Funzioni e Procedure

Alcuni Esempi:

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

main()
{int  A, B;
 A=0;
 B=0;
 h(A, &B); /*B e` un parametro
                di uscita*/
 printf("\n %d \t %d", A, B);
}

void h(int  X, int *Y)
{
 X=X+1;
 *Y=*Y+1;
 printf("\n %d \t %d", X, *Y);
}
```

```
 1   1          (stampa di “h”)
 0   1          (stampa di “main”)
```


Esercizio:

Calcolo delle radici di una equazione di secondo grado.
$Ax^2 + Bx + C = 0$

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

typedef enum {false,true} boolean;

boolean radici(int A, int B, int C,
          float *X1, float *X2);

main()
{ int    A,B,C;
  float  X,Y;
  scanf("%d%d%d%\n",&A,&B,&C);
  if ( radici(A,B,C,&X,&Y) )
            printf("%f%f\n",X,Y);
}
```



```
boolean  radici(int A, int B, int C,
           float *X1, float *X2)
{float D;
 boolean result;

 D= B*B-4*A*C;
 if (D<0) result=false;
 else
  {  D=sqrt(D);
     *X1 = (-B+D)/(2*A);
     *X2=  (-B-D)/(2*A);
         result=true;
  }
 return result;
}
```


Esercizio:

Programma che stampa i numeri primi compresi tra 1 ed n (n dato).

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

typdef enum{false,true} boolean;

int isPrime(int n); /*dichiarazioni*/
int primes(int n);

void main()
{
int n, primi;

  do {
       printf("\nNumeri primi non
              superiori a:\t");
       scanf("%d",&n);
     } while (n<1);

  primi=primes(n);
  if(primi>0)
   printf("\nTrovati %d numeri primi.\n",
            primi);
}
```

```
int isPrime(int n)
{
int max,I,result;
boolean trovato;

if (n>0 && n<4) result=true;
     /* 1, 2 e 3 sono primi */
else if (!(n%2)) result=false;
/* escludi i pari > 2    */
max = sqrt( (double)n );
/* CAST:sqrt ha arg double */
trovato=false;
for(i=3; i<=max && !trovato; i+=2)
 /* provo per tutti i dispari */
 if (!(n%i)){
     result=false;
     trovato=true;
 }
if (!trovato) result=true;
return result;
}
```



```
int primes(int n)
{
 int i,count;

 count=0;
 if (n>=1)
    { printf("%d\t",1);
      count++;
    }
 if (n>=2)
    { printf("%d\t",2);
      count++; }
 for(i=3;i<=n;i+=2)
   if (isPrime(i))
      { printf("%d\t",i);
        count++;
      }
 printf("\n");
 return count;
}
```


Esercizio:

Scrivere una procedura che risolve un sistema lineare di due equazioni in due incognite

$$a_1x + b_1y = c_1$$
$$a_2x + b_2y = c_2$$

$$x = (c_1b_2-c_2b_1) / (a_1b_2- a_2b_1) = XN / D$$
$$y = (a_1c_2- a_2c_1) / (a_1b_2- a_2b_1) = YN / D$$

Soluzione:

```
#include <stdio.h>
void sistema(int A1, int B1, int C1,
         int A2, int B2, int C2,
         float *X, float *Y);

main()
{
  int    A1,B1,C1,A2,B2,C2;
  float  X,Y;

  scanf("%d%d%d%\n",&A1,&B1,&C1);
  scanf("%d%d%d%\n",&A2,&B2,&C2);

  sistema(A1,B1,C1,A2,B2,C2,&X,&Y);
  printf("%f%f\n",X,Y);
}
```



```
void sistema  (int A1, int B1, int C1,
         int A2, int B2, int C2,
         float *X, float *Y)
{
  int    XN,YN,D;
  XN = (C1*B2 - C2*B1);
  D =  (A1*B2 - A2*B1);
  YN = (A1*C2 - A2*C1);
  if (D == 0)
    {if (XN ==  0)
       printf("sistema indeterminato\n");
     else
       printf("sistema impossibile\n");
    }
  else
     { printf("Determinante%d",D);
       *X=(float) (XN) /D;
       *Y=(float) (YN) /D;
     }
}
```

Vettori come parametri di funzioni

In C i vettori sono sempre passati **attraverso il loro indirizzo**:

Ad esempio:

```
#include <stdio.h>
#define MAXDIM 30
typedef enum{false,true} boolean;

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

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

Definizione della funzione:

```
int getvet(int *v, int maxdim);
{ int i=0;
  char s[3];
  boolean fine=false;
  while(i<maxdim && !fine)
    {  printf("%d elemento:\t", i+1);
       scanf("%d", &v[i]);
       printf("Fine?(si/no)");
       scanf("%s",s);
       if (!strcmp(s,"si")) fine=true;
       i++;
```



```
    }
    return(i);
  }
```

- Il vettore è modificato all'interno della funzione e le modifiche sopravvivono all'esecuzione della funzione poiché in C i vettori sono trasferiti ***per indirizzo***.

