

Funzioni

Programmazione strutturata

La scrittura di un programma di grandi dimensioni, si basa sulla possibilità che offre un linguaggio di programmazione di

- implementare facilmente la scomposizione di problemi in sottoproblemi
- la possibilità di suddividere un programma su più files

Il C è sotto entrambi questi aspetti uno dei linguaggi più flessibile.

In particolare la scomposizione di problemi in sottoproblemi, viene implementata in C scomponendo un problema in tante funzione.

In particolare un programma C

Programmazione strutturata

Un programma in C è composto da uno o più files, ognuno dei quali contiene zero o più funzioni delle quali una è obbligatoriamente la funzione main. Le funzioni non possono essere innestate ma possono chiamarsi una con l'altra

Le funzioni operano sulle variabili del programma. Nel C sono definite delle regole di visibilità che consentono di dire con assoluta precisione tutte le variabili che sono accessibili in un determinato punto.

Definizione e dichiarazione di funzione

Per poter usare una funzione dobbiamo

- **definire la funzione**, cioè specificarne le istruzioni di C che definiscono ciò che la funzione deve eseguire,
- **dichiarare la funzione**, perchè le altre funzioni del programma "siano a conoscenza" della presenza di quella funzione.

L'uso di una funzione avviene attraverso la **chiamata** ad una funzione.

Mentre in genere vedremo che è bene tenere separate la definizione e la dichiarazione di una funzione, in certi stili di programmazione è possibile far coincidere la dichiarazione con la definizione.

Definizione di funzione

La definizione di una funzione e' il codice C che la implementa.

La sintassi delle definizioni di una funzione è data da:

Sintassi

$\langle \text{tipo} \rangle \langle \text{nome funzione} \rangle (\langle \text{lista parametri} \rangle) \{ \langle \text{dichiarazioni} \rangle \langle \text{istruzioni} \rangle \}$

L'intestazione è tutto ciò che precede la parentesi {. Il **corpo** della funzione è tutto ciò che è incluso tra le due {}

$\langle \text{lista parametri} \rangle$ contiene una lista di parametri che rappresentano l'interfaccia di input della funzione.

Definizione di funzione

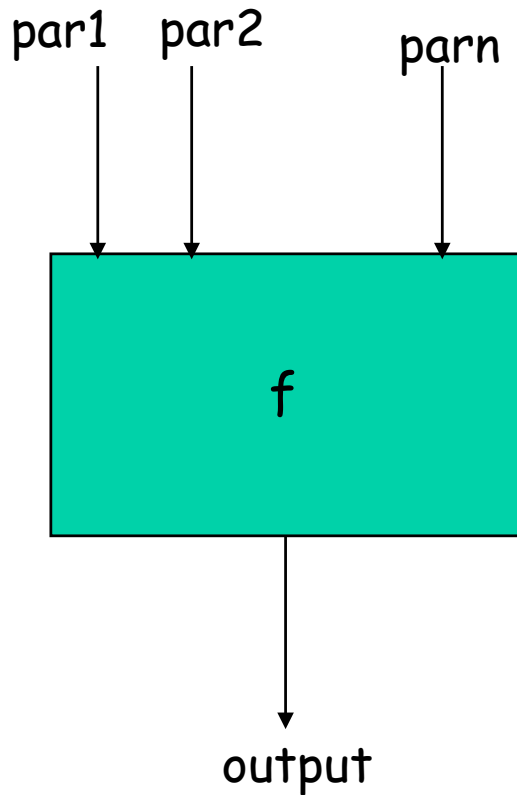
Esempio

```
int fattoriale (int n){  
    int i=1, product =1;  
    for (i=2;i<=n,++i)  
        product*=i;  
    return product;  
}
```

Il primo int indica che il valore restituito dalla funzione viene convertito a int prima di essere passato. La lista dei parametri in questo caso è formata da int n che specifica che la funzione riceve in input un valore intero. Il parametro formale n può essere usato come una normale variabile all'interno del corpo della funzione.

