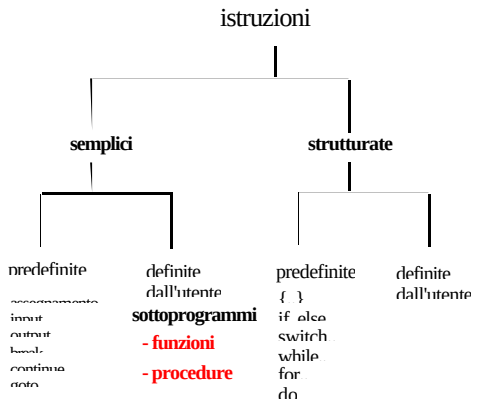
Sottoprogrammi: Funzioni e Procedure



I linguaggi di alto livello permettono di definire istruzioni non primitive per risolvere parti specifiche di un problema: i **sottoprogrammi** (funzioni e procedure).

Funzioni e Procedure

Ad esempio: Ordinamento di un insieme

```
#include <stdio.h>
#define dim 10

main()
{int V[dim], i,j, max, tmp, quanti;

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


- Potrebbe essere conveniente scrivere lo stesso algoritmo in modo più **astratto**:

```
#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()`, `stampa()` sono ***sottoprogrammi*** : il main "chiama" leggi, ordina e stampa.

Vantaggi:

- sintesi
- leggibilità
- possibilità di riutilizzo del codice

Sottoprogrammi: *funzioni* e *procedure*

- Rappresentano nuove istruzioni che agiscono sui dati utilizzati dal programma, "nascondendo" la sequenza delle operazioni effettivamente eseguite dalla macchina.
- Vengono realizzate mediante la definizione di unità di programma (***sottoprogrammi***) distinte dal programma principale (*main*).

↣ **D'ora in poi:** il programma è una **collezione di unità di programma** (tra le quali compare l'unità *main*)

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

Ciò è reso possibile da:

- costrutti per la **definizione** di sottoprogrammi
- meccanismi per l'**utilizzo** di sottoprogrammi (meccanismi di ***chiamata***)

Funzioni e Procedure

Definizione:

Nella fase di **definizione** di un sottoprogramma (funzione o procedura) si stabilisce:

- un **identificatore** del sottoprogramma
- si esplicita il **corpo** del sottoprogramma (cioè, l'insieme di istruzioni che verrà eseguito ogni volta che il sotto-programma verrà *chiamato*);
- si stabiliscono le **modalità di comunicazione** tra l'unità di programma che usa il sottoprogramma ed il sottoprogramma stesso (definizione dei **parametri formali**).

Utilizzo di funzioni/procedure (*chiamata*):

- Per chiamare un sottoprogramma (cioè, per richiedere l'esecuzione del suo corpo), si utilizza l'identificatore assegnato al sottoprogramma in fase di definizione (***chiamata*** o invocazione del sottoprogramma).

Meccanismo di Chiamata

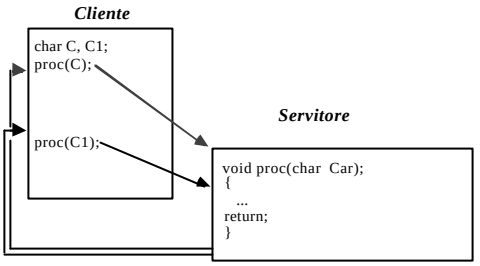
Quando si verifica una chiamata a sottoprogramma, si possono individuare due entità:

- l'unità di programma **chiamante**;
- l'unità di programma **chiamata** (il sotto-programma).

Quando avviene la chiamata, l'esecuzione dell'unita` di programma "chiamante" (quella, cioè, che contiene l'invocazione) viene **sospesa**, ed il controllo passa al sottoprogramma chiamato (che eseguirà le istruzioni contenute nel corpo).

L'unità chiamante funge da **cliente** dell'unita` chiamata (che svolge il ruolo di **servitore**).

Modello Cliente-Servitore



Parametri

I ***parametri*** costituiscono il mezzo di comunicazione tra unità chiamante ed unità chiamata.
Supportano lo scambio di informazioni tra chiamante e sottoprogramma.

parametri formali: sono quelli specificati nella definizione del sotto-programma. Sono in numero prefissato e ad ognuno di essi viene associato un tipo. Le istruzioni del corpo del sottoprogramma utilizzano i parametri formali.

parametri attuali: sono i valori effettivamente forniti dall'unità chiamante al sottoprogramma all'atto della chiamata.

