Fondamenti di Informatica

Allievi Automatici
A.A. 2015-16

Modello di Esecuzione

Il blocco

- Può comparire ovunque la sintassi preveda un'istruzione
- Si compone di due parti racchiuse tra { }
  - una parte dichiarativa (*facoltativa*)
  - una sequenza di istruzioni
- Due blocchi possono essere:
  - ***annidati*** (uno è interno all'altro)
  - ***paralleli*** (entrambi interni a un terzo blocco, ma non annidati tra loro)

Visibilità degli identificatori

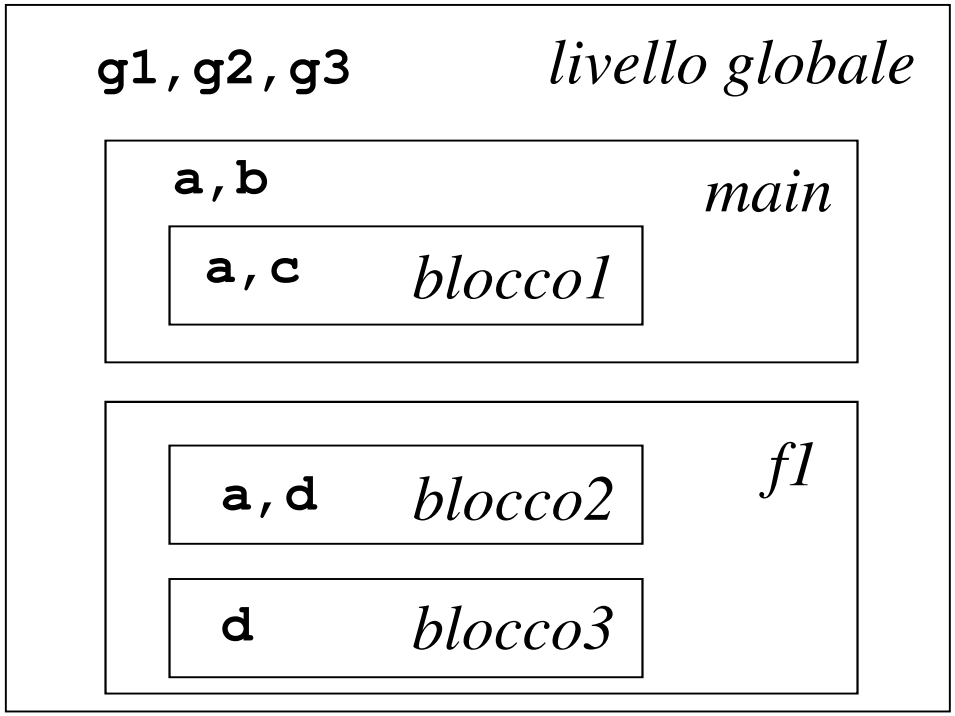
- Si possono dichiarare variabili all'interno di ogni blocco { … }
- Le variabili dichiarate all'interno di un blocco sono **visibili** solo all'interno di quel blocco
- Le variabili dichiarate all'interno di un sottoprogramma (o nella sua testata) sono visibili nell'intero sottoprogramma
- Variabili dichiarate a livello globale sono visibili nell'intero programma (main + sottoprogrammi)
- Lo stesso vale per le definizioni di tipo

Ambiti di visibilità

- Con “visibile” intendiamo:
  - può essere **vista**, cioè usata o referenziata tramite il suo identificatore
    - Usata e valutata in un’espressione
    - Gli si può assegnare un valore (se è una *variabile*)
    - Può essere usata per dichiarare tipi e variabili (se è una *definizione di tipo*)
    - Può essere invocata (se è una *funzione*)

Una deroga alla visibilità

- La dichiarazione di un elemento in una funzione o in un blocco ***maschera*** *le* eventuali entità omonime più “esterne”

