensabile che **delete** sappia con che tipo di oggetto sta lavorando. Questo è l'output:

```
Costruzione oggetto a, dimensione = 40
Distruzione oggetto a
Costruzione oggetto b, dimensione = 40
```

Poichè **delete a** sa che **a** punta ad un **Object**, il distruttore viene chiamato e così la memoria allocata per **data** viene liberata. Tuttavia, se si manipola un oggetto tramite un **void*** come nel caso di **delete b**, l'unica cosa che accade è che la memoria di **Object** viene liberata, ma il distruttore non viene chiamato quindi non c'è memoria rilasciata a cui **data** punta. Quando questo programma viene compilato, probabilmente non si vedranno messaggi di warning; il compilatore assume che si sappia cosa si sta facendo. Si ottiene quello che in gergo viene detto un memory leak (una falla di memoria).

Se si ha un memory leak nel programma, si ricerchino tutti i **delete** controllando il tipo di puntatore che si sta cancellando. Se è un **void*** allora si è trovata probabilmente una sorgente del memory leak (tuttavia il C++ fornisce altre ampie opportunità per i memory leak).

La responsabilità della pulizia con i puntatori

Per rendere flessibili i contenitori **Stash** e **Stack** (capaci di gestire qualsiasi tipo di oggetto), essi gestiranno puntatori **void**. Ciò significa che quanto viene restituito un puntatore dall'oggetto **Stash** o **Stack**, si deve fare il cast di esso al tipo opportuno prima di usarlo; come abbiamo visto sopra, si deve usare il cast verso il tipo opportuno prima di cancellarlo o si avrà un memory leak.

Gli altri casi di memory leak hanno a che fare con la sicurezza che quel **delete** è realmente chiamato per ogni puntatore ad oggetto gestito nel contenitore. Il contenitore non può possedere un puntatore perchè lo gestisce come un **void*** e quindi non può eseguire la corretta pulizia. L'utente ha la responsabilità della pulizia degli oggetti. Ciò produce una serie di problemi se si aggiungono puntatori agli oggetti creati nello stack ed oggetti creati nella heap dallo stesso contenitore perchè l'espressione delete non è sicura per un puntatore che non era stato allocato nella heap (e quando si prende un puntatore dal container, come facciamo a sapere dove è stato allocato il suo oggetto?). Quindi, si deve essere sicuri che gli oggetti memorizzati nelle versioni che seguono di **Stash** e **Stack** vengano fatti solo nella heap, sia tramite una attenta programmazione o creando classi che possono solo essere allocate nella heap.

È anche importante essere sicuri che il programmatore client si prenda la responsabilità di pulizia di tutti i puntatori nel container. Si è visto negli esempi precedenti come la classe **Stack** controlla nel suo distruttore che tutti gli oggetti **Link** siano stati poppati. Per uno **Stash** di puntatori, tuttavia, c'è bisogno di un altro approccio.

Stash per puntatori

Questa nuova versione della classe **Stash**, chiamato **PStash** gestisce puntatori ad oggetti che esistono di per sè nella heap, mentre la vecchia **Stash** nei capitoli precedenti copiava gli oggetti per valore nel container **Stash**. Usando **new** e **delete**, è facile e sicuro gestire puntatori ad oggetti che sono stati creati nella heap.

Qui c'è l'header file per il puntatore Stash:

```
//:      C13:PStash.h
//      Gestisce puntatori invece di oggetti
#ifndef      PSTASH_H
#define      PSTASH_H
class      PStash {
       int quantity; // Numero      di spazio libero
  int next; // prossimo spazio      libero
        // puntatore allo spazio:
       void** storage;
       void inflate(int increase);
public:
       PStash() : quantity(0), storage(0), next(0) {}
       ~PStash();
       int add(void* element);
       void* operator[](int index) const; // accesso
       // rimuove il riferimento da questo PStash:
       void* remove(int      index);
       // Numero di elementi in Stash:
       int count() const      { return next; }
};
#endif      // PSTASH_H ///:~
```

I sottostanti elementi dei dati sono simili, ma ora **storage** è un array di puntatori **void** e l'allocazione dello spazio per questo array è eseguita con **new** invece di **malloc()**. Nell'espressione

```
void** st = new void*[quantity + increase];
```

il tipo di oggetto allocato è un **void***, quindi l'espressione alloca un array di puntatori **void**.

Il distruttore elimina lo spazio dove sono tenuti i puntatori **void** piuttosto che tentare di cancellare ciò che essi puntano (che, come precedentemente notato, rilascerà la loro memoria e non chiamerà il distruttore perchè un puntatore **void** non ha informazione di tipo).

L'altro cambiamento è **operator[]** con **fetch()** , che ha più senso sintatticamente. Di nuovo, tuttavia, viene restituito un **void***, quindi l'utente deve ricordarsi quali tipi sono memorizzati nel container ed eseguire un cast sui puntatori quando li usa (un problema che sarà risolto nei prossimi capitoli).

Qui ci sono le definizioni delle funzioni membro:

```
//:      C13:PStash.cpp {O}
//      definizioni Pointer Stash
#include      "PStash.h"
#include      "../require.h"
#include      <iostream>
#include      <cstring> // funzioni 'mem'
using namespace std;
int PStash::add(void* element) {
       const int inflateSize      = 10;
       if(next>= quantity)
         inflate(inflateSize);
       spazio[next++] = element;
       return(next - 1); // numero      indice
}
// No ownership:
PStash::~PStash()      {
       for(int i = 0; i <      next; i++)
         require(storage[i] == 0,
      "PStash non ripulito");
       delete []storage;
}
//      overloading Operatore  per rimpiazzare       fetch
void* PStash::operator[](int index)      const {
       require(index>= 0,
         "PStash::operator[] indice negativo");
       if(index>= next)
         return 0; // Per indicare      la fine
//      produce un puntatore all'elemento desiderato:
return storage[index];
}
void* PStash::remove(int index) {
       void* v = operator[](index);
       // "Rimuove" il       puntatore:
       if(v != 0) storage[index] = 0;
       return v;
}
void PStash::inflate(int increase) {
       const int psz = sizeof(void*);
       void** st = new void*[quantity + increase];
       memset(st, 0, (quantity + increase) * psz);
       memcpy(st, storage, quantity * psz);
       quantity += increase;
       delete []storage; // Vecchio      spazio
       storage = st; // Punta a memoria nuova
} ///:~
```

La funzione **add()** è la stessa di prima, tranne che è memorizzato un puntatore invece della copia di un intero oggetto.

Il codice **inflate()** è stato modificato per gestire l'allocazione di un array di **void***, mentre nella versione precedente funzionava solo con byte grezzi. Qui, invece di usare l'approccio di copiare con un indice di array, viene usata prima la funzione della libreria Standard C **memset()** per impostare a zero la memoria nuova (ciò non è strettamente necessario, poichè **PStash** sta presumibilmente gestendo la memoria correttamente, ma di solito non fa male un bit di attenzione in più). Poi **memcpy()** muove i dati esistenti dalla vecchia locazione alla nuova. Spesso, funzioni come **memset()** e **memcpy()** sono state ottimizzate nel tempo, quindi possono essere più veloci dei loop mostrati

precedentemente. Ma con una funzione come **inflate()** che probabilmente non sarà usata spesso, si però non vedere questa differenza di performance. Tuttavia, il fatto che le chiamate a funzione sono più concise dei loop possono aiutare a prevenire errori nel codice.

Per addossare la responsabilità della pulizia degli oggetti sulle spalle del programmatore client, ci sono due modi di accedere ai puntatori in **PStash:** l'**operator[],** che semplicemente restituisce il puntatore ma lo lascia come un membro del container ed una seconda funzione membro **remove()**, che restituisce il puntatore ma lo rimuove anche dal container assegnando quella posizione a zero. Quando viene chiamato il distruttore per **PStash**, esso controlla per essere sicuro che tutti i puntatori agli oggetti sono stati rimossi; in caso contrario, si è avvertiti in modo che si può prevenire un memory leak (le soluzioni più eleganti sono presenti negli ultimi capitoli).

Un test

ecco il vecchio programma di test per **Stash** riscritto per **PStash**:

```
//:      C13:PStashTest.cpp
//{L}      PStash
//      Test per puntatore Stash
#include      "PStash.h"
#include      "../require.h"
#include      <iostream>
#include      <fstream>
#include      <string>
using namespace std;
int      main() {
       PStash stashIntero;
       // 'new' funziona anche con tipi predefiniti. Si noti
       // la sintassi "pseudo-costruttore":
       for(int      i = 0; i < 25; i++)
         stashIntero.add(new int(i));
       for(int j = 0; j <      stashIntero.count(); j++)
         cout << "stashIntero["      << j << "] = "
              << *(int*)stashIntero[j]      << endl;
       // pulizia:
       for(int k = 0; k <      stashIntero.count(); k++)
         delete stashIntero.remove(k);
       ifstream in ("PStashTest.cpp");
       assure(in, "PStashTest.cpp");
       PStash stashStringa;
       string linea;
       while(getline(in, linea))
         stashStringa.add(new string(linea));
       // Stampa le stringhe:
       for(int u = 0; stashStringa[u]; u++)
         cout << "stashStringa[" << u << "] = "
              << *(string*)stashStringa[u] << endl;
       // Pulizia:
       for(int      v = 0; v < stashStringa.count(); v++)
         delete (string*)stashStringa.remove(v);
} ///:~
```

Come prima, gli Stash vengono creati ed in essi memorizzata l'informazione, ma quest'ultima è il puntatore che si ottiene dalle **new**. Nel primo caso, si noti la linea:

```
stashIntero.add(new int(i));
```

L'espressione **new int(i)** usa la pseudo forma del costruttore, quindi la memoria per un nuovo oggetto **int** viene creata nella heap e l'**int** è inizializzato al valore **i**.

Durante la stampa, il valore restituito dall'operatore di **PStash::operator[ ]** deve essere castato al tipo opportuno; ciò viene ripetuto per i restanti oggetti **PStash** del programma. Non è un effetto desiderato usare puntatori **void** e ciò verrà aggiustato nei prossimi capitoli.

Il secondo test apre i file sorgenti e legge una linea alla volta dentro un altro **PStash**. Ogni linea viene letta dentro una **string** usando **getline()**, poi una nuova stringa viene creata da **linea** per fare una copia indipendente di quella linea. Se passiamo solo l'indirizzo di **linea** ogni volta, avremo un mucchio di puntatori a **linea**, che conterrebbero solo l'ultima linea che è stata letta dal file.

Quando si usa il puntatore si vede l'espressione:

```
*(string*) stashStringa[v]
```

Il puntatore ritornato dall'operatore **[]** deve essere castato ad un **string*** per aver il giusto tipo. Poi la **string *** è dereferenziata in modo da valutare l'espressione ad un oggetto, al punto in cui il compilatore vede un oggetto **string** e lo manda a **cout**.

L'oggetto creato nella heap deve essere distrutto con l'uso del comando **remove()** o altrimenti si avrà un messaggio a runtime che ci dice che non sono stati puliti completamente gli oggetti in **PStash**. Si noti che nel caso di puntatori a **int**, nessun cast è necessario perchè non c'è distruttore per un **int** e tutto ciò di cui abbiamo bisogno è rilasciare la memoria:

```
delete stashIntero.remove(k);
```

Tuttavia, per i puntatori **string**, se si dimentica di fare il cast si avrà un altro memory leak, quindi il cast è essenziale:

```
delete (string*) stringStash.remove(k);
```

Alcuni di questi problemi (ma non tutti) possono essere eliminati usanto i template (che si impareranno ad usare nel Capitolo 16).

new & delete per gli array

Nel C++, si possono creare array di oggetti nello stack o nella heap con uguale facilità, e (naturalmente) il costruttore viene chiamato per ogni oggetto dell'array. C'è un vincolo, tuttavia: ci deve essere un costruttore di default, tranne per l'inizializzazione dell'aggregato sullo stack (si veda il Capitolo 6), perchè un costruttore con nessun argomento deve essere chiamato per ogni oggetto.

Quando si creano array di oggetti sulla heap usando **new**, c'è qualcosa altro che si deve fare. Un esempio di tale array è

```
MioTipo* fp = new MioTipo[100];
```

Questo alloca sufficiente spazio nella heap per 100 oggetti **MioTipo** e chiama il costruttore per ognuno. Ora, tuttavia, si ha semplicemente un **MioTipo***, che è esattamente lo stesso che si avrebbe se si avesse scritto

```
MioTipo* fp2 = new MioTipo;
```

per creare un singolo oggetto. Poichè noi abbiamo scritto il codice, sappiamo che **fp** è realmente l'indirizzo di partenza di un array, quindi ha senso selezionare gli elementi dell'array usando un espressione tipo **fp[3]**. Ma cosa accade quando si distrugge l'array? Le due righe

```
delete fp2; // OK
delete fp;  // non si ottiene l'effetto desiderato
```

sembrano esattamente le stesse e i loro effetti saranno gli stessi. Il distruttore sarà chiamato per l'oggetto **MioTipo** puntato dall'indirizzo dato e poi la memoria sarà rilasciata. Per **fp2** ciò va bene, ma per **fp** ciò significa che le altre 99 chiamate al distruttore non saranno fatte. L'esatto totale della memoria sarà ancora liberato, tuttavia, poichè è allocato in un grosso pezzo e la dimensione dell'intero pezzo è conservata da qualche parte dalla routine di allocazione.

La soluzione richiede che si dia al compilatore l'informazione che questo è in realtà l'indirizzo di partenza di un array. Questo viene fatto con la seguente sintassi:

```
delete []fp;
```

Le parentesi vuote dicono al compilatore di generare codice che prende il numero di oggetto dell'array, memorizzati da qualche parte quando l'array viene creato e chiama il distruttore tante volte quanti sono gli oggetti. Questa è realmente una sintassi migliorata dalla forma precedente, che si può ancora vedere occasionalmente nei vecchi sorgenti:

```
delete [100]fp;
```

che obbliga il programmatore ad includere il numero di oggetti dell' array ed ad introdurre la possibilità che il programmatore commetta un errore. L'overhead aggiuntivo per il compilatore è molto basso ed è stato preferito specificare il numero di oggetti in un posto invece che in due.

Un puntatore più simile ad un array

Come digressione, **fp** definito sopra può puntare a qualsiasi cosa, che non ha senso per l'indirizzo di partenza di un array. Ha più senso definirlo come una costante , in modo che qualsiasi tentativo di modificare il puntatore sarà segnalato da un errore. Per ottenere questo effetto, si può provare

```
int const* q = new int[10];
```

oppure

```
const int* q = new int[10];
```

ma in entrambi i casi il **const** sarà vincolato all' **int**, cioè, ciò a cui si punta, piuttosto che la qualità del puntatore stesso. Invece, si deve scrivere:

```
int* const q = new int[10];
```

Ora gli elementi dell'array in **q** possono essere modificati, ma qualsiasi cambiamento a q (tipo **q++**) è illegale, come con un ordinario identificatore di array.

Esaurimento della memoria

Cosa accade quando l'operatore **new()** non trova un blocco contiguo di memoria largo abbastanza per mantenere l'oggetto desiderato? Una speciale funzione chiamata **gestore del new** viene chiamato. O piuttosto, un puntatore ad una funzione viene controllato e se il puntatore non è zero, allora la funzione a cui esso punta viene chiamata.

Il comportamento di default per il gestore del new è di *lanciare un'eccezione*, un argomento trattato nel Volume 2. Tuttavia, se si sta usando l'allocazione nella heap nel nostro programma, è saggio almeno rimpiazzare il gestore del new con un messaggio che dice che si è terminata la memoria e termina il programma. In questo modo, durante il debugging, si avrà un indizio di ciò che è successo. Per il programma finale si avrà bisogno di usare un recupero più robusto.

Si rimpiazza il gestore del new includendo **new.h** e chiamando poi **set_new_handler( )** con l'indirizzo della funzione che si vuole :

```
//:      C13:NewHandler.cpp
//      Cambiare  il gestore del new
#include      <iostream>
#include      <cstdlib>
#include      <new>
using namespace      std;
int      conteggio = 0;
void      fine_della_memoria() {
       cerr << "memoria esaurita dopo" << conteggio
<<      " allocazioni!" << endl;
       exit(1);
}
int main() {
       set_new_handler(fine_della_memoria);
       while(1) {
         conteggio++;
         new int[1000]; // esaurisce la memoria
       }
} ///:~
```

La funzione gestore del new deve non avere argomenti ed avere un valore di ritorno **void**, il ciclo **while** manterrà allocati oggetti **int** (e getterà via i loro indirizzi di ritorno) fino a che la memoria libera viene esaurita. Alla prossima chiamata di **new**, non può essere allocata memoria, quindi il gestore del new verrà chiamato.

Il comportamento del gestore del new è legato all'operatore **new(),** si si vuole fare l'overload di **new ()** (spiegato nella prossima sezione) il gestore del new non sarà chiamato per default. Se si vuole ancora chiamare il gestore del new si deve scrivere il codice per fare ciò dentro operatore **new** sovraccaricato.

Naturalmente, si possono scrivere gestori del new più sofisticati, persino uno che cerca di recuperare memoria (comunemente conosciuto come garbage collector). Questo non è un compito per programmatori novizi.

Overload di new & delete

Quando si usa **new**, accadono due cose. Primo, la memoria viene allocata usando l'operatore **new()**, poi viene chiamato il costruttore. Con **delete**, viene chiamato distruttore, la memoria viene deallocata usando l'operatore **delete()**. Le chiamate al costruttore e distruttore non sono mai sotto il proprio controllo (altrimenti le si potrebbero sovvertire accidentalmente), ma si possono cambiare le funzioni di allocazione della memoria **new()** e **delete().**

Il sistema di allocazione della memoria usato da **new** e **delete** è progettato per scopi generali. In situazioni speciali, tuttavia, non soddisfa i propri bisogni. Il motivo più comune per cambiare l'allocatore è l'efficienza: si potrebbe aver creato e distrutto così tanti oggetti di una particolare classe che è diventato un collo di bottiglia per la velocità. Il C++ permette di utilizzare l'overload per **new** e **delete** per implementare il proprio meccanismo di allocazione della memoria, quindi si possono gestire problemi di questo genere.

Un altro problema è la frammentazione della heap. Allocando oggetti di dimensioni diverse è possibile frazionare la heap al punto che effettivamente si finisce lo spazio. Cioè, la memoria potrebbe essere disponibile, ma a causa della frammentazione nessun pezzo è grande abbastanza per soddisfare le proprie esigenze. Creando il proprio allocatore per una particolare classe, si garantisce che ciò non accada mai.

Nei sistemi embedded e real-time, un programma può dover essere eseguito per un lungo periodo di tempo con risorse ristrette. Tale sistema può anche richiedere che l'allocazione della memoria prenda sempre lo stesso tempo e non sia permessa la mancanza o frammentazione di memoria. Un allocatore di memoria custom è la soluzione; altrimenti, i programmatori evitano del tutto di usare **new** e **delete** in tali casi perdendo un elemento prezioso del C++.

Quando si utilizzano gli operatori **new**() e **delete()** sovraccaricati è importante ricordare che si sta cambiando solo il modo in cui la memoria grezza viene allocata. Il compilatore chiamerà semplicemente la nostra **new** invece di quella di default per allocare memoria, poi chiamerà il costruttore per quella memoria. Quindi, sebbene il compilatore alloca memoria e chiama il costruttore quando vede una **new**, tutto ciò che si può cambiare quando si utilizza l'overload di **new** è la parte di allocazione della memoria. (**delete** ha una limitazione simile).

Quando si utilizza l'overload di **new()**, si può anche rimpiazzare il comportamento quando si esaurisce la memoria, quindi si deve decidere cosa fare nell'operatore **new()**: restituire zero, scrivere un ciclo per chiamare il gestore di new e ritentare l'allocazione o (tipicamente) lanciare un eccezione di bad_alloc (discussa nel Volume 2)

L'overload di **new** e **delete** è identico all'overload di qualsiasi altro operatore. Tuttavia, si ha la possibilità di fare l'overload dell'allocatore globale oppure utilizzare un allocatore differente per una particolare classe.

Overload globale di new & delete

Questo è un approccio drastico, utilizzato quando le versioni globali di new e delete non soddisfano l'intero sistema. Se si sovraccaricano le versioni globali, si rendono le versioni di default completamente inaccessibili e non le si possono chiamare nemmeno dentro le proprie ridefinizioni.

La versione **new** sovraccaricata prende come argomento **size_t** (il tipo standard del C Standard per le dimensioni). Questo argomento viene generato e passato a noi dal compilatore ed è la dimensione dell'oggetto di cui si responsabile per l'allocazione. Si deve restituire un puntatore ad un oggetto di quella dimensione (o più grande , se si ha una ragione di farlo) o a zero se non c'è memoria libera (in tal caso il costruttore non viene chiamato!). Tuttavia, se non c'è memoria, si dovrebbe fare qualcosa in più che restituire solo zero, tipo chiamare il gestore del new o lanciare un'eccezione per segnalare che c'è un problema.

Il valore di ritorno dell'operatore **new()** è un **void*,** non un puntatore ad un tipo particolare. Tutto ciò che si fa è produrre memoria, non un oggetto finito, ciò non accade finchè il costruttore non viene chiamato, un atto che il compilatore garantisce e che è fuori dal nostro controllo.

L'operatore **delete()** prende un **void*** alla memoria che è stata allocata dall'operatore new. È un **void*** perchè l'operatore delete ottiene un puntatore solo dopo che è stato chiamato il distruttore, che rimuove l'oggetto dalla memoria. Il tipo di ritorno è **void**.

Ecco un esempio che mostra come sovraccaricare le versioni globali di **new** e **delete**:

```
//: C13:GlobalOperatorNew.cpp
// Overload globale di new/delete
#include      <cstdio>
#include      <cstdlib>
using namespace std;
void* operator new(size_t      sz) {
       printf("operatore new: %d Bytes\n", sz);
       void* m = malloc(sz);
       if(!m) puts("fine della      memoria);
       return m;
}
void operator delete(void* m) {
       puts("operatore delete");
       free(m);
}
class S {
       int i[100];
public:
       S() { puts("S::S()"); }
       ~S() { puts("S::~S()"); }
};
int      main() {
       puts("creazione & distruzione di un int");
       int* p = new      int(47);
       delete p;
       puts("creazione & distruzione di un s");
       S* s = new      S;
       delete s;
       puts("creazione & distruzione S[3]");
       S* sa = new      S[3];
```

```
        delete []sa;
}       ///:~
```

Qui si può vedere la forma generale dell'overload di new e delete. Esse usano le funzioni del C Standard **malloc()** e **free()** per gli allocatori (che usano per default anche **new** e **delete**!). Tuttavia, esse stampano anche messaggi di ciò che stanno facendo. Si noti che **printf()** e **puts()** vengono usate al posto di **iostreams**, perchè quando un oggetto **iostream** viene creato (come **cin**, **cout** e **cerr**), esso chiama **new** per allocare memoria. Con **printf()**, non si incorre in un punto morto perchè non chiama **new** per inizializzarsi.

Nel **main()**, gli oggetti dei tipi predefiniti vengono creati per provare che **new** e **delete** sovraccaricati vengono chiamati anche in quel caso. Poi un singolo oggetto di tipo **S** viene creato, seguito da un array di **S**. Per l'array, si vedrà dal numero dei byte richiesti che memoria aggiuntiva viene allocata per memorizzare (dentro l'array) il numero di oggetti gestiti. In tutti i casi vengono usate le versioni globali di **new** e **delete** sovraccaricate.

Overload di new & delete per una classe

Sebbene non si deve esplicitamente usare **static**, quando si sovraccarica **new** e **delete** per una classe, si stanno creando funzioni membro statiche. Come prima, la sintassi è la stessa usata per gli operatori. Quando il compilatore vede che si usa **new** per creare un oggetto della propria classe, esso sceglie il membro operatore **new()** al posto della versione globale. Tuttavia, le versioni globali di **new** e **delete** vengono usate per tutti gli altri tipi di oggetti (a meno che essi non hanno i loro **new** e **delete**).

Nel esempio seguente, viene creato un semplice sistema di allocazione della memoria per la classe **Framis**. Un pezzo di memoria è impostato a parte nell'area di memoria statica alla partenza del programma e quella memoria viene usata per allocare spazio per gli oggetti di tipo **Framis**. Per determinare quali blocchi sono stati allocati, viene usato un semplice array, un byte per ogni blocco:

```
//: C13:Framis.cpp
// Local overloaded new & delete
#include      <cstddef> // Size_t
#include      <fstream>
#include      <iostream>
#include      <new>
using namespace      std;
ofstream      out("Framis.out");
class Framis {
        enum { sz = 10 };
        char c[sz]; // occupiamo      spazio, non utilizzato
        static unsigned      char pool[];
        static bool mappa_allocazione[];
public:
        enum { pdimensione = 100 };       // frami permesso
        Framis() { out << "Framis()\n";      }
        ~Framis() { out << "~Framis() ... "; }
        void* operator new(size_t) throw(bad_alloc);
        void operator delete(void*);
};
unsigned char Framis::pool[pdimensione * sizeof(Framis)];
bool      Framis::mappa_allocazione[pdimensione] = {false};
//      la dimensione è ignorata -- si assume       un oggetto
Framis
void*
```

```
Framis::operator new(size_t)      throw(bad_alloc) {
       for(int i = 0; i <      pdimensione; i++)
         if(!mappa_allocazione[i]) {
           out << "utilizzo del blocco" << i <<      " ... ";
           mappa_allocazione[i] = true; // memorizziamo che      è
usato
      return pool + (i * sizeof(Framis));
         }
       out << "fine della memoria" << endl;
       throw bad_alloc();
}
void Framis::operator delete(void* m) {
       if(!m) return; // Controlliamo se è un puntatore null
//      Si assume che sia stato creato nel pool
       // calcolo del numero del blocco
       unsigned long      blocco = (unsigned long)m
         - (unsigned long)pool;
       blocco /= sizeof(Framis);
       out << "rilascio blocco" << blocco<<      endl;
       // impostiamo libero
       mappa_allocazione[blocco] = false;
}
int main() {
       Framis* f[Framis::pdimensione];
       try {
         for(int i = 0; i <      Framis::pdimensione; i++)
           f[i] = new Framis;
         new Framis; // Fine memoria
       } catch(bad_alloc) {
         cerr << "Fine memoria!" << endl;
       }
       delete f[10];
       f[10] = 0;
       // Uso della memoria liberata:
       Framis* x = new      Framis;
       delete x;
       for(int j = 0; j <      Framis::pdimensione; j++)
         delete f[j]; // Delete f[10]      OK
}      ///:~
```

Lo spazio di memoria per la heap di **Framis** viene creato allocando un array di byte grande abbastanza per gestire **pdimensione** oggetti **Framis**. La mappa di allocazione è lunga **pdimensione** elementi, quindi c'è un unico **bool** per ogni blocco. Tutti i valori della mappa di allocazione sono inizializzati a **false**, usando un trucco di inizializzazione che prevede di settare il primo elemento in modo che il compilatore automaticamente inizializza tutti gli altri al valore di default (che è false, nel caso di **bool**).

L'operatore locale new() ha la stessa sintassi di quello globale. Tutto ciò che fa è cercare nella mappa di allocazione i valori **false**, poi setta quella locazione a **true** per indicare che è stato allocato e restituisce l'indirizzo del corrispondente blocco di memoria. Se non trova memoria, stampa un messaggio per tracciare il file e lancia un eccezione **bad_alloc**.

Questo è il primo esempio di eccezione che si vede in questo libro. Questo è un semplice utilizzo delle eccezioni, la cui discussione è rimandata al Volume 2. Nel operatore **new()** ci sono due punti importanti. Il primo, la lista degli argomenti della funzione è seguito da **throw(bad_alloc)**, che dice al compilatore ed al lettore che questa funzione può lanciare un eccezione di tipo **bad_alloc**. Secondo, se non c'è più memoria la funzione lancia davvero l'eccezione con l'istruzione **throw bad_alloc**. Quando un eccezione viene

lanciata, la funzione ferma l'esecuzione ed il controllo viene passato ad un *gestore delle eccezione*, che è espresso dalla clausola **catch**.

Nel **main()**, si vede l'altra parte della figura, che la clausola *try-catch*. Il blocco **try** è circondato da parentesi e contiene il codice che può lanciare le eccezioni, in questo caso qualsiasi chiamata a **new** che coinvolge oggetti **Framis**. Segue immediatamente al blocco **try** uno o più clausole **catch**, ognuna delle quali specifica il tipo di eccezione che trattano. In questo caso **catch(bad_alloc)** indica che le eccezioni bad_alloc saranno trattate qui. Questo particolare catch viene eseguito solo quando viene lanciata un'eccezione **bad_alloc** e l'esecuzione continua dopo la fine dell'ultima clausola **catch** del gruppo (ce n'è una sola qui, ma ce ne potrebbero essere di più).

In questo esempio, va bene l'uso di iostreams perchè le versioni globali dell'operatore **new()** e **delete()** non sono state toccate.

L'operatore **delete()** assume che l'indirizzo di **Framis** sia stato creato nel pool. Ciò è corretto, perchè l'operatore locale **new()** verrà chiamato ogni volta si crea un singolo oggetto **Framis** nella heap, ma non un array di essi: il **new** globale viene usato per gli array. Quindi l'utente potrebbe aver usato accidentalmente l'operatore **delete()** senza la sintassi con le parentesi vuote per indicare la distruzione dell'array. Ciò causerebbe un problema. L'utente potrebbe anche cancellare un puntatore ad un oggetto creato nello stack. Se si pensa che queste cose accadono, sarebbe meglio aggiungere una linea per essere sicuri che l'indirizzo si trova nel pool e su un confine esatto (si comincia a vedere anche il potenziale di **new** e **delete** sovraccaricati per trovare i memory leak).

L'operatore **delete()** calcola il blocco nel pool che questo puntatore rappresenta e poi imposta il flag della mappa di allocazione per quel blocco a false per indicare che il blocco è stato liberato.

In **main()**, vengono allocati tanti oggetti **Framis** quanto basta per finire la memoria, ciò evidenzia il comportamento in questo caso. Poi uno degli oggetti viene liberato ed un altro viene creato per mostrare che la memoria liberata viene riusata.

Poichè questo meccanismo di allocazione è specifico per gli oggetti **Framis**, probabilmente è molto più veloce di meccanismo di allocazione generico usato per default da **new** e **delete**. Tuttavia, si dovrebbe notare che non funziona automaticamete se viene utilizzata l'ereditarietà (discussa nel Capitolo 14).

Overload di new & delete per gli array

Se si vuole usare l'overload di **new** e **delete** per una classe, questi operatori vengono chiamati ogni volta che si crea un oggetto di una classe. Tuttavia, se si crea un array di questi oggetti delle classi, l'operatore globale **new()** viene chiamato per allocare abbastanza spazio per l'array tutto in una volta e l'operatore globale **delete()** viene chiamato per rilasciare quella memoria. Si può controllare l'allocazione di array di oggetti usando la versione speciale dell'overload per la classe. Ecco un esempio che mostra quando due diverse versioni vengono chiamate:

```
//: C13:ArrayOperatorNew.cpp
// Operatore new per arrays
#include      <new> // Size_t definizione
#include      <fstream>
```

```
using namespace std;
ofstream      trace("ArrayOperatorNew.out");
class Widget {
       enum { sz = 10 };
       int i[sz];
public:
       Widget() { trace << "*"; }
       ~Widget() { trace << "~"; }
       void* operator new(size_t sz) {
         trace << "Widget::new: "
              << sz << " bytes" << endl;
         return ::new      char[sz];
       }
       void operator delete(void* p) {
         trace << "Widget::delete" << endl;
         ::delete []p;
       }
       void* operator new[](size_t sz) {
         trace << "Widget::new[]: "
              << sz << " bytes" << endl;
         return ::new      char[sz];
       }
       void operator delete[](void* p) {
         trace << "Widget::delete[]" << endl;
         ::delete []p;
       }
};
int main() {
       trace << "new Widget" << endl;
       Widget* w = new Widget;
       trace << "\ndelete Widget" << endl;
       delete w;
       trace << "\nnew Widget[25]" << endl;
       Widget* wa = new Widget[25];
       trace << "\ndelete []Widget" << endl;
       delete []wa;
}      ///:~
```

Qui, le versioni globali di **new** e **delete** vengono chiamate in modo che l'effetto è lo stesso di non avere le versioni con l'overload di **new** e **delete** tranne che viene aggiunta un informazioni di trace. Naturalmente, si può usare qualsiasi meccanismo di allocazione che si vuole con **new** e **delete** sovraccaricate.

Si può vedere che la sintassi di **new** e **delete** per gli array è la stessa per gli oggetti individuali tranne per l'aggiunta delle parentesi. In entrambi i casi si gestisce la dimensione della memoria che si deve allocare. La dimensione per versione array sarà la dimensione dell'intero array. Vale la pena di tenere in mente che l'unica cosa che è richiesta di fare con l'operatore **new()** è di fornire un puntatore ad un blocco di memoria abbastanza grande. Sebbene si può inizializzare la memoria, normalmente questo è il lavoro del costruttore che sarà chiamato automaticamente dal compilatore.

Il costruttore ed il distruttore stampano quindi si può vedere quando vengono chiamati. Ecco come appare il file di trace per un compilatore:

```
new Widget
Widget::new: 40 bytes
*
delete Widget
```

```
~Widget::delete
new Widget[25]
Widget::new[]: 1004 bytes
*************************
delete []Widget
~~~~~~~~~~~~~~~~~~~~~~~~~~Widget::delete[]
```

Creare un oggetto individuale richiede 40 byte, come ci si aspetta. (Questa macchina usa quattro byte per un **int**). Viene chiamato prima l'operatore **new()** e poi il costruttore(indicato da un *). In maniera complementare, chiamando **delete** viene chiamato il prima il distruttore e poi l'operatore **delete()**.

Quando viene creato un array di **Widget**, viene usata la versione array dell'operatore **new()**, come promesso. Ma si noti che la dimensione richiesta è più grande dei quattro byte che ci si aspetta. Questi ulteriori quattro byte servono al sistema per le informazioni sull'array, in particolare, il numero di oggetti dell'array. Quindi quando si scrive:

```
delete []Widget;
```

le parentesi dicono al compilatore che è un array di oggetti, quindi il compilatore genera codice per conoscere il numero di oggetti nell'array e per chiamare il distruttore tante volte quanto serve. Come si vede, sebbene l'operatore **new()** e **delete()** per array sono chiamati solo una volta per l'intero blocco di array, il costruttore ed il distruttore di default vengono chiamati per ogni oggetto dell'array.

Chiamate al costruttore

Si consideri

```
MioTipo* f = new MioTipo;
```

chiama **new** per allocare un pezzo di memoria grande quanto **MioTipo**, poi invoca il costruttore di **MioTipo** sulla memoria, cosa accade se non si riesce ad allocare memoria con **new**? Il costruttore non viene chiamato in questo caso, quindi sebbene si crea un oggetto senza successo, almeno non si invoca il costruttore e si gestisce un puntatore **this** che vale zero. Ecco un esempio:

```
//:      C13:NoMemory.cpp
//      Il Costruttore non viene chiamato se new fallisce
#include      <iostream>
#include      <new> // definizione bad_alloc
using namespace std;
class NoMemory      {
public:
       FineMemoria() {
         cout << "FineMemoria::FineMemoria()" << endl;
       }
       void* operator new(size_t sz) throw(bad_alloc){
         cout << "FineMemoria::operatore new" << endl;
         throw bad_alloc(); // "Fine      Memoria"
       }
};
int main() {
       FineMemoria* nm = 0;
```

```
        try {
          nm = new FineMemoria;
        } catch(bad_alloc) {
          cerr << "eccezione: Memoria terminata" <<      endl;
        }
        cout << "nm = " << nm << endl;
} ///:~
```

Quando il programma viene eseguito, non stampa il messaggio del costruttore, ma il messaggio del operatore **new()** ed il messaggio del gestore dell'eccezione. Poichè **new** non ritorna mai, il costruttore non viene mai chiamato quindi il suo messaggio non viene stampato.

È importante che **nm** sia inizializzato a zero poichè l'espressione **new** non termina mai ed il puntatore dovrebbe essere zero per essere sicuri che non se ne faccia un cattivo uso. Tuttavia, si dovrebbe fare qualcosa in più nel gestore dell'eccezione che stampare solo un messaggio e continuare come se l'oggetto fosse stato creato correttamente. Idealmente si farà qualcosa che risolverà il problema e si uscirà dopo aver riportato l'errore su un file di log.

Nelle prime versioni del C++ era una pratica normale restituire zero da un **new** se falliva l'allocazione della memoria. Tuttavia, se si prova a restituire zero da un **new** con un compilatore Standard, ci viene detto che si dovrebbe invece lanciare un'eccezione di **bad_alloc**.

placement new & delete(new e delete con piazzamento)

Ci sono altri due, meno comuni, utilizzi del overload per l'operatore **new().**

1. Si può volere piazzare un oggetto in una specifica locazione di memoria. Ciò ha particolare importanza con i sistemi embedded hardware dove un oggetto può essere sinonimo di un particolare pezzo di hardware.
2. Si può volere scegliere fra diversi allocatori quando si usa **new**.

Entrambe le situazione sono gestite con lo stesso meccanismo: l'operatore **new()** sovraccaricato accetta più di un argomento. Come si è visto prima, il primo argomento è sempre la dimensione dell'oggetto, che è calcolato di nascosto e passato dal compilatore. Ma l'altro argomento può essere qualsiasi cosa si voglia, l'indirizzo dove si vuole che l'oggetto sia piazzato, un riferimento alla funzione di allocazione della memoria oppure oggetto o qualsiasi altra cosa.

Il modo in cui si passa l'argomento extra all'operatore **new()** durante una chiamata può sembrare un pò curioso all'inizio. Si mette la lista degli argomenti (*senza* l'argomento **size_t**, che è gestito dal compilatore) dopo la parola chiave new e prima del nome della classe dell'oggetto che si sta creando. Per esempio,

```
X* xp = new(a) X;
```

passerà a come secondo argomento all'operatore **new().** Naturalmente, ciò funziona solo se tale operatore **new()** è stato dichiarato.

Ecco un esempio che mostra come su può piazzare un oggetto in una particolare locazione:

```
//:      C13:PlacementOperatorNew.cpp
//      Piazzamento con operatore new()
#include      <cstddef> // Size_t
#include      <iostream>
using namespace std;
class X {
       int i;
public:
       X(int ii = 0) : i(ii) {
         cout << "this = " << this << endl;
       }
       ~X() {
         cout << "X::~X(): " << this << endl;
       }
       void* operator new(size_t, void* loc)      {
         return loc;
       }
};
int main() {
       int l[10];
       cout << "l = " << l << endl;
       X* xp = new(l) X(47); // X alla      locazione location l
       xp->X::~X(); // Chiamata esplicita al distruttore
//      Da usare solo con il piazzamento!
} ///:~
```

Si noti che l'operatore **new** restituisce solo il puntatore che gli viene passato. Quindi, il chiamante decide dove l'oggetto verrà situato ed il costruttore viene chiamato per quella memoria come parte dell'espressione new.

Sebbene questo esempio mostra solo un argomento addizionale, nulla ci impedisce di aggiungerne altri se ce ne è il bisogno per altri scopi.

Un dilemma accade quando si vuole distruggere l'oggetto. C'è un'unica versione dell'operatore **delete**, quindi non c'è modo di dire: "Usa il mio speciale deallocatore per questo oggetto". Si vuole chiamare il distruttore, ma non si vuole che la memoria venga liberata dal meccanismo che gestisce la memoria dinamica poichè non è stata allocata nella heap.

La risposta è data da una sintassi molto speciale. Si può esplicitare la chiamata del distruttore:

```
xp->X::~X(); // Chiamata esplicita al distruttore
```

Un austero warning è di regola qui. Qualcuno vede ciò come un modo per distruggere gli oggetti un pò di tempo prima della fine dello scope, piuttosto che modificare lo scope o (più correttamente) usare la creazione dinamica degli oggetti se si vuole che il tempo di vita dell'oggetto sia determinato a runtime. Si avranno seri problemi se si chiama il distruttore in questo modo per un oggetto creato in modo normale sullo stack perchè il distruttore verrà chiamato di nuovo alla fine dello scope. Se si chiama il distruttore per un oggetto creato nella heap, il distruttore verrà eseguito, ma la memoria non verrà rilasciata, che probabilmente non è ciò che si vuole. L'unica ragione per cui il distruttore può essere chiamato esplicitamente in questo modo è supportare la sintassi per il piazzamento dell'operatore **new**.

C'è anche un operatore **delete** con piazzamento che viene chiamato solo se un costruttore per un **new** con piazzamento lanci un' eccezione (in modo che la memoria viene automaticamente pulita durante l'eccezione). L'operatore delete con piazzamento ha una lista di argomenti che corrisponde all'operatore **new** con piazzamento che viene chiamato prima che il costruttore lancia l'eccezione. Questo argomento verrà esplorato nel capitolo della gestione delle eccezioni nel Volume 2.

Sommario

È conveniente e molto efficiente creare oggetti automatici nello stack, ma per risolvere il programma della programmazione generale si deve poter creare e distruggere oggetti in ogni momento durante l'esecuzione di un programma, in particolare per rispondere alle informazioni dall'esterno del programma. Sebbene l'allocazione dinamica della memoria in C avverrà nella heap, essa non fornisce la facilità di utilizzo e le garanzie del C++. Portando la creazione dinamica di oggetti nel nucleo del linguaggio con **new** e **delete**, si possono creare oggetti nella heap facilmente come si fa nello stack. In aggiunta, si ottiene una grande flessibilità. Si può cambiare il comportamento di **new** e **delete** se non sono adatti ai propri bisogni, in particolare se non sono abbastanza efficienti. Si può anche modificare ciò che avviene quando finisce lo spazio nella heap.

---

[50] C'è un sintassi speciale chiamata *placement new* la quale permette di chiamare un costruttore per un pezzo di memoria pre-allocato. Questo argomento verrà introdotto più in avanti nel capitolo.

14: Ereditarietà & Composizione

Una delle caratteristiche più irresistibili del C++ è il riutilizzo del codice. Ma per essere rivoluzionari, si ha bisogno di essere capaci di fare molto di più che copiare codice e cambiarlo.

Questo è stato l'approccio del C e non ha funzionato molto bene. Per la maggior parte del C++, la soluzione gira intorno alla classe. Si riusa il codice creando classi nuove, ma invece di crearle da zero, si usano classi esistenti che qualcun altro ha costruito e ha debuggato.

Il trucco è usare classi senza sporcare il codice esistente. In questo capitolo si vedranno due modi per ottenere ciò. Il primo è piuttosto facile: si creano semplicemente oggetti della propria classe esistente nella classe nuova. Ciò è chiamato *composizione*, perchè la classe nuova è composta di oggetti di classi esistenti.

Il secondo approccio è più sottile. Si crea una classe nuova come un *tipo* di una classe esistente. Si prende letteralmente la forma della classe esistente e si aggiunge codice ad essa, senza cambiare la classe esistente. Questo atto magico è chiamato *ereditarietà* e la maggior parte del lavoro è fatto dal compilatore. L'ereditarietà è una delle pietre angolari della programmazione orientata agli oggetti e ha implicazioni aggiuntive che saranno esplorate in Capitolo 15.

Ne risulta che molto della sintassi e comportamento è simile sia alla composizione e all'ereditarietà (questo ha senso, sono due modi di creare tipi nuovi da tipi esistenti). In questo capitolo, s'imparerà come utilizzare questi meccanismi per il riutilizzo del codice.

Sintassi della composizione

A dire il vero si è sempre usata la composizione per creare classi. Si sono composte classe primariamente con tipi predefiniti (e qualche volta le stringhe). Risulta facile usare la composizione con tipi definiti dall'utente.

Si consideri una classe che è utile per qualche motivo:

```
//: C14:Useful.h
//Una classe da riutilizzare
#ifndef USEFUL_H
#define USEFUL_H

class X {
  int i;
public:
  X() { i = 0; }
  void set(int ii) { i = ii; }
  int read() const { return i; }
  int permute() { return i = i * 47; }
};
#endif // USEFUL_H ///:~
```

I membri dato sono **private** in questa classe, quindi è totalmente sicuro includere un oggetto di tipo **X** come un oggetto **public** in una nuova classe, che rende l'interfaccia semplice:

```
//: C14:Composition.cpp
// riutilizzo del codice con la composizione
#include "Useful.h"

class Y {
  int i;
public:
  X x; //  oggetto incorporato
  Y() { i = 0; }
  void f(int ii) { i = ii; }
  int g() const { return i; }
};

int main() {
  Y y;
  y.f(47);
  y.x.set(37); // accesso  all'oggetto incorporato
} ///:~
```

Le funzioni membro dell'oggetto incorporato (indicato come un *suboggetto*) semplicemente richiedeno un'altra selezione del membro.

è più comune fare gli oggetti incorporati privati, poichè divengono parte della realizzazione sottostante (che significa che si può cambiare l'implementazione se si vuole). Le funzioni pubbliche dell'interfaccia per la propria classe nuova coinvolgono poi l'uso dell'oggetto incorporato, ma non necessariamente mimano l'interfaccia dell'oggetto:

```
//: C14:Composition2.cpp
// oggetti incorporati privatamente
#include "Useful.h"

class Y {
  int i;
  X x; // oggetto incorporato
public:
  Y() { i = 0; }
  void f(int ii) { i = ii; x.set(ii); }
  int g() const { return i * x.read(); }
  void permute() { x.permute(); }
};

int main() {
  Y y;
  y.f(47);
  y.permute();
} ///:~
```

Qui, la funzione **permute( )** è presente nell'interfaccia della classe nuova, ma le altre funzioni del membro di **X** sono usate fra i membri di **Y**.

sintassi dell'ereditarietà

La sintassi per la composizione è ovvia, ma per usare l'ereditarietà c'è una forma nuova e diversa.

Quando si eredita, si sta dicendo,"Qu*esta nuova classe è come quella vecchia classe*". Si afferma questo nel codice dando come al solito il nome della classe, ma prima della parentesi di apertura del corpo della classe, si mettono due punti ed il nome della classe base (o classi base, separate da virgole per l'ereditarietà multipla). Quando si fa questo, si ottengono automaticamente tutti i membri dato e funzioni membro della classe base. Ecco un esempio:

```
//: C14:Inheritance.cpp
// Semplice ereditarietà
#include "Useful.h"
#include <iostream>
using namespace std;

class Y : public X {
  int i; // diverso da i di X
public:
  Y() { i = 0; }
  int change() {
    i = permute(); // diversa chiamata di nome
return i;
  }
  void set(int ii) {
    i = ii;
    X::set(ii); // chiamata a funzione con lo stesso nome
  }
};

int main() {
  cout << "sizeof(X) = " << sizeof(X) << endl;
  cout << "sizeof(Y) = "
       << sizeof(Y) << endl;
  Y D;
  D.change();
  // utilizzo delle funzioni dell'interfacci di X :
  D.read();
  D.permute();
  // le funzioni ridefinite occultano le versioni base:
  D.set(12);
} ///:~
```

Si può vedere che **Y** è ereditato da **X**, che significa che **Y** conterrà tutti gli elementi dato di **X** e tutte le funzioni membro di **X**. Infatti, **Y** contiene solo un suboggetto di **X** proprio come se si avesse creato un oggetto membro di **X** in **Y** invece di ereditarlo da **X.** Ci si riferisce come suboggetti sia agli oggetti membro che ai dati della classe base.

Tutti gli elementi privati di **X** ancora sono privati in **Y**; ovvero, poichè **Y** eredita da **X**, ciò non significa che **Y** può rompere il meccanismo di protezione. Gli elementi privati di **X** sono ancora là, prendono spazio, non si può accedere ad essi direttamente.

In **main( )** si possono vedere che gli elementi dato di **Y** sono combinati con quelli di **X**, perchè il **sizeof(Y)** è grande due volte **sizeof(X)**.

Si noti che la classe base è preceduta da **public**. Usando l'ereditarietà viene assunto tutto come **private**. Se la classe base non fosse preceduta da **public**, vorrebbe dire che tutti dei membri pubblici della classe base sarebbero privati nella classe derivata. Questo non è quasi mai quello che si vuole[51]; il risultato desiderato è mantenere tutti i membri pubblici della classe base pubblici nella classe derivata. Si fa questo usando la parola riservata **public** durante l'ereditarietà.

In **change( )**, **il permute()** della classe base viene chiamata. La classe derivata ha accesso diretto a tutte le funzioni pubbliche della classe base.

La funzione **set( )** nella classe derivata ridefinisce la funzione **set( )** della classe base. Ovvero, se si chiama il **read( )** e **permute( )** per un oggetto di tipo **Y**, si ottengono le versioni della classe base di quelle funzioni (si può vedere questo accadere in **main( )**). Ma se si chiama **set( )** per un oggetto di **Y**, si ottiene la versione ridefinita. Questo vuole dire che se non piace la versione di una funzione che si ottiene durante l'ereditarietà, si può cambiare quello che fa (si possono aggiungere anche funzioni completamente nuove come **change( )**).

Comunque, quando si ridefinisce una funzione, si può volere ancora chiamare la versione della classe base. Se, in **set( )**, si chiama semplicemente **set( )** si ottiene la versione locale della funzione, una chiamata ricorsiva di funzione. Si deve chiamare esplicitamente la classe base usando l'operatore di risoluzione dello scope per chiamare la versione della classe base.

La lista di inizializzazione del costruttore

Si è visto come sia importante in C++ garantire un'inizializzazione corretta e non è diverso durante la composizione e l'ereditarietà. Quando un oggetto viene creato, il compilatore garantisce che vengano chiamati i costruttori per tutti i suoi suboggetti. Negli esempi visti finora, tutti i suboggetti hanno costruttori per default e il compilatore li chiama automaticamente. Ma cosa accade se i suboggetti non hanno costruttori per default o se si vuole cambiare un argomento di default in un costruttore? Questo è un problema perchè il costruttore della nuova classe non hanno il permesso di accedere agli elementi dato privati del suboggetto, quindi non può inizializzarli direttamente.

La soluzione è semplice: chiamare il costruttore per il suboggetto. Il C++ fornisce una sintassi speciale per questo, la *lista di inizializzazione del costruttore*. La forma della *lista di inizializzazione del costruttore* imita l'atto di ereditarietà. Con l'ereditarietà, si mette la classe base dopo un due punti e prima della parentesi di apertura del corpo della classe. Nella lista di inizializzazione del costruttore, si mettono le chiamate ai costruttori dei suboggetti dopo *lista di inizializzazione del costruttore* e un due punti, ma prima della parentesi di apertura del corpo della funzione. Per una classe **MioTipo**, ereditata da **Barra** ciò appare come:

```
MioTipo::MiTipo(int i) : Barra(i) { // ...
```

se **Barra** ha un costruttore che prende un solo argomento **int**.

Inizializzazione dell'oggetto membro

Si usa questa sintassi molto simile per l'inizializzazione dell' oggetto membro quando si utilizza la composizione. Per la composizione, si danno i nomi degli oggetti invece dei nomi delle classi. Se si ha più di una chiamata di costruttore nella lista di inizializzazione del costruttore, si separano le chiamate con virgole:

```
MioTipo2::MioTipo2(int i) : Barra(i), m(i+1) { // ...
```

Questo è l'inizio di un costruttore per la classe **MioTipo2** che è ereditata da **Barra** e contiene un oggetto membro chiamato **m**. Si faccia attenzione che mentre si può vedere il tipo della classe base nella lista di inizializzazione del costruttore, si vede solamente l'identificativo del oggetto membro.

Tipi predefiniti nella lista di inizializzazione

La lista di inizializzazione del costruttore permette di chiamare esplicitamente i costruttori per oggetti membro. Infatti, non c'è nessun altro modo di chiamare quei costruttori. L'idea è che i costruttori sono tutti chiamati prima che si entra nel corpo del costruttore delle classi nuove. In questo modo, qualsiasi chiamata si faccia a funzioni membro di suboggetti andrà sempre ad oggetti inizializzati. Non c'è modo di andare alla parentesi di apertura del costruttore senza che alcun costruttore sia chiamato per tutti gli oggetti membro e gli oggetti della classe base, anche se il compilatore deve fare una chiamata nascosta ad un costruttore per default. Questo è un ulteriore rafforzamento del C++ a garanzia che nessuno oggetto (o parte di un oggetto) possa uscire dalla barriera iniziale senza che il suo costruttore venga chiamato.

Questa idea che tutti gli oggetti membro siano inizializzati nel momento in cui si raggiunge la parentesi di apertura del costruttore è un aiuto alla programmazione. Una volta che si giunge alla parentesi apertura, si può presumere che tutti i suboggetti sono inizializzati propriamente e ci si concentra su i compiti specifici che si vuole completare nel costruttore. Comunque, c'è un intoppo: cosa succede agli oggetti membro dei tipi predefiniti che non hanno costruttori?

Per rendere la sintassi coerente, si è permesso di trattare i tipi predefiniti come se avessero un solo costruttore che prende un solo argomento: una variabile dello stesso tipo come la variabile che si sta inizializzando. Quindi si può scrivere:

```
//: C14:PseudoConstructor.cpp
// Pseudo Costruttore
class X {
  int i;
  float f;
  char c;
  char* s;
public:
  X() : i(7), f(1.4), c('x'), s("howdy") {}
};

int main() {
  X x;
  int i(100);  // applicato ad un'ordinaria definizione
```

```
int* ip = new int(47);
} ///:~
```

L'azione di queste pseudo chiamate al costruttore è di compiere una semplice assegnazione. È una tecnica utile ed un buon stile di codifica, quindi la si vedrà spesso. È anche possibile usare la sintassi dello pseudo-costruttore quando si crea una variabile di un tipo predefinito fuori di una classe:

```
int i(100);
int* ip = new int(47);
```

Ciò rende i tipi predefiniti un poco più simili agli oggetti. Si ricordi, tuttavia, che questi non sono i veri costruttori. In particolare, se non si fa una chiamata esplicitamente ad uno pseudo-costruttore, nessuna inizializzazione viene compiuta.

Combinare composizione & ereditarietà

Chiaramente, si può usare la composizione e l'ereditarietà insieme. L'esempio seguente mostra la creazione di una classe più complessa che le usa entrambe.

```
//: C14:Combined.cpp
// Ereditarietà & composizione

class A {
  int i;
public:
  A(int ii) : i(ii) {}
  ~A() {}
  void f() const {}
};

class B {
  int i;
public:
  B(int ii) : i(ii) {}
  ~B() {}
  void f() const {}
};

class C : public B {
  A a;
public:
  C(int ii) : B(ii), a(ii) {}
  ~C() {} // chiama ~A() e ~B()
  void f() const {  // ridefinizione
    a.f();
    B::f();
  }
};

int main() {
  C c(47);
} ///:~
```

C eredita da **B** e ha un oggetto membro (è composta di) del tipo **A**. Si può vedere che la lista di inizializzazione del costruttore contiene chiamate ad entrambi i costruttori della classe base e al costruttore del oggetto membro.

La funzione **C::f( )** ridefinisce **B::f( ),** che eredita e chiama anche la versione della classe base. In aggiunta chiama **a.f( )**. Si noti che l'unica volta che si può parlare di ridefinizione di funzioni è con l'ereditarietà; con un oggetto membro si può manipolare solamente l'interfaccia pubblica dell'oggetto, non ridefinirla. In aggiunta, chiamando **f( )** per un oggetto della classe **C** non si chiamerebbe **a.f( )** se **C::f( )** non fosse stato definito, mentre si chiamerebbe **B::f( ).**

Chiamate automatiche al distruttore

Sebbene spesso sia richiesto di fare chiamate esplicite al costruttore nella lista di inizializzazione, non si ha mai bisogno di fare chiamate esplicite al distruttore perchè c'è solamente un distruttore per classe ed esso non prende nessun argomento. Il compilatore ancora assicura comunque, che tutti i distruttori vengano chiamati e cioè tutti i distruttori dell'intera gerarchia , cominciando dal distruttore più derivato e risalendo indietro alla radice.

Vale la pena di sottolineare che costruttori e distruttori sono piuttosto insoliti nel modo in cui vengono chiamati nella gerarchia, laddove con una funzione membro normale viene chiamata solamente quella funzione, ma nessuna delle versioni della classe base. Se si vuole chiamare anche la versione della classe base di una funzione membro normale che si sta sovrascrivendo, lo si deve fare esplicitamente.

Ordine delle chiamate al costruttore & distruttore

È interessante conoscere l'ordine delle chiamate al costruttore e distruttore quando un oggetto ha molti suboggetti. L'esempio seguente mostra precisamente come funziona:

```
//: C14:Order.cpp
// ordine costruttore/distruttore
#include <fstream>
using namespace std;
ofstream out("order.out");

#define CLASS(ID) class ID { \
public: \
  ID(int) { out << #ID " costruttore\n"; } \
  ~ID() { out << #ID " distruttore\n"; } \
};

CLASS(Base1);
CLASS(Member1);
CLASS(Member2);
CLASS(Member3);
CLASS(Member4);

class Derived1 : public Base1 {
  Member1 m1;
  Member2 m2;
public:
  Derived1(int) : m2(1), m1(2), Base1(3) {
    out << "Derived1 costruttore\n";
  }
```

```
  ~Derived1() {
    out << "Derived1 distruttore\n";
  }
};

class Derived2 : public Derived1 {
  Member3 m3;
  Member4 m4;
public:
  Derived2() : m3(1), Derived1(2), m4(3) {
    out << "Derived2 costruttore\n";
  }
  ~Derived2() {
    out << "Derived2 distruttore\n";
  }
};

int main() {
  Derived2 d2;
} ///:~
```

Per primo, un oggetto dell'**ofstream** viene creato per spedire tutto l'output ad un file. Poi, per risparmiare caratteri da digitare e dimostrare una tecnica che usa le macro (sostituita da una migliore tecnica del Capitolo 16), ne si crea una per costruire alcune delle classi che sono usate poi con l'ereditarietà e composizione. Ognuno dei costruttori e distruttori riporta se stesso nel file. Si noti che i costruttori non sono costruttori per default; ognuno di loro ha un argomento **int**. L'argomento stesso non ha identificativo; la sua unica ragione di esistenza deve costringerlo a chiamare esplicitamente i costruttori nella lista di inizializzazione (eliminare l'identificatore evita che il compilatore dia messaggi di warning).

L' output del programma è

```
Base1 costruttore
Member1 costruttore
Member2 costruttore
Derived1 costruttore
Member3 costruttore
Member4 costruttore
Derived2 costruttore
Derived2 distruttore
Member4 distruttore
Member3 distruttore
Derived1 distruttore
Member2 distruttore
Member1 distruttore
Base1 distruttore
```

Si può vedere che la costruzione inizia alla radice della gerarchia della classe e che a ciascun livello il costruttore della classe base viene chiamato prima, seguito dai costruttori dell' oggetto membro. I distruttori sono chiamati precisamente nell'ordine inverso dei costruttori, questo è importante a causa delle dipendenze potenziali (nel costruttore della classe derivata o distruttore, si deve potere presumere che il suboggetto della classe base è ancora disponibile per l'uso ed è già stato costruito o non è stato distrutto ancora).

È anche interessante che l'ordine di chiamata del costruttore per oggetti membro è completamente non soggetto all'ordine delle chiamate nella lista di inizializzazione del costruttore. L'ordine è determinato dall'ordine in cui gli oggetti membro sono dichiarati nella classe. Se si potesse cambiare l'ordine di chiamata dei costruttori con lista di inizializzazione del costruttore, si potrebbero avere due sequenze della chiamata diverse in due costruttori diversi, ma il povero distruttore non saprebbe come invertire propriamente l'ordine delle chiamate per la distruzione e si potrebbe finire con un problema di dipendenza.

Occultamento del nome

Se si eredita una classe e si fornisce una definizione nuova per una delle sue funzioni membro, ci sono due possibilità. Il primo è che si fornisce la firma esatta ed il tipo di ritorno nella definizione della classe derivata come nella definizione della classe base. Questo viene chiamato *ridefinizione* di funzioni membro ordinarie e *overriding* quando la funzione membro della classe base è una funzione **virtual** (virtuale, funzioni virtuali sono comuni e saranno illustrate in dettaglio nel Capitolo 15). Ma cosa accade se si cambia la lista degli argomenti della funzione membro o il tipo restituito dalla classe derivata? Ecco un esempio:

```
//: C14:NameHiding.cpp
// occultamento dei nomi sovraccaricati durante l'ereditarietà
#include <iostream>
#include <string>
using namespace std;

class Base {
public:
  int f() const {
    cout << "Base::f()\n";
    return 1;
  }
  int f(string) const { return 1; }
  void g() {}
};

class Derived1 : public Base {
public:
  void g() const {}
};

class Derived2 : public Base {
public:
  // ridefinizione:
  int f() const {
    cout << "Derived2::f()\n";
    return 2;
  }
};

class Derived3 : public Base {
public:
  // cambio del tipo restituito:
  void f() const { cout << "Derived3::f()\n"; }
};

class Derived4 : public Base {
public:
```

```
  // cambio della lista dei argomenti:
  int f(int) const {
    cout << "Derived4::f()\n";
    return 4;
  }
};

int main() {
  string s("ciao");
  Derived1 d1;
  int x = d1.f();
  d1.f(s);
  Derived2 d2;
  x = d2.f();
//!  d2.f(s); // versione stringa occultata
  Derived3 d3;
//!  x = d3.f(); // restituisce la versione intera occultata
  Derived4 d4;
//!  x = d4.f(); // versione occultata di f()
  x = d4.f(1);
} ///:~
```

In **Base** si vede una **f()** sovraccaricata, e **Derived1** non fa nessun cambiamento a **f( )** ma ridefinisce **g( )**. In **main( )**, si può vedere che entrambe le versioni sovraccaricate di **f( )** sono disponibili in **Derived1**. Comunque, **Derived2** ridefinisce una versione sovraccaricaricata di **f( )** ma non l'altra ed il risultato è che la seconda forma sovraccaricata non è disponibile. In **Derived3**, cambiare il tipo del ritorno nasconde entrambe le versioni della classe base, e **Derived4** mostra che cambiare la lista di inizializzazione del costruttore nasconde entrambe le versioni della classe base. In generale, possiamo dire che ognivolta che si ridefinisce un nome di funzione sovraccaricata da una classe base, tutte le altre versioni sono nascoste automaticamente alla classe nuova. Nel Capitolo 15, si vedrà che l'aggiunta della parola chiave **virtual** influenza un pò di più l'overloading.

Se si cambia l'interfaccia della classe base cambiando la firma e/o il tipo restituito da una funzione membro dalla classe base, poi si usa la classe in un modo diverso in cui l'ereditarietà normalmente intende. Non necessariamente significa che si sta sbagliando, è solo che la meta ultima dell' ereditarietà è sostenere il *polimorfismo* e se si cambia poi la firma della funzione o il tipo restitutito si sta cambiando davvero l'interfaccia della classe base. Se questo è quello che si è inteso di fare allora si sta usando l'ereditarietà per riusare il codice e non per mantenere l'interfaccia comune della classe base (che è un aspetto essenziale del polimorfismo). In generale, quando si usa l'ereditarietà in questo modo vuol dire che si sta prendendo una classe per scopo generale e la si sta specializzando per un particolare bisogno, che di solito è, ma non sempre, considerato il reame della composizione.

Per esempio, si consideri la classe **Stack** del Capitolo 9. Uno dei problemi con quella classe è che si doveva compiere un cast ogni volta che si otteneva un puntatore dal contenitore. Questo non solo è tedioso, ma è anche pericoloso, si potrebbe castare il puntatore a qualsiasi cosa che si vuole.

Un miglior approccio ad un primo sguardo è specializzare la classe generale **Stack** usando l'ereditarietà. Ecco un esempio che utilizza una classe dal Capitolo 9:

```
//: C14:InheritStack.cpp
// Specializzare la classe Stack
#include "../C09/Stack4.h"
#include "../require.h"
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

class StringStack : public Stack {
public:
  void push(string* str) {
    Stack::push(str);
  }
  string* peek() const {
    return (string*)Stack::peek();
  }
  string* pop() {
    return (string*)Stack::pop();
  }
  ~StringStack() {
    string* top = pop();
    while(top) {
      delete top;
      top = pop();
    }
  }
};

int main() {
  ifstream in("InheritStack.cpp");
  assure(in, "InheritStack.cpp");
  string line;
  StringStack textlines;
  while(getline(in, line))
    textlines.push(new string(line));
  string* s;
  while((s = textlines.pop()) != 0) { // Nessun cast!
    cout << *s << endl;
    delete s;
  }
} ///:~
```

Poichè tutte delle funzioni del membro in **Stack4.h** sono inline, nulla ha bisogno di essere linkato.

StringStack specializza **Stack** in modo che il **push( )** accetterà solamente puntatori **String**. Prima, **Stack** avrebbe accettato puntatori **void**, quindi l'utente non aveva nessuno controllo del tipo per essere sicuro fare che venissero inseriti i puntatori corretti. In aggiunta, **peek( )** e **pop( )** ora restituiscono puntatori **String** invece di puntatori **void**, quindi nessuno cast è necessario per usare il puntatore.

Straordinariamente, questo extra controllo del tipo è gratis in **push( )**, **peek( )**, e **pop( )**! Si stanno dando informazioni del tipo in più al compilatore che esso usa a tempo di compilazione, ma le funzioni sono inline e nessuno codice addizionale viene generato.

L'occultamento dei nomi entra in gioco qui perchè, in particolare, la funzione **push( )** ha una firma diversa: la lista degli argomenti è diversa. Se si avessero due versioni di **push( )**

nella stessa classe, questo sarebbe overloading, ma in questo caso sovraccaricare non è quello che noi vogliamo perchè, ancora si permetterebbe di passare qualsiasi genere di puntatore in **push( )** come un **void** *. Fortunatamente, il C++ nasconde **push(void *)** della classe base in favore della versione nuova definito nella classe derivata e perciò permette solamente di usare **push( )** con puntatori **string** dentro **StringStack**.

Poichè, noi ora possiamo garantire di conoscere precisamente che genere di oggetti ci sono nel contenitore, il distruttore lavora correttamente ed il problema della proprietà è risolto o almeno si ha un approccio al problema della proprietà. Qui, se si usa **push( )** per un puntatore alla **StringStack**, poi (secondo le semantiche del **StringStack**) si passa anche la proprietà di quel puntatore al **StringStack**. Se si usa **pop( )** per ottenere il puntatore, non solo si ottiene il puntatore, ma si ottiene anche la proprietà di quel puntatore. Qualsiasi puntatore che viene lasciato in **StringStack** quando il suo distruttore viene chiamato è cancellato da quel distruttore. E poichè questi sono sempre puntatori **string** e l'istruzione **delete** lavora su puntatori **string** invece di puntatori **void**, avviene la corretta distruzione e tutto funziona correttamente.

C'è un inconveniente: questa classe funziona solamente con puntatori **string**. Se si vuole un **Stack** che funziona con qualche altro genere di oggetto, si deve scrivere una nuova versione della classe in modo che funziona solamente col il nuovo tipo di oggetto. Questo diviene rapidamente tedioso ed è risolto finalmente usando i template, come si vedrà nel Capitolo 16.

Possiamo fare un'osservazione supplementare circa questo esempio: cambia l'interfaccia della **Stack** nel processo di ereditarietà. Se l'interfaccia è diversa, allora un **StringStack** realmente non è uno **Stack** e non si potrà mai usare correttamente un **StringStack** come uno **Stack**. Questo rende discutibile l'uso dell'ereditarietà; se non si crea un **StringStack** che è un tipo di **Stack**, allora perchè si sta ereditando? Una versione più adatta di **StringStack** sarà mostrata più avanti in questo capitolo.

Funzioni che non ereditano automaticamente

Non tutte le funzioni sono ereditate automaticamente dalla classe base nella classe derivata. Costruttori e distruttori trattano la creazione e distruzione di un oggetto e sanno che fare solamente con gli aspetti dell'oggetto per la loro particolare classe, quindi tutti i costruttori e distruttori nella gerarchia sotto di loro devono essere chiamati. Dunque costruttori e distruttori ereditano e devono essere creati per ogni classe derivata.

In aggiunta, l'operatore = non eredita perchè, compie un attività come quella del costruttore. Ovvero, solo perchè, si sa come assegnare tutti i membri di un oggetto sul lato sinistro del = da un oggetto sul lato destro, non significa che l'assegnazione avrà ancora lo stesso significato dopo ereditarietà.

Al posto dell'ereditarietà, queste funzioni sono sintetizzate dal compilatore se non le si crea (con i costruttori, non si possono creare qualsiasi costruttore in modo che il compilatore sintetizzi il costruttore di default e il costruttore di copia). Questo è stato descritto brevemente nel Capitolo 6. I costruttori sintetizzati usano l'inizializzazione membro a membro e l'operatore sintetizzato = usa l'assegnazione membro a membro. Ecco un esempio delle funzioni che sono sintetizzate dal compilatore:

```
//: C14:SynthesizedFunctions.cpp
```

```
// Funzioni che sono sintetizzati dal compilatore
#include <iostream>
using namespace std;

class GameBoard {
public:
  GameBoard() { cout << "GameBoard()\n"; }
  GameBoard(const GameBoard&) {
    cout << "GameBoard(const GameBoard&)\n";
  }
  GameBoard& operator=(const GameBoard&) {
    cout << "GameBoard::operator=()\n";
    return *this;
  }
  ~GameBoard() { cout << "~GameBoard()\n"; }
};

class Game {
  GameBoard gb; // Composizione
public:
  // costruttore di GameBoard di default :
  Game() { cout << "Game()\n"; }
  // Si deve chiamare il costruttore di copia  di  GameBoard
  // oppure  il costruttore di default
  // viene invece chiamato automaticamente:
  Game(const Game& g) : gb(g.gb) {
    cout << "Game(const Game&)\n";
  }
  Game(int) { cout << "Game(int)\n"; }
  Game& operator=(const Game& g) {
    // Si deve chiamare esplicitamente l'operatore di assegnazione di GameBoard
    // altrimenti non avviene nessuna assegnazione per gb!

    gb = g.gb;
    cout << "Game::operator=()\n";
    return *this;
  }
  class Other {}; // classe incoporata
// conversione automatica del tipo:
  operator Other() const {
    cout << "Game::operator Other()\n";
    return Other();
  }
  ~Game() { cout << "~Game()\n"; }
};

class Chess : public Game {};

void f(Game::Other) {}

class Checkers : public Game {
public:
  // costruttore della classe base di default:
  Checkers() { cout << "Checkers()\n"; }
  // Si deve chiamare esplicitamente il costruttore di copia
  // della classe base altrimenti verrà chiamato
  // il costruttore di default
  Checkers(const Checkers& c) : Game(c) {
    cout << "Checkers(const Checkers& c)\n";
  }
  Checkers& operator=(const Checkers& c) {
    // Si deve chiamare esplicitamente la versione della classe base
    // dell'operatore=() altrimenti nessuna assegnazione
```

```
    // della classe base avverrà:
    Game::operator=(c);
    cout << "Checkers::operator=()\n";
    return *this;
  }
};

int main() {
  Chess d1;  // Costruttore di Default
  Chess d2(d1); // Costruttore di Copia
//! Chess d3(1); // Errore: nessun costruttore di  int
  d1 = d2; // Operatore = sintetizzato
  f(d1); // la conversione di tipo viene ereditata
  Game::Other go;
//!  d1 = go; // Operatore = non  sintetizzato
           // per tipi diversi
  Checkers c1, c2(c1);
  c1 = c2;
} ///:~
```

I costruttori e l'operatore = per **GameBoard** e **Gioco** annunciano loro stessi quindi si può vedere quando sono usati dal compilatore. In aggiunta, l'operatore **Other( )** compie conversione del tipo automatica da un oggetto **Game** a un oggetto incorporato della classe **Other**. La classe **Chess** eredita semplicemente da **Game** e non crea funzioni (per vedere come il compilatore risponde). La funzione **f()** prende un oggetto **Other** per esaminare la funzione di conversione di tipo automatica.

In **main( )**, il costruttore di default sintetizzato ed il costruttore di copia per la classe derivata **Chess** vengono chiamati. Le versioni **Game** di questi costruttori sono chiamate come parte della gerarchia delle chiamate del costruttore. Anche se assomiglia all'ereditarietà, costruttori nuovi vengono sintetizzati davvero dal compilatore. Come ci si aspetterebbe, nessun costruttore con argomenti viene creato automaticamente, perchè ciò è troppo per il compilatore.

L'operatore = è sintetizzato anche come una funzione nuova in **Chess** usando l'assegnamento membro a membro (quindi, la versione della classe base viene chiamata) perchè, quella funzione non è stata scritta esplicitamente nella classe nuova. E chiaramente il distruttore è stato sintetizzato automaticamente dal compilatore.

A causa di tutti queste regole sul rimaneggiamento delle funzioni che gestiscono la creazione dell'oggetto, può sembrare un pò strano a prima vista che l'operatore di conversione di tipo automatico venga ereditato. Ma non è troppo irragionevole se ci sono abbastanza pezzi in **Game** per fare un oggetto **Other**, quei pezzi sono ancora là in qualsiasi cosa sia derivata da **Game** e l'operatore di conversione di tipo ancora è valido (anche se si può infatti voler ridefinirlo).

L'operatore = è sintetizzato *solamente* per assegnare oggetti dello stesso tipo. Se si vuole assegnare uno tipo ad un altro si deve sempre scrivere il proprio operatore = .

Se si guarda più da vicino **Game**, si vede che il costruttore di copia e gli operatori di assegnazione hanno chiamate esplicite al costruttore di copia dell'oggetto membro e all'operatore di assegnazione. Si farà normalmente ciò perchè, nel caso del costruttore di copia, il costruttore dell'oggetto membro di default verrà altrimenti usato e, nel caso dell'operatore di assegnazione, nessuna assegnazione sarà fatta per gli oggetti membro!

Infine, si guardi a **Checkers** dove esplicitamente è scritto il costruttore di default, costruttore di copia e l'operatore di assegnazione. Nel caso del costruttore di default, il costruttore della classe base di default è stato chiamato automaticamente e questo tipicamente è quello che si vuole. Ma e questo è un importante punto, appena si decide di scrivere il proprio proprio costruttore di copia e operatore di assegnazione, il compilatore presume che si sa ciò che si fa e non chiama automaticamente le versioni della classe base, come fa nelle funzioni sintetizzate. Se si vuole che le versioni della classe basi siano chiamate (e tipicamente si vuole) poi li si devono chiamare esplicitamente. Nel costruttore di copia della **Checkers**, questa chiamata appare nella lista di inizializzazione del costruttore.

```
Checkers(const Checkers& c) : Game(c) {
```

Nell'operatore assegnazione di **Checkers**, la classe base è la prima linea del corpo della funzione:

```
Game::operator=(c);
```

Queste chiamate dovrebbero essere parte della forma canonica che si usa ogni volta che si eredita una classe.

Ereditarietà e funzioni membro statiche

Le funzioni membro statiche agiscono allo stesso modo delle funzioni membro non-statiche:

1. Ereditano nella classe derivata.
2. Se si ridefinisce un membro statico, tutte le altre funzioni sovraccaricate nella classe base sono occultate.
3. Se si cambia la firma di una funzione nella classe base, tutte le versioni della classe base con quel nome di funzione sono occultate (questa realmente è una variazione del punto precedente).

Tuttavia funzioni membro statiche non possono essere **virtual** (un argomento trattato completamente nel Capitolo 15).

Scegliere tra composizione ed ereditarietà

Sia la composizione che l'ereditarietà piazzano suboggetti nella propria classe nuova. Entrambe usano la lista di inizializzazione del costruttore per costruire questi suboggetti. Ci si può ora star chiedendo qual è la differenza tra i due e quando scegliere uno o l'altro.

La composizione generalmente si usa quando si vogliono le caratteristiche di una classe esistente nella propria classe nuova, ma non la sua interfaccia. Ovvero, si ingloba un oggetto per perfezionare caratteristiche della classe nuova, ma l'utente della classe nuova vede l'interfaccia definita piuttosto che l'interfaccia della classe originale. Si segue il percorso tipico di inglobare oggetti privati di classi esistenti nella propria classe nuova per fare questo.

Ha comunque, di quando in quando, senso permettere all'utente della classe di accedere direttamente la composizione della classe nuova, ovvero, fare i membri oggetto **public**. I

membri oggetto usano essi stessi il controllo di accesso, così questa è una cosa sicura da fare e quando l'utente sa che si stanno assemblando un gruppo di parti, rende l'interfaccia più facile da capire. Una classe **Car** è un buon esempio:

```
//: C14:Car.cpp
// Composizione public
class Engine {
public:
  void start() const {}
  void rev() const {}
  void stop() const {}
};

class Wheel {
public:
  void inflate(int psi) const {}
};

class Window {
public:
  void rollup() const {}
  void rolldown() const {}
};

class Door {
public:
  Window window;
  void open() const {}
  void close() const {}
};

class Car {
public:
  Engine engine;
  Wheel wheel[4];
  Door left, right; // 2-door
};

int main() {
  Car car;
  car.left.window.rollup();
  car.wheel[0].inflate(72);
} ///:~
```

Poichè, la composizione di una **Car** fa parte dell'analisi del problema (e non semplicemente parte del progetto), fare i membri **public** aiuta il programmatore client a capire come usare la classe e richiede meno complessità del codice per il creatore della classe.

Pensandoci un pò, si vedrà anche che non avrebbe senso comporre una **Car** usando un oggetto "Vehicle" una macchina non contiene un veicolo, è un veicolo. La relazione *è-un e*spressa con l'ereditarietà e la relazione *ha-un* è espressa con la composizione.

Subtyping

Ora si supponga che si vuole creare un tipo di oggetto dell'**ifstream** che non solo apre un file ma anche monitorizza il nome del file. Si può usare la composizione e inglobare un **ifstream** ed una **string** nella classe nuova:

```
//: C14:FName1.cpp
// Un fstream con un nome di file
#include "../require.h"
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

class FName1 {
  ifstream file;
  string fileName;
  bool named;
public:
  FName1() : named(false) {}
  FName1(const string& fname)
    : fileName(fname), file(fname.c_str()) {
    assure(file, fileName);
    named = true;
  }
  string name() const { return fileName; }
  void name(const string& newName) {
    if(named) return; // Non sovrascrive
    fileName = newName;
    named = true;
  }
  operator ifstream&() { return file; }
};

int main() {
  FName1 file("FName1.cpp");
  cout << file.name() << endl;
  // Errore: close() non è  un membro:
//!  file.close();
} ///:~
```

C'è un problema qui, tuttavia. Viene fatto un tentativo per permettere dovunque l'uso dell'oggetto **FName1**, un oggetto dell'**ifstream** è usato includendo un operatore di conversione di tipo automatico da **FName1** a un **ifstream&**. Ma nel main, la linea

```
file.close();
```

non verrà compilata perchè, la conversione di tipo automatica accade solamente nelle chiamate di funzione, non durante la selezione del membro. Quindi quest'approccio non funziona.

Un secondo approccio è aggiungere la definizione di **close( )** a **FName1**:

```
void close() { file.close(); }
```

Questo funzionerà se ci sono solamente alcune funzioni che si vogliono portare dalla classe **ifstream**. In quel caso si usa solamente parte della classe ed è adatta la composizione.

E se si vuole passare tutto nella classe ? Questo è detto *subtyping* (sottotipare) perchè, si fa un tipo nuovo da un tipo esistente e si vuole un tipo nuovo per avere precisamente la stessa interfaccia del tipo esistente (più qualsiasi altra funzione membro che si vuole aggiungere), quindi si può usarlo dovunque si userebbe il tipo esistente. Ecco dove l'eredità è essenziale. Si può vedere che sottotipare risolve perfettamente il problema nell'esempio precedente:

```
//: C14:FName2.cpp
// Sottotipare risolve il problema
#include "../require.h"
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

class FName2 : public ifstream {
  string fileName;
  bool named;
public:
  FName2() : named(false) {}
  FName2(const string& fname)
    : ifstream(fname.c_str()), fileName(fname) {
    assure(*this, fileName);
    named = true;
  }
  string name() const { return fileName; }
  void name(const string& newName) {
    if(named) return; // Non sovrascrive
    fileName = newName;
    named = true;
  }
};

int main() {
  FName2 file("FName2.cpp");
  assure(file, "FName2.cpp");
  cout << "nome: " << file.name() << endl;
  string s;
  getline(file, s); // Anche questo funziona!
  file.seekg(-200, ios::end);
  file.close();
} ///:~
```

Ora qualsiasi funzione membro disponibile per un oggetto **ifstream** è disponibile per un oggetto **FName2**. Si può vedere anche che quelle funzioni non-membro come **getline( )** che si aspettano un **ifstream** possono lavorare anche con un **FName2**. Questo perchè, un **FName2** è un tipo di **ifstream**, non ne contiene semplicemente uno. Questo è un problema molto importante che sarà esplorato alla fine di questo capitolo e nel prossimo.

Ereditarietà privata

Si può ereditare privatamente una classe base tralasciando **public** nella lista della classe base o scrivendo esplicitamente **private** (probabilmente una procedura migliore perchè, è chiaro all'utente cosa si vuole dire). Quando si eredita privatamente, si sta implementando in termini di, cioè si crea una classe nuova che ha tutti i dati e le funzionalità della classe base, ma quella funzionalità è occultata, quindi è solamente parte della realizzazione sottostante. L'utente della classe non ha accesso alla funzionalità sottostante ed un oggetto non può essere trattato come un'istanza della classe base (come era nel **FName2.cpp**).

Ci si può chiedere qual è lo scopo dell' ereditarietà privata, perchè, l'alternativa di usare la composizione per creare un oggetto **private** nella classe nuova sembra più adatta. L'ereditarietà privata è inclusa nel linguaggio per completezza, ma se per nessuna altra ragione che ridurre la confusione, di solito si userà la composizione piuttosto che l'ereditarietà privata. Ci possono essere di quando in quando comunque, situazioni dove si vuole produrre parte della stessa interfaccia come quella della classe base e respingere il

trattamento dell'oggetto come se fosse un oggetto della classe base. L'ereditarietà privata fornisce questa abilità.

Pubblicare membri privatamente ereditati

Quando si eredita privatamente, tutti i membri pubblici della classe base diventano **private**. Se si vuole che qualcuno di loro sia visibile, si usa la parola chiave **using** vicino il loro nome (senza nessun argomento o tipi di ritorno) nella sezione **public** della classe derivata:

```
//: C14:PrivateInheritance.cpp
class Pet {
public:
  char eat() const { return 'a'; }
  int speak() const { return 2; }
  float sleep() const { return 3.0; }
  float sleep(int) const { return 4.0; }
};

class Goldfish : Pet { // ereditarietà privata
public:
  using Pet::eat; // il nome publicizza il membro
using Pet::sleep; // Entrambi i membri sovraccaricati sono esposti
};

int main() {
  Goldfish bob;
  bob.eat();
  bob.sleep();
  bob.sleep(1);
//! bob.speak();// Errore: funzione membro privata
} ///:~
```

Quindi, l'ereditarietà privata è utile se si vuole occultare parte della funzionalità della classe base.

Si noti che esporre il nome di una funzione sovraccaricata, espone tutte le versioni della funzione sovraccaricata nella classe base.

Si dovrebbe far attenzione prima di usare l'ereditarietà privata invece della composizione; l'ereditarietà privata ha particolari complicazioni quando combinata con l'identificazione del tipo a runtime (questo è il tema di un capitolo nel Volume 2 di questo libro, scaricabile da *www.BruceEckel.com*).

protected

Ora che è stata presentata l'ereditarietà, la parola riservata **protected** finalmente ha un significato. In un mondo ideale, membri privati sarebbero sempre **private**, ma nei progetti veri a volte si vuole occultare qualcosa e ancora permettere l'accesso ai membri delle classi derivate. La parola riservata **protected** è un cenno al pragmatismo; dice:"Questo è privato per quanto concerne l'utente della classe, ma disponibile a chiunque eredita da questo classe".

Il migliore approccio è lasciare i membri dato **private**, si dovrebbe sempre preservare il proprio diritto di cambiare l'implementazione sottostante. Si può permettere poi l'accesso controllato a chi eredita dalla propria classe attraverso funzioni membro **protected**:

```
//: C14:Protected.cpp
// La parola riservata protected
#include <fstream>
using namespace std;

class Base {
  int i;
protected:
  int read() const { return i; }
  void set(int ii) { i = ii; }
public:
  Base(int ii = 0) : i(ii) {}
  int value(int m) const { return m*i; }
};

class Derived : public Base {
  int j;
public:
  Derived(int jj = 0) : j(jj) {}
  void change(int x) { set(x); }
};

int main() {
  Derived d;
  d.change(10);
} ///:~
```

Si troveranno esempi del uso di **protected** più avanti in questo libro e nel Volume 2.

Ereditarietà protetta

Quando si eredita, per default la classe base è **private**, che significa che tutte le funzioni membro pubbliche sono private all'utente della classe nuova. Normalmente, si rende l'ereditarietà pubblica in modo che l'interfaccia della classe base è anche l'interfaccia della classe derivata. Si può usare anche la parola chiave **protected** con l'ereditarietà.

La derivazione protetta vuole dire: "implementata in termini di" altre classi ma "è-un" per classi derivate e friend. Non si usa molto spesso, ma è presente nel linguaggio per completezza.

Operatore overloading & ereditarietà

Tranne per l'operatore assegnamento, gli operatori sono ereditati automaticamente in una classe derivata. Questo può essere dimostrato ereditando da **C12:Byte.h**:

```
//: C14:OperatorInheritance.cpp
// Ereditare operatori sovraccaricati
#include "../C12/Byte.h"
#include <fstream>
using namespace std;
```

```
ofstream out("ByteTest.out");

class Byte2 : public Byte {
public:
  // I costruttori non ereditano:
  Byte2(unsigned char bb = 0) : Byte(bb) {}
  // l'operatore non eredità, ma
  // viene sintetizzato per assegnazione membro a membro.
  // Tuttavia, solo l'operatore = per StessoTipo = StessoTipo
  // viene sintetizzato, quindi si devono
  // scrivere gli altri :
  Byte2& operator=(const Byte& right) {
    Byte::operator=(right);
    return *this;
  }
  Byte2& operator=(int i) {
    Byte::operator=(i);
    return *this;
  }
};

// funzione di test come  in C12:ByteTest.cpp:
void k(Byte2& b1, Byte2& b2) {
  b1 = b1 * b2 + b2 % b1;

  #define TRY2(OP) \
    out << "b1 = "; b1.print(out); \
    out << ", b2 = "; b2.print(out); \
    out << ";  b1 " #OP " b2 produce"; \
    (b1 OP b2).print(out); \
    out << endl;

  b1 = 9; b2 = 47;
  TRY2(+) TRY2(-) TRY2(*) TRY2(/)
  TRY2(%) TRY2(^) TRY2(&) TRY2(|)
  TRY2(<<) TRY2(>>) TRY2(+=) TRY2(-=)
  TRY2(*=) TRY2(/=) TRY2(%=) TRY2(^=)
  TRY2(&=) TRY2(|=) TRY2(>>=) TRY2(<<=)
  TRY2(=) //  operator assegnazione

  // Conditionals:
  #define TRYC2(OP) \
    out << "b1 = "; b1.print(out); \
    out << ", b2 = "; b2.print(out); \
    out << ";  b1 " #OP " b2 produce"; \
    out << (b1 OP b2); \
    out << endl;

  b1 = 9; b2 = 47;
  TRYC2(<) TRYC2(>) TRYC2(==) TRYC2(!=) TRYC2(<=)
  TRYC2(>=) TRYC2(&&) TRYC2(||)