Scatole Nere

Le funzioni possono essere pensate come delle scatole nere, che eseguono determinate operazioni e che comunicano con l'esterno attraverso i parametri e il valore restituito in output.



Quando un'altra funzione ha bisogno di usare la funzione *f*, esegue una chiamata a *f* passandole in input i parametri ai quali vuole applicare la funzione *f*, nello stesso ordine in cui sono specificate nella definizione della funzione *f*.

```
/* calcola i fattoriali da 1 a MAX e li stampa */  
for(i=1;i<=MAX,++i){  
    fattoriale(i);  
    printf("%d",i);  
}
```

Altri Esempi di definizioni

```
void stampa (void){  
    printf(".....");  
    printf("*****");  
    printf("+++++++");  
}
```

```
double stampa (int n, double x, char c){  
    .....  
    .....  
    return x*a;  
}
```

```
void inutile (void){}
```

Una funzione potrebbe anche **non restituire un valore** in output. In questo caso il suo **tipo** si dichiara come **void**. Potrebbe anche non avere alcun **parametro formale**. Anche in questo caso viene usato **void**. Se il tipo della funzione non viene specificato il compilatore assume il tipo **int**. Ma è buona regola specificare sempre un tipo per ogni funzione.

Prototipi di funzioni

Per poter usare una funzione f in una altra funzione g è necessario che g sia "a conoscenza" dell'esistenza di f . In C ciò avviene mediante la dichiarazione della funzione mediante ciò che viene chiamati normalmente il **prototipo** della funzione.

Esattamente come per la variabili è buona regola dichiarare le funzioni prima di usarle. Ciò avviene sempre attraverso i prototipi delle funzioni.

In un prototipo specifichiamo

- il nome della funzione
- Il tipo di dato restituito dalla funzione
- i tipi dei parametri nell'ordine esatto in cui compaiono nella definizione della funzione

Prototipi di funzioni: Esempi

```
double pow(double, double);
```

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

Sintassi

<prototipo funzione> ::= <tipo> <nome funzione> (<lista parametri>);

```
void funzione(char c, int i);
```

equivalente a

```
void funzione (char, int);
```

Buona norma di programmazione

È ben dichiarare i prototipi di tutte le funzioni usate in un programma in un file prima della definizione del main e subito dopo le direttive `include` e `define` del preprocessore.

Istruzione return

L'istruzione return si usa nelle funzioni per

- restituire il controllo all'ambiente chiamante
- per permettere alla funzione di restituire l'output.

Esempio . Una funzione che calcola il valore assoluto

```
double val_ass (double x){  
    if (x>=0.0)  
        return x;  
    else return -x;  
}
```



Non ha bisogno di parentesi

Chiamata di funzione e ordine dei parametri

Supponiamo di avere una funzione `cmp` che riceve due interi e stampa un messaggio di errore se la radice quadrata del primo è maggiore del secondo.

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

prototipo

```
void cmp(int, int);
```

prima chiamata

```
int main(void) {
```

```
int n,m;
```

```
...../* legge m,n e li modifica */
```

```
cmp(m,n);
```

```
cmp(n,m);
```

```
}
```

seconda chiamata

```
void cmp(int a, int b) {
```

```
    if sqrt(a) > b printf("ERRORE");
```

```
}
```

definizione

Esempio

Scrivere una funzione che dato un intero n calcoli e stampi la radice quadrata di tutti gli interi da 1 a n usando l'algoritmo visto prima.

