Linguaggio C: le Funzioni

Sottoprogrammi

Spesso può essere utile avere la possibilità di costruire nuove istruzioni, o nuovi operatori che risolvano parti specifiche di un problema:

Un **sottoprogramma** permette di dare un nome a una parte di programma, rendendola *parametrica*.

Esempio: algoritmo *naïve sort*

```
#include <stdio.h>
#define dim 10
main()
{   int V[dim], i,j, max, tmp;

    /* lettura dei dati */
    for (i=0; i<dim; i++)
    {     printf("valore n. %d: ",i);
          scanf("%d", &V[i]);
    }
    /*ordinamento */
    for(i=dim-1; i>1; i--)
    {     max=i;
          for( j=0; j<i; j++)
          if (V[j]>V[max])
                    max=j;
          if (max!=i) /*scambio */
          {         tmp=V[i];
                    V[i]=V[max];
                    V[max]=tmp;
          }
    }
    /* stampa */
    for (i=0; i<dim; i++)
          printf("\n%d", V[i]);
}
```

Limiti di questa soluzione:

- difficile leggibilita`
- funziona solo con vettori di 10 elementi
- non e` riutilizzabile

Soluzione:

si puo` assegnare un nome ad ogni parte del programma, racchiudendone le istruzioni che la definiscono all'interno di un *componente software riutilizzabile*:

il *sottoprogramma*.

Esempio: algoritmo *naïve sort*

```
#include <stdio.h>
#define dim 10
...
main()
{ int V[dim];

  /* lettura dei dati */
  leggi(V, dim);

  /*ordinamento */
  ordina(V, dim);

  /* stampa */
  stampa(V,dim);
}
```

- **`leggi`**, **`ordina`** e **`stampa`** sono nomi di sottoprogrammi, ognuno dei quali rappresenta una parte del programma (nella prima versione).
- **`leggi(V, dim)`**: V e dim sono parametri, e rappresentano i dati dell'algoritmo che il sottoprogramma rappresenta

Vantaggi di questa soluzione:

- leggibilita`
- sintesi
- possibilita` di riutilizzo delle diverse parti

Riutilizzabilita`

Mediante i sottoprogrammi e` possibile eseguire piu` volte lo stesso insieme di operazioni senza doverlo riscrivere.

Ad esempio: ordinamento di due vettori.

```
#include <stdio.h>
#define dim 10
#define dim2 25
..
main()
{  int V1[dim], V2[dim2];
   leggi(V1, dim);
   leggi(V2, dim2);
   ordina(V1, dim);
   ordina(V2, dim2);
   stampa(V1,dim);
   stampa(V2, dim2);
}
```

Sottoprogrammi: funzioni e procedure

Un sottoprogramma e` una nuova istruzione, o un nuovo operatore definito dal programmatore per sintetizzare una sequenza di istruzioni.

In particolare:

- ***procedura***: e` un sottoprogramma che rappresenta un'istruzione non primitiva
- ***funzione***: e` un sottoprogramma che rappresenta un operatore non primitivo.

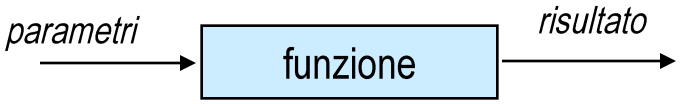
Tutti i linguaggi di alto livello offrono la possibilita` di definire funzioni e/o procedure.

- Il linguaggio *C* realizza solo il concetto di **funzione**.

Funzioni come componenti software

Una funzione è un "***componente software***" che cattura l'idea matematica di *funzione:*

- molti possibili **ingressi** (che non vengono modificati!)
- una sola uscita (il **risultato**)



- Una funzione:
  - riceve dati di ingresso attraverso i ***parametri***
  - esegue una **espressione**, la cui valutazione fornisce un **risultato**
  - denota un **valore** in corrispondenza al suo *nome*

Funzioni come componenti software

Esempio: Data una funzione $f : R \rightarrow R$

$$f(x) = 3 * x^2 + x - 3$$

→ se x vale 1 allora *f(x)* denota il valore 1.

