Programmazione a oggetti

Introduzione alla programmazione a oggetti

A.A. 2020/2021
Francesco Fontanella

Programmazione Modulare

- Un programma può essere visto come un insieme di moduli che interagiscono tra di loro. Tutte le funzioni di un modulo sono scritte nello stesso file.
- I moduli non fanno parte del linguaggio, ma sono un'organizzazione del codice da parte del programmatore.
- La programmazione modulare offre enormi vantaggi nell'implementazione di programmi, in particolare al crescere della complessità dei programmi da scrivere.

La Programmazione ad Oggetti

- La programmazione ad oggetti (Object Oriented Programming, OOP) rappresenta un ulteriore sviluppo rispetto alla programmazione modulare
- È un paradigma di programmazione, in cui un programma viene visto come un insieme di oggetti che interagiscono tra di loro.
- Nei linguaggi OOP esiste un nuovo tipo di dato, la classe, per modellare un insieme di oggetti dello stesso tipo.
- Una classe è caratterizzata da attributi (variabili) e funzionalità (metodi)

- La programmazione orientata agli oggetti si basa su alcuni concetti fondamentali:
  - **Classe**
  - **Incapsulamento**
  - **Oggetto**
  - **Ereditarietà**
  - **Polimorfismo**

Classi e Oggetti: un Esempio

Classi

Istanze (**Oggetti**)

Classi e oggetti

- Nei linguaggi a oggetti, il costrutto *class* consente di definire nuovi tipi di dato e le relative operazioni, chiamate anche *metodi*.
- Nel gergo dei linguaggi OO una variabile definita di un certo tipo (classe) rappresenta **un'istanza** della classe.
- Lo **stato** di un oggetto, invece, è rappresentato dai valori correnti delle variabili contenute nell'oggetto.

Il Linguaggio C++

- Il C++ supporta più paradigmi di programmazione :
  - la programmazione procedurale (è un C “evoluto”);
  - la programmazione orientata agli oggetti;
  - la programmazione generica.

- Per questo motivo il C++ è quindi un linguaggio ibrido

L'Incapsulamento

- L'incapsulamento (*information hiding*) è la suddivisione di un oggetto in due parti:
  - **Interfaccia:**
    - È l'insieme di metodi che possono essere invocati dall'utente per accedere alle funzionalità dell'oggetto;
  - **Implementazione**
    - L'insieme di metodi che implementano le funzionalità dell'oggetto e delle strutture dati necessarie
    - Non è visibile all'utente

- L'incapsulamento è realizzato per mezzo delle istruzioni:
  - **`private:`**
    - le funzioni e le variabili definite private NON sono accessibili all'esterno della classe.
  - **`public:`**
    - le funzioni e le variabili definite publiche sono accessibili all'esterno della classe.

Le classi in C++: un Esempio

```
class Contatore {
  public:
    void Incrementa();
    void Decrementa();
    unsigned int get_value();

  private:
    unsigned int value;
    const unsigned int max;
};
```

Varabili e funzioni che seguono sono visibili all'esterno

Interfaccia

Varabili e funzioni che seguono NON sono accessibili dall'esterno

Implementazione

- Di **default,** i.e. se non specificato diversamente, tutti gli attributi di una classe sono **privati** (per le struct è il contrario)
- Pertanto la classe dell' esempio precedente può essere scritta così:

```
class Contatore {
    unsigned int value;
    const unsigned int max;
  public:
    void Incrementa();
    void Decrementa();
    unsigned int get_value();
};
```

I Costruttori

- Il C++ consente l'inizializzazione automatica degli oggetti al momento della loro creazione.
- Questa inizializzazione automatica è possibile utilizzando delle funzioni, dette **costruttore**, che hanno lo stesso nome della classe.
- **Esempio**

contatore.h

```
class Contatore {
  public:
    Contatore();
        .
        .
};
```

contatore.cpp

```
Contatore::Contatore()
{
 value = 0;
}
```

NOTA

In C++ le funzioni costruttore non possono restituire valore

- la definizione di una variabile di tipo oggetto, non è solo un'istruzione per così dire passiva di allocazione di memoria, ma implica l'esecuzione del costruttore sulla variabile definita.

- **Esempio**

```
#include “Contatore.h”
Contatore cont1, cont2;
Contatore *cont_ptr;
    .
    .
    .
 // Allocazione dinamica
cont_ptr = new Contatore;
```

Il costruttore viene eseguito per ognuna delle variabili definite

Il costruttore viene eseguito al momento dell'allocazione

Costruttori parametrizzati

- I costruttori possono ricevere anche degli argomenti.
- Normalmente lo scopo degli argomenti è quello di passare un valore di inizializzazione.
- **Esempio**

contatore.h

```
class Contatore {
  public:
    Contatore();
    Contatore(int val);
      .
      .
};
```

contatore.cpp

```
Contatore::Contatore()
{
 value = 0;
}

Contatore::Contatore(int val)
{
 value = val;
}
```

- È possibile definire costruttori con più parametri.

