



UNIVERSITÀ
degli STUDI
di CATANIA

Implementazione di classi in C++

Corso di programmazione I AA 2019/20

Corso di Laurea Triennale in Informatica

Prof. Giovanni Maria Farinella

Web: <http://www.dmi.unict.it/farinella>

Email: gfarinella@dm.unict.it

Dipartimento di Matematica e Informatica

La classe è la **descrizione generale** di come devono essere costruiti gli oggetti corrispondenti.

In altre parole una classe è una sorta di **progetto** con cui ogni particolare oggetto di quel tipo dovrà essere costruito.

Un oggetto costruito sulla base della descrizione contenuta nella classe X si dice **istanza della classe X**.

Dichiarazione di classi in C++

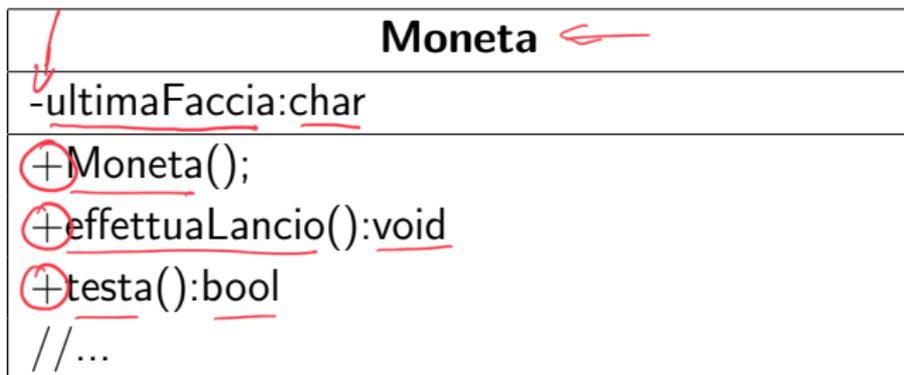
Schema generale **definizione** di una classe.

```
class A {  
    [...] // variabili o campi o attributi (STATO) -  
    [...] // metodi -  
};
```

Dichiarazione di classi in C++

Modificatore di accesso per i membri della classe: attributo `ultimaFaccia`, simbolo `-` (**private**):

- accessibile solo da codice all'interno della classe `moneta`.



Dichiarazione di classi in C++

Modificatore di accesso per i metodi `Moneta()` (costruttore), `effettuaLancio()` e `testa()` indicato da simbolo **+**, ovvero **public**:

- metodi accessibili da **qualsunque blocco di codice dell'applicazione**.

Moneta
-ultimaFaccia:char
+Moneta();
+effettuaLancio():void
+testa():bool
//...

Dichiarazione di classe vs definizione di metodi

Dichiarazione della classe Moneta → file moneta.h

```
1 class Moneta{  
2   public:  
3     Moneta ();  
4     void effettuaLancio (); //  
5     bool testa () const ;  
6     bool croce () const ;  
7     char getFaccia () const ;  
8   private:  
9     char ultimaFaccia ;  
10 };
```

Dichiarazione di classe vs definizione di metodi

Definizione dei metodi di Moneta → `moneta.cpp`

NB: Attenzione all'operatore risolutore di scope e al nome della classe, elementi necessari per definire il metodo fuori dalla dichiarazione della classe!

```
<tipo ritorno> <nome_classe>::<nome_metodo>
```

```
1 bool Moneta::testa() {  
2   return (ultimaFaccia == 'T');  
3 }
```

Dichiarazione di classe vs definizione di metodi

Alternativa: **Dichiarazione della classe e definizione dei suoi metodi** all'interno del file `moneta.cpp`

```
1  class Moneta{  
2  
3  private:  
4      char ultimaFaccia;  
5  
6  public:  
7      //...  
8  
9      bool testa() {  
10         return (ultimaFaccia == 'T');  
11     }  
12 };
```

