Lezione 8 – Funzioni in C

Programmazione

Modulo 4 – Sottoprogrammi

Unità didattica 1-5 – Programmazione Imperativa in C

Marco Anisetti

Università degli Studi di Milano - SSRI

Concetto di Funzione

In modo astratto possiamo dire che un programma computa una funzione che mappa input nell'output.

Concretamente il programma è una funzione ovvero al funzione **main()**

Così come, in matematica, le funzioni si possono **comporre**, allo stesso modo i programmi possono invocare altre funzioni al fine di eseguire la computazione richiesta.

Ne abbiamo già invocate molte tipo printf(), scanf(), sqrt()

Non ne abbiamo mai definite di nostre oltre alla indispensabile main()

Sottoprogrammi

Sottoprogrammi: brani di codice abbastanza importanti da avere un nome, dei dati e nel caso dei risultati, come i programmi.

Il programma viene suddiviso in sottoprogrammi, come un problema in sottoproblemi

Tali sottroprogrammi vengono realizzati come:

- **Procedure:** sottoprogrammi che non generano un valore di ritorno.
- **Funzioni:** sottoprogrammi che generano un valore di ritorno

In C questa distinzione non è così evidente, si parla sempre di funzioni anche se non ritornano un valore.

Suddivisione del programma

Obiettivi: dividere un programma in brani di codice che siano più facili da **capire e da modificare**

- **non ripetere** brani di codice identici o molto simili
- **riutilizzare** brani di codice in programmi diversi

Strumento per *incapsulare* una data computazione che abbia senso compiuto in sé stessa.