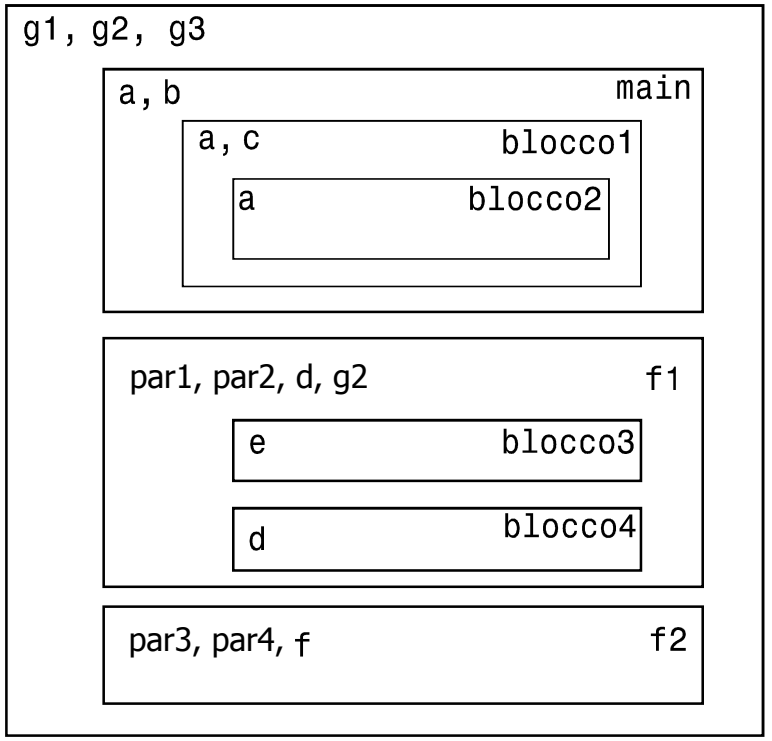


È possibile riusare nomi di variabili

```
int         g1, g2;
char        g3;
int         f1(int  par1, int  par2);   /* Prototipo di f1 */
main () {
        int  a, b;
        int  f2(int  par3, int  par1);   /* Prototipo di f2 */
        {    char  a, c;
             ...
             {    float         a;
                  ...
             } /* Fine blocco2 */
        } /* Fine blocco1 */
} /* Fine main */
int     f1(int  par1, int  par2) {
        int  d, g2;
        {    int  e;
             ...
        } /* Fine blocco3 */
        {    int  d;
             ...
        } /* Fine blocco4 */
} /* Fine f1 */
int     f2(int  par3, int  par4) {
        int  f;
             ...
} /* Fine f2 */
```

modello "a contorni"

Programma `ComplessoInStruttura`



Durata delle variabili (tempo di vita)

- Va dalla "**creazione**"
  (allocazione della memoria)
  alla "**distruzione**"
  (rilascio della memoria allocata)
- Due classi di variabili:
  - variabili **statiche** (non automatiche)
  - variabili **automatiche**

Variabili statiche e automatiche

- **Statiche**:
  - Allocate una volta per tutte
  - Distrutte solo al termine dell'esecuzione del **programma**
  - Sono tutte le variabili globali e quelle locali al main()
    - Anche le variabili di funzione o blocco dichiarabili **`static`**
  - Possono fungere da canale di comunicazione tra funzioni (quelle globali)
- **Automatiche**:
  - Sono **create** quando il flusso di esecuzione "entra" nel loro **ambito di visibilità**
  - Sono **distrutte** all'uscita del flusso da tale ambito
  - Sono le variabili dichiarate nei blocchi e nelle funzioni (inclusi i parametri formali)

Variabili automatiche

- N.B.: le variabili automatiche di blocchi (o funzioni) eseguiti più volte:
  - Sono allocate di volta in volta in celle differenti
  - Non conservano i valori prodotti da precedenti esecuzioni della funzione o del blocco
    - Per conservare i valori delle precedenti esecuzioni occorre dichiararle esplicitamente come **`static`**

```
typedef struct { int actualSize, contents[SIZE]; } tabella;
tabella myTab;
void stampa(void);
int search(int k); /*dichiarazione di funzione*/
int main() {
    int i, pos, val;
    printf("Dimensione tabella?( < %d) : ", SIZE);
    scanf("%d", &myTab.actualSize);
    /*bisognerebbe controllare che sia < SIZE*/
    for( i=0; i<myTab.actualSize; i++ ) {
            printf("dammi un valore della tabella : ");
            scanf("%d", &myTab.contents[i]);
    }
    printf("dammi valore da cercare in tabella : ");
    scanf("%d", &val);
    stampa();
    pos = search(val);
    if ( pos != -1 ) printf("elemento trovato in posizione %d \n", pos);
    else             printf("valore non trovato in tabella\n");
    return 0;
}
```

```
void stampa( void ){
      int j;
      printf("ecco la tabella :\n");
      for (j = 0; j < myTab.actualSize; j++)
            printf("%d  ", myTab.contents[j]);
      printf("\n");
}
```

```
int search( int k ) {
      int   n;
      for( n=0; n<myTab.actualSize; n++ )
            if (myTab.contents[n]==k)
                  return n;
      return -1;
}
```