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

```
double rad_quad(int); /* prototipo della funzione rad_quad*/
```

```
int main(void) {
```

```
    int n;                /*input*/
```

```
    printf("Immetti n:");
```

```
    scanf("%d");         /* legge l'input */
```

```
    for (i=1;i<=n,++i)
```

```
        printf("La radice quadrata di %d e`%f",i,rad_quad(i));
```

```
}
```

Esempio

```
double rad_quad(int a){
/* [Pre: a=A] */
/* [Post:rad_quad = radice quadrata di a secondo
      l'algoritmo Newton Raphson */

    double x0,x1;
    x0 = 1;           // inizializzo x0
    x1 = 0.5*(x0+(a/x0)); // inizializzo x1
    while (x0 != x1){ // calcola la radice
        x0 = x1;     // quadrata
        x1 = 0.5*(x0+(a/x0));
    }
    return x0;
}
```

Precondizione e Postcondizione

Ogni volta che scriviamo e implementiamo una funzione dobbiamo specificarne l'interfaccia con l'esterno. Vuol dire specificare

- precondizione sui parametri e
- postcondizione sull'output.

Cio e' bene per una verifica di correttezza, per la leggibilità e soprattutto anche per la gestione degli errori.

Esempio

```
int sum (int n) {  
  /* [Pre: n>0]  
   * [Post: sum =  $\sum_{i=1..n} i^2$  ]  
  
   if (n<=0) printf("Error");  
   else ..... /* calcola la somma dei quadrati */  
}
```

Parametri formali e attuali

Come abbiamo visto le funzioni prevedono in fase di definizione la specifica dei parametri. Tali parametri sono detti parametri formali in quanto specificano l'ordine con cui i parametri sono passati all'atto di una chiamata e perchè sono usati all'interno della definizione di una funzione.

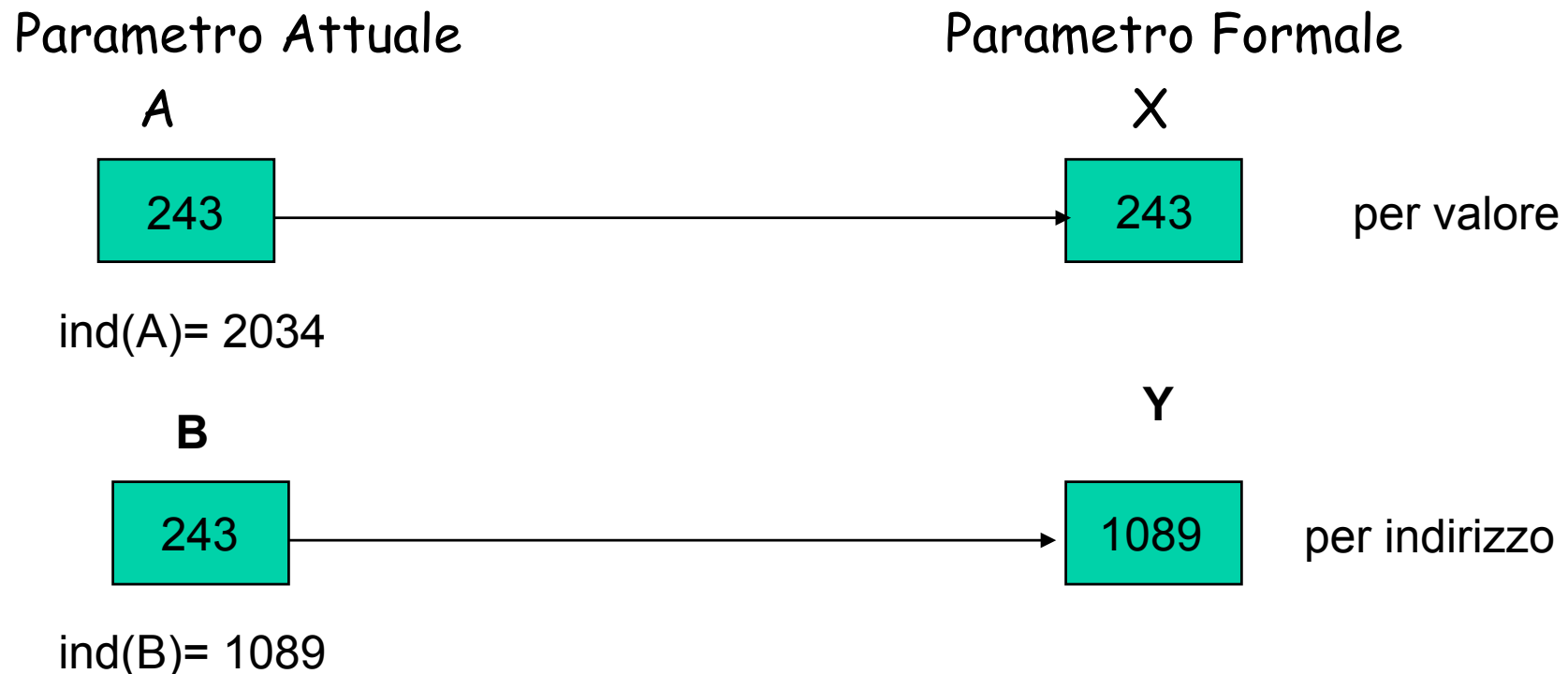
I parametri usati all'interno di una chiamata di funzione invece sono detti parametri attuali in quanto indicano i valori attuali sulla quale una funzione è chiamata a lavorare.

