

# Introduzione al C++

# Sommario

- ▶ Le differenze fra C e C++
  - ▶ commenti
- ▶ Funzioni
  - ▶ parametri per funzioni
  - ▶ funzioni inline
  - ▶ overloading di funzione
- ▶ Varie
  - ▶ input/output
  - ▶ nuovi header
  - ▶ punto di dichiarazione delle variabili locali
- ▶ Memoria
  - ▶ allocazione/deallocazione dinamica di memoria

# Il C++ e diversi paradigmi di programmazione

- ▶ Programmazione imperativa
  - ▶ Si specificano i passi da eseguire per raggiungere uno stato voluto
  - ▶ Vs. programmazione dichiarativa (ex. *Prolog*)
- ▶ Programmazione procedurale
  - ▶ Si usano procedure (sub-routine o funzioni) per rendere modulare la soluzione di un problema
- ▶ Programmazione orientata agli oggetti (*OOP*)
  - ▶ Si vede il codice come oggetti che interagiscono fra loro e si scambiano messaggi
- ▶ Programmazione generica
  - ▶ Si usa lo stesso codice indipendentemente dai tipi di dato (*template*)

# Differenze C vs C++

- ▶ Il C++ contiene il C
  - ▶ è possibile scrivere un programma in C e compilarlo sotto un compilatore C++
  - ▶ NOTA: il compilatore deve essere informato di quale compilazione viene richiesta se C (gcc) o C++ (g++)
- ▶ Differenze
  - ▶ il C++ è orientato agli oggetti e alla programmazione generica; il C no
  - ▶ Qualche differenze sintattica

# Commenti

- ▶ I commenti vengono ignorati dal compilatore, ovvero non vengono tradotti in codice eseguibile

- ▶ in C

`/*`

    righe multiple

    seconda riga

`*/`

- ▶ in C++

`//` unica riga

# L'importanza della documentazione

- ▶ Scrivere commenti all'interno del programma è utile poiché si rendono più leggibili e facilmente **interpretabili** sia per un secondo programmatore che deve intervenire sul codice che per se stessi a distanza di tempo
- ▶ Cosa scrivere:
  - ▶ In una **funzione**: una breve spiegazione sul compito che esegue la funzione (prima della sua dichiarazione)
  - ▶ Nelle dichiarazioni dei **parametri**: quale è il significato dei vari parametri (usare nomi lunghi e autoesplicativi è una strategia migliore)
  - ▶ all'interno di un blocco di **codice**: separare e identificare le varie fasi di elaborazione

# Dilemma: *troppo o troppo poco*

- ▶ Scrivere il **minimo** indispensabile perché sia possibile **comprendere** il funzionamento del programma/funzione eliminando il codice e leggendo **solo i commenti**

