Corso di Programmazione ad Oggetti

Il concetto di “classe”

a.a. 2008/2009
Claudio De Stefano

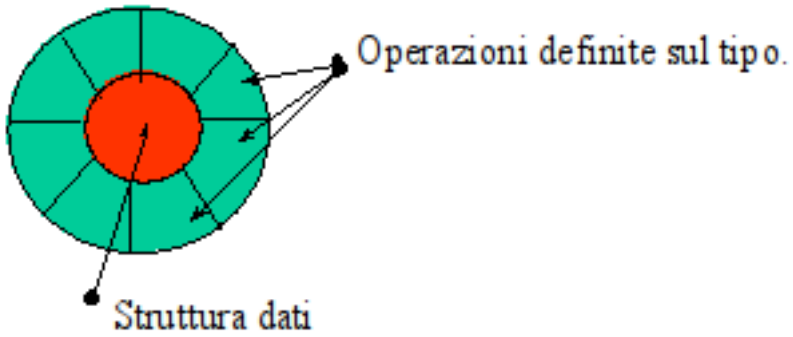
La programmazione ad oggetti

- La programmazione ad oggetti rappresenta un ulteriore sviluppo rispetto alla programmazione modulare.
- La programmazione orientata agli oggetti (Object Oriented Programming, OOP) 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. Questo tipo di dato serve appunto a modellare un insieme di oggetti dello stesso tipo.
- In generale, un oggetto è caratterizzato da un insieme di attributi e da un insieme di funzionalità

La Classe: Definizioni Generali

- Una Classe è un tipo di dato definito dall'utente.

- E costituita da:
  - Una struttura dati.
  - Un insieme di operazioni consentite sulla struttura dati.



- Una Classe è rappresentata da due entità separate:
  - Una specifica
  - Un'implementazione

Le classi in C++: un esempio

```
class Contatore {
  public:
    void Incrementa();    // operazione incremento
    void Decrementa();    // operazione decremento
    unsigned int give_value(); // restituisce il valore
                               // corrente

  private:
    unsigned int value;     // valore corrente
    const unsigned int max; // valore massimo
};
```

Punto e virgola

Incapsulamento

- La programmazione OOP consente di proteggere (incapsulare) i dati interni alla classe:

```
// Modulo utilizzatore
del modulo
// Contatore
#include "Contatore.h"

unsigned int i;
Contatore cont1, cont2;

    cont1.value = 3;
    .
    .
  cont1.incrementa();
  cont2.decrementa();
```

ERRORE!
NON VIENE COMPILATO!

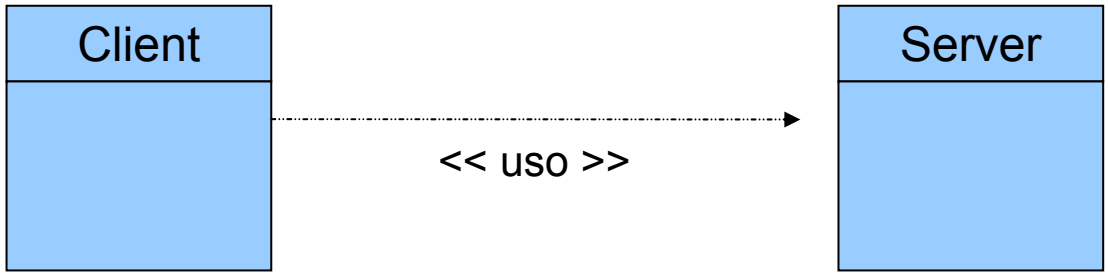
Rappresentazione grafica

- La notazione grafica è una metafora nella quale sono rappresentati in tre sezioni contigue:
  - il nome della Classe
  - la struttura dati (sequenza di variabili membro)
  - i metodi della Classe (funzioni membro della classe).

Nome della classe
-Nome variabile 1: Tipo -Nome variabile 2: Tipo
+funzione 1 (lista parametri formali): Tipo di ritorno +funzione 2 (lista parametri formali): Tipo di ritorno

Ruoli di una classe

- Una Classe può giocare un duplice ruolo:
  - Ruolo Client: cioè può utilizzare le risorse messe a disposizione da altre Classi.
  - Ruolo Server: cioè essere usata da altre Classi o da un Programma Utente.



