



Capitolo 5

Funzioni

pag. 123-160

Presenta: Prof. Misael Mongiovì



struttura di una funzione

- sottoprogramma che mette in corrispondenza funzionale certi dati presi in input con altri dati prodotti in output, e può essere mandato in esecuzione da altri programmi
- evita ripetizioni di codice e facilita la programmazione rendendola modulare, cioè rendendo possibile riutilizzare infinite volte programmi già fatti all'interno di altri programmi
- ha un nome 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

Capitolo 5. Funzioni



```
emacs: *shell*
File Edit Mule Apps Options Buffers Tools Complete In/Out Signals Help
Open Dired Save Print Cut Copy Paste Undo Spell Replace Mail Info Compile Debug News

// mcd.cpp
#include <iostream.h>

int main()
{
    int alfa, beta, resto, i, n;
    cout << "Quante coppie di numeri vuoi esaminare?\n";
    cin >> n;
    for (i = 1; i <= n; i++)
    {
        cout << "coppia numero " << i << "?\n";
        cin >> alfa >> beta;
        cout << "il M.C.D. fra " << alfa << " e " << beta
            << " e` ";
        while (beta != 0)
        {
            resto = alfa % beta;
            alfa = beta;
            beta = resto;
        }
        cout << alfa << '\n';
    }
    return 0;
}

ISO8:T-----XEmacs: mcd2.cpp (C++ Font Abb
[root@localhost programmi]# ./a.out
Quante coppie di numeri vuoi esaminare?
1
coppia numero 1?
23 45
il M.C.D. fra 23 e 45 e` 1
[root@localhost programmi]#

ISO8--*-XEmacs: *shell* (Shell:run)----L9--Bot-----
```

questa parte del codice svolge un compito di interesse generale (calcolo del MCD) che potrebbe essere svolto anche in ulteriori punti nel programma o addirittura in altri programmi

modulo autonomo capace di accettare diversi input



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**

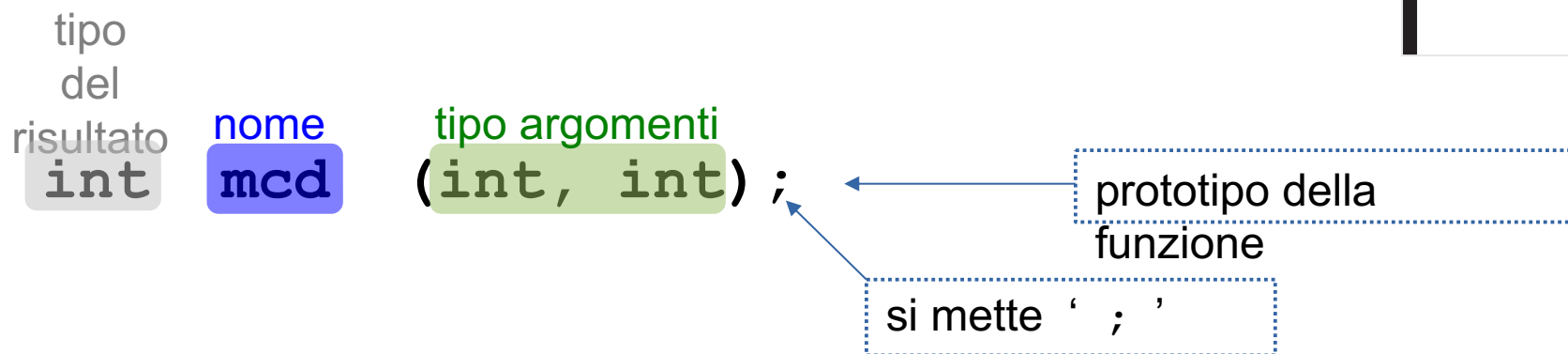
- nome della funzione
- dati presi in input

definizione

```
tipo  
del  
risultato  nome      argomenti – nome e tipo  
int      mcd      (int alfa, int beta) ← intestazione  
{  
    int resto ← variabile locale ← esistono solo durante  
    while (beta != 0) l'esecuzione della funzione  
    {  
        resto = alfa % beta;  
        alfa = beta; beta = resto;  
    }  
    return alfa; ← se manca non si restituisce alcun risultato,  
} ← non si mette ' ; ' la funzione si chiama "procedura" ed il tipo  
    restituito deve essere void
```