  // assegnazione a catena:
  Byte2 b3 = 92;
  b1 = b2 = b3;
}

int main() {
  out << "funzioni membro:" << endl;
  Byte2 b1(47), b2(9);
  k(b1, b2);
} ///:~
```

Il codice di prova è identico a quello nel **C12:ByteTest.cpp** tranne che **Byte2** è usato al posto di **Byte**. In questo modo tutti gli operatori funzionanto con **Byte2** mediante l'ereditarietà.

Quando si esamina la classe **Byte2**, si vede che il costruttore deve essere definito esplicitamente e che solamente l'operatore = che assegna un **Byte2** a un **Byte2** viene sintetizzato; qualsiasi altro operatore di assegnazione di cui si ha bisogno deve essere scritto per proprio conto.

Ereditarietà multipla

Si può ereditare da una classe, quindi sembrerebbe avere senso ereditare da più di una classe alla volta. Davvero si può, ma se ha senso nella parte di un progetto è soggetto di continuo dibattito. Su una cosa generalmente si è d'accordo: non la si dovrebbe provare fino a che non si programma da un pò e si capisce completamente il linguaggio. Per quel tempo, si comprenderà che probabilmente dove servirebbe assolutamente l'ereditarietà multipla, quasi sempre va bene l'ereditarietà singola.

L'ereditarietà multipla sembra inizialmente abbastanza semplice: si aggiungono più classi nella lista della classe base durante l'ereditarietà, separata da virgole. Comunque, l'ereditarietà multipla presenta delle ambiguità, perciò un capitolo nel Volume 2 è dedicato all'argomento.

Sviluppo incrementale

Uno dei vantaggi dell' ereditarietà e della composizione è che questi sostengono lo *sviluppo incrementale* permettendo di presentare codice nuovo senza causare bachi nel codice esistente. Se appaiono bachi, sono isolati nel codice nuovo. Ereditando da (o componendo con) una esistente e funzionale classe, aggiungendo membri dato e funzioni membro (e ridefinendo funzioni membro esistenti durante l'ereditarietà) si lascia il codice esistente · che qualcuno altro ancora può star usando · intatto e non bacato. Se c'è un baco, si sa che è nel codice nuovo che è molto più corto e più facile leggere che se si avesse cambiato il corpo del codice esistente.

Stupisce piuttosto come le classi siano separate. Non si ha nemmeno bisogno del codice sorgente per le funzioni membro per riusare il codice, solo l'header file che descrive la classe e il file oggetto o la libreria con le funzioni membro compilate (questo è vero sia per l'ereditarietà che per la composizione).

È importante rendersi conto che lo sviluppo del programma è un processo incrementale, proprio come l'apprendimento umano. Si può fare tanta analisi quanto si vuole, ma ancora non si conosceranno tutte le risposte quando si intraprenderà un progetto. Si avrà molto più successo e responsi più immediati se si incomincia a far crescere il proprio progetto come una creatura organica, evolutiva, piuttosto che costruendolo tutto in una volta come un grattacielo [52].

Sebbene l'ereditarietà per sperimentazione sia una tecnica utile, a un certo punto dopo che le cose si stabilizzano, si ha bisogno di dare alla propria gerarchia di classe un nuovo look, collassando tutto in una struttura assennata[53]. Si ricordi che l'ereditarietà serve a esprimere una relazione che dice: "Questa nuova classe è un tipo di quella classe vecchia".

Il proprio programma non si dovrebbe preoccupare di gestire bit, ma invece di crere e manipolare oggetti di vari tipio per esprimere un modello nei termini dati dallo spazio del problema.

Upcasting (cast all'insù)

Nei primo capitoli, si è visto come un oggetto di una classe derivata da **ifstream** ha tutte le caratteristiche e comportamenti di un oggetto dell'**ifstream**. In **FName2.cpp**, qualsiasi funzione membro di **ifstream** potrebbe essere chiamata per un oggetto **FName2**.

Il più importante aspetto dell' ereditarietà non è che fornisce funzioni membro per la classe nuova, comunque. È la relazione espressa tra la classe nuova e la classe base. Questa relazione può essere ricapitolata dicendo: " La classe nuova *è un tipo della* classe esistente ".

Questa descrizione non è solo un modo fantastico di spiegare l' ereditarietà, è sostenuto direttamente dal compilatore. Come esempio, si consideri una classe base chiamata **Instrument** che rappresenta strumenti musicali e una classe derivata chiamata **Wind**. Poichè ereditare significa che tutte le funzioni della classe base sono anche disponibili nella classe derivata, qualsiasi messaggio che si può spedire alla classe base può essere spedito anche alla classe derivata. Quindi se la classe **Instrument** ha una funzione membro **play( )**, allo stesso modo l'avrà **Wind**. Questo vuol dire che noi possiamo dire accuratamente che un oggetto **Wind** è anche un tipo **Instrument**.
L'esempio seguente mostra come il compilatore sostiene questa nozione:

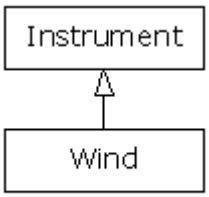
```
//: C14:Instrument.cpp
// Ereditarietà & upcasting
enum note { middleC, Csharp, Cflat }; // Etc.

class Instrument {
public:
  void play(note) const {}
};

// oggetti Wind  sono  Instruments
// perchè hannno la stessa interfaccia:
class Wind : public Instrument {};

void tune(Instrument& i) {
  // ...
  i.play(middleC);
}

int main() {
  Wind flute;
  tune(flute); // Upcasting
} ///:~
```

Ciò che è interessante in questo esempio è la funzione **tune( )**, che accetta un riferimento **Instrument**. Tuttavia, in **main( )** la funzione **tune( )** viene chiamata gestendo un riferimento a un oggetto **Wind**. Dato che il C++ è molto particolare circa il controllo del tipo, sembra strano che una funzione che accetta un tipo accetterà prontamente un altro tipo, finchè si comprenderà che un oggetto **Wind** è anche un oggetto **Instrument**, e non c'è nessuna funzione che **tune( )** potrebbe chiamare per un **Instrument** che non è anche

in **Wind** (ciò è quello che l'ereditarietà garantisce). In **tune( )** il codice lavora per **Instrument** e qualsiasi cosa si deriva da **Instrument** ; l'atto di convertire un riferimento o puntatore **Wind** in un riferimento o puntatore **Instrument** è chiamato *upcasting*.

Perchè "upcasting?"

La ragione per il termine è storica ed è basata sul modo in cui i diagrammi delle classi ereditate sono disegnate tradizionalmente: con la classe base alla cima della pagina. (Chiaramente, si può disegnare i diagrammi in qualsiasi modo si trova utile.) Il diagramma dell'eredità per **Instrument.cpp** è:



Castando dalla derivata alla base ci si muove in su sul diagramma dell'eredità, quindi ci si riferisce comunemente come upcasting. L'upcasting è sempre sicuro perché si va da un tipo più specifico ad un tipo più generale, l'unica cosa che può accadere all'interfaccia della classe è che può perdere funzioni membro, non guadagnarle. Questo accade perchè il compilatore permette l'upcasting senza qualsiasi cast esplicito o altre notazioni speciale.

Upcasting ed il costruttore di copia

Se si permette al compilatore di sintetizzare un copia di costruttore per una classe derivata, esso chiamerà automaticamente il costruttore di copia della classe base e poi i costruttori di copia per tutto gli oggetti membro (o compie una copia bit a bit sui tipi predefiniti) quindi si ottiene il giusto comportamento :

```
//: C14:CopyConstructor.cpp
// creare correttamente il costruttore di copia
#include <iostream>
using namespace std;

class Parent {
  int i;
public:
  Parent(int ii) : i(ii) {
    cout << "Parent(int ii)\n";
  }
  Parent(const Parent& b) : i(b.i) {
    cout << "Parent(const Parent&)\n";
  }
  Parent() : i(0) { cout << "Parent()\n"; }
  friend ostream&
    operator<<(ostream& os, const Parent& b) {
    return os << "Parent: " << b.i << endl;
  }
};

class Member {
  int i;
public:
```

```
  Member(int ii) : i(ii) {
    cout << "Member(int ii)\n";
  }
  Member(const Member& m) : i(m.i) {
    cout << "Member(const Member&)\n";
  }
  friend ostream&
    operator<<(ostream& os, const Member& m) {
    return os << "Member: " << m.i << endl;
  }
};

class Child : public Parent {
  int i;
  Member m;
public:
  Child(int ii) : Parent(ii), i(ii), m(ii) {
    cout << "Child(int ii)\n";
  }
  friend ostream&
    operator<<(ostream& os, const Child& c){
    return os << (Parent&)c << c.m
              << "Child: " << c.i << endl;
  }
};

int main() {
  Child c(2);
  cout << "chiamo  il costruttore di copia: " << endl;
  Child c2 = c; // chiama  il costruttore di copia
  cout << "values in c2:\n" << c2;
} ///:~
```

L'operatore **<<** per **Child** è interessante per il modo in cui chiama l'operatore << per la parte **Parent** in esso: castando l'oggetto **Child** a un **Parent&** (se si casta a un oggetto della classe base invece di un riferimento di solito si ottengono risultati indesiderati):

```
return os << (Parent&)c << c.m
```

Poichè il compilatore poi vede un **Parent,** chiama la versione **Parent** dell'operatore <<.

Si vede che un **Child** non ha un costruttore di copia esplicitamente dichiarato. Il compilatore quindi sintetizza il costruttore di copia (poichè è una delle quattro funzioni che esso sintetizza, insieme al costruttore di default, se non si crea nessun costrutore, l'operatore = ed il distruttore) chiamando il costruttore di copia **Parent** ed il costruttore di copia di **Member**. Ecco l'output:

```
Parent(int ii)
Member(int ii)
Child(int ii)
chiamo il costruttore di copia:
Parent(const Parent&)
Member(const Member&)
values in c2:
Parent: 2
Member: 2
Child: 2
```

Tuttavia, se si prova a scrivere il proprio costruttore di copia per **Child** e si fa un errore innocente :

```
Child(const Child& c) : i(c.i), m(c.m) {}
```

allora il costruttore di *default* verrà chiamato automaticamente per la parte della classe base di **Child**, poichè ciò è quello che il compilatore fa quando non nessun altro costruttore da chiamare (si ricordi che un costruttore deve essere sempre chiamato per ogni oggetto, anche se è un suboggetto di un'altra classe). L'output sarà:

```
Parent(int ii)
Member(int ii)
Child(int ii)
chiamo il costruttore di copia:
Parent()
Member(const Member&)
values in c2:
Parent: 0
Member: 2
Child: 2
```

Questo non è probabilmente ciò che ci si aspettava, poichè generalmente si vorrà che la porzione della classe base sia copiata da un oggetto esistente ad un nuovo oggetto come parte del costruttore di copia.

Per rimediare al problema si deve ricordare di chiamare propriamente il costruttore di copia della classe base (come il compilatore fa) ogni qualvolta si scrive il proprio costruttore di copia . All'inizio ciò può sembrare un pò strano ma ecco un altro esempio di upcasting:

```
  Child(const Child& c)
    : Parent(c), i(c.i), m(c.m) {
    cout << "Child(Child&)\n";
 }
```

La parte strana è dove il costruttore di copia di **Parent** viene chiamato: **Parent(c)**. Che vuol dire passare un oggetto **Child** a un costruttore di **Parent**? Ma **Child** è ereditato da **Parent**, quindi un riferimento a **Child** è una riferimento a **Parent**. Il costruttore di copia della classe base fa l'upcasting di riferimento a **Child** ad un riferimento a **Parent** ed lo usa per eseguire il costruttore di copia. Quando si scrivono i propri costruttori di copia, si vuole che essi facciano quasi sempre la stessa cosa.

Composizione ed ereditarietà (rivisitata)

Uno dei modi più chiari di determinare se si deve usare la composizione o l'ereditarietà è chiedersi se si avrà mai bisogno di un upcasting dalla propria nuova classe.
Precedentemente in questo capitolo, la classe della **Stack** è stata specializzata usando l'ereditarietà. Tuttavia, le gli oggetti di **StringStack** saranno usati solamente come contenitori di **string** e mai con l'upcasingt, quindi un'alternativa
più adatta è la composizione:

```
//: C14:InheritStack2.cpp
// composizione ed ereditarietà
#include "../C09/Stack4.h"
#include "../require.h"
```

```
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

class StringStack {
  Stack stack; // incorporato invece di ereditato
public:
  void push(string* str) {
    stack.push(str);
  }
  string* peek() const {
    return (string*)stack.peek();
  }
  string* pop() {
    return (string*)stack.pop();
  }
};

int main() {
  ifstream in("InheritStack2.cpp");
  assure(in, "InheritStack2.cpp");
  string line;
  StringStack textlines;
  while(getline(in, line))
    textlines.push(new string(line));
  string* s;
  while((s = textlines.pop()) != 0) // Nessun cast!
    cout << *s << endl;
} ///:~
```

Il file è identico a **InheritStack.cpp**, tranne che un oggetto **Stack** è incorporato in **StringStack** e le funzioni membro vengono chiamate per l'oggetto incorporato. Non c'è ancora overhead di tempo o spazio perché il suboggetto prende lo stesso ammontare di spazio e tutto il controllodi tipo supplementare avviene a tempo di compilazione.

Sebbene può confondere, si può anche usare l'ereditarietà privata per esprimere "implementato in termini di". Anche ciò risolverebbe il problema adeguatamente. Un posto in cui questo diventa importante, tuttavia, è dove l'ereditarietà multipla dovrebbe essere garantita. In quel caso, se si vede un progetto in cui la composizione può essere usata invece dell'ereditarietà, si può eliminare il bisogno dell'ereditarietà multipla.

Upcasting di puntatori & riferimenti

In **Instrument.cpp**, l'upcasting avviene durante la chiamata a funzione, viene preso il riferimento ad un oggetto **Wind** esterno alla funzione e diventa un riferimento **Instrument** dentro la funzione. L' upcasting avviene anche con una semplice assegnazione ad un puntatore o riferimento:

```
Wind w;
Instrument* ip = &w; // Upcast
Instrument& ir = w; // Upcast
```

Come la chiamata a funzione, nessuno di questi casi richiede un cast esplicito.

Un problema

Naturalmente, con l'upcasting si perde l'informazione del tipo di un oggetto. Se si scrive:

```
Wind w;
Instrument* ip = &w;
```

il compilatore può trattare un **ip** solo come un puntatore **Instrument** e nient'altro. Cioè non sa che **ip** punta a un oggetto **Wind**. Quindi quando si chiama la funzione membro **play()** scrivendo:

```
ip->play(middleC);
```

il compilatore conosce solo che sta chiamando **play()** per un puntatore **Instrument** e chiama la versione di **Instrument::play()** della classe base invece di ciò che dovrebbe fare, che è chiamare **Wind::play( )**. Quindi non si otterrà il giusto comportamento.

Questo è un problema serio ed è risolto nel Capitolo 15, dove viene presentata la terza pietra miliare della programmazione ad oggetti: il polimorfismo (implementato in C++ con le funzioni virtuali).

Sommario

Sia l'ereditarietà che la composizione permettono di creare un tipo nuovo da tipi esistenti ed entrambi inglobano suboggetti di tipi esistenti nel tipo nuovo. Si usa comunque, tipicamente, la composizione per riutilizzare tipi esistenti come parte della implementazione sottostante del tipo nuovo e l'ereditarietà quando si vuole costringere il tipo nuovo ad essere dello stesso tipo della classe base (l'equivalenza dei tipi garantisce l'equivalenza delle interfacce). Poichè la classe derivata ha l'interfaccia della classe base, si può usare l'upcasting verso la classe base, che è critico per il polimorfismo come si vedrà nel Capitolo 15.

Sebbene il riutilizzo del codice attraverso composizione ed ereditarietà sia molto utile per lo sviluppo rapido dei progetti, generalmente si vuole ridisegnare la propria gerarchia di classe prima di permettere agli altri programmatori di divenire dipendenti da essa. La meta è una gerarchia nella quale ciascuna classe ha un uso specifico e nessuna è nè troppo grande (comprendendo così molte funzionalità
che sono difficili da riusare) né molestamente piccolo (non la si può usare di per se o senza aggiungere funzionalità).

---

[51] In Java, il compilatore non permetterà di diminuire l'accesso di un membro con l'ereditarietà.

[52] Per imparare qualcosa di più circa questa idea, si veda *Extreme Programming Explained* di Kent Beck (Addison-Wesley 2000).

[53] Si veda *Refactoring: Improving the Design of Existing Code* di Martin Fowler (Addison-Wesley 1999).

15: Polimorfismo & Funzioni Virtuali

Il polimorfismo (implementato in C++ mediante le funzioni virtuali) è la terza caratteristica fondamentale di un linguaggio di programmazione object-oriented, dopo l'astrazione dei dati e l'ereditarietà.

Il polimorfismo fornisce un ulteriore grado di separazione tra l'interfaccia e l'implementazione, per separare il *"cosa fare"* dal *"come farlo"*. Il polimorfismo permette una migliore organizzazione e leggibilità del codice, così come la creazione di programmi *espandibili*, cioè capaci di "crescere" non solo durante la prima stesura del codice, ma anche quando si desidera aggiungere nuove funzionalità.

L'incapsulamento permette di creare nuovi tipi mettendo insieme i dati (membri) e i servizi (metodi). Il controllo d'accesso separa l'interfaccia dall'implementazione dichiarando tutti i dettagli come privati. Questo modo di strutturare il codice ha un senso e può essere facilmente capito da chi proviene dalla programmazione strutturata. Le funzioni virtuali, invece, riguardano il concetto di separazione dei *tipi*. Nel capitolo 14, è stato mostrato come l'ereditarietà permetta di trattare un oggetto come un'istanza della propria classe oppure della classe base. Questa possibilità è critica perché permette che tipi diversi (derivati dallo stesso tipo base) vengano trattati come un tipo unico e che un pezzo di codice possa funzionare nello stesso modo con tutti i tipi. L'uso delle funzioni virtuali permette di esprimere le differenze di comportamento tra tipi derivati dallo stesso tipo base. Questa distinzione è espressa mediante comportamenti diversi de

In questo capitolo, verranno trattate le funzioni virtuali, partendo dai principi con esempi semplici che rimuoveranno il concetto di programma "virtuale".

Evoluzione dei programmatori C++

Sembra che i programmatori C imparino il C++ in tre passi. Il primo, semplicemente come un "C migliore", perché il C++ obbliga a dichiarare tutte le funzioni prima che queste vengano usate ed a scegliere attentamente come usare le variabili. Spesso si possono trovare gli errori in un programma C semplicemente compilandolo con un compilatore C++.

Il secondo passo è quello di imparare un C++ "basato su oggetti". In questa fase si apprezzano i benefici portati all'organizzazione del codice dal raggruppare le strutture dati insieme alle funzioni che lavorano su di esse, il valore dei costruttori, dei distruttori e la possibilità di utilizzare un primo livello di ereditarietà. Molti programmatori che hanno lavorato con il C per un periodo, notano subito l'utilità di tutto ciò perché, ogni volta che creano una libreria, questo è quello che cercano di fare. Con il C++, in questo, si viene aiutati dal compilatore.

Ci si può fermare a questo livello ed utilizzare il C++ come un linguaggio basato su oggetti, perché si arriva presto a questo livello di comprensione e si ottengono molti benefici senza un grande sforzo intellettuale. Si creano tipi di dati, si costruiscono classi e oggetti, si mandano messaggi a questi oggetti e tutto sembra bello e semplice.

Ma non bisogna farsi ingannare. Se ci si ferma a questo punto, si lascia da parte la parte più grande del C++, vale a dire il passaggio alla vera programmazione object-oriented. Questa può essere fatta solo con l'uso delle funzioni virtuali.

Le funzioni virtuali ampliano il concetto di tipo e sono qualcosa di diverso rispetto all'incapsulamento del codice in strutture e dietro muri che rendono parte dell'implementazione inaccessibile; senza dubbio, quindi, le funzioni virtuali rappresentano il concetto più difficile da affrontare per i programmatori C++ alle prime armi. Allo stesso tempo, rappresentano anche il punto di svolta nella comprensione della programmazione object-oriented. Se non si usano funzioni virtuali, non si è ancora capita l'OOP.

Siccome il concetto di funzione virtuale è legato intimamente con quello di tipo ed il tipo è il cuore della programmazione object-oriented, non c'è niente di analogo alle funzioni virtuali in un linguaggio di programmazione procedurale tradizionale. Dal punto di vista di un programmatore "procedurale", non c'è nessun riferimento a cui pensare per tracciare un analogia con le funzioni virtuali, al contrario delle altri aspetti caratteristici del C++. Le caratteristiche di un linguaggio procedurale possono essere capite da un punto di vista algoritmico, mentre le funzioni virtuali possono essere capite solo dal punto di visto della progettazione del software.

Upcasting

Nel capitolo 14 si è visto come un oggetto può essere usato come se stesso oppure come un istanza della tipo base. Un oggetto può essere manipolato anche attraverso un indirizzo del tipo base. L'operazione di usare l'indirizzo di un oggetto (un puntatore o un riferimento) e di trattarlo come l'indirizzo al tipo base viene chiamata *upcasting* (casting "all'insù") perché i diagrammi delle classi vengono disegnati con la classe base in cima.

Si può notare che sorge presto un problema, come mostrato dal codice seguente:

```
//: C15:Instrument2.cpp
// Ereditarietà & upcast
#include <iostream>
using namespace std;
enum note { middleC, Csharp, Eflat }; // Etc.

class Instrument {
public:
  void play(note) const {
    cout << "Instrument::play" << endl;
  }
};

// Gli oggetti Wind sono Instruments
// perché hanno la stessa interfaccia:
class Wind : public Instrument {
public:
  // Ridefinisce la funzione d'interfaccia:
  void play(note) const {
    cout << "Wind::play" << endl;
  }
};

void tune(Instrument& i) {
```

```
  // ...
  i.play(middleC);
}

int main() {
  Wind flute;
  tune(flute); // Upcast
} ///:~
```

La funzione **tune()** accetta (come riferimento) un oggetto **Instrument**, e , anche, senza nessuna segnalazione di errore, qualsiasi oggetto derivato da **Instrument**. Nel **main()**, si può vedere come questo avvenga quando un oggetto **Wind** viene passato a **tune()**, senza nessun bisogno di operazioni di cast. Questo è possibile; l'interfaccia di **Instrument** esiste anche in **Wind**, perché **Wind** è ereditato pubblicamente da **Instrument**. L'upcasting da Wind a Instrument può "restringere" l'interfaccia di **Wind**, ma non si potrà mai ottenere un oggetto con un interfaccia più piccola di quella di **Instrument**.

Le stesse considerazioni valgono anche quando si lavora con i puntatori; l'unica differenza è che l'utente deve fornire esplicitamente l'indirizzo dell'oggetto così come questo deve essere passato alla funzione.

Il problema

Il problema presente in **Instrument2.cpp** può essere visto semplicemente mandando in esecuzione il programma. L'uscita è quella di **Instrument::play**. Ovviamente questo non è il comportamento desiderato, perché in questo caso è noto che l'oggetto in questione è un **Wind** e non solo un **Instrument**. La chiamata alla funzione **tune()** dovrebbe produrre una chiamato a **Wind::play**. A questo scopo, ogni istanza di una classe derivata da **Instrument** dovrebbe avere la propria versione di **play()** da usare in qualsiasi situazione.

Il comportamento di Instrument2.cpp non sorprende, se si utilizza un approccio alle funzioni in stile C. Per capire la questione, bisogna introdurre il concetto di *binding*.

Binding delle chiamate a funzioni

Il collegamento della chiamata ad una funzione con il corpo della funzione stessa viene chiamato *binding*. Nel caso in cui il binding venga fatto prima che il programma vada in esecuzione (dal compilatore e dal linker), si ha l'*early binding*. Il termine binding potrebbe risultare sconosciuto ai più, perché nel caso dei linguaggi procedurali non ci sono alternative: i compilatori C hanno un solo metodo per realizzare le chiamate alle funzioni ed è l'early binding.

Il problema nel programma precedente è causato dall'early binding, perché il compilatore non può conoscere la funzione corretta da chiamare se ha a disposizione solo l'indirizzo di un un oggetto **Instrument**.

La soluzione è chiamata *late binding*, che consiste nel binding effettuato a runtime, basato sul tipo di oggetto. Il late binding viene anche chiamato *binding dinamico* o *binding a runtime*. Quando un linguaggio implementa il late binding, devono esserci dei meccanismi per poter determinare il tipo dell'oggetto a runtime e chiamare la funzione membro appropriata. Nel caso di un linguaggio compilato, il compilatore non conosce il tipo dell'oggetto che verrà passato alla funzione, ma inserisce del codice per trovare e chiamare

la funzione corretta. Il meccanismo del late binding varia da linguaggio a linguaggio, ma, in ogni caso, si può pensare che alcune informazioni devono essere inserite negli oggetti. Più avanti in questo capitolo verrà mostrato il funzionamento di questo meccanismo.

funzioni virtuali

Per fare in modo che venga effettuato il late binding per una particolare funzione, il C++ impone di utilizzare la parola chiave **virtual** quando viene dichiarata la funzione nella classe base. Il late binding viene realizzato solo per le funzioni **virtuali**, solo quando viene utilizzato un indirizzo della classe base in cui esistono delle funzioni **virtuali**; queste funzioni devono anche essere state definite nella classe base.

Per creare una funzione come **virtuale**, bisogna semplicemente far precedere la dichiarazione della funzione dalla parola chiave **virtual**. Solo la dichiarazione richiede l'uso della parola chiave **virtual** e non la definizione. Se una funzione viene dichiarata come **virtuale** nella classe base, resterà **virtuale** per tutte le classi derivate. La ridefinizione di una funzione **virtuale** in una classe derivata viene solitamente chiamata *overriding*.

Bisogna notare che l'unica cosa necessaria da fare è quella di dichiarare una funzione come **virtual** nella classe base. Tutte le funzioni delle classi derivate che avranno lo stesso prototipo verranno chiamate utilizzando il meccanismo delle funzioni virtuali. La parola chiave **virtual** può essere usata nelle dichiarazioni delle classi derivate (non c'è nessun pericolo nel farlo), ma è ridondante è può causare confusione.

Per ottenere il comportamento desiderato da **Instrument2.cpp**, basta semplicemente aggiungere la parola chiave **virtual** nella classe base prima di **play()**.

```
//: C15:Instrument3.cpp
// Late binding con parola chiave virtual
#include <iostream>
using namespace std;
enum note { middleC, Csharp, Cflat }; // Etc.

class Instrument {
public:
  virtual void play(note) const {
    cout << "Instrument::play" << endl;
  }
};

// Gli oggetti Wind sono Instruments
// perché hanno la stessa interfaccia:
class Wind : public Instrument {
public:
  // Override della funzione d'interfaccia:
  void play(note) const {
    cout << "Wind::play" << endl;
  }
};

void tune(Instrument& i) {
  // ...
  i.play(middleC);
}
```

```
int main() {
  Wind flute;
  tune(flute); // Upcast
} ///:~
```

Questo listato è identico a quella di **Instrument2.cpp** tranne per l'aggiunta della parola chiave **virtual**, ma il suo comportamento è diverso in maniera significativa: ora l'uscita corrisponde a quella di **Wind::play**.

Estendibilità

Avendo definito **play()** come **virtuale** nella classe base, si possono aggiungere tanti nuovi tipi quanti se ne desidera senza dover cambiare la funzione **tune()**. In un programma OOP ben progettato, la maggior parte delle funzioni dovrà ricalcare il modello di **tune()** e comunicare solo con l'interfaccia della classe base. Un programma di questo tipo è *estendibile* perché si possono aggiungere nuove funzionalità ereditando nuovi tipi di dati da una classe base comune. Le funzioni che interagiscono con l'interfaccia della classe base non dovranno essere cambiate per poter lavorare anche con le nuove classi.

Di seguito è riportato l'esempio degli strumenti con più funzioni virtuali e con un numero maggiore di classi nuove, le quali funzionano tutte correttamente con la vecchia funzione **tune()** che è rimasta invariata:

```
//: C15:Instrument4.cpp
// Estendibilità in OOP
#include <iostream>
using namespace std;
enum note { middleC, Csharp, Cflat }; // Etc.

class Instrument {
public:
  virtual void play(note) const {
    cout << "Instrument::play" << endl;
  }
  virtual char* what() const {
    return "Instrument";
  }
  // Si supponga che adjust() modifichi l'oggetto:
  virtual void adjust(int) {}
};

class Wind : public Instrument {
public:
  void play(note) const {
    cout << "Wind::play" << endl;
  }
  char* what() const { return "Wind"; }
  void adjust(int) {}
};

class Percussion : public Instrument {
public:
  void play(note) const {
    cout << "Percussion::play" << endl;
  }
  char* what() const { return "Percussion"; }
  void adjust(int) {}
};
```

```
class Stringed : public Instrument {
public:
  void play(note) const {
    cout << "Stringed::play" << endl;
  }
  char* what() const { return "Stringed"; }
  void adjust(int) {}
};

class Brass : public Wind {
public:
  void play(note) const {
    cout << "Brass::play" << endl;
  }
  char* what() const { return "Brass"; }
};

class Woodwind : public Wind {
public:
  void play(note) const {
    cout << "Woodwind::play" << endl;
  }
  char* what() const { return "Woodwind"; }
};

// Identica alla funzione di prima:
void tune(Instrument& i) {
  // ...
  i.play(middleC);
}

// Funzione nuova:
void f(Instrument& i) { i.adjust(1); }

// Upcast durante l'inizializzazione dell'array:
Instrument* A[] = {
  new Wind,
  new Percussion,
  new Stringed,
  new Brass,
};

int main() {
  Wind flute;
  Percussion drum;
  Stringed violin;
  Brass flugelhorn;
  Woodwind recorder;
  tune(flute);
  tune(drum);
  tune(violin);
  tune(flugelhorn);
  tune(recorder);
  f(flugelhorn);
} ///:~
```

Si noti che un altro livello di ereditarietà è stato derivato a partire da **Wind**, ma il meccanismo delle funzioni **virtuali** continua a funzionare in maniera corretta indipendentemente dal numero di livelli di derivazione delle classi. Per la funzione **adjust()** delle classi **Brass** e **Woodwind** *non* è stato applicato l'override. In questo caso,

la definizione gerarchicamente “più vicina” viene usata automaticamente – il compilatore garantisce che ci sia sempre una definizione per una funzione virtuale, quindi una chiamata ad una funzione sarà sempre collegata al corpo di una funzione. (se questo non avvenisse sarebbe disastroso).

L’array **A[]** contiene dei puntatori ad oggetti derivati dalla classe base **Instrument**, quindi delle operazione di upcasting verranno fatte durante l’inizializzazione. Questo array e la funzione **f()** verranno presi in considerazione di nuovo più avanti in questa trattazione.

Nella chiamata a **tune()**, l’upcasting viene fatto per ogni tipo di oggetto differente, in maniera da ottenere ogni volta il comportamento desiderato. Questo comportamento può essere descritto nella maniera seguente: “un messaggio viene mandato ad un oggetto e a questo viene lasciato l’onere di interpretare il messaggio ed eseguire le relative operazioni”. Le funzioni **virtuali** sono lo strumento da utilizzare quando si fa l’analisi durante la progettazione del software: A quale livello collocare le classi base? Come si vuole estendere il programma? In ogni modo, anche se non viene individuata la classe base appropriata e le funzioni virtuali nella prima stesura del programma, possono sempre essere scoperte più avanti, anche molto tardi, quando si pensa a come estendere oppure a come manutenere il programma. Questo modo di procedere non va considerato come un errore di analisi o di progetto; significa, semplicemente, che non si hanno o non si possono avere tutte le informazioni all’inizio del processo di sviluppo del software. Grazie alla modularizzazione in classi che si ha utilizzando il C++, il dover apportare modifiche al progetto non rappresenta un grande problema, perché i cambiamenti fatti in una parte del sistema tendono a non propagarsi in altre parti del sistema stesso, al contrario di quanto accade con il C.

Come viene realizzato il late binding in C++

Cosa succede quando viene fatto il late binding? Tutto il lavoro viene fatto dietro il sipario dal compilatore, che installa i meccanismi necessari per il late binding quando ne viene fatta richiesta (la richiesta viene fatta creando funzioni virtuali). Dato che i programmatori spesso traggono beneficio dalla comprensione del meccanismo delle funzioni virtuali in C++, questo paragrafo è incentrato sul modo in cui il compilatore implementa questo meccanismo.

La parola chiave **virtual** comunica al compilatore che non dovrà realizzare l’early binding. Al suo posto, dovrà automaticamente installare i meccanismi necessari per realizzare il late binding. Questo vuol dire che se verrà chiamata la funzione **play()** per un oggetto **Brass** attraverso l’indirizzo della classe base **Instrument**, si dovrà ottenere una chiamata alla funzione corretta.

Per realizzare questo meccanismo, il compilatore tipico [54] crea una tabella (chiamata VTABLE) per ogni classe che contiene funzioni **virtuali**. Il compilatore inserisce gli indirizzi delle funzioni virtuali di una classe particolare nella VTABLE. In ogni classe con funzioni virtuali, viene messo, in maniera segreta e non visibile al programmatore, un puntatore chiamato *vpointer* (abbreviato in VPTR), che punta alla VTABLE corrispondente alla classe. Quando viene fatta una chiamata ad una funzione virtuale attraverso un puntatore alla classe base (questo avviene quando viene fatta una chiamata polimorfica ad una funzione), il compilatore semplicemente inserisce del codice per caricare il VPTR e ottenere l’indirizzo della funzione contenuto della VTABLE, in modo da chiamare la funzione corretta e far si che il late binding abbia luogo.

Tutto questo – la creazione di una VTABLE per ogni classe, l'inizializzazione del VPTR, l'inserzione del codice per la chiamata ad una funzione virtuale – avviene in maniera automatica. Con le funzioni virtuali, per ogni oggetto viene chiamata la funzione corretta, anche se il compilatore non conosce lo specifico oggetto.

La sezione seguente illustrerà nel dettaglio questo meccanismo.

Memorizzazione dell'informazione sul tipo

Come si è visto precedentemente, in ogni classe non è memorizzata esplicitamente alcuna informazione riguardante il tipo. Gli esempi precedenti ed il buonsenso, però, portano a pensare che questa informazione deve essere memorizzata negli oggetti, altrimenti il tipo non potrebbe essere stabilito durante l'esecuzione del programma. Questa informazione c'è, quindi, ma è nascosta. Ecco un esempio che mette in evidenza la presenza di questa informazione confrontando le dimensioni delle classi che fanno uso di funzioni virtuali e delle classi che non ne fanno uso:

```
//: C15:Sizes.cpp
// Dimensione di oggetti con o senza funzioni virtuali
#include <iostream>
using namespace std;

class NoVirtual {
  int a;
public:
  void x() const {}
  int i() const { return 1; }
};

class OneVirtual {
  int a;
public:
  virtual void x() const {}
  int i() const { return 1; }
};

class TwoVirtuals {
  int a;
public:
  virtual void x() const {}
  virtual int i() const { return 1; }
};

int main() {
  cout << "int: " << sizeof(int) << endl;
  cout << "NoVirtual: "
       << sizeof(NoVirtual) << endl;
  cout << "void* : " << sizeof(void*) << endl;
  cout << "OneVirtual: "
       << sizeof(OneVirtual) << endl;
  cout << "TwoVirtuals: "
       << sizeof(TwoVirtuals) << endl;
} ///:~
```

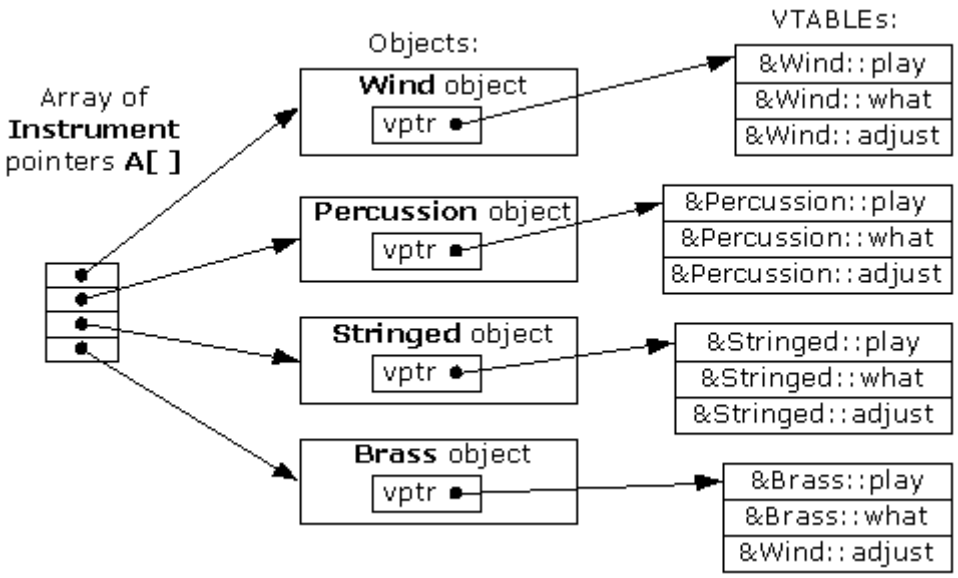
Senza funzioni virtuali, la dimensione dell'oggetto è esattamente quella che ci si aspetterebbe : la dimensione di un singolo[55] **int**. **OneVirtual** ha una sola funzione virtuale e la dimensione di una sua istanza è pari a quella di un'istanza di **NoVirtual** sommata alla dimensione di un puntatore a **void**. Da questo esempio si deduce che il

compilatore inserisce un unico puntatore (il VPTR) se c'è *almeno una* funzione virtuale. Infatti non c'è alcuna differenza tra le dimensioni di **OneVirtual** e **TwoVirtual**. Questo accade perché tutti gli indirizzi delle funzioni virtuali sono contenute in un'unica tabella.

In questo esempio è necessario che ci sia almeno un dato membro nelle classi. Se non ci fossero stati dati membri, infatti, il compilatore C++ avrebbe forzato gli oggetti ad avere dimensione diversa da zero, perché ogni oggetto deve avere un proprio indirizzo diverso dagli altri. Per poter capire meglio questa affermazione si provi ad immaginare come si possa accedere agli elementi di un array di oggetti a dimensione nulla. Un membro fittizio viene inserito negli oggetti, che altrimenti avrebbero dimensione nulla. Quando viene utilizzata la parola chiave **virtual**, viene inserita l'informazione riguardane il tipo che prende il posto del membro fittizio. Si provi a commentare **int a** in tutte le classi dell'esempio precedente per osservare questo comportamento.

Rappresentazione delle funzioni virtuali

Per comprendere esattamente cosa succede quando viene utilizzata una funzione virtuale, può essere d'aiuto visualizzare le operazioni che avvengono dietro il sipario. Di seguito è mostrato uno schema dell'array di puntatori **A[]** presente in **Instrument4.cpp**:



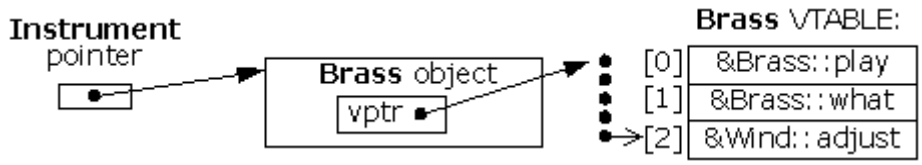
L'array di puntatori a **Instrument** non ha nessuna informazione specifica; ogni puntatore punta ad un oggetto di tipo **Instrument**. **Wind**, **Percussion**, **Stringed** e **Brass** rientrano tutti in questa categoria perché derivati da **Instrument** (quindi hanno la stessa interfaccia di **Instrument** e possono rispondere agli stessi messaggi), quindi i loro indirizzi possono essere memorizzati nell'array. Ciononostante, il compilatore non conosce che loro sono qualcosa in più rispetto a degli oggetti **Instrument**, quindi, normalmente, dovrebbe chiamare la versione della classe base di tutte le funzioni. In questo caso, però, tutte le funzioni sono state dichiarate utilizzando la parola chiave **virtual**, quindi succede qualcosa di diverso.

Ogni volta che viene creata una classe che contiene delle funzioni virtuali, oppure ogni volta che viene derivata una classe da una che contiene funzioni virtuali, il compilatore crea un'unica VTABLE per ogni classe, come mostrato alla destra dello schema precedente. In questa tabella il compilatore inserisce gli indirizzi di tutte le funzioni che sono dichiarate come virtuali in quella classe oppure nella classe base. Se non viene fatto l'override di una

funzione dichiarata virtuale nella classe base, il compilatore usa l'indirizzo della versione della classe base anche nella classe derivata. (Questo comportamento avviene per la funzione **adjust** nella VTABLE di **Brass**). Successivamente, il compilatore inserisce il VPTR (scoperto nell'esempio **Sizes.cpp**) nella classe. Quando viene utilizzata l'ereditarietà semplice, come in questo caso, c'è un solo VPTR per ogni oggetto. Il VPTR deve essere inizializzato in maniera tale da puntare all'indirizzo iniziale della VTALBE relativa a quella classe. (Questo avviene nel costruttore, come si vedrà in dettaglio più avanti).

Una volta inizializzato il VPTR in modo che punti alla VTABLE appropriata, è come se l'oggetto "conoscesse" il proprio tipo. Ma questa auto-coscienza non serve a nulla fino a quando non viene chiamata una funzione virtuale.

Quando viene chiamata una funzione virtuale attraverso l'indirizzo ad un oggetto della classe base (che è la situazione in cui il compilatore non ha tutte le informazioni necessarie per poter fare l'earl binding), accade qualcosa di speciale. Invece di effettuare un tipica chiamata ad una funzione, che è semplicemente un'istruzione **CALL** in linguaggio assembler ad un particolare indirizzo, il compilatore genera del codice differente. Ecco cosa avviene quando viene fatta una chiamata ad **adjust()** per un oggetto **Brass** mediante un puntatore a **Instrument** (un riferimento a **Instrument** produce lo stesso comportamento):



Il compilatore parte dal puntatore a **Instrument**, che punta all'indirizzo iniziale dell'oggetto. Tutti gli oggetti di tipo **Instrument** e quelli che derivano da **Instrument** hanno il loro VPTR nello stesso posto (spesso all'inizio dell'oggetto), quindi il compilatore può prelevare il VPTR dall'oggetto. Il VPTR punta all'indirizzo iniziale di VTABLE. Tutti gli indirizzi delle funzioni in VTABLE sono disposti nello stesso ordine, a prescindere dal tipo particolare di oggetto. L'indirizzo di **play()** è il primo, quello di **what()** è il secondo e quello di **adjust()** è il terzo. Il compilatore conosce che, per qualsiasi oggetto particolare, il puntatore a **adjust()** è alla locazione VPTR+2. Quindi, invece di dire "Chiama la funzione all'indirizzo assoluto **Instrument::adjust**" (early-binding, il comportamento errato in questo caso), viene generato del codice che dice, di fatto, "Chiama la funzione all'indirizzo contenuto nella locazione VPTR+2". Dato che il caricamento di VPTR e la determinazione dell'indirizzo avviene durante l'esecuzione, si ottiene il desiderato late binding. Il messaggio viene mandato all'oggetto e l'oggetto lo elabora.

Sotto il cappello

Può essere d'aiuto prendere in considerazione il codice assembler generato da una chiamata ad una funzione virtuale, per vedere come avviene il meccanismo di late-binding. Di seguito è mostrata l'uscita di un compilatore per la chiamata al metodo

```
i.adjust(1);
```

all'interno della funzione **f(Instrument& i)**:

```
push  1
push  si
mov   bx, word ptr [si]
call  word ptr [bx+4]
add   sp, 4
```

I parametri di una chiamata ad una funzione C++, così come avviene per una chiamata ad una funzione C, vengono inseriti in cima allo stack a partendo da destra verso sinistra (questo tipo di ordinamento è richiesto per poter supportare le liste di argomenti del C), quindi il parametro 1 viene inserito per primo nello stack. A questo punto nella funzione, il registro **si** (un registro presente nei processori con architettura Intel X86) contiene l'indirizzo di **i**. Anche il contenuto di questo registro viene inserito nello stack essendo l'indirizzo di partenza dell'oggetto preso in considerazione. Bisogna ricordare che l'indirizzo iniziale di un oggetto corrisponde al valore di **this** e **this** viene sempre inserito nello stack come se fosse un parametro prima di qualsiasi chiamata ad una funzione membro, in questo modo la funzione membro conosce su quale oggetto particolare sta lavorando. Quindi, nello stack, verrà inserito sempre un parametro in più rispetto al numero di parametri reali prima di fare la chiamata alla funzione (eccezione fatta per le funzioni membro definite come **static**, per le quali **this** non è definito).

A questo punto può essere fatta la chiamata alla funzione virtuale vera e propria. Come prima cosa, bisogna ricavare il valore di VPTR, in modo da poter trovare la VTABLE. Per il compilatore preso in considerazione il VPTR è inserito all'inizio dell'oggetto, quindi **this** punta alla locazione di memoria in cui si trova VPTR. La linea

```
mov bx, word ptr [si]
```

carica la word puntata da si (cioè da this), corrispondente al VPTR. VPTR viene caricato nel registro **bx**.

Il VPTR contenuto in **bx** punta all'indirizzo iniziale di VTABLE, ma il puntatore alla funzione da chiamare non si trova alla locazione zero di VTABLE, bensì nella seconda locazione (essendo la terza funzione nella lista). Per questo modello di memoria, ogni puntatore a funzione è lungo due byte, per questo motivo il compilatore aggiunge quattro al valore di VPTR per calcolare l'indirizzo esatto della funzione da chiamare. Si noti che questo che questo è un valore costante, stabilito durante la compilazione, quindi l'unica cosa che bisogna conoscere è che il puntatore a funzione che si trova nella seconda locazione è quello che punta a **adjust()**. Fortunatamente, il compilatore si prende cura di mettere le cose in ordine, assicurando che tutti i puntatori a funzione in tutte le VTABLE di una particolare gerarchia di classi siano disposti nello stesso ordine, non tenendo conto dell'ordine in cui è stato fatto l'override nelle classi derivate.

Una volta determinato l'indirizzo del puntatore alla funzione opportuna nella VTABLE, la funzione viene chiamata. Quindi l'indirizzo viene caricato e puntato con l'unica istruzione

```
call word ptr [bx+4]
```

Infine, il puntatore allo stack viene mosso in basso per liberare la memoria dai parametri che erano stati inseriti prima della chiamata. Nel codice assembler generato dal C e dal C++ spesso si può avere che la memoria della stack venga deallocata dalla funzione chiamante, ma questo può variare a seconda del processore e dell'implementazione del compilatore.

Installare il vpointer

Dato che il VPTR rende possibile il comportamento virtuale di un oggetto, si può notare come sia importante e critico che il VPTR punti sempre alla VTABLE esatta. Non deve essere possibile chiamare una funzione virtuale prima che il VPTR sia propriamente inizializzato. Solo se un'azione viene messa nel costruttore c'è la garanzia che questa azione venga eseguita, ma in nessuno degli esempi visti fin ora **Instrument** aveva un costruttore.

Questo è il caso in cui la creazione di un costruttore di default è essenziale. Negli esempi visti, il compilatore crea un costruttore di default che non fa altro che inizializzare il VPTR. Questo costruttore, quindi, viene chiamato automaticamente per ognuno degli oggetti **Instrument** prima che si possa fare qualsiasi cosa con essi, quindi si può essere certi che la chiamata ad una funzione virtuale è sempre un'operazione sicura.

Le implicazioni dovute all'inizializzazione automatica del VPTR all'interno del costruttore verranno prese in considerazione in un paragrafo successivo.

Gli oggetti sono differenti

È importante capire che l'upcasting funziona solo con gli indirizzi. Se il compilatore sta processando un oggetto, ne conosce il tipo e quindi (in C++) non utilizzerà il meccanismo del late binding. Per motivi d'efficienza, molti compilatori faranno l'early binding anche per la chiamata ad una funzione virtuale quando possono risalire esattamente al tipo d'oggetto. Ecco un esempio:

```
//: C15:Early.cpp
// Early binding & e funzioni virtuali
#include <iostream>
#include <string>
using namespace std;

class Pet {
public:
  virtual string speak() const { return ""; }
};

class Dog : public Pet {
public:
  string speak() const { return "Bark!"; }
};

int main() {
  Dog ralph;
  Pet* p1 = &ralph;
  Pet& p2 = ralph;
  Pet p3;
  // Late binding per entrambi:
  cout << "p1->speak() = " << p1->speak() <<endl;
  cout << "p2.speak() = " << p2.speak() << endl;
  // Early binding (probabilmente):
  cout << "p3.speak() = " << p3.speak() << endl;
} ///:~
```

In **p1->speak( )** e **p2.speak( )**, vengono utilizzati gli indirizzi, quindi l'informazione a disposizione è incompleta: **p1** e **p2** possono rappresentare l'indirizzo di un **Pet** *oppure* di qualcosa derivato da **Pet**, quindi deve essere utilizzato il meccanismo virtuale. Quando si

chiama **p3.speak()** non c'è ambiguità. Il compilatore conosce il tipo esatto e questo è un oggetto, ma non un oggetto derivato da **Pet**, bensì *esattamente* un **Pet**. Quindi, probabilmente, in questo caso viene usato l'earl binding. Comunque, se il compilatore non vuole fare un lavoro troppo complesso, può comunque utilizzare il late binding è si otterrà lo stesso comportamento finale.

Perché le funzioni virtuali?

A questo punto si potrebbe porre una domanda: "Se questa tecnica è così importante e se permette che ogni volta venga chiamata la funzione 'giusta', perché viene data come opzione? Perché si deve farne esplicitamente riferimento?"

La risposta a questa bella domanda si trova nella filosofia fondamentale del C++: "Perché non è molto efficiente." Si può notare dal codice assembler generato precedentemente, che invece di una semplice CALL ad un indirizzo assoluto, sono necessarie due – più complesse – istruzioni assembler per realizzare la chiamata alla funzione virtuale. Questo richiede spazio in memoria per il codice e tempo di esecuzione.

Alcuni linguaggi orientati ad oggetti hanno adottato l'approccio di utilizzare sempre il meccanismo del late binding, perché viene considerato intrinsecamente legato alla programmazione orientata agli oggetti; non è più un opzione ed il programmatore non deve tenerne conto. Questa è una scelta di progetto quando viene creato un linguaggio e questo approccio è appropriato per molti linguaggi.[56] Il C++, invece, deriva dal C, dove l'aspetto dell'efficienza è critico. Dopotutto, il C fu creato per prendere il posto del linguaggio assembler nella realizzazione di un sistema operativo (con il risultato di rendere questo sistema operativo – Unix – molto più portabile rispetto ai suoi predecessori). Una delle ragioni principali che hanno portato all'invenzione del C++ è stata quella di rendere i programmatori C più efficienti. [57] E la prima domanda che viene posta quando un programmatore C si avvicina al C++ è, "Che impatto avrà sull'occupazione di memoria e sulla velocità d'esecuzione?" Se la risposta fosse, "Migliora tutto tranne per quanto riguarda le chiamate a funzione dove c'è sempre un piccolo aumento delle risorse richieste", molte persone continuerebbero ad utilizzare il C piuttosto che passare al C++. Inoltre, non si potrebbero utilizzare le funzioni inline, perché le funzioni virtuali devono avere un indirizzo da poter mettere nella VTABLE. Per questi motivi la funzione virtuale è un opzione *e* il default del linguaggio è una funzione non virtuale, che rappresenta la configurazione più veloce. Stroustrup disse che la sua idea era, "Se non si utilizza, non si paga."

Quindi, le parola chiave **virtual** viene fornita per migliorare l'efficienza. Quando si progettano le proprie classi, solitamente non ci si vuole preoccupare dell'efficienza. Se si sta utilizzando il polimorfismo, quindi, è bene utilizzare le funzioni virtuali ovunque. Si deve fare attenzione alle funzioni che possono essere realizzate come non-virtuali quando si vuole migliorare la velocità del proprio codice (e solitamente si ottengono risultati migliori lavorando su altri aspetti – un buon profiler potrà trovare i colli di bottiglia del programma in maniera più efficiente rispetto a quanto si potrebbe fare attraverso una semplice stima).

L'evidenze pratiche suggeriscono che il miglioramento in termini di occupazione di memoria e di velocità di esecuzione nel passare al C++ è di circa il 10 percento rispetto al C e spesso le prestazioni sono simili. La ragione per cui si riescono ad ottenere migliori prestazioni consiste nel fatto che un programma C++ può essere progettato e realizzato in

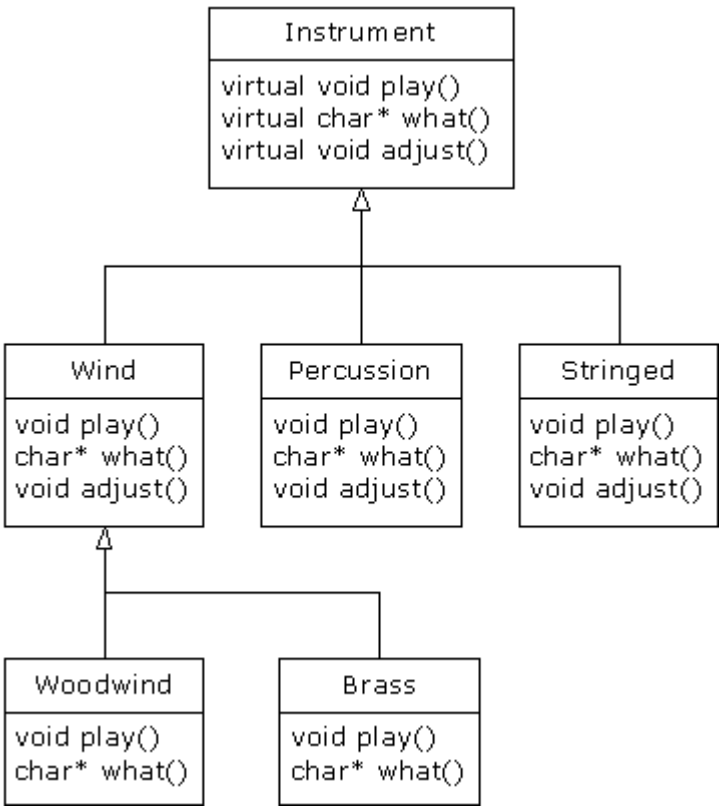
maniera più veloce e occupando meno spazio rispetto a quanto si potrebbe fare utilizzando il C.

Classi base astratte e funzioni virtuali pure

Spesso nella progettazione, si vuole che una classe base implementi solo un'interfaccia per le sue classi derivate. In questo caso,si vuole che nessuno possa creare un oggetto della classe base, ma si vuole rendere possibile solo l'upcast a questa classe base in maniera da poter utilizzarne l'interfaccia. Questo risultato è raggiunto realizzando questa classe come *astratta*, cioè dotandola di almeno una *funzione virtuale pura*. Un funzione virtuale pura può essere riconosciuta perché utilizza la parola chiave **virtual** ed è seguita da **=0**. Se qualcuno prova ad istanziare un oggetto di una classe astratta, il compilatore lo impedirà. Questo meccanismo permette di forzare una particolare idea di progetto.

Quando una classe astratta viene ereditata, tutte le sue funzioni virtuali pure devono essere implementate, altrimenti la classe derivata sarà anch'essa astratta. La creazione di una funzione virtuale pura permette di inserire una funzione membro all'interno di un'interfaccia senza dover realizzare un implementazione della funzione stessa. Allo stesso tempo, una funzione virtuale pura forza le classi derivate a fornirne un'implementazione.

In tutti gli esempi con gli strumenti, le funzioni nella classe base **Instrument** erano funzioni fittizie, "dummy". Se queste funzioni vengono chiamate, qualcosa non ha funzionato. Questo perché l'intento di **Instrument** è quello di creare un'interfaccia comune per tutte le classi che verranno derivate da essa.



L'unica ragione per realizzare l'interfaccia comune è quella di poterla specificare differentemente per ogni sottotipo. Crea la struttura base che determina cosa c'è in comune nelle classi derivate – nient'altro. Quindi **Instrument** è il giusto candidato per

diventare una classe astratta. Si crea una classe astratta quando si vuole manipolare un insieme di classi attraverso un'interfaccia comune, ma quest'ultima non necessita un'implementazione (oppure, un'implementazione completa).

Se si ha un concetto simile a **Instrument** che si comporta come una classe astratta, oggetti appartenenti a questa classe non hanno mai significato. **Instrument** ha l'unico scopo di esprimere l'interfaccia e non un'implementazione particolare, quindi creare un oggetto che sia solo un **Instrument** non ha senso e probabilmente si vuole prevenire che un utente possa farlo. Questo può essere ottenuto realizzando tutte le funzioni virtuali di **Instrument** in maniera tale che stampino un messaggio d'errore; questo sposta la comunicazione di una situazione d'errore a runtime e richiede un test esaustivo del codice da parte dell'utente. È molto meglio risolvere il problema durante la compilazione.

Ecco la sintassi utilizzata per la dichiarazione di una funzione virtuale pura:

```
virtual void f() = 0;
```

Facendo questo, viene comunicato al compilatore di riservare uno spazio nella VTABLE per questa funzione, ma non viene messo alcun indirizzo in questo spazio. Quindi anche se una sola funzione viene dichiarata come virtuale pura, la VTABLE è incompleta.

Se la VTABLE per una classe è incompleta, cosa può fare il compilatore quando qualcuno cerca di creare un'istanza di questa classe? Dato che non può creare in maniera sicura un'istanza di una classe astratta, darà un messaggio d'errore. È il compilatore, quindi, a garantire la purezza di una classe astratta. Rendendo un classe astratta, ci si assicura che nessun programma che utilizzerà questa classe possa crearne un'istanza.

Di seguito è mostrato l'esempio **Instrument4.cpp** modificato in maniera da utilizzare le funzioni virtuali. Siccome tutti i metodi di questa classe sono funzioni virtuali pure, verrà chiamata *classe astratta pura*:

```
//: C15:Instrument5.cpp
// Classi base astratte pure
#include <iostream>
using namespace std;
enum note { middleC, Csharp, Cflat }; // Etc.

class Instrument {
public:
  // Funzioni virtuali pure:
  virtual void play(note) const = 0;
  virtual char* what() const = 0;
  // Si supponga che adjust() modifichi l'oggetto:
  virtual void adjust(int) = 0;
};
// La parte rimanente del file è identica...

class Wind : public Instrument {
public:
  void play(note) const {
    cout << "Wind::play" << endl;
  }
  char* what() const { return "Wind"; }
  void adjust(int) {}
};
```

```
class Percussion : public Instrument {
public:
  void play(note) const {
    cout << "Percussion::play" << endl;
  }
  char* what() const { return "Percussion"; }
  void adjust(int) {}
};

class Stringed : public Instrument {
public:
  void play(note) const {
    cout << "Stringed::play" << endl;
  }
  char* what() const { return "Stringed"; }
  void adjust(int) {}
};

class Brass : public Wind {
public:
  void play(note) const {
    cout << "Brass::play" << endl;
  }
  char* what() const { return "Brass"; }
};

class Woodwind : public Wind {
public:
  void play(note) const {
    cout << "Woodwind::play" << endl;
  }
  char* what() const { return "Woodwind"; }
};

// Identica alla funzione di prima:
void tune(Instrument& i) {
  // ...
  i.play(middleC);
}

// Funzione nuova:
void f(Instrument& i) { i.adjust(1); }

int main() {
  Wind flute;
  Percussion drum;
  Stringed violin;
  Brass flugelhorn;
  Woodwind recorder;
  tune(flute);
  tune(drum);
  tune(violin);
  tune(flugelhorn);
  tune(recorder);
  f(flugelhorn);
} ///:~
```

Le funzioni virtuali pure sono utili perché rendono esplicita l'astrattezza di una classe e comunicano sia all'utente che al compilatore come questa classe dovrebbe essere utilizzata.

Si noti che le funzioni virtuali pure impediscono che una classe astratta venga passata ad una funzione *per valore*. Perciò, rappresentano anche un metodo per prevenire il fenomeno dell'*object slicing* (che sarà descritto poco più avanti). Rendendo una classe astratta, si può essere sicuri che verrà sempre usato un puntatore o un riferimento durante un'operazione di upcasting alla classe astratta.

Il fatto che una funzione virtuale pura non permetta il completamento della VTABLE non significa che non ci possano essere situazioni in cui si vogliano specificare le altre funzioni della classe astratta. Spesso si vuole chiamare la versione della classe base di una funzione, anche se questa è virtuale. È buona abitudine porre il codice comune il più possibile nella radice della gerarchia delle classi. In questo modo non si risparmia solo memoria, ma si ha anche una più semplice propagazione delle modifiche.

Definizione di funzioni virtuali pure

È possibile dare una definizione ad una funzione virtuale pura in una classe base. In questo caso, si sta ancora comunicando al compilatore che non è permesso istanziare oggetti della classe astratta base e che le funzioni virtuali pure dovranno ancora essere definite nelle classi derivate in maniera da poter creare gli oggetti. Nonostante questo, ci potrebbe essere un pezzo di codice comune e si vuole che le classi derivate lo richiamino, piuttosto che duplicare questo codice in ogni funzione.

Ecco come si presenta la definizione di una funzione virtuale pura:

```
//: C15:PureVirtualDefinitions.cpp
// Definizioni di funzioni virtuali pure
#include <iostream>
using namespace std;

class Pet {
public:
  virtual void speak() const = 0;
  virtual void eat() const = 0;
  // Definizioni inline di funzioni virtuali pure non sono permesse:
  //!  virtual void sleep() const = 0 {}
};

// OK, non definita inline
void Pet::eat() const {
  cout << "Pet::eat()" << endl;
}

void Pet::speak() const {
  cout << "Pet::speak()" << endl;
}

class Dog : public Pet {
public:
  // Usa il codice comune di Pet:
  void speak() const { Pet::speak(); }
  void eat() const { Pet::eat(); }
};

int main() {
  Dog simba;  // Il cane di Richard
  simba.speak();
  simba.eat();
```

```
} ///:~
```

Nella VTABLE di **Pet** continua ad esserci uno spazio vuoto, ma c'è una funzione che è possibile richiamare dalla classe derivata.

Un ulteriore vantaggio che si ottiene è la possibilità di cambiare una funzione virtuale ordinaria in una virtuale pura senza dover cambiare il codice esistente. (Questo può essere un modo per verificare quali sono le classi che non fanno l'override delle funzioni virtuali.)

Ereditarietà e VTABLE

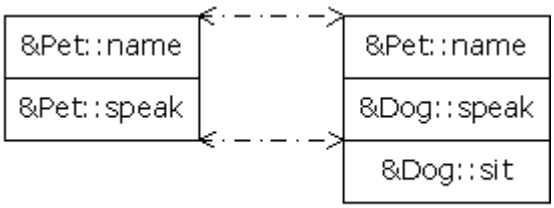
Si è visto cosa succede quando si eredita da una classe e si fa l'override di alcune funzioni virtuali. Il compilatore crea una nuova VTABLE per la nuova classe e qui inserisce gli indirizzi delle nuove funzioni usando gli indirizzi delle funzioni della classe base per tutte le funzioni virtuali per le quali non è stato fatto l'override. In un modo o nell'altro, per ogni oggetto che può essere creato (cioè la cui classe di appartenenza non contenga funzioni virtuali pure) c'è sempre un insieme completo di indirizzi nella VTABLE, quindi non sarà possibile effettuare la chiamata ad un indirizzo diverso da quelli ivi contenuti (la qual cosa potrebbe essere disastrosa).

Ma cosa succede quando si eredità e si aggiunge una nuova funzione virtuale alla classe *derivata*? Ecco un esempio:

```
//: C15:AddingVirtuals.cpp
// Aggiungere funzioni virtuali nelle classi derivate
#include <iostream>
#include <string>
using namespace std;

class Pet {
  string pname;
public:
  Pet(const string& petName) : pname(petName) {}
  virtual string name() const { return pname; }
  virtual string speak() const { return ""; }
};

class Dog : public Pet {
  string name;
public:
  Dog(const string& petName) : Pet(petName) {}
  // Funzione virtuale nuova nella classe Dog:
  virtual string sit() const {
    return Pet::name() + " sits";
  }
  string speak() const { // Override
    return Pet::name() + " says 'Bark!'";
  }
};

int main() {
  Pet* p[] = {new Pet("generic"),new Dog("bob")};
  cout << "p[0]->speak() = "
       << p[0]->speak() << endl;
  cout << "p[1]->speak() = "
       << p[1]->speak() << endl;
//! cout << "p[1]->sit() = "
```

```
//!      << p[1]->sit() << endl; // Non legale
} ///:~
```

La classe **Pet** contiene due funzioni virtuali: **speak( )** e **name( )**. **Dog** aggiunge una terza funzione virtuale chiamata **sit()**, così come effettua l'override di **speak()**. Un diagramma può aiutare a visualizzare cos'è successo. Ecco la VTABLE creata dal compilatore per **Pet** e **Dog**:



Si noti che il compilatore mappa la locazione con l'indirizzo di **speak()** esattamente nello stesso posto sia nella VTABLE di **Dog** che in quella di **Pet**. In maniera simile, se la classe **Pug** viene ereditata da **Dog**, la sua versione di **sit()** viene messa nella sua VTABLE esattamente nello stesso posto che occupa in **Dog**. Questo perché (come è stato visto nell'esempio in linguaggio assembler) il compilatore genera un codice che utilizza un semplice offset numerico all'interno della VTABLE per selezionare la funzione virtuale. Indipendentemente dal particolare sottotipo a cui appartiene l'oggetto, la sua VTABLE è fatta sempre allo stesso modo e le chiamate alle funzioni virtuali vengono fatte sempre in maniera uguale.

In questo caso, comunque, il compilatore lavora solo con il puntatore ad un oggetto appartenente alla classe base. La classe base ha solo le funzioni **speak()** e **name()**, e queste sono gli unici metodi che il compilatore permette di utilizzare. Come potrebbe, infatti, sapere che si sta lavorando con un oggetto di tipo **Dog**, se ha a disposizione solo il puntatore ad un oggetto della classe base? Questo puntatore potrebbe puntare ad un altro tipo di oggetto, che non ha la funzione **sit()**. Ci potrebbero essere o meno gli indirizzi di altre funzioni nella VTABLE, ma in ogni caso, avendo fatto una chiamata virtuale a questa VTABLE non si vuole utilizzare questa possibilità. Per questo il compilatore fa il suo lavoro impedendo che si possano fare chiamate a funzioni virtuali che esistono solo nelle classi derivate.

Ci sono alcuni casi poco comuni in cui si vuole conoscere se il puntatore sta puntando ad un particolare tipo di oggetto. Se si vuole chiamare una funzione che esiste solo in questa sottoclasse, allora si deve fare il cast del puntatore. Si può eliminare l'errore presente nel programma precedente utilizzando una chiamata di questo tipo:

```
((Dog*)p[1])->sit()
```

In questo caso, si deve conoscere che p[1] punta a un oggetto di tipo **Dog**, ma in generale questa informazione non è disponibile. Se il problema che si ha è tale da dover conoscere il tipo esatto di tutti gli oggetti, si dovrebbe ripensare alla soluzione, perché probabilmente non si stanno usando in modo corretto le funzioni virtuali. Esistono, però, delle situazioni in cui il progetto è più efficiente (oppure non si hanno alternative) se si conosce il tipo esatto di tutti gli oggetti contenuti in un generico contenitore. Questo problema va sotto il nome di *identificazione del tipo a run-time* (*run-time type identification* RTTI).

L'RTTI consiste nel fare il cast *verso il basso* o *down-cast* verso i puntatori alla classe derivata dei puntatori alla classe base ("up" e "down" sono relativi al tipico diagramma delle classi, dove la classe base è disegnata in cima). Il cast *verso l'alto* avviene automaticamente, senza nessuna forzatura, perché è un'operazione sicura. Il cast verso il *basso*, invece, non è un'operazione sicura, perché non si ha nessuna informazione a tempo di compilazione circa i tipo effettivo, quindi bisogna conoscere esattamente di che tipo è l'oggetto. Se viene fatto il cast verso un tipo sbagliato, si avranno dei problemi.

L'RTTI viene trattato più avanti in questo capitolo e il Volume 2 di questo libro ha un capitolo dedicato a questo argomento.

Object slicing

Quando si usa il polimorfismo c'è una differenza tra il passare gli indirizzi degli oggetti e passare gli oggetti per valore. Tutti gli esempi presentati, e teoricamente tutti gli esempi che si vedranno, passano gli indirizzi e non i valori. Questo perché gli indirizzi hanno tutti la stessa dimensione[58], quindi passare l'indirizzo di un oggetto appartenente ad un tipo derivato (che tipicamente è un oggetto più grande) è la stessa cosa di passare l'indirizzo di un oggetto appartenente alla classe base (che tipicamente è un oggetto più piccolo). Come si è visto prima questo è quello che si desidera quando si usa il polimorfismo – un codice che operi sui tipi base può operare in maniera trasparente anche su oggetti appartenenti a classi derivate.

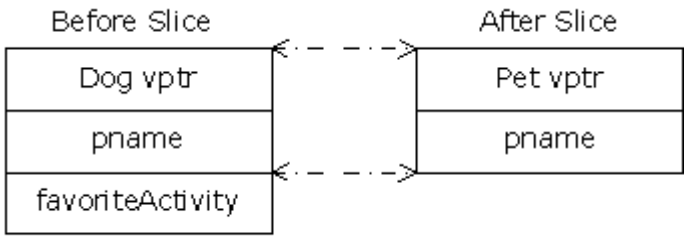
Se si effettua l'upcast ad un oggetto invece che ad un puntatore o ad un riferimento, succede qualcosa che potrebbe sorprendere: l'oggetto viene "affettato" ("sliced") fino a che tutto quello che rimane non è altro che il suboggetto che corrisponde al tipo destinazione dell'operazione di cast. Nell'esempio successivo si può vedere cosa succede quando un oggetto viene affettato:

```
//: C15:ObjectSlicing.cpp
#include <iostream>
#include <string>
using namespace std;

class Pet {
  string pname;
public:
  Pet(const string& name) : pname(name) {}
  virtual string name() const { return pname; }
  virtual string description() const {
    return "This is " + pname;
  }
};

class Dog : public Pet {
  string favoriteActivity;
public:
  Dog(const string& name, const string& activity)
    : Pet(name), favoriteActivity(activity) {}
  string description() const {
    return Pet::name() + " likes to " +
      favoriteActivity;
  }
};

void describe(Pet p) { // "Affetta" l'oggetto
```

```
  cout << p.description() << endl;
}

int main() {
  Pet p("Alfred");
  Dog d("Fluffy", "sleep");
  describe(p);
  describe(d);
} ///:~
```

Alla funzione **describe()** viene passato *per valore* un oggetto del tipo **Pet**. Viene chiamata, poi, la funzione virtuale **description()** per un oggetto di tipo **Pet**. Nel **main()**, ci si aspetterebbe che la prima chiamata produca "This is Alfred", e la seconda produca "Fluffy likes sleep". Di fatto, entrambe le chiamate utilizzano la versione della classe base di **description()**.

In questo programma avvengono due cose. Primo, siccome **describe()** accetta un *oggetto* **Pet** (piuttosto che un puntatore o un riferimento), ogni chiamata a **describe()** provocherà l'inserimento di un oggetto delle dimensioni di **Pet** nello stack e la sua cancellazione dopo la chiamata. Questo significa che se un oggetto di una classe che eredita da **Pet** viene passato a **describe()**, il compilatore darà nessuna segnalazione, ma copierà solo la parte dell'oggetto corrispondente ad un oggetto **Pet**. Il compilatore *taglierà via* la parte derivata dell'oggetto, in questo modo:



è interessante vedere, ora, cosa succede quando si chiama una funzione virtuale. **Dog::description()** fa uso di entrambi **Pet** (che continua ad esistere) e **Dog**, che non esiste più perché è stato tagliato via! Cosa succede quando la funzione virtuale viene chiamata?

Si è al sicuro da eventuali effetti disastrosi perché l'oggetto è stato passato per valore. Per questo motivo, il compilatore conosce il tipo preciso di oggetto visto che l'oggetto derivato è stato trasformato in maniera forzata in un oggetto delle classe base. Quando si utilizza il passaggio di un oggetto per valore, viene utilizzato il costruttore di copia per un oggetto **Pet**, che inizializza il VPTR con l'indirizzo della VTABLE di **Pet** e copia solo le parti corrispondenti all'oggetto **Pet** stesso. Non essendoci un costruttore di copia esplicito, è il compilatore a sintetizzarne uno. Da qualunque punto di vista, a causa dell'object slicing l'oggetto passato per valore diventa un vero è proprio oggetto **Pet**.

Di fatto, l'object slicing rimuove parte dell'oggetto esistente quando viene creato un nuovo oggetto, piuttosto che cambiare il significato di un indirizzo così come avviene quando si usa un puntatore o un riferimento. Per questo motivo, l'upcasting in un altro oggetto non viene utilizzato spesso; di fatto, solitamente, si cerca di tenersi alla larga e di prevenire l'uso dell'upcasting. Si noti che, in questo esempio, se description() fosse stata una funzione virtuale pura nella classe base (che è un'ipotesi ragionevole, visto che di fatto non effettua nessuna operazione nella classe base), allora il compilatore avrebbe impedito

l'object slicing perché non avrebbe permesso la "creazione" di un oggetto della classe base (che è quello che succede quando si effettua l'upcast del valore). Questo potrebbe essere l'utilizzo più importante per le funzioni virtuali pure: prevenire l'object slicing generando un messaggio d'errore durante la compilazione se si cerca di utilizzarlo.

Overloading & overriding

Nel capitolo 14, si è visto che la ridefinizione di una funzione nella classe base attraverso l'overload nasconde tutte le altre versioni della funzione stessa nella classe base. Quando si lavora con funzioni **virtuali**, il comportamento e leggermente diverso. Si consideri una versione modificata dell'esempio **NameHiding.cpp** presentato nel capitolo 14:

```
//: C15:NameHiding2.cpp
// Restrizioni all'overload delle funzioni virtuali
#include <iostream>
#include <string>
using namespace std;

class Base {
public:
  virtual int f() const {
    cout << "Base::f()\n";
    return 1;
  }
  virtual void f(string) const {}
  virtual void g() const {}
};

class Derived1 : public Base {
public:
  void g() const {}
};

class Derived2 : public Base {
public:
  // Override di una funzione virtuale:
  int f() const {
    cout << "Derived2::f()\n";
    return 2;
  }
};

class Derived3 : public Base {
public:
  // non si può cambiare il tipo restituito:
  //! void f() const{ cout << "Derived3::f()\n";}
};

class Derived4 : public Base {
public:
  // Lista degli argomenti cambiata:
  int f(int) const {
    cout << "Derived4::f()\n";
    return 4;
  }
};

int main() {
  string s("hello");
```

```
  Derived1 d1;
  int x = d1.f();
  d1.f(s);
  Derived2 d2;
  x = d2.f();
//!  d2.f(s); // la versione string è nascosta
  Derived4 d4;
  x = d4.f(1);
//!  x = d4.f(); // la versione f() è nascosta
//!  d4.f(s); // la versione string è nascosta
  Base& br = d4; // Upcast
//!  br.f(1); // Versione derivata non disponibile
  br.f(); // Versione base disponibile
  br.f(s); // Versione base disponibile
} ///:~
```

La prima cosa da notare è che in **Derive3**, il compilatore non permette di cambiare il tipo restituito da una funzione ridefinita mediante overidde (sarebbe permesso se **f()** non fosse virtuale). Questa restrizione è importante perché il compilatore deve garantire la possibilità di chiamare "polimorficamente" la funzione attraverso la classe base e se la classe base suppone che **f()** restituisca un **int**, allora la versione di **f()** nella classe derivata deve onorare questa aspettativa altrimenti le cose non funzionano.

La regola esposta nel Capitolo 14 è ancora valida: se si effettua l'override di un metodo della classe base per cui è stato fatto l'overload, le altre versioni del metodo nella classe base non sono più accessibili, vengono nascoste. Nel **main()** il codice che testa **Derived4** mostra che anche se la nuova versione di **f()** non è ottenuta attraverso l'override dell'interfaccia di una funzione virtuale esistente – entrambe le versioni di **f()** della classe base vengono nascoste da **f(int)**. Ciononostante, se si effettua l'upcast di **d4** in **Base**, solo le versioni della classe base sono disponibili (perché questo è quello che viene assicurato dalla classe base) mentre quella della classe derivata non lo è più (perché non è specificata nella classe base).

Cambiare il tipo restituito

La classe **Derived3** di prima mostra come non si possa modificare il tipo restituito da una funzione virtuale con l'override. Questo vale in generale, ma c'è un caso speciale in cui si può modificare leggermente il tipo restituito. Se si sta restituendo un puntatore oppure un riferimento ad una classe base, allora la versione della funzione ottenuta tramite override può restituire un puntatore oppure in riferimento ad una classe derivata da quella restituita dalla funzione nella classe base. Per esempio:

```
//: C15:VariantReturn.cpp
// Restituire un puntatore o un riferimento ad un tipo
// derivato durante l'override
#include <iostream>
#include <string>
using namespace std;

class PetFood {
public:
  virtual string foodType() const = 0;
};

class Pet {
public:
  virtual string type() const = 0;
```

```
  virtual PetFood* eats() = 0;
};

class Bird : public Pet {
public:
  string type() const { return "Bird"; }
  class BirdFood : public PetFood {
  public:
    string foodType() const {
      return "Bird food";
    }
  };
  // Upcast al tipo base:
  PetFood* eats() { return &bf; }
private:
  BirdFood bf;
};

class Cat : public Pet {
public:
  string type() const { return "Cat"; }
  class CatFood : public PetFood {
  public:
    string foodType() const { return "Birds"; }
  };
  // Restituisce il tipo esatto:
  CatFood* eats() { return &cf; }
private:
  CatFood cf;
};

int main() {
  Bird b;
  Cat c;
  Pet* p[] = { &b, &c, };
  for(int i = 0; i < sizeof p / sizeof *p; i++)
    cout << p[i]->type() << " eats "
         << p[i]->eats()->foodType() << endl;
  // Può restituire il tipo esatto:
  Cat::CatFood* cf = c.eats();
  Bird::BirdFood* bf;
  // Non può restituire il tipo esatto:
//!  bf = b.eats();
  // Bisogna effettuare il downcast:
  bf = dynamic_cast<Bird::BirdFood*>(b.eats());
} ///:~
```

La funzione membro **Pet::eats()** restituisce un puntatore a un **PetFood**. In **Bird**, questa funzione membro viene derivata esattamente dalla classe base tramite overload, incluso il tipo restituito. Si ha che **Bird::eats()** effettua l'upcast di **BirdFood** in **PetFood**.

In **Cat**, invece, il tipo restituito da **eats()** è un puntatore a **CatFood**,un tipo derivato da **PetFood**. L'unica ragione per cui questa classe viene compilata è che il tipo restituito viene ereditato dal tipo restituito dalla funzione nella classe base. In questo modo il comportamento atteso viene rispettato; **eats()** restituisce sempre un puntatore ad un **PetFood**.

Se si ragiona in maniera "polimorfica", un comportamento di questo tipo non sembra necessario. Perché non effettuare l'upcast a **PetFoods*** dei tipi restituiti, così come viene fatto da **Bird::eats()**? Tipicamente questa è una buona soluzione, alla fine del main()

viene mostrata la differenza: **Cat::eats()** può restituire il tipo esatto di **PetFood**, mentre del valore restituito da **Bird::eats()** bisogna effettuare il downcast verso il tipo desiderato.

Quindi riuscire a restituire il tipo esatto garantisce maggiore generalità e non causa la perdita di informazioni sul tipo dovute all'upcast automatico. Spesso restituendo il tipo base generalmente si riescono a risolvere i proprio problemi, quindi questa caratteristica viene sfruttata solo in casi particolari.

funzioni virtuali & costruttori

Quando viene creato un oggetto contenente delle funzioni virtuali, il suo VPTR deve essere inizializzato in maniera tale da puntare alla VTABLE appropriata. Questo deve essere fatto prima che venga data la possibilità di chiamare una qualsiasi funzione virtuale. Come si può intuire, siccome il costruttore ha il compito di creare un oggetto, si occuperà anche di inizializzare il VPTR. Il compilatore inserisce segretamente all'inizio del costruttore del codice per l'inizializzazione del VPTR. Come descritto nel Capitolo 14, se non viene creato esplicitamente un costruttore per una classe, sarà il compilatore a sintetizzarne uno. Se la classe possiede delle funzioni virtuali, il costruttore sintetizzato comprenderà il codice appropriato per l'inizializzazione del VPTR. Tenendo conto di questo comportamento è possibile fare alcune considerazioni.

La prima riguarda l'efficienza. Il motivo per cui vengono utilizzate le funzioni **inline** è quello di ridurre l'overhead dovuto alle chiamate delle funzioni piccole. Se il C++ non avesse fornito le funzioni **inline**, si sarebbe potuto utilizzare il preprocessore per creare queste "macro". Il preprocessore, però, non ha il concetto di accesso o di classe, per cui non potrebbe essere utilizzato per creare funzioni membro come macro. Inoltre, con i costruttori che hanno codice nascosto inserito dal compilatore, una macro per il preprocessore non funzionerebbe.

Quando si è alla caccia di falle nell'efficienza dei programmi, bisogna tenere presente che il compilatore sta inserendo codice nascosto nel costruttore. Non solo inizializza il VPTR, ma controlla il valore di **this** (nel caso l'**operatore new** restituisca zero) e chiama i costruttori delle classi base. Preso insieme, questo codice può influire su quella che si pensava essere una chiamata ad una piccola funzione inline. In particolare, la dimensione del costruttore potrebbe rendere nulli tutti gli sforzi fatti per ridurre l'overhead delle chiamate a funzione. Se vengono fatte molte chiamate ai costruttori inline, le dimensioni del codice possono crescere senza apportare alcun beneficio alla velocità di esecuzione.

Sicuramente non è preferibile implementare tutti i piccoli costruttori come non-inline, perché è più semplice scriverli come inline. Quando si stanno mettendo a punto le prestazioni del codice, però, bisogna ricordarsi di eliminare i costruttori inline.

Ordine delle chiamate ai costruttori

Il secondo aspetto interessante dei costruttori e delle funzioni virtuali riguarda l'ordine in cui i costruttori vengono chiamati e il modo in cui le chiamate virtuali vengono fatte all'interno dei costruttori.

Tutti i costruttori delle classi base vengono sempre chiamati dal costruttore di una classe ereditata. Questo comportamento ha senso perché il costruttore ha un compito speciale:

fare in modo che l'oggetto venga costruito in maniera corretta. Una classe derivata può inizializzare correttamente i propri elementi. Per questo motivo è essenziale che tutti i costruttori vengano chiamati; altrimenti l'oggetto nel suo insieme non verrebbe costruito correttamente. Questo è il motivo per cui il compilatore forza una chiamata al costruttore per ogni parte della classe derivata. Se nella lista dei costruttori non viene specificato esplicitamente quale costruttore della classe base utilizzare, il compilatore chiamerà il costruttore di default. Se non c'è alcun costruttore di default, il compilatore darà errore.

L'ordine delle chiamate ai costruttori è importante. Quando si eredita, si conosce tutto della classe base e si può accedere a qualunque suo membro **public** o **protected**. Questo significa che si dovrebbe poter ritenere validi tutti i membri della classe base quando si è nella classe derivata. In una funzione membro normale, la costruzione dell'oggetto è già stata effettuata, quindi tutti i membri di tutte le parti dell'oggetto sono stati costruiti. All'interno del costruttore, invece, si dovrebbe essere in grado di ritenere che tutti i membri che vengono utilizzati sono stati costruiti. L'unico modo per garantire questo è di chiamare per primo il costruttore della classe base. In questo modo quando ci si trova nel costruttore della classe derivata, tutti i membri della classe base a cui si può accedere sono stati inizializzati. "Poter ritenere tutti i membri validi" all'interno del costruttore è la ragione per cui, quando è possibile, bisognerebbe inizializzare tutti gli oggetti membro (vale a dire tutti gli oggetti contenuti nella classe) utilizzando la lista dei costruttori. Seguendo questa abitudine, si può ritenere che tutti i membri della classe base e tutti gli oggetti membro del oggetto corrente sono stati inizializzati.

Comportamento delle funzioni virtuali all'interno dei costruttori

L'ordine con cui vengono chiamati i costruttori porta con se un dilemma interessante. Cosa succede se all'interno di un costruttore si chiama una funzione virtuale? All'interno di una funzione membro ordinaria si può immaginare cosa succede – la chiamata virtuale viene risolta a runtime perché l'oggetto non sa se appartiene alla classe in cui la funzione membro si trova, oppure a qualche classe derivata da questa. Per consistenza, si è portati a pensare lo stesso avvenga anche all'interno dei costruttori.

Invece, non è questo quello che accade. Se si chiama una funzione virtuale all'interno di un costruttore, solo la versione locale della funzione viene utilizzata. Questo perché il meccanismo virtuale non funziona all'interno del costruttore.

Ci sono due motivi per cui questo comportamento ha senso. Concettualmente, il compito del costruttore è quello di creare un oggetto in modo tale che esso esista (che non è un'operazione molto ordinaria da fare). All'interno di ogni costruttore, l'oggetto può essere costruito solo parzialmente - si è sicuri che solo gli oggetti appartenenti alla classe base sono stati inizializzati, ma non si conoscono quali sono le classi ereditate da quella attuale. Una chiamata ad una funzione virtuale, invece, percorre "verso il basso" o "verso l'alto" la gerarchia delle eredità. Chiama una funzione in una classe derivata. Se si potesse fare questo anche all'interno di un costruttore, si potrebbe chiamare una funzione che opera su membri che non sono stati ancora inizializzati, una ricetta sicura per un disastro.

Il secondo motivo è di tipo meccanico. Quando un costruttore viene chiamato, una delle prime cose che fa è l'inizializzazione del proprio VPTR. Inoltre, può solo conoscere che lui è del tipo "corrente" – cioè del tipo per cui il costruttore è stato scritto. Il codice del costruttore ignora completamente se questo oggetto è o meno la base di un'altra classe.

Quando il compilatore genera il codice per questo costruttore, genera il codice per il costruttore di questa classe, non di una classe base o di una classe derivata da questa (perché una classe non conosce chi erediterà da lei). Quindi il VPTR che userà sarà per la VTABLE di *questa classe*. Il VPTR rimane inizializzato con il valore di questa VTABLE per il resto della vita dell'oggetto *solo se* questa è l'ultima chiamata ad un costruttore. Se dopo viene chiamato un costruttore derivato, questo setterà il VPTR con l'indirizzo della sua VTABLE, e così via, fino all'ultimo costruttore chiamato. Lo stato del VPTR è determinato dal costruttore che viene chiamato per ultimo. Questo è un altro motivo per cui i costruttori vengono chiamati in ordine da quello base a l'ultimo derivato.

Ma mentre tutta questa serie di chiamate a costruttore ha luogo, ogni costruttore ha settato il VPTR all'indirizzo della propria VTABLE. Se si usa il meccanismo virtuale per le chiamate a funzione, verrà effettuata la chiamata attraverso la propria VTABLE, non con la VTABLE dell'ultimo oggetto derivato (come succederebbe dopo le chiamate a *tutti* i costruttori). In più, molti compilatori riconoscono che una chiamata a funzione virtuale viene fatta da un costruttore ed effettuano l'early binding perché sanno che il late –binding produrrà una chiamata alla funzione locale. In entrambi i casi, non si otterranno i risultati aspettati effettuando una chiamata a funzione virtuale all'interno di un costruttore.

Distruttori e distruttori virtuali

Non si può utilizzare le parola chiave **virtual** con i costruttori, ma i distruttori possono, e spesso devono, essere virtuali.

Il costruttore ha lo speciale compito di mettere assieme un oggetto pezzo per pezzo, chiamando per primo il costruttore base e dopo i costruttori derivati nell'ordine in cui sono ereditati (durante questa sequenza di operazioni può anche chiamare i costruttori di oggetti membri). In modo simile, il distruttore ha un compito speciale: deve smontare un oggetto che potrebbe appartenere ad una gerarchia di classi. Per far questo, il compilatore genera del codice che chiama tutti i distruttori, ma in ordine *inverso* rispetto a quanto viene fatto per i costruttori. Quindi, il distruttore parte dalla classe più derivata e lavora all'indietro fino alla classe base. Questo è un comportamento sicuro e desiderabile perché il distruttore corrente può sempre ritenere che i membri della classe base sono vivi e attivi. Se è necessario chiamare una funzione membro della classe base all'interno del distruttore, questa è un'operazione sicura. Quindi, il distruttore, può effettuare la propria pulizia, poi chiama il distruttore successivo nella gerarchia, che effettuerà la propria pulizia, ecc. Ogni distruttore conosce che la propria classe è derivata *da altre classi* , ma non quelle che sono derivate dalla propria classe.

Si dovrebbe ricordare che i costruttori e i distruttori sono gli unici posti dove questa chiamata gerarchica deve essere fatta (cioè la gerarchia appropriata viene generata automaticamente dal compilatore). In tutti le altre funzioni, solo *quella* funzione viene chiamata (e non le versioni della classe base), che sia virtuale o meno. L'unico modo affinché le versioni della classe base della stessa funzione vengano chiamate nelle funzioni ordinarie (virtuali o meno) è quello di chiamare *esplicitamente* la funzione.

Normalmente, l'azione del distruttore è adeguata. Ma cosa succede se si vuole utilizzare un oggetto attraverso un puntatore alla sua classe base (cioè, si vuole utilizzare l'oggetto attraverso la sua interfaccia generica)? Questo tipo di operazione è uno degli obiettivi principali della programmazione orientata agli oggetti. Il problema si incontra quando si vuole fare il **delete** del puntatore di questo tipo per un oggetto che è stato creato nell'heap

con un **new**. Se è un puntatore alla classe base, il compilatore sa solo che deve chiamare la versione del distruttore della classe base durante un **delete**. Ricorda niente? Questo è lo stesso problema per risolvere il quale sono state create le funzioni virtuali. Fortunatamente, le funzioni virtuali funzionano per i distruttori come per tutte le altre funzioni tranne i costruttori.

