



UNIVERSITÀ  
degli STUDI  
di CATANIA

DIPARTIMENTO DI  
MATEMATICA E INFORMATICA

# Funzioni in C++

Alessandro Ortis

Image Processing Lab - [iplab.dmi.unict.it](http://iplab.dmi.unict.it)

[www.dmi.unict.it/ortis/](http://www.dmi.unict.it/ortis/)



# Struttura di una funzione

- sottoprogramma che può essere mandato in esecuzione da altri programmi
- evita ripetizioni di codice e facilita la programmazione rendendola modulare, cioè rendendo possibile riutilizzare infinite volte sottoprogrammi già esistenti
- ha un nome (identificativo) che serve al programma chiamante per mandarla in esecuzione
- funzioni definite all'interno di un programma possono essere mandate in esecuzione anche da altri programmi
- raggruppando funzioni ben collaudate in librerie tematiche, altri programmi potranno utilizzarle facilmente comprimendo i tempi di sviluppo del software e rendendolo più affidabile

# Componenti di una funzione

## ● **definizione**

- nome della funzione
- tipo e *nome* dei dati presi in input (*argomenti* o *parametri formali*)
- tipo di dato del risultato
- programma (*corpo della funzione*)

## ● **dichiarazione** (*prototipo*)

- nome della funzione
- tipo dei dati presi in input
- tipo di dato del risultato

## ● **chiamata**

- nome della funzione
- dati presi in input

# Definizione

tipo  
del  
risultato

```
int mcd (int alfa, int beta)
{
    int resto;
    while (beta != 0)
    {
        resto = alfa % beta;
        alfa = beta; beta = resto;
    }
    return alfa;
}
```

nome

argomenti – nome e tipo

intestazione

variabile locale

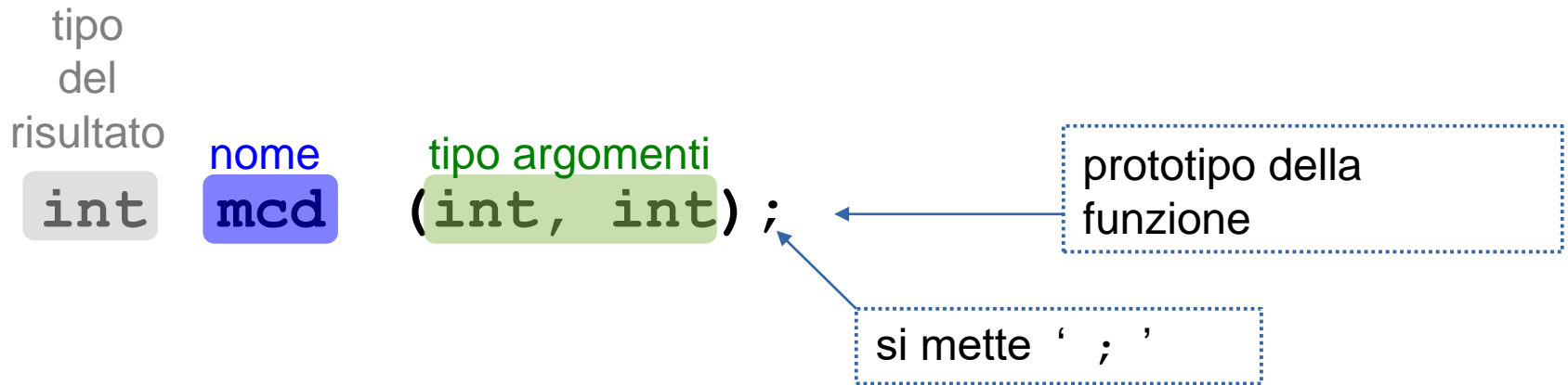
esistono solo durante l'esecuzione della funzione

se manca non si restituisce alcun risultato, la funzione si chiama "procedura" ed il tipo restituito deve essere **void**

non si mette ' ; '

The diagram illustrates the components of a C function definition. The function signature 'int mcd (int alfa, int beta)' is shown with 'int' as the return type, 'mcd' as the function name, and '(int alfa, int beta)' as the parameter list. Annotations explain that the return type is the 'tipo del risultato', the name is the 'nome', and the parameter list is 'argomenti – nome e tipo'. A callout 'intestazione' points to the entire signature. The function body contains a local variable 'int resto;' labeled as 'variabile locale' and a loop. A callout explains that local variables 'esistono solo durante l'esecuzione della funzione'. The 'return alfa;' statement is annotated with 'se manca non si restituisce alcun risultato, la funzione si chiama "procedura" ed il tipo restituito deve essere void'. Finally, the closing brace '}' is annotated with 'non si mette ' ; ''.