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

chiamata

- nome della funzione seguito dagli argomenti attuali
- stesso numero degli argomenti formali
- corrispondenza per posizione
- stesso tipo del corrispondente argomento formale (a meno di conversioni implicite)

```
emacs: *shell*
File Edit Mule Apps Options Buffers Tools Complete In/Out Signals Help
Open Dired Save Print Cut Copy Paste Undo Spell Replace Mail Info Compile Debug News

// mcd.cpp
#include <iostream.h>
int mcd(int alfa, int beta)
{
    int resto;
    while (beta != 0)
    {
        resto = alfa % beta;
        alfa = beta; beta = resto;
    }
    return alfa;
}
int main()
{
    int a, b, i, n;
    cout << "Quante coppie di numeri "
           "vuoi esaminare?\n";
    cin >> n;
    for (i = 1; i <= n; i++)
    {
        cout << "coppia numero " << i << "?\n";
        cin >> a >> b;
        cout << "il M.C.D. fra " << a << " e " << b
              << " e` " << mcd(a, b) << '\n';
    }
    return 0;
}

ISO8:T-----XEmacs: mcd.cpp (C++ Font Abbrev)-----L1--Al
[root@localhost programmi]# ./a.out
Quante coppie di numeri vuoi esaminare?
1
coppia numero 1?
23 45
il M.C.D. fra 23 e 45 e` 1
[root@localhost programmi]#

ISO8--**--XEmacs: *shell* (Shell:run)-----L17--Bot-----
```

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";
}
```



Capitolo 5. Funzioni



int main()

- il `main()` stesso altro non è che la definizione di una funzione che restituisce (al sistema operativo) un numero intero (che sarà 0 se è andato tutto bene)

```
emacsv: *shell*
File Edit Mule Apps Options Buffers Tools Complete In/Out Signals Help
Open Dired Save Print Cut Copy Paste Undo Spell Replace Mail Info Compile Debug News

// mcd.cpp
#include <iostream.h>
int mcd(int alfa, int beta)
{
    int resto;
    while (beta != 0)
    {
        resto = alfa % beta;
        alfa = beta; beta = resto;
    }
    return alfa;
}

int main()
{
    int a, b, i, n;
    cout << "Quante coppie di numeri "
           "vuoi esaminare?\n";
    cin >> n;
    for (i = 1; i <= n; i++)
    {
        cout << "coppia numero " << i << "?\n";
        cin >> a >> b;
        cout << "il M.C.D. fra " << a << " e " << b
              << " e` " << mcd(a, b) << '\n';
    }
    return 0;
}

ISO8:T-----XEmacs: mcd.cpp (C++ Font Abbrev)-----L1--Al
[root@localhost programmi]# ./a.out
Quante coppie di numeri vuoi esaminare?
1
coppia numero 1?
23 45
il M.C.D. fra 23 e 45 e` 1
[root@localhost programmi]#

