



- 5° Modulo
 - Funzioni Elementari
 - Passaggio dei Parametri
 - Recursione
 - Ambienti e Visibilità

<http://www.elet.polimi.it/upload/martucci/index.html>

Struttura di un Programma C

Un programma C ha in linea di principio la seguente forma:

- **Direttive per il preprocessore**
- **Definizione di tipi**
- **Prototipi di funzioni**, con dichiarazione dei tipi delle funzioni e dei parametri)
- **Dichiarazione delle Variabili Globali**
- **Dichiarazione Funzioni**, dove ogni dichiarazione di una funzione ha la forma:
Tipo NomeFunzione(Parametri)
{
 Dichiarazione Variabili Locali
 Istruzioni C
}

```
#include <stdio.h>

typedef struct point {
    int x; int y;
} ;

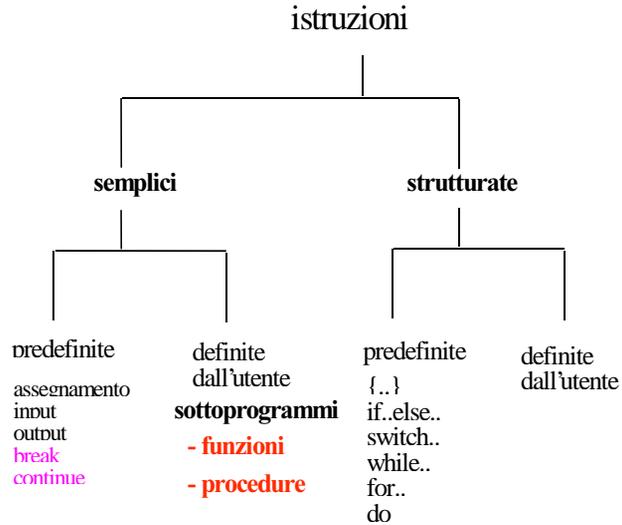
int f1(void);
void f2(int , double );

int sum;

void main( )
{
    int j;
    double g=0.0;
    for(j=0;j<2;j++)
        f2(j,g);
}

void f2(int i, double g)
{
    sum = sum + g*i;
}
```

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 sotto-programma verrà *chiamato*);
- si stabiliscono le **modalità di comunicazione** tra l'unità di programma che usa il sottoprogramma ed il sottoprogramma stesso (definizione dei **parametri formali**).

Utilizzo di funzioni/procedure (*chiamata*):

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

Meccanismo di Chiamata

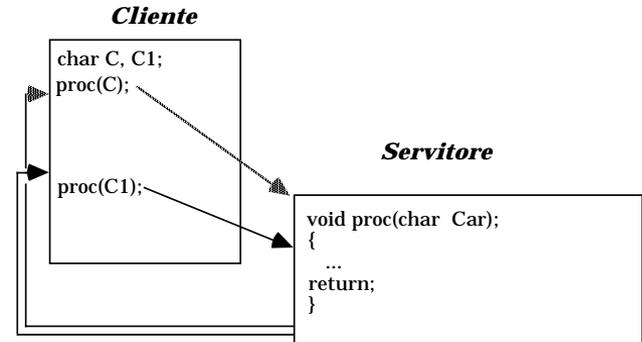
Quando si verifica una chiamata a sottoprogramma, si possono individuare due entità:

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

Quando avviene la chiamata, l'esecuzione dell'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 unita' chiamante ed unita' 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'unita' 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*, cioe' 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 verra' spiegato piu' avanti.



Funzioni e Procedure

Procedure e Funzioni

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.

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

Procedura:

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

Funzione:

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

Ad esempio:

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

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

Funzioni in C

Procedure e funzioni si definiscono seguendo regole sintattiche simili.

Definizione di funzione:

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

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

Struttura dell'intestazione:

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

dove:

• **<tipo-ris>**: e' un identificatore che indica il tipo di risultato restituito (*codominio*). Il tipo restituito puo' essere predefinito o definito dall'utente. Una funzione non puo' restituire valori di tipo:

- **vettore**
- **funzione**

- **<nome>**: e' l'identificatore della funzione
- **<lista-par-formali>** e' la 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>.