Ambiente globale e “side effects”

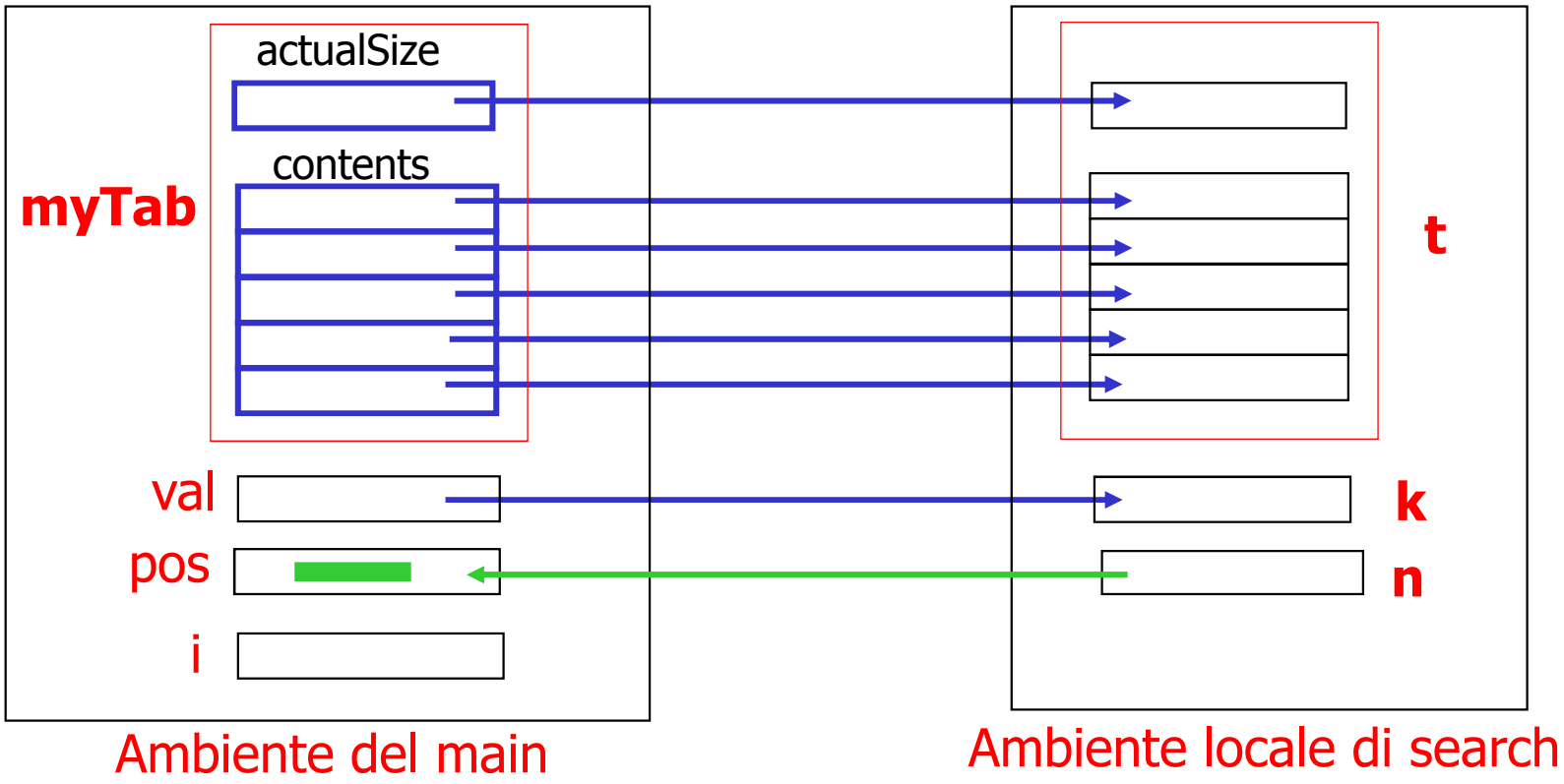
- Esiste un *ambiente globale*
  - Che contiene la variabile myTab
- Le macchine di *stampa()* e di *search()* possono accedere all’ambiente globale
  - Se lo modificano, possono generare ciò che si chiama un effetto collaterale (*side effect*) sul programma chiamante
    - Effetto **non previsto** dalla "semantica naturale" delle due funzioni (intuitivamente, non dovrebbero modificare myTab)
- Per garantire formalmente che non accada… passiamo la tabella per copia!

```
typedef int array [SIZE];   /* Parte dichiarativa globale */
typedef struct { int actualSize;     /* Dich. di tipo */
                 array contents; } tabella;
int search(tabella, int); /* Dich. funzione */

int main() {
        int i, pos, val;
        tabella myTab;
        printf("Dimensione tabella?(<%d): ", SIZE);
        scanf("%d", &myTab.actualSize);
        /*bisognerebbe controllare che sia < SIZE*/
        for(i=0; i<myTab.actualSize; i++) {
             printf("dammi un valore della tabella ");
             scanf("%d", &myTab.contents[i]);          }
        printf("dammi valore da cercare in tabella ");
        scanf("%d", &val);
        pos = search(myTab, val);
        if (pos!=-1)  printf("elemento trovato in posizione %d \n", pos);
        else          printf("valore non trovato in tabella\n");
        return 0;
}
```

```
int search( tabella t, int k ) {
  int  n;
  for( n=0; n < t.actualSize; n++ )
    if( t.contents[n] == k )
      return n;
  return -1;
}
```