ISO8--**--XEmacs: *shell* (Shell:run)-----L17--Bot-----
```



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 CercareRegistro(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** `scrive_risultati(float totale, int num_elementi);`



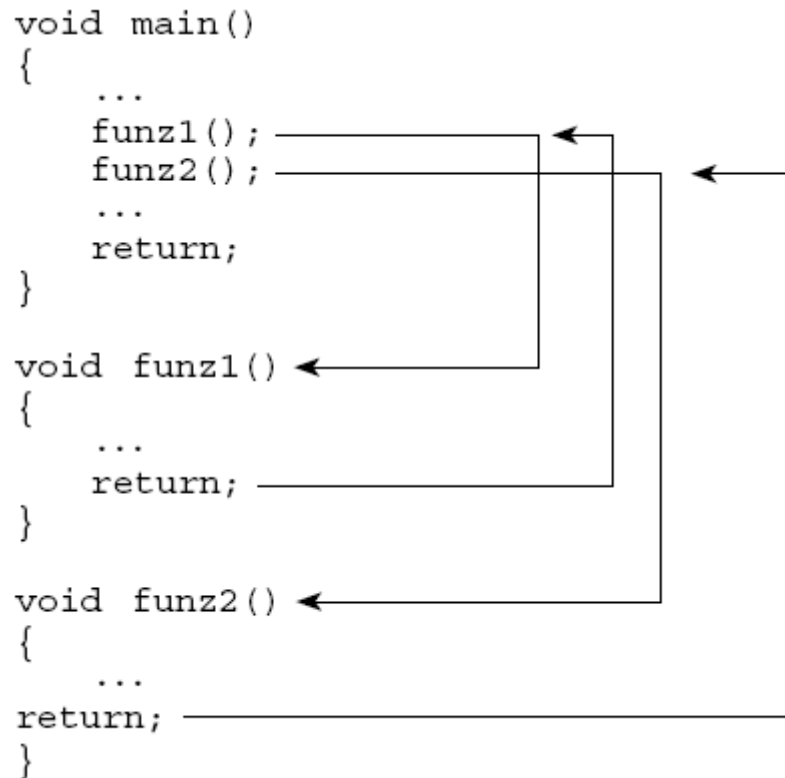
risultati di una funzione

- 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

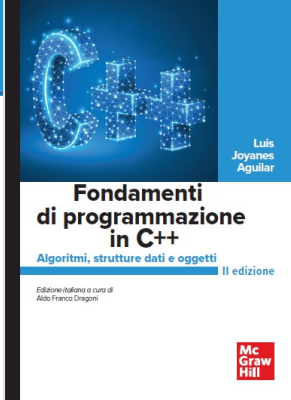
- una funzione va in esecuzione quando viene *chiamata* (o *invocata*) dalla funzione principale `main()` o da un'altra funzione
- la funzione che chiama un'altra funzione si denomina *funzione chiamante* e la funzione mandata in esecuzione si denomina *funzione chiamata*