Remark

- Quando un metodo viene invocato, il **flusso di controllo** “passa” a tale metodo:
 - saranno eseguite le istruzioni del metodo, dalla prima all’ultima, **fino alla fine** o fino ad un’istruzione `return`.
- **Dopo** l’esecuzione dell’**ultima istruzione** del metodo, oppure in corrispondenza di una istruzione **return**, il flusso “ritorna” al punto in cui è partita la chiamata:
 - sarà eseguita l’istruzione successiva alla chiamata del metodo

1. Invocazione di metodi

```
1  int main(){
2      Dado d1, d2;
3      //...
4      d1.effettuLancio();
5      //...
6      d2.effettuaLancio();
7      //..
8  }
```

Invocazione di metodi vs flusso di controllo

2. Invocazione di metodi in cascata

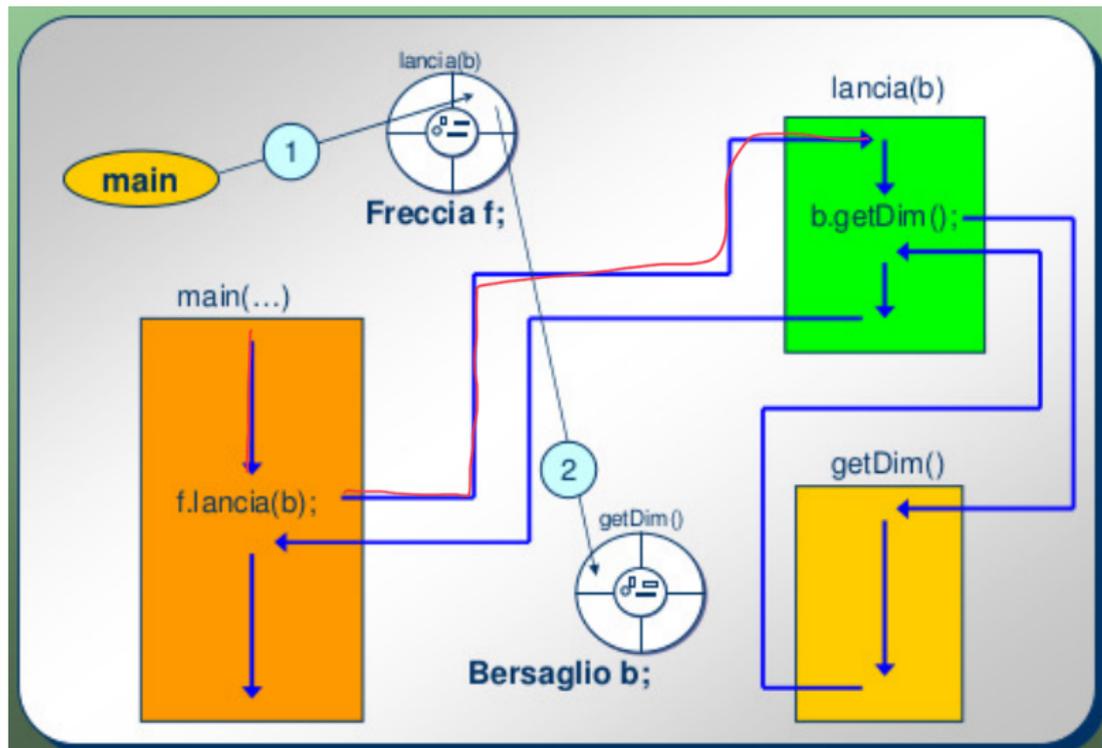
```
1  int main(){  
2      GestoreMonete g;  
3      //...  
4      g.depositaUnaMoneta(10);  
5      //...  
6  }
```

```
1  //...  
2  bool GestoreMonete::depositaUnaMoneta (int valore){  
3      //...  
4      short moneta = convertiValore(valore);  
5      //...  
6  }
```

2. Invocazione di metodi in cascata (cont.)

```
1  short GestoreMonete::convertiValore (int valore)
2      {
3          switch (valore)
4          {
5              // ...
6          }
7      }
```

Invocazione di metodi vs flusso di controllo



Invocazione di metodi vs flusso di controllo

3. Invocazione di metodi in cascata (oggetti distinti)

```
1  int main(){
2      Freccia f;
3      Bersaglio b;
4      //...
5      | f.lancia(b);
6  }
```



