Corso di Laboratorio 2 – Programmazione C++

Silvia Arcelli

24 Ottobre 2014

Una funzione membro importante: il costruttore

- In logica OOP l'accesso allo stato interno di un'istanza deve avvenire solo attraverso i **metodi pubblici** della classe. Per dichiarare ed inizializzare congiuntamente una istanza di una classe si utilizza un metodo particolare (che può essere anche overloaded) detto **costruttore**, che ha lo stesso nome della classe.
- Un costruttore puo` avere un qualsiasi numero di parametri, ma non restituisce mai alcun tipo (neanche void). Esempio di costruttore parametrico:

```
Class Complex {
public:
{ // costruttore
Complex(float a, float b)
 {  Re = a; Im = b; }
/* altre funzioni membro */
private:
float Re; // Parte reale
float Im; // Parte immaginaria
};
```

Nel main:

```
Complex C(3.5, 4.2); //oppure:
Complex C = Complex(3.5, 4.2);
```

Dichiarazione e inizializzazione di un oggetto C di tipo Complex

Costruttori e lista di inizializzazione

- In alcuni casi, alcune operazioni possono richiedere che tutti o parte degli attributi dell'oggetto siano **inizializzati prima** che incominci l'esecuzione del corpo del costruttore; in questi casi si utilizza la **lista di inizializzazione** (ad es. nel caso di membri const, inizializzazione della parte di base nel costruttore della derivata).

- La lista di inizializzazione è una caratteristica propria dei costruttori e appare sempre tra la lista di argomenti del costruttore e il suo corpo, preceduta da un ":":

```
class Complex
{ public:
Complex(float a, float b) : Re(a), Im(b) { }
 /* ... */
private: float Re; float Im;
};
```

- la notazione *Attributo(< Espressione >)* indica al compilatore che *Attributo* deve memorizzare il valore fornito da *Espressione*; *Espressione* puo` essere anche qualcosa di complesso come la chiamata ad una funzione.

Tipi di Costruttori

- Una classe può possedere più costruttori, cioè i costruttori possono essere overloaded (funzioni con lo stesso nome ma segnatura diversa fanno cose differenti);

- Non necessariamente devono essere tutti pubblici. Tuttavia, in generale, deve esserci almeno un costruttore pubblico (a parte il caso particolari di classi di tipo singleton)

- Un singleton è una classe strutturata in modo che si possa avere un'unica istanza di quella classe (classi "manager").

Tipi di Costruttori

Inoltre, alcuni costruttori assumono un significato speciale:

- il costruttore di default:

  ClassName::ClassName();

- il costruttore di copia:

  ClassName::ClassName(const ClassName& X);

- altri costruttori con un solo argomento;

  ClassName::ClassName(Type Arg);

Il Costruttore di Default

- Il costruttore di default non ha argomenti ed e` particolare, in quanto e` quello che il compilatore chiama quando il programmatore scrive:

```
class Complex
{ public:
Complex( ){Re=0,Im=0;}
Complex(float a, float b) : Re(a), Im(b)
{ }
 /* ... */
private: float Re; float Im;
};
```

Dichiarazione/inizializzazione con costruttore di default

```
Complex C;
```

- Se non esplicitamente implementato dal programmatore, Il costruttore di default è fornito automaticamente dal compilatore (che però non dà garanzie sul contenuto degli attributi dell'oggetto).

Il Costruttore di Default

- Se non si vuole che questo avvenga, occorre definirne esplicitamente almeno uno (anche se non è quello di default).

- Una volta definito un qualsiasi altro tipo di costruttore, va definito esplicitamente anche il costruttore di default se si vuole continuare a usarlo

- Una dichiarazione di questo tipo:

```
Complex C[100];
```

- Senza la definizione esplicita del costruttore di default non compila

- Consigliabile farlo nel caso di ereditarietà fra classi

esempio

```
class Complex
{ public:
Complex(float a, float b) : Re(a), Im(b)
{ }
 /* ... */
private: float Re; float Im;
};
```

Il Costruttore di Copia

Il costruttore di copia invece viene sempre invocato quando:

- un nuovo oggetto è inizializzato in base al contenuto di un altro oggetto;

```
Trace  B;
Trace D = B
Trace C(B);
```

- Passaggio per valore ad una funzione

```
Trace  B;
WriteToFile(B);
void WriteToFile(Trace O){/*..*/}
```

- Restituzione di un oggetto temporaneo

```
Trace  B;
Trace A=Transform(B);
Trace Transform(Trace O){/*..*/}
```