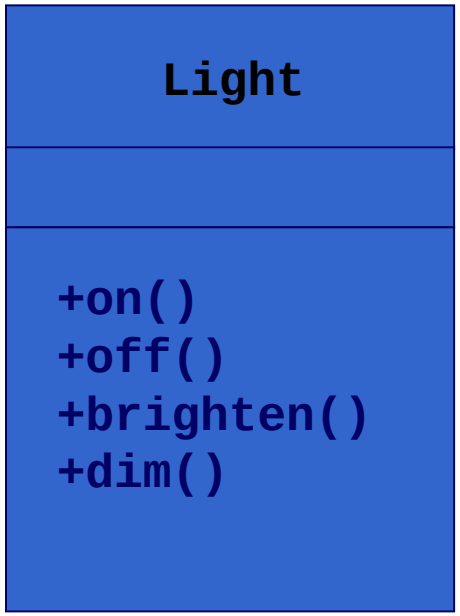
Una Classe ha un'interfaccia

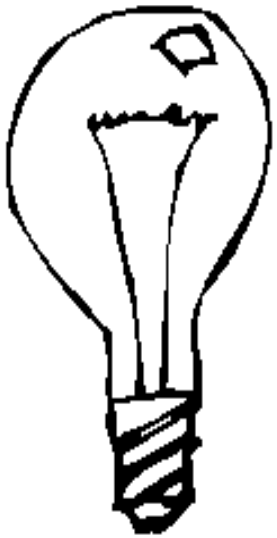
Un altro esempio

Nome della classe

Interfaccia

Operazioni





```
//Esempio di uso
Light lt;
lt.on();
```

La specifica di una classe

- Rappresenta un'interfaccia per la classe stessa in cui sono descritte:
  - le risorse messe a disposizione ai suoi potenziali utenti;
  - le regole sintattiche per il loro utilizzo;
- E' separata dalla implementazione, permette utilizzo senza che l'utente conosca i dettagli di implementazione.
- È a cura dello sviluppatore della classe.
- E' scritta in un apposito "file di intestazione".

Specifica: notazione base

```
//nome del file C.h
class C {
  public:
   //prototipi delle funzioni membro
      T1 f1(....);
   T2 f2(....);
    .........
   private:
   //struttura dati
    int i;
    char c;
        ........
}; //fine specifica della classe C
```

NOTA
di default tutti gli attributi sono **private**
Pertanto possiamo scrivere:

```
Class C {
    //struttura dati
    int i;
    char c;
     .....
  public:
      T1 f1(....);
   T2 f2(....);
    .....

};
```

L’implementazione di una classe

- E' la codifica in C++ delle singole operazioni presentate nell'interfaccia della Classe.
- E' una particolare soluzione (può cambiare l'implementazione senza che cambi l'interfaccia)
- È a cura dello sviluppatore della classe.
- E' scritta in un apposito "file di implementazione " (con estensione .cpp)

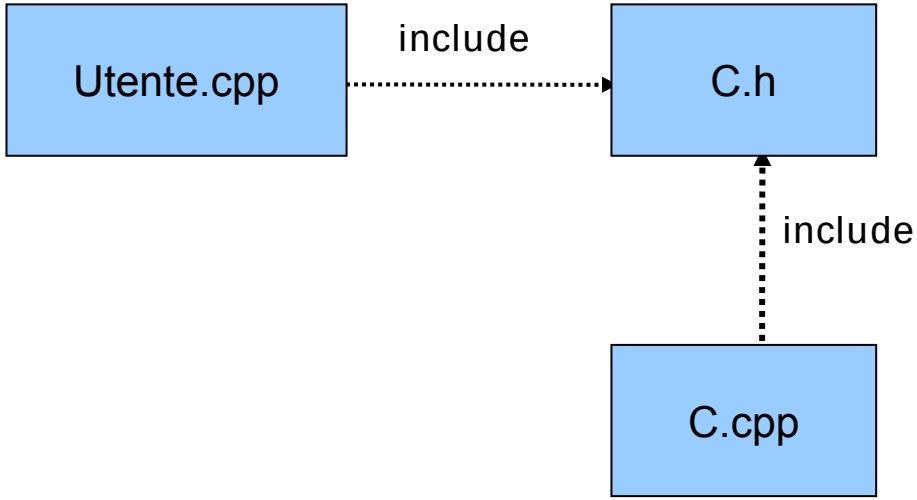
Implementazione

```
//nome del file C.cpp
#include "C.h"
T1  C::f1(....) {
   //realizzazione della funzione f1
}
T2  C::f2(....) {
   //realizzazione della funzione f2
...........
}
//fine del file C.cpp
```

deve includere anche l'header file

Uso di una classe da parte di un modulo utente.