Che un parametro attuale in una chiamata ad una funzione possono essere un' espressione e non sempre sono solo variabili.

Al contrario i parametri formali sono solo variabili

Passaggio dei parametri per indirizzo

A parte il passaggio dei parametri per valore esiste un'altra metodologia detta: **passaggio** dei parametri **per indirizzo** o **referenza**. L'idea è quella di permettere la **modifica** del valore del parametro **attuale**. Alla funzione viene passato non il valore del parametro attuale, ma il suo l'indirizzo in memoria.



Passaggio dei parametri per indirizzo

In C la modalità di passaggio dei parametri è sempre quella per valore. È possibile realizzare però l'effetto di un passaggio per indirizzo nel seguente modo:

- utilizzando il costruttore di tipo puntatore per la definizione dei parametri formali, che vedremo più avanti.
- usando l'operatore di dereferenziazione di puntatore all'interno del corpo della funzione (* o ->)
- passando al momento della chiamata della funzione come parametro attuale, un indirizzo di variabile (usando eventualmente l'operatore di indirizzo &)

Blocchi di istruzioni

Abbiamo visto che in *C* un'istruzione composta è una sequenza di più istruzioni. A volte in un programma è bene poter distinguere blocchi di istruzioni che eseguono compiti diversi.

In *C* ciò è si rende possibile racchiudendo tra parentesi `{}` il blocco di istruzioni che ci interessa. In particolare un blocco in *C* si distingue da una semplice istruzione composta in quanto dovrebbe includere almeno una dichiarazione

Ciò porta alla definizione di

- blocchi di istruzioni annidati
- blocchi di istruzioni paralleli

Blocchi annidati e paralleli possono essere a loro volta composti secondo schemi arbitrariamente complessi

Blocchi di istruzioni

```
#include <stdio.h>
/* parte dichiarativa globale */
int g1;g2;          /* variabili */
char g3;           /* variabili */
int f1(int, int);  /* prototipo funzione */

int main(void){
    int a,b;       /* variabili del main */
    int f2(int,int); /* prototipo f2 */
    {             /* inizio blocco 1 nel main */
        char a,c;
        .....
        {         /* inizio blocco 2 annidato nel blocco */
            float a;
            .....
        }         /* fine blocco 2 */
    }             /* fine blocco 1 */
}                /* fine main */
int f1(int par1, int par2){ /* definizione f1*/
    int d;
    {             /* inizio blocco 3*/
        int e;
        .....
    }             /* fine blocco 3 */
    {             /* inizio blocco 4 parallelo blocco 3*/
        int d;
        .....
    }             /* fine blocco 4*/
}                /* fine funzione f1 */
int f2(int par1, int par2){ /* definizione f2*/
    int f;
    ...
}                /* fine definizione f2*/
```