Funzioni come componenti software

Il corrispettivo in *C*:

```
float f(float x) {
  return 3 * x * x + x - 3;
}
```

Si chiama "chiamata" o "invocazione" il meccanismo che permette di ottenere il valore denotato dalla funzione:

```
float y = f(1)
```

Dopo la chiamata a funzione, y contiene il valore "1"

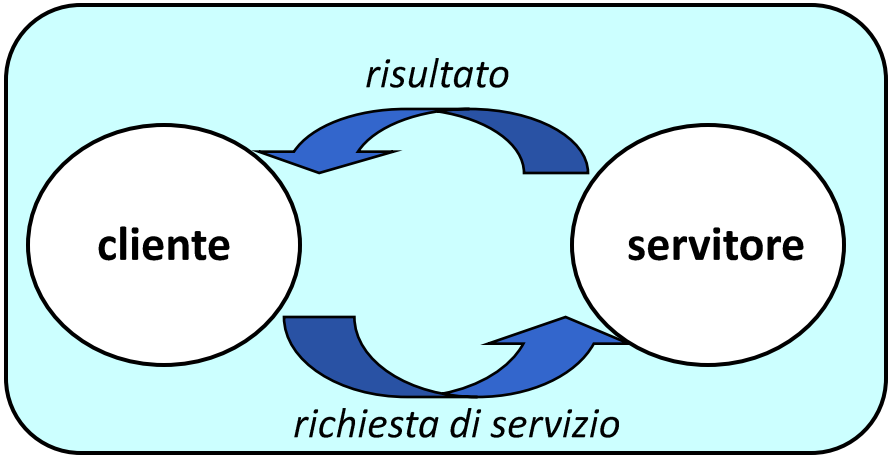
Funzioni

Il meccanismo di uso di funzioni nei linguaggi di programmazione fa riferimento allo schema di interazione tra componenti software

cliente/servitore

(*client – server*)

Modello Cliente-Servitore



Servitore:

- un qualunque ente capace di **nascondere la propria organizzazione interna**
- **presentando ai clienti una precisa *interfaccia*** per lo scambio di informazioni.

Cliente:

- qualunque ente in grado di **invocare uno o più servitori** per ottenere servizi.

Modello Cliente-Servitore

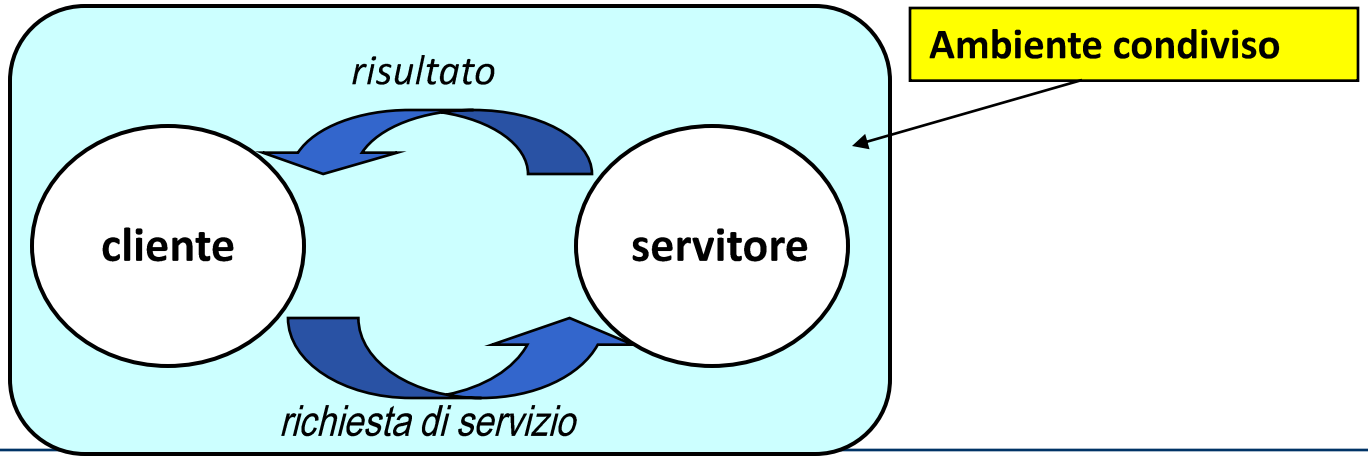
In generale, un servitore può

- essere **passivo** o **attivo**
- servire molti clienti, oppure costituire la risorsa privata di uno specifico cliente
  - in particolare: può servire un cliente alla volta, in sequenza, oppure più clienti per volta, in parallelo
- trasformarsi a sua volta in cliente, invocando altri servitori o anche se stesso.

