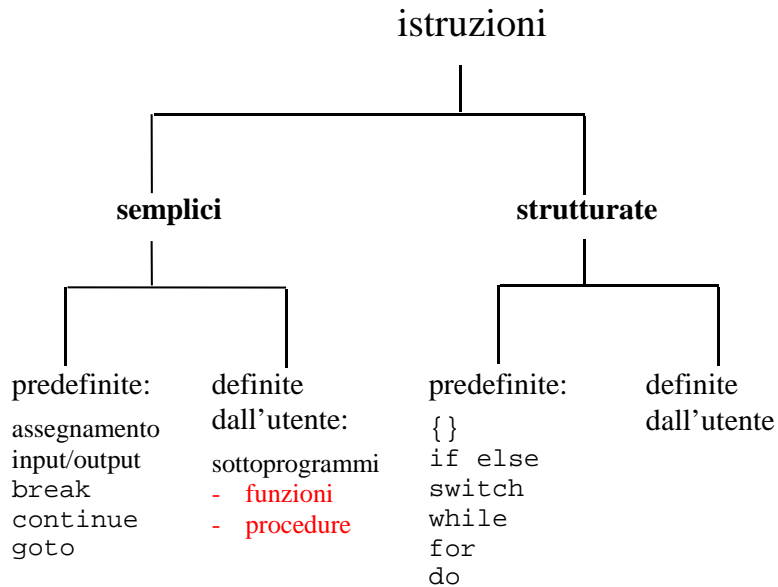


Sottoprogrammi: funzioni e procedure



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

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-1; i++) {
        quanti = dim-i;
        max = quanti-1;
        for(j=0; j<quanti-1; 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("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 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

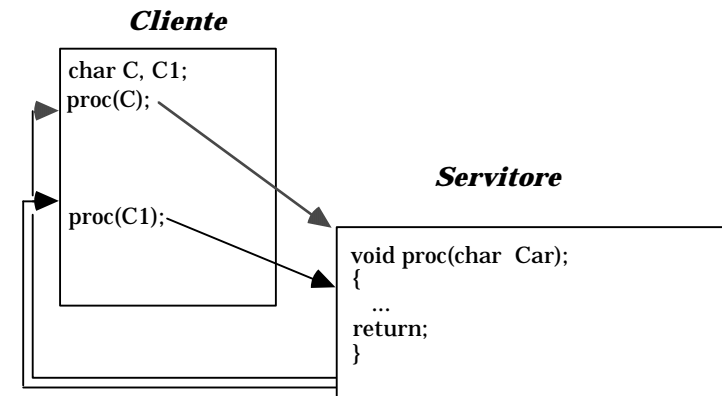
Quando si verifica una chiamata a sottoprogramma, si possono individuare due entità:

- 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 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.

Legame dei 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*, ovvero 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 sottoalgoritmo, 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.

Funzioni e procedure: differenze

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

Procedura

È un'astrazione della nozione di **istruzione**. È un'istruzione non primitiva utilizzabile in un qualunque punto del programma in cui può comparire un'istruzione.

Funzione

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

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 ed 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, ogni 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 di <espressione>.

Esempio:

```
int maggiorediC (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;
- **visibilità**: sono visibili (cioè utilizzabili) soltanto all'interno della funzione minimo.

Esempio:

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

☞ Possono esserci **più istruzioni return**:

Esempio:

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

☞ Può non esserci **nessuna istruzione return**:

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

☞ In questo caso, il sottoprogramma termina in corrispondenza del simbolo `}` ed il valore restituito è **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
*/
}
```

Chiamata di funzioni in C

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

... nomefunzione(<lista parametri attuali>) ...

Esempio:

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


Realizzazione di procedure in C

Una funzione può anche non restituire alcun 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 la si utilizza, **non si deve specificare alcun argomento**:

```
return;
```

Utilizzo:

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) /* definizione  
max */  
{  
    if(a>b) return a;  
    else return b;  
}  
  
void print_int(int a) /* definizione */  
{  
    printf("%d\\", a);  
    return;  
}  
  
void dummy() /*definizione 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?

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

<elenco 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) {
    if (a>b) return a;
    else return b;
}

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)
{
    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 è necessario che questa sia già stata **definita oppure dichiarata**.