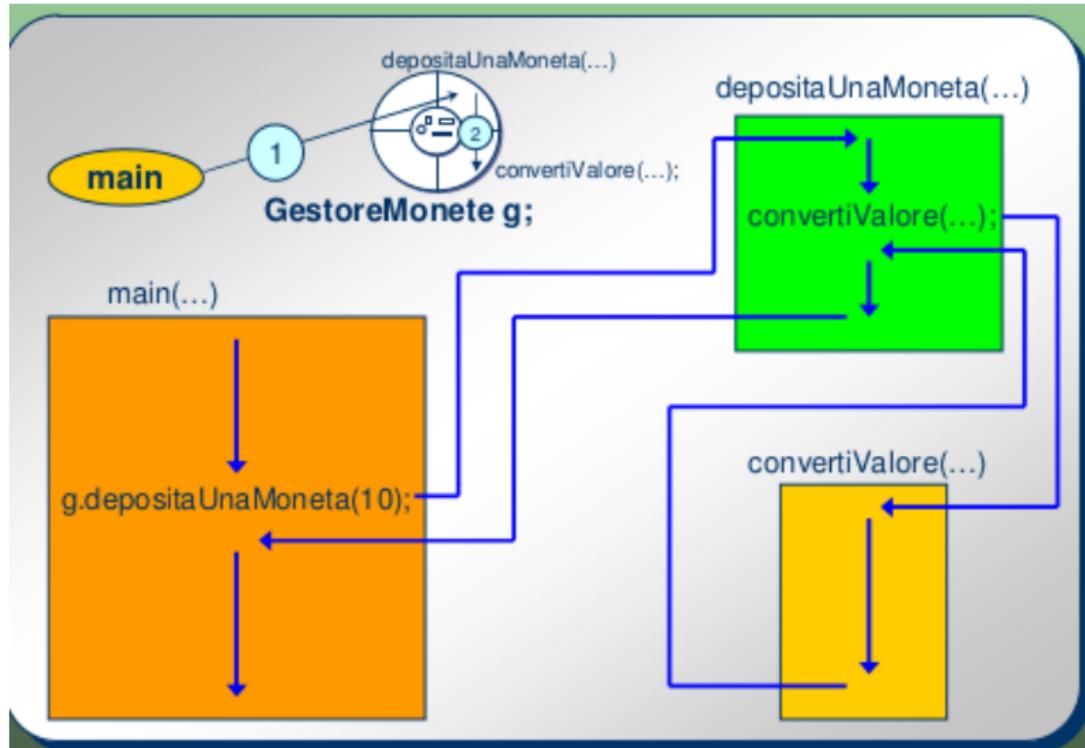
```
1  //...
2  bool Freccia::lancia(Bersaglio b){
3      //...
4      b.getDim();
5      //...
6  }
```



3. Invocazione di metodi in cascata (oggetti distinti – cont.)

```
1  int Bersaglio::getDim()  
2      {  
3          return dim;  
4      }
```

Invocazione di metodi vs flusso di controllo



Remark: Non confondere la **dichiarazione del metodo** con la **invocazione del metodo stesso**.

- La dichiarazione del metodo **definisce il suo prototipo** e l'appartenenza ad una ben determinata classe.
- La invocazione di un metodo è una **istruzione che avvia l'esecuzione del codice del corpo del metodo**.

Definizione del corpo dei metodi

Ricordiamo che i metodi sono particolari tipi di funzioni.

Quindi, all'interno dei metodi è possibile **referenziare dati locali** al metodo, ovvero **variabili locali**.

```
1 double Calcolatrice::sum(double a, double b){  
2     double c = a + b;  
3     return c;  
4 }
```

NB: Per variabile c avverrà allocazione automatica (sullo stack), essa sarà distrutta contestualmente alla fine della esecuzione del metodo.

Definizione del corpo dei metodi

Inoltre è possibile referenziare i parametri formali

```
1  double Calcolatrice::sum(double a, double b){  
2      return a+b;  
3  }
```

NB: Allocazione in memoria di parametri formali **identica** a quella delle variabili locali.