Comunicazione Cliente/Servitore

Lo scambio di informazioni tra un cliente e un servitore può avvenire

- in modo **esplicito** tramite le interfacce stabilite dal servitore
- in modo **implicito** tramite dati accessibili ad entrambi (*l'ambiente condiviso*).



Funzione come servitore

Una funzione è un **servitore:**

- **passivo**
- che realizza un **particolare servizio**
- che **serve un cliente per volta**
- che **può trasformarsi in cliente** invocando altre funzioni (o eventualmente se stessa)

- Il cliente ***chiede*** al servitore di svolgere il servizio
  - ***chiamando*** tale servitore (per ***nome***)
  - fornendogli i dati necessari (***parametri***)

- Nel caso di una funzione, cliente e servitore comunicano mediante ***l'interfaccia*** della funzione.

Interfaccia di una funzione

L'interfaccia (o *intestazione*, *firma*, *signature*, *prototipo*) di una funzione comprende

- ***nome*** della funzione
- lista dei ***parametri***
- ***tipo del valore*** calcolato dalla funzione

- enuncia le regole di comunicazione tra cliente servitore.

Cliente e servitore comunicano quindi mediante:

- i **parametri** trasmessi dal cliente al servitore all'atto della chiamata (direzione: **dal cliente al servitore**)
- il **valore** restituito dal servitore al cliente (direzione: dal servitore al cliente)

A scanso di equivoci: la comunicazione NON avviene tramite ambiente condiviso (almeno a livello astratto)

Interfaccia: esempio

```
int max (int x, int y) /* interfaccia */
{
    if (x>y) return x;
          else return y;
}
```

- Il simbolo **max** denota il nome della funzione
- Le variabili intere **x** e **y** sono i parametri della funzione
- Il valore restituito è un intero **int**
- Ambedue i parametri sono di tipo **int**

Comunicazione cliente → servitore

La comunicazione cliente → servitore avviene mediante i ***parametri***.

- **Parametri formali :**
  - sono specificati nell'interfaccia del servitore
  - indicano cosa il servitore si aspetta dal cliente
- **Parametri attuali :**
  - sono trasmessi dal cliente all'atto della chiamata
  - devono corrispondere ai parametri formali in **numero**, **posizione** e **tipo**.

Esempio

Parametri Formali

```
int max (int x, int y)
{   if (x>y) return x;
          else return y;
}
```

SERVITORE:

definizione della funzione

```
main(){
            int z = 8;
            int m;
            m = max(z , 4);
...}
```

Parametri Attuali

CLIENTE:

chiamata della funzione

Comunicazione cliente/servitore

L'associazione (***legame***) tra i parametri attuali e i parametri formali viene fatta *al momento della chiamata,* in modo **dinamico**.

Tale legame:

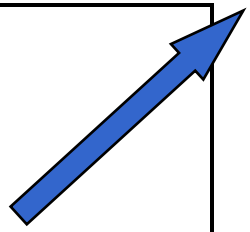
- vale solo per l'invocazione corrente
- vale solo per la durata dell'esecuzione della funzione.

ESEMPIO

Parametri Formali

```
int max (int x, int y){
    if (x>y) return x;
        else return y;
}
```

```
main(){
        int z = 8;
        int m;
        m1 = max(z,4);
        m2 = max(5,z);
    }
```



All'atto di questa chiamata della funzione si effettua un legame tra:

x e z

y e 4

ESEMPIO

Parametri Formali

```
int max (int x, int y){
    if (x>y) return x;
        else return y;
}
```

```
main(){
        int z = 8;
        int m;
        m1 = max(z,4);
        m2 = max(5,z);
    }
```

All'atto di questa chiamata della funzione si effettua un legame tra

x e 5

y e z

Comunicazione cliente/servitore

L'associazione (***legame***) tra i parametri attuali e i parametri formali viene fatta *al momento della chiamata,* in modo **dinamico**.