# Dichiarazione (prototipo)



- serve per dire al compilatore che il programma utilizzerà una funzione che ha quel **nome**, prende in input quel numero di **parametri** che saranno valori di quei **tipi**, e restituirà in output un valore di quel **tipo**
- il codice da eseguire sarà specificato nella definizione che comparirà:
  - dopo il programma principale, oppure
  - in un altro file che sarà collegato a questo in fase di collegamento

# Funzioni senza argomenti e procedure

- una **procedura** è una funzione che restituisce nulla

```
void scriviris(int a, int b, int c)
{
    cout << "il MCD fra " << a << " e ";
    cout << b << " è " << c << "\n";
}
```

- una funzione può anche non avere argomenti

```
void scrivi_licenza ()
{
    cout << "Contratto di licenza d'uso\n";
    cout << " ... 2002 \n";
}
```

# Programmazione modulare

- esempio: leggere una lista di caratteri dalla tastiera, metterli in ordine alfabetico e visualizzarli sullo schermo: funzione `main()` che chiama altre funzioni per realizzare quei sottocompiti

```
int main(){
    legge_caratteri(); // Chiama la funzione che legge i caratteri
    ordinare();         // Chiama la funzione che li ordina alfabeticamente
    scrivi_caratteri(); // Chiama la funzione che li scrive sullo schermo
    return 0;          // restituisce il controllo al sistema operativo
}

int legge_caratteri(){
    ...                // Codice per leggere una sequenza di caratteri dalla
tastiera
    return 0;          // restituisce il controllo al main()
}

int ordinare(){
    ...                // Codice per ordinare alfabeticamente la sequenza dei
caratteri
    return 0;          // restituisce il controllo al main()
}

int scrivi_caratteri(){
    ...                // Codice per visualizzare sullo schermo la sequenza
ordinata
    return 0;          // restituisce il controllo al main()
}
```

# Ricapitolando ..

- ***tipo del risultato***: tipo del dato che la funzione restituisce
- ***argomenti formali***: lista dei parametri tipizzati che la funzione richiede al programma che la chiama; vengono scritti nel formato:  
`tipo1 parametro1, tipo2 parametro2, ...`
- ***corpo della funzione***: è il sottoprogramma vero e proprio; si racchiude tra parentesi graffe senza punto e virgola dopo quella di chiusura
- ***passaggio di parametri***: quando viene mandata in esecuzione una funzione le si passano i suoi argomenti "attuali" e questo passaggio può avvenire o "*per valore*" o "*per riferimento*"
- ***dichiarazioni locali***: gli argomenti formali, le costanti e le variabili definite dentro la funzione sono ad essa locali, cioè esistono solo mentre la funzione è in esecuzione e non sono accessibili fuori di essa
- ***valore restituito dalla funzione***: mediante la parola riservata `return` si può ritornare il valore restituito dalla funzione al programma chiamante
- non si possono dichiarare funzioni annidate, ma una funzione può mandare in esecuzione un'altra funzione



# Tipo del dato di ritorno

- il tipo può essere uno dei tipi semplici, come `int`, `char` o `float`, un puntatore a qualunque tipo C++, o un tipo `struct`

```
double media(double x1, double x2)    // ritorna un tipo double
float funz0() {...}                  //ritorna un float
char* funz1() {...}                 //ritorna un puntatore a char
int* funz3() {...}                  //ritorna un puntatore ad int
struct InfoPersona CercaRegistro(int num_registro);
int max(int x, int y) // ritorna un tipo int
```

- funzioni che non restituiscono risultati si utilizzano solo come *subroutines*, vengono dette *procedure* e si specificano indicando la parola riservata `void` come tipo di dato restituito

```
void scrivi_risultati(float totale, int num_elementi);
```