Il Costruttore di Copia

```
class Complex
{ public:
Complex( ){}
Complex(const Complex & A) : Re(A.Re), Im(A.Im) { } ;
 /* ... */
private: float Re; float Im;
};
```

Inizializzazione con costruttore di copia (non è assegnamento):

```
Complex  D = B;
```

- L'argomento del costruttore di copia deve essere sempre un riferimento
- Se il programmatore non definisce un costruttore di copia, il compilatore ne fornisce uno di default ed esegue una copia bit a bit degli attributi.
- quando una classe contiene puntatori e` bene definirlo esplicitamente per evitare problemi di condivisione di aree di memoria (esempio)

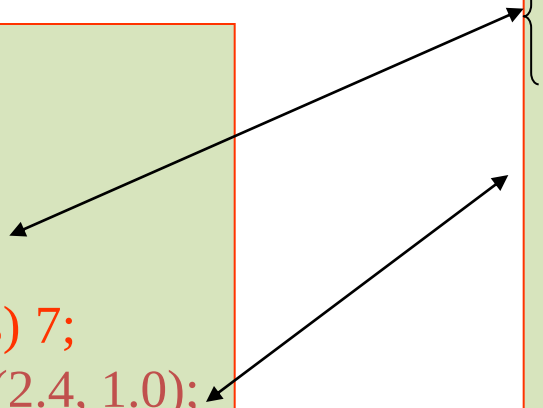
Costruttori con un solo argomento

Realizzano un operatore di conversione tra l'argomento di input e l'oggetto di cui si fa l'istanza:

```
int main(int, char* [])
{
MyClass A(1);
MyClass B = 5.5;
MyClass D = (MyClass) 7;
MyClass C = Complex(2.4, 1.0);
return 0;
}
```

```
class MyClass {
public:
MyClass(int);
MyClass(double);

MyClass(Complex);
 /* ... */
private:
/* ... */
};
```



Il distruttore

- Un distruttore e` un metodo che non riceve parametri, non ritorna alcun tipo e ha lo stesso nome della classe preceduto da una ~ (tilde): il suo compito è permettere una corretta deallocazione delle risorse occupate dall'oggetto, ed e` chiamato implicitamente dal compilatore quando un oggetto termina il suo ciclo di vita (oppure quando un oggetto allocato con new viene deallocato con delete).

```
Class Complex {
public:
~ Complex(){}
Complex(float a, float b) { // costruttore! Re = a; Im = b; }
/* altre funzioni membro */
private:
float Re; // Parte reale
float Im; // Parte immaginaria };
```

- Come nel caso del costruttore, se il programmatore non lo definisce, il compilatore ne fornisce uno di default

Il distruttore

- Poiche` il distruttore fornito dal compilatore non tiene conto di aree di memoria allocate tramite membri puntatore, e` sempre necessario definirlo esplicitamente quando sono definiti attributi di tipo puntatore:

```
class MyClass {
public:
~ MyClass(){ delete [] ptr;}
MyClass(int n):ptr(0),size(n){ ptr=new int[size]; }
/* altre funzioni membro */
private:
int *ptr; // membro puntatore
int size; };
```

- In tutti gli altri casi, il distruttore di default e` in generale piu` che sufficiente e non occorre scriverlo.

- Si noti che poiche` un distruttore non possiede argomenti, non e` possibile eseguirne l'overloading; ogni classe cioe` possiede sempre e solo un unico distruttore

Attributi Static

In alcuni casi e` desiderabile che il valore di alcuni attributi sia condiviso fra tutte le istanze (nell'esempio un contatore che conta il numero di istanze). In questo caso lo si dichiara static:

```
class MyClass {
public:
MyClass() {++Counter;}
~MyClass() { --Counter;}
/* ... */
private:
 static int Counter;
/* ... */ };
```

•Gli attributi static possono essere considerati come elementi propri della classe, non dell'istanza. Tutti i metodi (costruttore compreso) possono accedere sia in scrittura che in lettura all'attributo, ma non lo si puo` inizializzare tramite un costruttore (altrimenti sarebbe proprio dell'istanza).

Attributi Static

- In altre parole, una inizializzazione nel costruttoredel tipo:

```
MyClass::MyClass() : Counter(0) {/* ... */ }
```

è errata. Una inizializzazione eseguita tramite costruttore verrebbe ripetuta ogni volta che si crea un'istanza della classe.

