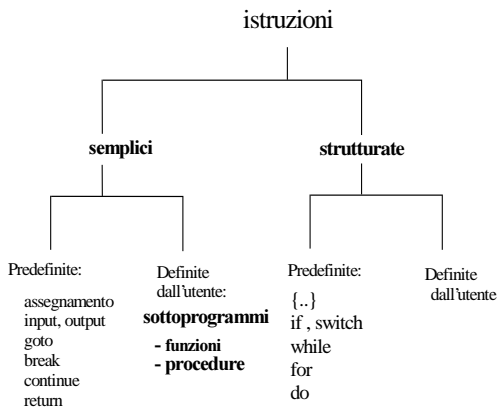


## Sottoprogrammi: Funzioni e Procedure



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 piu' **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
- ✓ leggibilita'
- ✓ possibilita' 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 unita' di programma (**sottoprogrammi**) distinte dal programma principale (*main*).

☞ **D'ora in poi**: il programma e' una **collezione di unita' di programma** (tra le quali compare l'unita' *main*)

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

### Cio' e' 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 sottoprogramma 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**).

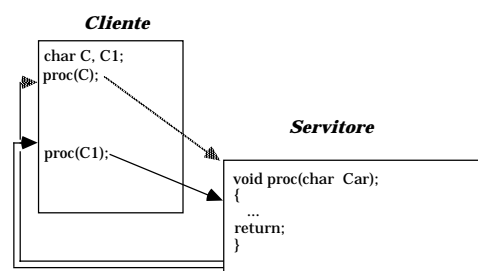
### Utilizzo di funzioni/procedure (*chiamata*):

- 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 sottoprogramma).
- Quando avviene la chiamata, l'esecuzione dell'unità 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'unità 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 sottoprogramma. 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' l'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, e' necessario specificare una **intestazione** e un **blocco** {..}:

### Struttura dell'intestazione:

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

dove:

- **tipo del risultato (codominio)**. Il tipo restituito può essere predefinito o definito dall'utente. Una funzione non può restituire valori di tipo:
  - **vettore**
  - **funzione**
- **identificatore (nome)** della funzione
- **lista dei parametri formali (dominio)**. Per ciascun parametro formale viene specificato il tipo ed un identificatore che e' 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, e' 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:

```
int maggioredi100 (int a) /*intest. */
{ /*parte dichiarazioni: */
  const int C=100;

  /* parte istruzioni: */
  if (a>C) return 1;
  else return 0;
}
```

### Esempio:

```
#define N 100

typedef   char vettore[N];

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

☞ i, v, min sono **variabili locali**

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

### Esempio:

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

Possono esserci **piu' istruzioni return**:

```
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; /* ritorna 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 puo' 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);
}
```

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

```
return;
```

### Uso:

La procedura e' 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*/
{
    if (a>b) return a;
    else return b;
}

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

### Esempio:

```
#include <stdio.h>

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

int sommax(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", sommax(A,B,C,D));
}
```

## 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 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) {
    if (a>b) return a;
    else return b;
}
```

- ☞ In questo caso il compilatore segnala un **errore** in corrispondenza della chiamata **max(A,B)**, perche' 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. */
{
    if (a>b) return a;
    else return b;
}
```

### 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 puo' essere **dichiarata** in punti diversi, ma e' **definita** una sola volta.

E' possibile inserire i prototipi delle funzioni utilizzate:

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

### Ad esempio:

```
main()
{
    long power (int base, int n);
    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 = 0;
    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 piu' comuni:

- 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 unita' di programma **C**, mediante la chiamata:  
**P(pa)**  
dove **pa** e' una variabile visibile in C.

Quindi, utilizzando la sintassi C:

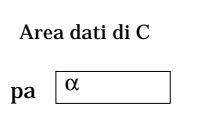
Unita' C
int pa; ... P(pa); ...

Unita' P:
void P(int pf) { ... }

## Legame per valore

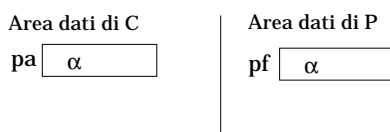
Se il legame dei parametri avviene per valore:

### 1. Prima della chiamata:



### 2. Al momento della chiamata:

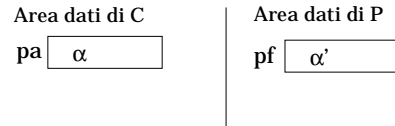
- viene allocata una cella di memoria associata a pf nell'area dati accessibile a P
- viene valutato pa, ed il suo valore viene **copiato** in pf



### 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 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, una **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;

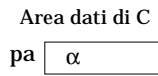
  P(pa);
  printf("valore finale di pa: %d\n",
        pa); /* pa vale 10 */
}

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

## Legame per indirizzo

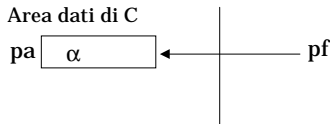
Se il legame dei parametri avviene per indirizzo:

### 1. Prima della chiamata:



### 2. Al momento della chiamata:

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

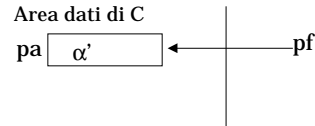


☞ `pf` è un “*alias*” di `pa`.

### 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  $\alpha'$  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 è 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'è il passaggio per indirizzo!).

Funzione che scambia due variabili X, Y (di tipo integer) se  $X > Y$  e restituisce il valore minore tra i due.

```
#include <stdio.h>
void scambia (int A, int B);
main()
{ int X=5,Y=0 ;
  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;
  if (A>B)
  { T=A;
    A=B;
    B=T;
    return; }
  else return;
}
```

## Passaggio dei parametri per indirizzo in C

In C questa tecnica di legame non e' prevista. Si puo' ottenere lo stesso effetto delo passaggio per indirizzo utilizzando *parametri di tipo puntatore*.