```
//: C15:VirtualDestructors.cpp
// Differenze di comportamento tra distruttori virtuali e non
// virtuali
#include <iostream>
using namespace std;

class Base1 {
public:
  ~Base1() { cout << "~Base1()\n"; }
};

class Derived1 : public Base1 {
public:
  ~Derived1() { cout << "~Derived1()\n"; }
};

class Base2 {
public:
  virtual ~Base2() { cout << "~Base2()\n"; }
};

class Derived2 : public Base2 {
public:
  ~Derived2() { cout << "~Derived2()\n"; }
};

int main() {
  Base1* bp = new Derived1; // Upcast
  delete bp;
  Base2* b2p = new Derived2; // Upcast
  delete b2p;
} ///:~
```

Quando si lancia il programma, si vede che **delete bp** chiama solo il distruttore della classe base, mentre **delete b2p** chiama il distruttore della classe derivate seguito dal distruttore della classe base, che è il comportamento desiderato. Dimenticare di dichiarare il distruttore **virtual** è un errore insidioso perché spesso non influisce direttamente sul comportamento del proprio programma, ma può tranquillamente introdurre una perdita di memoria. Inoltre, il fatto che *qualche* distruzione avviene può nascondere per molto tempo il problema.

Anche se il distruttore, come il costruttore, è una funzione “che fa eccezione”, è possibile per il distruttore essere virtuale perché l’oggetto conosce già di che tipo è (così non é durante la costruzione). Una volta che un oggetto è stato costruito, il suo VPTR è inizializzato, quindi si possono fare chiamate virtuali a funzione.

Distruttori virtuali puri

Nonostante i distruttori virtuali puri siano legali nel C++ Standard, c’è un ulteriore obbligo quando vengono utilizzati: bisogna dotare di un corpo i distruttori virtuali puri. Questo può sembrare non intuitivo; come può un funzione essere “pura” se ha bisogno di un

corpo? Se si ricorda, però, che i costruttori e i distruttori rappresentato operazioni speciali, questo obbligo acquista maggior significato, specialmente se si ricorda che tutti i distruttori vengono sempre chiamati in una gerarchia di classi. Se si potesse lasciare vuota la definizione di un distruttore virtuale puro, quale corpo di funzione verrebbe chiamato durante la distruzione? Quindi, è assolutamente necessario che il compilatore ed il linker forzino l'esistenza di un corpo per un distruttore virtuale puro.

Se è puro, ma ha il corpo, qual è il suo significato? L'unica differenza che si può vedere tra un distruttore virtuale puro e non-puro è che il primo costringe la classe ad essere astratta, quindi non si può creare un'istanza della classe base (questo accadrebbe anche se una qualsiasi altra funzione nella classe base fosse virtuale pura).

Le cose sono un poco più confuse, inoltre, quando si eredita una classe da una che contiene un distruttore virtuale puro. Differentemente da qualsiasi altra funzione virtuale pura, *non* è obbligatorio fornire una definizione di un distruttore virtuale puro nella classe derivata. Il fatto che il listato seguente venga compilato e linkato ne è una prova:

```
//: C15:UnAbstract.cpp
// Distruttori virtuali puri
// sembra comportarsi in modo strano

class AbstractBase {
public:
  virtual ~AbstractBase() = 0;
};

AbstractBase::~AbstractBase() {}

class Derived : public AbstractBase {};
// L'ovveride del distruttore non è necessario?

int main() { Derived d; } ///:~
```

Normalmente, un funzione virtuale pura in una classe base dovrebbe comportare che anche la classe derivata sia astratta se non ne venga data una definizione (e anche delle altre funzioni virtuali pure). Ma in questo caso, sembra che questo non sia vero. Bisogna ricordare che il compilatore *automaticamente* crea una definizione del distruttore per ogni classe se questo non viene fatto esplicitamente. Questo è quello che succede in questo caso – viene fatto l'override del distruttore della classe base e questa definizione viene utilizzata dal compilatore in maniera tale che **Derived** non sia più astratta.

Quanto visto porta a formulare una domanda interessante: Qual'è la caratteristica di un distruttore virtuale puro? Non è quella di una normale funzione virtuale pura, per cui bisogna fornire un corpo. In una classe derivata, non si è obbligati a fornire una definizione perché il compilatore sintetizza il distruttore per noi. Quindi qual è la differenza tra un distruttore virtuale normale e uno virtuale puro?

L'unica differenza si ha quando si ha una classe che possiede una sola funzione virtuale pura: il distruttore. In questo caso, l'unico effetto della purezza del distruttore è quello di prevenire che la classe base venga istanziata. Se ci fossero altre funzioni virtuali pure, queste provvederebbero ad evitare che la classe base venga istanziata, ma se no ce ne sono altre, allora questo verrà fatto dal distruttore virtuale puro. Quindi, mentre l'aggiunta di un distruttore virtuale è essenziale, che sia puro o no non è così importante.

Quando si esegue l'esempio seguente, si può vedere che il corpo della funzione virtuale pura viene chiamato dopo la versione della classe derivata, come avviene per ogni altro distruttore:

```
//: C15:PureVirtualDestructors.cpp
// Distruttori virtuali puri
// richiede un body per la funzione
#include <iostream>
using namespace std;

class Pet {
public:
  virtual ~Pet() = 0;
};

Pet::~Pet() {
  cout << "~Pet()" << endl;
}

class Dog : public Pet {
public:
  ~Dog() {
    cout << "~Dog()" << endl;
  }
};

int main() {
  Pet* p = new Dog; // Upcast
  delete p; // Chiamata al distruttore virtuale
} ///:~
```

Come linea guida, ogni volta che si ha una funzione virtuale in una classe, si dovrebbe immediatamente aggiungere un distruttore virtuale (anche se non fa nulla). In questo modo, ci si assicura contro ogni sorpresa futura.

Chiamate virtuali all'interno dei distruttori

C'è qualcosa che succede durante la distruzione che non ci si aspetta. Se ci si trova all'interno di una funzione membro ordinaria e si chiama una funzione virtuale, questa funzione viene chiamata utilizzando il meccanismo del late-binding. Questo non è vero con i distruttori, virtuali o non virtuali. All'interno di un distruttore, solo la versione "locale" di una funzione membro viene chiamata; il meccanismo virtuale viene ignorato.

```
//: C15:VirtualsInDestructors.cpp
// Chiamate virtuali all'interno dei distruttori
#include <iostream>
using namespace std;

class Base {
public:
  virtual ~Base() {
    cout << "Base1()\n";
    f();
  }
  virtual void f() { cout << "Base::f()\n"; }
};

class Derived : public Base {
public:
```

```
  ~Derived() { cout << "~Derived()\n"; }
  void f() { cout << "Derived::f()\n"; }
};

int main() {
  Base* bp = new Derived; // Upcast
  delete bp;
} ///:~
```

Durante la chiamata al distruttore, **Derived::f()** *non viene chiamata*, anche se **f()** è virtuale.

Che cosa succede? Supponiamo che il meccanismo virtuale *venga* utilizzato all'interno del distruttore. Allora sarebbe possibile per una chiamata virtuale arrivare ad una funzione "più esterna" (appartenente ad una classe derivata) nella gerarchia delle classi rispetto al distruttore corrente. Ma i costruttori vengono chiamati dal "più esterno al più interno" (dal distruttore dell'ultima classe derivata fino al distruttore della classe base), quindi la chiamata a funzione considerata dovrebbe dipendere da una parte di un oggetto che è stato *già distrutto*! Questo non succede, perché il compilatore risolve la chiamata a tempo di compilazione e chiama solo la versione "locale" della funzione. Si noti che lo stesso vale per il costruttore (come è stato descritto precedentemente), ma nel caso del costruttore l'informazione sul tipo non era disponibile, mentre nel distruttore l'informazione (vale a dire il VPTR) c'è, ma non è affidabile.

Creare una gerarchia object-based

Un argomento ricorrente in questo libro durante l'illustrazione delle classi **Stack** e **Stash** è stato "il problema dell'appartenenza". Il "padrone" fa riferimento a chi o cosa è responsabile della chiamata a **delete** per oggetti che sono stati creati dinamicamente (utilizzando **new**). Il problema quando si usano contenitori è che questi devono essere abbastanza flessibili da poter contenere diversi tipi di oggetti. Per fare questo, i contenitori posseggono dei puntatori a **void** e quindi non conoscono il tipo di oggetto che dovranno contenere. Quando si cancella un puntatore a **void** non si chiama il distruttore, quindi il contenitore non può essere responsabile della pulizia degli oggetti che contiene.

Una soluzione è stata presentata nell'esempio **C14:InheritStack.cpp**, dove **Stack** veniva ereditato da una nuova classe che accettava e produceva solo puntatori a **string**. Dato che sapeva di poter contenere solo puntatori ad oggetti **string**, poteva cancellarli in maniera propria. Questa era una bella soluzione, ma bisognava ereditare una nuova classe contenitore per ogni tipo che si voleva inserire nel contenitore. (Anche se ora questo sembra tedioso, funzionerà abbastanza bene nel Capitolo 16, quando verranno introdotti i template).

Il problema è che si vuole che il contenitore contenga più di un solo tipo e non si vogliono usare puntatori a **void**. Un'altra soluzione è quella di usare il polimorfismo forzando tutti gli oggetti che si metteranno nel contenitore ad essere ereditati dalla stessa classe base. In questo modo, il contenitore conterrà oggetti della classe base e si potranno chiamare funzioni virtuali – in particolare, si potranno chiamare distruttori virtuali per risolvere il problema dell'appartenenza.

Questa soluzione usa quella che è conosciuta come *gerarchia singly-root* o *gerarchia object –based* (perché la classe root della gerarchia viene chiamata solitamente "Object"). Ci sono molti altri vantaggi nell'utilizzare una gerarchia singly-root; infatti, tutti gli altri

linguaggi object-oriented oltre al C++ forzano l'uso di questa gerarchia – quando si crea una classe, automaticamente questa viene ereditata direttamente o indirettamente da una classe base comune, una classe che è stata stabilita dai creatori del linguaggio. Nel C++, si è pensato che l'uso forzato di una classe base comune avrebbe causato troppo overhead, quindi non è stato fatto. Comunque, si può decidere di utilizzare una classe base comune nei propri progetti e questo argomento verrà esaminato a fondo nel Volume 2 di questo libro.

Per risolvere il problema dell'appartenenza, si può creare un classe **Object** estremamente semplice come classe base, che contiene solo un distruttore virtuale. **Stack** può, allora, contenere classi ereditate da **Object**:

```
//: C15:OStack.h
// Uso di una gerarchia singly-rooted
#ifndef OSTACK_H
#define OSTACK_H

class Object {
public:
  virtual ~Object() = 0;
};

// Definizione richiesta:
inline Object::~Object() {}

class Stack {
  struct Link {
    Object* data;
    Link* next;
    Link(Object* dat, Link* nxt) :
      data(dat), next(nxt) {}
  }* head;
public:
  Stack() : head(0) {}
  ~Stack(){
    while(head)
      delete pop();
  }
  void push(Object* dat) {
    head = new Link(dat, head);
  }
  Object* peek() const {
    return head ? head->data : 0;
  }
  Object* pop() {
    if(head == 0) return 0;
    Object* result = head->data;
    Link* oldHead = head;
    head = head->next;
    delete oldHead;
    return result;
  }
};
#endif // OSTACK_H ///:~
```

Per semplificare le cose e tenere tutto nel file di header, la definizione (richiesta) per il distruttore virtuale puro viene fatta inline nel file di header stesso e anche **pop()** (che potrebbe essere considerata troppo grande per essere inline) viene definita come inline.

Gli oggetti **Link** ora contengono dei puntatori a **Object** invece che a **void** e **Stack** accetterà e restituirà solo puntatori a **Object**. Ora **Stack** è più flessibile, potrà contenere molti tipi diversi ma distruggerà anche ogni oggetto che verrà lasciato in **Stack**. Il nuovo limite (che verrà rimosso definitivamente quando verranno utilizzati i template per risolvere il problema nel Capitolo 16) è che tutto ciò che viene messo nello **Stack** deve essere ereditato da **Object**. Questo può andare bene se si sta realizzando la classe da inserire partendo dal progetto, ma cosa succede se la classe da mettere in **Stack** esiste già, come **string**? In questo caso, la nuova classe dove essere sia un **string** che un **Object**, ciò significa che deve essere ereditata da entrambe le classi. Questa operazione viene chiamata *ereditarietà multipla* ed è l'argomento di un intero capitolo nel Volume 2 di questo libro (scaricabile all'indirizzo *www.BruceEckel.com*). Quando il lettore leggerà questo capitolo, vedrà che l'ereditarietà multipla è legata alla complessità ed è una caratteristica da utilizzare limitatamente. In questa situazione, comunque, tutto è abbastanza semplice che non ci si vuole inoltrare in nessun vicolo buio legato all'ereditarietà multipla:

```
//: C15:OStackTest.cpp
//{T} OStackTest.cpp
#include "OStack.h"
#include "../require.h"
#include <fstream>
#include <iostream>
#include <string>
using namespace std;

// Usa l'ereditarietà multipla. Vogliamo
// entrambi string e Object:
class MyString: public string, public Object {
public:
  ~MyString() {
    cout << "deleting string: " << *this << endl;
  }
  MyString(string s) : string(s) {}
};

int main(int argc, char* argv[]) {
  requireArgs(argc, 1); // Il file name è un argomento
  ifstream in(argv[1]);
  assure(in, argv[1]);
  Stack textlines;
  string line;
  // Legge il file e memorizza le righe nello stack:
  while(getline(in, line))
    textlines.push(new MyString(line));
  // Estrae alcune righe dallo stack:
  MyString* s;
  for(int i = 0; i < 10; i++) {
    if((s=(MyString*)textlines.pop())==0) break;
    cout << *s << endl;
    delete s;
  }
  cout << "Letting the destructor do the rest:"
    << endl;
} ///:~
```

Anche se questa versione del programma di test per **Stack** è molto simile alla precedente, si noterà che solo 10 elementi vengono estratti dallo stack, il che significa che probabilmente degli oggetti rimangono dentro. Dato che **Stack** sa di essere un contenitore di **Object**, il distruttore può effettuare una cancellazione corretta, e si vedrà che questa è

l'uscita del programma, dato che gli oggetti **MyString** stampano un messaggio quando vengono distrutti.

Creare contenitori che contengano **Object** è un approccio ragionevole – *se* si ha una gerarchia singly-root (forzata dal linguaggio oppure dalla specifica che ogni classe erediti da **Object**). In questo caso, è garantito che tutto è un Object e quindi non è così difficile utilizzare i contenitori. In C++, comunque, non ci si può aspettare questo per ogni classe, quindi sarà d'obbligo arrivare all'ereditarietà multipla se si utilizza questo approccio. Si vedrà nel Capitolo 16 che i template risolvono il problema in modo più semplice è più elegante.

Overload degli operatori

Si possono avere operatori **virtual** così come si fa con le altre funzioni membro. Implementare operatori **virtual** spesso porta confusione, perché si dovrebbe operare con due oggetti, entrambi di tipo non noto. Solitamente questa è la situazione con componenti matematici (per i quali spesso si vuole l'overload degli operatori). Per esempio, si consideri un sistema che lavora con matrici, vettori e valori scalari, tutti e tre derivati dalla classe **Math**:

```
//: C15:OperatorPolymorphism.cpp
// Polimorfismo con overload di operatori
#include <iostream>
using namespace std;

class Matrix;
class Scalar;
class Vector;

class Math {
public:
  virtual Math& operator*(Math& rv) = 0;
  virtual Math& multiply(Matrix*) = 0;
  virtual Math& multiply(Scalar*) = 0;
  virtual Math& multiply(Vector*) = 0;
  virtual ~Math() {}
};

class Matrix : public Math {
public:
  Math& operator*(Math& rv) {
    return rv.multiply(this); // 2nd dispatch
  }
  Math& multiply(Matrix*) {
    cout << "Matrix * Matrix" << endl;
    return *this;
  }
  Math& multiply(Scalar*) {
    cout << "Scalar * Matrix" << endl;
    return *this;
  }
  Math& multiply(Vector*) {
    cout << "Vector * Matrix" << endl;
    return *this;
  }
};

class Scalar : public Math  {
```

```
public:
  Math& operator*(Math& rv) {
    return rv.multiply(this); // 2nd dispatch
  }
  Math& multiply(Matrix*) {
    cout << "Matrix * Scalar" << endl;
    return *this;
  }
  Math& multiply(Scalar*) {
    cout << "Scalar * Scalar" << endl;
    return *this;
  }
  Math& multiply(Vector*) {
    cout << "Vector * Scalar" << endl;
    return *this;
  }
};

class Vector : public Math  {
public:
  Math& operator*(Math& rv) {
    return rv.multiply(this); // 2nd dispatch
  }
  Math& multiply(Matrix*) {
    cout << "Matrix * Vector" << endl;
    return *this;
  }
  Math& multiply(Scalar*) {
    cout << "Scalar * Vector" << endl;
    return *this;
  }
  Math& multiply(Vector*) {
    cout << "Vector * Vector" << endl;
    return *this;
  }
};

int main() {
  Matrix m; Vector v; Scalar s;
  Math* math[] = { &m, &v, &s };
  for(int i = 0; i < 3; i++)
    for(int j = 0; j < 3; j++) {
      Math& m1 = *math[i];
      Math& m2 = *math[j];
      m1 * m2;
    }
} ///:~
```

Per semplicità, è stato fatto l'overload solo dell'operatore **operator***. L'obiettivo è quello di riuscire a moltiplicare qualsiasi coppia di oggetti **Math** e produrre il risultato desiderato – e si noti che moltiplicare una matrice per un vettore è un'operazione molto diversa dal moltiplicare un vettore per una matrice.

Il problema è che, nel **main()**, l'espressione **m1*m2** contiene due riferimenti a **Math** ottenuti mediante upcast, e quindi due oggetti di tipo sconosciuto. Una funzione virtuale è capace solo di effettuare un singolo compito – cioè determinare il tipo di uno degli oggetti sconosciuti. Per determinare entrambi i tipi la tecnica chiamata *multiple dispatching* è utilizzata in questo esempio, dove quella che sembra la chiamata ad un'unica funzione virtuale diviene la chiamata ad una seconda funzione virtuale. Quando la seconda chiamata viene fatta, vengono determinati i tipi di entrambi gli oggetti e si può fare l'operazione

appropriata. Non risulta subito chiaro, ma se si ferma un po' l'attenzione sull'esempio si dovrebbe iniziare a capirne il senso. Questo argomento viene trattato in maniera più approfondita nel capitolo Progettazione di Pattern del Volume 2, che può essere scaricato all'indirizzo *www.BruceEckel.com*.

Downcast

Come si può intuire, così come esiste una cosa chiamata upcast – che si muove verso l'alto in una gerarchia di classi – ci dovrebbe essere anche il downcast per muoversi verso il basso nella gerarchia. Ma l'upcast è semplice perché quando ci si muove verso l'alto in una gerarchia le classi convergono sempre a classi più generali. Quando si effettua l'upcast è sempre chiaro da quali genitori si è derivati (tipicamente uno, tranne nel caso dell'ereditarietà multipla), ma quando si effettua il downcast solitamente ci sono diverse possibilità per effettuare il cast. In particolare, un **Circle** è un tipo di **Shape** (questo è un upcast), ma se si prova a fare il downcast di un **Shape** potrebbe essere un **Circle**, un **Square**, un **Triangle**, ecc. Quindi si pone il problema di un downcast sicuro. (Ma una cosa più importante da chiedersi è perché si sta utilizzando il downcast invece di utilizzare semplicemente il polimorfismo per ottenere automaticamente il tipo corretto. I motivi per cui si preferisce non utilizzare il downcast sono trattati nel Volume 2 di questo libro.)

Il C++ mette a disposizione un *cast esplicito* (introdotto nel Capitolo 3) chiamato **dynamic_cast** che è un'operazione di *downcast sicura*. Quando si usa il dynamic_cast per provare a fare il downcast verso un tipo particolare, il valore restituito sarà un puntatore al tipo desiderato solo se il cast è consentito e viene effettuato con successo, altrimenti viene restituito zero per indicare che il tipo non era corretto. Ecco un piccolo esempio:

```
//: C15:DynamicCast.cpp
#include <iostream>
using namespace std;

class Pet { public: virtual ~Pet(){}};
class Dog : public Pet {};
class Cat : public Pet {};

int main() {
  Pet* b = new Cat; // Upcast
  // Prova a fare il cast a Dog*:
  Dog* d1 = dynamic_cast<Dog*>(b);
  // Prova a fare il cast a Cat*:
  Cat* d2 = dynamic_cast<Cat*>(b);
  cout << "d1 = " << (long)d1 << endl;
  cout << "d2 = " << (long)d2 << endl;
} ///:~
```

Quando si usa **dynamic_cast**, si deve lavorare con una vera gerarchia polimorfica – con funzioni virtuali – perché **dymanic_cast** utilizza informazioni memorizzate nella VTABLE per determinare il tipo corrente. In questo caso, la classe base contiene un distruttore virtuale e questo è sufficiente. Nel **main()**, viene fatto l'upcast di un puntatore a **Cat** verso uno a **Pet** e poi viene fatto un dowcast su entrambi i puntatori a **Dog** e a **Cat**. Entrambi i puntatori vengono stampati e si potrà vedere quando verrà lanciato il programma che il downcast sbagliato restituirà uno zero. Quindi, ogni volta che si effettua il downcast bisogna controllare il risultato dell'operazione per assicurarsi che sia diverso da zero. Inoltre, non si dovrebbe assumere che il puntatore sarà esattamente lo stesso,

perché a volte si hanno degli aggiustamenti sul puntatore durante l'upcast ed il downcast (in particolare nel caso di ereditarietà multipla).

Un **dynamic_cast** richiede un piccolo extra overhead quando viene eseguito; ma se vengono fatti molti **dynamic_cast** (nel qual caso bisognerebbe porsi domande serie sul progetto) questo potrebbe diventare un punto critico per le prestazioni. In alcuni casi si vuole conoscere qualcosa di speciale durante il downcast che permetta di essere sicuri sul tipo con cui si sta lavorando, in questo caso l'extra overhead dovuto al **dynamic_cast** non è necessario, ed, al suo posto, si può utilizzare un **static_cast**. Ecco come dovrebbe funzionare:

```
//: C15:StaticHierarchyNavigation.cpp
// Navigare gerarchie di classi con static_cast
#include <iostream>
#include <typeinfo>
using namespace std;

class Shape { public: virtual ~Shape() {}; };
class Circle : public Shape {};
class Square : public Shape {};
class Other {};

int main() {
  Circle c;
  Shape* s = &c; // Upcast: normale ed OK
  // Più esplicito ma non necessario:
  s = static_cast<Shape*>(&c);
  // (Dato che l'upcast e un'operazione comune e sicura,
  // il the cast diventa superfluo)
  Circle* cp = 0;
  Square* sp = 0;
  // La navigazione statica di una gerarchia di classi
  // richiede informazioni extra sul tipo:
  if(typeid(s) == typeid(cp)) // C++ RTTI
    cp = static_cast<Circle*>(s);
  if(typeid(s) == typeid(sp))
    sp = static_cast<Square*>(s);
  if(cp != 0)
    cout << "It's a circle!" << endl;
  if(sp != 0)
    cout << "It's a square!" << endl;
  // La navigazione statica è SOLO un trucco per l'efficienza;
  // Il dynamic_cast è sempre più sicuro. Comunque:
  // Other* op = static_cast<Other*>(s);
  // da un messaggio d'errore che può essere utile, mentre
  Other* op2 = (Other*)s;
  // non lo fa
} ///:~
```

In questo programma, è stata usata una nuova caratteristica che verrà descritta completamente nel Volume 2 di questo libro, dove un intero capitolo sarà dedicato all'argomento: *Identificazione del tipo a run-time con il C++* (RTTI). L'RTTI permette di scoprire l'informazione sul tipo andata persa dopo un upcast. Il **dynamic_cast** di fatto è una forma di RTTI. In questo caso la parola chiave **typeid** (dichiarata nel file di header **<typeinfo>**) viene utilizzata per identificare il tipo dei puntatori. Si può notare che il tipo di un puntatore a **Shape** ottenuto mediante upcast viene successivamente confrontato con un puntatore a **Circle** e a **Square** per scoprire il tipo reale. L'RTTI è molto di più che

l'utilizzo di **typeid** e si può anche immaginare che sarebbe abbastanza facile implementare il proprio sistema di informazioni sul tipo utilizzando le funzioni virtuali.

Un oggetto **Circle** viene creato e viene fatto l'upcast del suo indirizzo ad un puntantore a **Shape**; la seconda versione di quest'operazione mostra come si può utilizzare **static_cast** per essere più espliciti rispetto l'upcast. Comunque, dato che un upcast è sempre un'operazione sicura e comune, l'autore considera un upcast esplicito poco chiaro e non necessario

L'RTTI viene utilizzato per determinare il tipo, poi **static_cast** effettua il downcast. Si noti, però, che in questo esempio il procedimento è esattamente lo stesso utilizzato con **dynamic_cast**, cioè il programmatore deve effettuare dei test e scoprire il cast corretto. Tipicamente si vorrebbe una situazione che sia più deterministica rispetto a quella presentata nell'esempio precedente quando si utilizza **static_cast** piuttosto che **dynamic_cast** (e, di nuovo, si dovrà riesaminare il progetto con attenzione prima di utilizzare un **dynamic cast**).

Se una gerarchia di classi non possiede funzioni **virtuali** (che è una scelta discutibile) oppure se si hanno altre informazioni che permettono un downcast sicuro, è un po' più veloce effettuare il downcast staticamente piuttosto che con il **dynamic_cast**. In aggiunta, l'uso di **static_cast** non permette di effettuare il cast al di fuori della gerarchia, come permetterebbe il cast tradizionale, quindi è più sicuro. Comunque, navigare staticamente le gerarchie di classi è sempre rischioso e si dovrebbe utilizzare il **dynamic_cast** se non ci si vuole trovare in situazioni strane.

Sommario

Polimorfismo – implementato nel C++ attraverso l'uso delle funzioni virtuali – significa "forme diverse." Nella programmazione object-oriented, si ha la stessa faccia (l'interfaccia comune nella classe base) e modi differenti di utilizzarla: le differenti versioni delle funzioni virtuali.

Si è visto, in questo capitolo, che è impossibile capire, o anche creare, un esempio di polimorfismo senza utilizzare l'astrazione dei dati e l'ereditarietà. Il polimorfismo è una caratteristica che non può essere vista in maniera isolata (come, per esempio, le istruzioni **const** o **switch**), ma lavora assieme ad altre proprietà del linguaggio, come una parte del "grande quadro" delle relazioni tra classi. Le persone vengono spesso confuse dalle altre caratteristiche del C++ non orientate agli oggetti, come l'overload e gli argomenti di default, che a volte vengono presentati come caratteristiche orientate agli oggetti. Non bisogna farsi ingannare; se non è late-binding non è polimorfismo.

Per utilizzare il polimorfismo – e, quindi, le tecniche di programmazione object-oriented – nei propri programmi bisogna espandere il concetto di programmazione per includere non solo i membri e i messaggi di una sola classe, ma anche gli aspetti comuni e le relazioni tra le varie classi. Sebbene questo richiede uno sforzo significativo, è una battaglia che vale la pena affrontare, perché si otterranno una maggiore velocità nello sviluppo del programma, una migliore organizzazione del codice, programmi estendibili ed una manutenzione del codice semplificata.

Il polimorfismo completa la serie di caratteristiche object-oriented del linguaggio, ma ci sono altre due caratteristiche importanti nel C++: i template (che verranno introdotti nel

Capitolo 16 e verranno trattati in maggior dettaglio nel Volume 2) e la gestione delle eccezioni (che verrà trattata nel Volume 2). Queste caratteristiche forniscono un aumento della potenza di programmazione come ognuna delle caratteristiche object-oriented: astrazione dei dati, ereditarietà e polimorfismo.

---

[54] I compilatori possono implementare il meccanismo delle funzioni virtuali in qualsiasi modo, ma l'approccio qui descritto è quello adottato dalla maggior parte dei compilatori.

[55] Alcuni compilatori potrebbero utilizzare le stesse dimensioni mostrate in questo libro, ma questo sarà sempre più raro con il passare del tempo.

[56] Smalltalk, Java e Python, per esempio, usano questo approccio con grande successo.

[57] Ai Bell Labs, dove il C++ è stato inventato, ci sono molti programmatori C. Renderli tutti più produttivi, anche di poco, fa risparmiare all'azienda diversi milioni.

[58] In verità, non su tutte le macchine i puntatori hanno le stesse dimensioni . Nel contesto che si sta trattando, però, si può ritenere questa affermazione valida.

16: Introduzione ai Template

Ereditarietà e composizione forniscono un modo di riutilizzare il codice oggetto. La caratteristica *template* in C++ fornisce un modo di riutilizzare il codice *sorgente*.

Nonostante i template C++ siano un tool di programmazione per tutti gli usi, quando furono introdotti nel linguaggio, sembrarono scoraggiare l'uso di gerarchie di classi-contenitore basate sugli oggetti (dimostrate alla fine del Capitolo 15). Per esempio, i contenitori e gli algoritmi Standard C++ (spiegati in due capitoli del Volume 2 di questo libro, scaricabile da *www.BruceEckel.com*) sono costituiti esclusivamente da template e sono relativamente facili da usare per il programmatore.

Questo capitolo non solo spiega i concetti base dei template, ma è anche un'introduzione ai contenitori, che sono componenti fondamentali della progammazione orientata agli oggetti e sono quasi completamente realizzati attraverso i contenitori nella Libreria Standard C++. Si capirà che nel libro sono stati usati esempi di contenitori – lo **Stash** e lo **Stack** – , apposta per rendere il lettore avvezzo all'idea dei contenitori; in questo capitolo sarà anche aggiunto il concetto di *iteratore*. Nonostante i contenitori siano esempi ideali di uso di template, nel Volume 2 (in cui c'è un capitolo avanzato sui template) si imparerarà che esistono anche molti altri loro usi.

Container (Contenitori)

Si supponga di voler creare uno stack, come abbiamo fatto nel libro. Questa classe stack conserverà **int**, tanto per farla semplice:

```
//: C16:IntStack.cpp
// Semplice stack di interi
//{L} fibonacci
#include "fibonacci.h"
#include "../require.h"
#include <iostream>
using namespace std;

class IntStack {
  enum { ssize = 100 };
  int stack[ssize];
  int top;
public:
  IntStack() : top(0) {}
  void push(int i) {
    require(top < ssize, "Troppi inserimenti eseguiti chiamando push( )");
    stack[top++] = i;
  }
  int pop() {
    require(top > 0, "Troppe chiamate alla funzione pop( )");
    return stack[--top];
  }
};

int main() {
  IntStack is;
  // Aggiunge qualche numero di Fibonacci, per renderlo più interessante:
```

```
  for(int i = 0; i < 20; i++)
    is.push(fibonacci(i));
  // Li recupera & li stampa:
  for(int k = 0; k < 20; k++)
    cout << is.pop() << endl;
} ///:~
```

La classe **IntStack** è un esempio banale di stack push-down (implementa, cioè, una pila, che usa la tecnica *Last In First Out*: l'ultimo elemento inserito è il primo ad essere recuperato, *ndt*). Qui per semplicità è stato creato di dimensioni fisse, ma è possibile anche modificarlo espandendolo automaticamente allocando memoria fuori dall'heap, come nella classe **Stack** che è stata esaminata nel libro.

main( ) aggiunge qualche intero allo stack , e li restituisce anche. Per rendere l'esempio più interessante, gli interi sono creati con la funzione di **fibonacci( )**, che genera numeri secondo la tradizionale "rabbit-reproduction". Qui c'è l' header file che dichiara la funzione:

```
//: C16:fibonacci.h
// Generatore di numeri di Fibonacci
int fibonacci(int n); ///:~
```

Qui c'è l'implementazione:

```
//: C16:fibonacci.cpp {O}
#include "../require.h"

int fibonacci(int n) {
  const int sz = 100;
  require(n < sz);
  static int f[sz]; // Initializzato a zero
  f[0] = f[1] = 1;
  // cerca elementi dell'array ancora settati a zero:
  int i;
  for(i = 0; i < sz; i++)
    if(f[i] == 0) break;
  while(i <= n) {
    f[i] = f[i-1] + f[i-2];
    i++;
  }
  return f[n];
} ///:~
```

Questa è un'implementazione ragionevolmente efficiente , poichè non genera mai due numeri uguali. Usa un array **static** di **int**, e conta sul fatto che il compilatore inizializzerà un array **static** a zero. Il primo ciclo **for** incrementa l'indice **i** fino a che non si raggiunge il primo elemento nullo dell'array, quindi un ciclo **while** aggiunge i numeri di Fibonacci all'array finchè non viene raggiunto l'elemento desiderato. Ma si noti che se i numeri di Fibonacci fino all'elemento **n**-simo sono già inizializzati, il ciclo **while** viene saltato del tutto.

La necessità di contenitori

Ovviamente, uno stack di interi (siffatto) non è un di grande utilità. Il bisogno reale di contenitori viene quando si iniziano a creare oggetti sulla heap usando **new** e li si distrugge con **delete**. In un problema di programmazione generale, , non si sa di quanti

oggetti si avrà bisogno nel corso della stesura del programma. Per esempio, in un sistema di controllo del traffico aereo non si vuole limitare il numero di aerei che il sistema può controllare. Non si vuole che il programma termini soltanto perchè abbiamo superato qualche numero. In un sistema CAD (computer-aided design) , si ha a che fare con molte figure, ma solo l'utente determina (a tempo di esecuzione) esattamente di quante figure stiamo per aver bisogno. Dopo aver notato questa tendenza, il lettore scoprirà un gran numero di esempi nelle sue situazioni di programmazione.

I programmatori C che fanno affidamento sulla memoria virtuale per la loro "gestione della memoria" spesso trovano fastidiosa l'idea di **new**, **delete** e delle classi contenitore. Apparentemente, una pratica in C è creare un enorme array globale, più grande di qualsiasi cosa di cui il programma sembrerebbe necessitare. Questo può non richiedere molti pensieri (o la consapevolezza di **malloc( )** e **free( )**), ma produce programmi che non funzionano bene e che nascondono bachi sottili.

Inoltre, se si crea un enorme array di oggetti in C++, l'overhead (il dispendio di risorse) del costruttore e del distruttore può rallentare significativamente le cose. L'approccio C++ lavora molto meglio: quando si ha bisogno di un oggetto, lo si crea con **new,** e si inserisce il suo puntatore in un contenitore. Più tardi, lo si ripesca e ci si fa qualcosa. In questo modo, si creano soltanto gli oggetti di cui si ha assoluto bisogno. E usualmente all'avvio del programma non si hanno a disposizione tutte le condizioni di inizializzazione. **new** consente di aspettare finchè qualcosa accada nell'ambiente prima che si possa di fatto creare l'oggetto.

Così nella maggior parte delle situazioni comuni si farà un contenitore che conserva i puntatori a qualche oggetto di interesse. Si creeranno questi oggetti usando **new** ed inserendo il puntatore risultante nel contenitore (potenzialmente facendo su esso un casting all'insù nel processo), tirandolo fuori dopo nel momento in cui si vuole fare qualcosa con l'oggetto. Questa tecnica produce la maggior parte dei tipi flessibili, generali di programma.

Panoramica sui template

Ora si solleva un problema. Abbiamo un **IntStack**, che conserva interi. Ma vorremmo uno stack che conservi figure o velivoli o piante o qualsiasi altra cosa. Reinventare il nostro codice sorgente ogni volta non sembra essere un approccio molto intelligente con un linguaggio che procaccia la riusabilità. Ci dev'essere un modo migliore.

Ci sono tre tecniche per il riuso del codice sorgente in questa situazione: la strada del C, presentata qui per contrasto; l'approccio Smalltalk, che ha influenzato pesantemente il C++ e l'approccio C++: i template.

La soluzione C. Naturalmente state provando a tirarvi fuori dall'approccio C poichè è incasinato ed incline agli errori e completamente inelegante. In questo approccio, si copia il codice sorgente di **Stack** e si fanno le modifiche a mano, introducendo nuovi errori nel processo. Questa è certamente una tecnica non molto produttiva.

La soluzione Smalltalk. Smalltalk (e il Java, che segue il suo esempio) ha impiegato un approccio semplice e diretto: se si vuole riusare del codice, allora si usi l'ereditarietà. Per implementare ciò, ciascuna classe contenitore conserva gli elementi della classe base generica **Object** (simile all'esempio alla fine del Capitolo 15). Ma poichè la libreria in

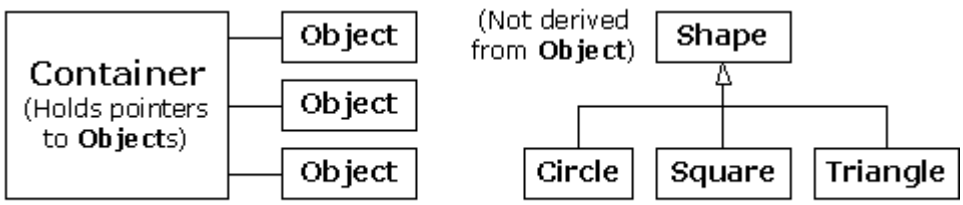
Smalltalk è di importanza fondamentale, non si crea mai una classe da zero. Al contrario, si deve sempre ereditarla da una classe esistente. Si trova una classe prossima il più possibile a quella voluta, si eredita da essa, e si fanno pochi cambiamenti. Ovviamente, questo è un un beneficio perché minimizza il nostro sforzo (e spiega perchè si spende molto tempo ad imparare la libreria della classe prima di diventare un effiente programmatore Smalltalk).

Ma significa anche che tutte le classi in Smalltalk finiscono con l'essere parte di un singolo albero di ereditarietà. Si deve ereditare da un ramo di questo albero quando si sta creando una nuova classe. La maggior parte dell'albero c'è già (è la libreria della classe Smalltalk), e alla radice dell'albero c'è una classe chiamata **Object** – la stessa classe che ciascun contenitore Smalltalk conserva.

Questo è un raggiro ordinato poichè significa che ogni classe nella gerarchia di classe Smalltalk (e Java[59]) è derivata da **Object**, così ogni classe può essere conservata in ogni contenitore (incluso quel contenitore stesso). Questo tipo di gerarchia ad albero singolo basata su un tipo generico fondamentale (spesso chiamato **Object**, anche in Java) è detta "gerarchia basata sugli oggetti ." E' possibile che si sia sentito questo termine e lo si sia interpretato come qualche nuovo concetto fondamentale nella OOP, come il polimorfismo. Semplicemente si riferisce ad una gerarchia di classe con **Object** (o qualche nome simile) alla sua radice e a classi contenitore che conservano **Object**.

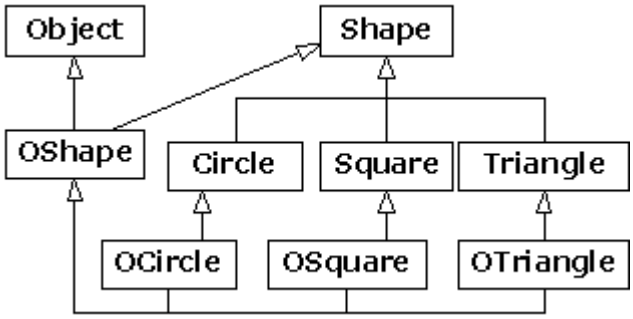
Poichè la libreria della classe Smalltalk ha una storia ed un'esperienza molto più lunga rispetto al C++, e poichè i compilatori C++ originali *non* avevano librerie di classi contenitore, sembrò una buona idea duplicare la libreria Smalltalk in C++. Questo fu fatto come esperimento con un'iniziale implementazione del C++ [60], e poichè rappresentava un porzione significativa di codice, molta gente ha cominciato usando questa implementazione. Provando ad usare le classi contenitore, scoprirono un problema.

Il problema era che in Smalltalk (e nella maggior parte degli altri linguaggi OOP di cui sia a conoscenza), tutte le classi sono automaticamente derivate da una singola gerarchia, ma questo non è vero in C++. Si poteva avere la propria bella gerarchia basata sugli oggetti con le sue classi contenitore, ma si poteva comperare un insieme di classi figura o di classi velivolo da un altro fornitore che non usava questa gerarchia. (Usare questa gerarchia impone overhead, che i programmatori C evitano.) Come fare ad inserire un albero di classe separato nella classe contenitore nella propria gerarchia basata sugli oggetti? Ecco come appare il problema:



Poichè il C++ supporta gerarchie indipendenti multiple, Smalltalk è una gerarchia basata sugli oggetti che non lavora così bene.

La soluzione è sembrata ovvia. Se si possono avere molte gerarchie di ereditarietà, allora si dovrebbe essere in grado di ereditare da più di una sola classe: l'ereditarietà multipla risolverà il problema. Così si fa quanto segue (un esempio simile era stato dato alla fine del Capitolo 15):



Ora **OShape** ha caratteristiche e comportamenti di **Shape**, ma poichè è anche derivata da **Object** può essere messa in **Container**. L'eredità extra in **OCircle**, **OSquare**, ecc. è necessaria affinchè queste classi possano essere castate all'insù verso **OShape** e quindi mantengano il comportamento corretto. Si può vedere le cose stanno ingarbugliandosi rapidamente.

I fornitori di compilatori hanno inventato ed incluso le proprie gerarchie di classi contenitore basate sugli oggetti, la maggior parte delle quali son state da allora rimpiazzate dalle versioni template. Si può arguire che per risolvere problemi di programmazione generale è stato necessario ricorrere all'ereditarietà multipla, ma si vedrà nel secondo volume di questo libro che la sua complessità è meglio evitarla eccetto in casi speciali.

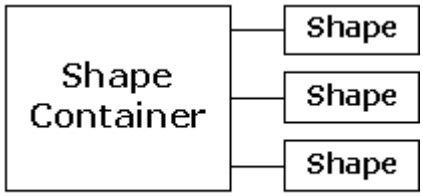
La soluzione template

Nonostante una gerarchia basata sugli oggetti con l'eredità multipla sia concettualmente semplice, risulta essere penosa da usare. Nel suo libro originale [61] Stroustrup ha dimostrato che cosa abbia considerato quale alternativa preferibile alla gerarchia basata sugli oggetti. Le classi contenitore sono state create come vaste macro del preprocessore con argomenti che potrebbero essere sostituiti con il tipo desiderato. Quando si voleva creare un contenitore che conservasse un tipo particolare, si facevano due chiamate macro.

Sfortunatamente, questo approccio era confuso da tutta l'esistente letteratura Smalltalk e dall'esperienza di programmazione, ed era un tantino poco maneggevole. Fondamentalmente, nessuno lo possedeva.

Nel frattempo, Stroustrup ed il team C++ dei Laboratori Bell avevano modificato il loro originale approccio con le macro, semplificandolo e spostandolo dal dominio del preprocessore a quello del compilatore. Questo nuovo dispositivo di sostituzione del codice è chiamato **template**[62], e rappresenta un modo completamente diverso di riusare il codice. Invece di riusare il codice oggetto, come l'ereditarietà e la composizione, un template riusa il *codice sorgente*. Il contenitore non conserva più una classe base generica detta **Object**, al contrario conserva un parametro non specificato. Quando si usa un template, il parametro è sostituito *dal compilatore*, come nel vecchio approccio con le macro, ma è più chiaro e semplice da usare.

Ora, invece di preoccuparsi dell'ereditarietà o della composizione quando si vuole usare una classe contenitore, si prende la versione template del contenitore e si caratterizza così da ottenerne una specifica versione per il proprio particolare problema, come questo:



Il compilatore fa il lavoro per noi, e alla fine il risultato è che abbiamo esattamente il contenitore che ci serve per il nostro lavoro, invece di una gerarchia di ereditarietà poco maneggevole. In C++, il template implementa il concetto di *tipo parametrizzato*. Un altro benefico dell'approccio dei template è che il programmatore principiante che può essere poco familiare o trovare poco facile l'ereditarietà può ancora usare le classi contenitore incapsulate da subito. (come abbiamo fatto con **vector** nel libro).

La sintassi di template

La parola chiave **template** dice al compilatore che la definizione di classe che segue manipolerà uno o più tipi non specificati. Nel momento in cui il codice della classe reale è generato dal template, questi tipi devono essere specificati, cosicchè il compilatore possa sostituirli.

Per dimostrare la sintassi, qui c'è un piccolo esempio che produce un array bounds-checked (si controlla che l'indice non cada al di fuori delle dimensioni dell'array, *ndt*) :

```
//: C16:Array.cpp
#include "../require.h"
#include <iostream>
using namespace std;

template<class T>
class Array {
  enum { size = 100 };
  T A[size];
public:
  T& operator[](int index) {
    require(index >= 0 && index < size,
      "Indice fuori dal range");
    return A[index];
  }
};

int main() {
  Array<int> ia;
  Array<float> fa;
  for(int i = 0; i < 20; i++) {
    ia[i] = i * i;
    fa[i] = float(i) * 1.414;
  }
  for(int j = 0; j < 20; j++)
    cout << j << ": " << ia[j]
         << ", " << fa[j] << endl;
} ///:~
```

Si può vedere che appare come una classe normale eccetto per la linea

```
template<class T>
```

che dice che **T** è il parametro della sostituzione , e che rappresenta un nome di tipo. Inoltre, si vede che **T** è usato nella classe ovunque si vorrebbe normalmente vedere il tipo specifico che conserva il contenitore.

In **Array**, gli elementi sono inseriti *e* estratti con la stessa funzione: la sovraccaricata **operator [ ]** . Questa ritorna un riferimento, così tale valore di ritorno può essere usato in entrambi i lati di un segno di uguaglianza (cioè, sia come un *lvalue* che un *rvalue*). Si noti che se l'indice è fuori dai confini, è usata la funzione **require( )** per stampare un messaggio. Poichè **operator[]** è una funzione **inline**, si portebbe usare questo approccio per garantire che non si verifichino violazioni dei limiti dell'array, rimuovendo quindi la funzione **require( )** per la portabilità del codice.

In **main( )**, si può vedere come sia facile creare **Array**s che conservino differenti tipi di oggetti. Quando si dice

```
Array<int> ia;
Array<float> fa;
```

il compilatore espande il template **Array** (questa è detta *istanziazione*) due volte, per creare due nuove *classi generate*, che si possono pensare come **Array_int** e **Array_float**. (Compilatori differenti possono decorare i nomi in modi diversi.) Queste classi sono proprio quelle che si sarebberero create se si fosse compiuta la sostituzione a mano, eccetto per il fatto che è il compilatore a crearle per noi quando definiamo gli oggetti **ia** e **fa**. Si noti anche che le definizioni di classe duplicate sono o evitate dal compilatore o fuse dal linker.

Definizioni di funzioni non inline

Naturalmente, ci sono delle volte in cui desidereremo avere definizioni di funzioni membro non inline . In tal caso, il compilatore ha bisogno di vedere la dichiarazione **template** prima della definizione della funzione membro. Qui c'è l'esempio sopra, modificato per mostrare la definizione non inline di una funzione membro:

```
//: C16:Array2.cpp
// Definizione di template non inline
#include "../require.h"

template<class T>
class Array {
  enum { size = 100 };
  T A[size];
public:
  T& operator[](int index);
};

template<class T>
T& Array<T>::operator[](int index) {
  require(index >= 0 && index < size,
    "Indice fuori dal range");
  return A[index];
}

int main() {
  Array<float> fa;
  fa[0] = 1.414;
} ///:~
```

Qualsiasi riferimento ad un nome di classe template dev'essere accompagnato dalla sua lista di argomenti template, come in **Array<T>::operator[]**. Si può immaginare che internamente, si stia decorando il nome della classe con gli argomenti nella lista degli argomenti template per produrre un identificatore di nome di classe unico per ogni istanziazione di template.

Header files

Anche se si creano definizioni non inline di funzioni, usualmente si vorrà inserire tutte le dichiarazioni *e* tutte le definizioni per un template in un header file. Questa può apparire come una violazione della normale regola dell'header file "non inserire alcunchè che allochi memoria," (la quale previene errori di definizioni multiple al momento del linkaggio), ma le definizioni template sono speciali. Qualsiasi cosa preceduta da **template<...>** significa che il compilatore non allocherà memoria per essa in quel punto, ma aspetterà finchè sia chiamata (da un'istanziazione di template), e che da qualche parte nel compilatore e nel linker c'è un meccanismo per la rimozione di definizioni multiple di un identico template. Così si metteranno quasi sempre nell'header file *sia* la dichiarazione *sia* la definizione dell'intero template, per la facilità d'uso.

A volte si può avere bisogno di mettere le definizioni template in un file **cpp** separato per soddisfare necessità particolari (per esempio, forzare le istanziazioni template ad esistere solamente in un singolo file Windows **dll**). La maggior parte dei compilatori ha qualche meccanismo che lo consente; si dovrà studiare la documentazione del proprio particolare compilatore per usarlo.

Alcuni ritengono che mettere tutto il codice sorgente per la propria implementazione in un header file, esponga alla possibilità che la gente rubi o modifichi il codice se acquista la nostra libreria. Questa può essere un'obiezione, ma probabilmente dipende dal modo in cui si osserva il problema: sta acquistando un prodotto od un servizio? Se è un prodotto, allora si deve fare tutto il possibile per proteggerlo, e probabilmente non si vorrà fornire il codice sorgente, ma solo il codice compilato. Ma molta gente vede il software come un servizio, e ancor più di questo, un servizio di abbonamento. Il cliente vuole la consulenza (del creatore del software), vuole che egli continui la manutenzione di questo pezzo di codice riusabile, cosicchè non debba farlo lui – così può focalizzarsi sull'acquisire il *suo* lavoro fatto. Personalmente penso che la maggior parte dei clienti tratterà il creatore del software come una valida risorsa e non vorranno compromettere il loro rapporto con lui. Per quanto riguarda i pochi che desiderino rubare piuttosto che comprare o fare un lavoro originale, probabilmente non possono continuare comunque con lui.

IntStack come un template

Qui ci sono il contenitore e l'iteratore di **IntStack.cpp**, implementati come una classe contenitore generica usando i template:

```
//: C16:StackTemplate.h
// Semplice stack template
#ifndef STACKTEMPLATE_H
#define STACKTEMPLATE_H
#include "../require.h"

template<class T>
class StackTemplate {
  enum { ssize = 100 };
```

```
  T stack[ssize];
  int top;
public:
  StackTemplate() : top(0) {}
  void push(const T& i) {
    require(top < ssize, "Troppi inserimenti eseguiti chiamando push( )");
    stack[top++] = i;
  }
  T pop() {
    require(top > 0, "Troppe chiamate alla funzione pop()");
    return stack[--top];
  }
  int size() { return top; }
};
#endif // STACKTEMPLATE_H ///:~
```

Si noti che un template fa certe assunzioni riguardo agli oggetti che sta conservando. Per esempio, **StackTemplate** assume che ci sia un qualche tipo di operazione di assegnamento per **T** dentro la funzione **push( )**. Si potrebbe dire che un template "implichi un'interfaccia" per i tipi che è capace di conservare.

Un altro modo per dire questo è che i template forniscono un tipo di meccanismo di *tipizzazione debole* per il C++, che è ordinariamente un linguaggio fortemente tipizzato. Invece di insistere affinchè un oggetto sia di un qualche tipo esatto allo scopo di essere accettabile, la tipizzazione debole richiede solo che le funzioni membro che esso vuole chiamare siano *disponibili* per un particolare oggetto. Allora, il codice debolmente tipizzato può essere applicato a qualsiasi ogegtto che possa accettare quelle chiamate a funzione, ed è quindi molto più flessibile.[63].

Qui c'è l'esempio rivisto per testare il template:

```
//: C16:StackTemplateTest.cpp
// Test sul semplice stack template
//{L} fibonacci
#include "fibonacci.h"
#include "StackTemplate.h"
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

int main() {
  StackTemplate<int> is;
  for(int i = 0; i < 20; i++)
    is.push(fibonacci(i));
  for(int k = 0; k < 20; k++)
    cout << is.pop() << endl;
  ifstream in("StackTemplateTest.cpp");
  assure(in, "StackTemplateTest.cpp");
  string line;
  StackTemplate<string> strings;
  while(getline(in, line))
    strings.push(line);
  while(strings.size() > 0)
    cout << strings.pop() << endl;
} ///:~
```

La sola differenza è la creazione di **is**. Dentro la lista degli argomenti template si specifica il tipo di oggetto che lo stack e l'iteratore dovrebbero conservare. Per dimostrare la genericità

del template, è creato anche uno **StackTemplate** per conservare **string**. Questo è testato leggendo linee dal file del codice sorgente.

Le costanti nei template

Gli argomenti dei template non si limitano ai tipi di classe; si possono usare anche tipi built-in. I valori di questi argomenti diventano allora costanti a tempo di compilazione per quella particolare istanziazione del template. Si possono usare inoltre i valori di default per questi argomenti. L'esempio seguente consente di settare le dimensioni della classe **Array** durante l'istanziazione, ma fornisce anche un valore di default:

```
//: C16:Array3.cpp
// Tipi built-in come argomenti di template
#include "../require.h"
#include <iostream>
using namespace std;

template<class T, int size = 100>
class Array {
  T array[size];
public:
  T& operator[](int index) {
    require(index >= 0 && index < size,
      "Indice fuori dal range");
    return array[index];
  }
  int length() const { return size; }
};

class Number {
  float f;
public:
  Number(float ff = 0.0f) : f(ff) {}
  Number& operator=(const Number& n) {
    f = n.f;
    return *this;
  }
  operator float() const { return f; }
  friend ostream&
    operator<<(ostream& os, const Number& x) {
      return os << x.f;
  }
};

template<class T, int size = 20>
class Holder {
  Array<T, size>* np;
public:
  Holder() : np(0) {}
  T& operator[](int i) {
    require(0 <= i && i < size);
    if(!np) np = new Array<T, size>;
    return np->operator[](i);
  }
  int length() const { return size; }
  ~Holder() { delete np; }
};

int main() {
  Holder<Number> h;
```

```
  for(int i = 0; i < 20; i++)
    h[i] = i;
  for(int j = 0; j < 20; j++)
    cout << h[j] << endl;
} ///:~
```

Come prima, **Array** è un array controllato di oggetti e fa in modo di prevenire che si indicizzi al di fuori dei limiti. La classe **Holder** è molto simile a **Array** eccetto per il fatto che ha un puntatore ad un **Array** invece di includere un oggetto di tipo **Array**. Questo puntatore non è inizializzato nel costruttore; l'inizializzazione è ritardata fino al primo accesso. Questa è detta *lazy initialization (inizializzazione pigra)*; si può usare una tecnica come questa se si stanno creando molti oggetti, ma non si accede a tutti, e si vuole salvare memoria.

Si noterà che il valore di **size** in entrambi i template non è mai immagazzinato internamente nella classe, ma è usato come se fosse un dato membro all'interno delle funzioni membro.

Stack e Stash come template

I ricorrenti problemi di "apparteneza" con le classi contenitore **Stash** e **Stack** che sono stati rivisitati durante questo libro vengono dal fatto che questi contenitori non erano in grado di sapere esattamente che tipi conservassero. L'ultimo arrivato è il "contenitore di **Object**" **Stack** che è stato visto alla fine del Capitolo 15 in **OStackTest.cpp**.

Se il programmatore cliente non rimuove esplicitamente tutti i puntatori agli oggetti che sono conservati nel contenitore, allora il contenitore dovrebbe essere in grado di cancellare correttamente questi puntatori. Cioè, il contenitore "possiede" qualsiasi oggetto che non sia stato rimosso, ed è quindi responsabile per la loro pulizia. L'ostacolo è stato che la pulizia richiede la conoscenza del tipo dell'oggetto, e la creazione di una classe contenitore generica richiede di *non* conoscere il tipo dell'oggetto. Con i template, tuttavia, si può scrivere codice che non conosce il tipo dell'oggetto, e istanziare facilmente una nuova versione di quel contenitore per ogni tipo che si desidera esso contenga. I contenitori istanziati individuali *conoscono* il tipo di oggetti che essi conservano e possono così chiamare il corretto distruttore (assumendo, nel caso tipico in cui sia incluso il polimorfismo, che sia stato fornito un distruttore virtuale).

Per lo **Stack** ciò risulta essere abbastanza semplice poiché tutte le funzioni membro possono essere ragionevolmente messe inline:

```
//: C16:TStack.h
// Lo Stack come un template
#ifndef TSTACK_H
#define TSTACK_H

template<class T>
class Stack {
  struct Link {
    T* data;
    Link* next;
    Link(T* dat, Link* nxt):
      data(dat), next(nxt) {}
  }* head;
public:
```

```
  Stack() : head(0) {}
  ~Stack(){
    while(head)
      delete pop();
  }
  void push(T* dat) {
    head = new Link(dat, head);
  }
  T* peek() const {
    return head ? head->data : 0;
  }
  T* pop(){
    if(head == 0) return 0;
    T* result = head->data;
    Link* oldHead = head;
    head = head->next;
    delete oldHead;
    return result;
  }
};
#endif // TSTACK_H ///:~
```

Se si confronta questo listato con l'esempio **OStack.h** alla fine del Capitolo 15, si vedrà che **Stack** è virtualmente identica, eccetto **Object** che è stato sostituito da **T**. Anche il programma di test è quasi identico, a parte che è stata eliminata la necessità della ereditarietà multipla da **string** e **Object** (ed allo stesso tempo per **Object** stesso). Ora non c'è la classe **MyString** ad annunciare la sua distruzione, così si è aggiunta una nuova piccola classe per mostrare un contenitore **Stack** che fa pulizia dei suoi oggetti:

```
//: C16:TStackTest.cpp
//{T} TStackTest.cpp
#include "TStack.h"
#include "../require.h"
#include <fstream>
#include <iostream>
#include <string>
using namespace std;

class X {
public:
  virtual ~X() { cout << "~X " << endl; }
};

int main(int argc, char* argv[]) {
  requireArgs(argc, 1); // Il nome del file è un argomento
  ifstream in(argv[1]);
  assure(in, argv[1]);
  Stack<string> textlines;
  string line;
  // Legge il  file e immagazzina le linee nello Stack:
  while(getline(in, line))
    textlines.push(new string(line));
  // Recupera qualche linea dallo stack:
  string* s;
  for(int i = 0; i < 10; i++) {
    if((s = (string*)textlines.pop())==0) break;
    cout << *s << endl;
    delete s;
  } // Il distruttore cancella le altre stringhe.
  // Mostra che avviene la corretta distruzione:
  Stack<X> xx;
```

```
  for(int j = 0; j < 10; j++)
    xx.push(new X);
} ///:~
```

Il distruttore per **X** è virtuale, non perchè sia necessario qui, ma perchè **xx** potrebbe essere usato dopo per conservare oggetti derivati da **X**.

Si noti quanto sia facile creare differenti tipi di **Stack** per **string** e per **X**. A causa del template, si ottiene il meglio di entrambi i mondi: la facilità d'uso della classe **Stack** assieme ad una adeguata pulizia.

Il puntatore Stash templatizzato

Riorganizzare il codice di **PStash** in un template non è proprio così semplice perchè ci sono diverse funzioni membro che dovrebbero essere non inline. Tuttavia, come template quelle definizioni di funzioni appartengono ancora all'header file (il compilatore ed il linker si prendono cura di qualsiasi problema di definizioni multiple). Il codice appare abbastanza simile all'ordinario **PStash** solo che si noterà che la dimensione dell'incremento (usato da **inflate( )**) è stata templatizzata come un parametro non di classe con un valore di default, cosicchè la dimensione dell'incremento possa essere modificata nel punto di istanzazione (si noti che questo significa che la dimensione dell'incremento è fissata; si può anche dedurre che la dimensione dell'incremento dovrebbe essere modificabile durante il tempo di vita dell'oggetto):

```
//: C16:TPStash.h
#ifndef TPSTASH_H
#define TPSTASH_H

template<class T, int incr = 10>
class PStash {
  int quantity; // Numbero di celle di memoria
  int next; // Prossima cella libera
  T** storage;
  void inflate(int increase = incr);
public:
  PStash() : quantity(0), next(0), storage(0) {}
  ~PStash();
  int add(T* element);
  T* operator[](int index) const; // Fetch
  // Rimuove il riferimento da questo PStash:
  T* remove(int index);
  // Numero di elementi in Stash:
  int count() const { return next; }
};

template<class T, int incr>
int PStash<T, incr>::add(T* element) {
  if(next >= quantity)
    inflate(incr);
  storage[next++] = element;
  return(next - 1); // Indice
}

// Appartenenza dei puntatori restanti:
template<class T, int incr>
PStash<T, incr>::~PStash() {
  for(int i = 0; i < next; i++) {
    delete storage[i]; // Puntatori NULL OK
```

```
    storage[i] = 0; // Solo per essere sicuro
  }
  delete []storage;
}

template<class T, int incr>
T* PStash<T, incr>::operator[](int index) const {
  require(index >= 0,
    "PStash::operator[] indice negativo");
  if(index >= next)
    return 0; // Per indicare la fine
  require(storage[index] != 0,
    "PStash::operator[] ha restituito un puntatore NULL");
  // Produce il puntatore all'elemento desiderato:
  return storage[index];
}

template<class T, int incr>
T* PStash<T, incr>::remove(int index) {
  // operator[] effettua controlli di validità:
  T* v = operator[](index);
  // "Rimuove" il puntatore:
  if(v != 0) storage[index] = 0;
  return v;
}

template<class T, int incr>
void PStash<T, incr>::inflate(int increase) {
  const int psz = sizeof(T*);
  T** st = new T*[quantity + increase];
  memset(st, 0, (quantity + increase) * psz);
  memcpy(st, storage, quantity * psz);
  quantity += increase;
  delete []storage; // Vecchia memoria
  storage = st; // Punta alla nuova memoria
}
#endif // TPSTASH_H ///:~
```

La dimensione di default dell'incremento usata qui è piccola al fine di garantire che si verifichino le chiamate ad **inflate( )** . In tal modo possiamo essere sicuri che lavori correttamente.

Per testare il controllo di appartenenza del **PStash** templatizzato, la seguente classe descriverà le creazioni e distruzioni di se stessa, e garantirà anche che tutti gli oggetti che sono stati creati siano stati anche distrutti. **AutoCounter** consentirà soltanto agli oggetti del suo tipo di essere creati sullo stack:

```
//: C16:AutoCounter.h
#ifndef AUTOCOUNTER_H
#define AUTOCOUNTER_H
#include "../require.h"
#include <iostream>
#include <set> // Contenitore della Libreria Standard C++
#include <string>

class AutoCounter {
  static int count;
  int id;
  class CleanupCheck {
    std::set<AutoCounter*> trace;
```

```
  public:
    void add(AutoCounter* ap) {
      trace.insert(ap);
    }
    void remove(AutoCounter* ap) {
      require(trace.erase(ap) == 1,
        "Si sta tentando di cancellare due volte AutoCounter");
    }
    ~CleanupCheck() {
      std::cout << "~CleanupCheck()"<< std::endl;
      require(trace.size() == 0,
       "Non tutti gli oggetti AutoCounter sono stati cancellati");
    }
  };
  static CleanupCheck verifier;
  AutoCounter() : id(count++) {
    verifier.add(this); // Registrazione di se stesso
    std::cout << "creato[" << id << "]"
              << std::endl;
  }
  // Prevennzione dell'assegnamento e della costruzione di copia:
  AutoCounter(const AutoCounter&);
  void operator=(const AutoCounter&);
public:
  // Si possono solo creare oggetti con con questo:
  static AutoCounter* create() {
    return new AutoCounter();
  }
  ~AutoCounter() {
    std::cout << "distruzione di[" << id
              << "]" << std::endl;
    verifier.remove(this);
  }
  // Stampa sia oggetti che puntatori:
  friend std::ostream& operator<<(
    std::ostream& os, const AutoCounter& ac){
    return os << "AutoCounter " << ac.id;
  }
  friend std::ostream& operator<<(
    std::ostream& os, const AutoCounter* ac){
    return os << "AutoCounter " << ac->id;
  }
};
#endif // AUTOCOUNTER_H ///:~
```

La classe **AutoCounter** fa due cose. Primo, numera sequenzialmente ogni istanza di **AutoCounter**: il valore di questo numero è conservato in **id**, ed il numero è generato usando il dato membro **static count**.

Secondo, e più complesso, una istanza **static** (detta **verifier**) della classe annidata **CleanupCheck** conserva una traccia di tutti gli oggetti **AutoCounter** che sono stati creati e distrutti, e segnala se non si sono cancellati tutti (cioè se c'è una falla di memoria). Questo comportamento è realizzato usando una classe **set** della libreria Standard C++, che è un meraviglioso esempio di come template ben progettati possano rendere facile la vita (si possono imparare tutti i contenitori nella libreria Standard C++ nel Volume 2 di questo libro, disponibile online).

La classe **set** è templatizzata sul tipo che essa conserva; qui è istanziata a conservare puntatori **AutoCounter**. Un **set** consentirà solo una istanza per ciascun distinto oggetto

da aggiungere; si può vedere che questo si compie in **add( )** con la funzione **set::insert( )**. **insert( )** in realtà ci informa con il suo valore di ritorno se stiamo provando ad aggiungere qualcosa che è già stato aggiunto; tuttavia, poichè si stanno aggiungendo gli indirizzi degli oggetti possiamo contare sulla garanzia del C++ che tutti gli oggetti abbiano indirizzi unici.

In **remove( )**, **set::erase( )** è usata per rimuovere un puntatore **AutoCounter** dal **set**. Il valore di ritorno ci dice quante istanze dell'elemento sono state rimosse; nel nostro caso ci aspettiamo solo o zero o uno. Se il valore è zero, tuttavia, significa che questo oggetto è stato già cancellato dal **set** e stiamo provando a cancellarlo una seconda volta, cosa che costituisce un errore di programmazione che sarà segnalato tramite **require( )**.

Il distruttore per **CleanupCheck** fa un controllo finale assicirandosi che la dimensione del **set** sia zero – questo significa che tutti gli oggetti sono stati cancellati correttamente. Se è diverso da zero, abbiamo una falla di memoria, segnalata attraverso **require( )**.

Il costruttore ed il distruttore di **AutoCounter** registrano and cancellano se stessi con l'oggetto **verifier** . Si noti che il costruttore, il costruttore di copia, e l'operatore di assegnamento sono **private**, così il solo modo per noi di creare un oggetto è con la funzione membro **static create( )** – questo è un semplice esempio di una *fabbrica*, ed essa garantisce che tutti gli oggetti siano creati nella heap, così **verifier** non si confonderà riguardo agli assegnamenti e alle costruzioni di copia.

Poichè tutte le funzioni membro sono state messe inline, la sola ragione per cui esiste il file di implementazione è quella di contenere le definizioni dei dati membro statici:

```
//: C16:AutoCounter.cpp {O}
// Definizione dei membri di classe statici
#include "AutoCounter.h"
AutoCounter::CleanupCheck AutoCounter::verifier;
int AutoCounter::count = 0;
///:~
```

Con **AutoCounter** in mano, possiamo ora testare le facilità di **PStash**. L'esempio seguente non solo mostra che distruttore di **PStash** cancella tutti gli oggetti che esso possiede correntemente, ma dimostra anche come la classe **AutoCounter** rilevi gli oggetti che non sono stati cancellati:

```
//: C16:TPStashTest.cpp
//{L} AutoCounter
#include "AutoCounter.h"
#include "TPStash.h"
#include <iostream>
#include <fstream>
using namespace std;

int main() {
  PStash<AutoCounter> acStash;
  for(int i = 0; i < 10; i++)
    acStash.add(AutoCounter::create());
  cout << "Rimuovine 5 manualmente:" << endl;
  for(int j = 0; j < 5; j++)
    delete acStash.remove(j);
  cout << "Rimuovine due senza cancellarli:"
       << endl;
  // ... per generare il messaggio di errore di cancellazione.
```

```
  cout << acStash.remove(5) << endl;
  cout << acStash.remove(6) << endl;
  cout << "Il distruttore cancella il resto:"
       << endl;
  // Ripeti il test dei capitoli precedenti:
  ifstream in("TPStashTest.cpp");
  assure(in, "TPStashTest.cpp");
  PStash<string> stringStash;
  string line;
  while(getline(in, line))
    stringStash.add(new string(line));
  // Stampa le stringhe:
  for(int u = 0; stringStash[u]; u++)
    cout << "stringStash[" << u << "] = "
         << *stringStash[u] << endl;
} ///:~
```

Quando gli elementi 5 e 6 di **AutoCounter** sono rimossi dal **PStash**, essi diventano responsabilità del chiamante, ma poichè questo non li cancella mai essi causano una falla di memoria, individuata da **AutoCounter** a run time.

Se si fa girare il programma, si vedrà che il messaggio di errore non è così specifico come potrebbe essere. Se si usa lo schema presentato in **AutoCounter** per scoprire falle di memoria nel proprio sistema, probabilmente si vorrà avere una versione di quello schema che stampi informazioni più dettagliate sugli oggetti che non sono stati cancellati. Il Volume 2 di questo libro mostra modi più sofisticati di fare questo.

Accendere e spegnere l’appartenenza

Ritorniamo al problema dell’appartenenza. I contenitori che conservano oggetti per valore usualmente non si sbagliano sull’appartenenza, poichè essi posseggono chiaramente gli oggetti che contengono. Ma se il nostro contenitore conserva puntatori (che è più comune in C++, specialmente con il polimorfismo), allora è molto probabile che questi puntatori possano anche essere usati da qualche altra parte nel programma, e noi non vogliamo necessariamente cancellare l’oggetto perchè allora gli altri puntatori nel programma si riferirebbero ad un oggetto ditrutto. Per evitare che questo accada, si deve considerare l’appartenenza quando si progetta e si usa un contenitore.

Molti programmi sono molto più semplici di questo, e non si incontra il problema dell’appartenenza: un contenitore conserva puntatori ad oggetti che sono usati solo da quel contenitore. In questo caso l’appartenenza è molto semplice: il contenitore possiede i suoi oggetti.

Il miglior approccio per trattare il problema dell’appartenenza è dare al programmatore cliente una scelta. Questo si realizza spesso tramite un argomento del costruttore che di default indica l’appartenenza (il caso più semplice). In più ci possono essere funzioni “get” e “set” per conoscere lo stato dell’appartenenza del contenitore e modificarlo. Se il contenitore ha funzioni che rimuovono un oggetto, lo stato di appartenenza usualmente influenza questa rimozione, così si possono anche trovare opzioni per controllare la distruzione nella funzione di rimozione. In teoria si sarebbero potuti aggiungere i dati di appartenenza per ciascun elemento nel contenitore, così ogni posizione avrebbe saputo se avesse avuto bisogno di essere distrutta; questa è una variante del conteggio del riferimento, ad eccezione del fatto che il contenitore e non l’oggetto sa il numero di riferimenti che puntano ad un oggetto.

```
//: C16:OwnerStack.h
// Stack con appartenenza controllabile a runtime
#ifndef OWNERSTACK_H
#define OWNERSTACK_H

template<class T> class Stack {
  struct Link {
    T* data;
    Link* next;
    Link(T* dat, Link* nxt)
      : data(dat), next(nxt) {}
  }* head;
  bool own;
public:
  Stack(bool own = true) : head(0), own(own) {}
  ~Stack();
  void push(T* dat) {
    head = new Link(dat,head);
  }
  T* peek() const {
    return head ? head->data : 0;
  }
  T* pop();
  bool owns() const { return own; }
  void owns(bool newownership) {
    own = newownership;
  }
  // Conversione di tipo in stesso tipo: vero se non vuoto:
  operator bool() const { return head != 0; }
};

template<class T> T* Stack<T>::pop() {
  if(head == 0) return 0;
  T* result = head->data;
  Link* oldHead = head;
  head = head->next;
  delete oldHead;
  return result;
}

template<class T> Stack<T>::~Stack() {
  if(!own) return;
  while(head)
    delete pop();
}
#endif // OWNERSTACK_H ///:~
```

Il comportamento di default del contenitore è di distrugegre i suoi oggetti, ma lo si può cambiare o modificando l'argomento del costruttore o usando le funzioni membro di lettura/scrittura **owns( )**.

Come con la maggior parte dei template che probabilmente si vedono, l'intera implementazione è contenuta nell'header file. Qui c'è un piccolo test che esercita le abilità di appartenenza:

```
//: C16:OwnerStackTest.cpp
//{L} AutoCounter
#include "AutoCounter.h"
#include "OwnerStack.h"
```

```
#include "../require.h"
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

int main() {
  Stack<AutoCounter> ac; // Appartenenza attiva
  Stack<AutoCounter> ac2(false); // Convertita in non
attiva
  AutoCounter* ap;
  for(int i = 0; i < 10; i++) {
    ap = AutoCounter::create();
    ac.push(ap);
    if(i % 2 == 0)
      ac2.push(ap);
  }
  while(ac2)
    cout << ac2.pop() << endl;
  // La distruzione è superflua poichè
  // ac "possiede" tutti gli oggetti
} ///:~
```

L'oggetto **ac2** non possiede gli oggetti che vengono in esso inseriti, così **ac** è il contenitore "master" che prende la responsabilità dell'appartenenza. Se si vuole cambiare, in parte nel corso del tempo di vita di un contenitore, il fatto che un contenitore possegga o meno i suoi oggetti, lo si può fare usando **owns( )**.

Sarebbe anche possibile cambiare la granulosità dell'appartenenza in modo tale che essa sia basata su un oggetto-da-oggetto, ma questo renderebbe probabilmente la soluzione al problema dell'appartenenza più complessa del problema stesso.

Conservare oggetti per valore

In realtà creare una copia degli oggetti all'interno di un generico contenitore è un problema complesso se non si hanno a disposizione i template. Con i template, le cose sono relativamente semplici – si dice solo che si stanno conservando oggetti invece che puntatori:

```
//: C16:ValueStack.h
// Conservare oggetti per valore in uno Stack
#ifndef VALUESTACK_H
#define VALUESTACK_H
#include "../require.h"

template<class T, int ssize = 100>
class Stack {
  // Il costruttore di default compie l'inizializzazione
  // dell'oggetto per ciascun elemento nell'array:
  T stack[ssize];
  int top;
```

```
public:
  Stack() : top(0) {}
  // Il costruttore di copia copia l'oggetto nell'array:
  void push(const T& x) {
    require(top < ssize, "Troppi inserimenti eseguiti chiamando push( )");
    stack[top++] = x;
  }
  T peek() const { return stack[top]; }
  // L'oggetto esiste ancora quando lo si recupera;
  // da questo momento esso semplicemente non è più disponibile:
  T pop() {
    require(top > 0, "Troppe chiamate alla funzione pop()");
    return stack[--top];
  }
};
#endif // VALUESTACK_H ///:~
```

Il costruttore di copia fa la maggior parte del lavoro per gli oggetti contenuti passando e ritornando gli oggetti per valore. Dentro **push( )**, la memorizzazione dell'oggetto dentro l'array **Stack** è compiuta con **T::operator=**. Per garantire che lavori, una classe detta **SelfCounter** tiene traccia delle creazioni dell'oggetto e delle copie fatte col costruttore di copia:

```
//: C16:SelfCounter.h
#ifndef SELFCOUNTER_H
#define SELFCOUNTER_H
#include "ValueStack.h"
#include <iostream>

class SelfCounter {
  static int counter;
  int id;
public:
  SelfCounter() : id(counter++) {
    std::cout << "Creato: " << id << std::endl;
  }
  SelfCounter(const SelfCounter& rv) : id(rv.id){
    std::cout << "Copiato: " << id << std::endl;
  }
  SelfCounter operator=(const SelfCounter& rv) {
    std::cout << "Assegnato " << rv.id << " to "
              << id << std::endl;
    return *this;
  }
  ~SelfCounter() {
    std::cout << "Distrutto: "<< id << std::endl;
  }
  friend std::ostream& operator<<(
    std::ostream& os, const SelfCounter& sc){
    return os << "SelfCounter: " << sc.id;
  }
};
#endif // SELFCOUNTER_H ///:~

//: C16:SelfCounter.cpp {O}
#include "SelfCounter.h"
int SelfCounter::counter = 0; ///:~

//: C16:ValueStackTest.cpp
//{L} SelfCounter
#include "ValueStack.h"
```

```
#include "SelfCounter.h"
#include <iostream>
using namespace std;

int main() {
  Stack<SelfCounter> sc;
  for(int i = 0; i < 10; i++)
    sc.push(SelfCounter());
  // OK a peek(), il risultato è un(a variabile) temporanea:
  cout << sc.peek() << endl;
  for(int k = 0; k < 10; k++)
    cout << sc.pop() << endl;
} ///:~
```

Quando un contenitore **Stack** viene creato, è chiamato il costruttore di default dell'oggetto contenuto per ogni oggetto nell'array. Inizialmente si vedranno 100 oggetti **SelfCounter** creati senza un'apparente ragione, ma questa è solo l'inizializzazione dell'array. Questo può essere un pò costoso, ma non c'è altro modo in un progetto semplice come questo. Una situazione effettivamente più complessa si presenta se si rende **Stack** più generico consentendo alle dimensioni di crescere dinamicamente, poichè nell'implementazione mostrata sopra questo includerebbe la creazione di un nuovo (più grande) array, copiare il vecchio array sul nuovo, e distruggere il vecchio array (questo è, appunto, quello che fa la classe **vector** della libreria Standard C++).

Introduzione agli iteratori

Un *iteratore* è un oggetto che si muove attraverso un contenitore di altri oggetti e li seleziona uno alla volta, senza fornire un accesso diretto all'implementazione di quel contenitore. Gli iteratori forniscono un modo standard di accedere agli elementi, che un contenitore fornisca un modo di accedere agli elementi direttamente o meno. Si vedranno gli iteratori usati più spesso in associazione con le classi contenitore, e gli iteratori sono un concetto fondamentale nel progetto e nell'uso dei contenitori Standard C++, completamente descritti nel Volume 2 di questo libro (scaricabile da *www.BruceEckel.com*). Un iteratore è anche un tipo di *design pattern*, che è il soggetto di un capitolo nel Volume 2.

In molti modi, un iteratore è un "puntatore astuto," e infatti si noterà che gli iteratori usualmente mimano la maggior parte delle operazioni dei puntatori. Diversamente da un puntatore, tuttavia, l'iteratore è progettato per essere sicuro, così si farà con molta meno probabilità l'equivalente del varcare il limite delle dimensioni di un array (o se succede, si accerta più facilmente).

Si consideri il primo esempio in questo capitolo. Qui è mostrato con l'aggiunta di un semplice iteratore:

```
//: C16:IterIntStack.cpp
// Semplice stack di interi con iteratori
//{L} fibonacci
#include "fibonacci.h"
#include "../require.h"
#include <iostream>
using namespace std;

class IntStack {
  enum { ssize = 100 };
```

```
  int stack[ssize];
  int top;
public:
  IntStack() : top(0) {}
  void push(int i) {
    require(top < ssize, "Troppi inserimenti eseguiti chiamando push( )");
    stack[top++] = i;
  }
  int pop() {
    require(top > 0, "Troppe chiamate alla funzione pop()");
    return stack[--top];
  }
  friend class IntStackIter;
};

// Un iteratore è come un puntatore "astuto":
class IntStackIter {
  IntStack& s;
  int index;
public:
  IntStackIter(IntStack& is) : s(is), index(0) {}
  int operator++() { // Prefisso
    require(index < s.top,
      "iteratore mosso fuori dal range");
    return s.stack[++index];
  }
  int operator++(int) { // Postfisso
    require(index < s.top,
      "iteratore mosso fuori dal range");
    return s.stack[index++];
  }
};

int main() {
  IntStack is;
  for(int i = 0; i < 20; i++)
    is.push(fibonacci(i));
  // Attraversamento con un iteratore:
  IntStackIter it(is);
  for(int j = 0; j < 20; j++)
    cout << it++ << endl;
} ///:~
```

L' **IntStackIter** è stato creato per lavorare solo con un **IntStack**. Si noti che **IntStackIter** è un **friend** di **IntStack**, che gli da l'accesso a tutti gli elementi **private** di **IntStack**.

Come un puntatore, il lavoro di **IntStackIter** è di muoversi attraverso un **IntStack** e recuperare valori. In questo semplice esempio, l'**IntStackIter** può muoversi solo in avanti (usando sia la forma pre- che postfissa dell'**operator++**). Tuttavia, non ci sono confini al modo in cui un iteratore possa essere definito, se non quelli imposti dalle restrizioni del contenitore con cui lavora. E' perfettamente accettabile per un iteratore (nei limiti del contenitore sottostante) che si muova in qualsiasi modo nell'attraversare il suo contenitore associato e che causi la modifica dei valori contenui.

E' consueto che un iteratore sia creato con un costruttore che lo unisce ad un singolo oggetto contenitore, e che l'iteratore non sia connesso a diversi contenitori durante il suo tempo di vita. (Gli iteratori sono usualmente piccoli e poco costosi, così si può facilmente farne un altro.)

Con l'iteratore, possiamo attraversare gli elementi dello stack senza recuperarli, proprio come un puntatore può spostarsi lungo gli elementi di un array. Tuttavia, l'iteratore conosce la sottostante struttura dello stack e come attraversare gli elementi, così anche se ci stiamo muovendo attraverso essi all'apparenza "incrementando un puntatore," ciò che succede sotto sotto è più complicato. Questa è la chiave dell'iteratore: sta astraendo il complicato processo del movimento da un elemento del contenitore al successivo in qualcosa che sembra un puntatore. L'obiettivo è avere per *ogni* iteratore nel nostro programma la stessa interfaccia cosicchè qualsiasi codice che usi l'iteratore non si preoccupi di che cosa esso stia puntando – sa solo che può riposizionare tutti gli iteratori nello stesso modo, così non ha alcuna importanza il contenitore al quale l'iteratore punta. In questo modo è possibile scrivere codice più generico. Tutti i contenitori e gli algoritmi nell libreria Standard C++ Library sono basati su questo principio degli iteratori.

Per aiutare a rendere le cose più generiche, sarebbe carino essere in grado di dire "ogni contenitore ha una classe associata chiamata **iteratore**," ma questo tipicamente causerà problemi con il dare i nomi. La soluzione è aggiungere una classe **iterator** annidata a ciascun contenitore (si noti che in questo caso, "**iterator**" inizia con una lettera minuscola cosicchè sia conforme allo stile della libreria Standard C++). Qui c'è **IterIntStack.cpp** con un **iterator** annidato:

```
//: C16:NestedIterator.cpp
// Annidare un iteratore all'interno del contenitore
//{L} fibonacci
#include "fibonacci.h"
#include "../require.h"
#include <iostream>
#include <string>
using namespace std;

class IntStack {
  enum { ssize = 100 };
  int stack[ssize];
  int top;
public:
  IntStack() : top(0) {}
  void push(int i) {
    require(top < ssize, "Troppi inserimenti eseguiti chiamando push( )");
    stack[top++] = i;
  }
  int pop() {
    require(top > 0, "Troppe chiamate alla funzione pop()");
    return stack[--top];
  }
  class iterator;
  friend class iterator;
  class iterator {
    IntStack& s;
    int index;
  public:
    iterator(IntStack& is) : s(is), index(0) {}
    // Per creare l'iteratore "sentinella della fine":
    iterator(IntStack& is, bool)
      : s(is), index(s.top) {}
    int current() const { return s.stack[index]; }
    int operator++() { // Prefisso
      require(index < s.top,
        "iteratore mosso fuori dal range");
      return s.stack[++index];
```

```
    }
    int operator++(int) { // Postfisso
      require(index < s.top,
        "iteratore mosso fuori dal range");
      return s.stack[index++];
    }
    // Salta un iteratore avanti
    iterator& operator+=(int amount) {
      require(index + amount < s.top,
        "IntStack::iterator::operator+=() "
        "provato a muovere fuori dai confini");
      index += amount;
      return *this;
    }
    // Per vedere se sei alla fine:
    bool operator==(const iterator& rv) const {
      return index == rv.index;
    }
    bool operator!=(const iterator& rv) const {
      return index != rv.index;
    }
    friend ostream&
    operator<<(ostream& os, const iterator& it) {
      return os << it.current();
    }
  };
  iterator begin() { return iterator(*this); }
  // Crea la "sentinella della fine":
  iterator end() { return iterator(*this, true);}
};

int main() {
  IntStack is;
  for(int i = 0; i < 20; i++)
    is.push(fibonacci(i));
  cout << "Attraversa l'intero IntStack\n";
  IntStack::iterator it = is.begin();
  while(it != is.end())
    cout << it++ << endl;
  cout << "Attraversa una porzione dell'IntStack\n";
  IntStack::iterator
    start = is.begin(), end = is.begin();
  start += 5, end += 15;
  cout << "start = " << start << endl;
  cout << "end = " << end << endl;
  while(start != end)
    cout << start++ << endl;
} ///:~
```

Quando si crea una classe **friend** annidata, si deve fare attraverso il processo di dichiarare, prima, il nome della classe, quindi dichiararla come una **friend**, infine definire la classe. Altrimenti , il compilatore andrà in confusione.

E' stata aggiunta qualche nuova distorsione all'iteratore. La funzione membro **current( )** produce l'elemento nel contenitore che l'iteratore sta selezionando correntemente. Si può (far) "saltare" un iteratore in avanti di un arbitrario numero di elementi usando **operator+=**. Inoltre, si vedranno due operatori sovraccaricati: == and **!=** che confronteranno un iteratore con un altro. Questi possono confrontare qualsiasi due **IntStack::iterator**s, ma sono progettati soprattutto come test per vedere se l'iteratore è alla fine della sequenza nello stesso modo in cui fanno i "reali" iteratori della libreria

Standard C++. L'idea è che due iteratori definiscono un range, che va dal primo elemento puntato dal primo iteratore incluso fino all'ultimo elemento puntato dal secondo iteratore *non* incluso. Così se ci vogliamo muovere attraverso il range definito dai due iteratori, diciamo qualcosa del genere:

```
  while(start != end)
  cout << start++ << endl;
```

dove **start** e **end** sono i due iteratori nel range. Si noti che l'iteratore **end**, al quale spesso ci riferiamo come la *sentinella della fine*, non sia deferenziato e sia là solo per dire che siamo alla fine della sequenza. Allora rappresenta "uno oltre la fine."

Gran parte del tempo vorremo muoverci attraverso l'intera sequenza in un contenitore, così il contenitore ha bisogno di un qualche modo per produrre gli iteratori che indichino l'inizio della sequenza e la sentinella della fine. Qui, come nella libreria Standard C++, questi iteratori sono prodotti dalle funzioni membro del contenitore **begin( )** e **end( )**. **begin( )** usa il primo costruttore di **iterator** che di default punta all'inizio del contenitore (questo è il primo elemento inserito nello stack). Tuttavia, un secondo costruttore, usato da **end( )**, è necessario per creare l'**iteratore** sentinella della fine. Essere "alla fine" significa puntare alla cima (top) dello stack, poichè **top** indica sempre il prossimo spazio disponibile – ma non usato – sullo stack. Questo costruttore di **iterator** prende un secondo argomento di tipo **bool**, che è una variabile fittizia (che serve solo) per distinguere i due costruttori.

I numeri di Fibonacci sono usati ancora per riempire l'**IntStack** in **main( )**, e gli **iteratori** sono usati per muoversi attraverso l'intero **IntStack** e anche all'interno di un range ristretto della sequenza.

Il prossimo passo, naturalmente, è rendere il codice generico templatizzandolo sul tipo che conserva, cosicchè invece di essere forzati a conservare solo **int**s possiamo conservare qualsiasi tipo:

```
//: C16:IterStackTemplate.h
// Semplice stack template con iteratore annidato
#ifndef ITERSTACKTEMPLATE_H
#define ITERSTACKTEMPLATE_H
#include "../require.h"
#include <iostream>

template<class T, int ssize = 100>
class StackTemplate {
  T stack[ssize];
  int top;
public:
  StackTemplate() : top(0) {}
  void push(const T& i) {
    require(top < ssize, "Troppi inserimenti eseguiti chiamando push( )");
    stack[top++] = i;
  }
  T pop() {
    require(top > 0, "Troppe chiamate alla funzione pop()");
    return stack[--top];
  }
  class iterator; // Dichiarazione richiesta
  friend class iterator; // Lo rende un friend
  class iterator { // Ora lo definiamo
```

```
    StackTemplate& s;
    int index;
  public:
    iterator(StackTemplate& st): s(st),index(0){}
    // Per creare l'iteratore "sentinella della fine":
    iterator(StackTemplate& st, bool)
      : s(st), index(s.top) {}
    T operator*() const { return s.stack[index];}
    T operator++() { // Forma prefissa
      require(index < s.top,
        "iteratore mosso fuori dal range");
      return s.stack[++index];
    }
    T operator++(int) { // Forma postfissa
      require(index < s.top,
        "iteratore mosso fuori dal range");
      return s.stack[index++];
    }
    // Salta un iteratore in avanti
    iterator& operator+=(int amount) {
      require(index + amount < s.top,
        " StackTemplate::iterator::operator+=() "
        "provato a muovere fuori dai confini");
      index += amount;
      return *this;
    }
    // Per vedere se sei alla fine:
    bool operator==(const iterator& rv) const {
      return index == rv.index;
    }
    bool operator!=(const iterator& rv) const {
      return index != rv.index;
    }
    friend std::ostream& operator<<(
      std::ostream& os, const iterator& it) {
      return os << *it;
    }
  };
  iterator begin() { return iterator(*this); }
  // Crea la "sentinella della fine":
  iterator end() { return iterator(*this, true);}
};
#endif // ITERSTACKTEMPLATE_H ///:~
```

Si può vedere che la trasformazione da una classe regolare ad un **template** è ragionevolmente trasparente. Questo approccio di creare e debaggare prima una classe ordinaria, e poi di renderla un template, è generalmente considerato essere più semplice che creare il template da zero.

Si noti che invece di dire solo:

```
friend iterator; // Lo rende un friend
```

Questo codice ha:

```
friend class iterator; // Lo rende un friend
```

Questo è importante poichè il nome “iterator” è già nello scope, da un file incluso.

Invece della funzione membro **current( )**, l' **iterator** ha un **operator*** per selezionare l'elemento corrente, che fa l'**iterator** molto più simile ad un puntatore ed è una pratica comune.

Qui c'è l'esempio modificato per testare il template:

```
//: C16:IterStackTemplateTest.cpp
//{L} fibonacci
#include "fibonacci.h"
#include "IterStackTemplate.h"
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

int main() {
  StackTemplate<int> is;
  for(int i = 0; i < 20; i++)
    is.push(fibonacci(i));
  // Attraversa con un iteratore:
  cout << "Attraversa l'intero StackTemplate\n";
  StackTemplate<int>::iterator it = is.begin();
  while(it != is.end())
    cout << it++ << endl;
  cout << "Attraversa una porzione\n";
  StackTemplate<int>::iterator
    start = is.begin(), end = is.begin();
  start += 5, end += 15;
  cout << "start = " << start << endl;
  cout << "end = " << end << endl;
  while(start != end)
    cout << start++ << endl;
  ifstream in("IterStackTemplateTest.cpp");
  assure(in, "IterStackTemplateTest.cpp");
  string line;
  StackTemplate<string> strings;
  while(getline(in, line))
    strings.push(line);
  StackTemplate<string>::iterator
    sb = strings.begin(), se = strings.end();
  while(sb != se)
    cout << sb++ << endl;
} ///:~
```

Il primo uso dell'iteratore lo fa soltanto marciare dall'inizio alla fine (e mostra che la sentinella della fine lavora correttamente). Nel secondo uso, si può vedere come gli iteratori ci consentano di specificare facilmente un range di elementi (i contenitori e gli iteratori nella libreria Standard C++ usano questo concetto di range quasi ovunque). La funzione sovraccaricata **operator+=** sposta gli iteratori **start** e **end** alle posizioni nella metà del range degli elementi in **is**, e questi elementi sono stampati. Si noti nell'uscita che la sentinella della fine *non* è inclusa nel range, allora può essere uno oltre la fine del range per consentire a noi di sapere che abbiamo superato la fine – ma non si deferenzia la sentinella della fine, altrimenti si può finire per deferenziare un puntatore null. (io ho inserito un guardiano nel **StackTemplate::iterator**, ma nei contenitori e negli iteratori della libreria Standard C++ tale codice non c'è – per ragioni di efficienza – così è necessario prestare attenzione.)

Infine, per verificare che lo **StackTemplate** lavora con oggetti di classe, ne è istanziato uno per **string** e riempito con le linee del file del codice sorgente, che sono quindi stampate in uscita.

Stack con iteratori

Noi possiamo ripetere il processo con la classe **Stack** dinamicamente dimensionata che è stata usata come esempio nel libro. Qui c'è la classe **Stack** con un iteratore annidato mescolato nella miscela:

```
//: C16:TStack2.h
// Stack templatizzato con iteratore annidato
#ifndef TSTACK2_H
#define TSTACK2_H

template<class T> class Stack {
  struct Link {
    T* data;
    Link* next;
    Link(T* dat, Link* nxt)
      : data(dat), next(nxt) {}
  }* head;
public:
  Stack() : head(0) {}
  ~Stack();
  void push(T* dat) {
    head = new Link(dat, head);
  }
  T* peek() const {
    return head ? head->data : 0;
  }
  T* pop();
  // Classe iteratore annidato:
  class iterator; // Dichiarazione richiesta
  friend class iterator; // Lo rende un friend
  class iterator { // Ora lo definiamo
    Stack::Link* p;
  public:
    iterator(const Stack<T>& tl) : p(tl.head) {}
    // Costruttore di copia:
    iterator(const iterator& tl) : p(tl.p) {}
    // L'iteratore sentinella della fine:
    iterator() : p(0) {}
    // operator++ ritorna un valore booleano che indica la fine:
    bool operator++() {
      if(p->next)
        p = p->next;
      else p = 0; // Indica la fine della lista
      return bool(p);
    }
    bool operator++(int) { return operator++(); }
    T* current() const {
      if(!p) return 0;
      return p->data;
    }
    // Operatore di deferenziazione di puntatore:
    T* operator->() const {
      require(p != 0,
        "PStack::iterator::operator->returns 0");
      return current();
```

```
    }
    T* operator*() const { return current(); }
    // conversione di bool per test condizionale:
    operator bool() const { return bool(p); }
    // Confronto per testare la fine:
    bool operator==(const iterator&) const {
      return p == 0;
    }
    bool operator!=(const iterator&) const {
      return p != 0;
    }
  };
  iterator begin() const {
    return iterator(*this);
  }
  iterator end() const { return iterator(); }
};

template<class T> Stack<T>::~Stack() {
  while(head)
    delete pop();
}

template<class T> T* Stack<T>::pop() {
  if(head == 0) return 0;
  T* result = head->data;
  Link* oldHead = head;
  head = head->next;
  delete oldHead;
  return result;
}
#endif // TSTACK2_H ///:~
```

Si noterà anche che la classe è stata cambiata per supporatre l'appartenenza, che ora lavora poichè la classe conosce il tipo esatto (o almeno il tipo base, che lavorerà assumendo che siano usati distruttori virtuali). Di default il contenitore distrugge i suoi oggetti ma noi siamo responsabili per qualsiasi puntatore che inseriamo (tramite **pop( )**).

L'iteratore è semplice, e fisicamente molto piccolo – ha le dimensioni di un singolo puntatore. Quando creiamo un **iteratore**, esso è inizializzato alla testa della lista collegata, e possiamo solo incrementarlo in avanti attraverso la lista. Se vogliamo cominciare sopra all'inizio, creiamo un nuovo iteratore, e se vogliamo ricordare un punto nella lista, creiamo un nuovo iteratore dall'iteratore esistente che punta in quel punto (usando il costruttore di copia dell'iteratore).

Per chiamare funzioni per l'oggetto cui si riferisce l'iteratore, si può usare la funzione **current( )**, l' **operator***, o l'operatore di deferenziazione del puntatore **operator->** (una vista comune negli iteratori). L'ultimo ha un'implementazione che *sembra* identica a quella di **current( )** poichè restituisce un puntatore all'oggetto corrente, ma è diversa perchè l'operatore di deferenziazione di puntatore compie livelli extra di deferenziazione (vedere il Capitolo 12).

La classe **iterator** segue la forma che abbiamo visto nell'esempio precedente. **class iterator** è annidato all'interno della classe contenitore, contiene i costruttori per creare sia un iteratore che punti ad un elemento nel contenitore che un iteratore "sentinella della fine", e la classe contenitore ha i metodi **begin( )** e **end( )** per produrre questi iteratori. (Quando impareremo di più circa la libreria Standard C++ , vedremo che i nomi **iterator**,

begin( ), e **end( )** che sono usati qui sono stati chiaramente elevati a classi contenitore standard. Alla fine di questo capitolo, si vedrà che questo consente a queste classi contenitore di essere usate come se fossero classi contenitore della libreria Standard C++.)

L'intera implementazione è contenuta nell'header file, quindi non c'è un file **cpp** separato. Qui c'è un piccolo test che esercita l'iteratore:

```
//: C16:TStack2Test.cpp
#include "TStack2.h"
#include "../require.h"
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

int main() {
  ifstream file("TStack2Test.cpp");
  assure(file, "TStack2Test.cpp");
  Stack<string> textlines;
  // Legge il file e carica le linee nello Stack:
  string line;
  while(getline(file, line))
    textlines.push(new string(line));
  int i = 0;
  // Usa l'iteratore per stampare linee dalla lista:
  Stack<string>::iterator it = textlines.begin();
  Stack<string>::iterator* it2 = 0;
  while(it != textlines.end()) {
    cout << it->c_str() << endl;
    it++;
    if(++i == 10) // Ricorda la decima linea
      it2 = new Stack<string>::iterator(it);
  }
  cout << (*it2)->c_str() << endl;
  delete it2;
} ///:~
```

Uno **Stack** è istanziato per conservare oggetti **string** e riempito con linee prese da un file. Quindi è creato un iteratore e usato per muoversi attraverso la sequenza. La decima linea è memorizzata da un secondo iteratore generato dal primo dal costruttore di copia; in seguito questa linea è stampata e l'iteratore – creato dinamicamente – è distrutto. Qui, la creazione dinamica di un oggetto è usata per controllare il tempo di vita dell'oggetto.

PStash con iteratori

Per la maggior parte delle classi contenitore ha senso avere un iteratore. Qui c'è un iteratore aggiunto alla classe **PStash** :

```
//: C16:TPStash2.h
// PStash templatizzata con iteratore annidato
#ifndef TPSTASH2_H
#define TPSTASH2_H
#include "../require.h"
#include <cstdlib>

template<class T, int incr = 20>
class PStash {
  int quantity;
```

```
  int next;
  T** storage;
  void inflate(int increase = incr);
public:
  PStash() : quantity(0), storage(0), next(0) {}
  ~PStash();
  int add(T* element);
  T* operator[](int index) const;
  T* remove(int index);
  int count() const { return next; }
  // Classe iteratore annidata:
  class iterator; // Dichiarazione richiesta
  friend class iterator; // Lo rende un friend
  class iterator { // Ora lo definiamo
    PStash& ps;
    int index;
  public:
    iterator(PStash& pStash)
      : ps(pStash), index(0) {}
    // Per creare la sentinella della fine:
    iterator(PStash& pStash, bool)
      : ps(pStash), index(ps.next) {}
    // Costruttore di copia:
    iterator(const iterator& rv)
      : ps(rv.ps), index(rv.index) {}
    iterator& operator=(const iterator& rv) {
      ps = rv.ps;
      index = rv.index;
      return *this;
    }
    iterator& operator++() {
      require(++index <= ps.next,
        "PStash::iterator::operator++ "
        "muove l'indice oltre i confini");
      return *this;
    }
    iterator& operator++(int) {
      return operator++();
    }
    iterator& operator--() {
      require(--index >= 0,
        "PStash::iterator::operator-- "
        "muove l'indice oltre i confini");
      return *this;
    }
    iterator& operator--(int) {
      return operator--();
    }
    // Fa saltare l'iteratore avanti o indietro:
    iterator& operator+=(int amount) {
      require(index + amount < ps.next &&
        index + amount >= 0,
        "PStash::iterator::operator+= "
        "tentativo di indicizzare oltre i limiti");
      index += amount;
      return *this;
    }
    iterator& operator-=(int amount) {
      require(index - amount < ps.next &&
        index - amount >= 0,
        "PStash::iterator::operator-= "
        "tentativo di indicizzare oltre i limiti");
      index -= amount;
```

```
      return *this;
    }
    // Crea un nuovo iteratore che è mosso in avanti
    iterator operator+(int amount) const {
      iterator ret(*this);
      ret += amount; // op+= fa un controllo sui limiti
      return ret;
    }
    T* current() const {
      return ps.storage[index];
    }
    T* operator*() const { return current(); }
    T* operator->() const {
      require(ps.storage[index] != 0,
        "PStash::iterator::operator->returns 0");
      return current();
    }
    // Rimuove l'elemento corrente:
    T* remove(){
      return ps.remove(index);
    }
    // Test di confronto per la fine:
    bool operator==(const iterator& rv) const {
      return index == rv.index;
    }
    bool operator!=(const iterator& rv) const {
      return index != rv.index;
    }
  };
  iterator begin() { return iterator(*this); }
  iterator end() { return iterator(*this, true);}
};

// Distruzione degli oggetti contenuti:
template<class T, int incr>
PStash<T, incr>::~PStash() {
  for(int i = 0; i < next; i++) {
    delete storage[i]; // Puntatori Null OK
    storage[i] = 0; // Solo per essere sicuri
  }
  delete []storage;
}

template<class T, int incr>
int PStash<T, incr>::add(T* element) {
  if(next >= quantity)
    inflate();
  storage[next++] = element;
  return(next - 1); // Indice
}

template<class T, int incr> inline
T* PStash<T, incr>::operator[](int index) const {
  require(index >= 0,
    "PStash::operator[] indice negativo");
  if(index >= next)
    return 0; // Per indicare la fine
  require(storage[index] != 0,
    "PStash::operator[] ha restituito un puntatore nullo");
  return storage[index];
}

template<class T, int incr>
```

```
T* PStash<T, incr>::remove(int index) {
  // operator[] attua controlli di validità:
  T* v = operator[](index);
  // "Rimuove" il puntatore:
  storage[index] = 0;
  return v;
}

template<class T, int incr>
void PStash<T, incr>::inflate(int increase) {
  const int tsz = sizeof(T*);
  T** st = new T*[quantity + increase];
  memset(st, 0, (quantity + increase) * tsz);
  memcpy(st, storage, quantity * tsz);
  quantity += increase;
  delete []storage; // Vecchia memoria
  storage = st; // Punta alla nuova memoria
}
#endif // TPSTASH2_H ///:~
```

La maggior parte di questo file è una traslazione chiaramente semplice di entrambi i precedenti **PStash** e dell'**iterator** annidato in un template. Questa volta, tuttavia, gli operatori restituiscono riferimenti all'iteratore corrente, che è il più tipico e flessibile approccio da prendere.

Il distruttore chiama **delete** per tutti i puntatori contenuti, e poichè il tipo è catturato dal template, avrà luogo la distruzione corretta. Si dovrebbe essere a conoscenza del fatto che se il contenitore conserva puntatori ad un tipo della classe base, questo tipo dovrebbe avere un distruttore **virtuale** per assicurare la cancellazione corretta degli oggetti derivati i cui indirizzi sono stati castati all'insù nel momento in cui sono stati inseriti nel contenitore.

Il **PStash::iterator** segue il modello dell'iteratore di legare l'oggetto per il suo tempo di vita ad un singolo contenitore. In più, il costruttore di copia ci consente di fare un nuovo iteratore che punta alla stessa locazione dell'iteratore esistente da cui è creato, realizzando effettivamente un segnalibro nel contenitore. Le funzioni membro **operator+=** e **operator-=** consentono di muovere un iteratore di un numero di punti, rispettando i confini del contenitore. Gli operatori sovraccaricati di incremento e decremento muovono l'iteratore di una posizione. L'**operator+** produce un nuovo iteratore che è mosso in avanti dell'ammontare dell'addendo. Come nel precedente esempio, gli operatori di deferenzazione del puntatore sono usati per operare sull'elemento al quale si riferisce l'iteratore, e **remove( )** distrugge l'oggetto corrente chiamando il **remove( )** del contenitore.

Lo stesso tipo di codice visto sopra (*a la* i contenitori della Libreria Standard C++) è usato per creare la sentinella della fine: un secondo costruttore, la funzione membro **end( )** del contenitore e gli operatori **operator==** e **operator!=** per il confronto.

Il seguente esempio crea e testa due diversi tipi di oggetti **Stash**, uno per una nuova classe chiamata **Int** che annuncia la sua costruzione e la sua distruzione e una che conserva oggetti della classe **string** della Libreria Standard.

```
//: C16:TPStash2Test.cpp
#include "TPStash2.h"
#include "../require.h"
#include <iostream>
```

```
#include <vector>
#include <string>
using namespace std;

class Int {
  int i;
public:
  Int(int ii = 0) : i(ii) {
    cout << ">" << i << ' ';
  }
  ~Int() { cout << "~" << i << ' '; }
  operator int() const { return i; }
  friend ostream&
    operator<<(ostream& os, const Int& x) {
      return os << "Int: " << x.i;
  }
  friend ostream&
    operator<<(ostream& os, const Int* x) {
      return os << "Int: " << x->i;
  }
};

int main() {
  { // Per forzare la chiamata del distuttore
    PStash<Int> ints;
    for(int i = 0; i < 30; i++)
      ints.add(new Int(i));
    cout << endl;
    PStash<Int>::iterator it = ints.begin();
    it += 5;
    PStash<Int>::iterator it2 = it + 10;
    for(; it != it2; it++)
      delete it.remove(); // Rimozione di default
    cout << endl;
    for(it = ints.begin();it != ints.end();it++)
      if(*it) // Remove() causa "buchi"
        cout << *it << endl;
  } // qui è chiamato il distruttore di "ints"
  cout << "\n--------------------\n";
  ifstream in("TPStash2Test.cpp");
  assure(in, "TPStash2Test.cpp");
  // Istanzazione per String:
  PStash<string> strings;
  string line;
  while(getline(in, line))
    strings.add(new string(line));
  PStash<string>::iterator sit = strings.begin();
  for(; sit != strings.end(); sit++)
    cout << **sit << endl;
  sit = strings.begin();
  int n = 26;
  sit += n;
  for(; sit != strings.end(); sit++)
    cout << n++ << ": " << **sit << endl;
} ///:~
```

Per convenienza, **Int** ha un **ostream operator<<** associato sia per un **Int&** che per un **Int***.

Il primo blocco di codice in **main( )** è circondato dalle parentesi graffe per forzare la distruzione del **PStash<Int>** e quindi la pulizia automatica operata da quel distruttore.

Un range di elementi è rimosso e cancellato a mano per mostrare che **PStash** cancella il resto.

Per entrambe le istanze di **PStash**, viene creato un iteratore ed usato per muoversi nel contenitore. Si noti l'eleganza ottenuta usando questi costrutti; non ci si affligga con i dettagli dell'implementazione dell'uso di un array. Si dice agli oggetti contenitore e iteratore *cosa* fare, non come. Questo rende la soluzione più facile da concettualizzare, da realizzare, e da modificare.

Perchè gli iteratori?

Finora abbiamo visto i meccanismi degli iteratori, ma capire perchè siano così importanti richiede un esempio più complesso.

E' comune vedere il polimorfismo, la creazione di oggetti dinamici, e i contenitori usati tutti insieme in un vero programma orientato agli oggetti. I contenitori e la creazione di oggetti dinamici risolvono il problema della mancata conoscenza di quanti o di che tipo di oggetti avremo bisogno. E se il contenitore è configurato per conservare puntatori a oggetti di una classe base, avviene un cast all'insù ogni volta che si inserisce un puntatore ad una clase derivata nel contenitore (con i benefici associati di organizzazione e di estensibilità del codice). Come codice finale nel Volume 1 di questo libro, questo esempio forzerà anche insieme diversi aspetti di tutto ciò che abbiamo imparato finora – si si riesce a seguire questo esempio, allora si è pronti per il Volume 2.

Supponiamo di stare creando un programma che consenta all'utente di editare e di produrre diversi tipi di disegni. Ciascun disegno è un oggetto che contiene una collezione di oggetti **Shape**:

```
//: C16:Shape.h
#ifndef SHAPE_H
#define SHAPE_H
#include <iostream>
#include <string>

class Shape {
public:
  virtual void draw() = 0;
  virtual void erase() = 0;
  virtual ~Shape() {}
};

class Circle : public Shape {
public:
  Circle() {}
  ~Circle() { std::cout << "Circle::~Circle\n"; }
  void draw() { std::cout << "Circle::draw\n";}
  void erase() { std::cout << "Circle::erase\n";}
};

class Square : public Shape {
public:
  Square() {}
  ~Square() { std::cout << "Square::~Square\n"; }
  void draw() { std::cout << "Square::draw\n";}
  void erase() { std::cout << "Square::erase\n";}
};
```

```
class Line : public Shape {
public:
  Line() {}
  ~Line() { std::cout << "Line::~Line\n"; }
  void draw() { std::cout << "Line::draw\n";}
  void erase() { std::cout << "Line::erase\n";}
};
#endif // SHAPE_H ///:~
```

Questo usa la classica struttura di funzioni virtuali nella classe base che sono riscritte nella classe derivata. Si noti che la classe **Shape** include un distruttore **virtuale** , qualcosa che dovremmo automticamente aggiungere ad ogni classe con funzioni **virtuali**. Se un contenitore conserva puntatori o riferimenti ad oggetti **Shape**, allora quando sono chiamati i distruttori **virtuali** per questi oggetti ogni cosa sarà cancellata correttamente.

Ciascun diverso tipo di disegno nel seguente esempio fa uso di un diverso tipo di classe contenitore teplatizzata: la **PStash** e la **Stack** che sono state definite in questo capitolo, e la classe **vector** dalla Libreira C++. L'"uso"' dei contenitori è estremamente semplice, ed in generale l'ereditarietà potrebbe non essere l'approccio migliore (la composizione potrebbe avere più senso), ma in questo caso l'ereditarietà è un approccio semplice e non riduce il valore del punto fatto nell'esempio.

```
//: C16:Drawing.cpp
#include <vector> // Usa anche vettori Standard!
#include "TPStash2.h"
#include "TStack2.h"
#include "Shape.h"
using namespace std;

// Un disegno (Drawing) è innanzitutto un contenitore di figure (Shapes):
class Drawing : public PStash<Shape> {
public:
  ~Drawing() { cout << "~Drawing" << endl; }
};

// Un piano (Plan) è un contenitore diverso di figure:
class Plan : public Stack<Shape> {
public:
  ~Plan() { cout << "~Plan" << endl; }
};

// Uno Schema (Schematic) è un contenitore diverso di figure:
class Schematic : public vector<Shape*> {
public:
  ~Schematic() { cout << "~Schematic" << endl; }
};

// Una funzione template:
template<class Iter>
void drawAll(Iter start, Iter end) {
  while(start != end) {
    (*start)->draw();
    start++;
  }
}

int main() {
  // Ciascun tipo di contenitore ha
  // una differente interfaccia:
```

```
  Drawing d;
  d.add(new Circle);
  d.add(new Square);
  d.add(new Line);
  Plan p;
  p.push(new Line);
  p.push(new Square);
  p.push(new Circle);
  Schematic s;
  s.push_back(new Square);
  s.push_back(new Circle);
  s.push_back(new Line);
  Shape* sarray[] = {
    new Circle, new Square, new Line
  };
  // Gli iteratori e la funzione template
  // consentono loro di essere trattati genericamente:
  cout << "Drawing d:" << endl;
  drawAll(d.begin(), d.end());
  cout << "Plan p:" << endl;
  drawAll(p.begin(), p.end());
  cout << "Schematic s:" << endl;
  drawAll(s.begin(), s.end());
  cout << "Array sarray:" << endl;
  // Lavora anche con i puntatori di array:
  drawAll(sarray,
    sarray + sizeof(sarray)/sizeof(*sarray));
  cout << "Fine di main" << endl;
} ///:~
```

I diversi tipi di contenitori conservano tutti puntatori a **Shape** e puntatori a oggetti castati all'insù di classi derivate da **Shape**. Tuttavia, a causa del polimorfismo, avrà luogo l'appropriato comportamento anche quando sono chiamate le funzioni virtuali.

Si noti che **sarray**, l'array di **Shape***, può anche essere pensato come un contenitore.

Funzioni template

In **drawAll( )** si vede qualcosa di nuovo. Fino ad ora in questo capitolo abbiamo usato solo *classi template*, che istanziano nuove classi basate su uno o più parametri di tipo. Tuttavia, si può altrettanto facilmente creare *funzioni template*, che creano nuove funzioni basate su parametri di tipo. La ragione per cui si crea una funzione template è la stessa per cui si usa una classe template: si sta provando a creare un codice generico, e si fa questo ritardando la specifica di uno o più tipi. Abbiamo bisogno di dire soltanto che questi parametri di tipo supportano certe operazioni, non esattamente che tipi siano.

La funzione template **drawAll( )** può essere pensata come un *algoritmo* (e questo è il modo in cui sono chiamate la maggior parte delle funzioni template nella libreria Standard C++). Questo dice solo come fare qualcosa dati gli iteratori che descrivono un range di elementi, a condizione che questi iteratori possano essere deferenziati, incrementati e confrontati. Questi sono esattamente il tipo di iteratori che abbiamo sviluppato in questo capitolo, ed anche – non è una coincidenza – il tipo di iteratori che sono prodotti dai contenitori nella Libreria Standard C++, evidenziati dall'uso di **vector** in questo esempio.

Ci piacerebbe anche che **drawAll( )** sia un *algoritmo generico*, cosicchè i contenitori possano essere di qualsiasi tipo e noi non dobbiamo scrivere una nuova versione

dell'algoritmo per ogni tipo diverso di contenitore. Ecco dove le funzioni template sono essenziali, poichè esse generano automaticamente il codice specifico per ciascun diverso tipo di contenitore. Ma senza l'extra apportato dagli iteratori alla proprietà di essere pianificato non direttamente per qualcosa, questa genericità non sarebbe possibile. Questo è il motivo per cui gli iteratori sono importanti; ci consentono di scrivere un codice general-purpose (progettato o fruibile per più di un solo uso) che includa contenitori senza conoscere la struttura sottostante del contenitore. (Si noti che, in C++, gli iteratori e gli algoritmi generici richiedono le funzioni template per funzionare.)

Si può vedere la dimostrazione di questo in **main( )**, poichè **drawAll( )** lavora senza cambiamenti con ciascun diverso tipo di contenitore. E ancor più interessante, **drawAll( )** lavora anche con i puntatori all'inizio e alla fine dell'array **sarray**. Questa abilità di trattare gli array come contenitori è essenziale per il progetto della Libreria Standard C++, i cui algoritmi assomigliano molto a **drawAll( )**.

Poichè le classi contenitore template sono raramente soggette all'ereditarietà e al casting all'insù che vediamo con le classi "ordinarie", non vedremo quasi mai funzioni **virtuali** nelle classi contenitore. Il riuso delle classi contenitore è implementato con i template, non con l'ereditarietà.

Sommario

Le classi conteniore sono una parte essenziale della programmazione orientata agli oggetti. Esse sono un altro modo di semplificare e nascondere i dettagli di un programma e di velocizzare il processo di sviluppo del programma. Inoltre, forniscono un ampio apporto di sicurezza e flessibilità sostituendo i primitivi array e le tecniche relativamente grezze della struttura di dati scoperti in C.

Poichè il programmatore cliente ha bisogno di contenitori, è essenziale che essi siano facili da usare. Qui ci venono in aiuto i **template**. Con i template la sintassi per il riuso del codice sorgente (opposto al riuso del codice oggetto fornito dall'ereditarietà e dalla composizione) diventa abbastanza banale (pure) per l'utente inesperto. Infatti, riusare il codice con i template è notevolmente più facile che con l'ereditarietà e la composizione.

Nonostante si sia imparato a creare classi contenitore e iteratore in questo libro, in pratica è molto più conveniente imparare contenitori e iteratori nella Libreria Standard C++ , poichè ci si può aspettare che essi siamo disponibili con ogni compilatore. Come si vedrà nel Volume 2 di questo libro (scaricabile da *www.BruceEckel.com*), i contenitori e gli algoritmi nella Libreria Standard C++ soddisferanno virtualmente sempre le nostre necessità cos non dovremo crearne di nuovi.

Le questioni connesse alla progettazione di una classe contenitore sono state appena accennate in questo capitolo, ma il lettore può arguire che possono andare ben oltre. Una libreria di classi contenitore complicata può coprire tutti i tipi di questioni aggiuntive, incluso il multithreading, la persistenza e il garbage collection (letteralmente "accumulo di immondizia", riguarda dati non corretti, effetti collaterali, perdita di informazioni e simili, *ndt*).

[59] Con l'eccezione, in Java, dei tipi di dato primitivi. Questi furono resi non-**Object**s in nome dell'efficienza.

[60] La libreria OOPS, di Keith Gorlen mentre era al NIH.

[61] *The C++ Programming Language* di Bjarne Stroustrup (I edizione, Addison-Wesley, 1986).

[62] L'ispirazione dei template sembra essere generico ADA .

[63] Tutti i metodi sia in Smalltalk che in Python sono debolmente tipizzati, e così questi linguaggi non hanno bisogno di un meccanismo di template. In effetti, si ottengono i template senza template.

A: Stile di codifica

Questa appendice non tratta dell'indentazione e posizionamento di parentesi tonde e graffe, sebbene ciò sarà menzionato.E' dedicato alle linee guida generali usate in questo libro per l'organizzazione dei listati del codice.

Sebbene molti di questi problemi sono stati introdotti in tutto il libro, questa appendice compare alla fine del libro in modo che si possa presumere che ogni argomento è aperto a critiche e se qualcosa non è chiaro ci si può rimandare alla sezione appropriata.

Tutte le decisioni riguardanti lo stile di codifica in questo libro sono state deliberatamente considerate e prese, qualche volta lungo un periodo di anni. Naturalmente, ognuno ha le proprie ragioni per organizzare il codice nel modo che vuole ed io proverò solo a dire come sono arrivato al mio considerando i vincoli e i fattori ambientali che mi hanno guidato a queste decisioni.

Nozioni generali

Nel testo di questo libro, gli identificatori (funzioni, variabili e nomi di classi) sono scritte in **grassetto**. Molte parole chiavi saranno scritte in grassetto, eccetto per quelle usate moltissimo per cui il grassetto può diventare tedioso, come "class" e "virtual".

Uso un particolare stile di codifica per gli esempi in questo libro. Questo è' stato sviluppato su un certo numero di anni ed è stato parzialmente ispirato dallo stile di Bjarne Stroustrup nel suo originale *The C++ Programming Language*.[64] Lo stile di formattazione è un buon tema per ore di dibattito, per cui dirò solo che non sto tentando di dettare il corretto stile con i miei esempi; ho le mie motivazioni per usare lo stile che uso. Poiché il C++ è un linguaggio di programmazione a stile libero, si può continuare ad usare qualsiasi stile con cui ci si trova comodi.

Detto questo, noterò che è importante avere uno stile di formattazione coerente all'interno di un progetto. Cercando in Internet, si troveranno un certo numero di tools che si possono usare per riformattare tutto il codice in un progetto per ottenere questa preziosa coerenza.

I programmi in questo libro sono file che vengono automaticamente estratti dal testo del libro, il che permette ad essi di essere testati per assicurarsi che funzionino correttamente. Quindi il codice stampato nel libro funzionerà senza errori di compilazione quando è compilato con un'implementazione conforme allo Standard C++ (da notare che non tutti i compilatori supportano tutte le caratteristiche del linguaggio). Gli errori che *dovrebbero* causare problemi di compilazione sono descritti con il commento **//!** per cui possono essere facilmente scoperti e testati usando gli strumenti automatici. Errori scoperti e segnalati all'autore compariranno prima nella versione elettronica del libro (su *www.BruceEckel.com*) ed in seguito aggiornati sul libro.

Uno degli standard in questo libro è che tutti i programmi saranno compilati e linkati senza errori (sebbene qualche volta causeranno dei warning). A questo fine, qualcuno dei programmi, che dimostra solo un esempio di codifica e non rappresenta un programma a se stante, avrà una funzione **main( )** vuota, come questo

```
int main() {}
```

Questo permette al linker di finire senza errori.

Lo standard per **main( )** è di restituire un **int**, ma lo Standard C++ stabilisce che se non c'è un istruzione di **return** all'interno di **main( )**, il compilatore genererà automaticamente il codice **return 0**. Questa opzione (niente **return** nel **main( )**) sarà usata in questo libro (alcuni compilatori potranno ancora generare dei warning per ciò, ma questi non sono compatibili con lo Standard C++).

Nomi di file

In C, è tradizione chiamare gli header file (contenenti le dichiarazioni) con l'estensione **.h** e i file di esecuzione (che fanno sì che la memoria sia allocata e il codice generato) con l'estensione **.c**. Il C++ è passato attraverso un evoluzione. Fu prima sviluppato su Unix, dove il sistema operativo distingueva tra minuscole e maiuscole nei nomi di file. I nomi di file originali erano semplicemente la versione maiuscola dell'estensione C: **.H** e **.C**. Questo naturalmente non funziona per sistemi operativi che non distinguono minuscole e maiuscole, come il DOS. I venditori C++ sotto DOS usano estensioni come **hxx** e **cxx** per i file header e esecutivi, rispettivamente, **hpp** e **cpp**. Più tardi, qualcuno riuscì a capire che l'unica ragione per cui bisognava avere differenti estensioni per un file era perchè il compilatore potesse determinare se compilarlo come file C o C++. Poichè il compilatore non compilava mai gli header file direttamente, solo l'estensione dei file esecutivi doveva essere cambiata. La consuetudine, attraversando virtualmente tutti i sistemi, ha portato ad usare ora **cpp** per i file esecutivi e **h** per gli header file. Da notare che quando includiamo i file header standard di C++, si usa l'opzione di non avere estensione per i file, ad esempio: **#include <iostream>**.

Tag di inizio e fine commento

Un problema molto importante di questo libro è che tutto il codice che si vede in esso deve essere verificato per essere corretto (con almeno un compilatore). Questo è realizzato estraendo automaticamente il file dal libro. Per facilitare ciò, tutti i listati che sono proposti per essere compilati (al contrario dei frammenti di codice, dei quali ci sono solo alcuni) hanno tag di commento all'inizio e alla fine. Questi tag sono usati dal tool di estrazione del codice **ExtractCode.cpp** nel Volume 2 di questo libro (che si può trovare sul sito *www.BruceEckel.com*) per estrarre ogni listato dalla versione ordinaria di testo ASCII del libro.

Il tag di fine listato dice ad **ExtractCode.cpp** che è alla fine del listato, ma il tag di inizio listato è seguito da informazioni quali la sotto-directory a cui il file appartiene (generalmente organizzate in capitoli, così un file appartenente al Capitolo 8 avrà un tag tipo **C08**), seguito da due punti ed il nome del listato.

Poichè **ExtractCode.cpp** crea anche un **makefile** per ogni sotto-directory, vengono anche incorporate nel listato informazioni su come un programma è fatto e la linea di comando usata per testarlo. Se un programma è a se stante (non necessita di essere linkato con nessun altro cioè) non ha informazioni extra. Questo vale anche per gli header file. Comunque, se non contiene un **main( )** e si intende che deve essere linkato con qualcos'altro, allora esso ha uno **{O}** dopo il nome del file. Se per il listato si intende che deve essere il programma principale ma necessita di essere linkato con altre componenti,

c'è una linea separata che inizia con **//{L}** e continua con tutti i file che necessitano di essere linkati (senza estensioni, dal momento che questi possono variare da piattaforma a piattaforma).

Si possono trovare esempi in tutto il libro.

Se un file deve essere estratto ma i tag di inizio e fine non devono essere inclusi nel file che ne deriva (per esempio se esso è un file di dati di test) allora il tag di inizio è immediatamente seguito da un '**!**'.

Parentesi, graffe e indentazione

Si può notare che lo stile di formattazione in questo libro è differente da molti stili tradizionali di C. Naturalmente, ognuno pensa che il proprio stile sia il più razionale. Comunque, lo stile qui usato ha dietro una logica semplice, che sarà presentata qui insieme alle idee sul perchè sono stati sviluppati altri stili.

Lo stile di formattazione è motivato da una cosa: la presentazione, sia nella forma stampata che per i seminari. Si può avere la sensazione che i propri bisogni siano differenti perchè non si fanno molte presentazioni. Comunque, il codice operativo viene letto più volte di quanto è scritto e quindi deve essere facile per il lettore comprenderlo. I miei due importanti criteri sono la "scansione" (come è facile per il lettore capire il significato di una singola linea) e il numero di linee che può adattarsi su una pagina. Quest'ultimo può suonare strano, ma quando si sta tenendo una presentazione in pubblico, si può far distrarre molto la platea se il presentatore deve spostarsi avanti e indietro tra le slide e ciò può essere causato da poche linee sparse.

Tutti sembrano concordare che il codice tra parentesi graffe dovrebbe essere indentato. Ciò su cui la gente non concorda - ed è il punto dove c'è maggior inconsistenza tra gli stili di formattazione - è questo: Dove vanno messe le parentesi aperte? Questa domanda, penso, è quella che causa tante variazioni tra stili di codifica (per un enumerazione di stili di codifica, vedi *C++ Programming Guidelines*, di Tom Plum e Dan Saks, Plum Hall 1991). Proverò a convincerti che molti degli stili odierni derivino dai vincoli del pre-Standard C (prima dei prototipi di funzione) e ora sono quindi inappropriati.

Prima, la mia risposta a quella domanda chiave: le parentesi graffe aperte dovrebbero sempre andare sulla stessa linea come il "precursore" (per questo intendo " se il corpo è del tipo: una classe, funzione, definizioni di oggetti, istruzioni if, etc.."). Questa è una semplice, coerente regola che applico a tutto il codice che scrivo e rende la formattazione molto più semplice. Rende la "scansione" più semplice quando si guarda questa linea:

```
int func(int a);
```

si sa, dal punto e virgola alla fine della linea, che questa è una dichiarazione e non ha seguito, ma quando si vede la linea:

```
int func(int a) {
```

immediatamente si sa che è una definizione perchè la linea finisce con una parentesi aperta e non il punto e virgola. Usando questo approccio, non c'è differenza su dove mettere le parentesi di apertura per una definizione su più linee:

```
int func(int a) {
  int b = a + 1;
  return b * 2;
}
```

e per una definizione su linea singola spesso usata per l'inline:

```
int func(int a) { return (a + 1) * 2; }
```

In modo simile per una classe:

```
class Thing;
```

è una dichiarazione di un nome di classe e

```
class Thing {
```

è una definizione di classe. Si può dire guardando la linea singola in tutti i casi se è una dichiarazione o una definizione. E naturalmente mettendo le parentesi aperte sulla stessa linea, invece che su una linea a parte, è possibile inserire più linee nella stessa pagina.

E quindi perchè abbiamo tanti altri stili? In particolare, si noterà che molte persone creano classi seguendo lo stile di sopra (il quale è usato da Stroustrup in tutte le edizioni del suo libro *The C++ Programming Language* ed. Addison-Wesley) ma creano definizioni di funzioni mettendo la parentesi aperta su una linea a sé (il che genera molti differenti stili di indentazione). Stroustrup fa questo eccetto che per brevi funzioni inline. Con l'approccio che descrivo qui, tutto è coerente - si può dire di cosa si tratta (**class**, funzione, **enum**, etc.) e sulla stessa linea si mette la parentesi aperta per indicare che il corpo per questa cosa è scritto a seguire. Inoltre, la parentesi aperta è la stessa sia per brevi funzioni inline che per le definizioni di funzioni ordinarie.

Sostengo che lo stile delle definizioni di funzioni usato da molte persone, derivi dal prototipo C pre-funzioni, nel quale non si dovevano dichiarare gli argomenti all'interno delle parentesi tonde, ma invece tra la parentesi tonda chiusa e la parentesi graffa aperte (questo mostra le radici assembly del linguaggio C):

```
void bar()
 int x;
 float y;
{
 /* il corpo qui */
}
```

Qui, sarebbe completamente inopportuno mettere le parentesi graffe aperte sulla stessa linea, per cui nessuno lo fece. Comunque, si presero diverse decisioni in merito alla questione se le graffe dovrebbero essere indentate con il corpo o se esse dovrebbero essere al livello del "precursore". Per questo abbiamo differenti stili di formattazione.

Ci sono altri argomenti sul posizionamento delle graffe sulla linea immediatamente seguente la dichiarazione (di una classe, struct, funzione, etc.). La seguente viene da un lettore, ed è presentata in modo che si possa conoscere quale siano i problemi:

Utenti esperti di '*vi*' sanno che schiacciando il tasto ']' due volte porterà alla prossima occorrenza di "{" (o ^L) in colonna 0. Questa caratteristica è estremamente utile per

muoversi nel codice (per saltare alla prossima funzione o definizione di classe). [Il mio commento: quando stavo inizialmente lavorando sotto Unix, apparvero le GNU Emac e rimasi intrappolato tra esse. Come conseguenza, non ho mai raggiunto una piena conoscenza di vi e quindi non penso in termini di "locazione di colonna 0". Comunque, c'è un ampio gruppo di utenti di *vi* , ed essi sono colpiti da tale questione.]

Mettendo la "{" sulla linea successiva, si eliminano un po di confusione dal codice con istruzioni condizionali complesse, favorendo la "scansione". Esempio:

```
if(cond1
   && cond2
   && cond3) {
   istruzione;
}
```

Il codice precedente [afferma il lettore] ha una scansione limitata. Comunque,

```
if (cond1
&& cond2
&& cond3)
{
istruzione;
}
```

spezza l'*if* dal corpo, ma ha la conseguenza di una migliore leggibilità. [le nostre opinioni sul fatto che ciò sia vero varieranno in dipendenza da cosa siamo abituati a fare.]

Alla fine, è più facile allineare visualmente le graffe quando esse sono allineate nella stessa colonna. Esse visivamente "emergono" meglio. [fine del commento del lettore.]

Il problema di dove mettere le parentesi graffe aperte è probabilmente il più discordante. Ho imparato a scandire entrambe le forme e alla fine ne viene fuori quella con cui ci si trova più comodi. Comunque, noto che lo standard ufficiale Java (trovato sul sito Java della Sun) è effettivamente lo stesso di quello che presento qui - da quando molte persone stanno iniziando a programmare in entrambi i linguaggi, la coerenza tra stili di codifica può essere utile.

L'approccio che uso rimuove tutte le eccezioni e casi speciali, e logicamente fornisce uno stile singolo di indentazione. Perfino all'interno del corpo di una funzione, la coerenza rimane, come in:

```
for(int i = 0; i < 100; i++) {
  cout << i << endl;
  cout << x * i << endl;
}
```

Lo stile è semplice da insegnare e ricordare - si usa una singola, coerente regola per tutte le nostre formattazioni del codice, non una per le classi, due per le funzioni (una per quelle inline e un'altra quelle non), e possibilmente altre per cicli **for**, istruzioni **if**, ecc. La sola coerenza, penso, è meritevole di considerazione. Soprattutto, C++ è un linguaggio più nuovo che C, e sebbene dobbiamo fare molte concessioni al C, non dovremmo portare anche troppi manufatti con noi che causano problemi in futuro. Piccoli problemi moltiplicati per molte linee di codice diventano grandi problemi. Per un completo esame

della materia, sebbene per il C, vedi *C Style: Standards and Guidelines*, di David Straker (Prentice-Hall 1992).

L'altro vincolo sotto il quale ho lavorato è la larghezza della linea, infatti il libro ha una limitazione a 50 caratteri. Cosa succede quando qualcosa è più lungo di una linea? Bene, di nuovo mi sono sforzato per avere una politica coerente per il modo in cui spezzare le linee, così che esse possano facilmente essere identificate. Finché qualcosa è parte di una singola definizione, lista di definizione, etc. la linea di continuazione dovrebbe essere indentata di un livello dentro dall'inizio della definizione, lista di argomenti, ecc.

Nomi degli identificatori

Coloro che hanno familiarità con Java noteranno che ho cambiato nell'usare lo stile standard di Java per tutti i nomi degli identificatori. Comunque, non posso essere completamente coerente qui perchè gli identificatori nelle librerie standard C e C++ non seguono questo stile.

Lo stile è abbastanza lineare. Solo la prima lettera di un identificatore è maiuscola se tale identificatore è una classe. Se si tratta di una funzione o variabile allora la prima lettera è minuscola. Il resto degli identificatori consistono di un o più parole, legate insieme ma distinte mettendo la maiuscola per ogni inizio parola. Così una classe appare in questo modo:

```
class FrenchVanilla : public IceCream {
```

Un identificatore di un oggetto appare così:

```
FrenchVanilla myIceCreamCone(3);
```

e una funzione invece:

```
void eatIceCreamCone();
```

(sia per una funzione membro che per una regolare funzione).

La sola eccezione riguarda le costanti compile-time (**const** o **#define**), per le quali tutte le lettere nell'identificatore sono maiuscole.

Il pregio dello stile è che scrivere con le maiuscole ha un suo significato - si può vedere dalla prima lettera se si sta parlando di una classe o di un oggetto/metodo. Ciò è particolarmente utile nell'accesso a classi membro di tipo **static**.

Ordine di inclusione dei file header

Gli header file sono inclusi in ordine "dal più specifico al più generale". Cioè ogni file header nella directory locale viene incluso per primo, poi ogni header del mio "tool" come **require.h**, successivamente vengono inclusi i file header di librerie di terzi, poi gli header della libreria Standard C++ e finalmente i file header della libreria C.

La giustificazione per questo viene da John Lakos nel libro *Large-Scale C++ Software Design* (Addison-Wesley, 1996):

Errori d'uso nascosti possono essere evitati assicurando che i file .h di una componente facciano l'analisi di se stessi - senza dichiarazioni o definizioni fornite dall'esterno.... Includendo i file .h nella prima linea del file .c ci si assicura che nessuno pezzo critico di informazione intrinseca all'interfaccia fisica del componente manchi dal file .h (o, se c'è, ciò che si scoprirà su di esso non appena si proverà a compilare il file .c).

Se l'ordine di inclusione dei file header va "dal più specifico al più generale" allora è più verosimile che il proprio header non si analizzi da se, lo si scoprirà ben presto prevenendo fastidi lungo la strada.

Include guard nei file header

Gli *include guard* sono sempre usati all'interno dei file header per prevenire inclusioni multiple di un file header durante la compilazione di un singolo file **.cpp**. Le include guard sono implementate usando la direttiva del preprocessore **#define** controllando per vedere che un nome non sia già definito. Il nome usato per guard è basato sul nome del file header, con tutte le lettere del nome del file in maiuscolo e rimpiazzando il "." con un trattino di sottolineatura. Per esempio:

```
// IncludeGuard.h
#ifndef INCLUDEGUARD_H
#define INCLUDEGUARD_H
// Il corpo dell'header qui...
#endif // INCLUDEGUARD_H
```

L'indentificatore sull'ultima linea è incluso per chiarezza. Sebbene alcuni preprocessori ignorino ogni carattere dopo un **#endif**, ciò non è un comportamento standard per questo l'identificatore è commentato.

Uso dello spazio nei nomi

Nei file header, ogni violazione della spaziatura nei nomi nel quale è inclusa l'header deve essere scrupolosamente evitata. Cioè se si cambia la spaziatura fuori da una funzione o classe, si avrà che quel cambiamento si ritroverà per ogni file che include il nostro header, portando ad una serie di problemi. Nessuno dichiarazione **using** di qualsiasi tipo è permessa fuori dalla definizione di funzione, e nessuna direttiva **using** è permessa nei file header.

Nei file **cpp**, ogni direttiva **using** globale riguarderà solo quel file, per cui in questo libro esse sono generalmente usate per produrre codice più leggibile, specialmente in piccoli programmi.

Uso di require( ) e assure( )

Le funzioni **require( )** e **assure( )** definite in **require.h** sono usate in modo consistente in gran parte del libro, per cui potrebbero riportare errori. Se si ha familiarità con i concetti di *precondizioni* e *postcondizioni* (introdotti da Bertrand Meyer) si riconoscerà che l'uso di **require( )** e **assure( )** più o meno fornisce precondizioni (tipicamente) e postcondizioni (occasionalmente). Quindi all'inizio di una funzione, prima che parte del "nucleo" della funzione sia eseguito, vengono verificate le precondizioni per assicurarsi che ogni cosa sia giusta e che tutte le condizioni necessarie siano corrette. Poi il "nucleo" della funzione

viene eseguito, e qualche volta vengono verificate alcune postcondizioni per essere sicuri che il nuovo stato dei dati rientri tra i parametri definiti. Si noterà che le verifiche delle postcondizioni sono are in questo libro, e **assure( )** è principalmente usata per sincerarsi che i file siano aperti con successo.

---

[64] Ibid.

B: Linee guida di programmazione

Quest'appendice è un insieme di suggerimenti per la programmazione in C++. Sono stati messi insieme nel corso della mia esperienza di insegnamento e

di programmazione, nonché dai consigli di amici come Dan Saks (coautore con Tom Plum di *C++ Programming Guidelines*, Plum Hall, 1991), Scott Meyers (autore di *Effective C++*, 2nd editioN, Addison-Wesley, 1998), e Rob Murray (autore di *C++ Strategies & Tactics*, Addison-Wesley, 1993). Inoltre, molti suggerimenti sono direttamente tratti dalle pagine di *Thinking in C++*.

1. Prima fatelo funzionare, poi rendetelo veloce. Questo è vero anche se siete certi che un frammento di codice sia realmente importante e che diverrà il collo di bottiglia principale del vostro sistema. Non fatelo. Prima di tutto cercate di ottenere un sistema funzionante con un progetto il più semplice possibile. Soltanto dopo, se non è abbastanza veloce, analizzatelo. Quasi sempre scoprirete che il vero problema non è il "vostro" collo di bottiglia. Risparmiate tempo per le cose veramente utili.
2. L'eleganza paga sempre. Non è un'attività frivola. Avrete in mano un programma non solo più facile da compilare e testare, ma anche da comprendere e da mantenere: ed è qui che si ritrova il valore economico. Potrebbe servire un po' di esperienza prima di credere a questo fatto, in quanto potrebbe sembrare che, mentre cercate di rendere elegante un frammento di codice, non siate produttivi. La produttività emergerà quando il codice si integrerà perfettamente nel vostro sistema, ed ancor più quando il codice o il sistema verrà modificato.
3. Ricordatevi del principio "divide et impera". Se il problema che state affrontando è troppo complesso, cercate di immaginare quali potrebbero essere le operazioni basilari del programma, se esistesse un essere soprannaturale che si occupasse delle parti più difficili. Quest'essere soprannaturale è un oggetto - scrivete il codice che utilizza quell'oggetto, quindi studiate l'oggetto ed incapsulate le *sue* parti complicate all'interno di altri oggetti, e così via.
4. Non riscrivete automaticamente in C++tutto il vostro codice C già esistente, a meno che non dobbiate variarne le funzionalità in maniera significativa (in pratica, se non è guasto, è inutile ripararlo). *Ricompilare* il codice C in C++ è invece un'attività utile, perché potrebbe rivelare dei bug nascosti. Comunque, prendere del codice C che funziona bene e riscriverlo in C++ potrebbe non essere il modo migliore per passare il vostro tempo, a meno che la versione C++, essendo una classe, non offra molte più possibilità di riutilizzo.
5. Se avete un grosso blocco di codice C che ha bisogno di modifiche, cominciate ad isolarne le parti che non verranno modificate, eventualmente inglobandole in una "classe API" come metodi statici. Successivamente, focalizzate la vostra attenzione sul codice che verrà modificato, ristrutturandolo in classi in modo da rendere agevoli le modifiche man mano che la vostra manutenzione procede.
6. Tenete ben distinti il creatore della classe dal suo utilizzatore (il *programmatore client*). Chi utilizza la classe è il "cliente", e non ha bisogno né vuole sapere cosa accade dietro le quinte. Chi crea la classe dev'essere l'esperto di progettazione di classi e deve scriverla in modo che possa essere usata anche dal più principiante dei programmatori, continuando a comportarsi in maniera robusta nell'applicazione. L'utilizzo di una libreria è semplice soltanto se è trasparente.

7. Quando create una classe, usate i nomi più chiari possibile. Il vostro obiettivo è quello di rendere l'interfaccia di programmazione concettualmente semplice. Cercate di rendere i vostri nomi talmente chiari da rendere superflui i commenti. A tal fine, sfruttate l'overloading delle funzioni e gli argomenti di default per creare un'interfaccia intuitiva e facile da usare.
8. Il controllo dell'accesso consente a voi (creatori della classe) di fare in futuro le più grandi modifiche possibili senza danneggiare il codice client nel quale la classe è utilizzata. In questa prospettiva, mantenete tutto il più private possibile, e rendete public solo l'interfaccia della classe, utilizzando sempre le funzioni anziché i dati. Rendete i dati public solo quando siete costretti. Se gli utilizzatori di una classe non hanno bisogno di chiamare una funzione, dichiaratela private. Se una parte della vostra classe deve restare visibile agli eredi come protected, fornite un'interfaccia a funzioni piuttosto che esporre direttamente i dati. In questo modo, le modifiche di implementazione avranno un impatto minimo sulle classi derivate.
9. Non commettete l'errore di bloccarvi durante l'analisi. Ci sono certe cose delle quali non vi renderete conto finché non inizierete a scrivere codice e ad avere una specie di sistema funzionante. Il C++ ha al proprio interno delle barriere protettive; lasciate che lavorino per voi. Gli errori che commetterete in una classe o in un insieme di classi non distruggeranno l'integrità dell'intero sistema.
10. La vostra analisi ed il vostro progetto devono produrre, quantomeno, le classi del vostro sistema, le loro interfacce pubbliche, e le loro relazioni con le altre classi, in particolare con le classi base. Se il vostro metodo di progetto produce più di questo, chiedetevi se tutto ciò che viene prodotto dalla vostra tecnica serve a qualcosa durante il ciclo di vita di un programma. Se la risposta è no, mantenerlo avrà un costo per voi. I membri dei team di sviluppo tendono a non mantenere nulla che non contribuisca alla loro produttività; questo è un dato di fatto che molte tecniche di progetto non prendono in considerazione.
11. Prima scrivete il codice di test (prima di scrivere la classe), e tenetelo insieme alla classe. Rendete automatica l'esecuzione dei vostri test per mezzo di un makefile o di uno strumento simile. In questo modo, qualunque modifica può essere controllata automaticamente lanciando il codice di test, e gli errori verranno immediatamente scoperti. Sapendo di avere la rete di sicurezza dell'ambiente di test, sarete più propensi ad effettuare modifiche consistenti quando ne sentirete il bisogno. Ricordate che i maggiori miglioramenti nei linguaggi di programmazione vengono dai controlli interni forniti da controllo del tipo, gestione delle eccezioni e così via, ma queste caratteristiche servono fino ad un certo punto. Dovete arrivare in fondo alla strada che porta alla creazione di sistemi robusti introducendo i test che verificano le caratteristiche specifiche della vostra classe o del vostro programma.
12. Prima scrivete il codice di test (prima di scrivere la classe) in modo da controllare che il progetto della vostra classe sia completo. Se non siete in grado di scrivere il codice di test, significa che non sapete che aspetto presenta la vostra classe. Inoltre, il semplice atto di scrivere il codice di test farà spesso emergere caratteristiche aggiuntive o vincoli di cui avete bisogno nella classe - queste caratteristiche e questi vincoli non appaiono sempre durante l'analisi ed il progetto.
13. Ricordate una regola fondamentale dell'ingegneria del software [1]: *Tutti i problemi di progetto del software possono essere semplificati introducendo un ulteriore livello di dereferenziamento concettuale*. Quest'idea sta alla base dell'astrazione, la caratteristica primaria della programmazione orientata agli oggetti.
14. Rendete le classi le più atomiche possibile; in altri termini, date ad ogni classe un unico scopo chiaro. Se le vostre classi o il vostro progetto del sistema diventano troppo complicati, suddividete le classi complesse in classi più semplici. Il segnale

più ovvio di questo fatto è proprio la stessa dimensione: se una classe è grande, c'è la possibilità che stia facendo troppo e che andrebbe divisa.

15. Guardatevi dalle definizioni dei metodi lunghe. Una funzione lunga e complicata è difficile e costosa da mantenere, e probabilmente sta cercando di fare troppo da sola. Se trovate una funzione del genere, significa che, quantomeno, andrebbe suddivisa in alcune funzioni più piccole. Potrebbe anche suggerire la creazione di una nuova classe.
16. Guardatevi dalle liste di argomenti lunghe. Le chiamate di funzione diventano difficili da scrivere, leggere e mantenere. Piuttosto, cercate di spostare il metodo in una classe alla quale sia (più) adatto, e/o a passargli un oggetto come parametro.
17. Non ripetetevi. Se un pezzo di codice compare in molte funzioni nelle classi derivate, spostate quel codice in un'unica funzione nella classe base e chiamatelo dalle funzioni delle classi derivate. Non solo risparmierete spazio, ma consentirete un'agevole propagazione delle modifiche. Potete utilizzare una funzione inline per l'efficienza. Talvolta, la scoperta di questo codice comune porta considerevoli benefici alla funzionalità della vostra interfaccia.
18. Guardatevi dalle istruzioni switch o dagli if-else concatenati. Tipicamente, questo è un indicatore della programmazione di tipo *type-check*, che significa che state scegliendo quale codice eseguire in base ad un qualche genere di informazione sul tipo (il tipo esatto potrebbe non esservi immediatamente chiaro). Solitamente, potete rimpiazzare questo genere di codice sfruttando ereditarietà e polimorfismo; la chiamata ad una funzione polimorfica eseguirà il controllo di tipo per voi, e consentirà un estensibilità più affidabile e semplice.
19. Al momento del progetto, cercate e separate le cose che cambiano da quelle che restano uguali. In altre parole, individuate gli elementi del sistema che potreste voler modificare senza dover riprogettare tutto, quindi incapsulate quegli elementi nelle classi. Potete avere maggiori dettagli su questo concetto nel capitolo dedicato ai Design Pattern nel Volume 2 di questo libro, disponibile presso *www.BruceEckel.com*.
20. Guardatevi dalla *varianza*. Due oggetti semanticamente differenti potrebbero avere identiche funzioni, o responsabilità, e c'è una naturale tentazione di cercare di renderla una sottoclasse dell'altra, con l'unico scopo di trarre benefici dall'ereditarietà. Questa tecnica si chiama varianza, ma non c'è un valido motivo per forzare una relazione superclasse/sottoclasse quando questa non esiste. Una soluzione migliore sarebbe creare una classe base generale che genera per entrambe un'interfaccia - richiede un po' più spazio ma vi consente ancora di trarre vantaggio dall'ereditarietà e probabilmente di fare interessanti scoperte sul progetto.
21. Guardatevi dalle *limitazioni* nell'ereditarietà. I progetti più puliti aggiungono nuove funzionalità a quelle ereditate. Un progetto di cui diffidare rimuove le vecchie funzionalità durante l'eredità senza aggiungerne altre. Ma le regole sono fatte per essere infrante, e se state lavorando con una vecchia libreria di classi potrebbe essere più efficiente restringere una classe esistente nelle sue sottoclassi, piuttosto che ristrutturare la gerarchia in modo che la vostra nuova classe si vada ad inserire dove dovrebbe, al di sopra della vecchia classe.
22. Non estendete le funzionalità fondamentali nelle sottoclassi. Se un elemento dell'interfaccia è fondamentale per una classe si dovrebbe trovare nella classe base, e non essere aggiunto nel corso delle derivazioni. Se state aggiungendo dei metodi tramite l'eredità, forse dovreste ripensare il progetto.
23. Meno è più. Iniziate da un'interfaccia minimale per la classe, semplice e piccola quanto basta per risolvere il vostro problema corrente, ma non cercate di anticipare tutti i modi nei quali la vostra classe *potrebbe* essere usata. Al momento del suo utilizzo, scoprirete il modo nel quale dovrete espandere l'interfaccia. Comunque,

una volta che la classe è in uso, non potete modificarne l'interfaccia senza disturbare il codice client. Se avete bisogno di aggiungere più funzioni, va bene; non creerà problemi al codice, se non la necessità di una ricompilazione. Ma anche se i nuovi metodi rimpiazzano le funzionalità di quelle vecchie, lasciate da sola l'interfaccia già esistente (se volete, potete combinare le funzionalità nell'implementazione sottostante). Se avete bisogno di espandere l'interfaccia di una funzione esistente aggiungendo nuovi argomenti, lasciate gli argomenti già esistenti nel loro ordine, ed assegnate dei valori di default a tutti quelli nuovi; in questo modo, non creerete problemi a nessuna chiamata già esistente a quella funzione.

24. Leggete le vostre classi ad alta voce per assicurarvi che siano logiche, riportando la relazione tra classe base e classe derivata come "è un", e quella tra classe ed oggetto membro come "ha un".
25. Al momento di scegliere fra eredità e composizione, chiedetevi se avete bisogno di fare un upcast al tipo base. In caso negativo, privilegiate la composizione (oggetti membri) all'eredità. Ciò può eliminare la necessità percepita di usare l'eredità multipla. Se ereditate, gli utenti penseranno che prevediate che facciano upcast.
26. Talvolta, avete bisogno di ereditare per poter accedere ai membri protected della classe base. In questo modo potreste pensare di avere bisogno dell'eredità multipla. Se non avete bisogno di upcast, dapprima derivate una nuova classe per eseguire l'accesso protected. In seguito, rendete questa nuova classe un oggetto membro di qualunque classe che abbia bisogno di utilizzarlo, piuttosto che ereditarlo.
27. Tipicamente, una classe base verrà utilizzata principalmente per creare un'interfaccia alle classi derivate da lei. Pertanto, quando create una classe base, create tendenzialmente le funzioni membro come virtuali pure. Anche il distruttore può essere virtuale puro (per forzare gli eredi a sovrascriverlo esplicitamente), ma ricordate di dare al distruttore un corpo, perché tutti i distruttori di una gerarchia vengono sempre chiamati.
28. Quando inserite una funzione virtual in una classe, rendete virtual tutte le funzioni di quelle classe, ed inserite un distruttore virtual. Questo approccio previene sorprese nel comportamento dell'interfaccia. Iniziate a rimuovere la parola chiave virtual solo quando avete problemi di efficienza ed il vostro analizzatore vi ha suggerito questa soluzione.
29. Utilizzate le variabili membro per le variazioni di valore, e le funzioni virtual per le variazioni di comportamento. In altre parole, se trovate una classe che utilizza variabili di stato unitamente a funzioni membro che modificano il loro comportamento in base a quelle variabili, probabilmente dovreste riprogettarla esprimendo le differenze di comportamento tramite sottoclassi e funzioni virtual soprascritte.
30. Se dovete fare qualcosa di non portabile, create un astrazione per quel servizio ed inseritelo in una classe. Questo livello ulteriore di redirezione impedisce alla non portabilità di essere distribuita in tutto il programma.
31. Evitate l'eredità multipla. Serve a tirarvi fuori da situazioni spiacevoli, in particolare per riparare interfacce di classi sulle quali non avete il controllo (vedete il Volume 2). Dovreste essere programmatori esperti prima di progettare eredità multiple per il vostro sistema.
32. Non usate eredità private. Sebbene sia prevista dal linguaggio e talvolta sembri essere funzionale, introduce notevoli ambiguità quando è combinata all'identificazione run-time del tipo. Create un oggetto membro private, piuttosto che utilizzare l'eredità privata.
33. Se due classi sono in qualche maniera funzionalmente associate tra di loro (ad esempio come i contenitori e gli iteratori), cercate di rendere una delle due una classe public e friend annidata nell'altra, così come la Libreria Standard C++ fa con

gli iteratori dentro i contenitori (esempi di questa tecnica sono mostrati nella parte finale del Capitolo 16). In questo modo non solo viene enfatizzata l'associazione fra le due classi, ma, anidandolo all'interno di un'altra classe, si consente al nome della classe di essere riutilizzato. La Libreria Standard C++ fa questo definendo una classe iterator annidata in ciascuna classe contenitore, fornendo così ai contenitori un'interfaccia comune. L'altra ragione per la quale potreste voler annidare una classe è come parte di un'implementazione private. In questo caso, l'annidamento è utile più per nascondere l'implementazione che per sottolineare l'associazione fra le classi o prevenire l'inquinamento del namespace come descritto sopra.

34. L'overloading degli operatori è soltanto "zucchero sintattico": un modo diverso per chiamare una funzione. Se l'overloading di un operatore non rende l'interfaccia della classe più chiara e semplice da usare, non fatelo. Per una classe create solo un operatore per la conversione automatica di tipo. In generale, seguite le linee guida ed il formato descritto nel Capitolo 12 quando effettuate l'overloading degli operatori.
35. Non preoccupatevi di un'ottimizzazione prematura. È pura follia. In particolare, non preoccupatevi di scrivere (o evitare) le funzioni inline, di rendere non virtual certe funzioni, o di forzare il codice ad essere efficiente quando state appena costruendo il sistema. Il vostro scopo principale è di verificare il progetto, a meno che lo stesso richieda una certa efficienza.
36. Di norma, evitate che sia il compilatore a creare per voi costruttori, distruttori ed operator=. I progettisti delle classi dovrebbero sempre dire esattamente che cosa la classe dovrebbe fare e mantenere la classe completamente sotto controllo. Se non volete avere un costruttore di copia o un operator=, dichiarateli come private. Ricordate che, se create un qualunque costruttore, impedirete al costruttore di default di essere sintetizzato.
37. Se la vostra classe contiene puntatori, per farla funzionare correttamente dovete creare il costruttore di copia, operator= ed il distruttore.
38. Quando scrivete il costruttore di copia per una classe derivata, ricordatevi di richiamare esplicitamente il costruttore di copia della classe base (anche nelle versioni che hanno un oggetto come membro) (Vedete il Capitolo 14). Se non lo fate, per la classe base (o per l'oggetto membro) verrà richiamato il costruttore di default, e probabilmente non è quello che volete. Per chiamare il costruttore di copia della classe base, passategli l'oggetto derivato dal quale state copiando:

```
Derived(const Derived& d) : Base(d) { // ...
```

39. Quando scrivete un operatore di assegnazione per una classe derivata, ricordate di chiamare esplicitamente la versione dell'operatore di assegnazione della classe base. (Vedete il Capitolo 14.) Se non lo fate, allora non accadrà nulla (lo stesso vale per gli oggetti membri). Per richiamare l'operatore di assegnazione della classe base, usate il nome della classe base e la risoluzione di ambiente:

```
Derived& operator=(const Derived& d) {
  Base::operator=(d);
```

40. Se avete bisogno di minimizzare le ricompilazioni durante lo sviluppo di un progetto esteso, utilizzate la tecnica della classe di gestione/gatto Cheshire mostrata nel Capitolo 5, e rimuovetela solo se l'efficienza a runtime costituisce un problema.
41. Evitate il preprocessore. Usate sempre const per sostituire i valori e le funzioni inline per le macro.

42. Mantenete gli scope più piccoli possibile, in modo che la visibilità e la vita dei vostri oggetti siano le più ridotte possibile. In questo modo, diminuisce la possibilità di utilizzare un oggetto in un contesto sbagliato e di nascondere un bug difficile da trovare. Per esempio, supponete di avere un contenitore ed un frammento di codice che itera su di lui. Se copiate quel codice per usarlo con un altro contenitore, potreste trovarvi accidentalmente ad utilizzare la dimensione del vecchio contenitore come limite superiore per quello nuovo. Se, comunque, il vecchio contenitore è esterno allo scope, l'errore verrà scoperto al momento della compilazione.
43. Evitate le variabili globali. Cercate sempre di inserire i dati all'interno delle classi. È più probabile imbattersi in funzioni globali piuttosto che in variabili globali, sebbene potreste rendervi conto in seguito che una funzione globale troverebbe una collocazione più consona come metodo static di una classe.
44. Se avete bisogno di dichiarare una classe o una funzione di una libreria, fatelo sempre utilizzando un file header. Per esempio, se volete creare una funzione per scrivere su di un ostream, non dichiarate mai ostream per conto vostro utilizzando una specificazione di tipo incompleta come questa,

    ```
    class ostream;
    ```

    Questo approccio rende il vostro codice vulnerabile a cambiamenti di rappresentazione. (Per esempio, ostream potrebbe essere in effetti un typedef). Piuttosto, utilizzate il file header:

    ```
    #include <iostream>.
    ```

    Quando create le vostre classi, se una libreria è grande, fornite ai vostri utenti una versione abbreviata del file header, con dichiarazioni di tipo incomplete (vale a dire, dichiarazioni dei nomi delle classi) nei casi in cui debbano usare solamente i puntatori. (La compilazione potrebbe risultarne accelerata).

45. Quando scegliete il tipo restituito dall'overloading di un operatore, considerate che cosa accadrebbe se le espressioni venissero concatenate assieme. Restituite una copia di un reference ad un lvalue (`return *this`) in modo che possa essere utilizzato in espressioni concatenate (`A=B=C`). Quando definite operator=, ricordatevi di `x = x`.
46. Quando scrivete una funzione, come prima scelta passatele gli argomenti come const reference. Fintantoché non dovete modificare l'oggetto passato, questa è la scelta migliore, in quanto possiede la semplicità del passaggio per valore ma non richiede costose operazioni di costruzione e distruzione per creare un oggetto locale, cosa che accade quando un parametro viene passato per valore. Normalmente, non dovreste preoccuparvi eccessivamente delle problematiche di efficienza quando progettate e costruite il vostro sistema, ma quest'abitudine è di certo un punto vincente.
47. Siate consapevoli degli oggetti temporanei. Quando cercate le prestazioni, controllate la creazione di oggetti temporanei, soprattutto in presenza di overload degli operatori. Se i vostri costruttori e distruttori sono complicati, il costo di creare e distruggere oggetti temporanei può essere elevato. Quando restituite un oggetto da una funzione, cercate sempre di costruire l'oggetto "in loco" con una chiamata al costruttore nell'istruzione return:

    ```
    return MyType(i, j);
    ```

piuttosto di

```
MyType x(i, j);
return x;
```

La prima istruzione return (la cosiddetta ottimizzazione del valore restituito) elimina una chiamata al costruttore ed una chiamata al distruttore.

48. Quando create i costruttori, prendete in considerazione le eccezioni. Nella migliore delle ipotesi, il costruttore non farà nulla che lanci un'eccezione. Nello scenario appena peggiore, la classe sarà composta ed erediterà solo da classi robuste, che quindi si puliranno automaticamente se un'eccezione viene lanciata. Se dovete avere puntatori "scoperti", siete responsabili di catturare le vostre eccezioni e quindi di deallocare tutte le risorse puntate prima di lanciare un eccezione nel vostro costruttore. Se un costruttore deve fallire, il modo migliore per farlo è lanciare un'eccezione.
49. Nei vostri costruttori fate solo il minimo necessario. In questo modo, non solo produrrete un costo minimo per le chiamate al costruttore (molte delle quali potrebbero non essere sotto il vostro controllo), ma è anche meno probabile che i vostri costruttori lancino eccezioni o creino problemi.
50. La responsabilità del distruttore è di rilasciare le risorse allocate durante l'intera vita dell'oggetto, non solo durante la costruzione.
51. Utilizzate le gerarchie di eccezioni, preferibilmente derivandole dalla gerarchia di eccezioni standard del C++ ed annidate come classi public all'interno della classe che lancia l'eccezione. La persona che cattura l'eccezione può così catturare i tipi specifici di eccezione, seguiti dal tipo base. Se aggiungete nuove eccezioni derivate, il codice client esistente catturerà ancora l'eccezione attraverso il tipo base.
52. Lanciate le eccezioni per valore, e catturatele per riferimento. Lasciate che sia il meccanismo di gestione delle eccezioni ad occuparsi della gestione della memoria. Se lanciate puntatori ad oggetti eccezione che sono stati creati sull'heap, chi la cattura deve sapere di doverla distruggere, e questo è un comportamento scorretto. Se catturate le eccezioni per valore, provocate ulteriori costruzioni e distruzioni; peggio ancora, le porzioni derivate dei vostri oggetti eccezione potrebbero andare perdute nel corso dell'upcast per valore.
53. Non create dei vostri propri class template, a meno che non vi siate costretti. Prima guardate nella Libreria Standard C++, quindi rivolgetevi ai venditori che creano strumenti di tipo special-purpose. Acquistate dimestichezza con il loro uso ed incrementerete notevolmente la vostra produttività.
54. Quando create i template, cercate il codice che non dipende dal tipo e collocatelo in una classe base esterna al template per prevenire un inutile rigonfiamente del codice. Utilizzando l'ereditarietà o la composizione, potete creare template nei quali la maggior parte del codice dipende dal tipo ed è quindi essenziale.
55. Non utilizzate le funzioni <cstdio>, come printf(). Piuttosto, imparate ad utilizzare gli iostream; sono funzioni type-safe e type-extensible, e nettamente più potenti. Il vostro sforzo sarà regolarmente ricompensato. In generale, usate sempre le librerie C++ piuttosto delle librerie C.
56. Evitate i tipi nativi del C. Sono supportati dal C++ solo per ragioni di compatibilità all'indietro, ma sono molto meno robusti delle classi C++, quindi il tempo dedicato alla ricerca dei bug aumenta.
57. Tutte le volte che usate i tipi nativi come globali o automatici, non definiteli fintantoché non potete anche inizializzarli. Definite la variabili una per riga, ciascuna con la propria inizializzazione. Quando definite i puntatori, posizionate la

'*' vicino al nome del tipo. Potete farlo senza pericolo, se definite una variabile per riga. Questo stile tende a confondere meno l'utente.

58. Garantite che l'inizializzazione abbia luogo in tutti gli aspetti del vostro codice. Eseguite tutte le inizializzazioni dei membri nella lista di inizializzazione del costruttore, anche per i tipi nativi (utilizzate chiamate a pseudocostruttori). L'utilizzo della lista di inizializzazione del costruttore è spesso più efficiente quando si inizializzano gli oggetti membro; in caso contrario viene chiamato il costruttore di default, e finite per chiamare, dopo di lui, altre funzioni membro (probabilmente operator=) per ottenere l'inizializzazione desiderata.
59. Non utilizzate la forma `MyType a = b;` per definire un oggetto. Questa caratteristica è una delle maggiori fonti di confusione, perché viene chiamato il costruttore e non operator=. Per chiarezza, siate sempre precisi ed utilizzate piuttosto la forma `MyType a(b);`. Il risultato è identico, ma non confonderete le idee agli altri programmatori.
60. Utilizzate i cast espliciti descritti nel Capitolo 3. Un cast scavalca il sistema normale dei tipi, ed è una fonte potenziale di errori. Dal momento che i cast specifici suddividono il cast "tuttofare" del C in categorie di cast ben definiti, chiunque esegua il debug o mantenga il codice è in grado di trovare facilmente tutti i punti nei quali è più probabile il verificarsi di errori logici.
61. Perché un programma sia robusto, ogni singolo componente deve essere robusto. Sfruttate tutti gli strumenti offerti dal C++: controllo degli accessi, eccezioni, correttezza delle costanti, controllo del tipo, e così via, e questo in ogni classe che create. In questo modo, quando costruite il vostro sistema, potete muovervi con sicurezza verso il livello di astrazione successivo.
62. Utilizzate la correttezza delle costanti. Consente al compilatore di evidenziare dei bug che, altrimenti, sarebbero stati infidi e difficili da individuare. Questa abitudine ha bisogno di una certa disciplina e dev'essere utilizzata in maniera consistente in tutte le classi, ma paga.
63. Sfruttate a vostro vantaggio il controllo degli errori effettuato dal compilatore. Compilate il vostro codice abilitando tutti i warning, e correggete il vostro codice in modo da eliminarli tutti. Scrivete codice che utilizza gli errori ed i warning al momento della compilazione, piuttosto che codice che provochi errori di runtime (ad esempio, non utilizzate liste di argomenti variadic, che impediscono qualsiasi controllo dei tipi). Utilizzate assert() per il debug, ma a runtime usate le eccezioni.
64. Preferite gli errori di compilazione agli errori di runtime. Cercate di gestire un errore il più vicino possibile al punto in cui si è verificato. È preferibile gestire l'errore in quel punto piuttosto che lanciare un eccezione. Catturate le eccezioni nel gestore più vicino che abbia informazioni sufficienti per gestirle. Fate il possibile col l'eccezione al livello corrente; se in questo modo non risolvete il problema, rilanciatela nuovamente. (Vedete il Volume 2 per maggiori dettagli.)
65. Se state utilizzando le specifiche delle eccezioni (vedete il Volume 2 di questo libro, scaricabile a *http://www.BruceEckel.com*, per imparare a gestire le eccezioni) installate una vostra funzione unexpected() utilizzando set_unexpected(). La vostra funzione unexpected() dovrebbe registrare l'errore e rilanciare l'eccezione corrente. In questo caso, se una funzione esistente viene scavalcata e comincia a lanciare eccezioni, avrete a disposizione una traccia di quanto è accaduto e potrete modificare il codice chiamante per gestire l'eccezione.
66. Create una funzione terminate() (che indica un errore di programmazione) personalizzata per registrare l'errore che ha portato all'eccezione, dopodiché rilasciate le risorse del sistema ed uscite dal programma.
67. Se un distruttore chiama una qualunque funzione, quella funzione potrebbe lanciare un'eccezione. Un distruttore non può lanciare un'eccezione (ne potrebbe risultare

una chiamata a terminate(), che indica un errore di programmazione), quindi qualunque distruttore che chiama funzioni deve catturarne e gestirne le eccezioni.

68. Non create una vostra notazione "personalizzata" per i nomi delle variabili membro (underscore prefissi, notazione ungherese e così via), a meno che non abbiate una gran quantità di variabili globali preesistenti; se così non è, lasciate che le classi ed i namespace svolgano il lavoro per voi.
69. Prestate attenzione all'overload. Una funzione non dovrebbe eseguire del codice condizionatamente, in base al valore di un argomento, che sia o meno di default. In questo caso, dovreste invece creare due o più funzioni in overload.
70. Nascondete i vostri puntatori all'interno di classi contenitore. Portateli all'esterno solo immediatamente prima di eseguire operazioni su di essi. I puntatori sono sempre stati una grossa fonte di errori. Quando utilizzate new, cercate di racchiuderne il risultato in un contenitore. Fate in modo che i contenitori "possiedano" i loro puntatori, in modo che siano responsabili per il rilascio delle risorse. Ancora meglio, inserite il puntatore in una classe; se volete ancora che si comporti come un puntatore, effettuate l'overload di operator-> e di operator*. Se dovete necessariamente avere un puntatore libero, inizializzatelo sempre, preferibilmente con l'indirizzo di un oggetto, ma se necessario anche a zero. Impostatelo a zero quando chiamate la delete, per prevenire eliminazioni multiple.
71. Evitate l'overload degli operatori new e delete a livello globale, ma fatelo sempre classe per classe. L'overload globale influenza sempre l'intero progetto, che invece andrebbe controllato solamente dal creatore del progetto. Al momento dell'overload di new e delete per una classe, non assumete di conoscere la dimensione dell'oggetto; qualcuno potrebbe stare ereditando da voi. Utilizzate l'argomento fornito. Se fate qualcosa di particolare, considerate l'effetto che potrebbe avere sugli eredi.
72. Prevenite lo sfaldamento degli oggetti. Non ha praticamente mai senso fare l'upcast di un oggetto per valore. Per prevenire l'upcast per valore, inserite funzioni virtuali pure nella vostra classe base.
73. Talvolta la semplice aggregazione è tutto ciò che serve. Un "sistema di confort per i passeggeri" di una linea aerea si compone di elementi sconnessi: sedile, aria condizionata, video, e così via; perdipiù avete bisogno di crearne molte istanze per un aereo. Forse che costruite dei membri privati e costruite da zero una nuova interfaccia? No - in questo caso i componenti appartengono anche all'interfaccia pubblica, quindi dovreste creare oggetti membro pubblici. Questi oggetti hanno una loro implementazione privata, che è ancora sicura. Tenete conto che l'aggregazione pura e semplice non è una soluzione da usare spesso, ma talvolta accade.

---

[1] Spiegatami da Andrea Koenig

C: Letture consigliate

Approfondimenti di C e C++.

C

Thinking in C: Foundations for Java & C++, di Chuck Allison (è un seminario MindView, Inc., ©2000, disponibile su C-ROM; disponibile con il CD allegato al libro oppure all'indirizzo *www.BruceEckel.com*). Si tratta delle lezioni e delle trasparenze di un corso introduttivo al linguaggio C e propedeutico allo studio del Java o del C++. Il corso non è da ritenersi esaustivo; vengono trattati solo gli argomenti ritenuti necessari per l'apprendimento degli altri linguaggi. In sezioni specifiche vengono introdotti gli aspetti utili per i futuri programmatori C++ o Java. Prerequisiti consigliati: qualche esperienza con linguaggi di programmazione ad alto livello, come Pascal, BASIC, Fortran, oppure LISP (è possibile intraprendere la lettura del CD senza avere queste conoscenze, anche se il corso non è stato pensato come introduzione ai fondamenti della programmazione).

C++ in generale

The C++ Programming Language, 3rd edition, di Bjarne Stroustrup (Addison-Wesley 1997). In un certo senso, lo scopo di *Penare in C++* è di permettere l'utilizzo del libro di Bjarne come un manuale. Siccome questo libro contiene la descrizione del linguaggio fatta dall'autore, rappresenta la fonte migliore a cui attingere per risolvere qualunque incertezza sul C++. Quando si è acquistata una buona confidenza con il linguaggio e si è pronti per programmare seriamente, questo libro diventa indispensabile.

C++ Primer, 3rd Edition, di Stanley Lippman e Josee Lajoie (Addison-Wesley 1998). Non è un libro per principianti così come potrebbe suggerire il titolo. È un libro voluminoso che scende in molti dettagli, ed è il testo al quale fare riferimento, insieme allo Stroustrup, per risolvere i problemi. *Pensare in C++* dovrebbe fornire le basi per poter affrontare entrambi *C++ Primer* ed il libro di Stroustrup.

C & C++ Code Capsules, di Chuck Allison (Prentice-Hall, 1998). Questo libro presuppone la conoscenza sia il C che il C++, e affronta alcuni argomenti che vengono facilmente dimenticati, o che non vengono compresi quando vengono incontrati per la prima volta. Utile, quindi, per rimediare ad alcune lacune che si potrebbero avere riguardo il C ed il C++.

The C++ Standard. Su questo documento il comitato di standardizzazione ha lavorato duro per anni. Purtroppo non si tratta di un documento gratuito. In ogni modo è possibile comprarlo per soli 18 dollari in formato PDF all'indirizzo *www.cssinfo.com*.

I miei libri

Di seguito sono riportati in ordine di pubblicazione i libri pubblicati dall'autore di *Pensare in C++*. Alcuni di essi non sono più disponibili.

Computer Interfacing with Pascal & C (pubblicato attraverso la Eisys imprint, nel 1988. Disponibile solo attraverso il sito *www.BruceEckel.com*). È un introduzione

all'elettronica che risale ai tempi in cui CP/M la faceva da padrone ed il DOS era solo agli inizi. Vengono utilizzati linguaggi ad alto livello e la porta parallela per pilotare i circuiti elettronici presentati. Si tratta di un adattamento degli articoli scritti per la prima, ed anche migliore, rivista con la quale l'autore ha collaborato: *Micro Cornucopia* (per parafrasare Larry O'Brien, per lungo tempo editore di *Software Development Magazine*: "la migliore rivista di computer mai pubblicata – avevano anche in progetto di costruire un robot in un vaso da fiori!"). Purtroppo Micro C è scomparsa prima che nascesse Internet.

Using C++ (Osborne/McGraw-Hill 1989). Uno dei primi libri sul C++ ad essere stati pubblicati. Oramai è andato fuori stampa ed è stato sostituito dalla seconda edizione intitolata **C++ Inside & Out**.

C++ Inside & Out (Osborne/McGraw-Hill 1993). Come già detto si tratta della 2ª edizione di **Using C++**. La trattazione di *C++ Inside & Out* è ragionevolmente accurata, ma risale al 1992 e *Thinking in C++* è stato pensato per rimpiazzare questo libro. Altre informazioni riguardo questo libro e il codice sorgente degli esempi sono disponibili al sito *www.BruceEckel.com*.

Thinking in C++, 1st edition (Prentice-Hall 1995).

Black Belt C++, the Master's Collection, edito da Bruce Eckel (M&T Books 1994). Fuori stampa. Si tratta di un insieme di estratti dagli interventi di vari esperti di C++ alla Software Development Conference, presieduta dall'autore. La copertina di questo libro ha stimolato l'autore ad occuparsi della realizzazione di tutte le copertine delle pubblicazioni successive.

Thinking in Java, 2ª edizione (Prentice-Hall, 2000). La prima edizione di questo libro vinse il*Software Development Magazine* Productivity Award e il *Java Developer's Journal* Editor's Choice Award nel 1999. È possibile scaricarne una copia al sito *www.BruceEckel.com*.

Approfondimenti e angoli oscuri

Questi libri approfondiscono vari aspetti del linguaggio, e aiutano a non commettere errori comuni durante lo sviluppo di programmi in C++.

Effective C++ (2nd Edition, Addison-Wesley 1998) e **More Effective C++** (Addison-Wesley 1996), di Scott Meyers. Due classici; entrambi sono libri da avere ed affrontano seriamente la risoluzione di problemi e la progettazione del codice utilizzando il C++. In *Pensare in C++* si è cercato di catturare molti dei concetti esposti da questi libri, ma sicuramente lo scopo non è stato raggiunto. Se si vuole lavorare seriamente con il C++, prima o poi si dovranno affrontare queste due letture. Disponibili anche in CD ROM.

Ruminations on C++, di Andrew Koenig e Barbara Moo (Addison-Wesley, 1996). Andrew è un'autorità avendo lavorato direttamente con Stroustrup su molti aspetti del linguaggio C++. I suoi suggerimenti sono particolarmente illuminanti, e si può imparare molto leggendo il suo libro oppure, se si ha la fortuna, incontrandolo di persona.

Large-Scale C++ Software Design, di John Lakos (Addison-Wesley, 1996). Questo libro affronta alcuni argomenti che tornano utili durante lo sviluppo di progetti grandi e piccoli, fornendo risposte a problemi frequenti.

C++ Gems, curato da Stan Lippman (SIGS publications, 1996). Una selezione di articoli tratti da *The C++ Report*.

The Design & Evolution of C++, di Bjarne Stroustrup (Addison-Wesley 1994). In questo libro l'inventore del linguaggio svela le ragioni di alcune delle scelte progettuali fatte durante la creazione del C++. Da non ritenersi essenziale, ma pur sempre d'interesse.

Analisi e progettazione

Extreme Programming Explained di Kent Beck (Addison-Wesley 2000). *Adoro* questo libro. Nonostante abbia mantenuto un atteggiamento radicale verso il mondo che ci circonda, ho sempre ritenuto possibile lo sviluppo di una metodologia di programmazione diversa e migliore, e penso che Extreme Programming ci vada dannatamente vicino. L'unico libro che ha avuto lo stesso impatto su di me è stato *PeopleWare* (vedi sotto), che tratta principalmente tematiche che riguardano l'ambiente e la cultura aziendale. *Extreme Programming Explained* parla di programmazione, trattando svariate tematiche, tra le quali alcune correlate a recenti "scoperte". L'autore si spinge oltre, affermando che molte novità vanno bene fintantoché non si spende troppo tempo per poterle utilizzare: in questo caso è meglio lasciare perdere (si noti che questo libro non ha il marchio di approvazione UML sulla copertina). Potrei decidere di andare a lavorare per un azienda basandomi solamente sul fatto che utilizzino o meno Extreme Programming. Libro piccolo, con capitoli brevi che si leggono senza sforzo, e che fornisce molti spunti di riflessione. Il libro ha la capacità di aprire al lettore una finestra su un nuovo mondo mentre è immerso nel proprio ambiente di lavoro.

UML Distilled di Martin Fowler (2ª edizione, Addison-Wesley, 2000). Quando si incontra per la prima volta l'UML, ci si scoraggia facilmente perché ci sono molti tipi diagrammi e molti dettagli ai quali bisogna fare attenzione. Secondo l'autore di questo libro, molti di questi particolari non sono necessari, quindi la trattazione è stata ridotta a ciò che viene ritenuto essenziale. Nella maggior parte dei progetti reali si ha bisogno solo di una parte degli strumenti grafici messi a disposizione dall'UML, e l'obiettivo di Flower è quello di ottenere un buon progetto, piuttosto che preoccuparsi dei dettagli. Lettura piccola e piacevole; è un libro per chi tenta il primo approccio con l'UML.

The Unified Software Development Process di Ivar Jacobsen, Grady Booch, e James Rumbaugh (Addison-Wesley 1999). Inizialmente ero totalmente prevenuto nei riguardi di questo libro. Ai miei occhi sembrava avere tutte le caratteristiche di un noioso libro universitario. Dopo la lettura, invece, sono rimasto piacevolmente sorpreso e ritengo che solo le versioni tascabili del libro spieghino l'UML in maniera da sembrare poco chiaro anche agli autori stessi. Il nucleo del libro oltre ad essere molto chiaro, è anche di piacevole lettura. La cosa migliore, poi, è che il metodo illustrato è molto utile dal punto di vista pratico. Non si tratta di un libro come Extreme Programming (e non ha la stessa semplicità di lettura), ma fa sempre parte del "carozzone" UML – ed anche se non utilizzerete Extreme Programming, dato che molte persone sono salite sul carro dell'UML (a prescindere dal loro reale livello di conoscenza del metodo), probabilmente incontrerete questo libro. Penso che questo libro rappresenti il libro sull'UML per antonomasia (anche perché è stato scritto dagli inventori del metodo n.d.t.), ed è il libro al quale fare riferimento, dopo aver letto *UML Distilled* di Flower, quando si vuole approfondire l'argomento.

Prima di scegliere una metodologia di progetto, potrebbe essere d'aiuto conoscere il punto di vista di qualcuno che non stia cercando di vendere una soluzione particolare. È semplice scegliere una tecnica senza conoscere bene cosa si vuole ottenere. A volte il solo fatto che venga utilizzata da altri ne giustifica la scelta. Il fattore psicologico gioca un ruolo particolare sul comportamento umano: se si crede che qualcosa possa risolvere i propri problemi, si è portati a provarla (che è una cosa positiva e si chiama sperimentazione). Spesso, però, se i problemi non vengono risolti, invece di abbandonare il metodo, si aumentano gli sforzi a si annunciano altre importanti scoperte (che è una cosa negativa e si chiama rifiuto di ammettere i propri errori). Il tentativo è quello di non affondare, tentando di far salire altre persone sulla propria nave mentre questa sta affondando.

Questo non vuol dire che tutte le metodologie di progetto non servono a nulla, ma che bisogna dotarsi di strumenti che aiutino a sperimentare nuove soluzioni ("Non funziona; proviamo qualcos'altro") e che non neghino l'evidenza ("Questo non è un problema reale. Va tutto bene , non c'è bisogno di nessun cambiamento"). I libri seguenti dovrebbero fornire questi strumenti e andrebbero letti *prima* di scegliere una particolare metodologia.

Software Creativity, di Robert Glass (Prentice-Hall, 1995). Personalmente ritengo che questo sia il miglior libro che affronti l'argomento delle metodologie di progettazione nel suo complesso. Si tratta di una raccolta di brevi articoli, scritti da Glass e da altri autori (tra i quali P.J. Plauger), nei quali vengono riportate le riflessioni e gli studi fatti sull'argomento nel corso degli anni. La lettura del libro è piacevole e gli autori si dilungano solo quando è strettamente necessario, senza annoiare. Nel caso si voglia approfondire qualche argomento, il libro è ricco di riferimenti bibliografici e rimandi ad altri articoli. Tutti i programmatori e i manager dovrebbero leggere questo libro prima di inoltrarsi nel mare delle metodologie di progetto.

Software Runaways: Monumental Software Disasters, di Robert Glass (Prentice-Hall 1997). La cosa bella di questo libro è che porta in primo piano una cosa di cui ancora non si è parlato: quanti sono i progetti che falliscono, e quanti sono quelli che lo fanno in maniera spettacolare. La maggior parte di noi pensa: "Questo non potrà mai accadere a me" (oppure "Potrà mai capitare *ancora*?"); questo comportamento porta solo svantaggi. Tenendo presente che le cose possono sempre mettersi per il verso sbagliato, ci si mette in una posizione favorevole affinché le cose vadano per il meglio.

Object Lessons di Tom Love (SIGS Books, 1993). Un altro buon libro che affronta il tema dello sviluppo delle metodologie di progettazione.

Peopleware, di Tom Demarco e Timothy Lister (Dorset House, 2ª edizione 1999). Nonostante gli autori abbiano una formazione come sviluppatori di software, questo libro parla di progetti e team di lavoro. L'attenzione è incentrata sulle *persone* e sui loro bisogni piuttosto che sulla tecnologia. Gli autori parlano di come creare un ambiente dove le persone siano felici e produttive, piuttosto che decidere quali regole queste persone dovrebbero seguire per essere parti della macchina produttiva. È mia convinzione che quest'ultimo approccio sia il principale responsabile dei sorrisi e dell'accondiscendere dei programmatori quando si addotta la nuova metodologia XYZ, anche se continuano a fare quello che hanno sempre fatto.

Complexity, di M. Mitchell Waldrop (Simon & Schuster, 1992). Questo libro racconta come un gruppo di studiosi di diverse discipline sia stato messo insieme a Santa Fe, in New Mexico, per discutere sui problemi che ognuna delle discipline non riusciva a risolvere (il mercato azionario in economia, l'origine della vita in biologia, perché le persone fanno

quello che fanno in sociologia, ecc.). Incrociando fisica, economia, chimica, matematica, computer science, sociologia ed alter materie, è stato sviluppato un approccio multi-disciplinare a questi problemi. In questo contesto è emerso un nuovo modo di approcciare i problemi complessi non più basato sul determinismo matematico e sull'illusione di poter scrivere un'equazione che predica tutti questi comportamenti, bensì basandosi sull'osservazione, cercando di individuare dei pattern da riprodurre nella maniera più fedele possibile (il libro, per esempio, racconta la nascita degli algoritmi genetici). Questo modo di affrontare i problemi è utile quando si ha la necessità di gestire progetti software sempre più complessi.