Ideale avere unità funzionali indipendenti

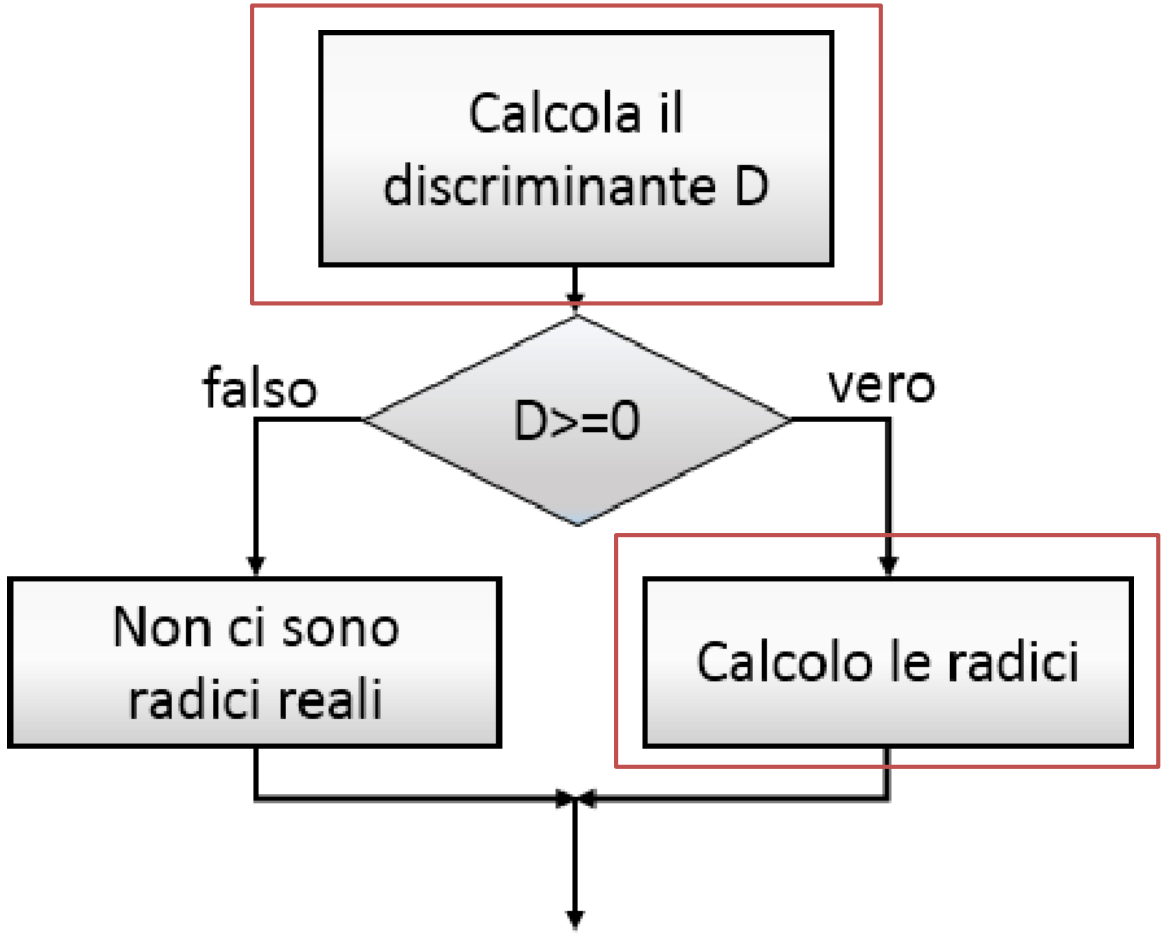
Esempio

- Data l'equazione $ax^2 + bx + c = 0$ con a, b e c presi come input, scrivere il programma che calcola il valore di x

- Già visto tra gli esercizi pensarlo a sottoprogrammi

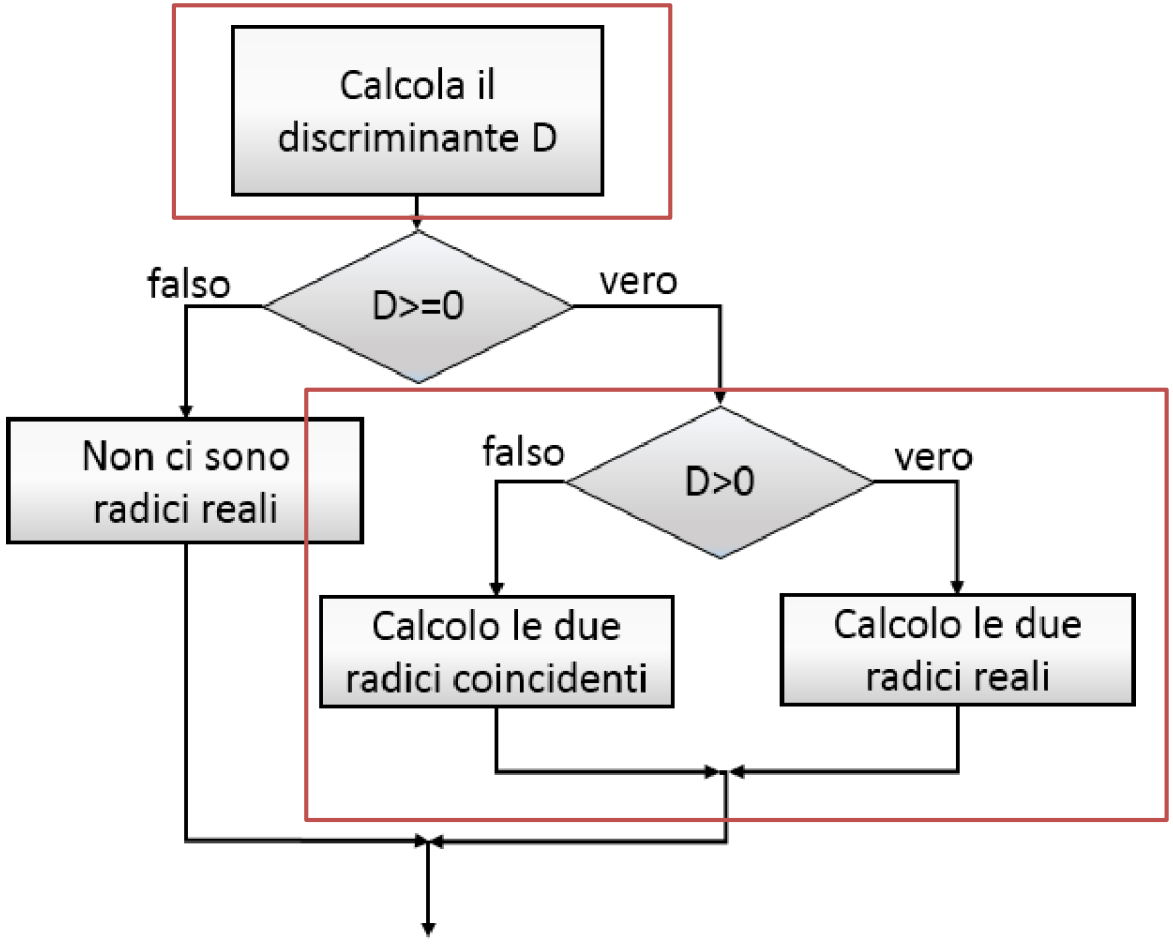
Scomposizione in sottoprogrammi

Calcolo delle soluzioni reali di $ax^2+bx+c=0$
Prima approssimazione della soluzione in sottoprogrammi