Functions in C

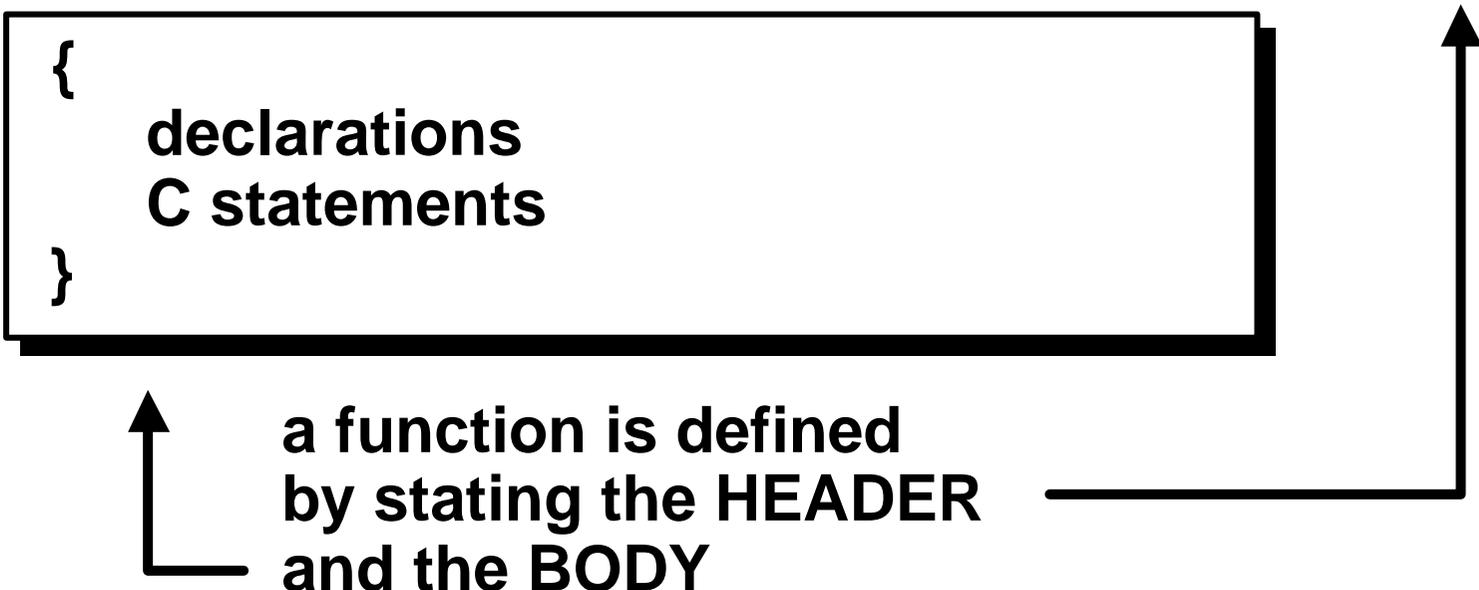
Definitions

Function definition general form:

```
return-type function-name( parameter declarations )
```

```
{  
  declarations  
  C statements  
}
```

**a function is defined
by stating the HEADER
and the BODY**



Function Definitions: The Header

The HEADER includes:

return-type function-name(parameter declarations)

↑
data type
returned by
the function

↑
function name

↑
0 or more parameters within
parenthesis, with a type declaration
for each parameter

Function Definitions: The Body

```
return-type function-name( parameter declarations )
```

```
{  
  declarations  
  C statements  
}
```

} BODY

The **BODY** is simply **C statements**
(including declaration statements)
bounded by braces

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 (cioe' 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);
}
```

Es. funzione che somma due valori di tipo int e restituisce un int:

```
int somma(int a, int b)
{
    int sum;
    sum = a+b;
    return(sum);
}
```

La chiamata della funzione viene fatta così:

```
void main(void)
{
    int A=23; int B=-31; int risultato;
    risultato = somma(A,B);
    printf("somma= %d\n", risultato);
}
```



Es. funzione che non restituisce alcun valore:

```
void somma(int a, int b)
{
    int sum;
    sum = a+b;
    printf("somma= %d\n", sum);
    /* non serve la return */
}
```

La chiamata della funzione viene fatta così:

```
void main(void)
{
    int A=23; int B=-31;
    somma(A,B);
}
```

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\\", 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();
}
```



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

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

Dichiarazione di funzione

Regola Generale:

Prima di utilizzare una funzione 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);
```

Function Prototypes

```
return-type function-name( parameter declarations );
```

- generally look like the function header
- end with a semi-colon (;)
- come before 1st use (call) of function
- tell compiler (and programmer!) what to expect

```
double sqrt(double num);  
void set_date(int, int, int);
```

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) /*intestaz. */
{
    if (a>b) return a;
    else return b;
}
```

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 **leggibilita'**).

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);
}
```



Functions in C

Definition Rules

- functions return max of **one** data type, if any
 - void, if none. **Default if not supplied is `int`, not `void`**
 - MUST have a **return statement** to return a value
 - **return** statement optional if void return data type
 - format: ***return expression;***
 - CONTROL returns to calling routine at return statement or at ending brace of function
- functions may call other functions
 - basis for all C programs
 - recursion: a function may call itself
- parameters
 - are local variables
 - list includes data type declarations; void, if none

Functions in C

Sample C Code

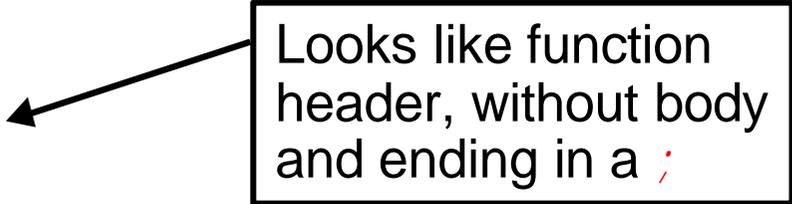
function prototype

```
#include <stdio.h>
main()
{
    int a, b, c;
    int maximum( int, int, int);