# Funzione

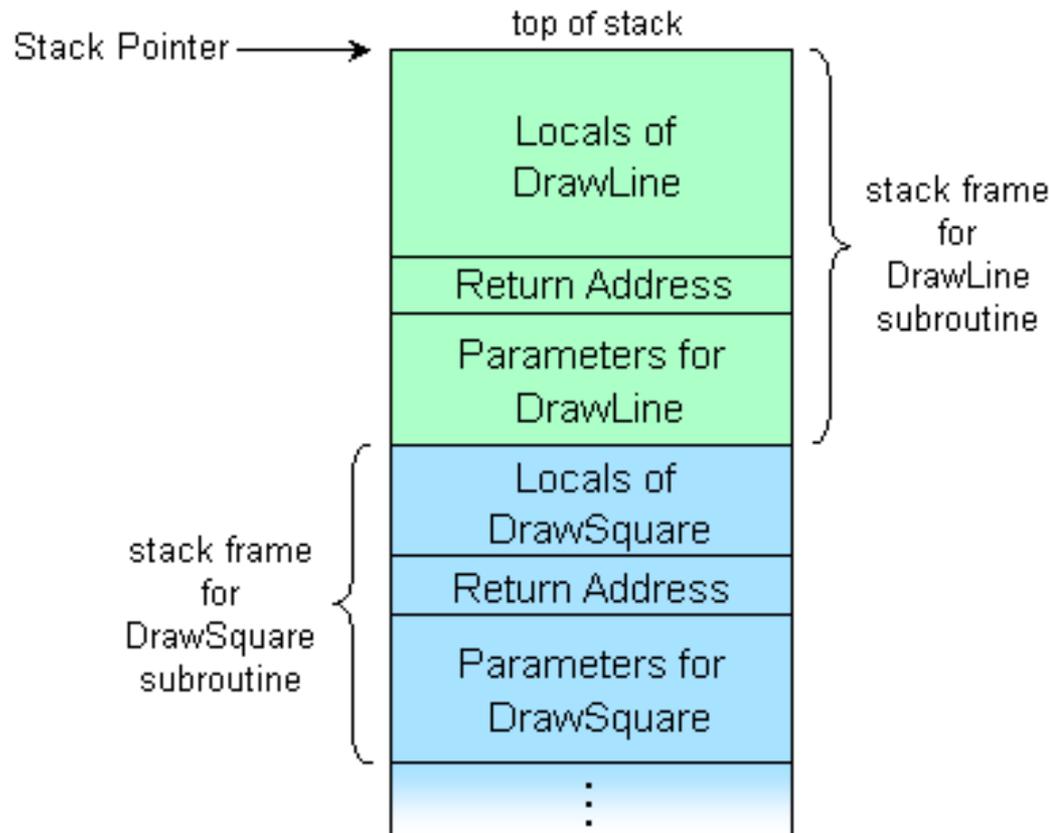
- ▶ **Denominazioni:** funzione, metodo, procedura, subroutine
- ▶ **Cosa:** una funzione e' una porzione di codice (in genere all'interno di un programma piu' grande) che esegue un compito ed e' **indipendente** dal resto del codice
- ▶ **Caratteristiche:**
  - ▶ Puo' essere chiamata o invocata piu' volte e da punti diversi di un programma
  - ▶ Puo' essere chiamata da se stessa (**ricorsione**)
  - ▶ Si possono raggruppare in librerie (riuso del codice)
- ▶ **Vantaggi:**
  - ▶ Usate per decomporre un problema in piu' sotto-problemi semplici
  - ▶ Nascondono parti di programma (**information hiding**)
  - ▶ Riducono la ridondanza del codice
  - ▶ Riutilizzo in diversi programmi
  - ▶ Migliorano la leggibilita' del codice
  - ▶ Migliorano l'estensibilita' e la manutenibilita'

# Stack Frame

- ▶ **Cosa:** Lo stack frame o call stack e' una struttura dati speciale (stack) usata per gestire le informazioni di una funzione attiva (cioe' che e' stata chiamata ma che non ha ancora finito la sua esecuzione)
- ▶ **Caratteristiche:**
  - ▶ Memorizza l'indirizzo di ritorno
  - ▶ Memorizza le variabili locali
  - ▶ Memorizza i parametri passati
- ▶ **Funzionamento:**
  - ▶ Ogni volta che si invoca una funzione si inserisce nello stack l'indirizzo di ritorno e i vari parametri
  - ▶ Ogni volta che termina l'esecuzione di una funzione questa passa il controllo all'indirizzo memorizzato nello stack
  - ▶ In questo modo si sa da dove riprendere l'esecuzione
  - ▶ *NOTA:* la ricorsione viene trattata automaticamente

# Esempio

- ▶ Una funzione DrawSquare puo' invocare piu' volte la funzione DrawLine
- ▶ ..ad ogni invocazione si aggiunge uno stack frame



# Il parametro void

- ▶ in C:
  - ▶ `function()` indica al compilatore di non controllare i parametri passati alla funzione, cioè si possono passare un numero arbitrario di parametri
- ▶ in C++:
  - ▶ `function()` è equivalente a `function(void)`, cioè non deve essere passato alcun parametro

