U.D. 7

Funzioni

- Concetti generali
- Le Funzioni in C
- Parametri
  - By value
  - By address

Fonti:

Antola Dispense del Corso
A. Ciampolini – UNI_BO – Lucidi del corso
V. Ghini . UNI BO - Lucidi del Corso

Struttura di un Programma C

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

- **Direttive per il preprocessore**

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

- **Definizione di tipi**

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

- **Prototipi di funzioni**, con dichiarazione dei tipi delle funzioni e dei parametri)

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

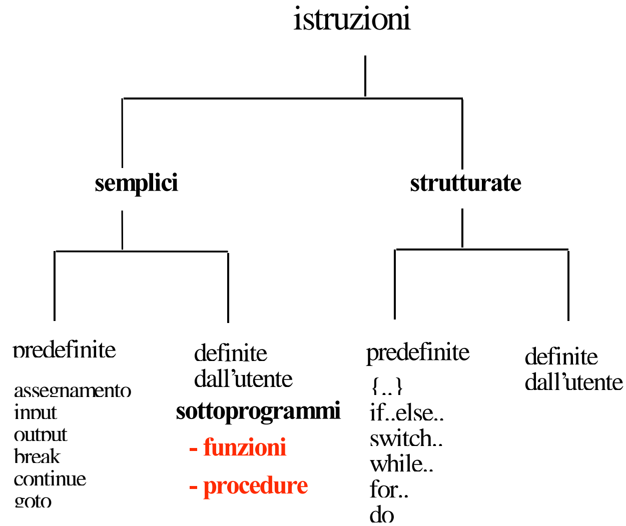
- **Dichiarazione delle Variabili Globali**

```
int   sum;
```

- **Dichiarazione Funzioni**, dove ogni dichiarazione di una funzione ha la forma:
  Tipo NomeFunzione(Parametri)
  {
     Dichiarazione Variabili Locali
     Istruzioni C
  }

```
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 (cioe‘, l’insieme di istruzioni che verra‘ eseguito ogni volta che il sotto-programma verra‘ *chiamato*);
- si stabiliscono le **modalita` di comunicazione** tra l’unita‘ di programma che usa il sottoprogramma ed il sottoprogramma stesso (definizione dei **parametri formali**).

Utilizzo di funzioni/procedure (*chiamata*):

- Per chiamare un sottoprogramma (cioe‘, 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

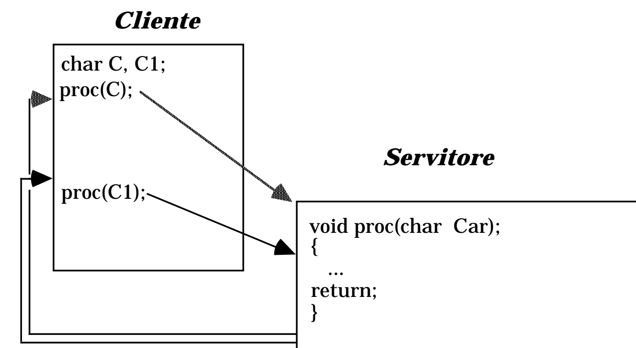
Quando si verifica una chiamata a sottoprogramma, si possono individuare due entita‘:

- l’unita‘ di programma **chiamante**;
- l’unita‘ di programma **chiamata** (il sotto-programma).

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

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

Modello Cliente-Servitore



Parametri

I ***parametri*** costituiscono il mezzo di comunicazione tra unita‘ chiamante ed unita‘ 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'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

Vantaggi:

- **riutilizzo di codice**: sintetizzando in un sottoprogramma un sotto-algoritmo, si ha la possibilita‘ di invocarlo piu‘ volte, sia nell’ambito dello stesso programma, che nell’ambito di programmi diversi (evitando di dover replicare ogni volta lo stesso codice).
- migliore **leggibilita`**: si ha in fatti una maggiore capacita‘ di astrazione
- sviluppo **top-down**: si delega a funzioni/procedure da sviluppare in una fase successiva la soluzione di sottoproblemi.
- testo del programma piu‘ **breve**: minore probabilita‘ di errori, dimensione del codice eseguibile piu‘ 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 puo` comparire un’istruzione.

Funzione:

E’ un’astrazione del concetto di ***operatore***. Si puo` 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 indentificatore 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>.

Definitions

Function definition general form:

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

```
{
    declarations
    C statements
}
```

Function Definitions: The Header

The HEADER includes:

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

- **return-type**: data type returned by the function
- **function-name**: function name
- **parameter declarations**: 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)**, 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 contentue 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);
}
```

