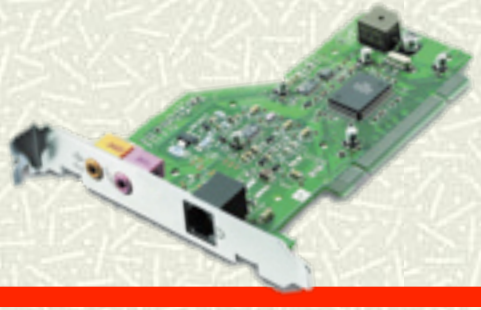
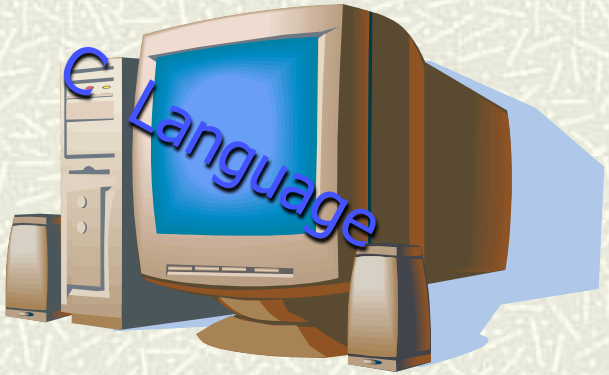


Informatica Generale

Linguaggio C

Argomenti Avanzati

- I puntatori
- Stack
- Visibilità delle Variabili



La dichiarazione di tipo: typedef - 1

- Il C consente di associare ai tipi di dati nomi definiti dal programmatore, mediante la parola chiave typedef
  - Dal punto di vista sintattico, la dichiarazione di tipo è analoga alla dichiarazione di variabile
  - Dal punto di vista semantico, il nome definito diviene un sinonimo di un tipo di dati e la dichiarazione non produce allocazione immediata di memoria
- Esempio:

```
typedef long int EIGHT_BYTE_INT;
```

  rende EIGHT_BYTE_INT un sinonimo di **long int**
- Per convenzione, i nomi di tipo sono scritti con lettere maiuscole, per non confonderli con i nomi di variabile

La dichiarazione di tipo: typedef - 2

- La dichiarazione di tipo deve apparire in un programma prima che il tipo venga adoperato per la dichiarazione di variabili
- Le dichiarazioni di tipo sono particolarmente utili nella definizione di tipi composti
- Avvertenza: **typedef** e #define non sono equivalenti…

```
#define PT_TO_INT  int *

PT_TO_INT p1, p2;
```

```
int  *p1, p2;
```

```
typedef  int *  PT_TO_INT;

PT_TO_INT p1, p2;
```

```
int  *p1, *p2;
```

Il reperimento dell'indirizzo di un oggetto - 1

- Per ottenere l'indirizzo di una variabile si usa l'operatore **&**
- **Esempio**: Se j è una variabile **long int** con indirizzo 2486, allora l'istruzione…

ptr=&j;

memorizza l'indirizzo 2486 nella variabile ptr

- **Esempio**:

```
#include<stdio.h>

main()
{
    int j=1;

    printf("Il valore di j è: %d\n", j);
    printf("L'indirizzo di j è: %p\n", &j);
    exit(0);
}
```

L'indirizzo che si ottiene varia per esecuzioni diverse dello stesso programma

%p è lo specificatore per stampare l'indirizzo di un dato

Il reperimento dell'indirizzo di un oggetto - 2

- L'operatore **&** non è utilizzabile nella parte sinistra di un'istruzione di assegnamento
- Non è possibile cambiare l'indirizzo di un oggetto, pertanto…

&x = 1000; * ILLEGALE *

- È il compilatore — che sfrutta i servizi offerti dal sistema operativo — l'unico gestore della memoria allocata all'esecuzione di un programma

Introduzione ai puntatori - 1

- Nell'istruzione di assegnamento

  ptr = &j;

  la variabile che contiene l'indirizzo di j non può essere una normale variabile intera, ma un tipo speciale di variabile, chiamato **puntatore**: memorizzando un indirizzo, esso "punta" ad un oggetto