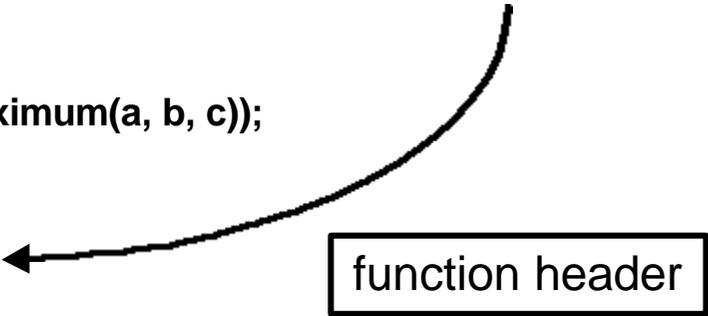
    printf("Enter three integers: ");
    scanf("%d%d%d", a, b, c);
    printf("Max value is %d\n", maximum(a, b, c));
}

int maximum( int y, int x, int z)
{
    if( x > y && y > z)
        return x;
    if( y > z) return y ; else return z;
}
```

Looks like function header, without body and ending in a ;



function header



Functions in C

Sample C Code

decl vs no decl

```
main()
```

```
{
```

```
    double x, y;
```

```
    x = 4.0;
```

```
    y = sqrt(x);
```

```
    printf("f\n", y);
```

```
}
```

No declaration

prints **wrong answer** of
16640.000000, because the double
returned is presumed to be an int!

```
main()
```

```
{
```

```
    double x, y;
```

```
    double sqrt(double);
```

```
    x = 4.0;
```

```
    y = sqrt(x);
```

```
    printf("%f\n", y);
```

```
}
```

with declaration

prints correct answer of
2.000000

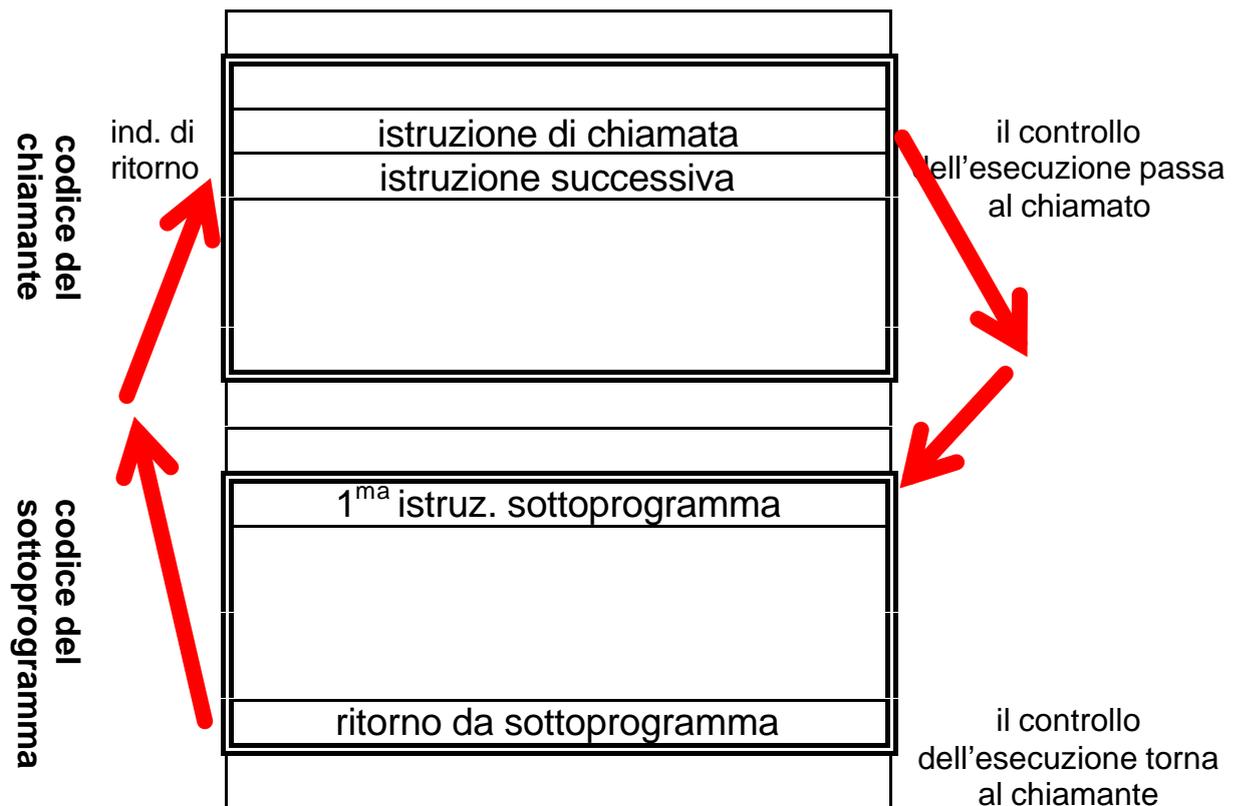
CHE COSA VUOL DIRE CHIAMATA DI UN SOTTOPROGRAMMA

La chiamata di un sottoprogramma implica:

- all'atto della chiamata, la cessione del controllo dell'esecuzione dal chiamante al chiamato
- l'esecuzione del codice del chiamato
- al termine dell'esecuzione il ritorno del controllo al chiamante, all'istruzione o operazione successiva a quella di chiamata

E' necessario il «salvataggio» dell'indirizzo di ritorno del chiamante

Rappresentazione in memoria



Calling Functions

- invoked by stating function name and argument list, or null list, in ()
- arguments can be any expression, including function calls, separated by commas
- value returned by function can be assigned to a variable or used directly