Scomposizione in sottoprogrammi

Versione più dettagliata



Dichiarazione di una funzione

- Come nelle dichiarazioni delle variabili:

tipo nome(tipo$_1$ param$_1$,...,tipo$_n$ param$_n$);

- Per le funzioni si chiama dichiarazione del **prototipo**
- Possono esistere funzioni che non ritornano nulla
- In tal caso non posso omettere il tipo di ritorno e basta (il c interpreterebbe comunque come un ritorno di un intero) devo indicarlo in qualche modo

void

- Se voglio indicare l'assenza di un ritorno di una funzione DEVO indicare come tipo il **void**
- **Ovvero il tipo che non ha valore associato**

- void posso usarlo (ma non è obbligatorio) anche per indicare che una funzione non hanno bisogno di parametri

Una funzione che torna void è in sostanza una procedura

Definizione di una funzione

Definizione: dichiaro + implemento

tipo nome(tipo$_1$ param$_1$,...,tipo$_n$ param$_n$)
{
<implementazione funzione>
}

Uso: chiamo la funzione

Esempio **Definizione**:

```
void stampa(void) {
 printf("Ciao\n");
 return;
}
```

Esempio **Uso:**

```
int main(void) {
 stampa();
return 0;
}
```

Dichiarazione Definizione e uso

```
int main(void) {
 stampa(); // Uso: Implicitamente int
 return 0;
}

void stampa() { // Definisco: Esplicitamente void
 printf("Ciao\n");
return;
}
```

La uso prima e la definisco dopo che succede?
N.B. definisco è dichiaro + implemento

Dichiarazione Definizione e uso

```
int main(void) {
 stampa(); // Uso: Implicitamente int
 return 0;
}

void stampa() { // Definisco: Esplicitamente void
 printf("Ciao\n");
return;
}
```

La uso prima e la definisco dopo che succede?

Nel momento che la uso implicitamente il tipo viene imposto a int e quindi dopo quando la definisco come void ho un *warning* per problemi di tipo

Tutto va dichiarato prima di usarlo!

Dichiarazione Definizione e uso

```
int main(void) {
 // dichiarazione (definisco il prototipo)
 void stampa();

 stampa();
return 0;
}

//definizione
void stampa() {
 printf("Ciao\n");
return;
}
```

Dichiarazione

- La dichiarazione di una funzione serve a dire al compilatore che vorrò usare una funzione fatta in un certo modo in quel blocco.
    - Nel caso di prima sarebbe dentro al main
- Quindi posso avere dichiarazioni multiple basta che siano consistenti tra di loro
- Avrò una sola definizione però
- Se dichiaro una funzione fuori dall'implementazione di qualsiasi altra funzione (ovvero fuori da tutto) questa ha una visibilità globale

Return

Fa parte delle istruzioni di salto che in sostanza trasferiscono il controllo in modo incondizionato

goto <etichetta>;
continue;
break;
return <expression>;

Return: ritorna un valore al chiamante e fa tornare il controllo al chiamante

N.B.: Il return del main ritorna il controllo al chiamante quindi la shell che ha lanciato il programma

Funzioni con parametri in ingresso

Le funzioni possono ricevere paramentri che hanno un tipo

I parametri vengono usati all'interno della funzione per implementare quanto serve

Viene passato il valore del parametro in chiamata che viene associato al nome del parametri dichiarato

Esercizio

Scrivere un programma che riceve in ingresso un numero n e stampa n volte “Ciao”. Vincolo n <=20
Leggere una stringa al posto che un intero e convertirlo in intero usando la funzione *int atoi(<stringa>)*

Estendere la funzione Stampa vista nelle slide precedenti perché prenda in ingresso il numero di volte in cui deve stampare la scritta