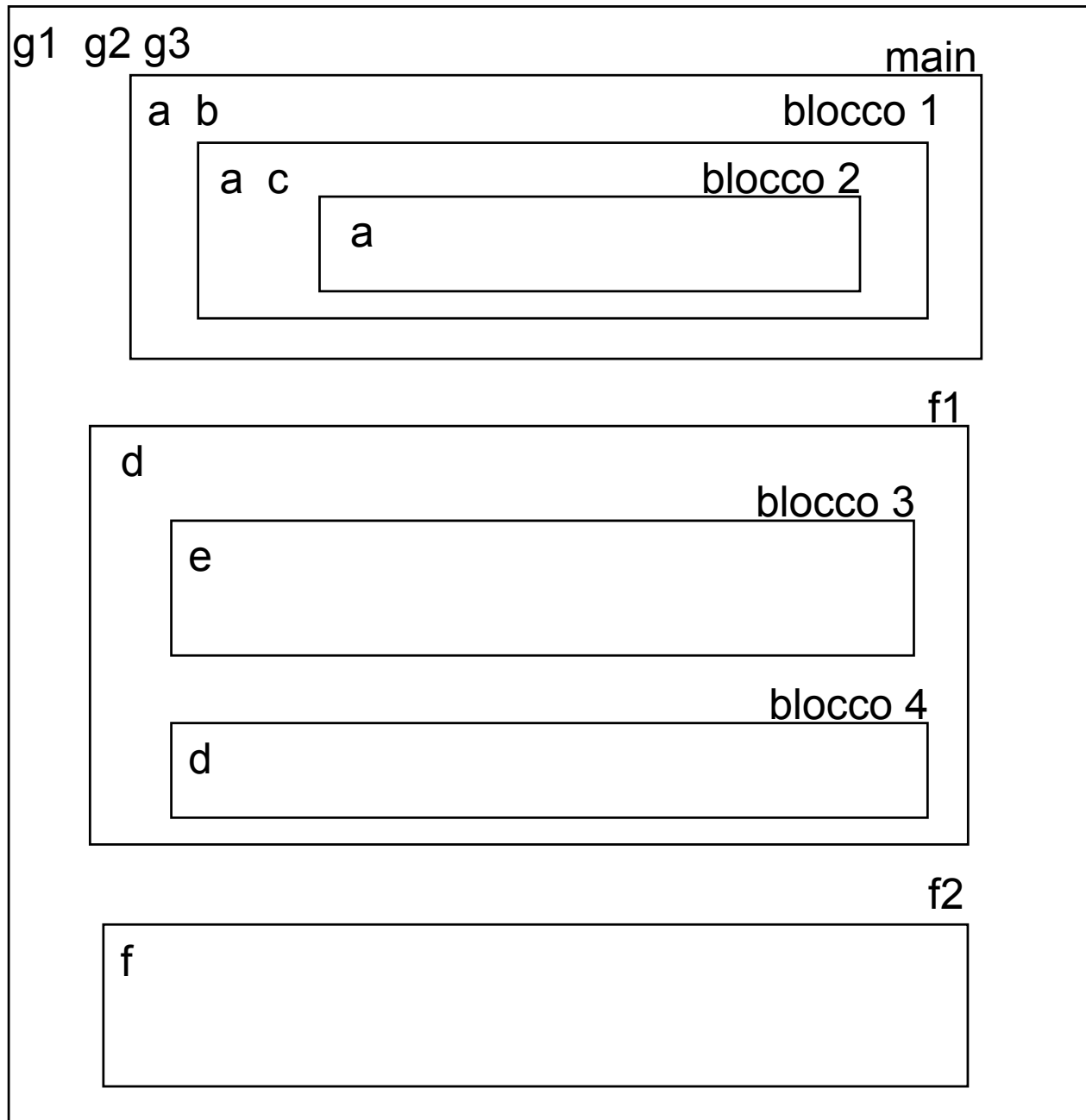
Visibilità delle variabili

Ambienti:

- **Ambiente globale** è l'insieme dei dati dichiarati nella parte dichiarativa globale
- **Ambiente locale** di una funzione è insieme di elementi dichiarati nella parte dichiarative e nell'intestazione
- **Ambiente di blocco** l'insieme dichiarati nella parte dichiarativa di un blocco

Il concetto di ambiente permette di usare gli identificatori con molta flessibilità. Infatti è permesso usare gli **stessi identificatori** anche con significati diversi **purché siano in ambienti diversi**

Modello a Contorni di un programma



Regole di visibilità

1. (Globale) gli elementi che costituiscono l'ambiente globale di un programma possono essere "visti" cioè usati e referenziati tramite identificatore da tutte le funzioni. Se internamente ad una funzione o ad un blocco esistono più definizioni dello stesso identificatore, la definizione valida è quella più vicina al punto di uso dell'identificatore
2. (Funzione) Gli elementi che costituiscono l'ambiente locale di una funzione possono essere "visti" dalle istruzioni proprie della funzione e di blocchi in essa contenuti. Se in un blocco esiste una definizione dello stesso identificatore, quella valida è quella più vicina al punto di utilizzo dell'identificatore.
3. (Blocchi) Gli elementi di un blocco possono essere visti dalle istruzioni proprie del blocco e dalle istruzioni di blocchi in esso contenuti. In caso di conflitto di definizione, quella più vicina al punto d'uso è quella valida

