Const, friend, static,this

Sommario

- Lo speficatore **const** per gli oggetti e le funzioni membro
- Le funzioni **friend**
- I membri **static** di una classe
- Il puntatore **this**

Oggetti `const`

- Alcuni oggetti devono poter essere modificati, altri no
- E' possibile dichiarare un oggetto costante (immodificabile) con

```
const Tempo mid(12,0);
```

- Ad es. l'oggetto *mid* che rappresenta mezzogiorno può essere utilizzato come costante in operazioni di confronto
- Un oggetto costante può essere solo inizializzato
- Qualsiasi tentativo di modificare un oggetto *const* genera un errore in fase di compilazione

Oggetti `const`

- Utilizzare oggetti *const* è una buona norma di ingegneria del software (codice più sicuro)
- Utilizzare oggetti *const* migliora l' efficienza: i compilatori riescono a ottimizzare il codice che opera su oggetti const

Dati membro `const`

- Anche i dati membro possono essere di tipo costante
- Un dato membro di una classe dichiarato come const non può essere usato a sinistra di operazioni di assegnamento
- Un dato membro const può solo essere inizializzato
- Un costruttore non può pertanto usare un operazione di assegnamento per attribuire un valore ad un dato const!

```
class Data{
public:
Data(int usr_dat=0){
    dat=usr_dat;  //ERRORE
}
const int dat;
};
```

Dati membro `const`

- Come fare una inizializzazione in una classe?
- Un costruttore deve essere modificato nel seguente modo:

```
class Data{
public:
   Data(int usr_dat=0):dat(usr_dat){}
   const int dat;
};
```

- La notazione `:dat(usr_dat)` inizializza dat con usr_dat
- **NOTA:** Non inizializzare un dato membro const è un errore

Dati membro `const`

- Se è necessario inizializzare più di una variabile lo si fa separando tramite virgole le varie inizializzazioni:

```
class Data{
public:
  Data(int usr_dat1=0, int usr_dat2=0)
  :dat1(usr_dat1), dat2(usr_dat2){}

  const int dat1;
  const int dat2;
};
```

Funzioni membro `const`

- Accedere alle funzioni membro per un oggetto dichiarato const potrebbe significare permettere che i dati membro (che devono rimanere costanti) siano alterati

- Il compilatore pertanto impedisce l'accesso a funzioni membro const

- ma come utilizzare allora funzioni membro che (correttamente) non alterano i dati membro di un oggetto costante?

Funzioni membro `const`

- Non è possibile chiamare funzioni membro per oggetti const
- *a meno che* le funzioni non siano dichiarate const

- una funzione membro si dichiara const sia nel prototipo che nella definizione secondo la seguente sintassi:

```
//nella dichiarazione di classe
int restituisci_ora() const;

//nella definizione
int Tempo::restituisci_ora() const {
    return ora;
}
```

Funzioni membro `const`

- E' un errore se una funzione membro costante tenta di modificare un qualsiasi dato membro
- cioè: una funzione membro costante non può modificare dati membro di alcun tipo
- ..ma internamente alla funzione possono essere dichiarate *e modificate* variabili locali

```
int Tempo::restituisci_ora() const {
    int a;
    a=ora+2;
    return a;
}
```

Funzioni membro `const`

- Non è possibile invocare funzioni const che a loro volto chiamino funzioni membro non-const
  - una funzione const puo' usare solo altre funzioni const
- Non si deve dichiarare const un costruttore o distruttore!
  - non e' di nessuna utilita' un costruttore o un distruttore che non puo' alterare i dati !
- Solo il costruttore di un oggetto costante può chiamare altre funzioni non costanti
  - un costruttore ha una deroga speciale: lui lavora in un momento dell'esistenza dei dati simile alla inizializzazione e puo' pertanto usare ad es. funzioni set
- Nota: questo caso è differente dal inizializzare dei dati membro costanti; l'inizializzazione ha una sintassi speciale, in questo caso si usano funzioni nel corpo del costruttore

Esempio

```
Class Dat{
   public:
   Dat(int usr_dat=0){set(usr_dat);}
   void set(int usr_dat){dat=usr_dat;}
   void print()const{cout<<dat;}
   private:
   int dat;
};

void main(){
   const Dat my_dat(12); //OK
   my_dat.set(2); //ERROR
   my_dat.print();//OK
}
```