Effetto dell'esecuzione di **pos = search(myTab, val);**



Passaggio parametri: **copia di valori da un ambiente all'altro**

Sottoprogramma di inserimento

-- tentativo errato--

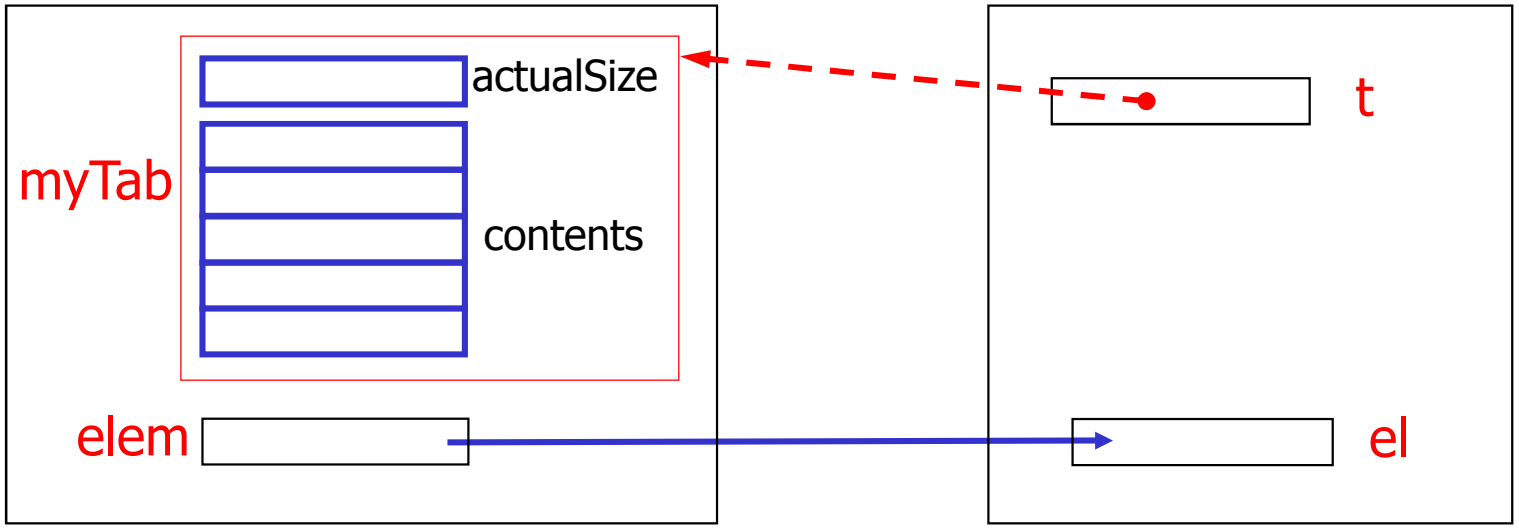
Vogliamo scrivere una procedura che inserisca nella tabella un intero el (nella prima posizione libera)

```
void insert (tabella t, int el) {\
      if (t.actualSize<SIZE) {
          t.actualSize++;
          t.contents[t.actualSize-1] = el;
      }
}
```