### Ad esempio:

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

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

int  scambia2 (int *A, int *B)
{int  T;
  if (*A>*B)
  { T=*A;
    *A=*B;
    *B=T;
    return *A;
  }
  else return *B;
}
```

### Esempio:

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

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

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

Se il legame e' *per valore* abbiamo le stampe:

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

Se il legame e' *per indirizzo*:

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

### Esempio:

```
#include <stdio.h>
void h (int X, int *Y);

main()
{int  A,B;
  A=0;
  B=0;
  h(A, &B); /*B e' un parametro
             di uscita*/
  printf("\n %d \t %d", A, B);
}

void h (int  X, int *Y)
{
  X=X+1;
  *Y=*Y+1;
  printf("\n %d \t %d", X, *Y);
}
```

1	1	(stampa di "h")
0	1	(stampa di "main")

### Esercizio:

Calcolo delle radici di una equazione di secondo grado.

$$Ax^2 + Bx + C = 0$$

```
#include <stdio.h>
#include <math.h>

typedef enum {false,true} boolean;

boolean radici(int A, int B, int C,
              float *X1, float *X2);

main()
{ int  A,B,C;
  float X,Y;
  scanf ("%d%d%d\n",&A,&B,&C);
  if ( radici(A,B,C,&X,&Y) )
    printf ("%f%f\n",X,Y);
}
```

```

boolean radici(int A, int B, int C,
              float *X1, float *X2)
{ float D;
  D= B*B-4*A*C;
  if (D<0) return 0;
  else
    { D=sqrt(D);
      *X1 = (-B+D)/(2*A);
      *X2= (-B-D)/(2*A);
      return 1;
    }
}

```

### ✍ Esercizio:

Programma che stampa i numeri primi compresi tra 1 ed n (n dato).

```

#include <stdio.h>
#include <math.h>

#define NO 0
#define YES 1

int isPrime(int n); /*dichiarazioni*/
int primes(int n);

void main()
{
  int n;

  do {
    printf("\nNumeri primi non
           superiori a:\t");
    scanf("%d",&n);
  } while (n<1);

  printf("\nTrovati %d numeri primi.\n",
        primes(n));
}

```

```

int isPrime(int n)
{
  int max,i;

  if (n>0 && n<4) return YES;
  /* 1, 2 e 3 sono primi */
  else if (!(n%2)) return NO;
  /* escludi i pari > 2 */
  max = sqrt( (double)n );
  /* CAST:sqrt ha arg double */
  for(i=3; i<=max; i+=2)
    if (!(n%i)) return NO;
  return YES;
}

int primes(int n)
{
  int i,count;

  if (n<=0) return -1;
  else count=0;

  if (n>=1)
    { printf("%5d\t",1);
      count++;
    }
  if (n>=2)
    { printf("%5d\t",2);
      count++; }
  for(i=3;i<=n;i+=2)
    if (isPrime(i))
      { printf("%5d\t",i);
        count++;
      }
  printf("\n");
  return count;
}

```

### ✍ Esercizio:

Scrivere una procedura che risolve un sistema lineare di due equazioni in due incognite

$$a_1x + b_1y = c_1$$

$$a_2x + b_2y = c_2$$

$$x = (c_1b_2 - c_2b_1) / (a_1b_2 - a_2b_1) = XN / D$$

$$y = (a_1c_2 - a_2c_1) / (a_1b_2 - a_2b_1) = YN / D$$

### Soluzione:

```

#include <stdio.h>
void sistema(int A1, int B1, int C1,
             int A2, int B2, int C2,
             float *X, float *Y);

main()
{
  int A1,B1,C1,A2,B2,C2;
  float X,Y;

  scanf("%d%d%d\n",&A1,&B1,&C1);
  scanf("%d%d%d\n",&A2,&B2,&C2);

  sistema(A1,B1,C1,A2,B2,C2,&X,&Y);
  printf("%f%f\n",X,Y);
}

```

```

void sistema (int A1, int B1, int C1,
             int A2, int B2, int C2,
             float *X, float *Y)
{
    int    XN,YN,D;
    XN = (C1*B2 - C2*B1);
    D = (A1*B2 - A2*B1);
    YN = (A1*C2 - A2*C1);
    if (D == 0)
        {if (XN == 0)
            printf("sistema indeterminato\n");
            else
            printf("sistema impossibile\n");
        }
    else
        { printf("Determinante%d",D);
          *X=(float) (XN) /D;
          *Y=(float) (YN) /D;
        }
}

```

## Vettori come parametri di funzioni

In C i vettori sono sempre passati **attraverso il loro indirizzo**:

### Ad esempio:

```

#include <stdio.h>
#define MAXDIM 30

int getvet(int v[], int maxdim);

main()
{
    int k, vet[MAXDIM];
    int dim;
    dim=getvet(vet,MAXDIM);
    ...
}

```

### Definizione della funzione:

```

int getvet(int *v, int maxdim);
{ int i;

  for(i=0;i<maxdim;i++)
    {printf("%d elemento:\t", i+1);
     scanf("%d", &v[i]); }
  return n;
}

```

☞ Il vettore e' modificato all'interno della funzione e le modifiche sopravvivono all'esecuzione della funzione poiche' in C i vettori sono trasferiti **per indirizzo**.

## Struttura di un Programma C

Se un programma fa uso di funzioni/procedure, la sua struttura viene estesa come segue:

```

#include <stdio.h>
/* variabili e tipi globali al programma:
   visibilita' nell'intero programma */
tipovar nomevar, ...;

/* dichiarazioni funzioni */
int F1 (parametri);
...
int FN (parametri);

/*main*/
main (void)
{ /* variabili locali al main:
   visibilita' nel solo main */

/*codice del main: si invocano le Fi */
} /* fine main */

/* definizioni funzioni */
int F1 (... )
{ /* parte dichiarativa */
/* codice di F1 */ }
}

```

☞ Le definizioni di funzioni non possono essere innestate in altre funzioni o blocchi.

## Variabili locali:

Nella parte dichiarativa di un sottoprogramma (procedura o funzione) possono essere dichiarati costanti, tipi, variabili (detti **locali** o **automatiche**).

