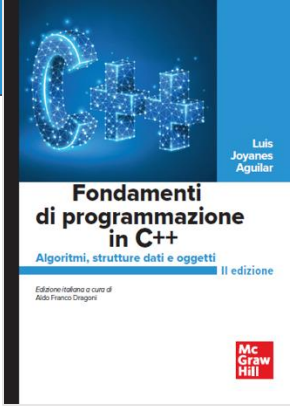
Capitolo 11

Classi e oggetti

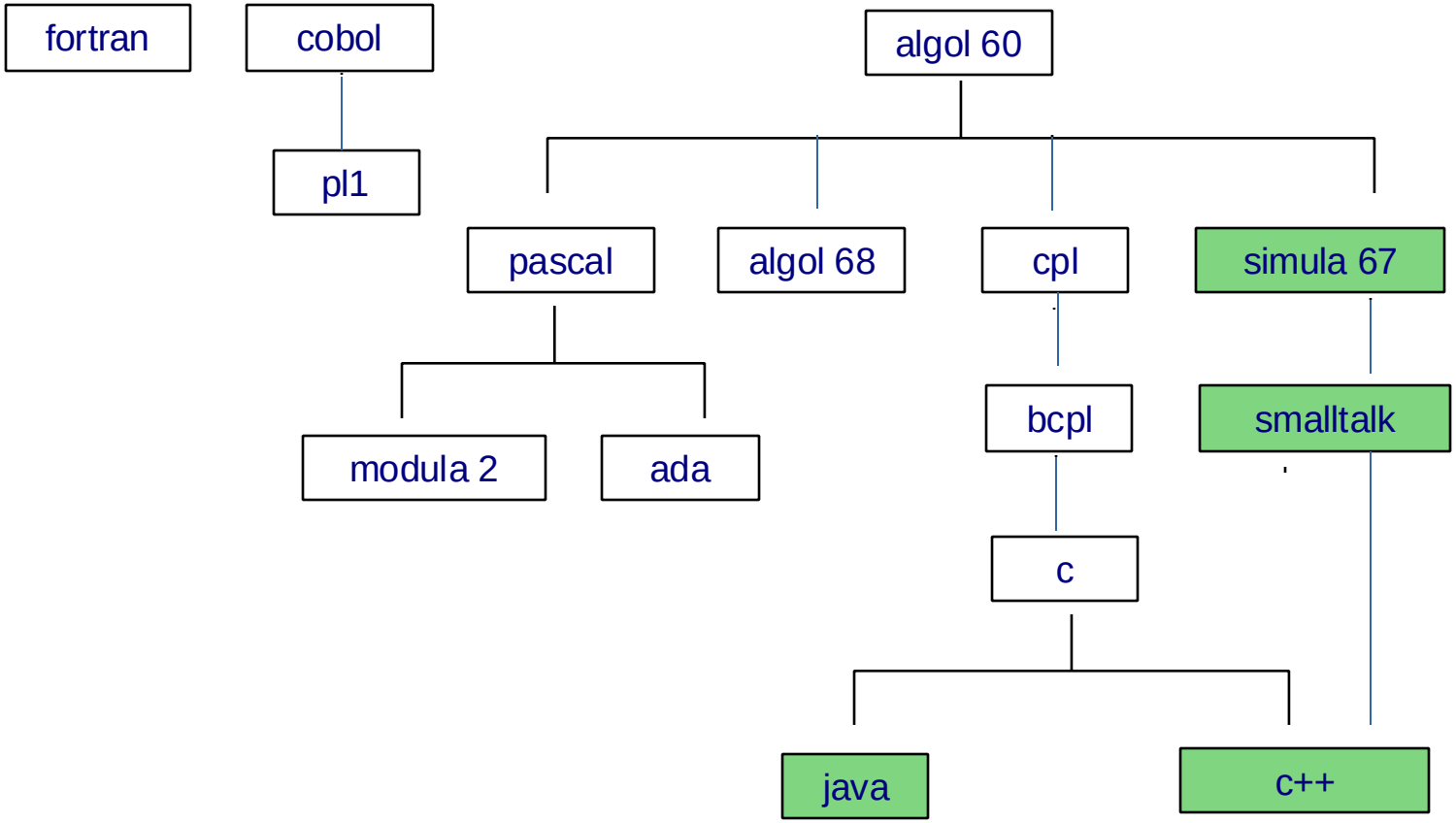
pag 289-314

Presenta: Prof. Misael Mongiovì



Object Oriented Programming

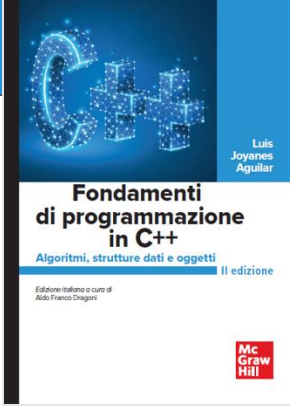
- incapsulare nella stessa struttura le descrizioni degli elementi (dati) con le azioni eseguibili su quegli elementi (funzioni)



classe

- insieme di oggetti che condividono *struttura* e *comportamento*

- contiene la specifica dei dati che descrivono ogni oggetto che ne fa parte (*attributi*), insieme alla descrizione delle azioni che l'oggetto stesso è capace di eseguire (*metodi*)

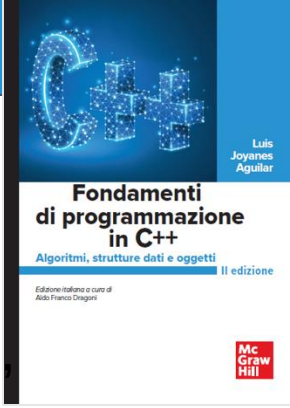


definizione di classe

- *dichiarazione*
  - dati
  - metodi accessibili dall'esterno; questi ultimi sono detti *interfaccia* della classe
- *definizioni dei metodi*

descrive l'implementazione dei metodi