Con questa procedura la copia locale della tabella è distrutta al termine dell'esecuzione della procedura, e la tabella del main resta immutata

```
insert(&myTab, elem);          /* Chiamata*/

void insert( tabella * t, int el ) {    /* Definizione della procedura insert */
   if (t–>actualSize<SIZE) {
      t–>actualSize++;
      t–>contents[actualSize-1] = el;
   }
}
```



Ambiente del main

Ambiente locale di insert

Side effects tramite passaggio di puntatori (passaggio parametri per indirizzo)

- Modifiche all'ambiente globale possono causare side effect
  - In quanti modi una funzione può agire sull'ambiente del chiamante?
- I sottoprogrammi possono solo:
  - restituire un valore,
  - modificare l'ambiente globale,
  - a loro volta passare parametri (per valore, cioè passarne una **copia**) ad altri sottoprogrammi.
  - usare i **puntatori** ricevuti come parametri per modificare ambienti diversi da quello locale (tipicamente, quello del chiamante)
- Questo ultimo è il modo principale per causare effetti sul programma chiamante tramite una funzione
  - I parametri passati sono contenuti nel ***record di attivazione (…)***

Funzioni: modello di esecuzione

- Immaginiamo che esista **una macchina dedicata** al compito di eseguire il programma **main()**
- Immaginiamo che sia **creata** una nuova macchina dedicata, all'atto della **chiamata** di ogni funzione
- Le macchine dedicate hanno una loro **memoria** (detta ***ambiente*** della funzione)
  - per le variabili locali,
  - per i valori dei parametri che ricevono, e
  - per il risultato che eventualmente restituiscono
- **Vediamo come UNA SOLA MACCHINA può simulare questo modello di esecuzione**

Chiamata a sottoprogramma

- Al livello di astrazione del programmatore
  - In seguito a una chiamata a sottoprogramma, il programma in corso viene **sospeso** e il controllo passa al sottoprogramma
- Al livello della macchina C:
  - Salvataggio del program counter (PC) e del **contesto** del programma chiamante
  - Assegnazione al PC dell'indirizzo del sottoprogramma
  - Esecuzione del sottoprogramma
  - Ritorno al programma chiamante con ripristino del suo **contesto**

Riassumendo

- Ad ogni chiamata a sottoprogramma
  - si crea ***virtualmente*** una macchina dedicata
  - Ogni macchina ha una sua memoria locale
    - per le variabili dichiarate localmente,
    - per i valori degli eventuali parametri ricevuti, e
    - per il risultato che eventualmente devono restituire
- All'uscita dal sottoprogramma
  - la macchina dedicata viene ***virtualmente*** distrutta

Queste "creazioni / distruzioni" sono simulate dal salvataggio / ripristino dei **contesti** e delle "memorie private" di ogni diversa **attivazione** delle funzioni, grazie al modo in cui è realizzato l'esecutore

Si ha una simulazione delle "macchine virtuali dedicate"

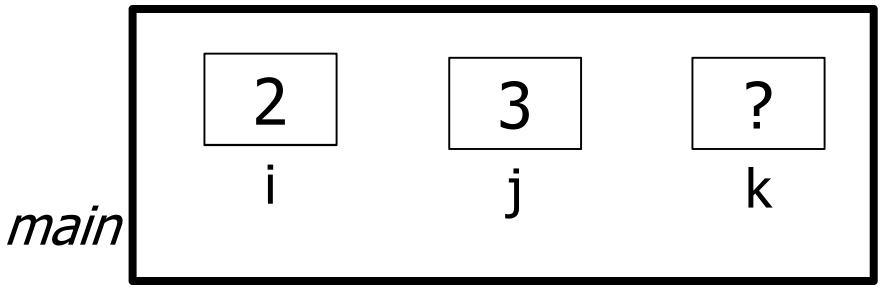
Record di attivazione

- Ogni funzione (incluso il main) ha associato un **record di attivazione** che contiene:
  - tutti i dati relativi all'ambiente locale del sottoprogramma
  - l'indirizzo di ritorno nel programma chiamante
  - altri dati utili
- Il record di attivazione è il *modello* di quello che abbiamo chiamato *ambiente*
- **Per ogni attivazione di sottoprogramma si crea un nuovo record di attivazione**

