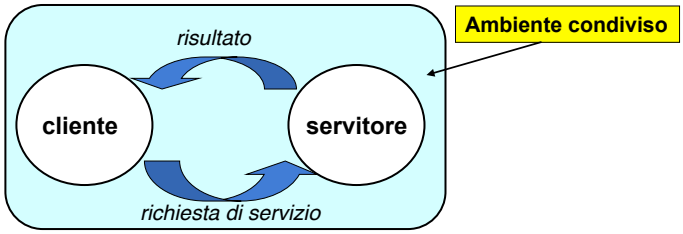
Il linguaggio C
Funzioni e procedure

modello cliente-servitore



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 leggibilità
- funziona solo con vettori di 10 elementi
- non è riutilizzabile

Soluzione:

si può 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:

- leggibilità
- sintesi
- riusabilità

Riusabilità

Mediante i sottoprogrammi è possibile eseguire più 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 è una nuova istruzione, o un nuovo operatore definito dal programmatore per sintetizzare una sequenza di istruzioni.

In particolare:

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

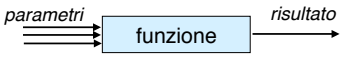
Tutti i linguaggi di alto livello offrono la possibilità di definire funzioni e/o procedure.

➔ 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 → R

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

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

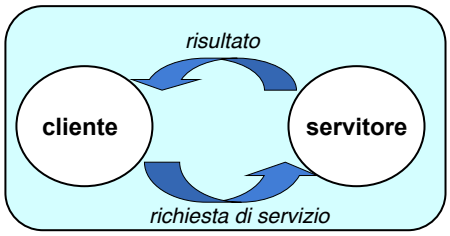
Funzioni

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.

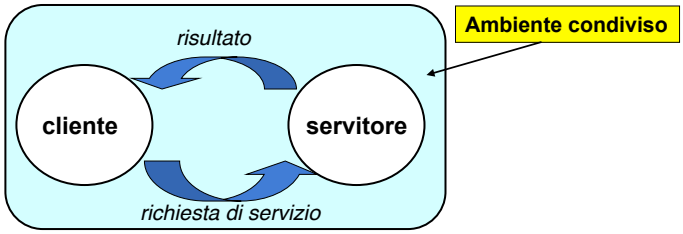
Cliente:

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

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 stesso)

- 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*) di una funzione comprende

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

➔ enuncia le regole di comunicazione tra cliente e 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)

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`** .

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 *effettivi*:**
  - 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);
        ...
}
```

CLIENTE: chiamata della funzione

Parametri Effettivi

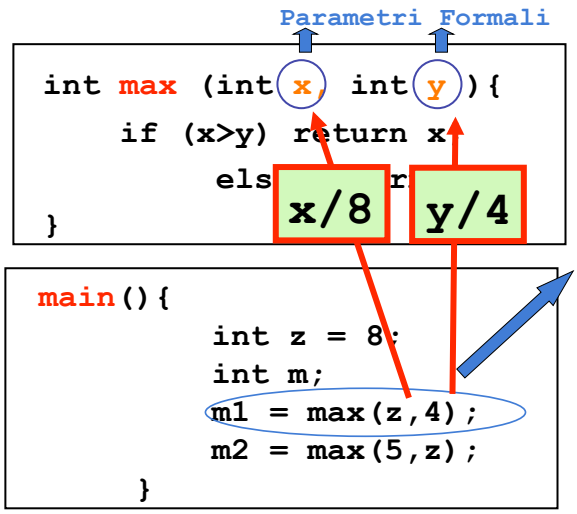
Comunicazione cliente/servitore

L'associazione (***legame***) tra i parametri effettivi 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 della funzione.

ESEMPIO

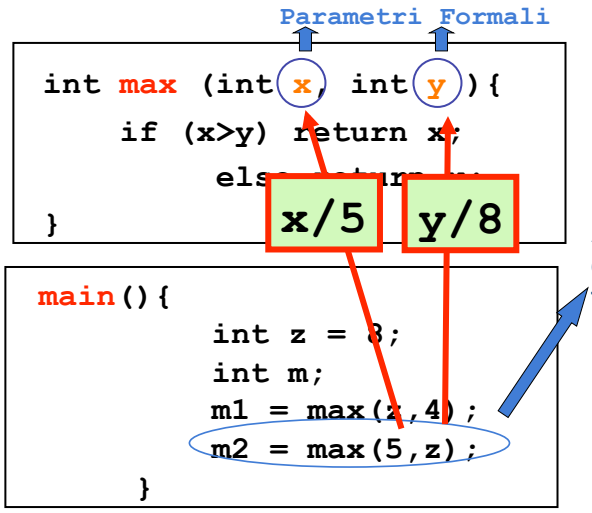


All'atto di questa chiamata della funzione si effettua un legame tra:

x e z

y e 4

ESEMPIO



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

x e 5

y e z

1 3

Programmi C con funzioni

Ogni funzione viene realizzata mediante la ***definizione*** di una *unità di programma* distinta dal programma principale (main).

➔ Generalizziamo la struttura di un programma C:
il programma è una collezione di *unità di programma* (tra le quali compare sempre l'unità *main*)

Richiamiamo la regola generale sulla visibilità degli identificatori:

"Prima di utilizzare un identificatore è necessario che sia già stato definito (oppure dichiarato)."

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

➔ formalmente anche il main è una funzione; essa viene invocata per prima, quando il programma viene messo in esecuzione

Esempio

```
<definizione funzione 1>
<definizione funzione 2>
...
main()
{
...
<chiamata di funzione 2>
<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: può essere
  - di tipo scalare (int, char, float o double),
  - di tipo struct, oppure
  - di tipo puntatore.