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***”

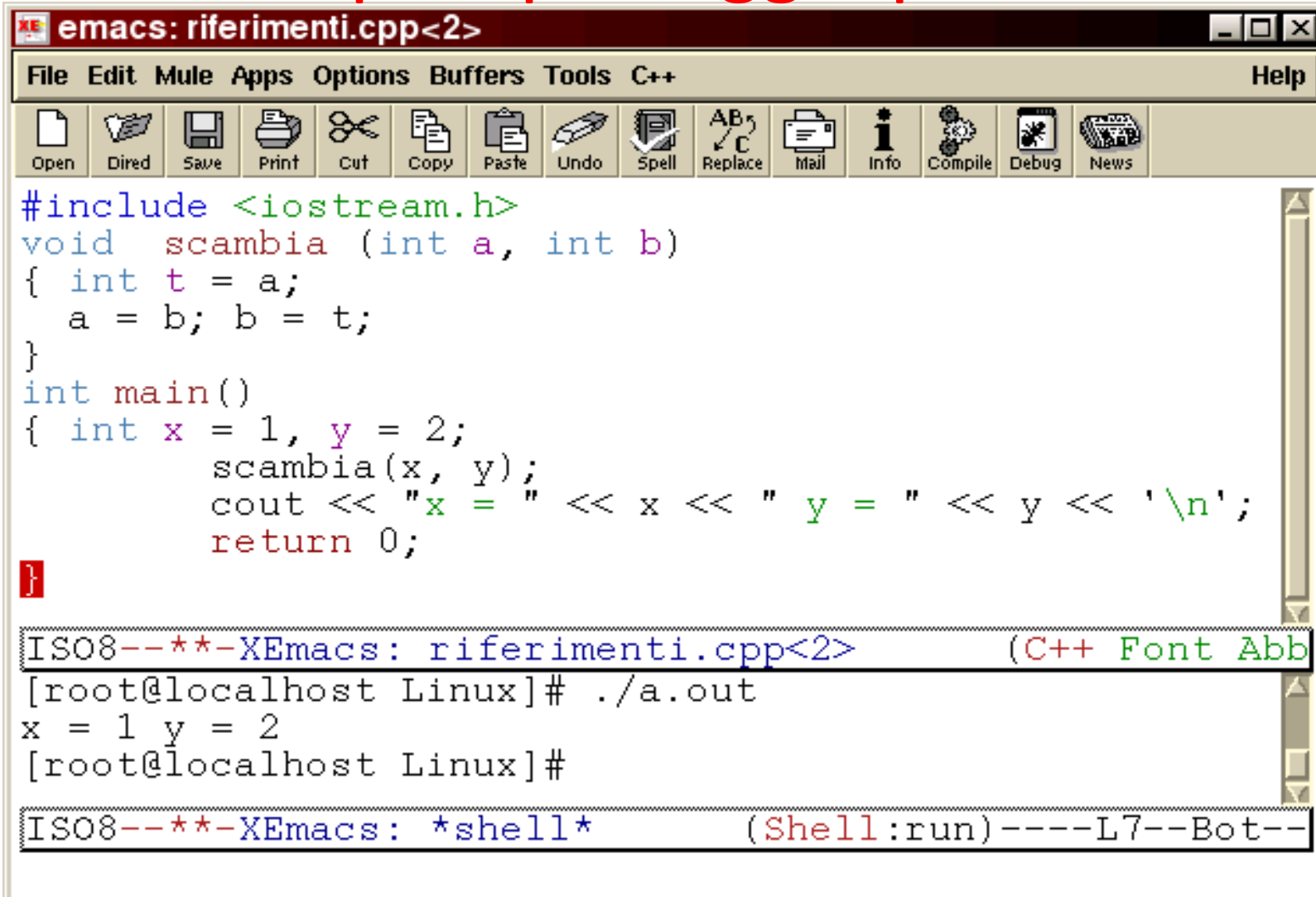


passaggio “per valore”

- 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 editor 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 code in the buffer is as follows:

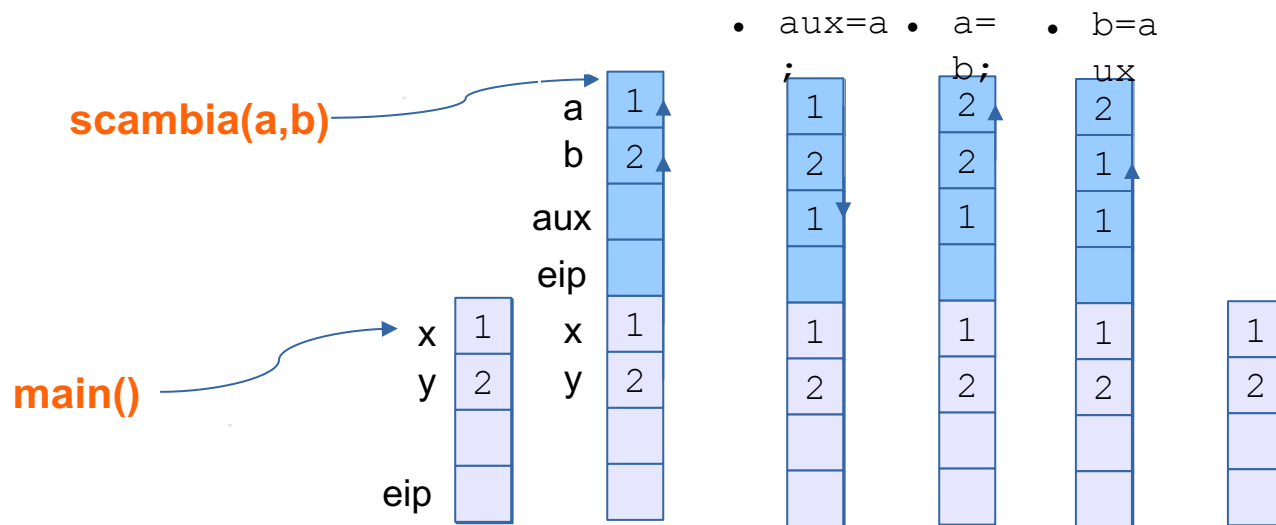
```
#include <iostream.h>
void scambia (int a, int b)
{ int t = a;
  a = b; b = t;
}
int main()
{ int x = 1, y = 2;
  scambia(x, y);
  cout << "x = " << x << " y = " << y << '\n';
  return 0;
}
```

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]#
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)





funzioni ricorsive

- è “ricorsiva” una funzione che richiama sé stessa