# Parametri di default

- ▶ Alcune funzioni sono invocate ripetutamente con lo **stesso valore** per un dato argomento
- ▶ E' utile e possibile indicare allora un parametro come argomento di default fornendone un valore
- ▶ Se in seguito si invoca la funzione **omettendo** il parametro, il compilatore lo inserirà automaticamente inizializzandolo con il valore di default
- ▶ Gli argomenti di default devono trovarsi agli **ultimi** posti della lista degli argomenti
- ▶ I valori di default possono essere indicati nel **prototipo** della funzione

# Parametri di default in C++

- ▶ In C++ parametri di default

```
int f(int a, int b=2, int c=3){return a+b+c;}
```

- ▶ In chiamata:

```
f(10) = 10+2+3 = 15
```

```
f(10,20) = 10+20+3 = 23
```

```
f(10,20,30) = 10+20+30 = 60
```

- ▶ I parametri con default devono essere sempre gli ultimi. E' un errore scrivere:

```
int f(int a, int b=2, int c)
```

# Esempio

```
#include <iostream>

int boxVolume(int length=1, int width=1, int height=1);

int main()
{
    cout << "The default box volume is: " << boxVolume()
        << "\n\nThe volume of a box with length 10,\n"
        << "width 1 and height 1 is: " << boxVolume( 10 )
        << "\n\nThe volume of a box with length 10,\n"
        << "width 5 and height 1 is: " << boxVolume( 10, 5 )
        << "\n\nThe volume of a box with length 10,\n"
        << "width 5 and height 2 is: " << boxVolume(10,5,2)
        << endl;
    return 0;
}

// Calculate the volume of a box
int boxVolume( int length, int width, int height )
{
    return length * width * height;
}
```

# Funzioni inline

- ▶ Ogni chiamata a funzione ha un costo, cioè per poter **richiamare** una funzione il processore impiega un certo tempo
- ▶ Quando è cruciale avere tempi di esecuzione il più brevi possibile, per funzioni di **uso frequente** e di **piccola dimensione** può essere conveniente richiedere che il compilatore generi una copia del codice della funzione nei diversi punti del programma in cui essa è chiamata
- ▶ Per far questo si premette il qualificatore **inline** al **prototipo** della funzione (non alla definizione)

```
inline int square( int );
```

- ▶ **Note:**

- ▶ Il codice può **aumentare** considerevolmente di dimensione
- ▶ Il compilatore può decidere di **ignorare** la richiesta inline se la funzione è troppo grande o viene chiamata ricorsivamente

# L'*overloading* di funzioni

- ▶ **Overloading:** in C++ è possibile definire diverse funzioni con lo **stesso nome** purché ognuna abbia un insieme di parametri diverso
- ▶ **Utilizzo:** l'*overloading* (sovraccarico) serve per scrivere funzioni diverse che effettuano operazioni **simili** su tipi di dati diversi
- ▶ Il compilatore sa quale funzione scegliere esaminando il tipo, l'ordine e il numero dei parametri presenti nella chiamata
- ▶ **Esempio:**
  - ▶ Una funzione di visualizzazione di un cerchio che prende in ingresso il centro e il raggio o le coordinate di due punti che definiscono il quadrato che lo circonda
  - ▶ Una funzione di stampa che prende in ingresso un numero variabile di elementi da stampare

# Esempio

```
#include <iostream>

inline int square( int );
inline double square( double );

int main()
{
    int x=7;
    double y=7.5
    cout << "The square of integer"<<x<< " is " << square( x )
         << "\nThe square of double"<<y<<" is " << square( y )
         << endl;
    return 0;
}

int square( int x ) { return x * x; }

double square( double y ) { return y * y; }
```

# Nuovi header

- ▶ Gli implementatori di un compilatore hanno già scritto la maggior parte delle funzioni di utilizzo generale. Queste sono contenute nelle librerie standard.
- ▶ Il C++ contiene l'intera libreria del C standard ANSI
- ▶ Ogni libreria standard ha un suo **file di intestazione** corrispondente o **header file**
- ▶ I file di intestazione per il C hanno nomi che terminano in .h
- ▶ Gli header mutuati dal C in C++ hanno nomi identici con la lettera "c" prefissa e nessuna terminazione, cioè l'header math.h è cmath