(anche detti *funzioni membro*)

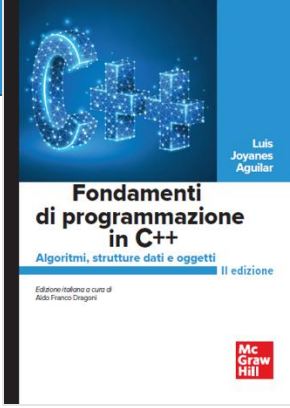
```
class NomeClasse  {
 dichiarazioni dei dati        // attributi
 dichiarazioni o definizioni delle funzioni // metodi
};
```



specificatori di accesso

- per default, i membri di una classe sono nascosti all'esterno, cioè, i suoi dati ed i suoi metodi sono *privati*
- è però possibile controllare la *visibilità* esterna mediante specificatori d'accesso:

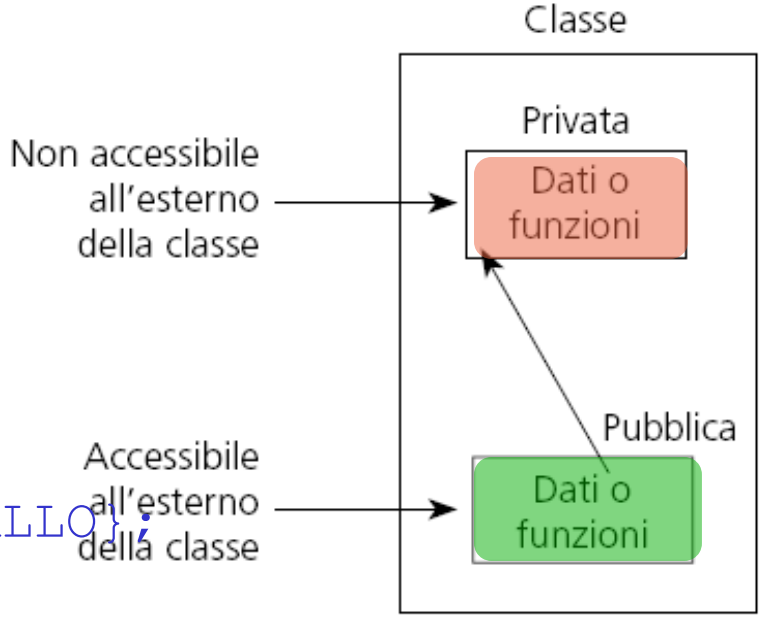
```
class NomeClasse {
 public:
 Sezione pubblica // dichiarazione membri pubblici
 // possono essere acceduti dall'esterno della classe
 protected:
 Sezione protetta // dichiarazione membri protetti
 // possono essere acceduti anche da metodi di classi
 // derivate da questa
 private:
 Sezione privata  // dichiarazione membri privati
 // possono essere acceduti solo dall'interno
 };
```