```

#include <stdio.h>
char saltabianchi (void);
main(void)
{char C;
  C = saltabianchi();
  printf("\n%c",C); /* stampa */
}

char saltabianchi (void)
{char Car; /* Car e' locale*/
  do
  { scanf("%c", &Car);}
  while (Car==' ');
  return Car;
}

```

☞ Alla variabile Car si puo' far riferimento solo nel corpo della funzione saltabianchi (**campo di azione**).

☞ Il **tempo di vita** di Car e' il tempo di esecuzione della funzione saltabianchi.

☞ I parametri formali vengono trattati come variabili locali.

## Variabili Locali

- Quando una funzione viene chiamata, viene creata una associazione tra l'identificatore di ogni variabile locale (e parametro formale) ed una cella di memoria allocata automaticamente.

### Esempio:

```
int f(char Car)
{
    int P;

    main()
    { char C;

      f(C);
    }
}
```

- Alla fine dell'attivazione ogni cella di memoria associata a variabili locali viene deallocata. Se la procedura viene attivata di nuovo, viene creata una nuova associazione.
- Non c'è correlazione tra i valori che Car assume durante le varie attivazioni della funzione f.

## Variabili esterne (o globali)

- Nell'ambito del blocco di un sottoprogramma ( del main) si può far riferimento anche ad identificatori **globali**, nella **parte dichiarazioni globali** del programma.
- Il **tempo di vita** delle variabili esterne è pari al tempo di esecuzione del programma (**variabili statiche**).

### Esempio:

```
#include <stdio.h>
void saltabianchi (void);

char C; /* def. variabile esterna */

main()
{
    saltabianchi();
    printf("\n%c",C);      /* stampa C */
}

void saltabianchi (void)
{
    do
        {scanf("%c", &C);}
    while (C!=' ');
}
```

## Variabili Esterne

### Nell'esempio:

- C è una **variabile esterna** sia al main che a saltabianchi.
- la definizione di C è visibile sia dalla funzione main che dalla procedura saltabianchi. Entrambe queste unità possono far riferimento alla variabile C.
- È la **stessa** variabile. Ogni modifica a C prodotta dalla funzione, viene "vista" anche dal main.

### ⇒ possibilità di effetti collaterali

## Effetti collaterali

Si chiama effetto collaterale (**side effect**) provocato dall'attivazione di una funzione la modifica di una qualunque tra le variabili **esterne**.

Si possono avere nei seguenti casi:

✓ *parametri di tipo puntatore;*

✓ assegnamento a *variabili esterne*.

☞ Se presenti, le funzioni non sono più funzioni in senso matematico.

### Esempio:

```
#include <stdio.h>
int B;
int f (int * A);

main()
{ B=1;
  printf("%d\n",2*f(&B)); /* (1) */
  B=1;
  printf("%d\n",f(&B)+f(&B)); /* (2) */
}

int f (int * A)
{ *A=2*(*A);
  return *A;
}
```

☞ Fornisce valori diversi, pur essendo attivata con lo stesso parametro attuale. L'istruzione (1) stampa 4 mentre l'istruzione (2) stampa 6.

## Effetti Collaterali

### Esempio:

```
int V=2;

float f (int X)
{
  V=V*X; /* origine side effect */
  return (X+1)
}

main()
{ int B;
  B=2;
  printf ("%f",V+f(B));
  B=2;V=2;
  printf ("%f",f(B)+V);
}
```

In questo caso:

$$V+f(X) \neq f(X) +V$$

### Eliminazione degli Effetti Collaterali:

Per evitare effetti collaterali in funzioni occorre:

- non avere parametri passati per indirizzo nelle intestazioni di funzioni;
- non introdurre assegnamenti a variabili esterne nel corpo di funzioni.

## Variabili esterne & passaggio dei parametri

Le variabili esterne rappresentano un modo alternativo ai parametri per far *interagire* tra loro le varie unita' di programma.

### Vantaggi:

- Evitano lunghe liste di parametri (da copiare se passati per valore).
- Permettono di restituire in modo diretto i risultati al chiamante.

### Svantaggi:

- I programmi risultano meno leggibili (rischio di errori).
- Interazione meno esplicita tra le diverse unita' di programma (effetti collaterali).
- Generalita', riusabilita' e portabilita' diminuiscono.

## Visibilita' degli Identificatori e Tempo di Vita

Dato un programma P, costituito da diverse unita' di programma, eventualmente scomposte in blocchi, si distingue tra:

- **Ambiente globale**  
e' costituito dalle dichiarazioni e definizioni che compaiono nella parte di dichiarazioni globali di P.
- **Ambiente locale a una funzione:**  
e' l'insieme delle dichiarazioni e definizioni che compaiono nella parte dichiarazioni della funzione, piu' i suoi parametri formali.
- **Ambiente di un blocco**  
e' l'insieme delle dichiarazioni e definizioni che compaiono all'interno del blocco.

## Visibilita' degli Identificatori

Dato un identificatore, il suo **campo di azione** (o scope di **visibilita'**) e' costituito dall'insieme di tutte le istruzioni che possono utilizzarlo.

### Regole di visibilita' degli identificatori in C:

- il campo di azione della dichiarazione (o definizione) di un identificatore **esterno** va dal punto in cui si trova la dichiarazione/definizione fino alla fine del file sorgente, a meno della regola (2);
- il campo di azione della dichiarazione (o definizione) di un identificatore **locale** e' il blocco (o la funzione) in cui essa compare e tutti i blocchi in esso contenuti, a meno di ridefinizioni (v. regola (2));
- quando un identificatore dichiarato in un blocco P e' ridichiarato (o ridefinito) in un blocco Q racchiuso da P, allora il blocco Q, e tutti i blocchi innestati in Q, sono esclusi dal campo di azione della dichiarazione dell'identificatore in P (*overriding*).

☞ Il campo di azione di ogni identificatore e' determinato **staticamente**, dalla struttura del testo del programma (**regole di visibilita' lessicali**).

## Visibilita' degli Identificatori

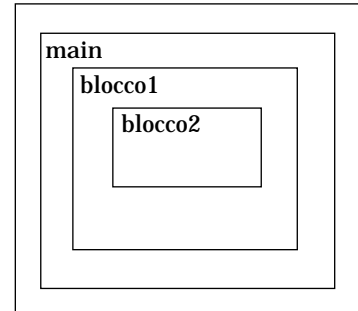
Dalle regole precedenti, possiamo concludere che:

- Per una **variabile locale** (e per i **parametri formali**), dichiarata in una funzione, il campo di azione e' la funzione stessa.
- Per una **variabile esterna** (cosi' come per le funzioni che sono tutte dichiarate esternamente) il campo di azione va dal punto in cui si trova la dichiarazione fino alla fine del file sorgente.

## Esempio:

```
#include <stdio.h>
main(void)
{int i=0;
  while (i<=3)
  { /* BLOCCO 1 */
    int j=4; /* def. locale al blocco 1*/
    j=j+i;
    i++;

    { /* BLOCCO 2: interno al blocco 1*/
      float i=j; /*locale al blocco 2*/
      printf("%f\t%d\t",i,j);
    }
    printf("%d\t\n",i);
  }
}
```



## Esempio:

```
#include <stdio.h>
int X=0;
void P1 (); /* superflua */
void P2 ();

main()
{ X++;
  P2;
}

void P1 ()
{ printf("\n%d",X);
}

void P2 ()
{float X;
  X=2.14;
  P1;
}
```

☞ Stampa il valore 1.

☞ Con regole di visibilita' dinamiche (non adottate dal C, ma ad esempio in LISP), stamperebbe il valore 2.14.

## Tempo di vita delle variabili

E' l'intervallo di tempo che intercorre tra l'istante della creazione (allocazione) della variabile e l'istante della sua distruzione (deallocazione).

- ➡ E' l'intervallo di tempo in cui la variabile **esiste** ed in cui, compatibilmente con le regole di visibilita', puo' essere utilizzata.

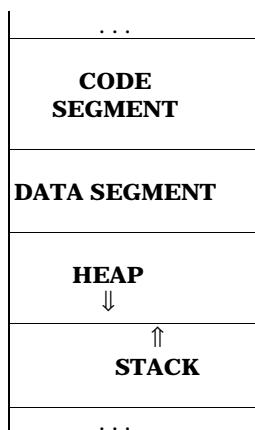
## In C:

- Variabili **esterne** sono allocate all'inizio del programma e vengono distrutte quando il programma termina ➡ il tempo di vita e' pari al tempo di esecuzione del programma.
- Variabili **locali** e **parametri formali** delle funzioni sono allocati ogni volta che si invoca la funzione e distrutti al termine della funzione ➡ Il **tempo di vita** e' quindi pari alla durata dell'attivazione della funzione in cui compare la definizione della variabile
- Variabili **dinamiche** hanno un tempo di vita pari alla durata dell'intervallo di tempo che intercorre tra la **malloc** che le alloca e la **free** che le dealloca.

## La macchina astratta C: modello a tempo di esecuzione

### Aree di memoria:

- codice (Code segment)
- area dati globale (statica): Data segment
- Heap (dinamica)
- Stack (dinamica)



## Stack

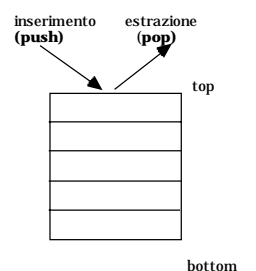
In C le attivazioni delle funzioni sono realizzate utilizzando l'area di memoria "stack" in cui risiede una struttura dati gestita seguendo una disciplina a pila: lo **stack**:

È una struttura dati su cui è possibile eseguire due operazioni:

- inserimento di un elemento (*push*)
- estrazione di un elemento (*pop*)

### Politica di gestione dello stack:

l'ultimo elemento inserito è il primo ad essere estratto (politica LIFO, Last In First Out).



☞ Ad ogni chiamata di sottoprogramma viene creato un elemento (**record d'attivazione**) ed inserito (**push**) in cima allo stack.

## Record d'attivazione

Un record d'attivazione contiene le informazioni relative ad una specifica chiamata di funzione/procedura.

### In particolare:

- 1) nome della funzione attivata e riferimento al codice;
- 2) punto di ritorno al chiamante (**return address**): è l'indirizzo dell'istruzione da eseguire al termine della attivazione;
- 3) riferimento di catena statica (**static link**): è un riferimento all'ambiente visto "staticamente" dal sotto-programma (variabili esterne);
- 4) parametri formali e loro legame con quelli attuali (se per indirizzo);
- 5) variabili locali;
- 6) riferimento al record di attivazione precedente sulla pila (catena dinamica, **dynamic link**): è un riferimento all'ambiente del chiamante.

Al termine dell'esecuzione (**return**), il record di attivazione viene deallocato dallo stack. (operazione di **pop**)

## Record di Attivazione

Quando, l'attivazione della funzione termina (istruzione **return**, o ultima istruzione) l'esecuzione prosegue dall'istruzione memorizzata nel **return address**.

### Esempio:

```
#include <stdio.h>

void R(int A)
{
    printf("Valore: %d\n", A);
}

void Q(int A)
{
    R(A);
}

void P()
{
    int a=10;
    Q(a);
    return;
}

main()
{
    P();
}
```

## Record di Attivazione

### Catena Dinamica:

La **catena dinamica** rappresenta la *storia* delle attivazioni delle unita' di programma.

Attivazioni: (S.O. -->) main --> P --> Q --> R

### Catena statica:

La **catena statica** indica dove cercare (in quale area) i riferimenti per le variabili non locali (**variabili esterne**).

☞ **In C:** le funzioni non possono essere innestate tra loro ⇒ la catena statica fa sempre riferimento all'area dati globale, contenente, ad esempio, le variabili esterne.

### Esempio:

```
#include <stdio.h>

void prova(int *a, int b, int n);

main()
{
    int c[3], d;
    c[0] = 100; c[1] = 15;
    c[2] = 20; d = 0;
    printf("Prima: %d,%d,%d,%d\n",
           c[0],c[1],c[2],d);
    prova(c,d,3);
    printf("Dopo: %d,%d,%d,%d\n",
           c[0],c[1],c[2],d);
}

void prova(int *a, int b, int n)
{
    int i;
    for (i = 1; i < n; i++) a[i] = b;
    b = a[0];
}
```

Il risultato dell'esecuzione di questo programma è:

Prima: 100,15,20,0

Dopo: 100,0,0,0                      **d?**

## Visibilita'/tempo di vita delle variabili:

In C esiste la possibilita' di alterare la visibilita' ed il tempo di vita delle variabili, utilizzando i qualificatori extern, static:

- **extern** (o esterne, globali)
  - dichiarazioni riferite a variabili globali
  - visibili a tutto il programma (anche in file diversi)
  - tempo di vita pari alla durata del programma
- **static** (o statiche)
  - variabili statiche interne alle funzioni
  - non sono visibili all'esterno di queste
  - tempo di vita pari alla durata del programma

### Esempio:

```
#include <stdio.h>
int A;

int f()
{ static int cont=0;
  cont++;
  return cont;
}

main()
{ printf("%d\n", f());
  printf("%d\n", f());
}
```

→ la variabile **static int cont** persiste tra una attivazione di **f** e la successiva: la prima printf stampa 1, la seconda printf stampa 2.

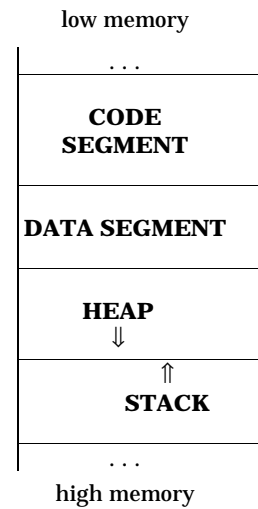
### Esempio:

In C e' possibile distribuire il codice sorgente di un programma su piu' file:

File "AAA.c"	File "BBB.c"
<pre>extern void fun2(...); ... int ncall = 0; ... fun1(...) {   ncall++;   ... }</pre>	<pre>extern fun1(...); void fun2(...); ... extern int ncall; ... void fun2(...) {   fun1();   ncall++;   ... }</pre>

☞ la variabile ncall e le funzioni fun1 e fun2 sono visibili ed utilizzabili in entrambi i file

### Struttura della memoria a tempo di esecuzione



### Allocazione delle variabili

#### Classe di memorizzazione extern:

- applicabile a variabili e funzioni
- default per variabili globali e funzioni
- visibilità globale: visibile ovunque, dal punto di definizione (o dichiarazione) in poi
  - ⇒ visibile anche al di fuori del file che ne contiene la definizione
- permanente: esiste dall'inizio dell'esecuzione del programma, sino alla sua fine
- allocazione:
  - su DATA SEGMENT (per le variabili)
  - su CODE SEGMENT (funzioni)

#### Classe di Memorizzazione static

- applicabile a variabili
- statica - definizione globale o locale
- non altera la **visibilità**:
  - ⇒ **globale** nel caso di definizione globale: visibile ovunque, dal punto di definizione in poi, ma **solo all'interno del file che la contiene**
  - ⇒ **locale** nel caso di definizione locale: visibile solo all'interno del blocco o della funzione in cui è stata definita, dal punto di definizione in poi.
- **permanente**: tempo di vita pari al tempo di esecuzione del programma
- allocazione su **DATA SEGMENT**

## Riassumendo

### Code Segment

- le funzioni nel segmento codice

### Data Segment

- variabili extern (globali multi-file)
- variabili static (globali single-file e locali)

### Stack

- variabili locali e parametri funzioni

### Heap

- strutture dati allocate (**malloc**) e deallocate (**free**) esplicitamente dall'utente e referenziate tramite puntatori
- ☞ Esiste un ulteriore classe di memorizzazione (**register**) che, se possibile, alloca variabili su registri macchina.

## La ricorsione

Una funzione matematica e' definita **ricorsivamente** quando nella sua definizione compare un riferimento a se stessa.

### Esempio:

Funzione fattoriale su interi non negativi:

$$f(n) = n!$$

e' definita ricorsivamente come segue:

$$\begin{aligned} f(n) &= 1 && \text{se } n=0 \text{ (caso base)} \\ f(n) &= n * f(n-1) && \text{se } n > 0 \end{aligned}$$

☞ Usando il metodo induttivo si specifica come tale funzione si comporta nel caso *base* e nel passo *generico*.

### Induzione matematica:

immaginando di avere  $x_k$ , costruisci  $x_{k+1}$ .

☞ Induttivamente, il calcolo del fattoriale di un numero  $n$  viene ricondotto al calcolo del fattoriale di  $n-1$ , in calcolo del fattoriale di  $n-1$  a quello di  $n-2$ , etc., fino a raggiungere un caso base (fattoriale di 0), a risultato noto.

Metodo particolarmente utile per alcuni problemi (intrinsecamente ricorsivi) o che lavorano su strutture dati ricorsive (liste, alberi).

### Esempi di problemi ricorsivi:

- **Somma** dei primi  $n$  numeri naturali:

$$\text{somma}(n) = \begin{cases} 0 & \text{se } n=0 \\ n + \text{somma}(n-1) & \text{altrimenti} \end{cases}$$

- Generare l' $n$ -esimo **numero di Fibonacci**:

$$\text{fib}(n) = \begin{cases} 0 & \text{se } n=0 \\ 1 & \text{se } n=1 \\ \text{fib}(n-1) + \text{fib}(n-2) & \text{altrimenti} \end{cases}$$

- Calcolo del **minimo** di una sequenza di elementi:

$$[a_1, a_2, a_3, \dots] \quad [a_1 \mid [a_2, a_3, \dots]]$$

$$\text{min}([a_1]) \quad \rightarrow \quad a_1$$

$$\text{min}([a_1, a_2]) \quad \rightarrow \quad a_1 \text{ se } a_1 < a_2; \text{ altrimenti } a_2$$

$$\text{min}([a_1 \mid Z]) \quad \rightarrow \quad \text{min}([a_1, \text{min}(Z)])$$

sviluppando

$$\text{min}([a_1 \mid Z])$$

si ottiene:  $\text{min}([a_1, \text{min}([a_2, \text{min}([a_3, \dots]))])$

- Calcolo della **lunghezza** di una sequenza:

$$\text{lung}([]) \quad \rightarrow \quad 0$$

$$\text{lung}([a_1 \mid Z]) \quad \rightarrow \quad 1 + \text{lung}(Z)$$

## Programmazione ricorsiva

Molti linguaggi di programmazione offrono la possibilità di definire funzioni/procedure ricorsive.

### Esempio:

Calcolo del fattoriale di un numero (in C).

```
#include <stdio.h>

int fattoriale(unsigned int n);

main(void)
{int n;
 printf("\nIntrodurre N:\t");
 scanf("%d",&n);
 printf("\nFattoriale di %d:\t%d\n",
        n, fattoriale(n));
}

int fattoriale(unsigned int n)
{
 if (n==0) return 1;
 else return n*fattoriale(n-1);
}
```

☞ Non tutti i linguaggi di alto livello supportano procedure ricorsive (ad esempio il FORTRAN non consente di scrivere sottoprogrammi ricorsivi).

## Calcolo della somma dei primi N naturali

```
#include <stdio.h>

int sum(unsigned int n);

main()
{
 int n;

 printf("\nIntrodurre N:\t");
 scanf("%d",&n);

 printf("\nSomma fino a %d:\t%d\n",
        n, sum(n));
}

int sum(unsigned int n)
{
 if (n==0) return 0;
 else return n+sum(n-1);
 /* ricorsione */
}
```

Sono esempi di **ricorsione lineare** (una sola chiamata ricorsiva nel corpo della funzione).

## N-esimo numero di Fibonacci

$$\text{fib}(n) = \begin{cases} 0 & \text{se } n=0 \\ 1 & \text{se } n=1 \\ \text{fib}(n-1)+\text{fib}(n-2) & \text{altrimenti} \end{cases}$$