# Header di libreria standard

- ▶ Le librerie di uso più comune sono:
  - ▶ `<cstdlib>`: conversione di numeri in stringhe e viceversa, allocazione dinamica di memoria, generazione di numeri casuali
  - ▶ `<cstdio>`: operazioni di input/output
  - ▶ `<cstring>`: elaborazione di stringhe
  - ▶ `<cmath>`: funzioni matematiche (pow sqrt log)
- ▶ E esclusive per il C++
  - ▶ `<iostream>`: operazioni di input/output
  - ▶ `<iomanip>`: manipolatori per la formattazione dei dati
  - ▶ `<fstream>`: operazioni di input/output dei file su disco
  - ▶ `<string>`: elaborazione di stringhe

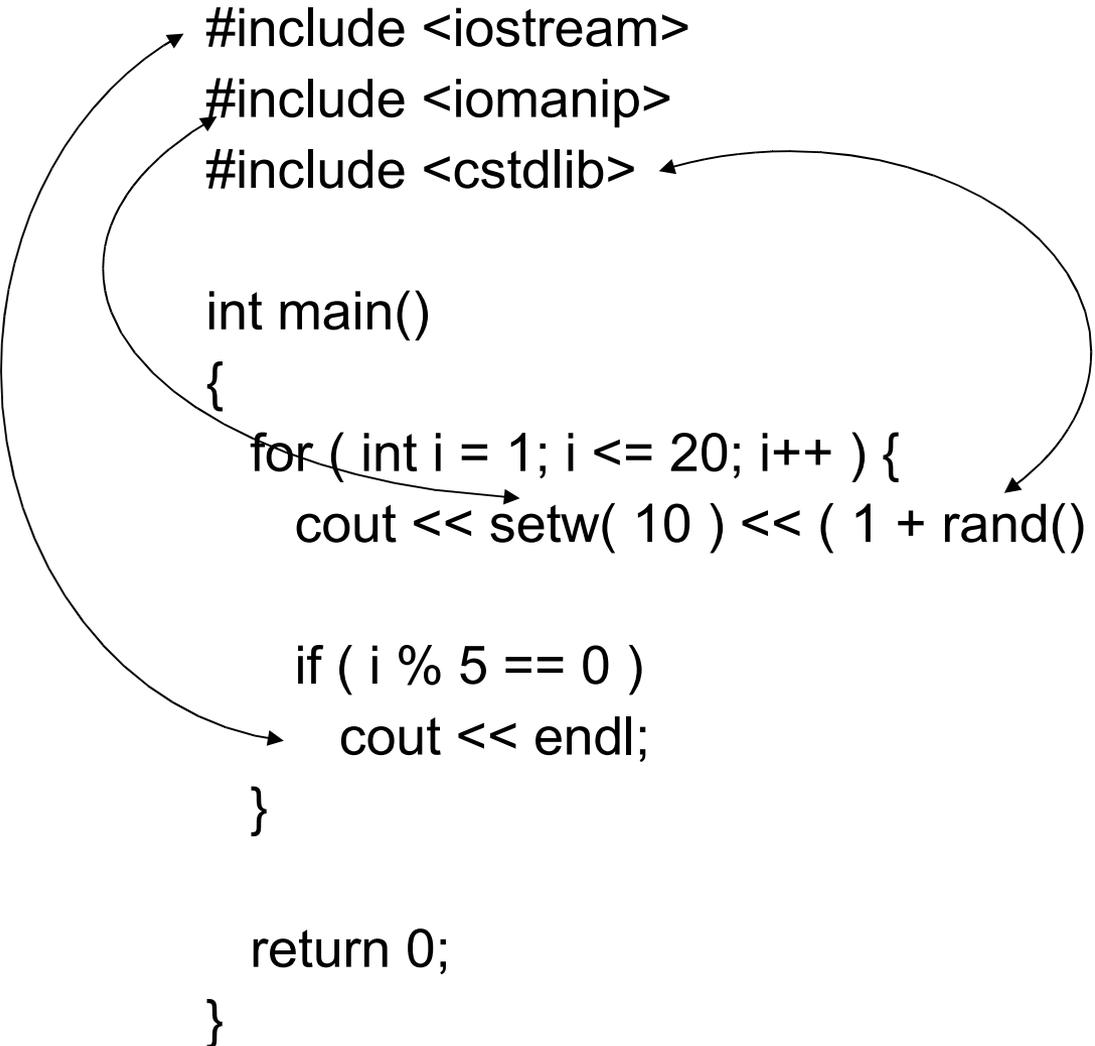
# Esempio

```
#include <iostream>
#include <iomanip>
#include <cstdlib>

int main()
{
    for ( int i = 1; i <= 20; i++ ) {
        cout << setw( 10 ) << ( 1 + rand() % 6 );

        if ( i % 5 == 0 )
            cout << endl;
    }

    return 0;
}
```