- Affinché un programma possa utilizzare un classe deve:
  - includere la specifica della classe (contenuta nel file nomeClasse.h)
  - dichiarare istanze della classe.



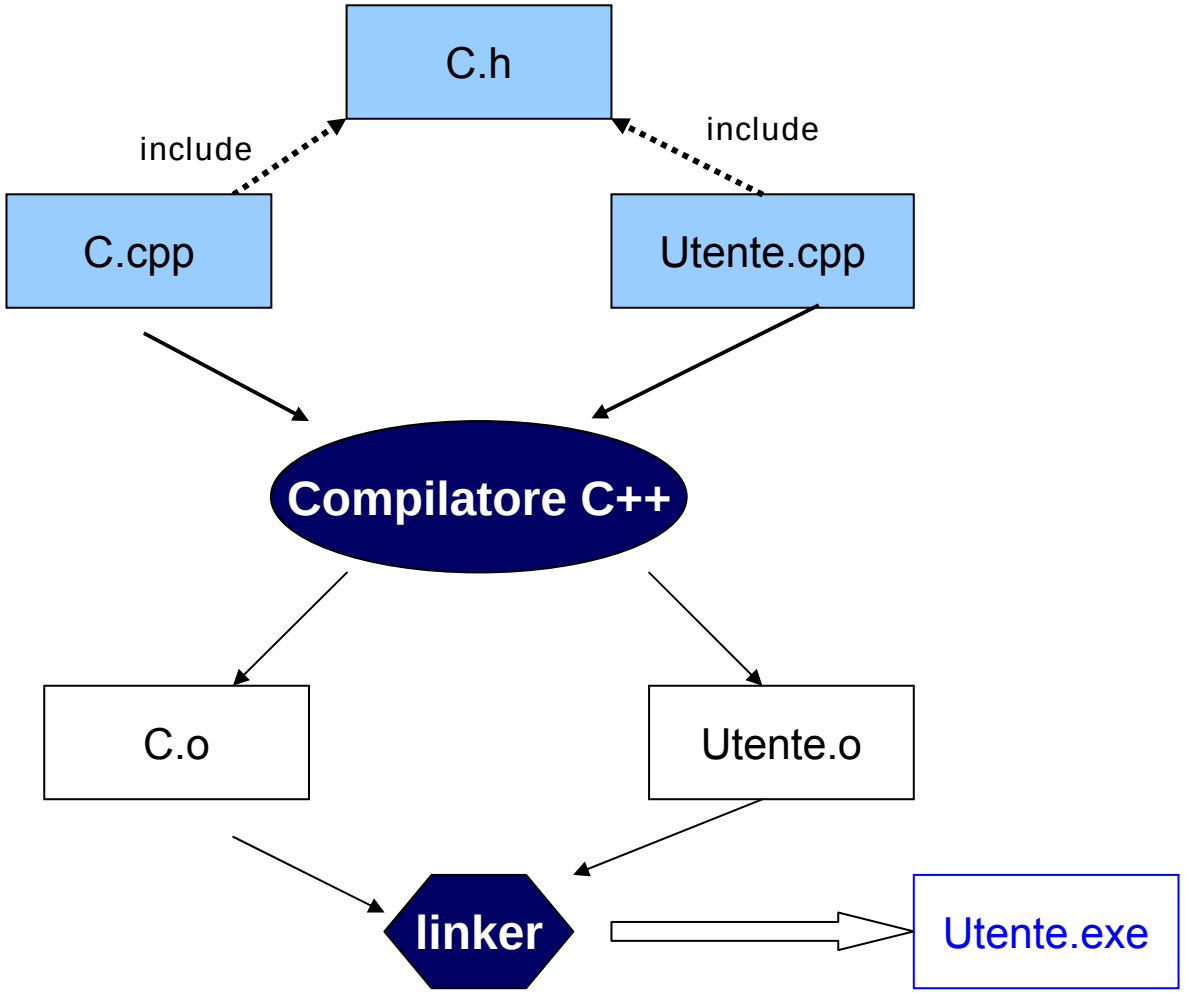
Uso di una classe

```
//PROGRAMMA UTENTE DELLA CLASSE C
#include "C.h"

main() {
C c1; //definisce oggetto c1 della classe C
C c2; //definisce oggetto c2 della classe C
..........
c1.f1(....); //applica ad oggetto c1 la funzione membro f1
c1.f2(....); //applica ad oggetto c1 la funzione membro f2
c2.f1(....); //applica ad oggetto c2 la funzione membro f1
    .
    .
    .
} //fine programma utente
```