- Per dichiarare una variabile puntatore, si fa precedere al nome un asterisco:

  long *ptr;

  il tipo di dati **long** fa riferimento al tipo di variabile a cui ptr può puntare

Introduzione ai puntatori - 2

ESEMPI

```
/* CORRETTO */
long *ptr;
long long_var;
ptr = &long_var;
```

```
/* NON CORRETTO */
long *ptr;
float float_var;
ptr = &float_var;
```

```
#include<stdio.h>

main()
{
    int j=1;
    int *pj;

    pj = &j;  /*Assegna l'indirizzo di j a pj */
    printf("Il valore di j è: %d\n", j);
    printf("L'indirizzo di j è: %p\n", pj);
    exit(0);
}
```

L'accesso a variabile puntata - 1

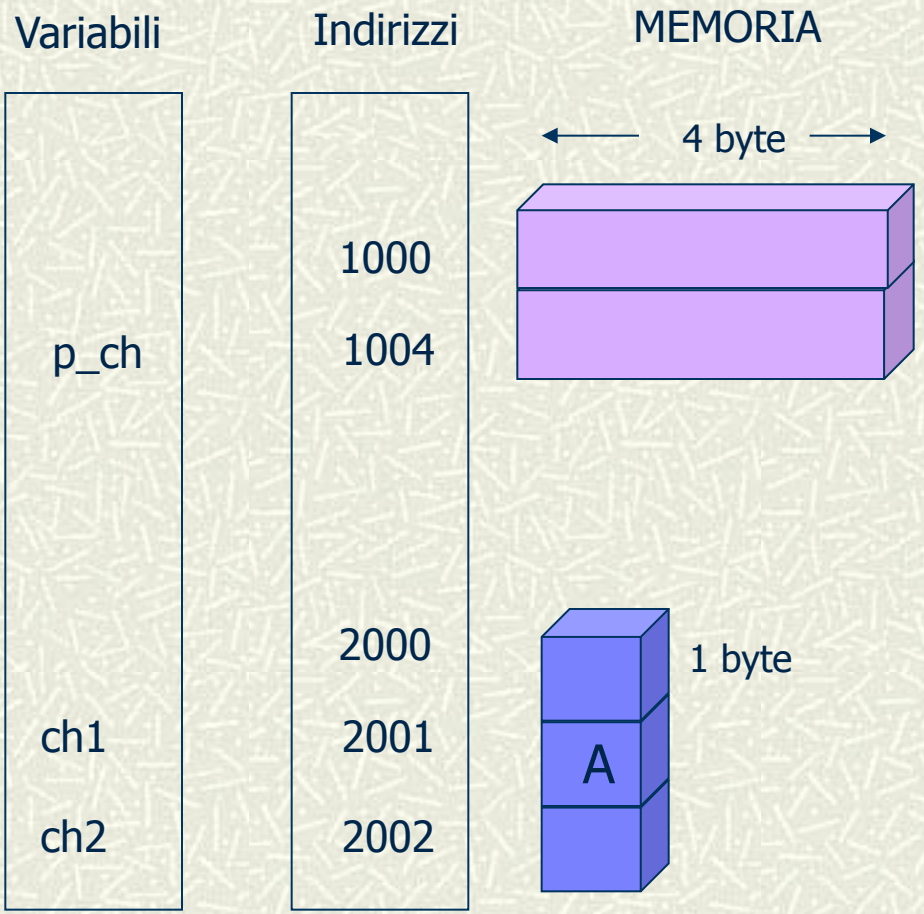
- Si usa l'asterisco * anche per accedere al valore che è memorizzato all'indirizzo di memoria contenuto in una variabile puntatore

```
#include<stdio.h>

main()
{
    char *p_ch;
    char ch1='A', ch2;

    printf("L'indirizzo di p_ch è: %p\n", &p_ch);
    p_ch = &ch1;
    printf("Il valore contenuto in p_ch è %p\n, p_ch);
    printf("Il valore contenuto all'indirizzo \
            puntato da p_ch è: %c\n", *p_ch);
    ch2 = *p_ch;
    exit(0);
}
```