Struttura di un Programma C

Se un programma fa uso di funzioni/procedure, la sua struttura viene estesa come segue:

```
#include <stdio.h>
/* variabili e tipi globali al programma:
   visibilità nell'intero programma */
tipovar  nomevar, ...;

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

/*main*/
main (void)
{   /* variabili  locali al main:
 visibilità nel solo main */

/*codice del main:  si invocano le Fi */
} /* fine main */

/* definizioni funzioni */
int F1 ( ... )
{  /* parte dichiarativa */
/*codice di F1*/.. }
```


- Le definizioni di funzioni non possono essere innestate in altre funzioni o blocchi.

Variabili locali:

Nella parte dichiarativa di un sottoprogramma (procedura o funzione) possono essere dichiarati costanti, tipi, variabili (detti ***locali*** o ***automatiche***).

```
#include <stdio.h>
char  saltabianchi (void);
main(void)
{char C;
 C = saltabianchi();
 printf("\n%c",C);/* stampa     */
}

char  saltabianchi (void)
{char Car; /* Car e` locale*/
   do
   {  scanf("%c", &Car);}
   while (Car==' ');
   return Car;
 }
```

- Alla variabile Car si può far riferimento solo nel corpo della funzione saltabianchi (***campo di azione***).
- Il ***tempo di vita*** di Car è il tempo di esecuzione della funzione saltabianchi.
- I parametri formali vengono trattati come variabili locali.

Variabili Locali

- Quando una funzione viene chiamata, viene creata una associazione tra l'identificatore di ogni variabile locale (e parametro formale) ed una cella di memoria allocata automaticamente.

Esempio:

```
int f(char Car)
{
int P;
}

main()
{ char C;

f(C);
}
```

- Alla fine dell'attivazione ogni cella di memoria associata a variabili locali viene deallocata. Se la procedura viene attivata di nuovo, viene creata una nuova associazione.
- Non c'è correlazione tra i valori che `Car` assume durante le varie attivazioni della funzione f.

Variabili esterne (o globali)

- Nell'ambito del blocco di un sottoprogramma (oppure nel blocco del main) si può far riferimento anche ad identificatori **globali**, nella ***parte dichiarazioni globali*** del programma.
- Il ***tempo di vita*** delle variabili esterne è pari al tempo di esecuzione del programma (***variabili statiche***).

Esempio:

```
#include <stdio.h>
void  saltabianchi (void);

char C; /* def. variabile esterna */

main()
{
 saltabianchi();
 printf("\n%c",C);   /* stampa C */
}

void  saltabianchi (void)
 {
  do
     {scanf("%c", &C);}
  while (C==' ');
 }
```


Variabili Esterne

Nell'esempio:

- C è una ***variabile esterna*** sia al main che a saltabianchi.
- la definizione di C è visibile sia dalla funzione main che dalla procedura saltabianchi. Entrambe queste unità possono far riferimento alla variabile C.
- E` la ***stessa*** variabile. Ogni modifica a C prodotta dalla funzione, viene "vista" anche dal main.

↣ **possibilità di effetti collaterali**

Effetti collaterali

Si chiama effetto collaterale (***side effect***) provocato dall'attivazione di una funzione la modifica di una qualunque tra le variabili **esterne.**
Si possono avere nei seguenti casi:

- *parametri di tipo puntatore;*
- assegnamento a *variabili esterne*.

Se presenti, le funzioni non sono più funzioni in senso matematico.

Esempio:

```
#include <stdio.h>
int B;
int f (int *A);

main()
{ B=1;
  printf("%d\n",2*f(&B));   /* (1) */
  B=1;
  printf("%d\n",f(&B)+f(&B)); /* (2) */
}

int f (int * A)
{ *A=2*(*A);
  return *A;
}
```


- Fornisce valori diversi, pur essendo attivata con lo stesso parametro attuale. L'istruzione (1) stampa 4 mentre l'istruzione (2) stampa 6.