Le funzioni `friend`

- Lo specificatore friend permette di definire funzioni al di fuori del campo di visibilità di una classe che possono accedere ai membri privati di tale classe

- Una classe B può essere dichiarata friend di una classe A
  - in pratica tutti i dati membro e le funzioni membro private della classe A che concede l'amicizia a B diventano accessibili per le funzioni membro di B
  - in genere non si dichiarano classi friend ma si ricorre al concetto di ereditarietà (vedremo)

Le funzioni `friend`

- Dichiarazione:
  - per dichiarare una funzione (classe) come friend di una classe basta precedere con la parola *friend* il prototipo della funzione (classe) nella dichiarazione della classe
  - la definizione della funzione è di tipo usuale (non compare cioe' la parola friend)

A cosa servono le funzioni `friend`

- Quando non e' possibile modificare una classe
- Quando si vuole creare metodi comuni che operino su classi diverse
- Esempi di uso frequente:
  - overloading di operatori
  - iteratori

Esempio

```
#include <iostream>
class Count {
   friend void setX( Count &, int ); // friend declaration
public:
   Count() { x = 0; }  // constructor
   void print() const { cout << x << endl; } //output
private:
   int x;  // data member
};

// Can modify private data of Count because
// setX is declared as a friend function of Count
void setX( Count &c, int val ){
   c.x = val;  // legal: setX is a friend of Count
}
```

```
int main()
{
   Count counter;

   cout << "counter.x after instantiation: ";
   counter.print();
   cout << "counter.x after call to setX friend function: ";
   setX( counter, 8 );  // set x with a friend
   counter.print();
   return 0;
}
```

Le funzioni `friend`

- *L'amicizia è concessa non presa:*

  all'interno della classe si deve dichiarare esplicitamente quali funzioni sono considerate friend
- non è possibile dichiarare una funzione friend fuori da una dichiarazione di classe
- la dichiarazione friend non costituisce una dichiarazione della funzione, questa deve essere dichiarata nel modo ordinario
- Per accedere ad un membro di un oggetto la funzione deve ricevere l'oggetto come parametro (in genere per riferimento)

Il puntatore **`this`**

- Ogni oggetto ha accesso al proprio indirizzo tramite il puntatore predefinito **`this:`**

```
class Dat{
public:
   Dat(int a){x=a;}
   void print() const {
        cout<< x <<endl;
        cout<< this->x <<endl;
        cout<< (*this).x <<endl;
   }
private:
   int x;
}
```

Il puntatore this

- Un uso importante del puntatore this è di consentire la chiamata a cascata di funzioni membro
- Questo si realizza con una funzione membro che restituisce un alias a se stessa

Esempio

```
class Tempo{
public:
   Tempo(int h=0, int m=0){set(h,m);}
   void set(int h, int m){setH(h);setM(m);}
   Tempo & setH(int h);
   Tempo & setM(int m);
   void print(){cout<<ora<<":"<<min;}
private:
   int ora, min;
};
```

```
Tempo & Tempo::setH(int h){
   ora=h;
   return *this;
}
Tempo & Tempo::setM(int m){
   min=m;
   return *this;
}
void main{
   Tempo t(10,20);
   t.setH(11).setM(15);
   t.setH(2).print(); //OK
   t.print().setM(2); //ERROR
}
```

Spiegazione

- Ecco cosa accade quando viene eseguita l'istruzione

```
t.setH(11).setM(15);
```

- l'operatore "." associa da sinistra a destra

```
(t.setH(11)).setM(15);
```

- viene creato un oggetto temporaneo senza nome per contenere il risultato restituito da `t.setH(11)`
- tale oggetto è un alias di t che adesso ha il dato membro *ora* modificato
- di tale oggetto (t) viene invocato il metodo

```
t.setM(15);
```

- che nuovamente restituisce un oggetto temporaneo che è un alias di t

Ma quando un oggetto temporaneo viene distrutto non si invoca il distruttore?