Soluzione

Dichiaro stampa che prenderà un intero come parametro

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

int main(void) {

  int stampa(int); char s[256];

  printf("Inserisci numero di stampe: ");
  fgets(s,256,stdin);
  if (stampa( n: atoi(s)) < 0 )
    printf("Errore.\n");
  return 0;
}

int stampa(int n) { //precondizione: n<20
  if (n>20)
     return -1; //errore
  for (; n>0; n--)
    printf("Ciao - (%d).\n",n);
  return 0; //tutto ok
}
```

Soluzione

Chiamo stampa passando il valore intero ritornato dalla Funzione di libreria *atoi(s)*

Notare che l'IDE mi dice che si chiamerà n all'interno della funzione stampa

N.B. IDE è l'interfaccia che uso per programmare
Nel mio caso CLion

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

int main(void) {

  int stampa(int); char s[256];

  printf("Inserisci numero di stampe: ");
  fgets(s,256,stdin);
  if (stampa( n: atoi(s)) < 0 )
    printf("Errore.\n");
  return 0;
}

int stampa(int n) { //precondizione: n<20
  if (n>20)
     return -1; //errore
  for (; n>0; n--)
    printf("Ciao - (%d).\n",n);
  return 0; //tutto ok
}
```

Soluzione

La funzione stampa usa il parametro passato che in stampa si chiama n per espletare le sue funzioni

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

int main(void) {

  int stampa(int); char s[256];

  printf("Inserisci numero di stampe: ");
  fgets(s,256,stdin);
  if (stampa( n: atoi(s)) < 0 )
    printf("Errore.\n");
  return 0;
}

int stampa(int n) { //precondizione: n<20
  if (n>20)
     return -1; //errore
  for (; n>0; n--)
    printf("Ciao - (%d).\n",n);
  return 0; //tutto ok
}
```

Esempio scope globale

Anche le variabili, come le funzioni, possono essere dichiarate come variabili globali, al di fuori di qualunque funzione.

```
int n;
int main(void) {
   void foo(void);
   n=5;
   foo();
   printf("%d\n",n);
   return 0;
}

void foo(void) {
   n++;
}
```

Cosa stampa?

Esempio scope globale

Anche le variabili, come le funzioni, possono essere dichiarate come variabili globali, al di fuori di qualunque funzione.

```
int n;
int main(void) {
   void foo(void);
   n=5;
   foo();
   printf("%d\n",n);
   return 0;
}

void foo(void) {
   n++;
}
```

Cosa stampa? 6

Scope approfondimenti: variabili locali

- Variabili automatiche (locali ad un blocco)
    - visibilità dalla dichiarazione alla fine del blocco

- Variabili locali con lo stesso nome ma in blocchi diversi (non contenuti l'uno nell'altro) sono due entità distinte

- Variabili locali con lo stesso nome ma in blocchi uno dentro l'altro hanno un problema di ambiguità
    - Effetto ombra o ***shadowing*** della variabile più interna sull'esterna

- I parametri della funzione sono variabili automatiche locali alla funzione stessa

Esempio shadowing

```
int n;
int main(void) {
   void foo(void);
   n=5;
   foo();
   printf("%d\n",n);
   return 0;
}

void foo(void) {
   int n=0;
   n++;
}
```

Cosa stampa?

Esempio shadowing

```
int n;
int main(void) {
   void foo(void);
   n=5;
   foo();
   printf("%d\n",n);
   return 0;
}

void foo(void) {
   int n=0;
   n++;
}
```

Cosa stampa? 5 (la n incrementata è quella locale)

Scope approfondimenti: variabili globali

Nel caso in cui sia necessario riferirsi alla variabile globale prima della sua dichiarazione, oppure in un altro file sorgente, occorre usare la parola chiave *extern*

Definizione di una variabile globale. La dichiara e l'alloca.

Dichiarazione di una variabile globale. La dichiara ma non l'alloca. Per ottenere questo risultato si antepone la keyword *extern*

La definizione di una variabile globale deve esistere una sola volta.

Le dichiarazioni (usando extern) saranno molteplici se la variabile è usata in più file e il programma è dato dalla compilazione separata di più file poi linkata in un unico eseguibile

Esempio Extern

Esempio foo con Progetto fatto da due file .c

main.c