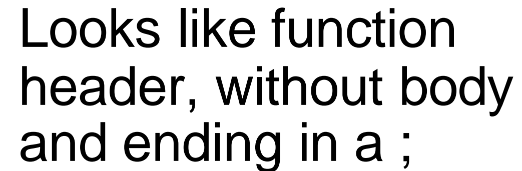
Definition Rules

- 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

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;
}
```



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

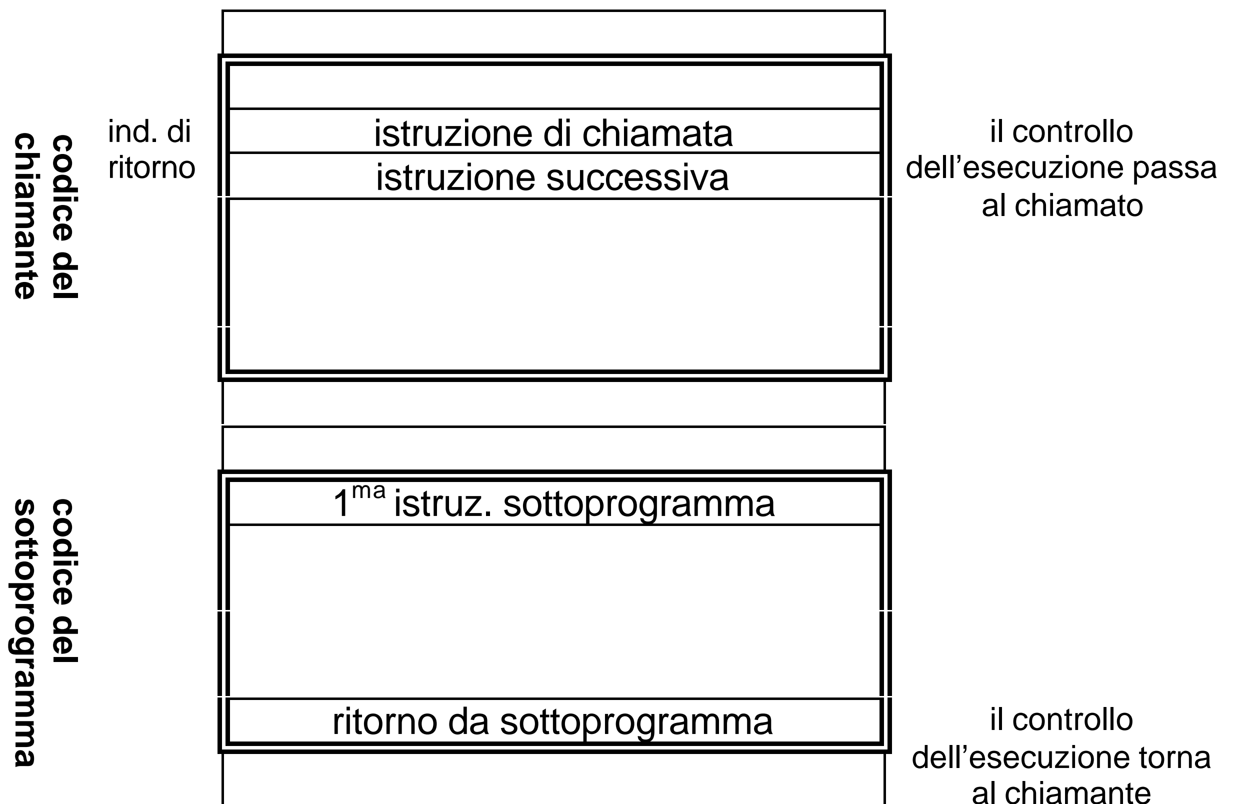
CHE COSA VUOL DIRE CHIAMATA DI UN SOTTOPROGRAMMA

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: oldest — Function: max — Arguments: age_a, age_b

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 IN C

Astrazioni di operazioni:
- forniscono un servizio al chiamante
- possono modificare lo stato di esecuzione (ambiente) del chiamante (**effetto** del sottoprogramma)

```
void nome_proc (lista parametri formali)
{
    parte dichiarativa locale
    parte esecutiva
}  /* qui termina l'esecuzione */
```

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: α

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

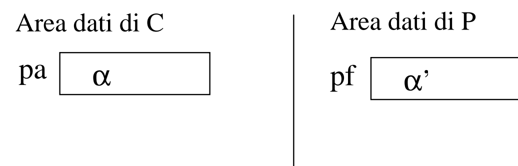
Area dati di C — pa: α

Area dati di P — pf: α

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:



- 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 <u>passati per valore</u> 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**.

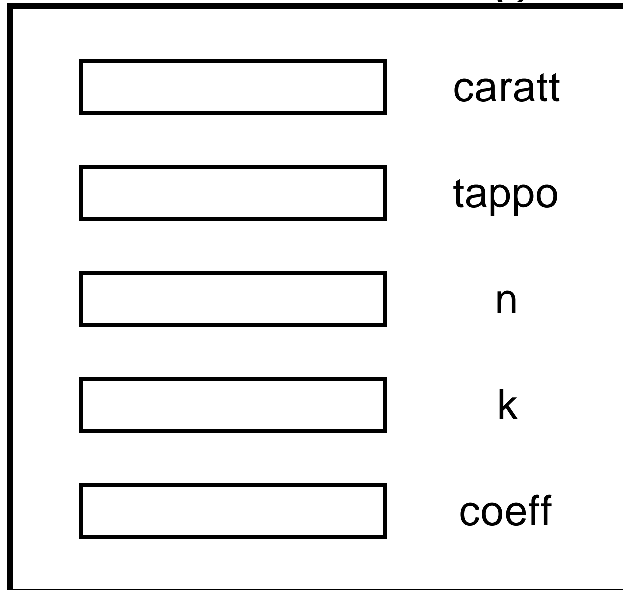
Il legame per valore e` l'unica tecnica di legame disponibile in C.

Ad esempio:

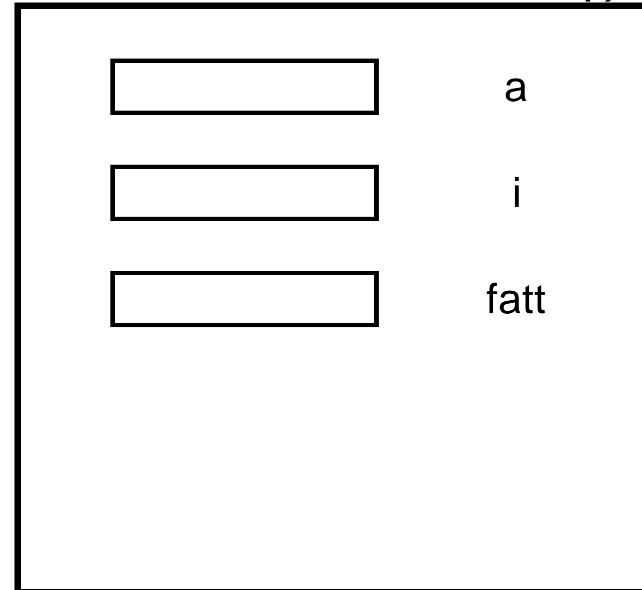
```
#include <stdio.h>