Parametri

- Parametri ***attuali*** (specificati nella chiamata) e ***formali*** (specificati nella definizione) devono corrispondersi in ***numero***, ***posizione*** e ***tipo***.

- All'atto della chiamata avviene il ***legame dei parametri***, cioè ai parametri formali vengono associati i parametri attuali.

Come avviene l'associazione tra parametri attuali e parametri formali ?

Esistono, in generale, varie forme di legame. Ad esempio:

- legame per **valore**;
- legame per **indirizzo**;

Il significato delle due tecniche di legame dei parametri verrà spiegato più avanti.

Funzioni e Procedure

Vantaggi:

- **riutilizzo di codice**: sintetizzando in un sottoprogramma un sotto-algoritmo, si ha la possibilità di invocarlo più volte, sia nell'ambito dello stesso programma, che nell'ambito di programmi diversi (evitando di dover replicare ogni volta lo stesso codice).
- migliore **leggibilità**: si ha in fatti una maggiore capacità di astrazione
- sviluppo **top-down**: si delega a funzioni/procedure da sviluppare in una fase successiva la soluzione di sottoproblemi.
- testo del programma più **breve**: minore probabilità di errori, dimensione del codice eseguibile più piccola.

Procedure e Funzioni

In generale, i sottoprogrammi si suddividono in **procedure** e **funzioni**:

Procedura:

E` un'astrazione della nozione di ***istruzione***. E` un'istruzione non primitiva attivabile in un qualunque punto del programma in cui può comparire un'istruzione.

Funzione:

E' un'astrazione del concetto di ***operatore***. Si può attivare durante la valutazione di una qualunque espressione e **restituisce un valore.**

Ad esempio:

```
main()
{ int Ris, N=7;
  stampa(N);/*procedura*/
  Ris=fattoriale(N)-10;/*funzione*/
};
```

- Formalmente, in C i sottoprogrammi sono soltanto ***funzioni***; le procedure possono essere realizzate come funzioni che non restituiscono alcun valore (**void**).

Funzioni in C

Procedure e funzioni si definiscono seguendo regole sintattiche simili.

Definizione di funzione:

<def-funzione> ::= <intestazione>
{ <parte-dichiarazioni> <parte-istruzioni> }

Quindi, per definire una funzione, è necessario specificare una ***intestazione*** e un ***blocco*** {..}:

Struttura dell'intestazione:

<intestazione>::=<tipo-ris> <nome> ([<lista-par-formali>])

dove:

- **<tipo-ris>**: è un indentificatore che indica il tipo di risultato restituito(***codominio***). Il tipo restituito può essere predefinito o definito dall'utente. Una funzione non può restituire valori di tipo:
  - **vettore**
  - **funzione**
- **<nome>:** è l'identificatore della funzione
- **<lista-par-formali>** è la lista dei parametri formali (*dominio*). Per ciascun parametro formale viene specificato il tipo e un identificatore che è un nome simbolico per rappresentare il parametro all'interno della funzione (nel *blocco*). I parametri sono separati mediante virgola.

Definizione di Funzioni in C

Blocco :

- Il blocco contiene il **corpo** della funzione e, come al solito, è strutturato in una <parte dichiarazioni> e una <parte istruzioni>:
  - la <parte dichiarazioni> contiene le dichiarazioni e definizioni *locali* alla funzione;
  - la <parte istruzioni> contiene la sequenza di istruzioni associata al corpo (rappresenta l'algoritmo eseguito dalla funzione)
- I dati riferiti nel blocco possono essere **costanti**, **variabili**, oppure **parametri formali**: all'interno del blocco, i parametri formali vengono trattati come variabili.

Istruzione return:

Per restituire il risultato, la funzione utilizza (all'interno della parte istruzioni) l'istruzione **return**:

return [<espressione>]

Effetto:
restituisce il controllo al chiamante e assegna all'identificatore della funzione il valore dell'<espressione>.

Esempio:

```
typedef enum{false,true} Boolean;
Boolean maggioredi100(int a) /*intest. */
{ /*parte dichiarazioni: */
  Boolean result;

  /* parte istruzioni: */
  if (a>100) result=true;
  else result=false;
  return result;
}
```

Esempio:

```
#define N 100
typedef char  vettore[N];

int minimo (vettore vet)
{
int i, min;/* def. locali a minimo */
  min=vet[0];
  for (i=1; i<N; i++)
    if (vet[i]<min) min=vet[i];
  return min;
}
```

↣ i e min sono ***variabili locali***:

- **tempo di vita**: esistono solo durante l'esecuzione della funzione minimo
- **visibilità**: sono visibili (cioè utilizzabili) soltanto all'interno della funzione minimo.