Effetti Collaterali

Esempio:

```
int V=2;

float  f (int X)
{
 V=V*X; /* origine side effect */
 return (X+1);
}

int v ()
{ return(V);
}

main()
{  int B=2;
   printf("%f",v()+f(B));
   V=2;
   printf("%f",f(B)+v());
 }
```

In questo caso:

$$v()+f(B) \neq f(B) +v()$$

Eliminazione degli Effetti Collaterali:

Per evitare effetti collaterali in funzioni occorre:

- non avere parametri passati per indirizzo nelle intestazioni di funzioni;
- non introdurre assegnamenti a variabili esterne nel corpo di funzioni.

Variabili esterne & passaggio dei parametri

Le variabili esterne rappresentano un modo alternativo ai parametri per far ***interagire*** tra loro le varie unità di programma.

Vantaggi:

- Evitano lunghe liste di parametri (da copiare se passati per valore).
- Permettono di restituire in modo diretto i risultati al chiamante.

Svantaggi:

- I programmi risultano meno leggibili (rischio di errori).
- Interazione meno esplicita tra le diverse unità di programma (effetti collaterali).
- Generalità, riusabilità e portabilità diminuiscono.

Visibilità degli Identificatori e Tempo di Vita

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

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

Visibilità degli Identificatori

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

Regole di visibilità degli identificatori in C:

1. il campo di azione della dichiarazione (o definizione) di un identificatore ***esterno*** va dal punto in cui si trova la dichiarazione/definizione fino alla fine del file sorgente, a meno della regola (3);
2. il campo di azione della dichiarazione (o definizione) di un identificatore ***locale*** è il blocco (o la funzione) in cui essa compare e tutti i blocchi in esso contenuti, a meno di ridefinizioni (v. regola (3));
3. quando un identificatore dichiarato in un blocco P è ridichiarato (o ridefinito) in un blocco Q racchiuso da P, allora il blocco Q, e tutti i blocchi innestati in Q, sono esclusi dal campo di azione della dichiarazione dell'identificatore in P (*overriding*).

- Il campo di azione di ogni identificatore è determinato ***staticamente***, dalla struttura del testo del programma (***regole di visibilità lessicali***).

Visibilità degli Identificatori

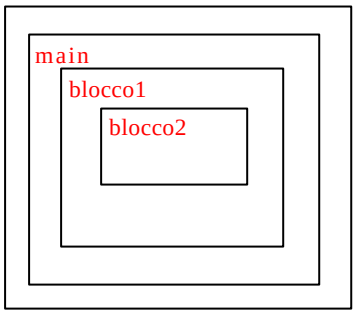
Dalle regole precedenti, possiamo concludere che:

- Per una ***variabile locale*** (e per i ***parametri formali***), dichiarata in una funzione, il campo di azione è la funzione stessa.
- Per una ***variabile esterna*** (così come per le funzioni che sono tutte dichiarate esternamente) il campo di azione va dal punto in cui si trova la dichiarazione fino alla fine del file sorgente.

Esempio:

```
#include <stdio.h>
void main()
{ int i=0;
  while (i<=3)
  { /* BLOCCO 1 */
    int j=4; /* def. locale al blocco 1*/
    j=j+i;
    i++;

    { /* BLOCCO 2: interno al blocco 1*/
      float i=j; /*locale al blocco 2*/
      printf("%f\t%d\t",i,j);
    }
    printf("%d\t\n",i);
  }
}
```



Esempio:

```
#include <stdio.h>
int   X=0;
 void P1 ();  /* superflua perché non
                    chiamata dal main */
void  P2 ();

main()
{ X++;
  P2();
}

void  P1()
{
  printf("\n%d",X);
}

void  P2()
{float  X;
  X=2.14;
  P1();
}
```

Stampa il valore 1.

➳ Con regole di visibilità dinamiche (non adottate dal C, ma ad esempio in LISP), stamperebbe il valore 2.14.

Tempo di vita delle variabili

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

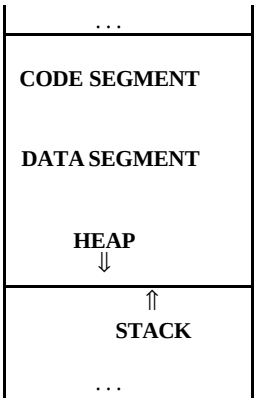
In C:

- Variabili ***esterne*** sono allocate all'inizio del programma e vengono distrutte quando il programma termina ➠ il tempo di vita è pari al tempo di esecuzio ne del programma.
- Variabili ***locali*** e ***parametri formali*** delle funzioni sono allocati ogni volta che si invoca la funzione e distrutti al termine della funzione
- Il ***tempo di vita*** é quindi pari alla durata dell'attivazione della funzione in cui compare la definizione della variabile
- Variabili ***dinamiche*** hanno un tempo di vita pari alla durata dell'intervallo di tempo che intercorre tra la *malloc* che le alloca e la *free* che le dealloca.