```
initialize();
```

```
oldest = max(age_a, age_b)
```

Variable Function Arguments

Using Functions

Alone: **Beep(4);**

Passing return value:

Younger=Min(HusbandAge,WifeAge);

In an expression:

if (IsValid(cust_num)) . . .

**total_fee = determine_base_fee(consultant)
+ calc_variable_fee(consultant, hrs);**

Calling Functions

- arguments(s) must have **same data type** as parameters (from definition); generally must have same number
- argument number, type, and return type are checked against prototype
- all arguments evaluated before function is called, but **order is NOT guaranteed**

```
total_fee (determine_base_fee(consultant),  
           calc_variable_fee(consultant, hrs));
```

if `calc_variable_fee` must be executed after `determine_base_fee`, then this will not work!

'Value' vs 'Reference'

- argument values are passed 'call by value'

HusbandAge = 40;

WifeAge = 30;

Younger=Min(HusbandAge,WifeAge);

a copy of the respective values, 40 and 30,
are passed to the function Min()

Min() cannot change the values of the
variables HusbandAge and WifeAge

- *'call by reference' is simulated*
the address of the variable is passed, not its value

FUNZIONI IN C

Astrazioni di valore:

- ricevono valori in ingresso (dal chiamante)
- elaborano
- producono un risultato il cui valore è associato all'identificatore di funzione e restituito al chiamante (**effetto** del sottoprogramma)

```
tipo nome_funz(lista parametri formali)
{
    parte dichiarativa locale
    parte esecutiva

    return <espressione>;
    .....
}
```

Istruzione C: **return <espressione>;**
dove <espressione> deve essere dello stesso tipo di nome_funz.

L'esecuzione dell'istruzione return implica:

- la valutazione di <espressione>
- l'assegnamento del valore di <espressione> al nome della funzione
- il ritorno al programma chiamante (l'esecuzione della funzione termina)

In una funzione:

- deve esserci almeno una istruzione **return**, altrimenti l'esecuzione termina quando s'incontra il simbolo }, senza restituire alcun valore significativo
- possono esserci più istruzioni return (dipende dal flusso di controllo)

Esempio: Calcolo del coefficiente binomiale

```

#include <stdio.h>
main ()
{
    char caratt,tappo;
    int n,k,coeff;
    int fattoriale(int);          prototipo

do
    {
        printf("Calcolo del coefficiente binomiale n su k\n");
        printf("inserisci il valore di n (>=0)\n");
        scanf("%d", &n);
        printf("inserisci il valore di k (>=0 e <=n)\n");
        scanf("%d", &k);          ATT! non si fanno i test sull'input!!!

        coeff=fattoriale(n)/(fattoriale(k)*fattoriale(n-k));

        printf ("Il valore del coefficiente e' %d \n",coeff);
        printf ("Vuoi continuare? S/N  ");
        scanf("%c",&caratt);
        scanf("%c", &tappo);
    }
    while (caratt=='S');
}

```

```
/* calcolo del fattoriale: versione iterativa */
int fattoriale (int a)
{
    int i, fatt;

    fatt=1;
    if(a!=0)
    for(i=1;i<=a;i=i+1)
        fatt=fatt*i;
    return fatt;
}
```

PROCEDURE E AMBIENTE GLOBALE

- ambiente globale: visibile sia al chiamante che al chiamato
- l'esecuzione della procedura modifica tale ambiente
- i parametri formali rappresentano dati specifici relativi al servizio offerto dalla procedura

Esempio

- array di struct
- Ordina ()
- Inserisci (....)
- Elimina (....)

Esempio

- matrici e vettori
- Somma
- Prodotto

I problemi delle variabili globali

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:

Area dati di C

pa

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**

Area dati di C

pa

Area dati di P

pf

Esecuzione di P:

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

Alla fine dell'esecuzione di P:

Area dati di C

pa

α

Area dati di P

pf

α'

- 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 e' necessariamente una variabile, ma puo' essere, in generale, una **espressione**.

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

Ad esempio:

```
#include <stdio.h>

void P(int xx);

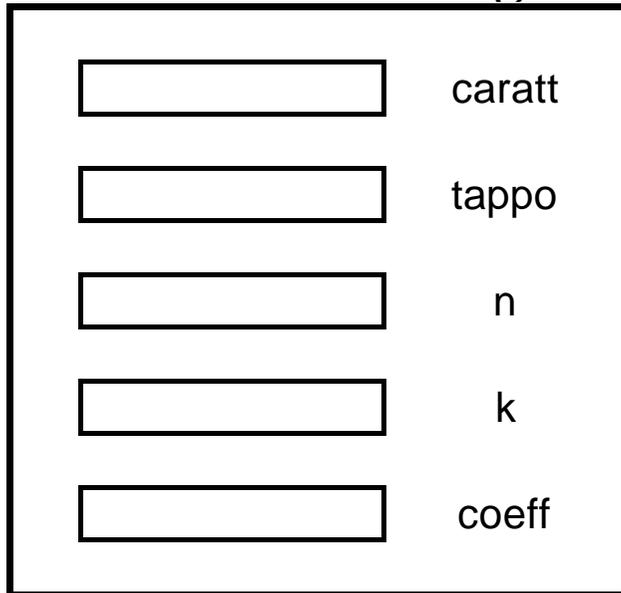
main()
{ int pa=10;