void P(int 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;
}
```

ESECUZIONE E AMBIENTI LOCALI

ambiente locale di ***main ( )***



ambiente locale di ***fattoriale ( )***

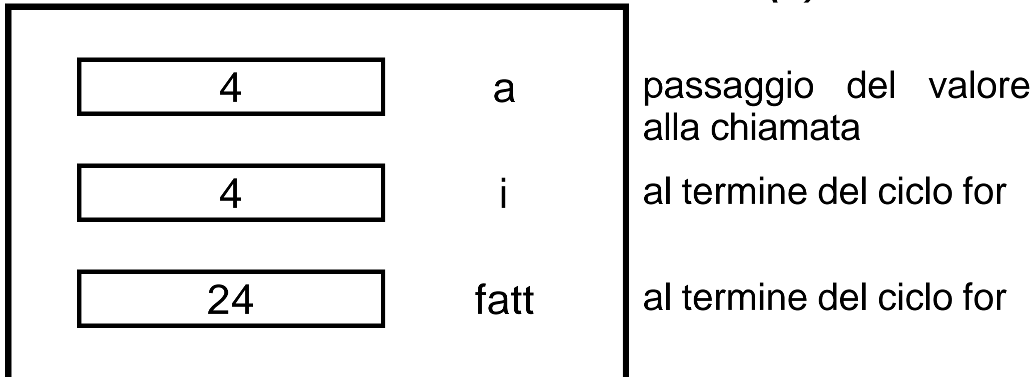


param formale

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

ESECUZIONE E AMBIENTI LOCALI

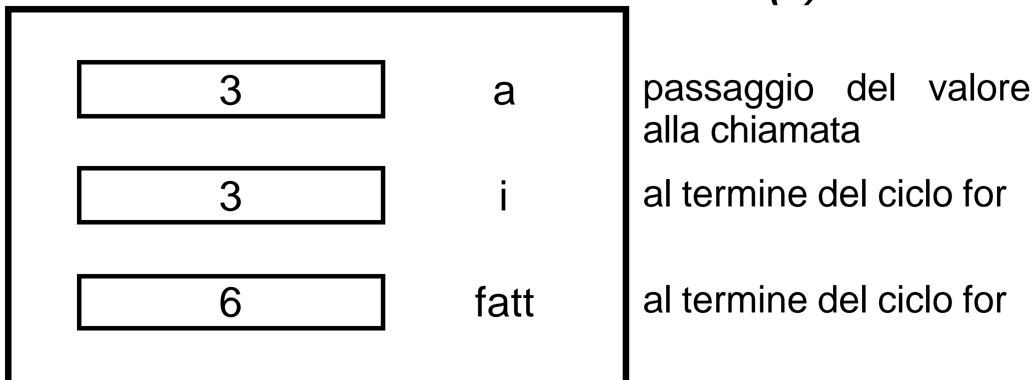
ambiente locale: chiamata di ***fattoriale (4)***



al ritorno dalla prima chiamata

coeff=24 / fattoriale(3)*fattoriale(1)

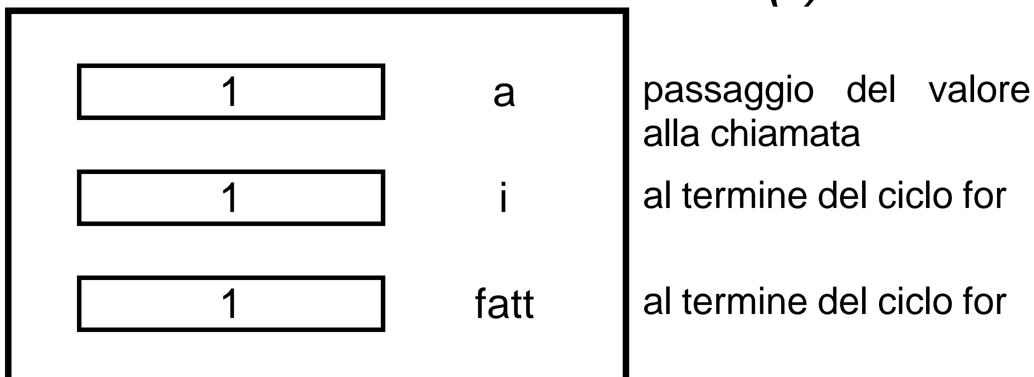
ambiente locale: chiamata di ***fattoriale (3)***



al ritorno dalla seconda chiamata

coeff=24 / 6 *fattoriale(1)

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



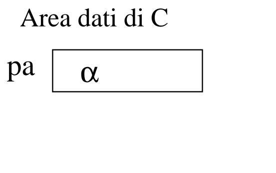
al ritorno dalla terza chiamata

coeff=24 / 6 * 1

Legame per indirizzo

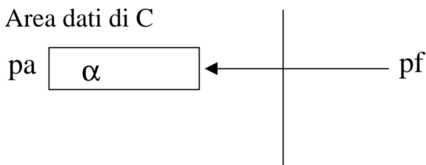
Se il legame dei parametri avviene per indirizzo:

1. Prima della chiamata:



2. Al momento della chiamata:

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

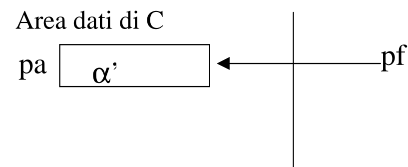


➽ pf e' un ***alias*** di pa.

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.

Esempio:

Utilizziamo la sintassi C per esemplificare il passaggio per indirizzo Il programma che segue e‘ solo a scopo esemplificativo (in C, non c’e‘ 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 ! Progr ERRATO*/
{
  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*** viene stampato:

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

Se il legame e‘ ***per indirizzo*** :

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

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 gia' 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 gia'
memorizzati nell'array) e la matricola da inserire in ordine. Restituisce come
valore la posizione in cui inserire la nuova matricola.
Algoritmo: l'array e' 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);
 }
```

PASSAGGIO DEI PARAMETRI PER VALORE E PER INDIRIZZO: EFFETTI DI UN SOTTOPPROGRAMMA

Passaggio per **valore**:
- all'atto della chiamata il valore del parametro attuale è copiato nelle celle di memoria del corrispondente parametro formale
- sottoprogramma in esecuzione: lavora nel suo ambiente e quindi sui parametri formali. I parametri attuali non si modificano

Passaggio per **indirizzo**:
- all'atto della chiamata, l'indirizzo dei parametri attuali è associato ai parametri formali
- sottoprogramma in esecuzione: lavora nel suo ambiente sui parametri formali. Ogni modifica sul parametro formale è a tutti gli effetti una modifica del corrispondente parametro attuale (referenziano lo stesso indirizzo di memoria). Gli effetti di un sottoprogramma si manifestano nel chiamante con modifiche al suo ambiente locale di esecuzione.

Nel linguaggio C **non esiste un costrutto sintattico** per distinguere tra passaggio dei parametri per valore e passaggio dei parametri per indirizzo.

Il passaggio è **sempre per valore** e per ottenere il passaggio per indirizzo è necessario utilizzare **parametri formali di tipo puntatore.**

L'uso di un parametro formale di tipo indirizzo (puntatore)consente di accedere, e quindi anche di modificare, a variabili appartenenti all'ambiente locale del chiamante. Al termine del sottoprogramma, il chiamante riprende l'esecuzione e il proprio ambiente locale mantiene gli effetti dell'esecuzione del sottoprogramma.