Esempio di codice

```
int moltiplica( int x, int y ) {
    int r;
    r = x * y;
    return r;
}

                int power( int b, int e ) {
                    int i, p=1;
                    for( i=1 ; i<=e ; i++ )
                        p = moltiplica( p, b );
                    return p;
                }
int main() {
    int i=2,j=3,k;
    k=moltiplica(i,j);
    k=power(i,j);
    return 0;
}
```

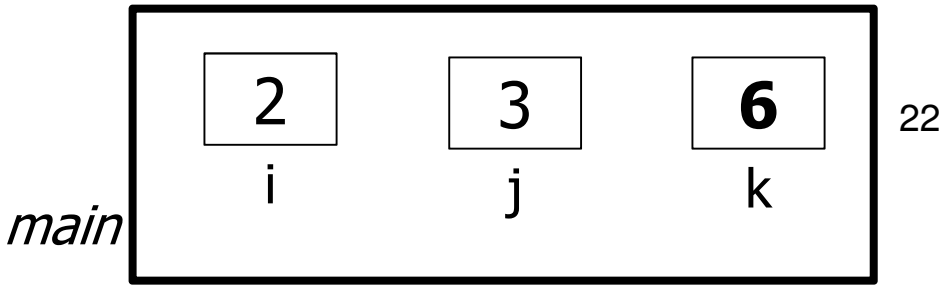


Esempio di codice

```
int moltiplica( int x, int y ) {
    int r;
    r = x * y;
    return r;
}
```

```
int power( int b, int e ) {
    int i, p=1;
    for( i=1 ; i<=e ; i++ )
        p = moltiplica( p, b );
    return p;
}
```

```
int main() {
    int i=2,j=3,k;
    k=moltiplica(i,j);
    k=power(i,j);
    return 0;
}
```