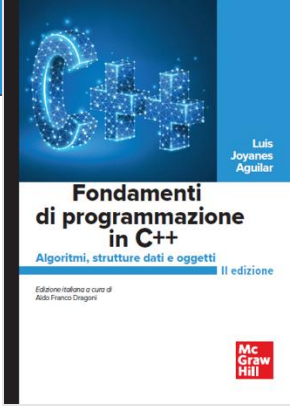


information hiding / incapsulamento

- l'*occultamento di dati (information hiding)* è una proprietà dell'OOP
- inerentemente associato al concetto di *incapsulamento*, limita molto gli errori rispetto alla semplice programmazione strutturata

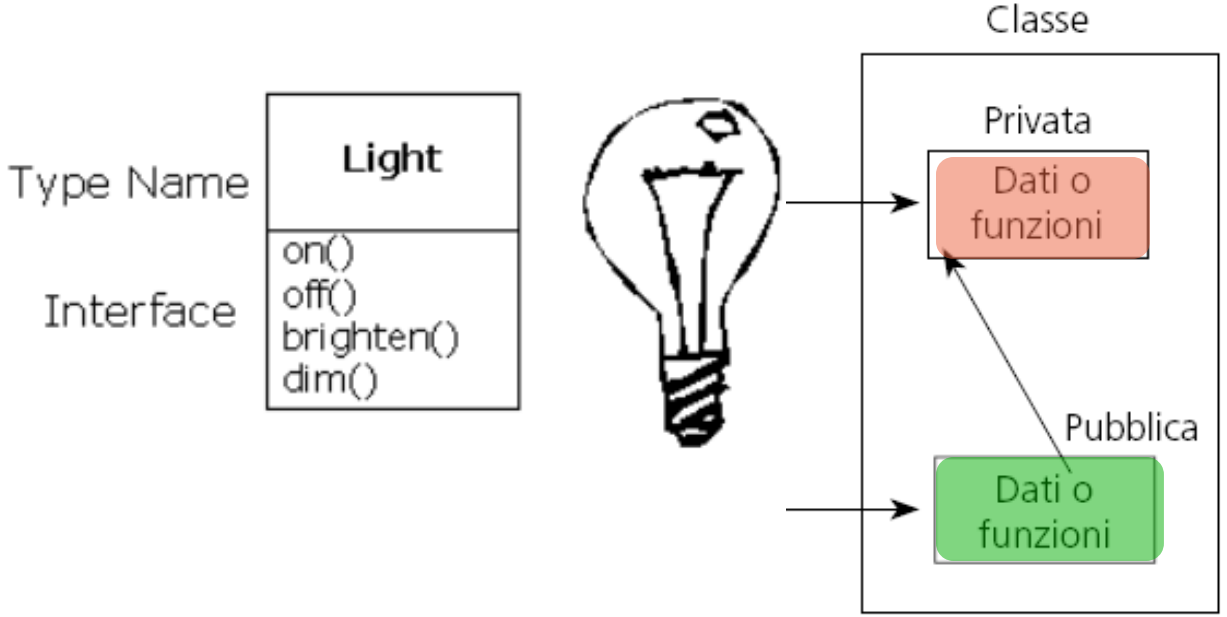
```
class Semaforo
{ public:
   void CambiaIlColore();
   //...
   private:
   enum Colore {VERDE, ROSSO, GIALLO};
   Colore c;
};
```



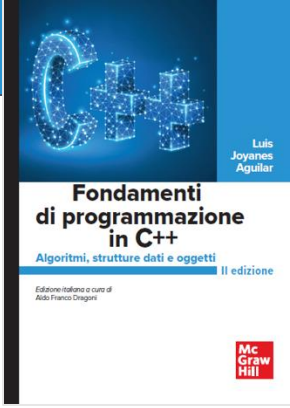


information hiding / incapsulamento

- l'*occultamento di dati (information hiding)* è una proprietà dell'OOP
- inerentemente associato al concetto di *incapsulamento*, limita molto gli errori rispetto alla semplice programmazione strutturata




```
Light lt;
lt.on();
```



alcune buone prassi

- le dichiarazioni dei metodi (cioè le intestazioni delle funzioni), normalmente, si collocano nella sezione pubblica e le dichiarazioni dei dati (attributi), normalmente, si mettono nella sezione privata