Tale legame:

- vale solo per l'invocazione corrente
- vale solo per la durata dell'esecuzione della funzione.
- **Per la precisione: i parametri formali assumono il <u>valore</u> dei parametri attuali**

Programmi C con funzioni

Ogni funzione viene costituisce una unità di programma, introdotta mediante una apposita ***definizione***.

In generale:

`programma C = una collezione di` *`unita` di programma`*

- formalmente anche il main e` una funzione (sempre presente); essa viene invocata per prima, quando il programma viene messo in esecuzione

Regola generale sulla visibilita` degli identificatori:

" Prima di utilizzare un identificatore e` necessario che sia gia` stato definito (oppure dichiarato)."

→ all'interno del file sorgente vengono specificate **prima le definizioni** delle ***funzioni***, ed infine viene esplicitato il ***main***.

Esempio

```
<definizione funzione 1>
<definizione funzione 2>
...
main()
{
...
<chiamata di funzione 1>
<chiamata di funzione 2>
..
}
```

Definizione di funzione in C

```
<definizione-di-funzione> ::=
<tipoValore>     <nome>(<parametri-formali>)
{
   <corpo>;
}
```

dove:

<parametri-formali>

- una lista (eventualmente vuota) di variabili, visibili dentro il corpo della funzione.

<tipoValore>

- deve coincidere con il tipo del valore risultato della funzione: puo` essere di tipo semplice (int, char, float o double), di tipo struct, oppure di tipo puntatore.

Definizione di funzione in C

```
<definizione-di-funzione> ::=
<tipoValore>   <nome>(<parametri-formali>)
{
   <corpo>;

}
```

- Nella parte <***corpo***> possono essere presenti definizioni e/o dichiarazioni locali (***parte dichiarazioni***) e un insieme di istruzioni (***parte istruzioni***).
- I dati riferiti nel corpo possono essere **costanti**, **variabili**, oppure **parametri formali**.
- All'interno del corpo, i **parametri formali** vengono trattati come ***variabili***.

Definizione di funzione in C -esempio

```
int somma(int x, int y, int z) {
      int s;       /* è possibile definire nuove
                      variabili all'inizio del corpo
                      di una funzione */
      s = x + y;
      x = 5;       /* istruzione inutile (ma serve a
                      far vedere che i parametri
                      formali sono trattati a tutti
                      gli effetti come variabili */
      return s + z;
}
```

Definizione di funzione in C -esempio 2

Main è una funzione:

```
int main(int argc, char** argv) {
      <definizioni & dichiarazione di variabili>

      <istruzioni>

      return 0; /* "0" è il valore restituito! Nella
                   fattispecie rappresenta un codice
                   di errore ("0" = "tutto ok")
}
```

Meccanismo di chiamata di funzioni

- All'atto della ***chiamata***, l'esecuzione del cliente viene ***sospesa*** e il controllo passa al servitore.
- Il servitore "vive" solo per il tempo necessario a svolgere il servizio.
- Al termine, il servitore "muore", e *l'esecuzione torna al cliente.*

Chiamata di Funzione

La chiamata di funzione è un'espressione della forma:

```
<nomefunzione> ( <parametri-attuali> )
```

dove:

```
<parametri-attuali> ::=
   [ <espressione> ]  { , <espressione> }
```

ESEMPIO

Parametri Formali

```
int max (int x, int y){
    if (x>y) return x;
        else return y;
}
```

SERVITORE

definizione della funzione

```
main(){
        int z = 8;
        int m;
        m = max(z , 4);
    }
```

Parametri Attuali

CLIENTE

chiamata della funzione

Risultato di una funzione: `return`

L'istruzione

```
return <espressione>
```

provoca la terminazione dell'attivazione della funzione (*il servitore muore*) e la restituzione del controllo al cliente, unitamente al valore dell'espressione che la segue.