Esempio:

```
int   read_int() /* intest. */
{
  int a;
  scanf("%d", &a);
  return a;
}
```

Possono esserci ***più istruzioni return*** *(non desiderabile)*:

```
int max (int a, int b) /*intest.*/
{
  if (a>b) return a;
  else   return b;
}
```

o ***nessuna***:

```
int   print_int(int a)/* intestazione */
{
  printf("%d", a);
}
```

↣ In questo caso, il sottoprogramma termina in corrispondenza del simbolo **}** ed il valore restituito e` ***indefinito.***

Esempio:

```
/* funzione elevamento a potenza */

long power(int base, int n)
{
  int i;
  long p=1;

  for (i=1;i<=n;++i)
        p *= base; /* p = p*base */
  return p; /* restituisce il risultato */
}
```


Funzioni in C

Chiamata di funzioni:

In generale, la chiamata di una funzione compare all'interno di una espressione secondo la sintassi:

…nomefunzione(<lista parametri attuali>)…

Ad esempio:

```
main()
{
int z, x=2;
...
z=power(x,2)+power(x,3);
x=max(power(z,2), 30);
printf("%d\n", x);
}
```

Realizzazione delle Procedure in C

Una funzione può anche avere nessun valore (void) come risultato:

void	insieme vuoto di valori (dominio vuoto)
void fun(...)	funzione che non restituisce alcun valore

↣ **In questo modo si realizza in C il concetto di** procedura

Esempio:

```
void print_int(int a)
{
  printf("%d", a);
}
```

↣ Poiché una procedura non restituisce alcun valore, non è necessario prevedere l'istruzione di **return** all'interno del corpo; se si utilizza, **non si deve specificare alcun argomento**:

```
return;
```

Uso:

La procedura è l'astrazione del concetto di istruzione:

```
main()
{ int X;
  scanf("%d", &X);
  print_int(X):
}
```


Esempio:

```
#include <stdio.h>

int max (int a, int b) /*def. max*/
{
  int result;

  if (a>b) result=a;
  else result=b;
  return result;
}

void print_int (int a) /* def. */
{
  printf("%d\", a);
  return;
}

void dummy()  /*def. dummy */
{
printf("Ciao!\n");
}

main()
{ int A, B;
  printf("Dammi A e B: ");
  scanf("%d %d", &A, &B);
  print_int(max(A,B));
  dummy();
}
```


Struttura dei Programmi C

Quale struttura devono avere i programmi che fanno uso di funzioni ?

E` necessario aggregare la definizione del main alle definizioni delle funzioni utilizzate, ad esempio secondo lo schema seguente:

<lista delle definizioni di funzioni>
<main>

- all'interno del file sorgente vengono prima elencate le definizioni delle funzioni necessarie, ed infine viene esplicitato il main.

Ad esempio:

```
#include <stdio.h>

int max (int a, int b)
{ int result;
  if (a>b) result=a;
  else result=b;
  return result;
}

main()
{ int A, B;
  printf("Dammi A e B: ");
  scanf("%d %d", &A, &B);
  printf("%d\n", max(A,B));
}
```

➳ Se all'interno di un blocco viene utilizzata una funzione f, la definizione di f deve comparire prima del blocco che la utilizza.

Esempio:

```
#include <stdio.h>

int max (int a, int b)
{ int result;
  if (a>b) result=a;
  else result=b;
  return result;
}

int sommamax(int a1, a2, a3, a4)
{  return max(a1,a2)+max(a3,a4);}

main()
{
  int A, B, C, D;
  scanf("%d%d%d%d",&A,&B,&C,&D);
  printf("%d\n", sommamax(A,B,C,D));
}
```


Dichiarazione di funzione

Regola Generale:

Prima di utilizzare una funzione è necessario che sia già stata **definita oppure *dichiarata***.