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

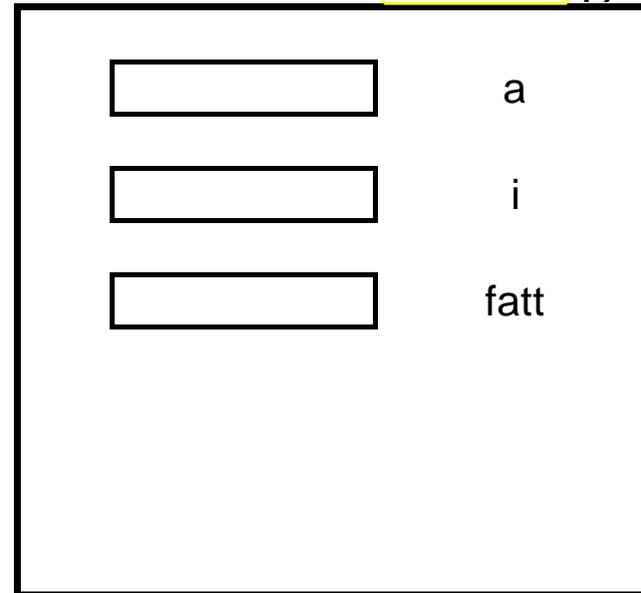
void P(int pf)
{
  pf=100;
  printf("valore finale di pf: %d\n",
        pf);
  return;
}
```

ESECUZIONE E AMBIENTI LOCALI

ambiente locale di *main* ()



ambiente locale di **fattoriale** ()



param formale

n=4 k=3 coeff=fattoriale(4)/fattoriale(3)*fattoriale(1)



ESECUZIONE E AMBIENTI LOCALI

ambiente locale: chiamata di **fattoriale (4)**

4	a	passaggio del valore alla chiamata
4	i	al termine del ciclo for
24	fatt	al termine del ciclo for

al ritorno dalla prima chiamata

$$\text{coeff} = 24 / \text{fattoriale}(3) * \text{fattoriale}(1)$$

ambiente locale: chiamata di **fattoriale (3)**

3	a	passaggio del valore alla chiamata
3	i	al termine del ciclo for
6	fatt	al termine del ciclo for

al ritorno dalla seconda chiamata

$$\text{coeff} = 24 / 6 * \text{fattoriale}(1)$$

ambiente locale: chiamata di **fattoriale (1)**

1	a	passaggio del valore alla chiamata
1	i	al termine del ciclo for
1	fatt	al termine del ciclo for

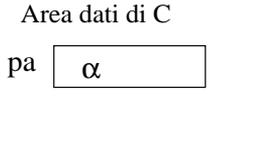
al ritorno dalla terza chiamata

$$\text{coeff} = 24 / 6 * 1$$

Legame per indirizzo

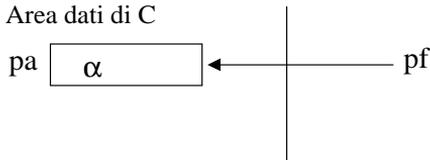
Se il legame dei parametri avviene per indirizzo:

1. Prima della chiamata:



2. Al momento della chiamata:

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

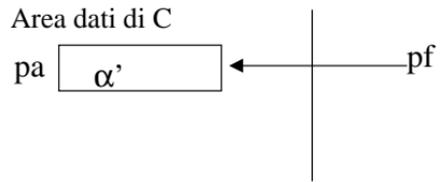


➔ pf è un *alias* di pa.

Esecuzione di P:

Il parametro formale **pf** viene trattato come una **variabile locale** al sottoprogramma P: puo' essere modificato mediante assegnamento, etc.. In generale, al termine della chiamata, pf (e quindi pa) potra' 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) puo' 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 (cioe', un oggetto dotato di un indirizzo).

In C, il legame per indirizzo non e' disponibile.

**PROCEDURE E AMBIENTE GLOBALE: ESEMPIO**

```
#include <stdio.h>
#define N 10
#define TRUE 1
#define FALSE 0

typedef struct {
    int matricola;
    int votiesa[29];
} STUDENTE;

STUDENTE iscritti[N]; /* variabile globale*/

void Inserisci_in_elenco_ordinato(int,int);
int Cerca_posizione (int, int);
void Sposta (int, int);

main ()
{
    int nuova_mat, n_iscr;
    char proseguire,continua,tappo;
```

```

proseguire=TRUE;
n_iscr =0;

while(proseguire)
{
    printf("inserire matricola (4 cifre)\n");
    scanf ("%d",&nuova_mat);    si considera almeno una nuova matr

    Inserisci_in_elenco_ordinato(n_iscr,nuova_mat);
    n_iscr ++;

    printf("Vuoi inserire un nuovo iscritto S/N?\n");
    scanf ("%c",&continua);
    scanf ("%c",&tappo);
    if (continua!='S')
        proseguire=FALSE;
}

printf("Si sono iscritti %d studenti", n_iscr);
}

```

/* Procedura: riceve come parametri il numero attuale di iscritti (elementi già memorizzati nell'array) e la matricola da inserire in ordine.

Algoritmo: cerca nell'array la posizione in cui inserire il nuovo elemento, se necessario sposta gli elementi successivi di una posizione nell'array e inserisce la nuova matricola */