Variabili locali (e parametri formali) caratterizzati da:

- Scope (portata) o visibilità limitata al blocco di codice del metodo;
- Ciclo di vita (quando esse vengono create e distrutte) limitato all'esecuzione del metodo;
 - NB: restituire valori di variabili locali (es: `return somma;`), non indirizzi di tali variabili!

Definizione del corpo dei metodi

A differenza delle funzioni, all'interno del codice dei metodi è possibile **referenziare variabili di istanza** della classe in cui il metodo è definito.

```
1 class Moneta{
2     [ private:
3         char valoreUltimoLancio;
4         //...
5     }
6
7     bool Moneta::testa () { ←
8         return ( valoreUltimoLancio == 'T');
9     }
```

Definizione del corpo dei metodi

```
1  class A{
2      private:
3          int a, b, c;
4      public:
5          void foo(int a, int x){
6              a = 2 * x - a;
7          }
8  }
```

A quale variabil si riferisce l'istruzione alla riga 6? Alla variabile locale o alla variabile di istanza?

- **Risposta:** la variabile di istanza è "oscurata" dalla variabile locale

Definizione del corpo dei metodi

```
1  class A{
2      private:
3          int (a), b, c;
4      public:
5          void foo(int a, int x){
6              // a = 2 * x - a; // NO
7              this -> a = 2 * x - a;
8          }
9  }
```

Soluzione: Puntatore **this** per referenziare esplicitamente variabili di istanza e metodi.

Esempio svolto

23_01.cpp

Definizione del corpo dei metodi

Restituzione di valori da metodi.

Come per le funzioni Il tipo restituito deve essere dichiarato nel prototipo e/o nella sua definizione.

```
1  class A {  
2  //...  
3  public:  
4  void foo(int x);  
5  double sum(double a, double b);  
6  }
```

NB: void indica che il metodo non restituisce alcun valore.

Definizione del corpo dei metodi

Restituzione di valori da metodi (cont.)

Come per le funzioni, affinché un metodo restituisca un valore, è **necessario** inserire almeno una istruzione `return` nel corpo del metodo.

```
1  double Calcolatrice::sum(double a, double b){  
2  // ...  
3  return a+b;  
4  }
```

Definizione del corpo dei metodi

Sintassi della istruzione **return**.

```
return <Espressione>;
```

Espressione viene **valutata** ed il risultato viene **restituito al chiamante**.

Il tipo di *Espressione* deve essere **compatibile** con il tipo nella dichiarazione.

Una `return` senza alcuna espressione può essere **inserita in un metodo che non restituisce valori**, per “forzare” l’uscita anticipata dal metodo.

Passaggio di parametri

1-Passaggio per valore (C/C++). Una copia del valore attuale del parametro viene depositata sullo stack.

```
1  class A{
2      public:
3          int stampa(int a){
4              a = a+1;
5              cout << a << endl;
6          }
7  }
```

Le modifiche al parametro formale non si riflettono sul valore della variabile usata come parametro attuale della funzione.

Passaggio di parametri

2-Passaggio mediante puntatore (C/C++).

```
1  class A{
2      public:
3          int stampa(int *a){
4              (*a)++; ←
5              cout << *a << endl;
6          }
7  };
8  //..
9  int b=10;
10 stampa(&b); //dopo la chiamata b sara' 11!
```

Il codice del metodo **può modificare il valore della variabile usata come parametro attuale**, mediante operatore di *indirizione* o *referenziazione*.

Passaggio di parametri

3-Passaggio **per riferimento** (introdotto con il C++!).

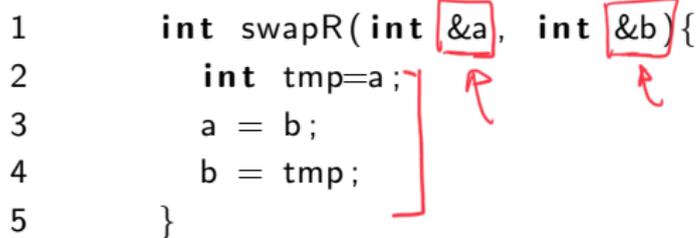
```
1  class A{
2      public:
3          int stampa(int &a){
4              a = a+1;
5              cout << a << endl;
6          }
7  }
8  int b=10;
9  stampa(b); //dopo la chiamata b sara' 11!
```

In pratica il **parametro formale a** diviene **alias di b**. Ogni modifica ad a all'interno del corpo del metodo si riflette in b.