# Risultati di una funzione

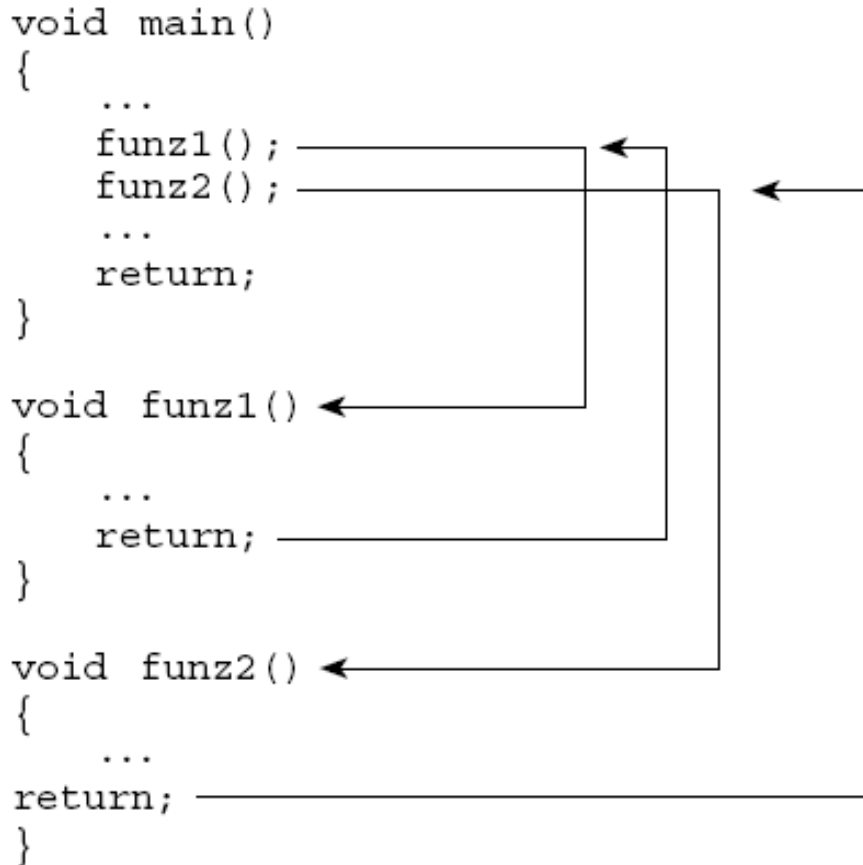
- una funzione *può* restituire un valore mediante l'istruzione `return` la cui sintassi è:

```
return(espressione);  
return espressione;  
return; // caso di una procedura, si può omettere
```

- `espressione` deve essere ovviamente del tipo definito come restituito dalla funzione; ad esempio, non si può restituire un valore `int` se il tipo di ritorno è un puntatore; tuttavia, se si restituisce un `int` e il tipo di ritorno è un `float`, il compilatore lo converte automaticamente
- una funzione può avere più di un'istruzione `return` e termina non appena s'esegue la prima di esse
- se non s'incontra alcun'istruzione `return` l'esecuzione continua fino alla parentesi graffa finale del corpo della funzione
- errore tipico: dimenticare l'istruzione `return` o metterla dentro una sezione di codice che non verrà eseguita; in questi casi il risultato della funzione è imprevedibile e probabilmente porterà a risultati scorretti

# Chiamata di una funzione

- una funzione va in esecuzione quando viene *chiamata* (o *invocata*) dalla funzione principale `main()` o da un'altra funzione
- la funzione che chiama un'altra funzione si denomina *funzione chiamante* e la funzione mandata in esecuzione si denomina *funzione chiamata*



# Passaggio di argomenti

- se la funzione chiamata ha dei parametri, bisogna passarle una lista di *valori* in corrispondenza di tipo
- si possono passare anche identificatori di costanti o di variabili, ma ovviamente non verranno passati alla funzione i contenitori, cioè i *left values*, ma solo i contenuti, cioè i *right values*
- il passaggio di argomenti ad una funzione si dice quindi essere fatto “***per valore***”

# Invocazione e stack

La porzione di memoria riservata allo stoccaggio dei dati utili alla esecuzione delle istruzioni contenute nel corpo delle funzioni e denominata stack (pila).

Lo stack è una struttura in cui i dati vengono inseriti e prelevati in base al meccanismo LIFO (Last In - First Out).

Il singolo dato viene depositato (push) sempre sul top dello stack.

Si può prelevare un dato alla volta (pop), solo dal top dello stack.