- è indifferente collocare prima la sezione pubblica o quella privata; è però consigliabile collocare la sezione pubblica prima, per mettere in evidenza le operazioni che fanno parte dell'interfaccia utente pubblica

- le parole chiavi `public` e `private` seguite da due punti, segnalano l'inizio delle rispettive sezioni pubbliche e private; benché non sia comune, una classe può avere varie sezioni pubbliche e private

oggetti

- definita una classe, si possono generare le *istanze* della classe, cioè gli *oggetti*

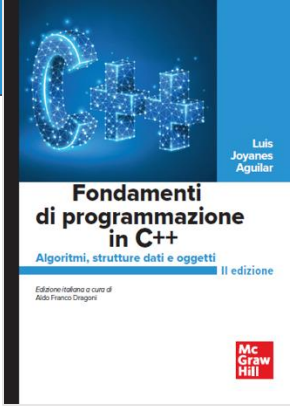
```
nome_classe identificatore ;

Punto P;
Semaforo S;
```

- un oggetto sta alla sua classe come una variabile al suo tipo
- l'operatore di accesso al campo (.) diventa *operatore di accesso al membro*

```
Punto p;
p.Valorizza_x(100);

cout << " l'ascissa del punto è " << p.Leggi_x();
```

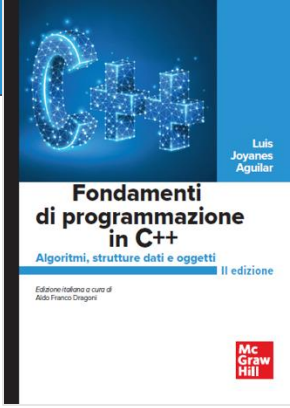


dati membro

- possono essere di qualunque tipo valido, ad eccezione del tipo della classe che si sta definendo
- non è permesso inizializzare un membro dato all'atto della sua definizione:

```
class C {
private:
 int T = 0;                    // Errore
 const int CInt = 25;          // Errore
 int& Dint = T                 // Errore
};
```

- ciò non avrebbe senso perché la definizione della classe indica solo il *tipo* di ogni dato, e non riserva realmente memoria (sarebbe come voler inizializzare un campo di una struttura);
  si deve invece inizializzare i membri dato ogni volta che si crea un'*istanza specifica* della classe mediante il *costruttore* della classe



funzioni membro

- i metodi possono essere sia solo *dichiarati* che (anche) *definiti* all'interno delle classi
- il prototipo della funzione *deve* essere dichiarato dentro la classe, mentre il corpo della funzione *può* essere definito altrove

```
class Razionale
{   public:
      void assegna (int, int); // dichiarazione
      double converti()
      { return (double)num/den;} // definizione
      void inverti(); // dichiarazione
      void stampa();  // dichiarazione
    private:
      int num, den;
 };
```

chiamate a funzioni membro

- i metodi di una classe s'invocano così come si accede ai dati di un oggetto, tramite l'operatore di accesso al membro

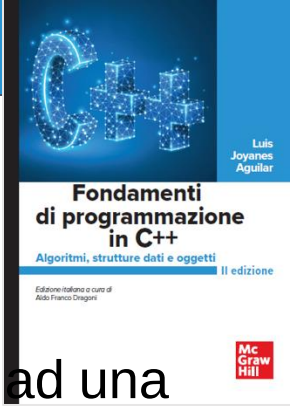
```
class Demo
{   public:
       void f1 (int P1)
       { ... }
       void f2 (float P2)
       { ... }
    private:
       // ...
 };
Demo d1, d2; // definizione degli oggetti d1 e d2
 ...
d1.f1(123);
d2.f2(3.14);
```