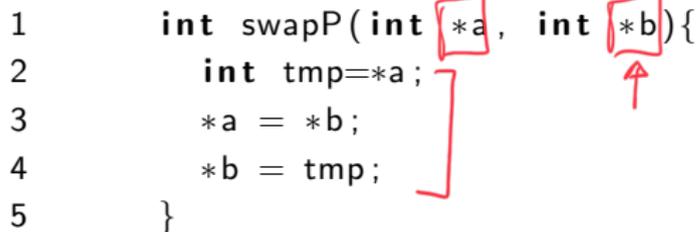
Passaggio di parametri

Passaggio per riferimento vs passaggio mediante puntatore (1)

```
1   int swapR(int &a, int &b){  
2       int tmp=a;  
3       a = b;  
4       b = tmp;  
5   }
```



```
1   int swapP(int *a, int *b){  
2       int tmp=*a;  
3       *a = *b;  
4       *b = tmp;  
5   }
```



(+) Passaggio per riferimento sintatticamente più leggero.

Passaggio di parametri

Passaggio per riferimento vs passaggio mediante puntatore (2)

```
1  → int swapR(int &a, int &b);
2  → int swapP(int *a, int *b); ←
3  //..
4  int x=2,y=4;
5  → swapP(&x, &y);
6  //vs
7  swapR(x, y);
```

Linea 5: passato indirizzo variabili x e y. Chiaro!

Linea 7: cosa succede? Passaggio per valore o by reference? Bisogna vedere prototipo funzione swapR()!!

(-) **Passaggio per riferimento sintatticamente opaco.**

Passaggio di parametri

Passaggio by reference: usi tipici/consigliati

1. il metodo **deve modificare uno o più dei suoi argomenti**
2. il metodo deve restituire più di un valore, ad esempio:

```
1 int sum(int a, int b, int &result){
2     if(a > 0 && b >0){
3         result = a+b;
4         return 0; // valore di controllo
5     }
6     else
7         return -1; // valore di controllo
8 }
```

Passaggio by reference: usi tipici/consigliati

3. il metodo deve ricevere oggetti (possibilmente grandi!)

```
1 void sum(Matrice3D &m1, Matrice3D &m2, Matrice 3D &result) {  
2     // ..calcola la somma..  
3 }
```

Ad esempio la somma di due matrici molto grandi (istanze di una (ipotetica) classe Matrice3D).

Passaggio by reference: const

```
1  int sommaElementi(const Matrice3D &matrice){  
2      // ..calcola la somma degli elementi della matrice  
3  }
```

Anteporre `const` al tipo del parametro indica al compilatore che l'oggetto **non può essere modificato** dal metodo:

- non si potranno invocare metodi che modificano lo stato dell'oggetto
- non si potrà modificare lo stato dell'oggetto mediante accesso ai suoi membri pubblici

Esempio svolto

23_02.cpp

Argomenti della funzione main

int main()

```
1  int main(int argc, char *argv[]){
2      cout << "Nome programma: " << argv[0] << endl;
3      cout << "Primo argomento: " << argv[1] << endl;
4  }
```

Opzionalmente, si può definire la funzione main con due parametri formali:

- argc rappresenta il numero dei parametri che provengono dalla linea di comando.

Argomenti della funzione main

```
1  int main(int argc, char *argv []) {
2    cout << "Nome programma: " << argv[0] << endl;
3    cout << "Primo argomento: " << argv[1] << endl;
4  }
```

- argv è array di puntatori a caratteri di lunghezza argc.
 - argv[0] è nome del programma (ES: a.out)
 - argv[i] (con $i > 0$) è stringa contenente (i-1)simo parametro passato da utente a linea di comando

Argomenti della funzione main

```
$ g++ mio_prog.cpp -o mio_prog  
$ ./mio_prog arg1 arg2
```

la stringa "arg1" sarà accessibile da `argv[1]`. la stringa "arg2" sarà accessibile da `argv[2]`.

mentre la stringa "mio_prog" sarà memorizzata in `argv[0]`.