# Invocazione e stack

## Segmento STACK



```
1  void foo(int x){  
2      double c, d;  
3      // ...  
4  }  
5  int main(){  
6      int a, b;  
7      // ...  
8      foo(a);  
9      // ..  
10 }
```

# Passaggio “per valore”

non vengono passate le variabili alla funzione ma solo i valori in esse contenuti; per esempio, si consideri la seguente procedura:

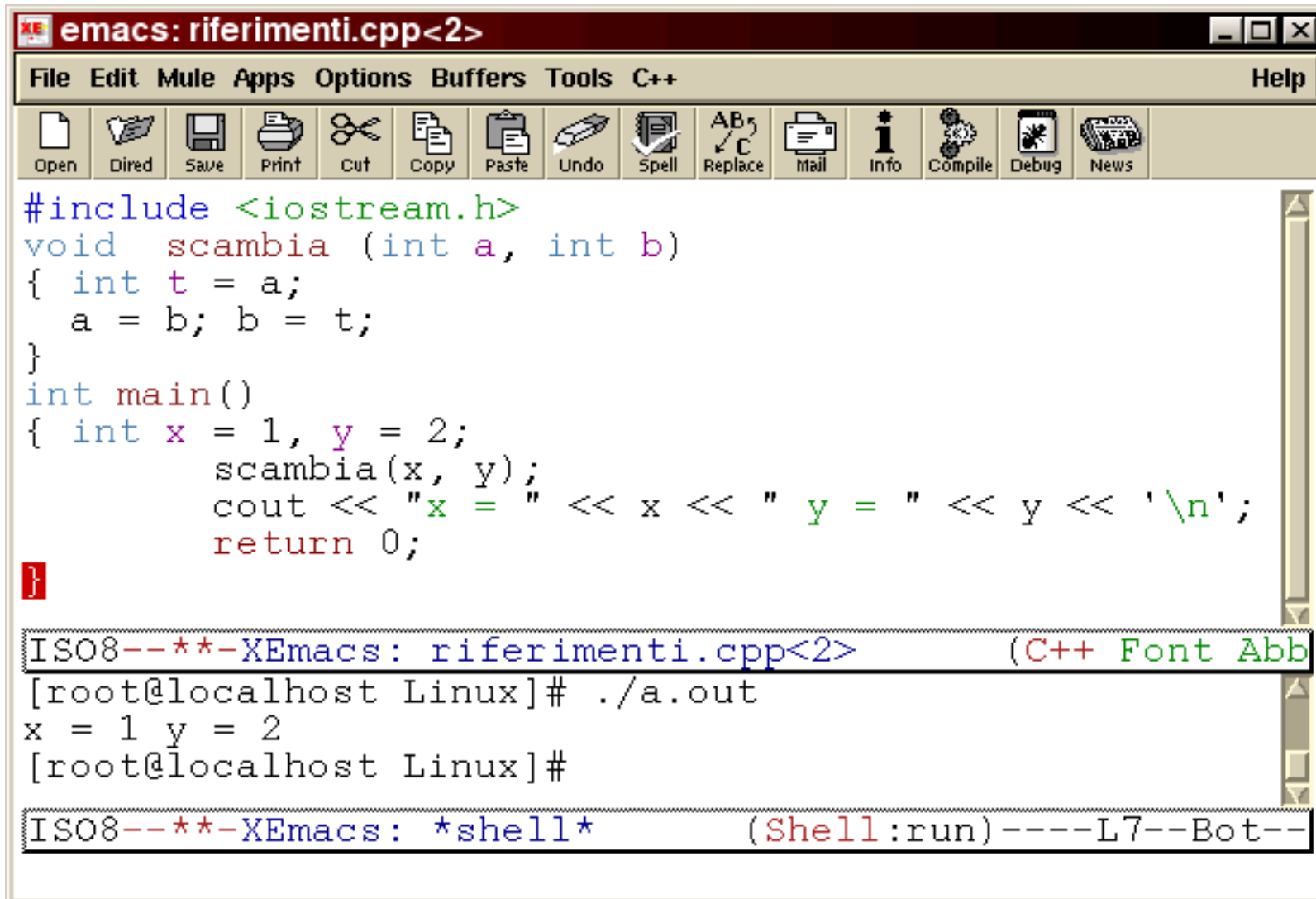
```
void scambia_valori_variabili(int a, int b)
{
    int aux; // definizione della variabile locale ausiliaria
    aux = a; // aux prende il valore del parametro a
    a = b;    // a prende il valore del parametro b
    b = aux; // b prende il valore della variabile locale aux
}
```

la chiamata:

```
int x=4, y=5;
scambia_valori(x, y);
```

non scambia i valori delle variabili  $x$  ed  $y$  che sono servite solo per passare ad  $a$  e  $b$  i loro rispettivi valori