deve includere anche l'header file

Generazione del file eseguibile



Caratteristiche di un classe

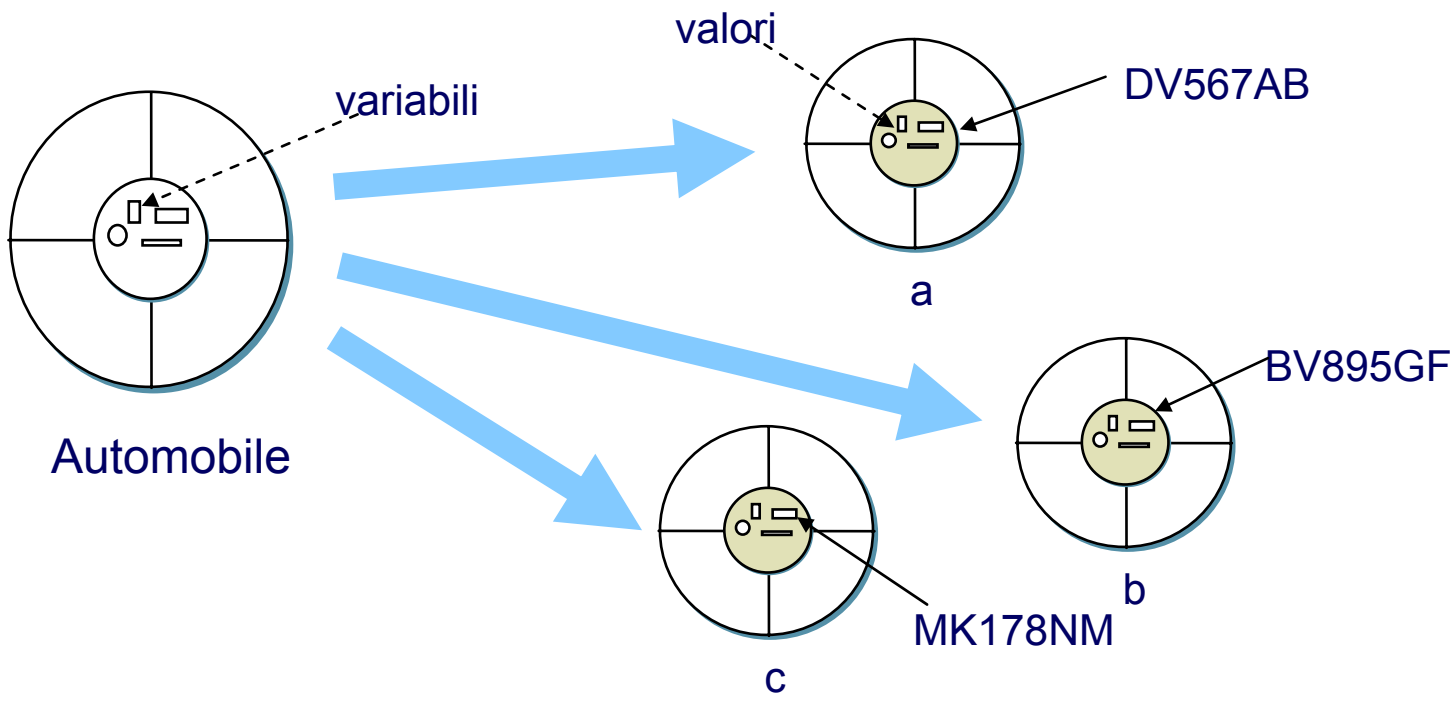
- La classe consente di definire un nuovo tipo di dato, crearne uno o più esemplari (oggetti), inizializzarli ed applicare loro i metodi di elaborazione disponibili.

- La struttura dati e gli algoritmi sono tenuti nascosti all'interno del modulo che implementa la classe.

- Lo stato dell'oggetto, cioè i valori correnti delle variabili che lo costituiscono, è incapsulato in una specifica struttura dati ed evolve unicamente in relazione ai metodi ad esso applicati.