```
void Inserisci_in_elenco_ordinato (int n_stud, int mat)
{
    int posizione;

    posizione = Cerca_posizione (n_stud, mat);

    if (posizione == n_stud)

        iscritti[posizione].matricola = mat;

    else
    {
        Sposta (posizione, n_stud);
        iscritti[posizione].matricola = mat;
    }
}
```

```

/* Procedura: riceve come parametri il valore iniziale e finale
dell'indice degli elementi da spostare */

void Sposta (int pos, int n_stud)

{ int j;

  for (j=n_stud-1; j >=pos; j--)
    iscritti[j+1].matricola = iscritti[j].matricola;
}

```

/* Funzione: riceve come parametri il numero attuale di iscritti (elementi già memorizzati nell'array) e la matricola da inserire in ordine. Restituisce come valore la posizione in cui inserire la nuova matricola.

Algoritmo: l'array è ordinato e quindi viene usato l'algoritmo di ricerca binaria che dimezza, ad ogni passo, la dimensione del sotto-array in cui effettuare la ricerca.*/

```
int Cerca_posizione (int n_stud, int mat)
{
    int inizio, fine, mediano;

    inizio=0;
    fine=n_stud -1;
    mediano=(inizio + fine)/2;

    while (inizio <= fine)
    {
        if (mat < iscritti[mediano].matricola)
            fine = mediano -1;
        else
            inizio = mediano + 1;

        mediano = (inizio + fine)/2;
    } /* fine while di ricerca */
    return (mediano+1);
}
```

AMBIENTE GLOBALE DEL PROGRAMMA

- insieme di identificatori (tipi, costanti, variabili) definiti nella parte dichiarativa globale
- **regole di visibilità**: visibili a tutte le funzioni del programma

AMBIENTE LOCALE DI UNA FUNZIONE

- insieme di identificatori (tipi, costanti, variabili) definiti nella parte dichiarativa locale e degli identificatori definiti nella testata (parametri formali)
- **regole di visibilità**: visibili alla funzione e ai blocchi in essa contenuti

AMBIENTE DI BLOCCO

- insieme di identificatori (tipi, costanti, variabili) definiti nella parte dichiarativa locale di blocco
- **regole di visibilità**: visibili al blocco e ai blocchi contenuti

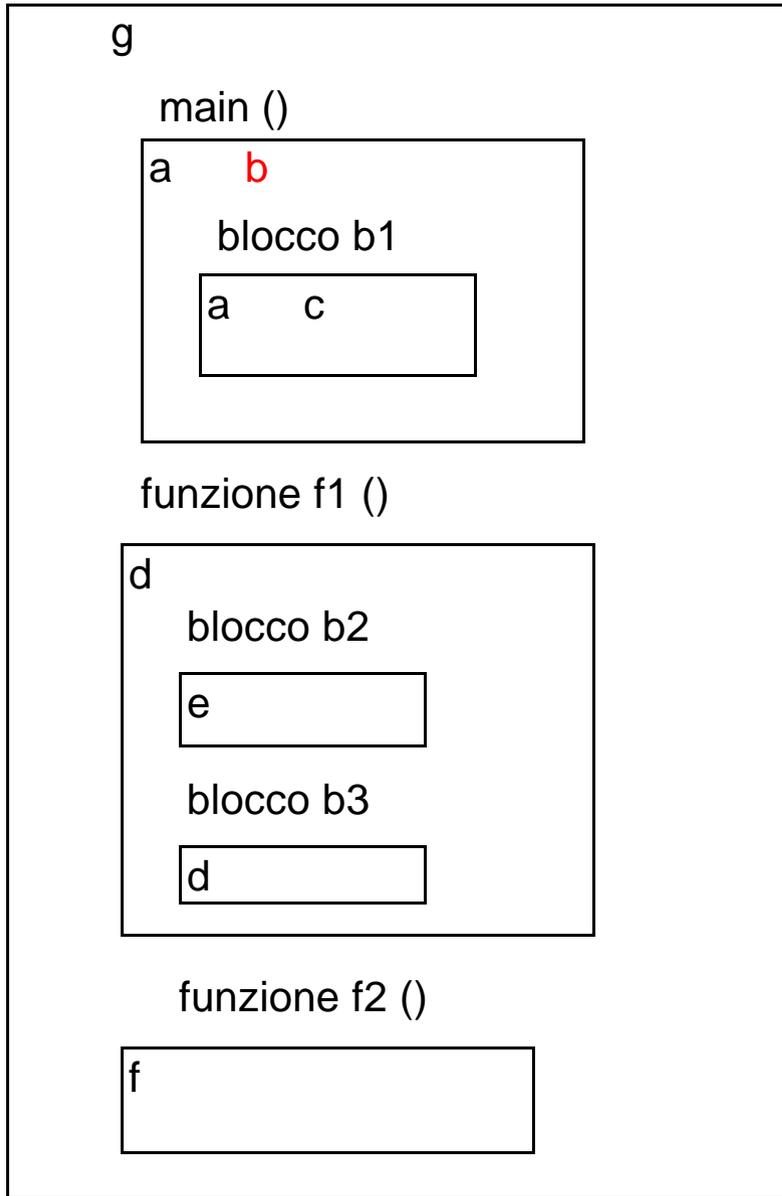
IDENTIFICATORE DI FUNZIONE:

- visibile a tutte le funzioni e a se stessa

OMONIMIA DI IDENTIFICATORI IN AMBIENTI DIVERSI

- è visibile quello dell'ambiente più «vicino»

ESEMPIO DI AMBIENTI E REGOLE DI VISIBILITÀ