- L'inizializzazione di un attributo static va fatta successivamente alla sua dichiarazione e al di fuori della definizione di classe (questo fa sì che l'attributo sia inizializzato prima della creazione di una qualsiasi istanza):

```
class MyClass {
public: MyClass();
 /* ... */
private:
static int Counter;
/* ... */ };
int MyClass::Counter = 0;
```

Metodi Static

- Oltre ad attributi static e` possibile avere anche metodi static; la keyword static in questo caso vincola il metodo ad accedere solo agli attributi statici della classe (un accesso ad un attributo non static costituisce un errore):

```
class MyClass {
 public:
static int GetCounterValue();
/* ... */
private:
static int Counter;
 /* ... */ };
int MyClass::Counter = 0;
int MyClass::GetCounterValue() { return Counter; }
```

- Si noti che nella definizione della funzione membro statica la keyword static non e` stata ripetuta, essa va posta solo nella dichiarazione
- In questo esempio avrei anche potuto chiedere di rendere il metodo inline

Metodi Static

- Tuttavia agli attributi statici si può liberamente accedere usando qualunque metodo della classe, anche non static. Che vantaggio si ha nel dichiarare static un metodo che accede solo ad attributi statici?

  - minor overhead di chiamata: i metodi non statici, per sapere a quale oggetto devono riferire, ricevono dal compilatore un puntatore all'istanza di classe per la quale il metodo e` stato chiamato; i metodi static per loro natura non hanno bisogno di tale parametro e quindi non comportano overhead di chiamata a funzione;

  - Per questa ragione, le funzioni membro statiche non possono essere qualificate come funzioni membro const....

  - Non e` possibile dichiarare static un costruttore o un distruttore (anche se la classe avesse solo attributi statici).

Singleton

```
class   MySingleton {
public:
~MySingleton() {delete ptr;};
 static MySingleton* instance() {
if (ptr == 0) ptr = new MySingleton();
return ptr; }
/* altri metodi*/
void method(){/*--*/}
private:
MySingleton() { };
static MySingleton* ptr;

};
// initialize static pointer outside
MySingleton* MySingleton::ptr = 0;
```

Punti essenzali:

-Costruttore privato, obbligatorio dichiararlo altrimenti ho il costruttore di default (pubblico)

-metodo statico e attributo statico che consentono di costruire ed accedere all'oggetto

-questa struttura assicura la creazione di una sola istanza

Nel main:

```
MySingleton *ptr=MySingleton::instance();
ptr->method() ;
```

Metodi Static

- Inoltre, i metodi statici, oltre a poter essere chiamati come un qualunque altro metodo, associandoli ad un oggetto , possono essere chiamati come una funzione senza necessita` di associarli ad una particolare istanza, ricorrendo solo all'operatore di risoluzione di ambito (::) proprio perchè non hanno "bisogno" del puntatore this.

```
MyClass Obj;
int Var1 = Obj.GetCounterValue(); // Ok!
int Var2 = MyClass::GetCounterValue(); // Ok
```

Attributi Const

Oltre ad attributi di tipo static, e` possibile avere attributi const. Che significato ha la keyword const relativamente agli attributi di una classe?

- viene allocato per ogni istanza come un normale attributo, tuttavia il valore che esso assume per ogni istanza viene stabilito una volta per tutte all'atto della creazione dell'istanza stessa.

- Questo attributo non potra` mai cambiare durante la vita dell'oggetto. E' const relativamente a quella istanza, istanze diverse possono liberamente avere valori diversi degli attributi const

Attributi Const

- Il valore di un attributo const va attribuito esclusivamente tramite la lista di inizializzazione del costruttore:

```
class MyClass {
public:
MyClass(int a, float b) : ConstMember(a), NonConstMember(b) { ...}
/* ... */
private:
const int ConstMember;
float NonConstMember;
 };
```

```
MyClass(int a, float b) : ConstMember(a), {NonConstMember=b;}
```

OK

NO

```
MyClass(int a, float b) : NonConstMember(b) { ConstMember=a;}
```

Metodi Const

- E' anche possibile avere funzioni membro const. Dichiarando un metodo const la funzione membro si impegna a non modificare nessun attributo della classe. Di fatto, tratta il puntatore this come un puntatore a costante.

- Sembrerebbe solo una questione formale, ma ha una concreta utilità:

  I metodi const sono gli unici a poter essere eseguiti su istanze costanti (che per loro natura non possono essere modificate).

- Quindi è buona prassi ricordarsi di definire const qualunque metodo che quando invocato non modifica gli attributi dell'oggetto.