- Eventuali istruzioni successive alla return *non saranno mai eseguite!*

```
int max (int x, int y ){
      if (x>y) return x;
      else return y;
      printf("ciao!"); /* mai eseguita !*/
  }
```

ESEMPIO

Parametri Formali

```
int max (int x, int y){
    if (x>y) return x;
        else return y;
}
```

SERVITORE

definizione della funzione

```
main(){
        int z = 8;
        int m;
        m = max(z , 4);
    }
```

CLIENTE

chiamata della funzione

Risultato

Esempio completo

```
#include <stdio.h>
int max (int x, int y )
{     if (x>y) return x;
            else return y;
}
main()
{
      int z = 8;
      int m;
      m = max(z , 4);
      printf("Risultato: %d\n", m);
}
```

Qualche definizione: BINDING & ENVIRONMENT

- La conoscenza di cosa un simbolo denota viene espressa da una *legame* (*binding*) tra il simbolo e un valore.
- L'insieme dei *binding* validi in (un certo punto di) un programma si chiama *environment (ambiente)*.

ESEMPIO

```
main(){
            int z = 8;
            int y, m;
            y = 5
            m = max(z, y); /* X */
      }
```

Consideriamo il punto X: In questo environment il simbolo z è legato al valore 8 tramite l'inizializzazione, mentre il simbolo y è legato al valore 5. Pertanto i valori dei parametri di cui la funzione max ha bisogno per calcolare il risultato sono noti all'atto dell'invocazione della funzione

ESEMPIO

```
main(){
            int z = 8;
            int y, m;
            m = max(z,y); /* X */
      }
```

Punto X: In questo environment il simbolo z è legato al valore 8 tramite l'inizializzazione, mentre il simbolo y non è legato ad alcun valore. Pertanto i valori dei parametri di cui la funzione max ha bisogno per calcolare il risultato NON sono noti all'atto dell'invocazione della funzione e la funzione non può essere valutata correttamente

ESEMPIO

- Il servitore...

```
int max (int x, int y ){
    if (x>y) return x;
    else return y;
}
```

- … e un possibile cliente:

```
main(){
    int z = 8;
    int m;
    m = max(2*z,13);
}
```

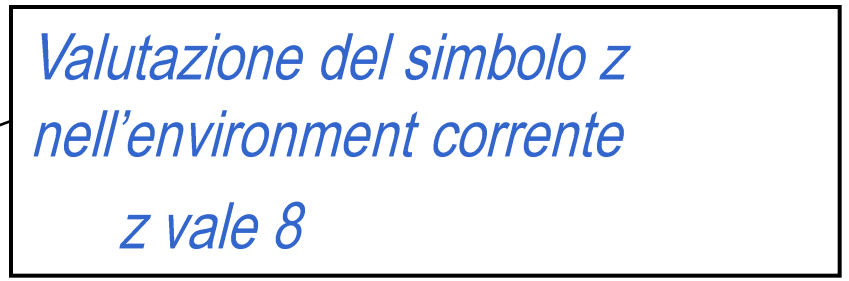
ESEMPIO

- Il servitore...

```
int max (int x, int y ){
    if (x>y) return x;
    else return y;
}
```

- … e un possibile cliente:

```
main(){
    int z = 8;
    int m;
    m = max(2*z,13);
}
```



ESEMPIO

- Il servitore...

```
int max (int x, int y ){
    if (x>y) return x;
    else return y;
}
```

- … e un possibile cliente:

```
main(){
    int z = 8;
    int m;
    m = max(2*z,13);
}
```

*Calcolo dell'espressione 2*z nell'environment corrente*

*2*z vale 16*

ESEMPIO

- Il servitore...

```
int max (int x, int y ){
    if (x>y) return x;
    else return y;
}
```

- … e un possibile cliente:

```
main(){
    int z = 8;
    int m;
    m = max(2*z,13);
}
```

Invocazione della chiamata a ***max*** *con parametri attuali 16 e 13*

IL CONTROLLO PASSA AL SERVITORE

ESEMPIO

- Il servitore...

```
int max (int x, int y ){
     if (x>y) return x;
     else return y;
}
```

- … e un possibile cliente:

```
main(){
     int z = 8;
     int m;
     m = max(2*z,13);
}
```

Viene effettuato il legame dei parametri formali x e y con quelli attuali 16 e 13.