The diagram illustrates the dependencies between the code lines. Arrows point from the code to the headers it uses: from the first line of the for loop to `<iostream>`, from the second line to `<iomanip>`, and from the third line to `<cstdlib>`. Additionally, an arrow points from the `return 0;` line back to the `int main()` line, indicating the function's return path.

# Dichiarazione variabili locali

- ▶ In C tutte le variabili devono essere dichiarate all'**inizio** del blocco che le contiene
- ▶ In C++ le variabili possono essere dichiarate in un **punto qualsiasi** del blocco che le contiene e referenziate da lì in poi

# Esempio di dichiarazione di variabili

```
main() {  
    int a;  
    cout<<"Immettere un numero:";  
    cin>>a;  
    cout<<"Immettere una stringa:";  
    cin>>str; //NON E' STATA ANCORA DICHIARATA  
  
    char str[100]; //DICHIARAZIONE  
    cout<<"Immettere una stringa:";  
    cin>>str;  
}
```

# Esempio di dichiarazione di variabili

```
main() {  
    ...  
    for(int i=0;i<10;i++) cout<<i;  
  
    cout<<i;  
    for(int i=0;i<10;i++) cout<<i;  
    ...  
}
```

- ▶ Nota: La visibilità per la variabile **i** è limitata all'interno del blocco istruzioni del *for*

# Memoria

- ▶ Esistono 2 tipi di memoria: stack e heap
- ▶ Memoria Stack:
  - ▶ Viene gestita automaticamente (allocazione e deallocazione)
  - ▶ Viene resa disponibile ad ogni invocazione di una funzione per le variabili locali
- ▶ Memoria Heap:
  - ▶ Viene richiesta dal programmatore (al gestore dello heap) sia in fase di allocazione che deallocazione
  - ▶ E' permanente rispetto agli ambiti di visibilita' di blocchi e funzioni

# Allocazione di memoria

- ▶ Esistono 3 tipi di allocazione della memoria per le variabili
  - ▶ Statica
  - ▶ Automatica
  - ▶ Dinamica
- ▶ *Allocazione di memoria=assegnazione della proprietà di una area di memoria ad una variabile di un programma*
- ▶ **Memorizzazione Statica:** la memoria e' allocata al tempo della compilazione (prima della esecuzione del programma)
- ▶ **Memorizzazione Automatica:** la memoria e' allocata sullo stack quando viene dichiarata la variabile e deallocata quando termina il suo ambito di visibilita'
- ▶ **Memorizzazione Dinamica:** la memoria e' allocata sullo heap facendone esplicitamente richiesta (tramite operatore new) e viene deallocata esplicitamente (tramite operatore delete)

# Allocazione dinamica della memoria

- ▶ Durante l'esecuzione (run-time) **non** è possibile aggiungere né variabili locali né globali
- ▶ Il programmatore non può decidere all'interno di un programma quando allocare una variabile statica o automatica: non ci sono istruzioni per farlo
- ▶ Quando non è possibile prevedere la dimensione della memoria richiesta in un programma al momento della compilazione o non è conveniente farlo si utilizza il metodo di allocazione **dinamica**

# Allocazione dinamica di memoria

- ▶ Le parole chiave per allocare/deallocare memoria dinamica
  - ▶ In C: malloc e free
  - ▶ In C++: new e delete