Definizione di funzione in C

```
int somma3(int x, int y)
{ int z = 1;
  x++; y++;
 return x+y+z;
}
```

- 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***.

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

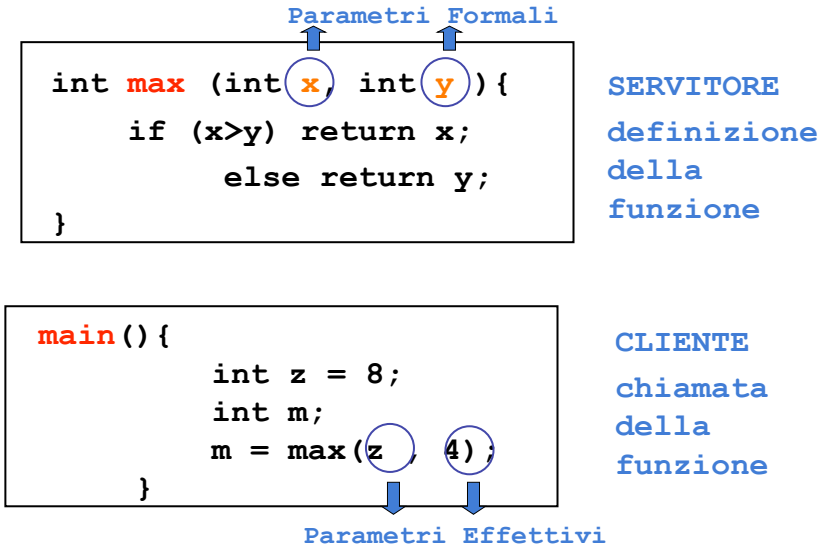
La chiamata di funzione è un'espressione nella forma:

```
somma3(45, 26);
somma3(24, a); /* a di tipo int */
```

dove:

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


ESEMPIO



Risultato di una funzione: `return`

L'istruzione

NOTA: return si usa senza parentesi…

```
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);
}
```


BINDING & ENVIRONMENT

`return x;` ⇒ devo sapere cosa denota il simbolo x

- 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 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 */
     }
```

Consideriamo il 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 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

3

Binding e visibilità

- Tutte le occorrenze di un nome nel testo di un programma a cui si applica un dato *binding* si dicono essere entro lo stesso ***scope*** (***visibilità***) del binding.
- Le regole in base a cui si stabilisce la *portata* di un binding si dicono ***regole di visibilità*** (o ***scope rules***).

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);
}
```

Valutazione del simbolo z nell'environment corrente

z vale 8

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);
}
```

8

*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);
}            16  13
```

Invocazione della chiamata a `max` con parametri effettivi 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);
}            16  13
```

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

INIZIA L'ESECUZIONE DEL SERVITORE

ESEMPIO

- **Il servitore...** 16 13

```
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 è vera. Pertanto si sceglie la strada

```
return x
```


ESEMPIO

- **Il servitore...** 16 13

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

16

- **… 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);
}        16
```

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
- si trasferisce il controllo al servitore
- si esegue il codice della funzione.

Procedure in C

Formalmente in C esiste solo il concetto di funzione:

e le *procedure* ?

È possibile costruire delle particolari funzioni che non restituiscono alcun valore:

```
void proc (int P){..}
```

è la definizione di una funzione (`proc`) che non restituisce alcun valore:

- **`void`** è un identificatore di tipo per classificare dati il cui dominio è l'insieme vuoto.

Procedure in C

La definizione di funzione:

```
void proc (int P){..}
```

realizza una procedura.

Osservazioni:

- la chiamata di **`proc`** non produce alcun risultato
- dall'interno del corpo della funzione **`proc`** non verrà restituito alcun risultato al cliente:
  - ➔il corpo potrà contenere o meno l'istruzione **`return`**, eventualmente utilizzata senza argomento: **`return;`**

Procedure in C

Esempio:

```
#include <stdio.h>
void stampafloat(float P) /* "procedura" */
{  printf("%f\n", P);
   return; /* termina l'attivazione*/
}

float quadrato(float X) /* funzione*/
{return X*X;}

main()
{  float V;
   scanf("%f", &V);
   V=quadrato(V);
   stampafloat(V); /* chiamata di "procedura"*/
}
```