Funzioni C:

- **Definizione:** descrive le proprietà della funzione (tipo, nome, elenco parametri formali) e la sua realizzazione (elenco 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’instestazione della funzione, seguita da ";":

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

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)**, 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) /* intestazione */
{
    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 può essere *dichiarata* in punti diversi, ma deve essere *definita* una sola volta.

È possibile inserire i prototipi delle funzioni utilizzate:

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

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>

E il main?

Formalmente, il main è una funzione che restituisce un valore intero ed i cui parametri formali sono costituiti da un intero e da un vettore di stringhe (i parametri del programma). Quindi:

```
int main(int argc, char **argv)
```

Per i nostri scopi, però, possiamo supporre che il main sia una funzione che non accetta alcun parametro e che non restituisce alcun valore:

```
void main(void)
```

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()
{
    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 più 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 un'unità di programma **C**, mediante la chiamata:

P(pa)

dove **pa** è una variabile visibile in C.

Quindi, utilizzando la sintassi C:

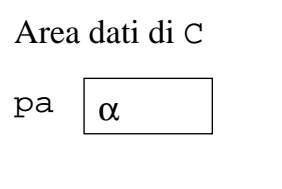
Unità C
<pre>int pa; ... P(pa); ...</pre>

Unità P
<pre>void P(int pf) { ... }</pre>

Legame per valore

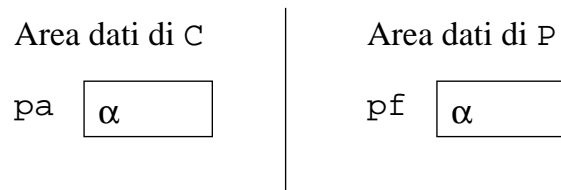
Se il legame dei parametri avviene per valore:

1. Prima della chiamata:



2. Al momento della chiamata:

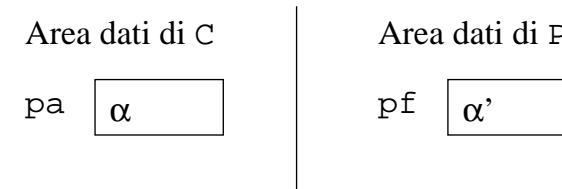
- Viene allocata una cella di memoria associata a pf nell'area dati accessibile a P.
- Viene valutato pa , ed il suo valore viene **copiato** in pf .



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 sottoprogramma.

☞ 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.

Esempio:

```
#include <stdio.h>

void P(int pf);

main()
{
    int pa=10;

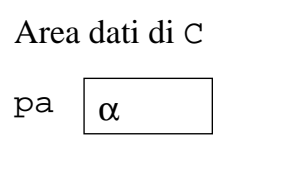
    P(pa);
    printf("valore finale di pa: %d\n",
           pa); /* pa vale 10 */
}

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


Legame per indirizzo

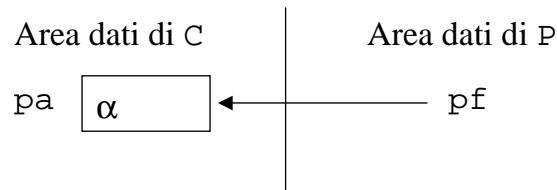
Se il legame dei parametri avviene per indirizzo:

1. Prima della chiamata:



2. Al momento della chiamata:

- Viene associata all'identificatore `pf` la stessa cella di memoria riferita da `pa`:

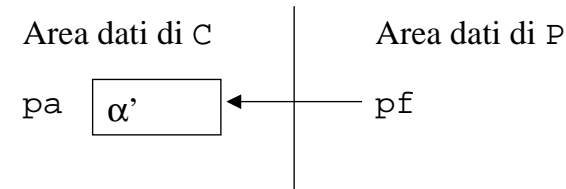


☞ `pf` è un *alias* di `pa`.

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 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 int) se $X > Y$ e restituisce il valore minore tra i due.

```
#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;
    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 è prevista**.

Si può ottenere lo stesso effetto del passaggio per indirizzo utilizzando *parametri di tipo puntatore*.