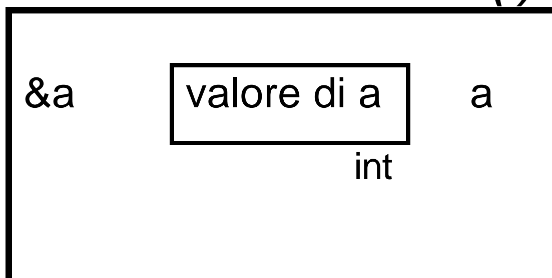
PASSAGGIO PER INDIRIZZO IN C RIEPILOGO

- il parametro formale è di tipo puntatore
- all'interno del sottoprogramma l'accesso al contenuto è fatto tramite dereferenziazione (* oppure →)
- nell'istruzione di chiamata, i parametri attuali devono essere degli indirizzi
- alla chiamata il valore del parametro attuale (un indirizzo) è copiato nel parametro formale (puntatore): è un passaggio per valore
- sottoprogramma in esecuzione: con il dereferencing del puntatore, si accede alla stessa zona di memoria allocata al parametro attuale

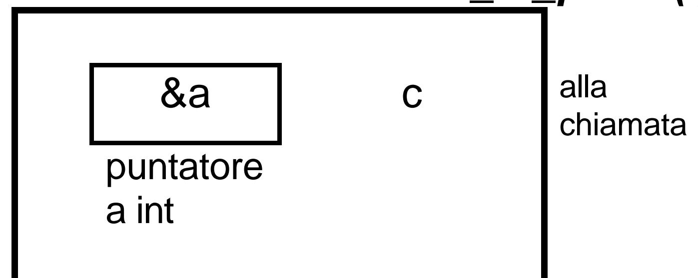
PASSAGGIO PER INDIRIZZO E AMBIENTI DI ESECUZIONE

```
#include <stdio.h>

main ()
{
 int a;
 void funz_di_prova(int *); /*prototipo*/

 printf("inserire il valore di a \n");
 scanf("%d",&a);

 funz_di_prova(&a);
 printf("il valore di a dopo la chiamata e'= %d
\n", a);

}

void funz_di_prova (int *c)
{
 int d;

 d= -27;
 *c=*c + d;
 printf("il valore di *c nella funzione e' = %d
\n", *c);
 }
```

ambiente locale di ***main ( )***



ambiente locale di ***funz_di_prova( )***



alla chiamata

Arguments to Functions

- **C passes arguments to functions by value**

  passes copies of value of the argument (call by value)

- **No direct way for called function to change value of argument**

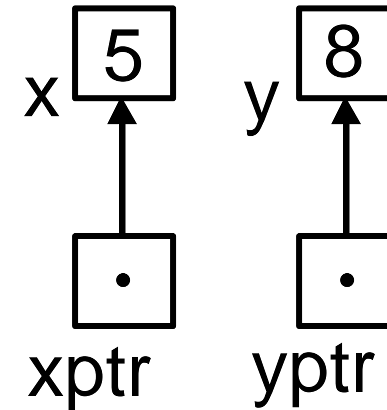
  so we “simulate” call by reference by passing address of variable; that is, a pointer!

```
swap( a, b );      /* function call */
      :
void swap( int x, int y)  /* won't work */
{
      int temp;
      temp = x;
      x = y;
      y = temp;
}
```

Arguments to Functions

- **previous code only swapped copies of arguments!**
- **To “reach” actual arguments, address is needed.**

```
swap( &x, &y );
      :
void swap( int *xptr, int *yptr )
{
      int temp;

      temp = *xptr;
      *xptr = *yptr;
      *yptr = temp;
}
```



Modalità di passaggio degli Argomenti delle Funzioni: Strutture

I dati di tipo **struct** possono venire passati alle funzioni sia per valore sia per puntatore, esplicitamente.

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

/* copia i valori di p1 nella struttura puntata da pp2 */
void copia_e_modifica_struct( struct point p1, struct point *pp2  )
{
        pp2-> x  = p1.x  + 10;              /* modifiche permanenti */
        pp2-> y  = p1.y   + 10;
}

void main(void)
{
        struct point p1 = {   14 , 27 };
        struct point p2;
        copia_e_modifica_struct( p1, & p2  );
}
```

OSS: Se la struttura da passare è molto grande, è bene passarla per puntatore, per non sovraccaricare lo stack, soprattutto quando le chiamate sono molto annidate.

With Functions

- **By passing members**

```
myfunct1( struc1.member1 );
myfunct1( aptr->member1 );
myfunct1( (*aptr).member1 );
```

- **By passing structure**

```
myfunct2( struc1 );
```

- **Passed call by value - cannot modify**
- **For call by reference, use address operator**

```
myfunct3( &struc1.member1 );
myfunct4( &struc1 );
```

Modalità di passaggio degli Argomenti delle Funzioni: Array

I dati di tipo array (vettori, matrici, array di dimensioni maggiori) char, int, long float, double e puntatori), che vengono passati come argomenti ad una funzione, **in C vengono passati per indirizzo, nel senso che "passando l'array per nome si passa l'indirizzo in cui comincia l'array".**

----- Vettori -----

Ciò significa che **quando, all'atto della chiamata, passiamo ad una funzione il nome di un vettore, passiamo l'indirizzo del primo elemento del vettore**, e non tutti i dati del vettore (i 100 interi dell' esempio)

```
void main(void)
{
        int vet[100];
        modifica_vet (vet  , 100 );
}
```

di conseguenza la definizione della funzione dovrà contenere come argomento formale il puntatore al tipo di dati del vettore (un puntatore ad int nell'esempio)

```
void modifica_vet( int *v , int size  )
{
        v[0] = 137;     /* modifica conservata fuori dalla funzione */
        v = NULL;       /* annullo l'indirizzo in v, ma questa modifica
                           NON viene mantenuta fuori dalla funzione */
}
```

In tal modo se il vettore passato come argomento viene modificato dalla funzione, cioè **se la funzione modifica il contenuto degli elementi del vettore, queste modifiche sono permanenti, cioè restano nel vettore anche dopo che la funzione è terminata**.

----- Vettori , notazione alternativa ma equivalente-----

Quando l'argomento passato è un vettore, nella definizione della funzione l'argomento può essere indicato o come un puntatore oppure in un modo alternativo, come segue:

```
void modifica_vet( int v[] , int size )
{
    v[0] = 137;     /* modifica conservata fuori dalla funzione */
    v = NULL;       /* annullo l'indirizzo in v, ma questa modifica
                       NON viene mantenuta fuori dalla funzione */
}
```

Questa notazione **int v[]** vuole indicare che v è un vettore di interi di dimensione sconosciuta (non si sa di quanti interi è composto il vettore).

Comunque le due notazioni int *v e int v[] sono assolutamente equivalenti, entrambi i parametri vengono trattati come puntatori.

----- Matrici -----

Analogamente ai vettori, **quando, all'atto della chiamata, passiamo ad una funzione il nome di una matrice, passiamo l'indirizzo del primo elemento della matrice, quindi le modifiche sui dati della matrice vengono conservate dopo l'uscita dalla funzione.**