Esempio svolto

23_03.cpp

Argomenti standard per le funzioni e i metodi

Dichiarazione prototipo può specificare uno o più **argomenti standard**.

```
1  int foo(int a, double b, char c); ←  
2  //oppure  
3  int foo(int a, double b, char c='Z'); ←
```

Gli argomenti standard vanno specificati a partire da quello più a destra. ES (genera un **errore del compilatore!**).

```
1  int foo(int a=1, double b, char c='Z');
```

Argomenti standard per le funzioni e i metodi

Anche definizione/implementazione può specificare argomenti standard.

```
1  int my_substring(string s, int start, int length=10);  
2  
3  int my_substring(string s, int start=1, int length){  
4      return s.substr(start, length);  
5  }  
6  //...  
7  s.my_substring(string("Hello world"));
```

Esempio svolto

23_04.cpp

Visibilita' di una variabile (Scope):

porzione del programma (scope) in cui la variabile può essere referenziata.

Ciclo di vita di una variabile: periodo di tempo durante il quale la variabile esiste in memoria.

Scope e durata delle variabili

In C/C++ lo scope è legato ai **blocchi di codice**. ES:

```
1 void ClasseA::m1(int x){
2     int y = 2*x;
3     {
4         double z = y/3;
5         while (y>0){
6             z = z / (1+x);
7             y--;
8         } //end while
9     } //end blocco
10 }
```

Scope di variabile y: l'intero corpo di m1!

Scope di z: limitato al blocco 3-9!

Un qualunque blocco può contenere dichiarazioni di variabili al suo interno, tali variabili si dicono **locali al blocco**.

Una variabile locale al blocco è **visibile**:

- nel blocco in cui è definita
- in ogni altro blocco contenuto nel blocco di
definizione



Ciclo di vita di una variabile locale al blocco:

- creazione contestuale all'ingresso nel blocco
- distruzione contestuale all'uscita dal blocco

Ridefinire variabile di blocco esterno? OK

```
1  {
2      int a = 2;
3      //...
4  {
5      int a = 100; // OK ridefinire a!
6      cout << a << endl; // a==100!
7  }
8      a+=10;
9      cout << a << endl; //a==12
10 }
```

La variabile a interna al blocco 5 – 8 “oscura” la variabile a creata all’ingresso del blocco 2 – 11.

Scope e durata delle variabili

Non è possibile ridefinire parametro formale

```
1 double f(int x, double y){
2     double y = 0.2; //Compil. ERR!
3     double result = x * y;
4 }
```

Ma..

```
1 double f(int x, double y){
2     { //OK ridefinizione y all'interno del blocco
3         double y = 0.2; //OK
4         cout << y << endl;
5     }
6 }
```

Esempio svolto

23_05.cpp

Variabili globali

```
1 #include <iostream>
2 //...
3
4 int x,y,z; // SCOPE globale !!
5 //...
6
7 int main(){
8     cout << x << " ," << y << " ," << z << endl;
9 }
```

Variabili x,y, e z definite fuori da qualsiasi blocco → **Scope globale.**

Funzioni aventi scope globale: metodi non associati ad alcuna classe.

NB: Usare variabili globali altamente sconsigliato in C/C++!

NB: Usare funzioni globali non più necessario in C++...

In generale, nella OOP (Object Oriented Programming):

- È bene **incapsulare i dati all'interno delle classi** (private!)
- È bene **Associare le funzioni alle classi**, ovvero i metodi, in modo che possano **operare sui dati della classe**.

Funzioni inline

```
1 inline double f(double y){
2     return y*y;
3 }
4 //...
5 f(0.5);
6 //...
```

Se presente la keyword `inline`, il compilatore sostituisce ad ogni chiamata alla funzione (ES: linea 5) il codice del corpo di `f()`.