```
main ()  
  g  
  a, b  
  f1, f2  
  
blocco b1  
  g  
  b (di main)  
  a, c  
  f1, f2  
  
funzione f1()  
  g  
  d  
  f1, f2  
  
blocco b2  
  g  
  d (di f1)  
  e  
  f1, f2
```

```
blocco b3  
  g  
  d (di b3)  
  f1, f2
```

```
funzione f2()  
  g  
  f  
  f1, f2
```

Functions in C

Sample C Code

nested blocks

This is legal, but
can be confusing!

```
main()
{
  exists from here... int x = 1, y = 2;
  printf("x = %d, y = %d\n", x, y);
  {
    'exist' only within here int x = 3, y = 4;
    printf("x = %d, y = %d\n", x, y);
  }
  to here printf("x = %d, y = %d\n", x, y);
}
```

This 'x' and 'y' ...

are not the same variables ...

as this 'x' and 'y' !

but these are!

Output is:

x = 1, y = 2

x = 3, y = 4

x = 1, y = 2

AMBIENTE DI ESECUZIONE DI UNA FUNZIONE

Sequenza di chiamate

main → f1 → f2
main ← f1 ←

- L'**ambiente di esecuzione** di una funzione viene creato al momento della chiamata e «rilasciato» solo quando la funzione termina.
- In una **sequenza di chiamate** l'ultimo chiamato è il primo a terminare.

RICORSIONE:

(la soluzione ad un problema si dice ricorsiva (l'algoritmo) è ricorsivo) se fa uso di se stessa.

La soluzione ad un problema (l'algoritmo) può ammettere o non ammettere formulazione ricorsiva.

IN PROGRAMMAZIONE:

dato un sottoprogramma P, la chiamata di P quando P è in esecuzione (P chiama se stesso) è ricorsione.

Per distinguere le varie chiamate si parla di **attivazione** del sottoprogramma

main → P' → P'' → P'''
main ← P' ← P'' ←

La chiamata di P può anche essere indiretta.

main → P' → Q → P''
main ← P' ← Q ←

AMBIENTE DI ESECUZIONE E RICORSIONE

Un linguaggio di programmazione (e quindi l'ambiente di programmazione) può supportare o meno la ricorsione. **Il C ammette la ricorsione.**

Perché la ricorsione sia possibile:

- l'ambiente di esecuzione deve essere associato all'attivazione di P (nasce un ambiente per ogni attivazione e tale ambiente viene rilasciato al termine dell'attivazione considerata)

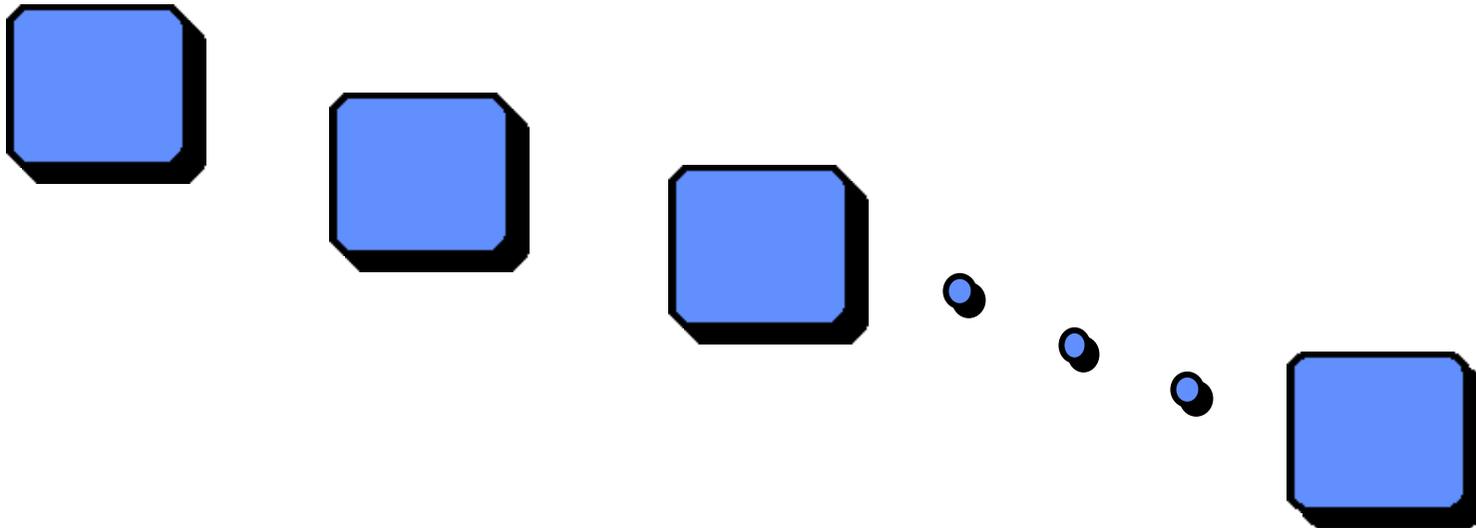
E' necessaria una opportuna gestione della memoria di lavoro allocata a contenere l'ambiente di esecuzione di un sottoprogramma: gestione a pila (stack) degli ambienti di esecuzione.

RECORD DI ATTIVAZIONE: è costituito da un'area di memoria adeguata a contenere

- l'ambiente locale della funzione (dichiarazioni locali e parametri formali)
- l'indirizzo di ritorno al chiamante
- informazioni per gestire la memoria a stack

Recursion

- A Recursive function is one that calls itself
- Sometimes recursion is most natural solution
- Higher overhead with multiple function calls
- Need to be able to recognize recursion



Recursion

Problem: compute a factorial (of positive integers)

Analysis: What is a factorial?

$$n! = n * (n-1) * (n-2) * \dots * 1$$

with 1! defined to be 1

and 0! defined to be 1

base case: when $n = 1$

recursion step: $n * (\text{factorial of } n - 1)$

Functions in C

Sample C Code

recursive