L'accesso a variabile puntata - 2

CODICE

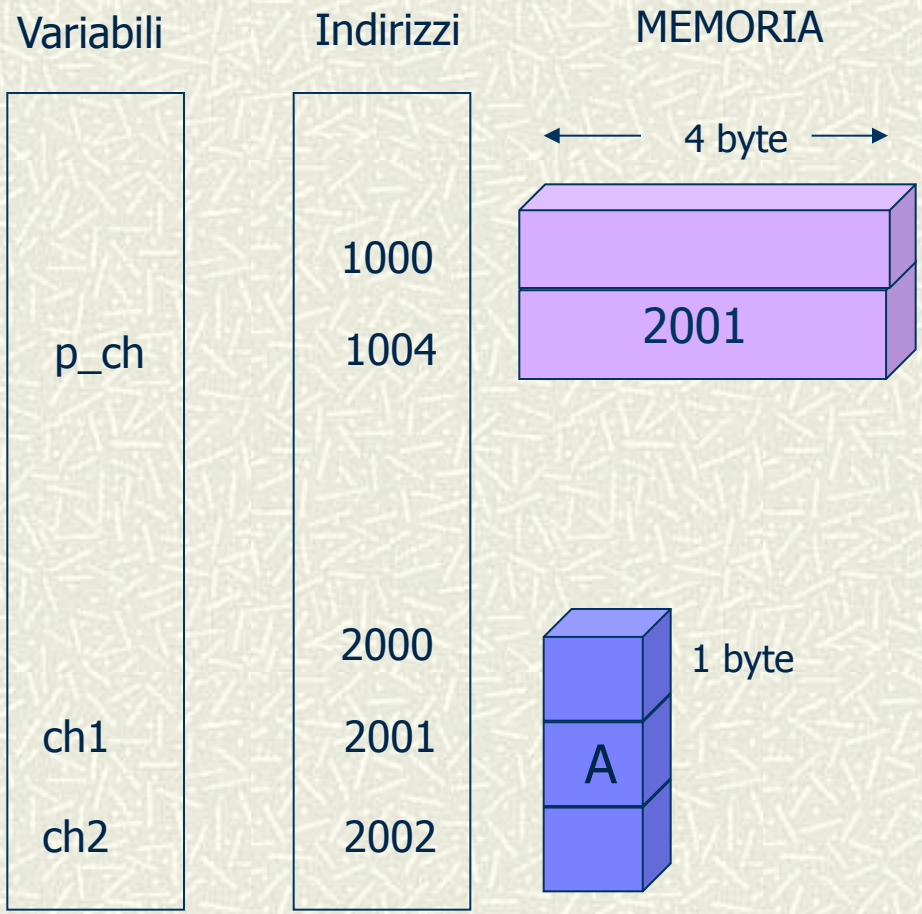
```
char *p_ch;
char ch1='A', ch2;
```



L'accesso a variabile puntata - 3

CODICE

```
p_ch = &ch1;
```

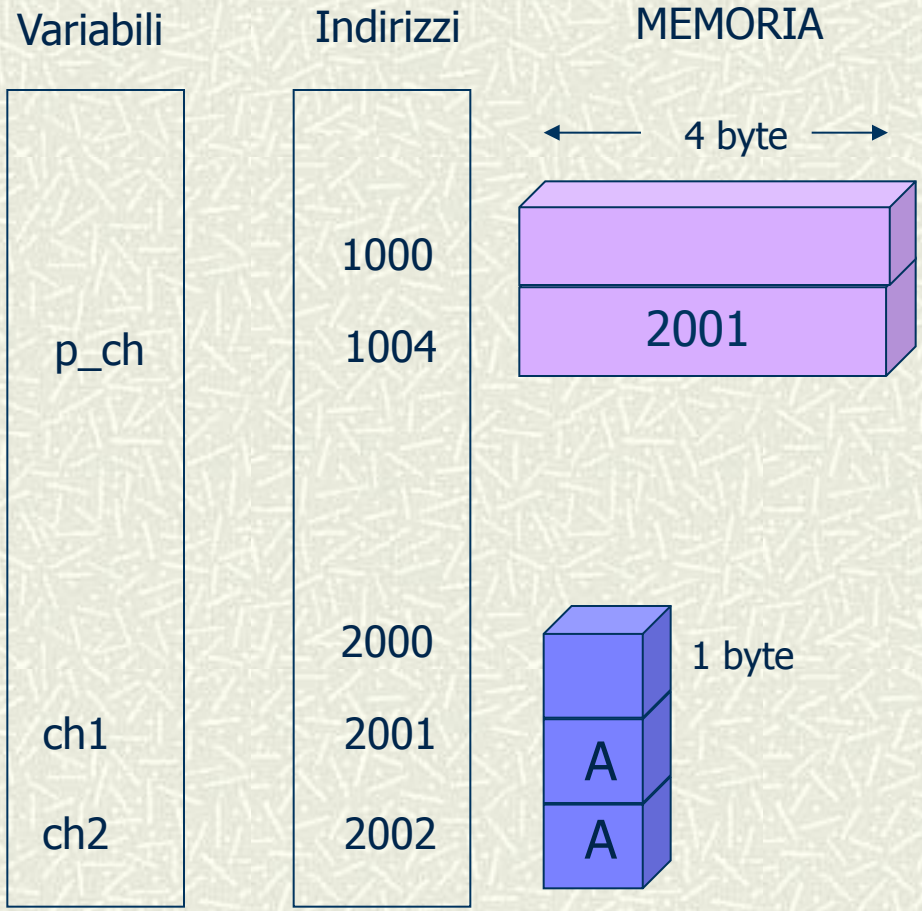


L'accesso a variabile puntata - 4

CODICE

```
ch2 = *p_ch;
```

Prendi l'indirizzo memorizzato in p_ch e leggi il valore contenuto nella locazione di memoria identificata da tale indirizzo; copialo in ch2



L'accesso a variabile puntata - 5

- Il tipo di dato contenuto nella dichiarazione del puntatore indica il tipo del risultato dell'operazione "accesso all'indirizzo contenuto in"
- Esempio: La dichiarazione

float *fp;

significa che quando *fp appare in un'espressione il risultato è di tipo **float**; l'espressione *fp può anche apparire alla sinistra di un'istruzione di assegnamento

***fp = 3.15;**

che memorizza il valore 3.15 nella locazione di memoria puntata da fp

- Esempio: L'assegnazione

fp = 3.15;

è scorretta poiché gli indirizzi "non sono numeri" interi né floating-point, e non possono essere "assegnati"

L'inizializzazione dei puntatori

- I puntatori possono essere inizializzati: il valore iniziale deve essere un indirizzo

```
int  j;
int  *ptr_to_j=&j;
```

- Non è possibile fare riferimento ad una variabile prima di averla dichiarata; la dichiarazione seguente non è corretta…

```
int  *ptr_to_j=&j;
int  j;
```

Stack

Uno Stack è un insieme dinamico in cui l'elemento rimosso dall'operazione di cancellazione è predeterminato.

In uno Stack questo elemento è l'ultimo elemento inserito.

Uno Stack implementa una lista di tipo "last in, first out" (LIFO)

- *Nuovi elementi vengono inseriti in testa e prelevati dalla testa*

Operazioni su Stack

Due Operazioni di Modifica:

Inserimento: `Push(S,x)`

- *aggiunge un elemento in cima allo Stack*

Cancellazione: `Pop(S)`

- *rimuove un elemento dalla cima dello Stack*

Altre operazioni: `Stack-Vuoto(S)`

- *verifica se lo Stack è vuoto (ritorna* `True` *o* `False`*)*

Operazioni su Stack

Due Operazioni di Modifica:

Inserimento: `Push(S,x)`

- *aggiunge un elemento in cima allo Stak*

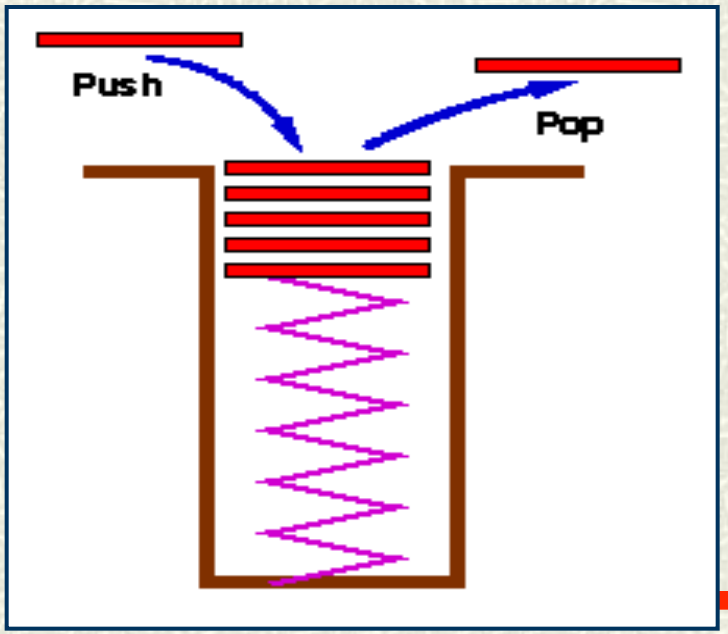
Cancellazione: `Pop(S)`

- *rimuove un elemento dalla cima dello Stack*

Altre operazioni: `Stack-Vuoto(S)`

- *verifica se lo Stack è vuoto*
- *(ritorna* `True` *o* `False`*)*

Uno Stack può essere immaginato come una pila di piatti!



Operazioni su Stack

```
Algoritmo Stack-Vuoto(S)
   IF top[S] = 0
      THEN return TRUE
      ELSE return FALSE
```

top[S]: un intero che denota, in ogni istante, il numero di elementi presenti nello Stack

Operazioni su Stack

```
Algoritmo Stack-Vuoto(S)
   IF top[S] = 0
      THEN return TRUE
      ELSE return FALSE
```

```
Algoritmo Push(S,x)
   top[S] = top[S]+1
   S[top[S]] = x
```

Assumiamo qui che l'operazione di *aggiunta* di un elemento nello Stack S sia realizzata come l'*aggiunta* di un *elemento ad un array*

Operazioni su Stack

- Problema:
  - Che succede se eseguiamo un operazione di `pop` (estrazione) di un elemento quando lo Stack è vuoto?
  - Questo è chiamato Stack Underflow. É necessario implementare l'operazione di `pop` con un meccanismo per verificare se questo è il caso.

Operazioni su Stack

```
Algoritmo Stack-Vuoto(S)
   IF top[S] = 0
      THEN return TRUE
      ELSE return FALSE
```

```
Algoritmo Push(S,x)
   top[S] = top[S]+1
   S[top[S]] = x
```

```
Algoritmo Pop(S)
   IF Stack-Vuoto(S)
      THEN ERROR "underflow"
      ELSE top[S] = top[S]-1
           return S[top[S]+1]
```