```
int fib(unsigned int n)
{
 if (n==0) return 0;
 else
  if (n==1) return 1;
  else return fib(n-1)+fib(n-2);
 /* ricorsione non lineare */
}
```

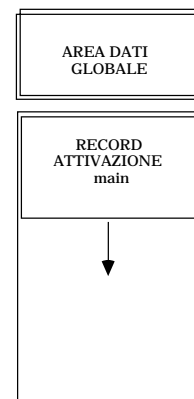
E' un esempio di **ricorsione non lineare** (piu' chiamate ricorsive nel corpo della funzione per determinare il valore restituito dalla funzione)

### ✓ Esercizio:

Pila di attivazioni per la chiamata

```
main()
{
 ...
 printf("Fattoriale di 2:%d\n", fattoriale(2))
}
```

All'inizio dell'esecuzione:



- Dopo la prima attivazione della funzione fattoriale (fattoriale(2)):



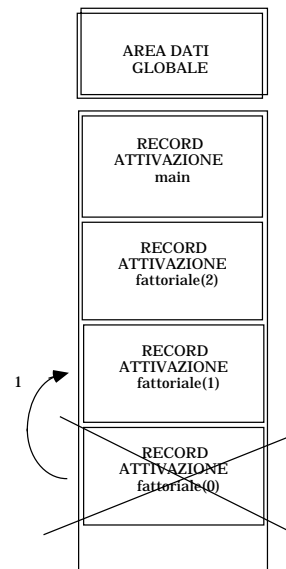
- Dopo la seconda attivazione (fattoriale(1)):



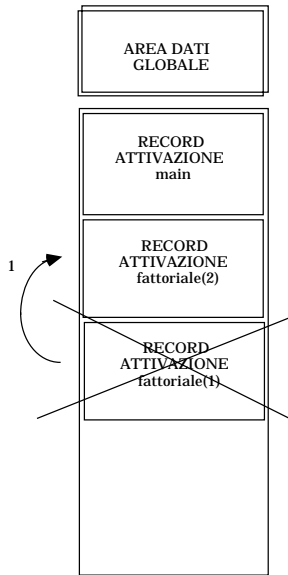
- Dopo la terza attivazione (fattoriale(0)):



- Termine della terza attivazione (return):



- Termine della seconda attivazione (return):



- ☞ Al termine della prima attivazione di fattoriale, viene restituito al main il valore 2, e questo stampa il risultato.

## Ricorsione ed iterazione

La ricorsione e' sempre realizzabile mediante **iterazione**.

### Ad esempio:

Realizzazione iterativa del fattoriale:

```
#include <stdio.h>

int fatt_it(unsigned int n);

main(void)
{
    int n;

    printf("\nIntrodurre N:\t");
    scanf("%d",&n);

    printf("\nFattoriale di %d:\t%d\n",
           n, fatt_it(n));
}

int fatt_it(unsigned int n)
{
    int naux, f;
    f = 1;
    for (naux=1; naux<=n; naux++)
        f *= naux;
    return f;
}
```

## Ricorsione e Iterazione

### Quando conviene utilizzare la ricorsione?

- ☞ Soluzioni ricorsive sono spesso piu' vicine alla definizione matematica di certe funzioni.
- ☞ Versioni iterative sono generalmente **piu' efficienti** di una soluzione ricorsiva (sia in termini di memoria che di tempo di esecuzione).

### Esercizi:

- 1) Scrivere la versione iterativa della procedura per il calcolo dell'n-esimo numero di Fibonacci.
- 2) Scrivere una procedura PrintRev che legge in ingresso una sequenza di caratteri (terminata da '.') e stampa la sequenza al contrario:

**ROMA.  
AMOR**

Definirne una versione ricorsiva senza utilizzare il tipo stringa.

### Soluzioni:

- 1) n-esimo numero di Fibonacci: versione iterativa

```
int fib_it(unsigned int n)
{
    unsigned int i,x=1,y=0,z;
    for(i=1;i<=n;i++)
    {
        z = x;
        x += y;
        y=z;
    }
    return x;
}
```

- 2) PrintRev: versione ricorsiva.

```
#include <stdio.h>
#include <string.h>
void print_rev(char car);
main(void)
{
    printf("\nIntrodurre una sequenza
           terminata da .:\t");
    print_rev(getchar());
}
void print_rev(char car);
{
    if (car != '.')
    {
        print_rev(getchar());
        putchar(car);
    }
    else return;
}
```

- ☞ Nella versione ricorsiva, ogni record di attivazione nello stack memorizza un singolo carattere letto (*push*); in fase di *pop*, i caratteri vengono stampati nella sequenza inversa.

☞ Per scriverne una **versione iterativa** e' necessario memorizzare in una struttura dati (vettore) la stringa.

2) PrintRev: versione **iterativa** con stringa (al max 30 caratteri).

```
#include <stdio.h>
#include <string.h>
#define MAXLEN 30

void print_rev_it(char word[])
{
    int i;
    for (i=strlen(word)-1; i>=0; i--)
        putchar(word[i]);
    return;
}

main()
{
    char parola[MAXLEN];

    printf("\nIntrodurre una parola:\t");
    scanf("%s",&parola);

    print_rev_it(parola);
}
```

2) PrintRev: versione **ricorsiva** con stringa.