- i metodi definiti nella classe sono funzioni *in linea*
- per funzioni grandi è preferibile codificare nella classe solo il prototipo del metodo; nella sua definizione *fuori linea* bisognerà premettere il nome della classe a cui appartiene, seguito dall'*operatore di risoluzione di visibilità* `::`

```
class Razionale
{    public:
       Razionale () {          // definizione
       num = 0;
       den = 1;
   }

       double converti()                                    //
definizione
       { return (double)num/den;}
       void inverti();                                      //
dichiarazione
       void stampa();                                       //
dichiarazione
     private:
       int num, den;
   };
```

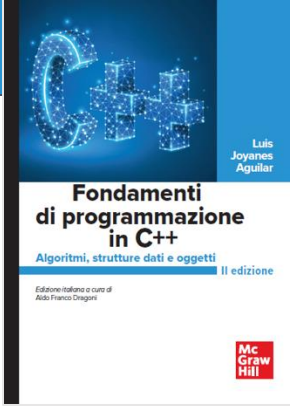
funzioni *inline* e *offline*



costruttori

- è bene che un oggetto si possa auto-inizializzare all'atto della creazione, senza dover effettuare una successiva chiamata ad una sua qualche funzione membro
- ***costruttore***
  - metodo che viene automaticamente eseguito all'atto della creazione di un oggetto
  - ha lo stesso nome della propria classe e può avere qualunque numero di parametri ma ***non restituisce alcun valore*** (neanche `void`)

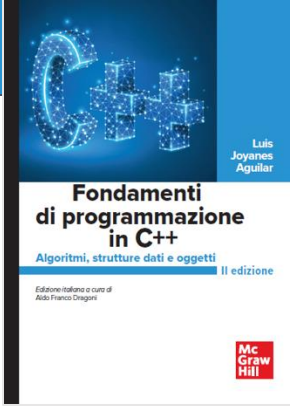
```
class Razionale {
public:
    Razionale () { // costruttore di default
    num = 0;
    den = 1;
    }
    double converti();
    void inverti();
 private:
    int num, den;
};
```



costruttori con parametri

- servono per inizializzare con valori passati all'atto della creazione dell'oggetto

```
class Razionale {
public:
      Razionale () {                  // costruttore di default
      num = 0;
      den = 1;
      }
      Razionale (int n, int d) { // costruttore con parametri
      num = n;
      den = d;
      }
      double converti();
      void inverti();
    private:
      int num, den;
 };
Razionale uno; // c viene inizializzato a 0/1
Razionale rapporto(5,3); // rapporto viene inizializzato a 5/3
Razionale fattore(4); // ERRORE!!! Nessun costruttore con un parametro
```



costruttori con parametri

- possiamo compattare la definizione del costruttore

```
class Razionale {
  public:
      Razionale (int n=0, int d=1) { // costruttore
      num = n;
      den = d;
      }
      double converti();
      void inverti();
   private:
      int num, den;
 };
Razionale* nuovo = new Razionale; // oggetto creato dinamicamente
```

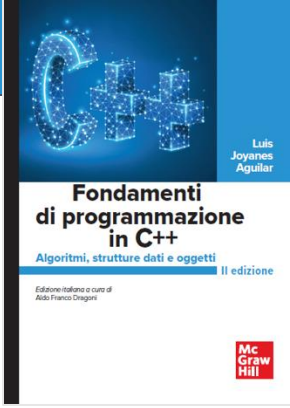
- C++ crea automaticamente un costruttore di default quando non vi sono altri costruttori, tuttavia esso non inizializza i membri dato della classe a valori predefiniti

- *costruttore di copia*: creato automaticamente quando si passa un oggetto per valore ad una funzione (si costruisce una copia locale dell'oggetto) o si definisce un oggetto inizializzandolo ad un altro oggetto dello stesso tipo

vettore di inizializzazione membri

- attributi *costanti* e *riferimenti* non potrebbero essere assegnati dal costruttore
- per questo si utilizza il *vettore di inizializzazione di membri* che viene posto immediatamente dopo la lista dei parametri del costruttore
- consiste nel carattere `:` seguito da uno o più inizializzazioni di membro separati da `,`