Funzioni C:

- **definizione**: descrive le proprietà della funzione (tipo, nome, lista parametri formali) e la sua realizzazione (lista delle istruzioni contenute nel blocco).
- **dichiarazione (*prototipo*)**: descrive le proprietà della funzione senza definirne 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:

```
int max(int a, int b);
```


Esempio:

```
#include <stdio.h>

main()
{
  int A, B;
  printf("Dammi A e B: ");
  scanf("%d %d", &A, &B);
  printf("%d\n", max(A,B));
}

int max (int a, int b) {
  int result;
  if (a>b) result=a;
  else result=b;
  return result;
}
```

- In questo caso il compilatore segnala un **errore** in corrispondenza della chiamata **max(A,B)**, perché viene usato un identificatore che viene definito successivamente (dopo il main())

Soluzione:

```
#include <stdio.h>

int max(int a, int b);

main()
{
  int A, B;
  printf("Dammi A e B: ");
  scanf("%d %d", &A, &B);
  printf("%d\n", max(A,B));
}

int max (int a, int b) /*intestaz. */
{ int result;
  if (a>b) result=a;
  else result=b;
  return result;
}
```

E le dichiarazioni di printf, scanf etc. ?

- sono contenute nel file stdio.h:

#include <stdio.h>

provoca l'inserimento del contenuto del file specificato.

Dichiarazione di Funzioni

Una funzione può essere ***dichiarata*** in punti diversi, ma è ***definita*** una sola volta.

E` possibile inserire i prototipi delle funzioni utilizzate:

- nella parte dichiarazioni globali di un programma *(consigliato)*,
- nella parte dichiarazioni del **main**.

Ad esempio:

```
void main()
{   long power (int base, int n);
    int X, exp;

    scanf("%d%d", &X, &exp);
    printf("%ld", power(X,exp));
}
```

oppure:

```
long power (int base, int n);
void main()
{   int X, exp;

    scanf("%d%d", &X, &exp);
    printf("%ld", power(X,exp));
}
```


Struttura dei Programmi C

Spesso si strutturano i programmi in modo tale che la definizione del main compaia prima delle definizioni delle altre funzioni (per favorire la **leggibilità**).

Protocollo da utilizzare:

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

Ad esempio:

Calcolo della radice intera di un numero intero letto a terminale.

```
#include <stdio.h>

/* dichiarazioni delle funzioni: */
   int   RadiceInt(int par);
   int   Quadrato(int par);

main(void)
{
   int   X;
   scanf("%d", &X);
   printf("Radice: %d\n", RadiceInt(X));
   printf("Quadrato: %d\n", Quadrato(X));
}

/* definizione funzioni: */
int   RadiceInt (int par)
{
 int cont = 1;
 while (cont*cont <= par)
  cont = cont + 1;
 return (cont-1);
}

int   Quadrato (int par)
{
 return (par*par);
}
```


Tecniche di legame dei parametri

Come viene realizzata l'associazione tra parametri attuali e parametri formali?

In generale, esistono vari meccanismi di legame dei parametri.

Meccanismi:

- Legame per **valore** (C, Pascal);
- Legame per **indirizzo**, o per riferimento (Pascal, Fortran).

Tecniche di Legame dei parametri

Per spiegare le varie tecniche di legame faremo riferimento alla seguente situazione:

Consideriamo una procedura **P** con un parametro formale **pf**.
Supponiamo che **P** venga chiamata da una unità di programma **Q**, mediante la chiamata:

P(pa)

dove **pa** è il parametro attuale, variabile visibile in **Q**.

Quindi, utilizzando la sintassi C:

Unità Q

```
int pa;
…
P(pa);
…
```

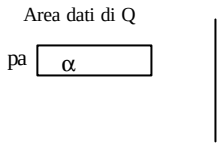
Unità P:

```
void P(int  pf)
{
…
}
```


Legame per valore

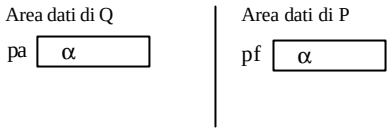
Se il legame dei parametri avviene per valore:

1. Prima della chiamata:



2. Al momento della chiamata:

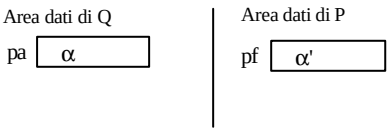
- viene allocata una cella di memoria associata a pf nell'area dati accessibile a P
- viene valutato pa, ed il suo valore viene **copiato** in pf



Esecuzione di P:

Il parametro formale **pf** viene trattato come una **variabile locale** al sottoprogramma P: può essere modificato mediante assegnamento, etc.. In generale, al termine della chiamata, pf potrà assumere un valore α', diverso da quello iniziale.