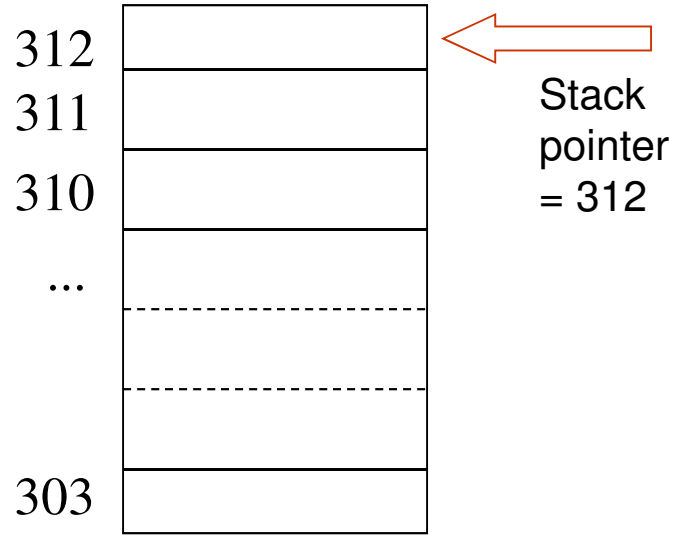


La soluzione: la pila (stack) di sistema

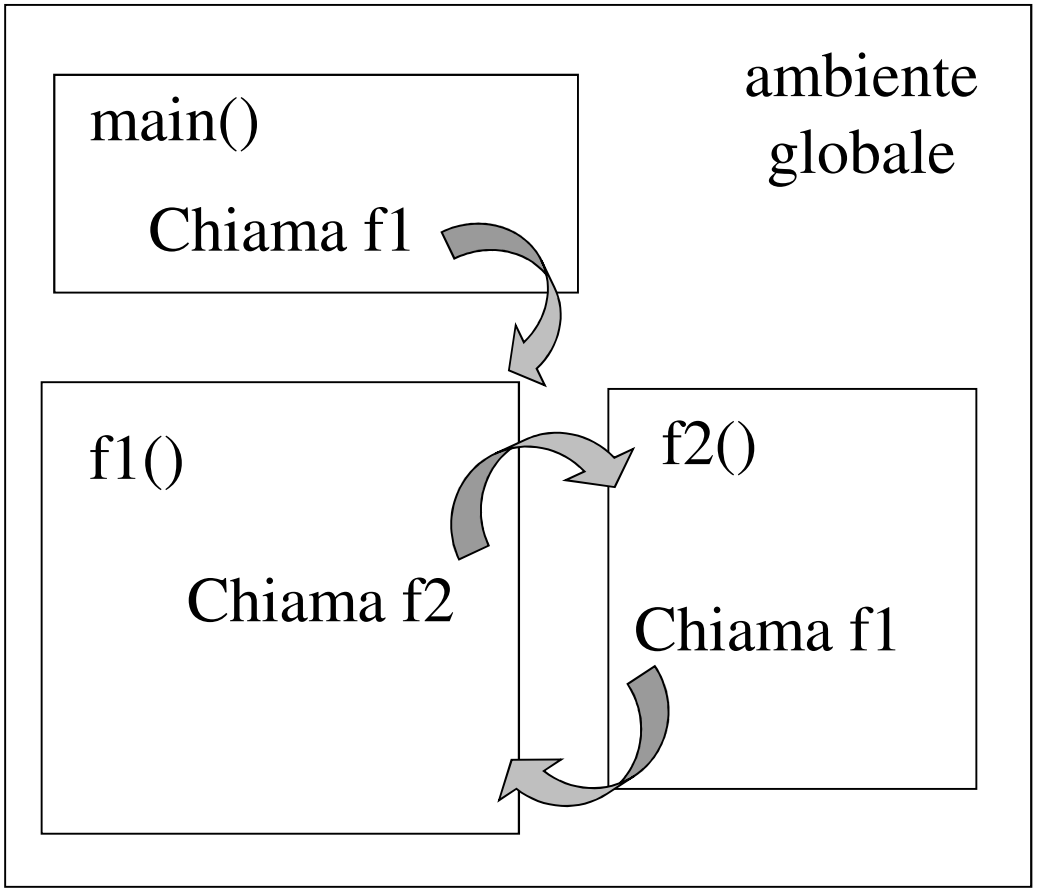
- Una porzione della memoria di lavoro, chiamata **stack** (***pila***)***:*** modalità **LIFO** (Last In First Out) permette al sistema operativo di gestire i processi e di eseguire le chiamate a sottoprogramma
- **Lo Stack Pointer (puntat. alla pila)** è un registro che contiene **l'indirizzo della parola di memoria da leggere nello stack**

SP | 312

- Operazione di ***inserimento:***
  - incremento SP
  - scrittura in parola indirizzata da SP
- Operazione di ***estrazione:***
  - lettura da parola indirizzata da SP
  - decremento SP



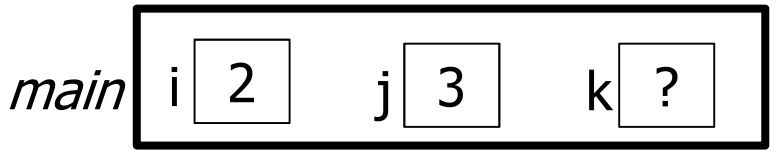
Esempio


ambiente
globale
main()
Chiama f1
f1()
Chiama f2
f2()
Chiama f1


crescita
verso
l'alto
...
r.d.a. di f1(3°)
r.d.a. di f2(2°)
r.d.a. di f1(2°)
r.d.a. di f2(1°)
r.d.a. di f1(1°)
r.d.a. di main()
variabili globali

Esempio di codice: facciamo un disastro!

```
int moltiplica( int x, int y ) {
    int r;
    r = power( x, y);
    return r;
}
                    int power( int b, int e ) {
                        int i, p=1;
                        for( i=1 ; i<=e ; i++ )
                            p = moltiplica(p,b);
                        return p;
                    }
int main() {
    int i=2,j=3,k;
    k=moltiplica(i,j);
    k=power(i,j);
    return 0;
}
```



Perché è necessario?

- In generale, una funzione può essere chiamata un numero imprecisato di volte
- Ogni chiamata a una funzione richiede allocazione di memoria per le sue variabili (automatiche)
  - Il compilatore potrebbe "preparare" un ambiente per ogni funzione definita? In generale no... perché?
- **Le funzioni possono richiamare se stesse** (ricorsione)
  - **Possono quindi esistere più "istanze" di una stessa funzione, "addormentate" in attesa della terminazione di una "gemella" per riprendere l'esecuzione**
- In questo ultimo caso il compilatore non può sapere quanto spazio allocare per le variabili del programma (nei vari ambienti)