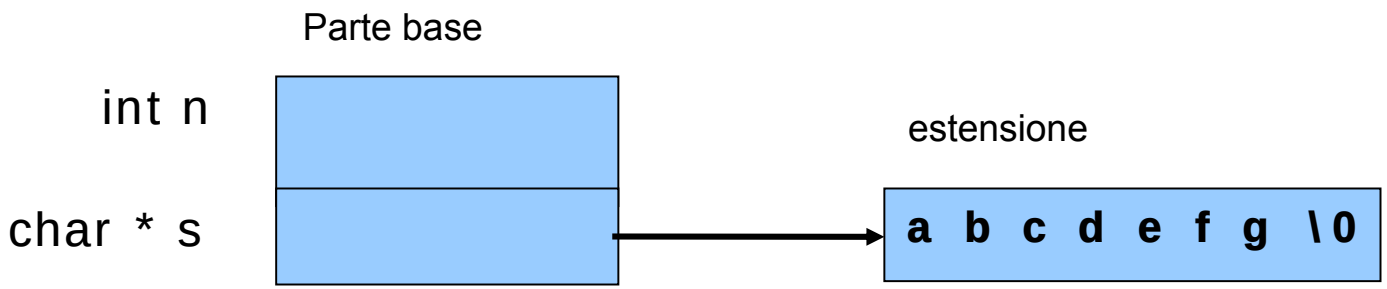
L'istanziazione degli oggetti

- Un oggetto è una istanza ("esemplare") di una classe.
- Due esemplari della stessa classe sono distinguibili soltanto per il loro stato (i valori dei dati membro), mentre il comportamento (definto dalle funzioni membro) è sempre identico.



Struttura degli oggetti

- Ciascun oggetto della classe è costituito:
  - da una parte base, allocata per effetto della definizione dell'oggetto nel programma utente in area dati statici, stack o heap, in base alla classe di memorizzazione;
  - da una eventuale estensione, allocata in area heap



I Costruttori

- È molto comune che una parte dell'oggetto debba essere inizializzata prima dell'uso. Per esempio nel caso della classe *Contatore* vista in precedenza è necessario inizilizzare il valore della variabile *value.*

- Poiché l'inizializzazione degli oggetti è un'operazione molto comune, il C++ consente di inizializzare gli oggetti al momento della loro creazione.

- Questa inizializzazione automatica è possibile utilizzando le funzioni costruttore.

I Costruttori

- Un costruttore è una particolare funzione membro di una classe che porta lo stesso nome della classe.

- Per le funzioni costruttore non deve essere specificato il tipo restituito, in quanto in C++ le funzioni costruttore non possono restituire valore .

Esempio

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

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

I Costruttori

- Il costruttore di un oggetto viene chiamato automaticamente nel momento in cui deve essere creato l'oggetto.
- Questo significa che la funzione costruttore di un oggetto viene chiamata al momento della dichiarazione dell'oggetto.
- Pertanto nella programmazione OO, la dichiarazione di una variabile di tipo oggetto, non è solo un'istruzione per così dire passiva di allocazione di memoria, ma implica l'esecuzione del codice contenuto nel costruttore della classe.

Esempio

```
#include "Contatore.h"
Contatore cont1, cont2;
```

Il costruttore viene eseguito per ognuna delle variabili dichiarate

I Costruttori

- I costruttori vengono chiamati anche per gli oggetti allocati dinamicamente.
- In questo caso, il costruttore viene chiamato al momento dell'allocazione.

Esempio

```
#include "Contatore.h"
Contatore *cont_ptr;
```

Il costruttore NON viene eseguito quando si dichiarano variabili di tipo puntaore

```
cont_ptr = new Contatore;
```

Il costruttore viene eseguito al momento dell allocazione dinamica

Costruttori parametrizzati

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

Esempio

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

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

Costruttori parametrizzati

- È possibile definire costruttori con più parametri.

Esempio

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

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

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

Costruttori parametrizzati

- I costruttori parametrizzati possono essere utilizzati in fase di dichiarazione, o allocazione utilizzando la seguente sintassi:

```
#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);
```

Costruttori con un solo parametro

- Nel caso dei costruttori con un solo parametro è anche possibile la seguente sintassi:

```
#include "Contatore.h"
Contatore cont = 100;
```

Viene chiamato il costruttore con parametro uguale a 100

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 risultato (neanche void)
  - non ha alcun parametro

- Generalmente lo scopo dei distruttori è quello di deallocare l'estensione dinamica di un oggetto

- NON può essere invocata esplicitamente dal programma utente, ma viene invocata implicitamente dal compilatore quando viene deallocato lo spazio di memoria assegnato all'oggetto.

Esempio di distruttore

- Per quanto detto in precedenza, il costruttore serve per deallocare le estensioni dinamiche degli oggetti.