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 F(? int X);
int N;

main()
{
    N=3;

    F(N);

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

void F(? int X)
{
    X=X+1;
    printf("%d", N); /* {1} */
    printf("%d", 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
```

Funzioni e procedure: esempi

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

main()
{
    int A,B;
    A=0;
    B=0;
    h(A, &B); /* B è 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);
}
```

Stampa:

```
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 false;
    else {
        D=sqrt(D);
        *X1=(-B+D)/(2*A);
        *X2=(-B-D)/(2*A);
        return true;
    }
}
```

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);

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

Esercizio

Scrivere una procedura che risolva 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("Soluzioni: %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);
    YN = (A1*C2 - A2*C1);
    D = (A1*B2 - A2*B1);
    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**:

Esempio:

```
#include <stdio.h>
#define MAXDIM 30

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

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

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 viene modificato all'interno della funzione e le modifiche sopravvivono all'esecuzione della funzione poiché 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:

```
/* variabili e tipi globali al programma:
   visibilità nell'intero programma */
tipovar nomevar, ...;

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

/* main */
main() {
/* variabili locali al main:
   visibilità 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 è locale */
    do {
        scanf("%c", &Car);
    } while(Car==' ');
    return Car;
}
```

- Alla variabile Car si può far riferimento solo nel corpo della funzione saltabianchi (*campo di azione*).
- Il *tempo di vita* di Car è 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 de-allocata. 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 (oppure nel blocco del main) si può far riferimento anche ad identificatori **globali**, nella *parte dichiarazioni globali* del programma.
- Il *tempo di vita* delle variabili globali (o 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 precedent:

- 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 tali effetti sono 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.

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 unità di un 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 unità di programma (effetti collaterali).
- Generalità, riusabilità e portabilità diminuiscono.

Visibilità degli identificatori e tempo di vita

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

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

Visibilità degli identificatori

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

Regole di visibilità degli identificatori in C

1. 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 3.
2. Il campo di azione della dichiarazione (o definizione) di un identificatore locale è il blocco (o la funzione) in cui essa compare e tutti i blocchi in esso contenuti, a meno di ridefinizioni (v. regola 3.).
3. Quando un identificatore dichiarato in un blocco P è 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 è determinato staticamente dalla struttura del testo del programma (**regole di visibilità lessicali**).