```
#include <stdio.h>
long factorial(long);

main()
{
    int j;

    for ( j = 0; j <= 10; j++)
        printf("%2d! = %ld\n", j, factorial(j));
}

long factorial(long number)
{
    if( number <= 1)
        return 1;
    else
        return( number * factorial(number - 1));
}
```

Base case

recursion step

Recursion

- Funzione recursiva: chiama sé stessa.

- Appropriate for iterative processes where each step mimics another, e.g., tree searches

- E.g., factorial calculation: $n! = n(n - 1)!$

```
// why a long?  
unsigned long factorial (unsigned int number) {  
    if (number > 1)  
        return (number * factorial (number - 1));  
    return 1ul;    // for 0 or 1  
}
```

- Matches the problem (i.e., the mathematics) nicely

Recursive vs. Iterative Functions

```
I
// iterative version
unsigned long factorial_iter (unsigned int number) {
    for (unsigned long product = 1ul; number > 1; number --)
        product *= number;
    return product;
}
```

- Recursive vs. iterative
 - * smaller
 - * less complex
 - * but, slower—due to additional function invocations

Unless otherwise required, do what is most natural.

RECORD DI ATTIVAZIONE: FUNZIONAMENTO

- ad ogni attivazione viene allocato un record di attivazione
- al termine dell'attivazione il record viene rilasciato (l'area di memoria è riutilizzabile)
- la dimensione del record di attivazione di una funzione è nota e fissa e viene determinata in fase di compilazione
- non è noto il numero di chiamate (di attivazioni) della funzione: tale numero dipende dall'esecuzione del programma
- i record vengono allocati in memoria «a pila» (**stack**): «uno sopra l'altro» e il primo record dello stack è relativo all'ultima funzione attivata e non ancora terminata. Lo stack «cresce» dal basso verso l'alto
- il primo record di attivazione è allocato per la funzione main().

INFORMAZIONI PER GESTIRE LA MEMORIA A STACK

stack pointer: è l'indirizzo della cima della pila in ogni record di attivazione è memorizzato il valore dello stack pointer relativo a quell'attivazione specifica

RAM E STACK OVERFLOW

Una parte della RAM è dedicata a contenere l'area di stack, che ha globalmente delle dimensioni prefissate. Si parla di **stack overflow** quando questa capacità viene superata («troppi» annidamenti di chiamate).

```
int X,Y,Z;
int F (int, int);
```

```
int F(int a, int b)
{ int Loc1, Loc2;
/* bla bla bla */
Loc1 = F(Loc1,Loc2) ;
/* bla bla bla */ };
```

Indir	nome
1000	23 X
1002	48 Y
1004	12 Z
1006	
1008	
1010	
1012	
1014	
1016	
1018	
1020	
1022	
1024	
1026	
1028	
1030	
1032	
1034	
1036	
1038	

X = F(Y,Z);

nel caso di recursione

Loc1 = F(Loc1,Loc2);

Loc1 = F(Loc1,Loc2);

next stack

Indir	nome
1000	X
1002	Y
1004	Z
1006	@X -result - 1000
1008	1°par=48
1010	2°par=12
1012	Loc1
1014	Loc2
1016	@Loc1 -result - 1012
1018	1°par=value of Loc1
1020	2°par=value of Loc2
1022	Loc1bis
1024	Loc2bis
1026	@Loc1bis -result - 1022
1028	1°par=value of Loc1bis
1030	2°par=value of Loc2bis
1032	Loc1ter
1034	Loc2ter
1036	
1038	

Allocazione stack



ESEMPIO: CONVERSIONE BINARIA RICORSIVA

```

#include <stdio.h>
#define TRUE 1
#define FALSE 0
void main ()
{
    int valore;
    char proseguire,continua,tappo;

    void Converti_bin(int);

    proseguire=TRUE;

    while (proseguire)
    {
        printf("inserire il valore intero positivo da convertire \n");
        scanf("%d",&valore);

        Converti_bin(valore);

        printf("\nVuoi convertire un altro valore? (S/N) \n");
        scanf("%c",&continua);
        scanf("%c",&tappo);
        if (continua!='S')
            proseguire=FALSE;
    }
} /* fine main */

```

```
void Converti_bin (int num)
{ int resto;

  resto=num%2;

  if (num >=2)
    Converti_bin (num/2);

  printf("%d", resto);
}
```