```
void main(void)
{   int mat[10][20];
    modifica_mat ( mat );
}
void modifica_mat( int m[][20] )
{   m[1][0] = 137;      /* modifica conservata fuori dalla funzione */
    m = NULL;           /* questa modifica  NON viene mantenuta  */
}
```

Analogamente ai vettori, la definizione della funzione dovrà contenere un puntatore ad array di 20 (= num. colonne) interi.

Notare: si passa la dimensione delle righe, per calcolare l'indice r*NC+c

Passing to Function

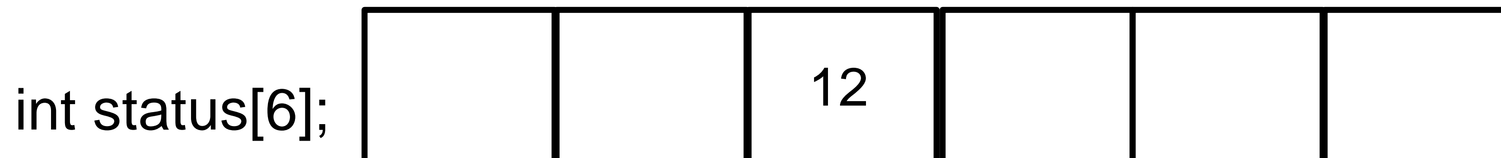
- **Passing array elements**

  call by value

  argument is array name with [ ] and index

  parameter is normal data type

int status[6];



status

```
print_sts(status[2]);
```

function call

```
void print_sts(int passed_status)
{
```

called function header

passed_status gets value of 12 copied to it

PASSAGGIO DEI PARAMETRI: GLI ARRAY completi!!!

- un parametro formale array è visto come un puntatore fisso
- nell'istruzione di chiamata il parametro attuale è il nome dell'array (quindi un indirizzo)
- all'atto della chiamata il valore del nome dell'array è copiato nel nome del parametro formale array e quindi entrambi referenziano la stessa cella di memoria (**il passaggio di array è sempre per indirizzo**)

CHIAMATA E AMBIENTE

Nel chiamante

```
int         v1[N];
void        Ordina(int vettore[N]);          PROTOTIPO

Ordina (v1);                                 CHIAMATA
```

In Ordina

```
void        Ordina(int vettore[N])           DEFINIZIONE
...................

.......vettore[i]....
```

Le modifiche non volute e la ricopiatura in variabile locale

Passing to Function

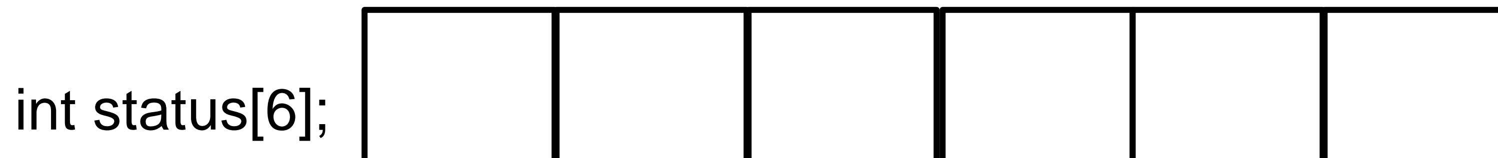
- **Passing arrays**

  call by reference

  argument is array name with no [ ]

  parameter has empty 1st [ ]

int status[6];



status

printf_sts(status);

array name is passed - an address!

•
•
•

void print_sts(int passed_array[ ])

parameter is an array of same type but called function does not know size; sizeof() will not help! (should pass as an additional parameter)

DIMENSIONI DI UN ARRAY E CALCOLO DEGLI INDIRIZZI

Accesso agli elementi di un array passato come parametro in una funzione:

per consentire il **corretto calcolo degli indirizzi** degli elementi è necessario specificare nella testata della funzione tutte le dimensioni dell'array tranne al più la prima (quella più a sinistra).

«EQUIVALENZA» TRA ARRAY E PUNTATORI: ARRAY MONODIMENSIONALI E PUNTATORI

accesso agli elementi

```
*(v+i)  v[i]
```

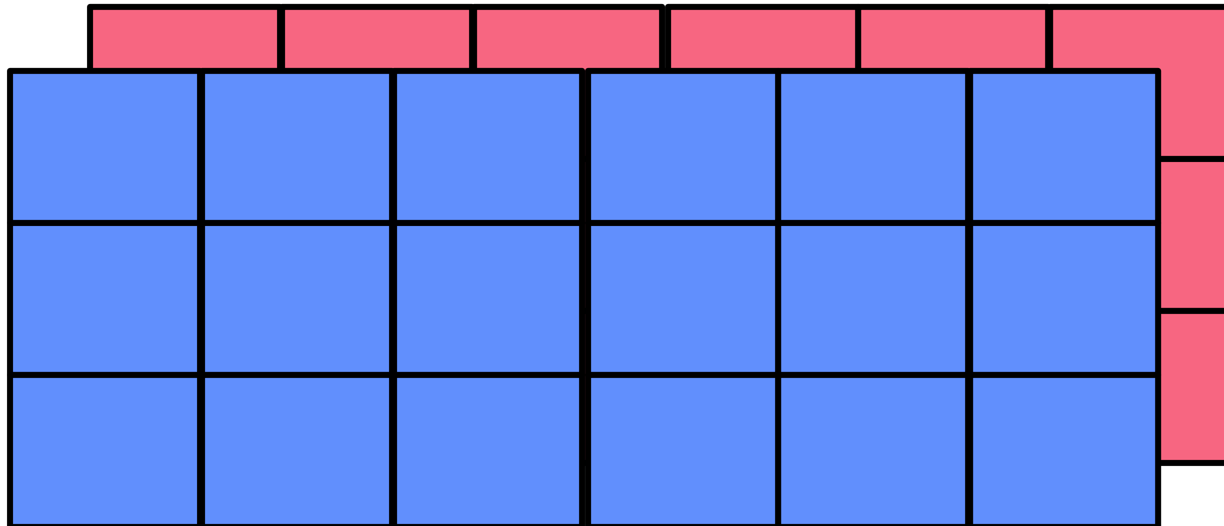
Passing Multidimensional Arrays

- **All but 1st subscript must be present**

function call -> init_array(rooms);

function header -> void init_array(int array[ ][3][6]

Highest dimension always unspecified because we may pass arrays of different sizes. Other dimensions just say how array will be manipulated.



really it's an array of arrays all in one array !

Tipi di dato restituiti dalle Funzioni

Le funzioni possono restituire:

- dati di tipo semplice, come char, int, long, float, double, puntatori a void, o puntatori a qualche tipo di dato,
- ma anche strutture (struct) o puntatori a struct.

All'atto della chiamata, il valore restituito da una funzione:

```
double somma( double f, double g);
```

- può essere utilizzato come espressione booleana,
```
if ( somma(a,b) > 100.3 ) ......
```
- può essere utilizzato come membro di destra in un'istruzione di assegnamento,
```
f = somma(a,b);
```
- oppure può non essere considerato affatto.
```
somma(a,b);
```

Vediamo un esempio di restituzione di una struct.

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

struct point crea_point( int x, int y  )
{
        struct point p;
        p.x = x;
        p.y = y;
        return p ;
}
void main(void)
{
        struct point p1;
        int x=21 , y = -10987;
        p1 = crea_point( x, y );
        printf ( "p1.x=%d p1.y=%d \n" , p1.x, p1.y );
}
```

Considerazioni sui Puntatori restituiti dalle Funzioni

Una funzione può restituire un puntatore ad un qualche tipo di dato, ma la correttezza nell'uso di questo puntatore si può fare, dipende da come lo spazio in memoria è allocato. Cioè:

esempio corretto: p è allocato dinamicamente quindi, anche se p è una variabile locale, lo spazio allocato sopravvive alla terminazione della funzione

```
int *alloca_vettore_1(  int size )
{
	int *p;
	p = malloc(size*sizeof(int));
	return p ;
}
```

```
void main(void)
{
	int *v;
	v = alloca_vettore ( 10 ) ;
	...  usa v ....
}
```

---

esempio sbagliato: p è una variabile locale, lo spazio allocato non sopravvive alla terminazione della funzione

```
int *alloca_vettore_2(  int size )
{
	int p[1000];
	return p ;
}
```

```
void main(void)
{
	int *v;
	v = alloca_vettore ( 10 ) ;
	... usa v ......
}
```

---

esempio corretto: vet_globale è una variabile globale, quindi sopravvive alla terminazione della funzione, ma cosìsia chiaro

```
int vet_globale[10000];

int *spazio_pre_allocato( void )
{
	return vet_globale ;
}
```

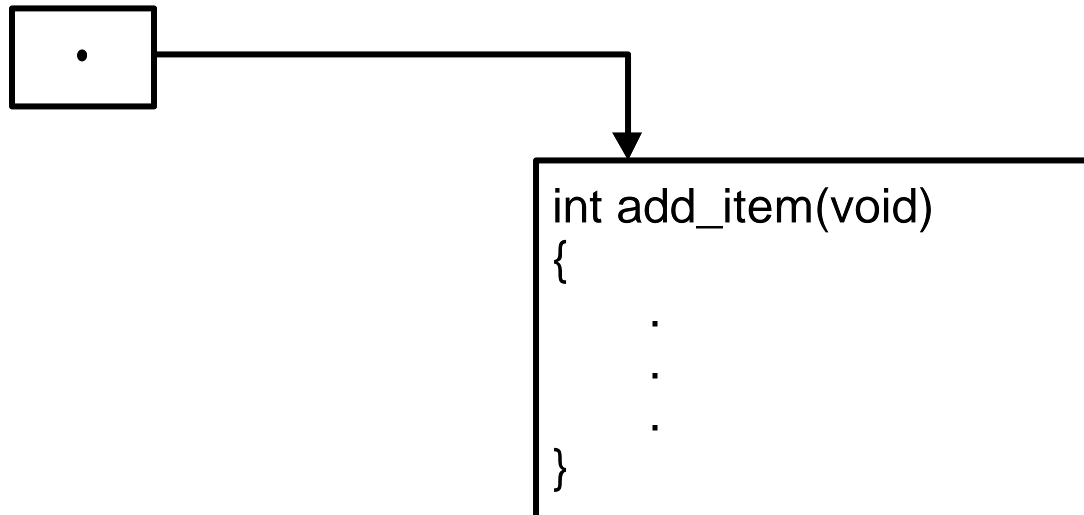
```
void main(void)
{
	int *v;
	v = spazio_pre_allocato();
	...  usa v ....
}
```

Pointers to Functions

Function names are pointer constants

- not a variable
- can regard as address of function code
- like array name is a pointer constant

So we can have pointers to functions



Puntatori a funzione.

In C è possibile utilizzare dei puntatori a funzioni, ovvero delle variabili a cui possono essere assegnati gli indirizzi in cui risiedono le funzioni, e tramite questi puntatori a funzione, le funzioni puntate possono essere chiamate all'esecuzione.

Confrontiamo la **dichiarazione di una funzione**:

tipo_restituito **nome_funzione** (paramdef1, paramdef2, ...)

con la **dichiarazione di un puntatore a funzione**:

tipo_restituito **(* nome_ptr_a_funz)** (paramdef1, paramdef2, ...)

dove:

- paramdef1, paramdef2, ecc. sono la definizione degli argomenti da passare alla funzione all'atto della chiamata (ad es.: int i).
- tipo_restituito è il tipo del dato che viene restituito come risultato dalla funzione.

ad es, la seguente dichiarazione definisce un puntatore a funzione che punta a funzioni le quali prendono come argomenti due double, e restituiscono un double .

```
double (*ptrf) ( double g, double f);
```

Il C tratta i nomi delle funzioni come se fossero dei puntatori alle funzioni stesse.

Quindi, **quando vogliamo assegnare ad un puntatore a funzione l'indirizzo di una certa funzione dobbiamo effettuare un operazione di assegnamento del nome della funzione al nome del puntatore a funzione**

Se ad es. consideriamo la funzione dell'esempio precedente:

double somma(double a, double b);

allora **potremo assegnare la funzione somma al puntatore ptrf** così:

ptrf = somma;

Analogamente, **l'esecuzione di una funzione mediante un puntatore che la punta,viene effettuata con una chiamata in cui compare il nome del puntatore come se fosse il nome della funzione**, seguito ovviamente dai necessari parametri.

--------- esempio di uso dei puntatori a funzione -----------

per es. riprendiamo l'esempio della somma:

```
double somma( double a, double b) ;          /*  dichiarazione  o prototipo */

void main( void)
{
      double A=10 , B=29,  C;
      double (*ptrf) ( double g, double f);

      ptrf = somma;
      C = ptrf (A,B);                        /* chiamata alla funz. somma */
}

double somma( double a, double b)            /* definizione */
{     return a+b ;    }
```

Osservazione: spesso è complicato definire il tipo di dato puntatori a funzione, ed ancora di più definire funzioni che prendono in input argomenti di tipo puntatori a funzione.
In queste situazioni **è sempre bene ricorrere alla typedef per creare un tipo di dato puntatore a funzione** per funzioni che ci servono, ed utilizzare questo tipo di dato nelle altre definizioni.