Visibilità degli identificatori

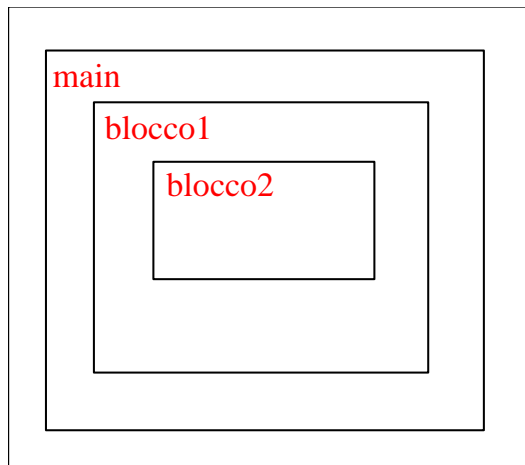
Dalle regole precedenti, possiamo concludere che:

- Per una **variabile locale** (e per i **parametri formali**), dichiarata in una funzione, il campo di azione è la funzione stessa.
- Per una **variabile esterna** (così 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 visibilità dinamiche (non adottate dal C, ma ad esempio, dal LISP), stamperebbe il valore 2.14.

Tempo di vita delle variabili

- È l'intervallo di tempo che intercorre tra l'istante della creazione (allocazione) della variabile e l'istante della sua distruzione (de-allocazione).
- È l'intervallo di tempo in cui la variabile **esiste** ed in cui, compatibilmente con le regole di visibilità, può essere utilizzata.

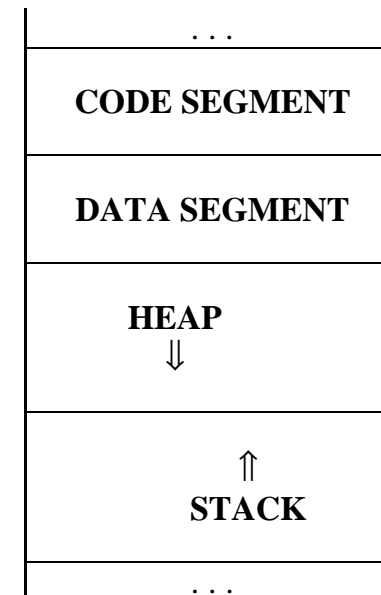
In C:

- Le variabili **esterne** sono allocate all'inizio del programma e vengono distrutte quando il programma termina ➡ il tempo di vita è pari al tempo di esecuzione del programma.
- Le variabili **locali** ed i **parametri formali** delle funzioni sono allocati ogni volta che si invoca la funzione e distrutti al termine della funzione ➡ il **tempo di vita** è quindi pari alla durata dell'attivazione della funzione in cui compare la definizione della variabile.
- Le 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 de-alloca.

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)



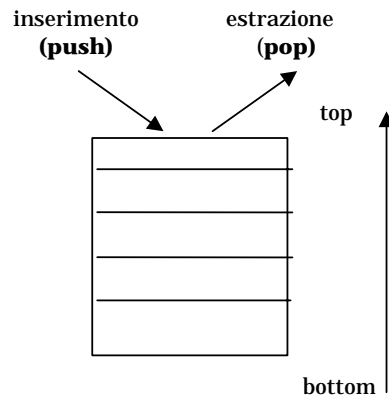
Stack

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 di attivazione**) ed inserito (*push*) in cima allo stack.

Record di 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 sottoprogramma (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**, o *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: punto di ritorno

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

Esempio:

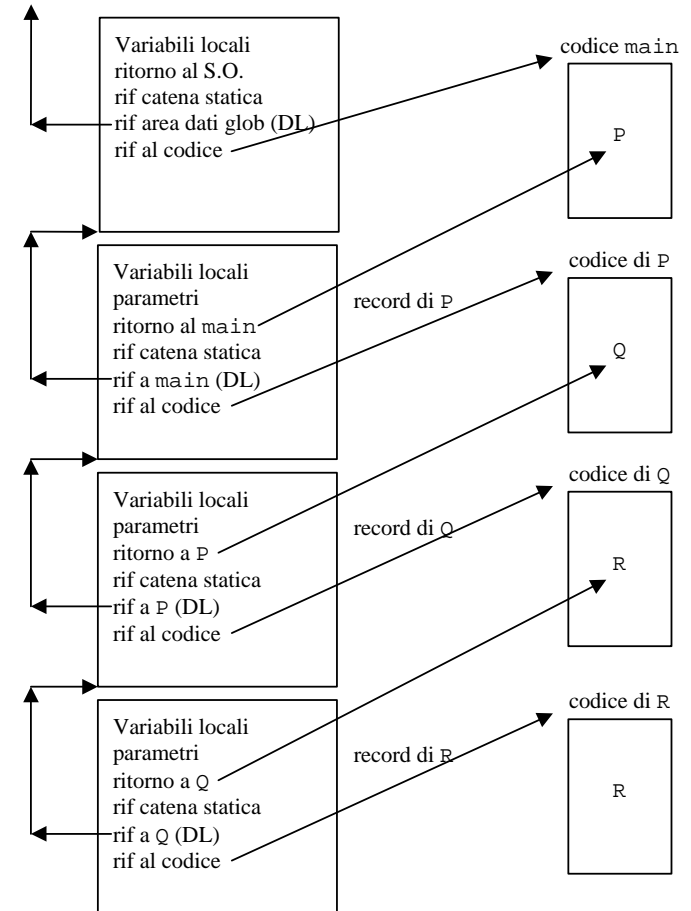
```
#include <stdio.h>

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: catene statica e dinamica

Catena dinamica

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

Attivazioni: (S.O. \rightarrow) `main` \rightarrow P \rightarrow Q \rightarrow 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 contenere definizioni di altre funzioni ➡ 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];
}
```

La ricorsione

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

Esempio:

Funzione fattoriale su interi non negativi:

$$f(n) = n!$$

È definita ricorsivamente come segue:

- $f(n) = 1$ se $n = 0$ (caso **base**)
- $f(n) = n * f(n-1)$ se $n > 0$ (caso **generico**)

☞ Usando il metodo induttivo si specifica come tale funzione si comporta nel caso **base** e nel caso **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 $n-1$; il 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

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

Generare l' n -esimo numero di Fibonacci

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

Calcolo del minimo di una sequenza di elementi

$$[a1, a2, a3, \dots] = [a1 | [a2, a3, \dots]]$$

$$\begin{aligned} min([a1]) &= a1 \\ min([a1, a2]) &= \text{se } a1 < a2, a1; \text{ altrimenti } a2 \\ min([a1 | Z]) &= min([a1, min(Z)]) \end{aligned}$$

Sviluppando $min([a1 | Z])$ si ottiene:

$$min([a1, min([a2, min([a3, \dots])])])$$

Calcolo della lunghezza di una sequenza:

$$\begin{aligned} lung([]) &= 0 \\ lung([a1 | Z]) &= 1 + lung(Z) \end{aligned}$$

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 */
}
```