# Esempio “passaggio per valore”



The screenshot shows an Emacs window titled "emacs: riferimenti.cpp<2>". The menu bar includes File, Edit, Mule, Apps, Options, Buffers, Tools, C++, and Help. The toolbar contains icons for Open, Dired, Save, Print, Cut, Copy, Paste, Undo, Spell, Replace, Mail, Info, Compile, Debug, and News. The main text area contains the following C++ code:

```
#include <iostream.h>
void scambia (int a, int b)
{ int t = a;
  a = b; b = t;
}
int main()
{ int x = 1, y = 2;
  scambia(x, y);
  cout << "x = " << x << " y = " << y << '\n';
  return 0;
}
```

Below the code is a terminal window showing the execution of the program:

```
ISO8--*-XEmacs: riferimenti.cpp<2> (C++ Font Abb
[root@localhost Linux]# ./a.out
x = 1 y = 2
[root@localhost Linux]#
```

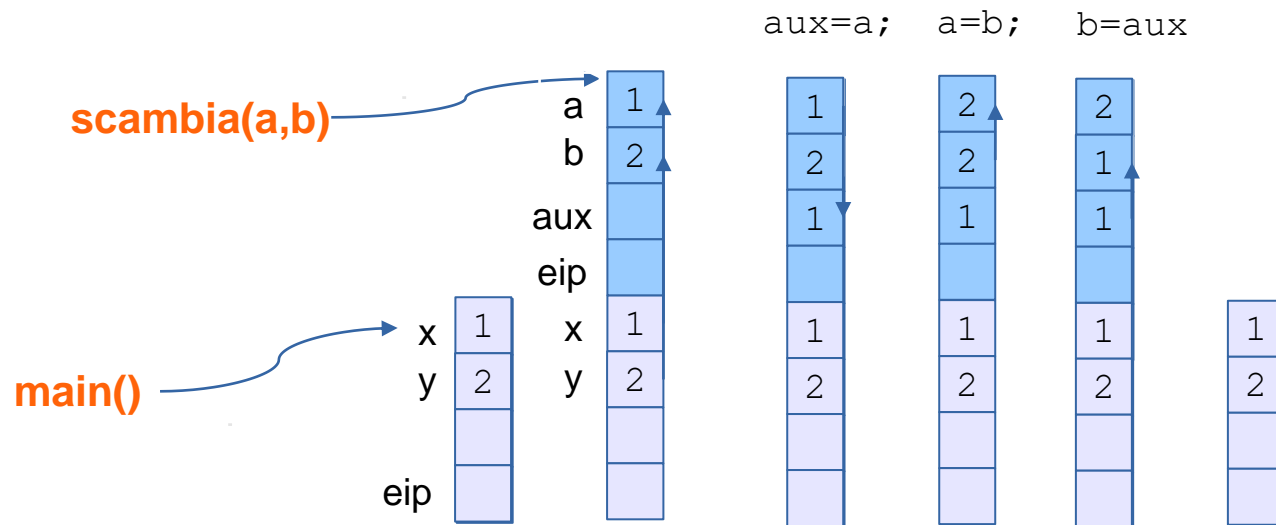
At the bottom, there is a shell prompt:

```
ISO8--*-XEmacs: *shell* (Shell:run)----L7--Bot--
```



# Passaggio “per valore” e stack

- la funzione è un sottoprogramma
- i suoi parametri come le sue variabili vanno nello stack
- ogni manipolazione sulle variabili locali o sui parametri formali non ha alcun effetto sui parametri attuali (che potranno essere variabili della funzione chiamante oppure variabili globali)



# Passaggio “per indirizzo”

La funzione riceve l'indirizzo del dato (il puntatore), quindi può operare modifiche al dato della funzione chiamante.

```
void foo(int *a) {  
    *a = 100;  
}  
int main() {  
    int x = 2;  
    foo(&x);  
    cout << x; // stampa 100  
}
```

# Visibilità variabili

Allocazione automatica:

- Dichiarazione di una variabile locale ad una funzione
- Scope/visibilità limitato al blocco di codice in cui è stata dichiarata
- Ciclo di vita del blocco allocato termina con la fine dell'esecuzione del blocco in cui viene dichiarata
- L'area di memoria usata è denominata STACK

```
void foo() {  
    int a = 100; // visibile solo in foo()  
}
```

# Visibilità variabili

## Allocazione dinamica:

- Effettuata mediante operatore new in qualsiasi punto del programma.
- L'area di memoria usata è denominata HEAP.
- Ciclo di vita del blocco di memoria termina con invocazione di operatore delete sul puntatore.

```
int *p = new int(2);  
// ...  
delete p; // deallocazione della cella
```

# Visibilità variabili

## Allocazione statica:

- Dichiarazione di variabili al di fuori da qualunque blocco.
- Il segmento di memoria ospitante è detto segmento DATA.
- Ciclo di vita / scope: inizia e termina con il programma stesso.
- La variabile si dice globale.

```
double pi = 3.14;  
  
int main() {  
    cout << pi;  
}
```

# Argomenti di default

- è possibile definire funzioni in cui alcuni argomenti assumono un valore di *default*; se alla chiamata non viene passato alcun valore per quel parametro allora la funzione assumerà per lui il valore di default stabilito nell'intestazione

- gli argomenti di default devono raggrupparsi a destra nell'intestazione

- il valore di default deve essere un'espressione costante

```
char funzdef(int arg1=1, char c='A', float f_val=45.7f);
```

- si può chiamare `funzdef` con qualunque delle seguenti istruzioni:

```
funzdef(9, 'Z', 91.5); // annulla i tre argomenti di default
funzdef(25, 'W'); // annulla i due primi argomenti di default
funzdef(50); // annulla il primo argomento di default
funzdef(); // utilizza i tre argomenti di default
```

- se si omette un argomento bisogna omettere anche tutti quelli alla sua destra; la seguente chiamata non è corretta:

```
funzdef( , 'Z', 99.99);
```

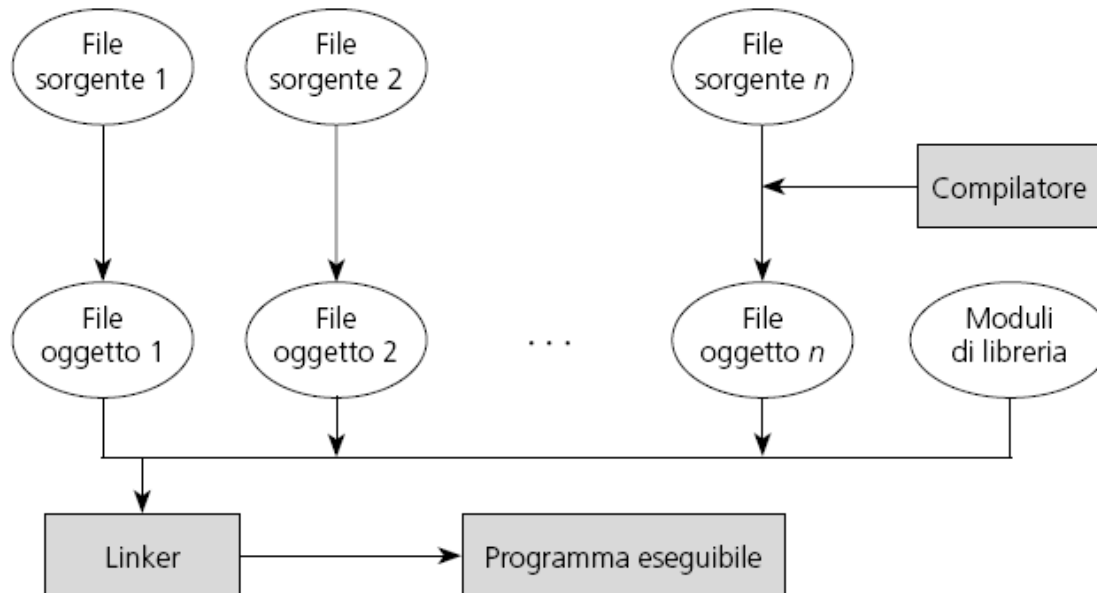
# Funzioni `inline`

- servono per aumentare la velocità del programma
- convenienti quando la funzione si richiama parecchie volte nel programma e il suo codice è breve
- il compilatore ricopia realmente il codice della funzione in ogni punto in cui essa viene invocata
- il programma verrà così eseguito più velocemente perché non si dovrà eseguire il codice associato alla chiamata alla funzione
- tuttavia, ogni ripetizione della funzione richiede memoria, perciò il programma aumenta la sua dimensione
- per creare una funzione in linea si deve inserire la parola riservata `inline` all'inizio dell'intestazione

```
inline int sommare15(int n) {return (n+15);}
```