- **Esempio**

myclasss.h

```
Class Myclass {
  public:
    Myclass(int i, int j);

       .
       .
  private:
    int a;
    int b;
};
```

myclasss.cpp

```
Myclass::Myclass(int i, int j)
{
 a = i;
 b = j;
}
```

- I costruttori parametrizzati possono essere utilizzati in fase di definizione, o allocazione di un oggetto

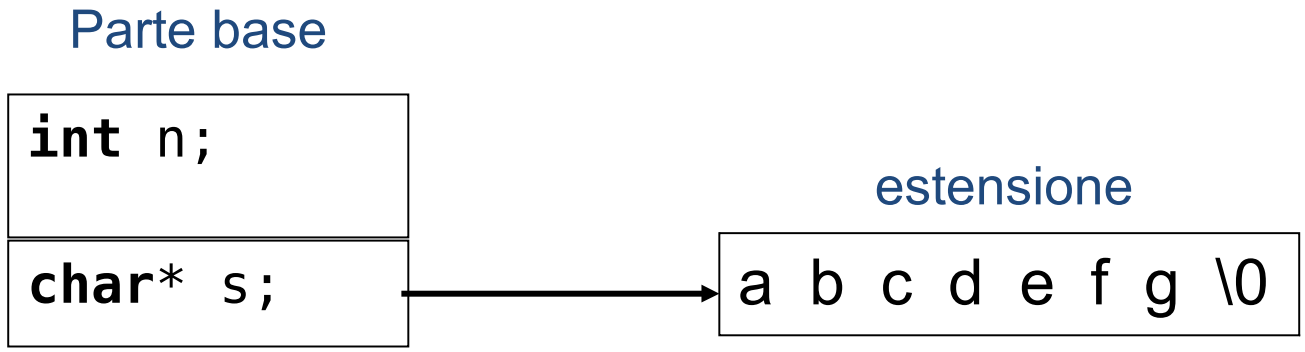
- **Esempio**

```
#include “contatore.h”
#include “myclass.h”

Contatore cont1, cont2(100), *cont_ptr;
Myclass  mc(0,0), *m_ptr;

cont_ptr = new Contatore(10);
m_ptr = new Myclass(1, 10);

Contatore cont = 100;
```

Modo alternativo per chiamare un costruttore con un solo parametro

Struttura degli Oggetti

- Ciascun oggetto della classe è costituito:
  - da una **parte base**, allocata nell' area data, stack o heap, in base alla classe di memorizzazione;
  - da una eventuale **estensione**, allocata nell' area heap



Distruttori

- Un distruttore è una funzione membro che:
  - è necessaria solo se l'oggetto presenta un'estensione dinamica
  - ha lo stesso nome della classe, preceduto da ~ (tilde)
  - Non restituisce valore (neanche void)
  - non ha alcun parametro
  - è invocata implicitamente dal compilatore quando viene deallocato lo spazio di memoria assegnato all'oggetto
  - non può essere invocata esplicitamente dal programma utente
- I distruttori servono a deallocare l'estensione dinamica di un oggetto

Distruttori: esempio

stack.h

```
Class Stack {
  public:
    Stack();
    ~Stack();
        .
        .

  private:
    int *st_ptr;
    int num;
        .
        .
};
```

stack.cpp

```
// Costruttore
Stack::Stack()
{
 st_ptr = new int[SIZE];
 num = 0;
}

// Distruttore
Stack::~Stack()
{
 delete [] st_ptr;
}
        .
        .
```

Allocazione dinamica del vettore

Dealloca il vettore

Esecuzione dei Distruttori

DEALLOCAZIONE

```
#include Stack.h
main()
{
    Stack *st_ptr;
        .
        .
    st_ptr = new Stack;
        .
        .
   delete st_ptr;
}
```

Viene invocato
il distruttore di Stack

VARIABILI LOCALI

```
#include Stack.h

void funz()
{
  Stack s1, s2;
        .
        .
        .

  return;
}
```

Il costruttore viene chiamato
prima su s1 e poi su s2

Il distruttore viene chiamato
prima su s2 e poi su s1

Costruttori e distruttori: variabili globali

```
#include stack.h
   .
   .
Stack s1;

main()
{
      .
      .
      .
   return;
}
```

Il costruttore su s1 viene chiamato PRIMA dell'esecuzione del main!

Il distruttore su s1 viene chiamato DOPO la conclusione del main!

NOTA

I costruttori globali vengono chiamati nell'ordine di dichiarazione nel file.

Non è possibile conoscere l'ordine di esecuzione dei costruttori di oggetti globali specificati in file diversi