---- Usare la typedef per rendere leggibile il codice C ---

Esempio:
esiste in ambiente unix (l'esempio è preso da LINUX Slackware) una funzione detta signal che puo' servire ad associare una funzione al verificarsi di un evento. Ad es. puo' servire a far eseguire una funzione allo scadere di un timer.
Il prototipo della funzione, contenuto in signal.h, e' il seguente:

```
#include <signal.h>
void (*signal(int signum, void (*handler)(int) ) )(int);          /* ????? */
```

NON E' SUBITO CHIARISSIMO COSA SIA STA ROBA !!!

Il significato è che
1) la funzione signal vuole come parametri un intero ***signum***, ed un puntatore ***handler*** ad una funzione che restituisce un void e che vuole come parametro un intero.
2) la funzione signal restituisce un puntatore ad una funzione che restituisce un void e che vuole come parametro un intero.

Converrebbe definire un tipo di dato come il puntatore a funzione richiesta:

```
typedef  void (*tipo_funzione) (int);
```

ed utilizzarlo per definire la signal così:

```
tipo_funzione   signal ( int signum,  tipo_funzione handler );
```

Nel man della signal e' presente questo **commento**:

If you're confused by the prototype at the top of this manpage, it may help to see it separated out thus:

```
typedef void (*handler_type)(int);
handler_type signal(int signum, handler_type handler);
```

Pointers to Functions

Declaring pointers to functions

cannot omit parenthesis

```
int (*p)();      p is ptr to function that returns an int
int *p();        p is function that returns ptr to int
```

assign address of function as with array

```
int fname(void);
int (*fnptr)(void);
fnptr = fname;       fname è un indirizzo!!!!!
```

Pointers to Functions

Usage of pointers to functions

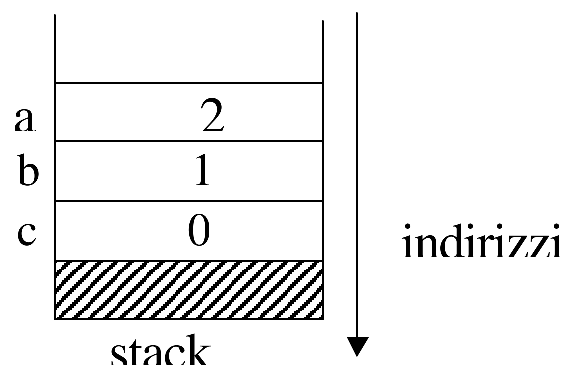
- are variables, just like other pointers
- can be used as arguments to functions to “pass” one function to another
- used to direct a sort
  - to make it data type independent
  - to allow choices of different function calls for ascending vs. descending
- used to implement “dispatch” tables
  - as might be used in a menuing system

Ordine di Valutazione degli argomenti passati alle funzioni.

Il compilatore C opera in modo che le espressioni passate come argomenti alle funzioni sono prima valutate, ed il risultato della valutazione viene scritto sullo stack per essere disponibile al codice che implementa la funzione stessa.
Sorgono due problemi:
- in che ordine vengono valutate le espressioni ?
- in che ordine vengono scritti sullo stack i risultati delle valutazioni ?

La risposta alle due domande è la stessa:
vengono prima valutate (e il risultato scritto sullo stack) le espressioni passate come ultimo argomento della funzione (le più a destra), e poi via via quelle più a sinistra.

es:
```
void stampa3( int a, int b, int c){
	printf("a=%d	b=%d	c=%d\n", a, b, c);
	printf("&a=%p	&b=%p	&c=%p\n", &a, &b, &c);
}
void main( void) {
	int x=0;
	stampa3 (x++, x++, x++);
}
```



L'output del programma sarà del tipo:
a=2 b=1 c=0 (c valutato per primo)
&a=9003:0FF8 &b=9003:0FFA &c=9003:0FFC (c primo su stack)
che indica come il terzo argomento (ultimo, cioè più a destra) venga valutato (e scritto sullo stack) per primo, e poi gli argomenti via via più a sinistra.

Funzioni con numero di argomenti variabile.

In C e' possibile definire funzioni aventi un numero di argomenti variabile, cioe' funzioni che in chiamate diverse possono avere un numero di argomenti diverso, **ma ne debbono avere sempre almeno uno**. Ne è un esempio la printf();

Si utilizza a questo scopo una struttura **va_list** definita nel file stdarg.h .

Vediamo un esempio di funzione che riceve in input n argomenti, di cui il primo e' un intero che contiene il numero di argomenti seguenti, e gli altri sono delle stringhe, cioe' dei puntatori a char. La funzione deve solo stampare tutti gli argomenti passati.

```
void print_lista_argomenti(int narg, ...)
{
va_list va;
int i;
char *ptrchar;

/* va_start inizializza va_list all'argomento, tra passati a print_lista_argomenti, che segue l'argomento narg indicato come secondo argomento nella va_start, cioè inizializza la lista va_list al primo degli argomenti variabili */
va_start(va,narg);
for(i=0;i<narg;i++)
   {
   ptrchar=va_arg(va,char*); /*ptrchar punta alla stringa passata*/
   printf("%d %s\n",i,ptrchar);
   }
va_end(va);
}
```

la funzione potra' essere chiamata in una di questi modi

```
print_lista_argomenti(0);
print_lista_argomenti(5,"primo","secondo","terzo","quarto","quinto");
```

Vediamo un esempio complicato ma utile di uso della lista di argomenti variabili. Stampa gli argomenti passati, anche se sono di tipo diverso. Usa un primo parametro come formato per sapere cosa viene passato nei successivi argomenti, indicando con s una stringa, con d un intero, con c un carattere.

```
#include <stdio.h>
#include <stdarg.h>
void stampa_argomenti(char *fmt, ...)
{
    va_list ap;
    int d;
    char c, *p, *s;

    va_start(ap, fmt);
    while (*fmt)
        switch(*fmt++) {
        case 's':           /* string */
            s = va_arg(ap, char *);
            printf("string %s\n", s);
            break;
        case 'd':           /* int */
            d = va_arg(ap, int);
            printf("int %d\n", d);
            break;
        case 'c':           /* char */
            c = va_arg(ap, char);
            printf("char %c\n", c);
            break;
        }
    va_end(ap);
}

void main (void) {
      stampa_argomenti ( "sdc" , "stringa" , (int) 10 , (char)'h' );
      stampa_argomenti ("s" , "stringa2");
}
```