Stack: applicazioni

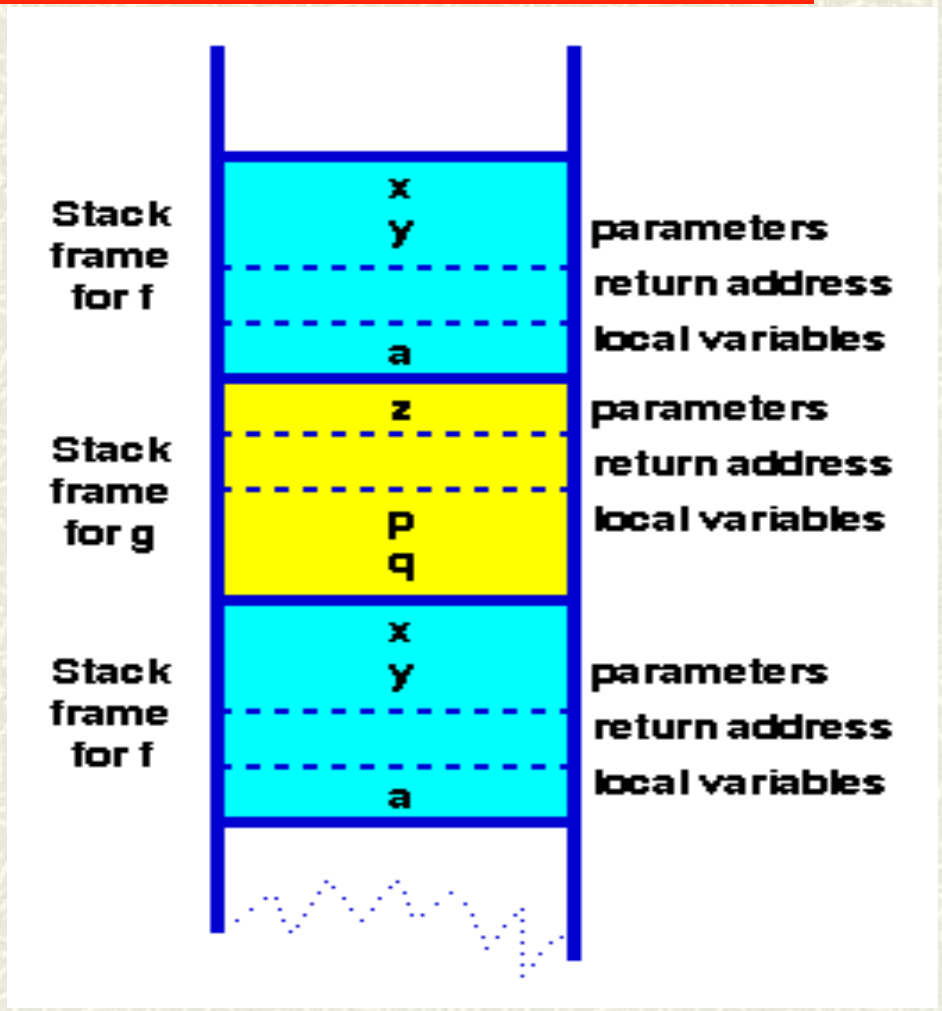
- Stacks sono molto frequenti:
  - Elemento chiave nel meccanismo che implementa la chiamata/return a funzioni/procedure
  - *Record di attivazione* permettono la ricorsione.
  - Chiamata: *push* di un record di attivazione
  - Return: *pop* di un record di attivazione
- Record di Attivazione contiene
  - Argomenti di funzioni
  - Indirizzo di ritorno
  - Valore di ritorno
  - Variabili locali della funzione

Stack di Record di Attivazione

```
function f(int x,int y)
 {
   int a;
   if ( term_cond )
        return …;
   a = ….;
   return g( a );
   }

function g( int z )
{
   int p, q;
   p = ….  ; q = ….  ;
   return f(p,q);
   }
```