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

N-esimo numero di Fibonacci

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

```
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*/
}
```

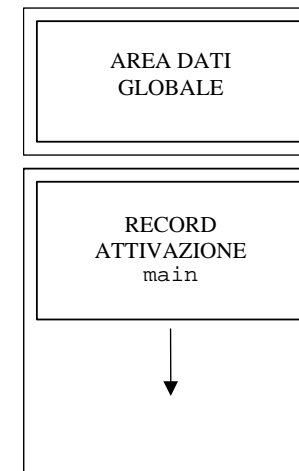
È un esempio di **ricorsione non lineare** (più chiamate ricorsive nel corpo della funzione per determinare il valore restituito dalla funzione).

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)):



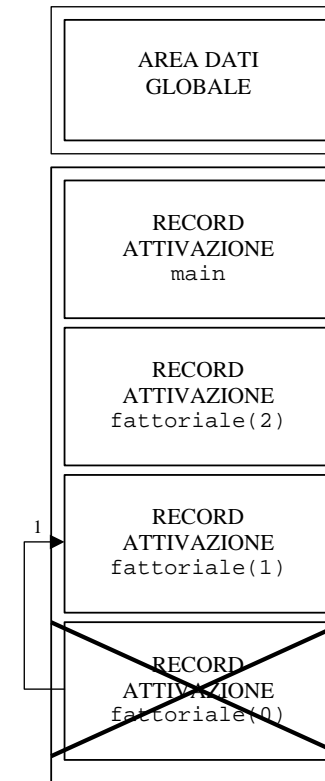
- Dopo la seconda attivazione (fattoriale(1)):



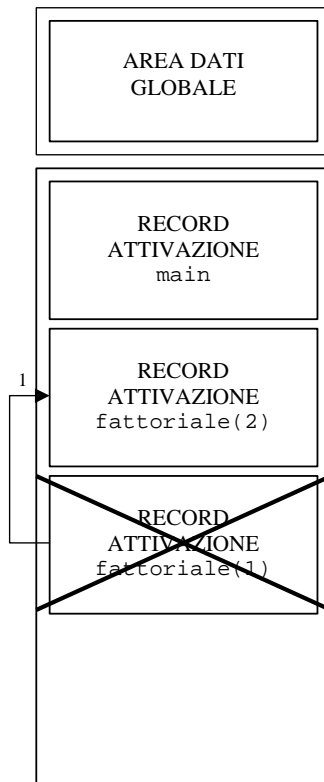
- Dopo la terza attivazione (`fattoriale(0)`):



- 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 è sempre realizzabile mediante **iterazione**.

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 vs. iterazione

Quando conviene utilizzare la ricorsione?

- Soluzioni ricorsive sono spesso più vicine alla definizione matematica di certe funzioni.
- Versioni iterative sono generalmente *più 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** è necessario memorizzare in una struttura dati (vettore) la stringa.

- 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);
}
```

- 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 è l'ultima istruzione del codice di F , si dice che F è *tail-ricorsiva*.

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

Nell'esempio:

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

$f(2, 3) \rightarrow f(1, 4) \rightarrow f(0, 5) \rightarrow \text{return } 5$

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'algoritmo 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'algoritmo 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 sotto-vettore */
}
```

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

Anche la funzione **main** può 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

È 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**) è considerato un argomento, quindi **argc** vale sempre almeno 1.

char **argv

È un 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* (cioè il nome del file eseguibile).

Esempio:

Se l'esecuzione è 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");
}
```

Invocazione

esempio a b c zeta

Stampa

esempio a b c zeta