Perchè? **Maggiore efficienza!** Inoltre il programmatore continua a ragionare in termini di funzioni/metodi.. (OK!)

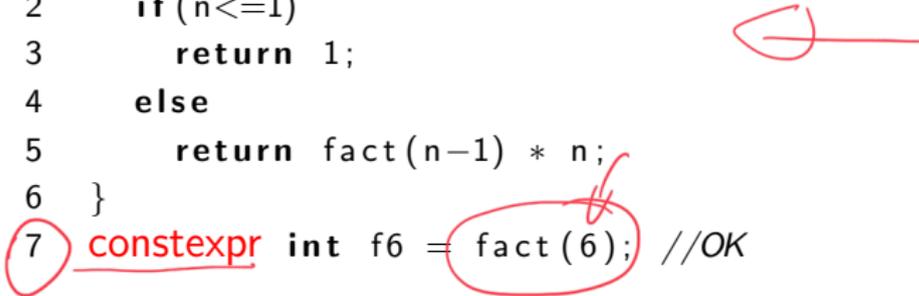
Ancora sulle funzioni inline..

```
1  inline int fact(int n){
2      if (n<=1)
3          return 1;
4      else
5          return fact(n-1) * n;
6  }
```

In questo caso **non è assicurato che il compilatore** inserisca il numero 720 al posto della chiamata `fact(6)`.

Keyword constexpr

```
1  constexpr int fact(int n){
2      if (n<=1)
3          return 1;
4      else
5          return fact(n-1) * n;
6  }
7  constexpr int f6 = fact(6); //OK
```



La direttiva `constexpr` indica al compilatore di valutare l'espressione a tempo di compilazione.

Eventuali funzioni usate nella espressione vanno dichiarate `constexpr`.

Esempio svolto

`23_06.cpp`

- **public**: visibile sia all'interno che all'esterno della classe. In particolare, i metodi public costituiscono l'interfaccia della classe.
- **private**: visibile solo all'interno della classe. In particolare, i metodi private si dicono metodi di servizio

Modificatori di accesso per i membri di una classe

- **protected**: all'esterno della classe sono a tutti gli effetti private. Ma differisce dai membri private nel contesto dell'ereditarietà. Sarà visto in seguito..
- **non specificato**: equivalente a **private**

Modificatori di accesso per i membri di una classe

```
1  class A{
2      //unqualified!
3      int y; // private
4      void f(); //private
5
6      private:
7          int x, z;
8
9      public:
10         int getX();
11         int getY();
12     }
```

Implementazione di Costruttori

```
1  class ClasseX {
2      int x;
3
4      public :
5          ClasseX(int x);
6              // ...
7          int getX ();
8          int getY ();
9  }
```

Il costruttore prende **il nome della classe**, e **non dichiara alcun tipo di ritorno**.

Di conseguenza il compilatore lo riconosce come tale e genera istruzioni relative alla sua **invocazione automatica durante la creazione della istanza**.

I parametri sono usati per **inizializzare lo stato dell'oggetto**.

Implementazione di Costruttori

```
1 ClasseX :: ClasseX (int x) {  
2     this -> x = x;  
3 }  
4 //..  
5 int main() {  
6     ClasseX *t = new ClasseX (10);  
7     ClasseX t1 (20);  
8     ClasseX t2 = ClasseX (30);  
9 }
```

Linea 6 allocazione dinamica, linea 7 e 8 allocazione automatica.

Esempio svolto

23_06.cpp

Implementazione di Costruttori

È obbligatorio fornire un costruttore? NO!

```
1  class ClasseX{
2      int x;
3      public:
4          int getX();
5  }
6  //..
7  int main(){
8      ClasseX t2; //valore iniziale di x??
9  }
```

Il compilatore genera un **Costruttore di default** (corpo vuoto!). NB: dati della classe non inizializzati (valori iniziali “random”).

Alla luce degli argomenti affrontati finora, rivedere gli esempi

- Moneta (moneta.cpp)
- Dado (dado.cpp)
- Serbatoio (serbatoio.cpp)

FINE