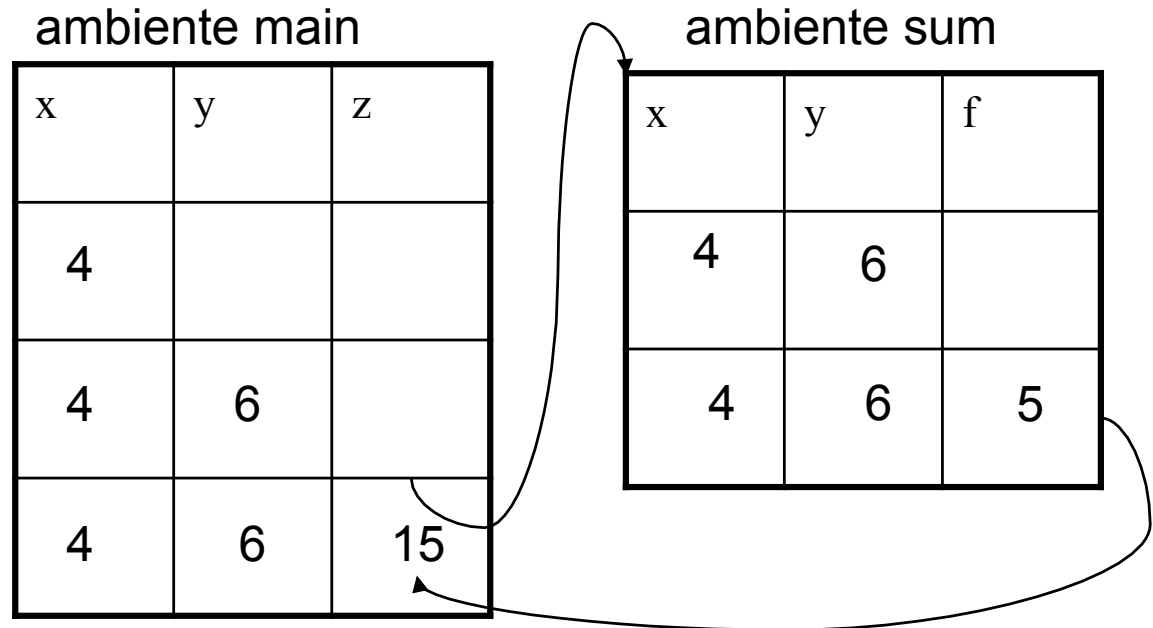
Regole di visibilità

- il **main** accede alle variabili g_1, g_2, g_3 e quelle locali a, b . Il main può inoltre chiamare f_1 e f_2
- il blocco1 accede alle variabili g_1, g_2 e g_3 alla variabile locale al main b e alle variabili locali al proprio ambiente a e c . Blocco1 può chiamare f_1 e f_2
- blocco2 accede alle variabili g_1, g_2, g_3 alle variabile b locale al main e alla variabile c del blocco 1 e alle variabili a del proprio ambiente. Può chiamare f_1 e f_2 .
- f_1 accede alle variabili g_1, g_2, g_3 e a quella locale d . Può chiamare f_2 e se stessa. Non accede alle variabili a, b, c ed f .
- blocco3 accede alle variabili g_1, g_2, g_3 alla variabili d locale a f_1 e alla variabile e del proprio ambiente. Può chiamare sia f_1 che f_2 .
- blocco4 accede alle variabili g_1, g_2, g_3 e alla variabile d del proprio ambiente. Non accede alla variabile e del blocco 3. Può chiamare f_1 e f_2 .
- f_2 accede a g_1, g_2, g_3 e a quella locale f . Può chiamare f_1 e se stessa.

Ambiente ed esecuzioni di funzioni

Abbiamo visto che la conseguenza di una chiamata di funzione è l'attivazione di un ambiente di memoria o stato della funzione (e di una corrispondente **allocazione di memoria**) che viene rilasciata nel momento in cui la funzione restituisce il valore dovuto con return oppure termina la sua esecuzione.

```
int sum (int, int);
int main(void) {
    int x,y,z;
    x= 4;
    y = 6;
    z = sum(x,y);
}
int sum (int x, int y) {
    int f;
    f =5;
    return x+y+f;
}
```



Classi di memorizzazione

Tutte le variabili in C hanno due attributi

- tipo
- Classe di memorizzazione.

Ci sono 4 classi di memorizzazione in C

- automatic (auto)
- esterna (extern)
- statica (static)
- registro (register)

Classi di memorizzazione: automatic

Le variabili definite all'interno delle funzioni o di blocchi sono per default automatic.

La parola chiave **auto** si può usare per specificarne la classe

```
auto int b;  
auto float x;
```

Significato.

All'ingresso in un blocco il sistema **alloca la memoria per le variabili automatiche**, tali variabili vengono considerate locali al blocco.

All'**uscita** dal blocco, il sistema **libera la memoria** assegnata alle variabili automatiche causando quindi la perdita dei loro valori.. Se si rientra nel blocco allora la memoria viene ri-allocata, senza però recuperare i valori precedenti.

Classi di memorizzazione: extern

Un metodo per trasferire informazioni tra i blocchi è quello di usare variabili esterne, cioè visibili a tutti. Quando una variabile è definita al di fuori di una funzione le viene allocata una memoria permanente, che non viene mai rilasciata. Tutte le dichiarazioni globali sono per default extern.

Esempio: Nello stesso file

```
int a=1,b=2,c=3;
int f (void);
int main (void){
    print("%d",f());
}
int f(void) {
    int b,c;          /* b e c sono locali */
    a=b=c=4;         /* b e c globali sono coperte */
    return (a+b+c);
}
```

Avremo potuto usare

```
extern int a=1,b=1,c=1; /* problemi con alcuni compilatori */
```

Classi di memorizzazione: extern

Variabili esterne in programmi su più file

File 1.

```
int a=1,b=2,c=3;
int f void();
int main (void){
    print("%d",f());
}
```

File 2

```
int f(void) {
    extern int a;      /* a è esterna .. cerca in altro file */
    int b,c;          /* b e c sono locali */
    a=b=c=4;
    return (a+b+c);
}
```

Problema ?? . Modifichiamo un parametro globale in una funzione senza passarlo come parametro. Perdiamo la modularità e leggibilità del programma

Classi di memorizzazione: extern

Le variabili extern non sono mai ne classe automatic ne register. Potrebbero essere static, ma in casi particolari che vedremo avanti.