# Allocazione dinamica di memoria

```
NomeTipo *var_ptr;
```

in C:

```
var_ptr= (NomeTipo*)malloc(sizeof(NomeTipo));  
free(var_ptr);
```

Nota:

- ▶ *malloc* restituisce un puntatore di tipo void e deve essere convertito (esplicitamente o implicitamente) in un puntatore di tipo opportuno
- ▶ L'uso di *sizeof()* permette la portabilità del codice

# Allocazione dinamica di memoria

```
NomeTipo *var_ptr;
```

in C++:

```
var_ptr= new NomeTipo;  
delete var_ptr;
```

- ▶ new crea un oggetto della dimensione appropriata senza dover usare sizeof
- ▶ new può inizializzare un oggetto (vedremo in seguito)

```
double * var_ptr=new double(3.1415);
```

- ▶ new invoca il costruttore di un oggetto

# Allocazione dinamica: array

```
NomeTipo *array_ptr;  
int dim=100;
```

▶ in C:

```
array_ptr=(NomeTipo*) malloc(sizeof(NomeTipo)*dim) ;  
free(array_ptr) ;
```

▶ in C++:

```
array_ptr= new NomeTipo[dim] ;  
delete [] array_ptr;
```

• **Non** si può inizializzare un array

```
double * array_ptr=new double[100] (3.1415) ;
```

# Attenzione!

- ▶ Non mischiare allocazioni con `new` e `malloc`
- ▶ Allocazioni fatte con `new` non possono essere deallocate con `free` e viceversa
- ▶ deallocare un array con `delete`  
invece di `delete [ ]` genera un errore

# Passaggio di parametri

- ▶ In C++ ci sono 3 modi per passare parametri ad una funzione:
  - ▶ per valore
  - ▶ per riferimento con puntatori
  - ▶ per riferimento con alias

# Passaggio di parametri: per valore

- ▶ Si crea una **copia locale** del parametro passato
- ▶ La modifica della copia locale **non ha influenza** sul parametro originale
- ▶ E' utile per isolare ciò che fa la funzione dal resto del programma

# Passaggio di parametri: per valore

```
#include <iostream>

int f(int);

main() {
    int dato=2;
    cout<<f(dato)<<endl; // stampa 9
    cout<<dato<<endl; //stampa 2
}

int f(int num){
    num=num+1;
    return num*num;
}
```

# Passaggio di parametri: con puntatori

- ▶ Si accede al dato passato
- ▶ La modifica all'interno della funzione **cambia** il parametro originale
- ▶ La sintassi cambia (\*num, &dat)
- ▶ E' utile quando si vuole passare ad una funzione degli oggetti di grandi dimensioni (es. vettori, strutture)

# Passaggio di parametri: con puntatori

```
#include <iostream>

int f(int *);

main() {
    int dato=2;
    cout<<f(&dato)<<endl; // stampa 9
    cout<<dato<<endl; //stampa 3
}

int f(int * num) {
    (*num)=(*num)+1;
    return (*num)*(*num);
}
```

# Passaggio di parametri: con alias

- ▶ Si accede al dato passato
- ▶ La modifica all'interno della funzione **cambia** il parametro originale
- ▶ Modifica della sintassi **solo** nel prototipo e nell'intestazione della funzione
- ▶ E' utile quando si vuole passare ad una funzione degli oggetti di grandi dimensioni (es. vettori, strutture)

# Passaggio di parametri: con alias

```
#include <iostream>

int f(int &);

main() {
    int dato=2;
    cout<<f(dato)<<endl; //stampa 9
    cout<<dato<<endl; //stampa 3
}

int f(int & num) {
    num=num+1;
    return num*num;
}
```

# Passaggio di parametri: con alias

- ▶ L'alias è un nome diverso per una variabile già istanziata:

```
int x=1;
int & y=x;
cout<<x; //stampa 1
y=2;
cout<<x; //stampa 2
```

- ▶ NOTA: per il compilatore il passaggio per alias è **equivalente** al passaggio per riferimento con puntatori *const* (vedremo poi cosa significa *const*), risulta solo semplificata la notazione