```c
#include <stdio.h>

extern int n;
void foo(void);

int main(void) {
    n=5;
    foo();
    printf("%d\n",n);
    return 0;
}
```

foo.c

```c
//
// Created by Marco Anisetti on 21/10/22.
//
int n=0;
void foo(void) {
    n++;
}
```

Extern

Quando dichiaro un prototipo è come se fosse implicitamente *extern*

Infatti non definisco dichiaro e basta.

Ricordarsi che si può dichiarare tante volte ma definire (allocare) solo una volta

Scope e funzioni

- Stesse regole delle variabili globali
    - Una funzione non può essere **definita** localmente ad un blocco (non posso **definire** funzioni dentro altre funzioni)
- La funzione è visibile da dove è **dichiarata** fino alla fine del file

- Coerentemente dovremmo pensare ad usare extern associato anche alle funzioni, ma i moderni compilatori considerano extern tutte le funzioni

Parametri

- La chiamata di una funzione (invocazione):
  - uso della funzione e relativa specifica i dati sui quali effettivamente la funzione opera

- **parametri formali** sono i nomi dei parametri indicati nel prototipo
  - *int sum(int a, int b);* a e b sono i parametri formali

- **parametri attuali** i valori indicati nelle chiamate
  - *s=sum(c,10);* c (il suo valore in realtà) e 10 sono i parametri attuali

- Il prototipo mette il nome della funzione nella tabella dei simboli

Terminologia

Prototipo: la dichiarazione della funzione.

La definizione consiste in

Intestazione: il prototipo senza “;”

Corpo: tra {} si tratta l’implementazione della funzione

Chiamata della funzione

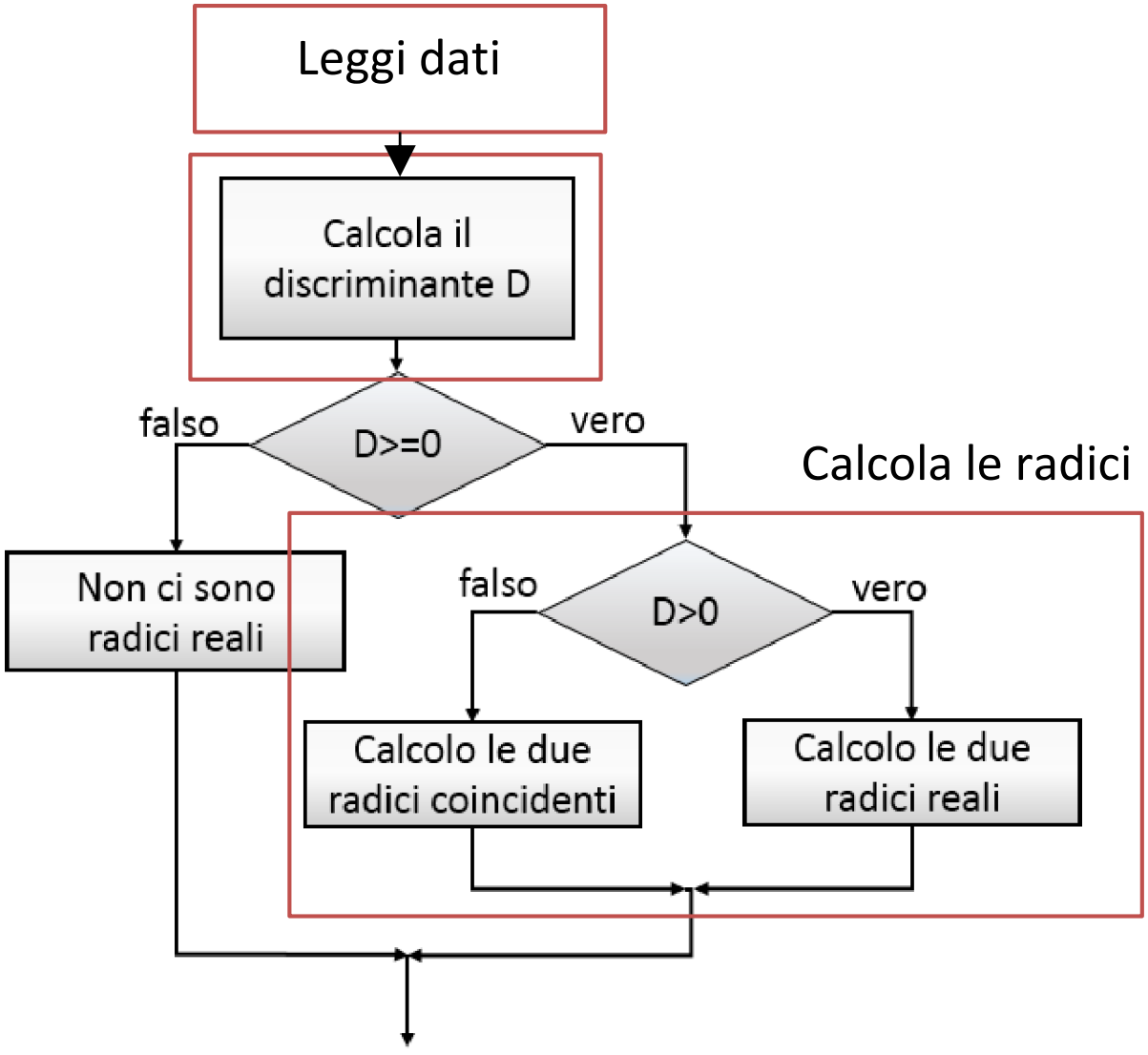
Chiamata o invocazione

Funzione: ritorno = funzione(parametri attuali);
Procedura: procedura(parametri attuali);

Parametri attuali si chiamano anche **argomenti**

Per ogni funzione si possono avere molte chiamate su dati diversi

Esempio


Leggi dati
Calcola il discriminante D
falso
D>=0
vero
Calcola le radici
Non ci sono radici reali
falso
D>0
vero
Calcolo le due radici coincidenti
Calcolo le due radici reali

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
// #### Dichiarazioni di prototipi #####
// ha la forma di una procedura ma dobbiamo approfondire
void coefficienti(float * c1,float * c2, float * c3);
// ha la forma di una funzione
double discriminante(float c1,float c2, float c3);
// ha la forma di una procedura
void radici(float c1,float c2, float c3, double d);

int main()
{
    float a, b, c;
    double d;
    coefficienti( c1: &a,  c2: &b,  c3: &c);
    if (a==0) {
        printf("\nequazione di primo grado!\n");
        return 0;
    }
    d=discriminante( c1: a, c2: b, c3: c);
    radici( c1: a,  c2: b,  c3: c, d);

}
```