```
#include <stdio.h>
#include <string.h>
#define MAXLEN 30

void print_rev(char word[], int i)
{
    if (strlen(word)-i>1)
        print_rev(word,i+1);
    putchar(word[i]);
    return;
}

main()
{
    int n;
    char parola[MAXLEN];

    printf("\nIntrodurre una parola:\t");
    scanf("%s",&parola);

    print_rev(parola,0);
}
```

## Ricorsione Tail

Quando la chiamata ricorsiva di una funzione/procedura F e' l'ultima istruzione del codice di F, si dice che F e' tail-ricorsiva.

**Ad esempio:**

```
#include <stdio.h>

int f (int x, int y);

main()
{int n,m;

    printf("\nIntrodurre due numeri:\t");
    scanf("%d%d",&n,&m);
    printf("Somma di %d e %d:\t %d",
          n,m,f(n,m));
}

int f (int x, int y)
{if (x==0) return y;
 else
  if (x>0) return f(x-1,y+1);
  else return f(x+1,y-1);
}
```

(in pratica, f somma x ad y)

## Ricorsione Tail

La computazione che si origina tramite l'invocazione di una funzione tail-ricorsiva corrisponde ad un **processo computazionale iterativo**.

- Un processo computazionale e' **ricorsivo** quando e' caratterizzato da una catena di operazioni posticipate, il cui risultato e' disponibile solo dopo che l'ultimo anello della catena si e' concluso.
- In un processo computazionale **iterativo**, ad ogni passo e' disponibile una frazione del risultato.

**Nell'esempio:**

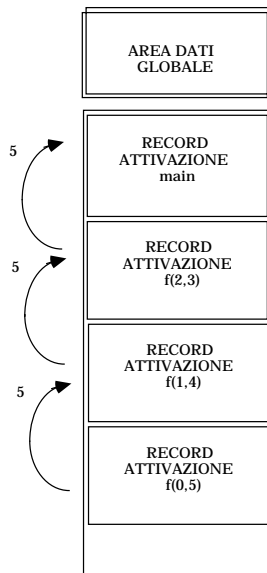
```
Hp:      n=2; m=3
        ...printf(...,f(n,m))
```

f(2,3) --> f(1,4) --> f(0,5) --> return 5

Il risultato non viene ri-elaborato dalle attivazioni intermedie, ma passato semplicemente da ciascuna al chiamante.

➔ Il compilatore, per ottimizzare l'occupazione dello stack potrebbe utilizzare il medesimo record di attivazione per tutte le attivazioni successive della funzione tail ricorsiva.

☞ Consente di ottimizzare lo spazio di memoria allocato sullo stack.



**Esempio:**

Versione tail-ricorsiva del fattoriale.

**Versione ricorsiva:**

```
int fattoriale(unsigned int n)
{
  if (n==0) return 1;
  else
    if (n==1) return 1
    else return n*fattoriale(n-1);
  /* ricorsione */
}
```

**Versione tail-ricorsiva:**

```
int fattoriale(unsigned int n)
{ return fatt_tail(1,n,1);}

int fatt_tail(unsigned int i,
              unsigned int n, long int f)
{ if (i<=n)
  return fatt_tail(++i,n,f*i);
  else return f;
}
```

**Esercizio:**

Scrivere una versione ricorsiva dell'algorithmo che calcola il prodotto come sequenza di somme.

**Soluzione:**

```
#include <stdio.h>
#include <stdlib.h>

int prodotto(int X, int Y);

main()
{
  int X,Y;
  printf("Dammi X ed Y: ");
  scanf("%d%d", &X, &Y);
  printf("Prodotto(%d, %d)=%d\n",X, Y,
        prodotto(X,Y));
}

int prodotto(int X, int Y)
{
  if(Y==0)
    return 0;
  else
    return X+prodotto(X,Y-1);
}
```

**Esercizio:**

Scrivere una versione ricorsiva dell'algorithmo di ordinamento di un vettore per massimi successivi.

**Soluzione:**

```
#include <stdio.h>
#include <stdlib.h>

void ordina(int *V, int N);

main()
{
  int n, *V, i, dim;
  printf("Quanti valori?");
  scanf("%d", &dim);
  V=(int *)malloc(dim*sizeof(int));

  /* lettura dei dati */
  for (i=0; i<dim; i++)
    {printf("valore n. %d: ",i);
     scanf("%d", &V[i]);
    }
  ordina(V,dim);
  /*stampa dei risultati */
  for(i=0; i<dim; i++)
    printf("Valore di V[%d]=%d\n",i,V[i]);
  free(V);
}
```

```

void ordina(int *V, int N)
{int j, max, tmp;

  if (N==1) return;
  else
  {
    for( max=N-1,j=0; j<N; j++)
      if (V[j]>V[max]) max=j;

    if (max<N-1)
      { /*scambio */
        tmp=V[N-1];
        V[N-1]=V[max];
        V[max]=tmp;
      }
    ordina(V, N-1);
  }
/*scansione sottovettore */
}

```

## Funzioni come parametri

### Funzioni:

#### Vincolo nell'uso dei parametri:

non e' possibile passare a funzioni e procedure parametri di tipo funzione.

#### Vincolo sul risultato:

le funzioni non possono ritornare funzioni come risultato.

### ma:

- e' possibile passare a funzioni parametri di tipo **puntatore a funzione**.
- e' possibile ottenere come risultato di una funzione, un **puntatore a funzione**.

#### Puntatore a funzione:

```
<tipo_f> (* <nomeF>)(<lista_par_formali>);
```

dove:

- <nomeF> e' l'identificatore del puntatore a funzione;
- <tipo\_f> e' il tipo di dato restituito dalla funzione puntata
- <lista\_par\_formali> e' la lista dei parametri formali della funzione puntata

## Puntatore a Funzione

### Esempio:

```
double (* f) (double par);
```

- ☞ Priorita' () rispetto a \*:  
f punta alla funzione con un parametro di ingresso double e restituisce un double
- ☞ Il C considera l'**identificatore** di una funzione come un **puntatore** al suo codice.

## Funzioni come parametri

Per avere un parametro F2 di tipo (puntatore a) funzione, occorre specificare un **parametro formale** di tipo funzione, come segue:

```
tipo1 F1(tipo2 (*F2)(lista-par-formali-F2),
        altri-par-formali-F1)
```

- la funzione F1 ha il puntatore a funzione F2 come parametro formale.
- Nel corpo di F1 e' possibile chiamare la funzione puntata da F2 mediante dereferencing:

```

tipo1 F1(tipo2 (*F2)(lista-par-formali-F2),
        altri-par-formali-F1)
{ tipo2 k;
  ...
  k=*F2(...); /*chiamata mediante
               dereferencing*/
}

```

- Un **parametro formale** funzione e' specificato indicando un **nome** (puntatore), la **lista dei parametri formali** ed il **tipo di risultato**.
- Il **parametro attuale** con cui si invoca F1 deve essere un identificatore di funzione con la stessa descrizione dei parametri e lo stesso tipo di risultato di F2.

## Alcuni Esempi

### Definizione/dichiarazione di funzioni:

```
double fun (double x);

double sommaquadratif
(double (* f)(double par), int m, int n)
{...}
```

### Uso del parametro

```
somma = somma + (* f)(k) * (* f)(k);
```

### Invocazione

```
sommaquadratif (fun, 1, 10000);
```

### Esempio

```
#include <math.h>
double fun(double x) /*reciproco */
{ return 1.0 / x;}

double sommaquadratif
(double (* f)(double par), int m, int n)
{int k;
double somma;
somma = 0;
for (k=m; k <= n; k++)
    somma = somma + (*f)(k)*(*f)(k);
return somma;
}

double sommacubif
(double (*f) (double par), int m, int n)
{ int k; double cubo;
cubo= 0;
for (k=m; k <= n; k++)
    cubo=cubo+(*f)(k)*(*f)(k)*(*f)(k);
return cubo;
}

main ()
{int a, b, c, i;
printf (" Inversi %.7f\n",
    sommaquadratif (fun,1, 10000));
printf (" Seni %.7f\n",
    sommaquadratif (sin, 2, 13));
printf (" Inversi %.7f\n",
    sommacubif (fun, 1, 10000));
printf (" Seni %.7f\n",
    sommacubif (sin, 2, 13));
}
```

### Esercizio:

Scrivere un programma C che rappresenta una funzione *Sigma*.

*Sigma*, dati in ingresso un parametro funzione F a valore intero, e due estremi (interi) Inferiore, Superiore, calcola la sommatoria:

$$\sum_{(i=\text{Inferiore}.. \text{Superiore})}F(i)$$

Scrivere un programma che utilizzando questa funzione calcoli:

$$\sum_{(i=0..100)}i$$

$$\sum_{(i=0..100)}i^2$$

$$\sum_{(i=0..100)}i^3$$

Rendere "parametrico" il programma nei valori inf, sup, n per calcolare:

$$\sum_{(i=\text{inf}..sup)}i^n$$

### Soluzione

```
#include <stdio.h>

int sommatoria
(int (* f)(int par, int N), int inf, int
sup, int n)
{int k;
double somma;
somma = 0;
for (k=inf; k <= sup; k++)
    somma = somma + (*f)(k, n);
return somma;
}

int power(int base, int exp)
{ int i, pow=1;
for (i=1; i<=exp; i++)
    pow*=base;
return pow;
}

main ()
{
int INF,SUP,N, RIS;
printf("Inserire INF e SUP:");
scanf("%d%d", &INF, &SUP);
printf("Esponente?");
scanf("%d", &N);
RIS=sommatoria(power,INF,SUP,N);
printf("\nRisultato: %d", RIS);
}
```

## Dichiarazione di un tipo funzione

E' possibile dichiarare un **tipo funzione**, da utilizzare poi per la dichiarazione di **variabili** o come identificatore di **tipo** di un **parametro formale** o **risultato** di un'altra funzione.

Si definiscono, in pratica, come variabili di tipo **puntatore a funzione**:

```
typedef int fprot1 (int a, int b); /*tipo funzione */
```

```
typedef fprot1 * funptr; /*tipo punt. a funzione*/
```

### Ad esempio:

```
typedef double (* ptrfunzione) (double);  
ptrfunzione vlptr;
```

☞ Variabili di questo tipo possono essere usate in **assegnamenti**:

```
vlptr = &sin;
```

☞ questo assegnamento non implica alcuna valutazione della funzione.

## Funzioni come risultati di funzioni

Una funzione puo' restituire un **puntatore a funzione**.

### Esempio:

```
typedef int fprot1 (int a, int b);  
typedef fprot1 *funptr;
```

```
fprot1 *select1 (int a) ...
```

La funzione `select1` ritorna un puntatore a funzione del tipo a due parametri interi e un intero di ritorno

### In alternativa:

```
funptr select2 (int a) ...
```

oppure:

```
int (* select3 (int a)) (int a, int b) ...
```

### Esempio:

```
#include <stdio.h>  
typedef int fprot1 (int a, int b);  
typedef fprot1 *funptr;  
  
int selecta (int a, int b)  
{ return a + b; }  
  
int selectb (int a, int b)  
{ return a - b; }  
  
int selectc (int a, int b)  
{ return a * b; }  
  
funptr select2 (int a)  
{  
  switch (a)  
  {  
    case 1: return selecta; break;  
    case 2: return selectb; break;  
    case 3: return selectc; break;  
    default: return selecta;  
  }  
}  
  
main()  
{  
  int x, y, scelta;  
  funptr F;  
  scanf("%d%dScelta:%d",&x,&y,&scelta);  
  F=select2(scelta);  
  printf("Risultato:%d\n", (*F)(x,y));  
}
```

## Argomenti delle linee di comando: *argc*, *argv*

Anche la funzione **main** puo' avere parametri. I parametri rappresentano gli eventuali argomenti passati al programma, quando viene messo in esecuzione:

```
prog arg1 arg2 ... argN
```

I parametri formali di `main`, differentemente dalle altre funzioni, sono sempre due:

- `argc`
- `argv`

### int argc:

e' un parametro di tipo **intero**. Rappresenta il numero degli argomenti effettivamente passati al programma nella linee di comando con cui si invoca la sua esecuzione. Anche il nome stesso del programma (nell'esempio, `prog`) e' considerato un argomento, quindi `argc` vale sempre almeno 1.

### char \*\*argv:

vettore di stringhe, ciascuna delle quali contiene un diverso argomento. Gli argomenti sono memorizzati nel vettore nell'ordine con cui sono dati dall'utente.

- Per convenzione, `argv[0]` contiene il **nome del programma stesso** (cioe', il nome del file eseguibile).

**Ad esempio, se l'esecuzione e' determinata da un comando del tipo:**

```
prog arg1 arg2 ... argN
```

**Allora:**

**argc** vale **N+1**

**argv** risulta:

argv[0]="prog"

argv[1]="arg1"

argv[2]="arg2"

...

argv[N]="argN"

Per convenzione, **argv[argc]** vale **NULL**.

### Esempio:

Programma che stampa i suoi argomenti.

```
#include <stdio.h>
/* programma esempio.exe */
main(int argc, char *argv[])
{
    int i;

    for(i=0; i<argc; i++)
        printf("%s%s",argv[i],
            (i<argc-1)?"\t":"\n");
    return 0;
}
```

### Invocazione:

esempio a b c zeta

### Stampa:

esempio a b c zeta

-