- Notare che metodi con la stessa segnatura ma che differiscono solo per la specifica const possono essere *"overloaded"*

Metodi Const

- Per dichiarare una funzione membro const e` necessario far seguire alla signature (sia in dichiarazione che definizione) del metodo la keyword const:

```
class MyClass {
public:
MyClass(int * a, float b) : ConstMember(a),NonConstMember(b) {}
const int *GetConstMember() const { return ConstMember; }
float GetNonConstMember() const { return NonConstMember; }
void  SetNonConstMember(float b) { NonConstMember = b; }
private:
const int * ConstMember;
float NonConstMember;
};
```

```
int main( ) {
int init=3;
MyClass A(&init, 5.3);
const MyClass B(&init, 3.2);
A.GetConstMember(); // Ok!
B.GetNonConstMember(); // Ok!
A.SetNonConstMember(1.2); // Ok!
B.SetNonConstMember(1.7); // Errore!
return 0; }
```

Metodi Const

•Se la funzione membro GetConstMember() fosse stata definita fuori dalla definizione di classe, avremmo dovuto nuovamente esplicitare il const (cosa che non avviene con le funzioni membro static).

```
class MyClass {
 public:
MyClass(int a, float b) : ConstMember(a), NonConstMember(b) {}
int GetConstMember() const;
/* ... */ };
int MyClass::GetConstMember() const { return ConstMember;}
```

- Come per il caso di static, non e` possibile definire costruttori e distruttori const (sebbene essi vengano utilizzati per costruire e distruggere anche le istanze costanti).

Metodi Const

- Poiche` gli attributi const sono attributi a sola lettura che vanno inizializzati tramite lista di inizializzazione, per allocare un array statico il seguente codice non può funzionare:

```
class Array {
public:
Array(const int b):Size(b){}
/* ... */
private:
const int Size;
char String[Size]; // Errore!
};
```

Attributi Static Const

- Questo perche` non si puo` stabilire a tempo di compilazione il valore di *Size*.Una soluzione possibile in questo caso specifico è utilizzare la keyword enum, che consente di associare agli enumeratori fra graffe dei valori costanti interi. In particolare, si può usare un enumeratore anonimo`:

```
class Array {
 public:
 /* ... */
private:
enum { Size = 20 };
char String[Size];
// Ok! };
```

- Questa soluzione ha la limitazione di applicarsi solo a costanti intere (in questo caso ok)

Attributi Static Const

- Una soluzione definitiva è data utilizzando contemporaneamente le keyword static e const:

```
class Array {
 public: /* ... *
/ private:
 static const int Size = 20;
char String[Size]; // Ok! };
```

- Essendo static, Size viene inizializzata prima della creazione di una qualunque istanza della classe, prefissato gia` a compile time. Notare che in questo caso (const int) si può (si deve) inizializzare l'attributo Size all'interno della definizione, anche se Size è un attributo statico

Quesiti Esame

- Quali di queste linee, all'interno della definizione di una classe, NON causerà un errore in compilazione:
  A) static const int a;
  B) static float a;
  C) int a = 3;
  D) static float a = 1;

Quesiti Esame

Se si volesse tenere traccia del numero di istanze create all'interno di un programma per una data classe, quale delle seguenti opzioni potrebbe essere utilizzata?

A) Si potrebbe definire un attributo della classe, di tipo "const", da inizializzare in modo progressivo nel costruttore della classe

B) Si potrebbe definire un attributo della classe, non "const", da inizializzare in modo progressivo nel costruttore della classe

C) Si potrebbe definire un attributo della classe "static" inizializzato a zero e incrementato in modo progressivo nel costruttore della classe

D) Nessuna delle precedenti soluzioni permette di contare le istanze

Quesiti Esame

Una classe ha un attibuto statico privato cosi' definito:

static int fNinstancies;

qual e' il modo corretto di implementare il corrispondente getter dell'attributo, che si desidera poter invocare anche senza doversi riferire ad una istanza della classe?

A) int GetNinstancies(){return fNinstancies;}
B) static int GetNinstancies(){return fNinstancies;}
C) static int GetNinstancies() const {return fNinstancies;}
D) const int GetNinstancies() const {return fNinstancies;}

Quesiti Esame

Quali delle seguenti definizioni all'interno di una classe comportano allocazione di memoria contestualmente alla creazione di una istanza della classe:

A) const int a;

B) int a;

C) static int a;

D) static const int a = 3;

Quesiti Esame

Dato un metodo "static" di una classe, quale operazione è sicuramente vietata al suo interno:

A) La modifica degli argomenti passati al metodo
B) L'utilizzo di cout e cin
C) L'utilizzo in lettura dei data member della classe
D) La modifica dei membri statici della classe