AMBIENTI, VISIBILITÀ DELLE VARIABILI

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

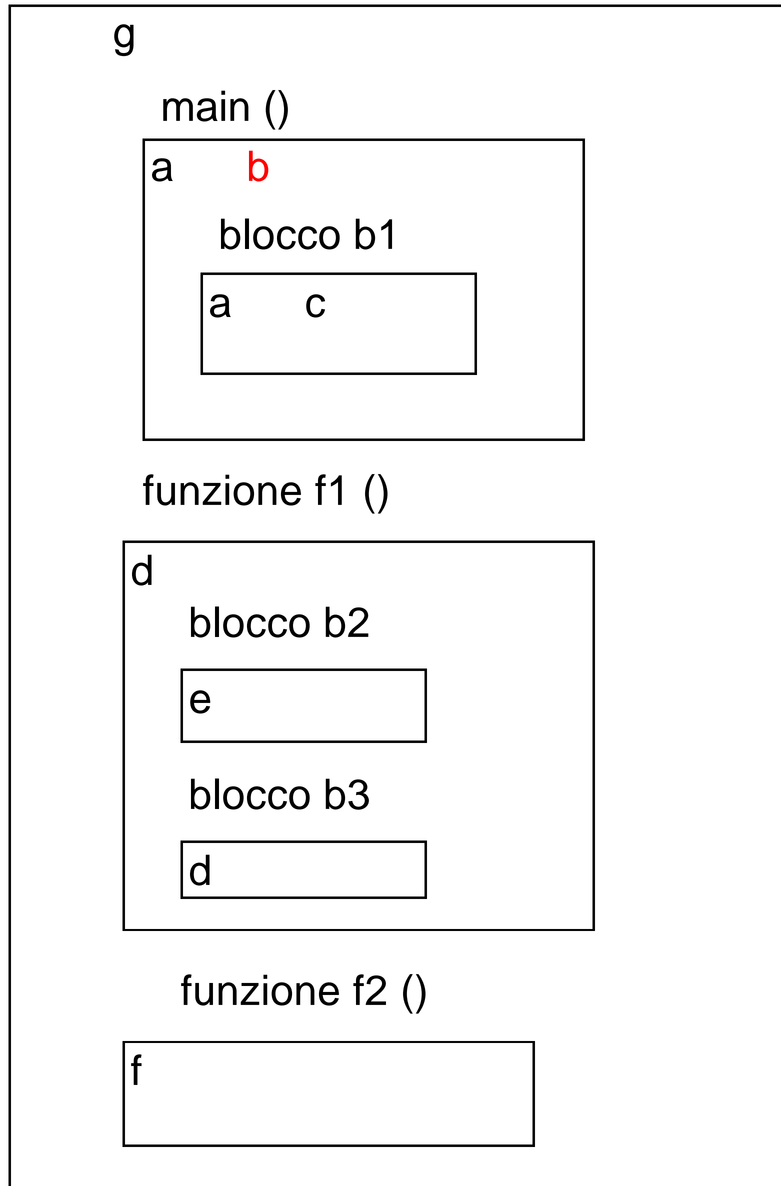
IDENTIFICATORE DI FUNZIONE:

- 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

Sample C Code

nested blocks

This is legal, but can be confusing!

```
main()
{
    int x = 1, y = 2;
    printf("x = %d, y = %d\n", x, y);
    {
        int x = 3, y = 4;
        printf("x = %d, y = %d\n", x, y);
    }
printf("x = %d, y = %d\n", x, y);
}
```

exists from here... to here

'exist' only within here

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)

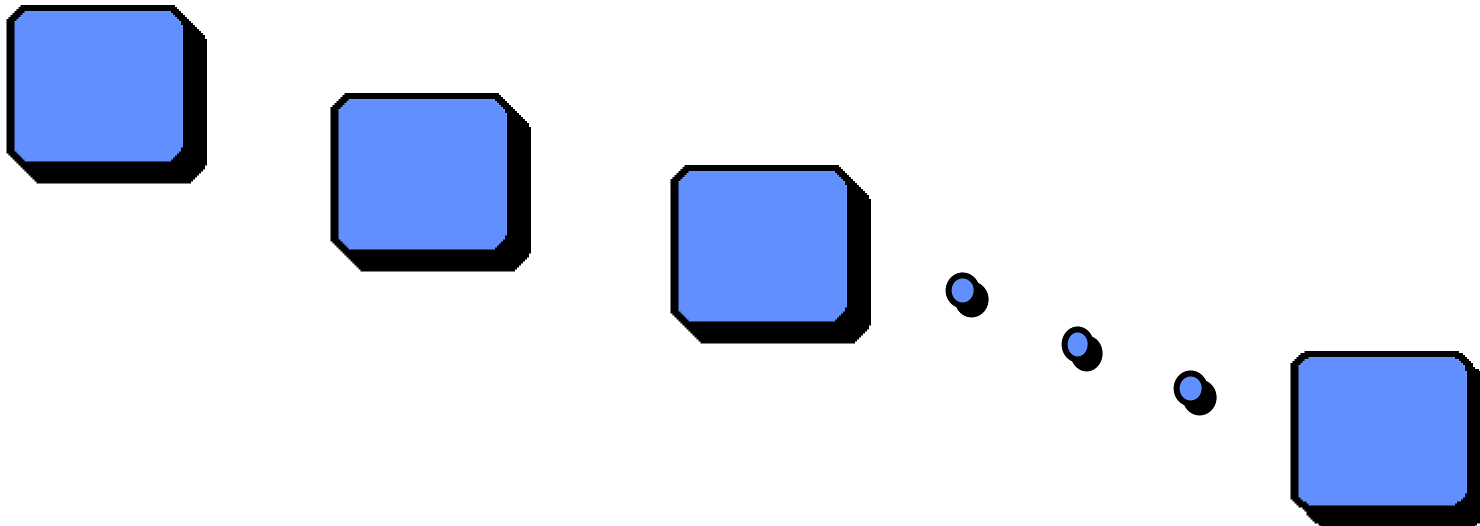
E' necessaria una opportuna gestione della memoria di lavoro allocata a contenere l'ambiente di esecuzione di un sottoprogramma: gestione a pila (stack) degli ambienti di esecuzione.

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) * ... 1

with 1! defined to be 1

and 0! defined to be 1

base case: when n = 1

recursion step: n * (factorial of n -1)

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

- Recursive function: it calls itself
- 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

Copyright ©1997 by A. E. Naiman

Recursive vs. Iterative Functions

```
// 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.

Copyright ©1997 by A. E. Naiman

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);
```

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		
1040		
1042		
1044		
1046		
1048		
1050		

X = F(Y,Z);

nel caso di recursione

Loc1 = F(Loc1,Loc2);

Loc1 = F(Loc1,Loc2);

next stack

next heap

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		
1040		
1042		
1044		
1046		
1048		
1050		

Allocazione stack

Allocazione Heap

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);
}
```