Alla fine dell'esecuzione di P:



- Al termine della chiamata, il valore di pa rimane **inalterato.**

Legame per valore

Quindi:

Se il legame dei parametri avviene per valore, immediatamente dopo l'esecuzione della chiamata, il parametro attuale (pa) mantiene il valore che aveva immediatamente prima della chiamata

- Parametri passati per valore servono soltanto a comunicare **valori in ingresso** al sotto-programma.
- Se il passaggio avviene per valore, pa non è necessariamente una variabile, ma può essere, in generale, un'**espressione**.

Il legame per valore è l'unica tecnica di legame disponibile in C.

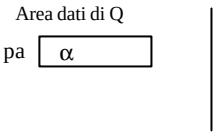
Ad esempio:

```
#include <stdio.h>

void P(int pf);

main()
{ int pa=10;

  P(pa);
  printf("valore finale di pa: %d\n",
    pa); /* pa vale 10 */
}

void P(int pf)
{
  pf=100;
  printf("valore finale di pf: %d\n",
           pf);
  return;
}
```


Legame per indirizzo

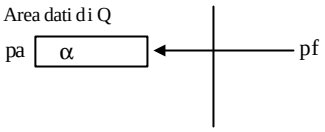
Se il legame dei parametri avviene per indirizzo:

1. Prima della chiamata:



2. Al momento della chiamata:

- viene associato all'identificatore pf la stessa cella di memoria riferita da pa:

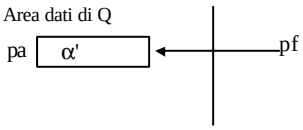


↦ pf è un ***alias*** di pa.

Esecuzione di P:

Il parametro formale **pf** viene trattato come una **variabile locale** al sottoprogramma P: può essere modificato mediante assegnamento, etc.. In generale, al termine della chiamata, pf (e quindi pa) potrà assumere un valore α' diverso da quello iniziale.

Alla fine dell'esecuzione di P:



- Al termine della chiamata, il valore di pa risulta **modificato.**

Legame per indirizzo

Quindi:

Se il legame dei parametri avviene per indirizzo, immediatamente dopo l'esecuzione della chiamata, il parametro attuale (pa) può avere un valore diverso da quello che aveva immediatamente prima della chiamata

↦ Parametri passati per indirizzo servono per comunicare valori **sia in ingresso che in uscita** dal sottoprogramma.

↦ Se il passaggio avviene per indirizzo, pa deve necessariamente essere una variabile (cioè, un oggetto dotato di un indirizzo).

In C, il legame per indirizzo non é disponibile.

Esempio:

Utilizziamo la sintassi C per esemplificare il passaggio per indirizzo Il programma che segue è solo a scopo esemplificativo (in C, non c'e` il passaggio per indirizzo!).

Funzione che scambia due variabili X, Y (di tipo integer).

```
#include <stdio.h>
void scambia (int A, int B);
main()
{ int   X, Y;
  scanf("%d %d", &X, &Y);
  scambia(X,Y);
  printf("\n%d \t %d %", X, Y);
}

void scambia (int A, int B)
/* se fosse per indirizzo ! */
{
  int T;

  T=A;
  A=B;
  B=T;
  return;
}
```


Passaggio dei parametri per indirizzo in C

In C questa tecnica di legame **non è prevista**. Si può ottenere lo stesso effetto del passaggio per indirizzo utilizzando ***parametri di tipo puntatore***.

Ad esempio:

```
#include <stdio.h>
void  scambia2(int *A, int *B);

main()
{ int X, Y;
  scanf("%d %d", &X, &Y);
  printf("\n Scambia: %d %d\n",X,Y);
  scambia2(&X,&Y);
  printf("\n%d \t %d %",X,Y);
}

void  scambia2(int *A, int *B)
  {int   T;

    T=*A;
    *A=*B;
    *B=T;
  }
```


Esempio:

```
#include <stdio.h>
void  Fun(int  X);
int N;
main()
{
  N=3;

  Fun(N);
  printf("%d\n", N);     {3}
}

void  Fun (int  X)
  {
    X=X+1;
    printf("%d\n", N);       {1}
    printf("%d\n", X);       {2}
  }
```

Se il legame è ***per valore*** viene stampato:

{1} 3
{2} 4
{3} 3

Se il legame è ***per indirizzo*** :

{1} 4
{2} 4
{3} 4