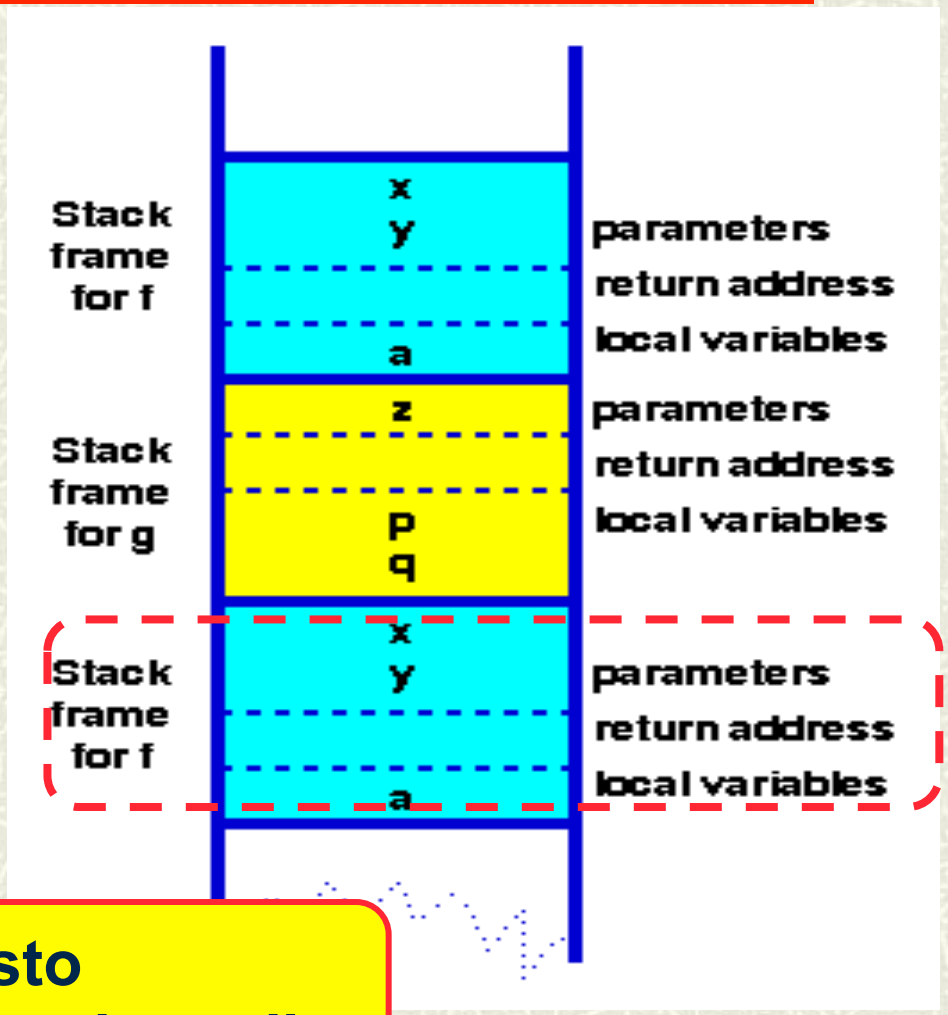
Stack di Record di Attivazione

```
function f(int x,int y)
 {
    int a;
    if ( term_cond )
          return …;
    a = ….;
    return g( a );
    }

function g( int z )
{
    int p, q;
    p = ….  ; q = ….  ;
    return f(p,q);
    }
```



Stack di Record di Attivazione

```
function f(int x,int y)
 {
    int a;
    if ( term_cond )
          return …;
    a = ….;
    return g( a );
    }

function g( int z )
{
    int p, q;
    p = ….  ; q = ….  ;
    return f(p,q);
    }
```



Contesto di esecuzione di `f`

Factorial

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

int main ()
{
  printf ("%d\n", factorial(3));
  return 0;
}
```

Visibilità delle variabili

- Ogni variabile è definita all'interno di un preciso *ambiente di visibilità (scope)*.
- Variabili *globali*
  - definite all'esterno al main sono visibili da tutti i moduli.
- Variabili *locali*
  - definite all'interno del main (sono visibili solo all'interno del main);
  - più in generale, definite all'interno di un blocco (sono visibili solo all'interno del blocco).

Struttura a blocchi

- In C è possibile aggregare gruppi di istruzioni in **blocchi** racchiudendole tra parentesi graffe;
- significato: *delimitazione di un ambiente di visibilità di "oggetti" (variabili).*
- Esempio:

```
{
    int a=2;
    int b;