Procedura coefficienti

Dobbiamo ancora capire cosa è float * ci basti sapere che permette alla funzione di modificare i parametri attuali attraverso i formali.

```
void coefficienti(float * c1,float * c2, float * c3)
{
    printf("coefficiente a:\t");
    scanf("%f", c1);
    printf("coefficiente b:\t");
    scanf("%f", c2);
    printf("coefficiente c:\t");
    scanf("%f", c3);
}

double discriminante(float c1,float c2, float c3)
{
    return(c2*c2-4*c1*c3);
}
```

```c
void radici(float c1, float c2, float c3, double d)
{   double x1,x2;
    if(d>=0) {
        if (d > 0) {
            x1 = (-c2 - sqrt(d)) / (2 * c1);
            x2 = (-c2 + sqrt(d)) / (2 * c1);
            printf("Due soluzioni reali distinte x1= %5.3lf e x2= %5.3lf \n", x1, x2);
        } else {
            x1 = (-c2) / (2 * c1);
            printf("Due soluzioni reali coincidenti x1 e x2 uguali a %5.3lf \n", x1);
        }
    }
    else
        printf("non possono esserci soluzioni reali");
    return;
}
```

Albero delle chiamate

Le chiamate a funzione sono organizzate ad albero

La radice è il main

Ogni chiamata di funzione dentro al main è un ramo collegato alla radice

Ogni chiamata dentro un’altra funzione è un ramo che si collega al ramo generato dalla funzione contenitrice

Esempio albero

```
int f();
int g();
int a,b;

void main(){
 a=f();
 for(int i=0;i<3;b=g(),i++);
}

int f(){
return 0;
}

int g(){
return f()+1;
}
```