- Quando l'alias temporaneo si distrugge non viene invocato alcun distruttore per la copia originale (si consideri il concetto di puntatore)
- Un alias è un puntatore (cambia solo la sintassi per riferirsi ad esso ed al dato puntato)
- pertanto quando viene deallocato si distrugge la variabile che conteneva l'indirizzo dell'oggetto
- solo quando si deve distruggere l'oggetto si richiama il distruttore

Spiegazione

- Nel caso di `t.print().setM(2);` invece si ha:
  `(t.print()).setM(2);`
- ma `t.print()` restituisce un tipo void che non possiede alcuna funzione membro, tantomeno la funzione setM( )

Membri static di una classe

- Ogni oggetto possiede una copia dei dati membro della sua classe
- In alcuni casi si può volere che un dato sia condiviso da tutti gli oggetti appartenenti alla classe
- Per realizzare questo si dichiara il dato membro come *static*

- Un dato static e' una specie di variabile globale ma con visibilità e accesso limitato agli oggetti di una unica specifica classe

Membri static di una classe

- Un membro static esiste anche quando non è stato ancora istanziato alcun oggetto della classe
- per accedervi:
  - se il membro è *public* si usa: NomeClasse::nomeVariabile
  - se il membro è *private* si devono usare le funzioni public (o friend) di un oggetto istanziato, oppure tramite una funzione membro static

Funzioni membro static

- Possono essere dichiarati static sia i dati membro che le funzioni membro
- Una funzione static può accedere solo ai membri static di una classe
  - infatti si puo' chiamare una funzione static anche se non e' stato istanziato alcun oggetto e dunque quando ancora non esistono i dati membro

Esempio

```
#ifndef EMPLOY1_H
#define EMPLOY1_H
class Employee {
public:
   Employee( const char*, const char* );  // constructor
   ~Employee();                           // destructor
   const char * getFirstName() const;  // return first name
   const char * getLastName() const;   // return last name
   static int getCount();  // return # objects instantiated

private:
   char *firstName;
   char *lastName;
   // static data member
   static int count;  // number of objects instantiated
};
#endif
```

Esempio

```
// Member function definitions for class Employee
#include <iostream>
#include <cstring>
#include <cassert>
#include "employ1.h"

int Employee::count = 0;
int Employee::getCount() { return count; }

Employee::Employee( const char *first, const char *last ){
   firstName = new char[ strlen( first ) + 1 ];
   assert( firstName != 0 );   // ensure memory allocated
   strcpy( firstName, first );
   lastName = new char[ strlen( last ) + 1 ];
   assert( lastName != 0 );    // ensure memory allocated
   strcpy( lastName, last );
   ++count;  // increment static count of employees
   cout << "Employee constructor for " << firstName
        << ' ' << lastName << " called." << endl;
}
```

```
Employee::~Employee()
{
   cout << "~Employee() called for " << firstName
        << ' ' << lastName << endl;
   delete [] firstName;  // recapture memory
   delete [] lastName;   // recapture memory
   --count;  // decrement static count of employees
}

const char * Employee::getFirstName() const{
   // Const before return type prevents client from modifying
   // private data. Client should copy returned string before
   // destructor deletes storage to prevent undefined pointer
   return firstName;
}

const char * Employee::getLastName() const
{return lastName;}
```

Esempio

```
#include <iostream>
#include "employ1.h"
int main()
{
   cout << "Number of employees before instantiation is "
        << Employee::getCount() << endl;   // use class name
   Employee *e1Ptr = new Employee( "Susan", "Baker" );
   Employee *e2Ptr = new Employee( "Robert", "Jones" );
   cout << "Number of employees after instantiation is "
        << e1Ptr->getCount();
   cout << "\n\nEmployee 1: "
        << e1Ptr->getFirstName()
        << " " << e1Ptr->getLastName()
        << "\nEmployee 2: "
        << e2Ptr->getFirstName()
        << " " << e2Ptr->getLastName() << "\n\n";
```

```
   delete e1Ptr;   // recapture memory
   e1Ptr = 0;
   cout << "Number of employees after deletion is "
        << Employee::getCount() << endl;

   delete e2Ptr;   // recapture memory
   e2Ptr = 0;
   cout << "Number of employees after deletion is "
        << Employee::getCount() << endl;

   return 0;
}
```

Note

- La macro assert è definita in <cassert>
- assert(cond); verifica la condizione *cond*
  - se è vera il programma continua
  - se è falsa viene chiamata la funzione abort() che termina l'esecuzione del programma e indica nel messaggio di errore la linea di codice in cui si è verificata la falsità di *cond*
- assert è utile in fase di debugging
- una volta terminata la fase di debugging basta aggiungere #define NDEBUG perché il preprocessore elimini il codice relativo a tutti gli utilizzi di assert