    b=2*a;
}
```

a e b sono definite solo all'interno del blocco!

Visibilità delle variabili - Esempio

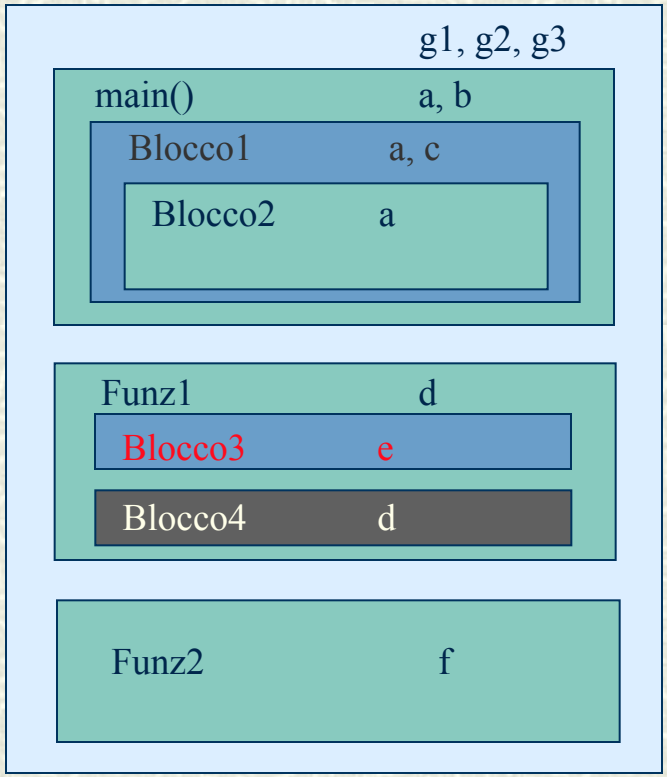
```
int n;
double x;
main()
{
    int a,b,c;
    double y;
    {
        int d;
        double z;
    }
}
```

- n, x: visibili in tutto il file
- a, b, c ,y: visibili in tutto il main
- d, z: visibili solo nel blocco

BLOCCHI E AMBIENTE DI VISIBILITA' DELLE VARIABILI

- Un blocco in C è composto da una parte dichiarativa (facoltativa) e una sequenza di istruzioni, racchiuse tra {}.
- Diversi blocchi (in successione o annidati) possono comparire nel main o nelle funzioni
- AMBIENTE GLOBALE: l'insieme di tutti gli elementi dichiarati esternamente al main e alle funzioni e validi ovunque nel programma.
- AMBIENTE LOCALE di una funzione è l'insieme di tutti gli elementi elencati nella parte dichiarativa e nella testata e validi solo all'interno della funzione.
- AMBIENTE DI BLOCCO: l'insieme di tutti gli elementi dichiarati in un blocco e validi localmente.

VISIBILITA' DELLE VARIABILI



main	g1, g2, g3	locali a, b	chiama Funz1, Funz2
Blocco1	g1, g2, g3, b	locali [a], c	chiama Funz1, Funz2
Blocco2	g1, g2, g3, b, c	locali {[a]}	chiama Funz1, Funz2
Funz1	g1, g2, g3	locali d	chiama Funz1, Funz2
Blocco3	g1, g2, g3, d	locali e, d	chiama Funz1, Funz2
Blocco4	g1, g2, g3	locali [d]	chiama Funz1, Funz2
Funz2	g1, g2, g3	locali f	chiama Funz1, Funz2

DURATA DELLE VARIABILI

- Le CLASSI DI MEMORIZZAZIONE definiscono quando una variabile è attiva (durata), lo scopo (globale o locale) e la sua visibilità
- CLASSE REGISTER: se una variabile viene usata frequentemente può essere utile dichiararla in classe register; il sistema la memorizzerà direttamente nei registri del processore.
- CLASSE AUTOMATIC: è la classe default; durante l'esecuzione di un blocco la variabile esiste ed è attiva; quando l'esecuzione del blocco termina, la variabile cessa di esistere. *Esistere=Occupare memoria!*

CLASSI DI MEMORIZZAZIONE

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

3
2
1

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

3

0 valori corrispondenti a var NON inizializzate

-27 valori corrispondenti a var NON inizializzate

CLASSI DI MEMORIZZAZIONE

- CLASSE EXTERNAL: lo scopo delle var external è globale;
  - le external DEVONO essere dichiarate al di fuori delle funzioni (senza modificatore Extern)
  - POSSONO essere dichiarate all'interno delle funzioni precedute dalla parola chiave extern.

```
Int x = 123;
main(){
        extern x ;              /* indica che x è dichiarata alttrove */
        printf("%d\n", x);
}
```

123