# Compilazione modulare

- i programmi grandi sono più facili da gestire se si dividono in vari files sorgenti, anche chiamati *moduli*, ognuno dei quali può contenere una o più funzioni; questi moduli verranno poi compilati separatamente ma linkati assieme
- per ridurre il tempo di compilazione, ad ogni ricompilazione verranno in realtà ricompilati solo i moduli che sono stati modificati





# Compilazione modulare

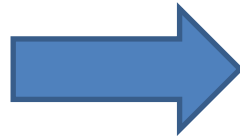
È buona pratica quindi:

- Raccogliere le **dichiarazioni dei prototipi** delle funzioni in appositi file header (es. modulo1.h). Un header in genere contiene:
  - Direttive #define e #include
  - Dichiarazioni di costanti globali
  - Prototipi di funzioni
  - Dichiarazioni di classi
- Raccogliere la **definizione delle funzioni** in un modulo sorgente (es. modulo1.cpp)
- Includere la direttiva #include “modulo1.h” in ogni file sorgente in cui si fa uso di tali funzioni.

# Compilazione modulare

I moduli contenenti una o più funzioni si possono compilare separatamente in uno o più file oggetto da assemblare in un momento successivo.

```
g++ -c modulo1.cpp  
g++ -c modulo2.cpp  
...  
g++ -c modulok.cpp  
g++ -c main.cpp
```



```
modulo1.o  
modulo2.o  
...  
modulok.o  
main.o
```

```
g++ main.o modulo1.o modulo2.o ... modulok.o
```

oppure

```
g++ main.cpp modulo1.cpp -o esegui
```


# Esempio extern

Con **extern** si indica al compilatore che la variabile è *definita* in un altro file sorgente che sarà *linkato* assieme

file1.cpp

```
#include <iostream>
using namespace std;
extern int x;
void stampa()
{
    cout << "x vale \n";
    cout << x << endl;
}
```

non può  
essere  
inizializzata



file2.cpp

```
void stampa();
int x=5;

int main()
{
    stampa();
}
```

```
g++ file1.cpp file2.cpp -o prova
```

```
./prova
```

```
x vale
```

```
5
```

# Funzioni di libreria

Tutte le versioni del linguaggio C++ contengono una grande raccolta di funzioni di libreria per operazioni comuni; esse sono raccolte in gruppi definite in uno stesso *header file*, esempi:

- I/O standard
- matematiche
- routines standard
- visualizzare finestra di testo
- di conversione (di caratteri e stringhe)
- di diagnostica (debugging incorporato)
- di manipolazione di memoria
- controllo del processo
- classificazione (ordinamento)
- cartelle
- data e ora
- di interfaccia
- ricerca
- manipolazione di stringhe
- grafici

# Funzioni di carattere

- verifiche alfanumeriche:

`isalpha(c)` ritorna `true` se e solo se `c` è maiuscola o minuscola  
`islower(c)` ritorna `true` se e solo se `c` è una lettera minuscola  
`isupper(c)` ritorna `true` se e solo se `c` è una lettera maiuscola  
`isdigit(c)` ritorna `true` se e solo se `c` è una cifra (cioè un carattere da 0 a 9)  
`isxdigit(c)` ritorna `true` se e solo se `c` è una cifra esadecimale (0 ÷ 9, A ÷ F)  
`isalnum(c)` ritorna `true` se e solo se `c` è una cifra o un carattere alfabetico

- verifiche di caratteri speciali:

`iscntrl(c)` ritorna `true` se e solo se `c` è un *carattere di controllo* (ASCII 0 a 31)  
`isgraph(c)` ritorna `true` se e solo se `c` non è un carattere di controllo, eccetto lo spazio  
`isprint(c)` ritorna `true` se e solo se `c` è un carattere stampabile (ASCII 21 ÷ 127)  
`ispunct(c)` ritorna `true` se e solo se `c` è qualunque carattere di interpunzione  
`isspace(c)` ritorna `true` se e solo se `c` è uno spazio, `\n`, `\r`, `\t` o tabulazione verticale `\v`