INIZIA L'ESECUZIONE DEL SERVITORE

ESEMPIO

- Il servitore...

```
int max (int x, int y ){
     if (x>y) return x;
     else return y;
}
```

- … e un possibile cliente:

```
main(){
     int z = 8;
     int m;
     m = max(2*z,13);
}
```

Viene valutata l'istruzione if (16 > 13) che nell'environment corrente e' vera. Pertanto si sceglie la strada

```
return x
```

ESEMPIO

- Il servitore...

```
int max (int x, int y ){
     if (x>y) return x;
     else return y;
}
```

- … e un possibile cliente:

```
main(){
     int z = 8;
     int m;
     m = max(2*z,13);
}
```

Il valore 16 viene restituito al cliente.

IL SERVITORE TERMINA E IL CONTROLLO PASSA AL CLIENTE.

NOTA: i binding di x e y vengono distrutti

ESEMPIO

- Il servitore...

```
int max (int x, int y ){
    if (x>y) return x;
    else return y;
}
```

- … e un possibile cliente:

```
main(){
    int z = 8;
    int m;
    m = max(2*z,13);
}
```

Il valore restituito (16) viene assegnato alla variabile m nell'environment del cliente.

RIASSUMENDO...

All'atto dell'invocazione di una funzione:

- si crea una *nuova attivazione (istanza) del servitore*
- si alloca la memoria per i parametri (e le eventuali variabili locali)
- si trasferiscono i parametri al servitore (il valore che hanno nell'environment locale)
- si trasferisce il controllo al servitore
- si esegue il codice della funzione.

Dichiarazione di funzione

Regola Generale:

Prima di utilizzare una funzione e` necessario che sia gia` stata ***definita*** oppure ***dichiarata***.

Funzioni C:

- ***definizione***: descrive le proprieta` della funzione (tipo, nome, lista parametri formali) e la sua realizzazione (lista delle istruzioni contenute nel blocco).
- ***dichiarazione*** (prototipo): descrive le proprieta` della funzione senza esplicitarne la realizzazione (blocco)
  - serve per "anticipare" le caratteristiche di una funzione definita successivamente.

Dichiarazione di una funzione:

La dichiarazione di una funzione si esprime mediante l'intestazione della funzione, seguita da ";":

```
<tipo-ris> <nome> ([<lista-par-formali>]);
```

Ad esempio: Dichiarazione della funzione "max":

```
void max(int x, int y);
```

Dichiarazione di Funzioni

Una funzione puo` essere dichiarata piu` volte (in punti diversi del programma), ma e` definita **una sola volta**.

E` possibile inserire i prototipi delle funzioni:

- nella parte dichiarazioni globali di un programma,
- nella parte dichiarazioni del main,
- nella parte dichiarazioni delle funzioni.

Dichiarazione di funzioni

Ad esempio:

```
#include <stdio.h>
long power (int base, int n); /* dichiarazione */

main()
{  long power (int base, int n); /* dichiarazione */
   int X, exp;
   scanf("%d%d", &X, &exp);
   printf("%ld", power(X,exp));
}

long power(int B, int N)
{ int i, RIS;
for (RIS=B, i=1; i<N; i++) RIS*=B;
return RIS;
}
```

Struttura dei Programmi C

Per migliorare la leggibilita`, spesso si strutturano i programmi in modo tale che la definizione del **`main`** compaia prima delle definizioni delle altre funzioni.

- **Convenzione:**

```
<lista dichiarazioni di funzioni>
<main>
<definizioni delle funzioni dichiarate>
```

(Dichiarazioni di) funzioni di libreria

- **printf**, **scanf**, ecc. sono funzioni
- il file **stdio.h** contiene le loro dichiarazioni
- Ad esempio, la direttiva:

  #include <stdio.h>

  provoca l'aggiunta nel file sorgente del contenuto del file specificato, cioe` delle dichiarazioni delle funzioni della libreria di I/O.

- Analogamente per le altre funzioni di libreria:
- le dich. di **malloc** e **free** sono contentue nel file **stlib.h**
- le dich. di **strlen**, **strcmp**, etc. sono contenute nel file **string.h**
- ecc.

Una nota in coda:

E' possibile passare un vettore come argomento di una funzione (maggiori dettagli nel prossimo blocco di slides).

Esempio: massimo di un vettore di numeri naturali:

```
int max(int[] v, int dim) {
      int m = -1, i;
      for(i = 0; i < dim; i++)
            if (v[i] > m) m = v[i];
      return m;
}
```