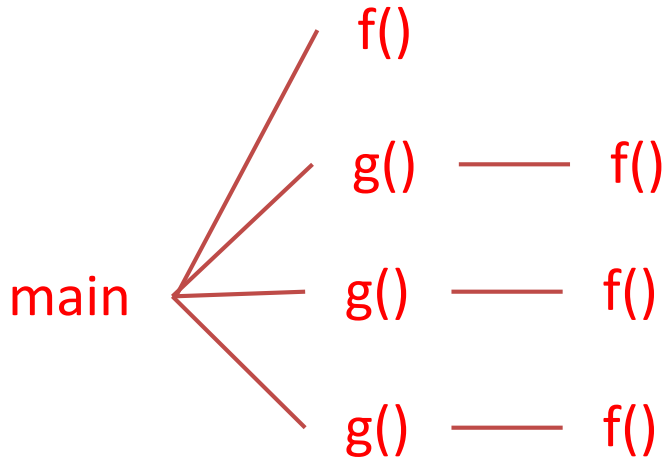
Esempio albero

```
int f();
int g();
int a,b;

void main(){
 a=f();
 for(int i=0;i<3;b=g(),i++);
}

int f(){
return 0;
}

int g(){
return f()+1;
}
```



Esercizio

Scrivere un programma che calcola quadrati e cubi dei primi 10 numeri interi.

Definire le funzioni e pensare a come riusarle

Soluzione

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

int quadrato (int y);
int cubo (int y);

int main (int argc, char *argv[])
{
  int x;
  for (x = 1; x <= 10; x++)
      printf("%d\n",quadrato(x));

  for (x = 1; x <= 10; x++)
      printf("%d\n",cubo(x));

  return 0;
}
```

```
int quadrato (int y){
    return y * y;
}

int cubo (int y){
  int q;
  q = quadrato(y);
  return q * y;
}
```

Chiamate a funzione

Ha senso che una funzione richiami se stessa al posto che un’altra? O che chiami una che poi la richiama?

Chiamate a funzione

Ha senso che una funzione richiami se stessa al posto che un’altra? O che chiami una che poi la richiama?

SI! Realizza una relazione di ricorrenza ovvero esegue più volte il corpo della funzione fino ad una condizione di terminazione

Si chiama **ricorsione**

Macro

Il C permette come abbiamo visto di definire una sorta di funzioni attraverso la #define

Tali "funzioni" si chiamano **macro** e sono soggette alla **macro espansione** da parte del preprocessore

Non sono quindi per nulla uguali alle funzioni che hanno una azione dinamica in esecuzione chiamata call and return

Macro

Esempi

#define AVVISO printf("programma terminato\n");

#define STAMPA(a) printf(" il valore è %d",a);

*#define MOLTIPLICA(a,b) a*b*

Macro

Esempi

#define AVVISO printf("programma terminato\n");

#define STAMPA(a) printf(" il valore è %d",a);

*#define MOLTIPLICA(a,b) a*b*

res=MOLTIPLICA(a,b) cosa diventa?

Macro

Esempi

#define AVVISO printf("programma terminato\n");

#define STAMPA(a) printf(« il valore è %d",a);

*#define MOLTIPLICA(a,b) a*b*

res=MOLTIPLICA(a,b) cosa diventa? res=a*b

Macro

Esempi

#define AVVISO printf("programma terminato\n");

#define STAMPA(a) printf(« il valore è %d",a);

*#define MOLTIPLICA(a,b) a*b*

res=MOLTIPLICA(a-c,b) cosa diventa?

Macro

Esempi

#define AVVISO printf("programma terminato\n");

#define STAMPA(a) printf(« il valore è %d",a);

*#define MOLTIPLICA(a,b) a*b*

res=MOLTIPLICA(a-c,b) cosa diventa? res=a-c*b (errore)

Esempio

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

/* run this program using the console pauser or add your own
#define MUL(a,b) a*b
#define STAMPA(a) printf("valore = %d\n",a)

int main()
{
 int a=10;
 int b=4;
 int c=3;

    printf("Mul vale %d\n",MUL(a+c,b));
    STAMPA(a);

    STAMPA(MUL(a+c,b));
    printf("fine\n");
return 0;
}
```

Esempio

Output del preprocessore

```
int main()
{
 int a=10;
 int b=4;
 int c=3;

    printf("Mul vale %d\n",a+c*b);
    printf("valore = %d\n",a);

    printf("valore = %d\n",a+c*b);
    printf("fine\n");
return 0;
}
```

Structure

Bookmarks