- conversione caratteri:

`tolower(c)` converte la lettera `c` in minuscola, se non lo è già  
`toupper(c)` converte la lettera `c` in maiuscola, se non lo è già

# Funzioni numeriche

- **matematiche:**

`ceil(x)` arrotonda all'intero più alto

`fabs(x)` restituisce il valore assoluto di  $x$  (un valore positivo)

`floor(x)` arrotonda all'intero più basso

`pow(x, y)` calcola  $x$  elevato ad  $y$

`sqrt(x)` restituisce la radice quadrata di  $x$

- **trigonometriche**

`acos(x)` calcola l'arco coseno di  $x$

`asin(x)` calcola l'arco seno di  $x$

`atan(x)` calcola l'arco tangente di  $x$

`atan2(x, y)` calcola l'arco tangente di  $x$  diviso  $y$

`cos(x)` calcola il coseno dell'angolo  $x$  ( $x$  si esprime in radianti)

`sin(x)` calcola il seno dell'angolo  $x$  ( $x$  si esprime in radianti)

`tan(x)` calcola la tangente dell'angolo  $x$  ( $x$  si esprime in radianti)

- **logaritmiche ed esponenziali**

`exp(x)` calcola l'esponenziale  $e^x$

`log(x)` calcola il logaritmo naturale di  $x$

`log10(x)` calcola il logaritmo decimale di  $x$

# Funzioni varie

- aleatorie

`rand()` genera un numero aleatorio fra 0 e `RAND_MAX`

`randomize()` inizializza il generatore di numeri aleatori con un seme aleatorio ottenuto a partire da una chiamata alla funzione `time`

`srand(seme)` inizializza il generatore di numeri aleatori in base al valore dell'argomento `seme`

`random(num)` restituisce un numero aleatorio da 0 a `num-1`

- di data ed ora

`clock(void)` restituisce il tempo di CPU in secondi trascorso dall'inizio dell'esecuzione del programma

`time(ora)` restituisce il numero di secondi trascorsi dalla mezzanotte (00:00:00) del primo gennaio 1970; questo valore di tempo si mette nella posizione puntata dall'argomento `ora`

# Sovraccaricamento delle funzioni

- *overloading*  
permette di dare lo stesso nome a funzioni con almeno un argomento di tipo diverso e/o con un diverso numero di argomenti
- C++ determina quale tra le funzioni sovraccaricate deve chiamare, in funzione del numero e del tipo dei parametri passati
- regole
  - se esiste, si seleziona la funzione che mostra la corrispondenza esatta tra il numero ed i tipi dei parametri formali ed attuali
  - se tale funzione non esiste, si seleziona una funzione in cui il matching dei parametri formali ed attuali avviene tramite una conversione automatica di tipo
  - la corrispondenza dei tipi degli argomenti può venire anche forzata mediante *casting*



# Overloading delle funzioni

```
#include <iostream>
using namespace std;

int somma(int a, int b) { return a + b; }
int somma(int a, int b, int c){
    return a + b + c;
}

int main(void)
{
    cout << somma(10, 20) << endl;
    cout << somma(12, 20, 23);
    return 0;
}
```

# Esercizi

1. Scrivere una funzione che calcola il quoziente della divisione intera tra due numeri interi mediante somme e sottrazioni.
2. Un numero  $n$  si dice perfetto se la somma dei suoi divisori (incluso 1) escludendo se stesso coincide con se stesso. Scrivere una funzione che determina se un numero è perfetto. Es. 6 è perfetto  $1+2+3=6$

# Esercizi

Definire un convertitore da Celsius a Fahrenheit

$$^{\circ}\text{F} = (^{\circ}\text{C} \times 9/5) + 32$$

$$^{\circ}\text{C} = (^{\circ}\text{F} - 32) \times 5/9$$

Sfruttare la programmazione modulare definendo un modulo per l'implementazione delle due conversioni ed un file separato per il main.

Nel main, chiedere all'utente di inserire un valore di temperatura seguito da una delle due unità di misura, verificare l'input e successivamente mostrare il risultato della conversione. Ad esempio:

-30 C   -22.0 F

-20 C   -4.0 F

-10 C   14.0 F