Le funzioni hanno sempre classe extern. Quindi sia il prototipo che la definizione di una funzione possono essere preceduti dalla parola chiave extern

Classi di memorizzazione: register

La classe di memorizzazione **register** serve per indicare al compilatore la memorizzazione di una variabile in un registro di memoria ad alta velocità.

Tale memorizzazione avviene solo nel caso in cui vi sia un registro libero che il sistema può assegnare.

Il motivo principale nell'uso di tale classe è aumentare l'efficienza e la velocità di esecuzione di un programma.

La strategia migliore è quella di identificare poche variabili alle quali viene fatto più di frequente riferimento e di dichiararle **register**.

Il tipico uso è quello di un indice di un ciclo `for`

```
register int i;  
for (i=1;i<=230; ++i) {....}
```

Se non si specifica un tipo si dichiara un `int`
`register i` è equivalente a `register int i`

Classi di memorizzazione: static

La classe di memorizzazione **static** ha due utilizzi importanti e distinti:

- permette ad una variabile locale, per esempio ad un blocco di mantenere il suo valore precedente al momento di rientro in un blocco

Esempio

```
void f(void) {  
    static int cnt =0;  
    ++cnt;  
    if (cnt %2==0)  
        .....  
    else ...  
}
```

Alla prima chiamata il valore di cnt viene inizializzato e all'uscita della funzione viene mantenuto. Alla chiamata successiva viene usato il precedente valore e incrementato. cnt e` privata ad f e non puo essere vista al di fuori

Classi di memorizzazione: static

Il secondo uso della classe di memorizzazione **static** è legato alle dichiarazioni esterne e fornisce un meccanismo per rendere private una serie di variabili (esterne) che altrimenti sarebbero visibili a tutti gli ambienti.

A prima vista sembrerebbe inutile dichiarare stati variabili esterne che sempre mantengono il loro valore in memoria. Il C invece con la classe `extern static` consente di dichiarare delle variabile che sono "visibili" globalmente solo a partire da un certo punto in poi del programma e non accessibili dalla parte del programma che le precede.

Esempio

```
void f(void) {
    ....          /* qui v non è visibile */
}
static int v;    /* variabili esterne statica */

void g(void) {   /* a partire da qui v è visibile */
    ...
}
```

Classi di memorizzazione: static

Se una funzione viene dichiarata e definita static allora e' visibile solo nel file in cui e' definita.

Esempio

```
static int g (void);  
void f(void){  
    int z= g();  
    ....  
}
```

```
static int g(void){    /* g non è visibile al di fuori di questo  
                       file */  
...  
}
```

Utilizzo di asserzioni

Il C mette a disposizione un file di intestazione `<assert.h>` che permette di accedere alla macro `assert` che consente di interrompere un programma nel caso in cui certe proprietà non si verifichino. L'uso più frequente della `assert` è quello di verificare che i parametri di una funzione o procedura verifichino certe proprietà (La Precondizione).

Esempio

```
int f (int a, int b){  
/* [Pre a <0 , -1 <= b <= 10 ] */  
  
assert(a <0) ;  
assert(b >= -1 && b <=10) ;  
.....  
}
```

Se l'espressione passata ad `assert` risulta falsa allora il programma viene interrotto e stampa un messaggio in output.

L'uso delle `assert` è considerato una buona norma di programmazione.

Problema 1

Problema:

si consideri la seguente funzione

```
int distanza(int a[], int b[] int dim)
```

```
/* pre: dim è la lunghezza di a e b;
```

```
    a è ordinata decrescentemente
```

```
    b è ordinata crescentemente
```

```
    a[0] >= b[0]
```

```
    A[dim-1] < b[dim-1] */
```

```
/* Post: se distanza(a,b,dim)=i allora a[i] >= b[i] e a[i+1] < b[i+1] */
```

Esempio:

Descrivere l'idea e le variabili usate, l'inizializzazione, il corpo dell'iterazione, la terminazione, la verifica di terminazione e l'invariante.