La macchina astratta C: modello a tempo di esecuzione

Aree di memoria:

- Codice (*Code segment*)
- Area dati globale (statica): *Data segment*
- *Heap* (dinamica)
- *Stack* (dinamica)



Stack

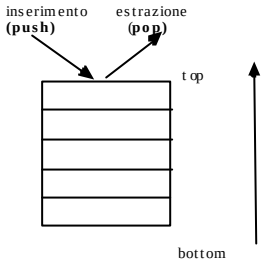
In C le attivazioni delle funzioni sono realizzate utilizzando l'area di memoria "stack" in cui risiede una struttura dati gestita seguendo una disciplina a pila: lo ***stack***:

E` una struttura dati su cui è possibile eseguire due operazioni:

- inserimento di un elemento (*push*)
- estrazione di un elemento (*pop*)

Politica di gestione dello stack:

l'ultimo elemento inserito è il primo ad essere estratto (politica LIFO, Last In First Out).



- Ad ogni chiamata di sottoprogramma viene creato un elemento (**record d'attivazione**) ed inserito (***push***) in cima allo stack.

Record d'attivazione

Un record d'attivazione contiene le informazioni relative ad una specifica chiamata di funzione/procedura.

In particolare:

1) **nome della funzione** attivata e riferimento al codice;
2) **punto di ritorno** al chiamante (***return address***): è l'indirizzo dell'istruzione da eseguire al termine della attivazione;
3) riferimento di **catena statica** (***static link***): é un riferimento all'ambiente visto "staticamente" dal sotto-programma (variabili esterne);
4) **parametri formali** e loro legame con quelli attuali (se per indirizzo);
5) **variabili locali**;
6) riferimento al record di attivazione precedente sulla pila (**catena dinamica**, o ***dynamic link***): è un riferimento all'ambiente del chiamante.

Al termine dell'esecuzione (**return**), il record di attivazione viene deallocato dallo stack. (operazione di ***pop***)

Record di Attivazione

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

Esempio:

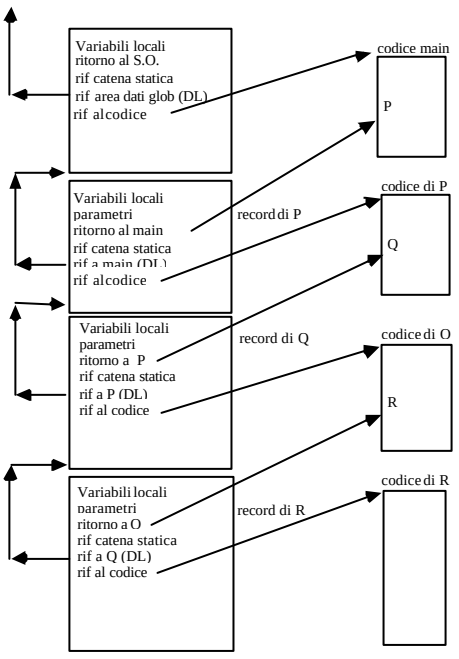
```
#include <stdio.h>

void R(int A)
{
  printf("Valore: %d\n", A);
}

void Q(int A)
{
  R(A);
}

void P()
{
  int a=10;
  Q(a);
  return;
}

main()
{
  P();
}
```

Record di Attivazione

Catena Dinamica:

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

Attivazioni: (S.O. -->) main –> P --> Q –> R

Catena statica:

La ***catena statica*** indica dove cercare (in quale area) i riferimenti per le variabili non locali (***variabili esterne***).

- **In C**: le funzioni non possono contenere definizioni di altre funzioni ⇒ la catena statica fa sempre riferimento all'area dati globale, contenente, ad esempio, le variabili esterne.

Esempio:

```
#include <stdio.h>

void prova(int *a, int b, int n);

main()
{
 int c[3], d;
 c[0] = 100; c[1] = 15;
 c[2] = 20;  d = 0;
 printf("Prima: %d,%d,%d,%d\n",
     c[0],c[1],c[2],d);
 prova(c,d,3);
 printf("Dopo: %d,%d,%d,%d\n",
     c[0],c[1],c[2],d);
}

void prova(int *a, int b, int n)
{
 int i;
 for (i = 1; i < n; i++) a[i] = b;
 b = a[0];
}
```

Il risultato dell'esecuzione di questo programma è:

Prima: 100,15,20,0
Dopo: 100,0,0,0 **d?**