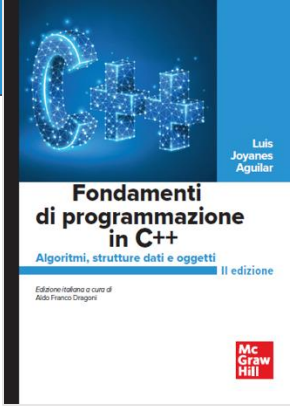
```
class C {
 private:
    int T;
    const int CInt;
    int& RInt;
 public:
    C(int Param) : T(Param), CInt(25), RInt(T)
    { ..... }                                    // codice
del costruttore
};
```



distruttore

- metodo speciale che viene chiamato automaticamente quando si distrugge un oggetto; serve per liberare la memoria assegnata dal costruttore
- ha lo stesso nome della sua classe preceduto dal carattere **~**
- *non* ha tipo di ritorno
- *non* accetta parametri
- *non* ve ne può essere più d'uno
- se non si dichiara esplicitamente un distruttore, C++ ne crea automaticamente uno vuoto

```
class Razionale {
public:
      Razionale (int n=0, int d=1) { // costruttore
      num = n;
      den = d;
      }
      ~Razionale () {} // distruttore
   private:
      int num, den;
};
```



esempio di distruttore

```
class MiaStringa
{
 public:
    MiaStringa(const char* Stringa="")
    { Lunghezza = strlen(Stringa) + 1;
      miaStringa = new char[Lunghezza];
      strncpy(miaStringa, Stringa, Lunghezza);
      miaStringa[Lunghezza-1] = '\0';
    }
    ~MiaStringa() // distruttore
    { delete[] miaStringa;
      miaStringa = 0;
    }
    char* RestituisciStringa() { return miaStringa; }
    int RestituisciLunghezza() { return Lunghezza; }
 private:
    char* miaStringa;
    int Lunghezza;
};

int main()
{ MiaStringa MioNome("Aldo");
    cout << "Il mio nome è: " << MioNome.RestituisciStringa() << endl;
    return 0;
} // il distruttore di MioNome viene chiamato qui!
```

header file e intestazioni di classe

- separare in files diversi il codice “cliente” della classe dal codice della classe
- separare il codice sorgente di una classe in due files con lo stesso nome della classe
  - uno con la definizione della classe (con solo i prototipi dei metodi) ed estensione .h
  - un altro con le implementazioni dei metodi della classe ed estensione `.cpp`

header file e intestazioni di classe


```
class MiaStringa
{
 public:
    MiaStringa(const char* Stringa="")
    { Lunghezza = strlen(Stringa) + 1;
      miaStringa = new char[Lunghezza];
      strncpy(miaStringa, Stringa, Lunghezza);
      miaStringa[Lunghezza-1] = '\0';
    }
    ~MiaStringa()
    { delete[] miaStringa;
      miaStringa = 0;
    }
    char* RestituisciStringa() { return miaStringa; }
    int RestituisciLunghezza() { return Lunghezza; }
 private:
    char* miaStringa;
    int Lunghezza;
};

int main()
{ MiaStringa MioNome("Aldo");
    cout << "Il mio nome è: " << MioNome.RestituisciStringa() <<
endl;
    return 0;
}
```

cliente.cpp

```
#include <iostream>
#include "MiaStringa.h"
int main()
{ MiaStringa MioNome("Aldo");
    cout << "Il mio nome è: " <<
    MioNome.RestituisciStringa() <<
    endl;
    return 0;
}
```

MiaStringa.h

```
class MiaStringa
{
 public:
    MiaStringa(const char*);
    ~MiaStringa();
    char* RestituisciStringa();
    int RestituisciLunghezza();
 private:
    char* miaStringa;
    int Lunghezza;
};
```

MiaStringa.cpp

```
#include <string>
#include "MiaStringa.h"
MiaStringa::MiaStringa(const char* Stringa="")
    { Lunghezza = strlen(Stringa) + 1;
      miaStringa = new char[Lunghezza];
      strncpy(miaStringa, Stringa, Lunghezza);
      miaStringa[Lunghezza-1] = '\0';
    }
MiaStringa::~MiaStringa()
    { delete[] miaStringa;
      miaStringa = 0;
    }
char* MiaStringa::RestituisciStringa()
    { return miaStringa; }
int MiaStringa::RestituisciLunghezza()
    { return Lunghezza; }
```