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 dinamico

Esecuzione dei distruttori

- I distruttori vengono eseguiti anche quando vengono deallocati oggetti precedentemente allocati dinamicamente.

Esempio

```
void funz()
{
  Stack *st_ptr;
    .
    .
    .
  delete st_ptr;

}
```

Viene invocato il distruttore di stack

Costruttori e distruttori per variabili locali

- La funzione costruttore di un oggetto locale (definito all'interno di una funzione o di un blocco di istruzioni) viene eseguita nel momento in cui viene incontrata l'istruzione di dichiarazione dell'oggetto stesso. In pratica i costruttori vengono chiamati subito dopo che è stato allocato spazio sullo stack per l'oggetto.

- Le funzioni distruttori per gli oggetti locali vengono eseguite in ordine inverso rispetto alle funzioni costruttore. In pratica esse vengono eseguite subito prima di deallocare lo spazio sullo stack che ospita la variabile.

Esempio

```
#include Stack.h

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

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 per variabili globali

- Le funzioni costruttore degli oggetti globali vengono eseguite **prima** che inizi l'esecuzione del main.
- I costruttori globali vengono dichiarati nell'ordine di dichiarazione nel file.
- Non è possibile conoscere l'ordine di esecuzione dei costruttori di oggetti globali specificati in file diversi.
- I distruttori globali vengono eseguiti in ordine inverso **dopo** il termine dell'esecuzione del main.

Passaggio di oggetti a funzioni

- Gli oggetti possono essere passati alle funzioni come qualsiasi altro tipo di variabile.

- In particolare gli oggetti possono essere passati per valore. Questo significa che in realtà alla funzione viene passata una copia dell'oggetto. Quindi è necessario creare un nuovo oggetto.

Domande

1. Quando viene creata la copia viene eseguita la funzione costruttore?
2. E quando la copia viene distrutta viene eseguita la funzione distruttore?

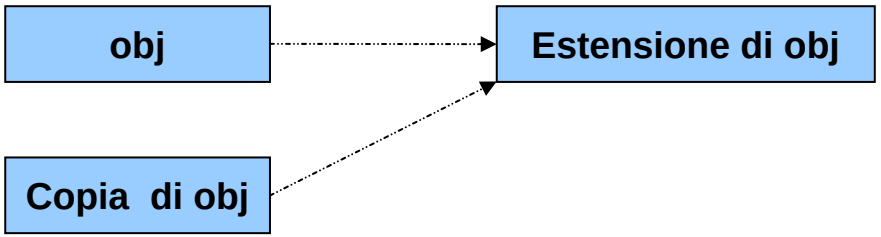
Passaggio di oggetti a funzioni

Risposta 1

- Quando viene costruita la copia per passarla alla funzione NON viene chiamato il costruttore.
- Il motivo è semplice: quando si passa un oggetto ad una funzione si intende lo stato corrente dell'oggetto. Se venisse richiamato il costruttore sulla copia, lo riporterebbe allo stato iniziale.

Risposta 2

- È necessario invece richiamare il distruttore nel momento in cui deve essere distrutta la copia.
- Notiamo che la copia è costruita bit a bit e questo può creare problemi quando l'oggetto copiato possiede un'estensione



Passaggio di oggetti a funzioni: Esempio

- Poniamo di trovarci nella situazione seguente:

```
#include Stack.h

void funz(Stack s)
{
   .
   .
}
main()
{
  Stack s1;

   .
   .
  funz(s1);

s1.pop();

}
```

Viene chiamato il costruttore di Stack
che effettua un'allocazione dinamicamente

Accadono i seguenti eventi:
1. si costruisce, sullo stack, **una copia** di s1 per passarla a funz, senza chiamare il costruttore. La copia punterà alla **stessa** area di memoria heap puntata da s1.

2. al termine della funzione viene chiamato il distruttore sulla copia, ma poiché la copia punta alla stessa area di s1, viene deallocata la memoria puntata da s1

ERRORE!: il vettore puntato da s1
è stato deallocato dal distruttore della sua copia
passato a funz!

Restituzione di oggetti

- Una funzione può restituire al chiamante un oggetto.

Esempio

```
#include Stack.h

Stack funz()
{
  Stack s;
    .
    .
  return s;
}

main()
{
  Stack s1;

  s1 = funz();

  return;
}
```

Questa assegnazione crea una copia **bit a bit** dell'oggetto locale di funz e la copia in s1.

Dopodichè l'oggetto interno a funz viene distrutto
Si hanno gli stessi problemi del caso precedente