```
int fattoriale (int n)
{ if (n == 0) return 1;          \\ tappo
  return n*fattoriale(n - 1);  \\ passo
}
```

```
int mcd (int alfa, int beta)
{ if (beta == 0) return alfa;    \\ tappo
  return mcd(beta, alfa % beta); \\ passo
}
```

- per ogni funzione ricorsiva ne esiste una iterativa
- normalmente la versione iterativa è più conveniente in termini di tempo di esecuzione ed occupazione di stack



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);}`

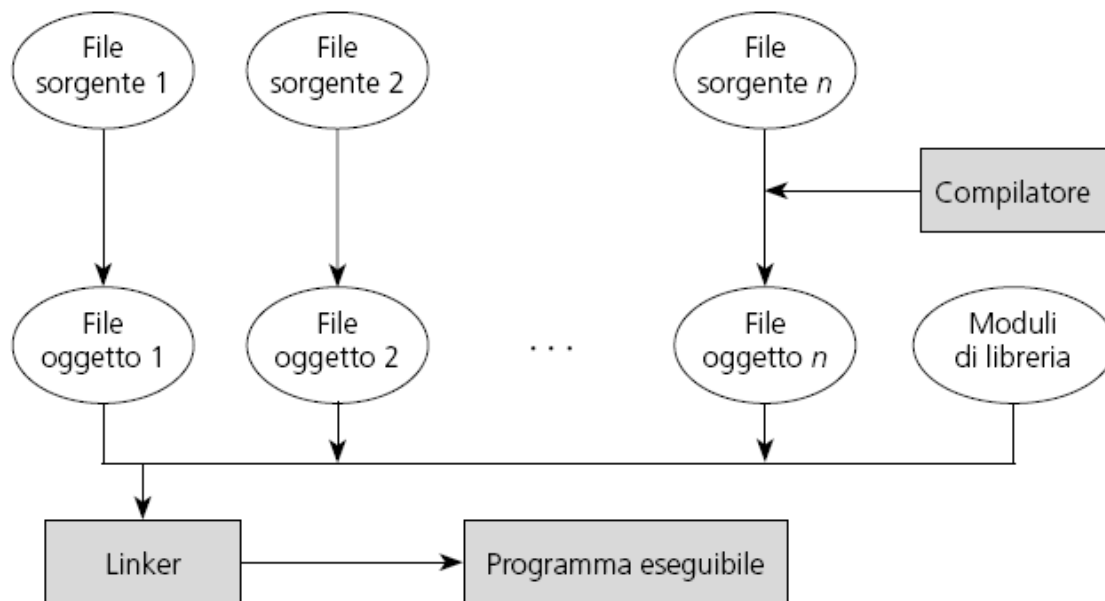


storage classes

- `extern`, `register`, `static` modificano la visibilità di una variabile o di una funzione
 - *variabili esterne*: una funzione può utilizzare una variabile globale definita in un altro file sorgente *dichiarandola* localmente con la parola riservata **`extern`**; in questo modo si indica al compilatore che la variabile è *definita* in un altro file sorgente che sarà *linkato* assieme
 - *variabili registro*: con la parola riservata **`register`** si chiede al compilatore di porre la variabile in uno dei registri del microprocessore; il compilatore può decidere di ignorare la richiesta; non possono essere variabili globali
 - *variabili statiche*: con la parola riservata **`static`** si chiede al compilatore di mantenere i valori delle variabili locali fra diverse chiamate di una funzione; quindi, al contrario delle normali variabili locali, una variabile statica s'inizializza una volta per tutte; purtroppo la keyword `static` ha anche altri significati, in particolare quello di rendere una funzione visibile solo nel file in cui è definita

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



esempio extern

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();
• }
```

```
aldo@Zagor:$ g++ file1.cpp file2.cpp -o prova
```

```
aldo@Zagor:$ ./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

Documentazione
completa in:

<https://devdocs.io/cpp/>



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



- *overheading*
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*
 - se una funzione sovraccaricata possiede un numero variabile di parametri (tramite l'uso di punti sospensivi [...]), può venire selezionata in mancanza di corrispondenze più specifiche



template di funzioni

- meccanismo per creare *funzioni* generiche, che possano cioè supportare simultaneamente differenti tipi di dato
- molto utili per i programmatori quando bisogna utilizzare la stessa funzione con differenti tipi di argomenti
- formato:

```
• template <class tipo> tipo funzione (tipo arg1, tipo arg2,...)  
• {  
•     // Corpo della funzione  
• }
```

- una dichiarazione tipica è:

```
• template <class T> T max (T a, T b)  
• {  
•     return a > b ? a : b;  
• }
```