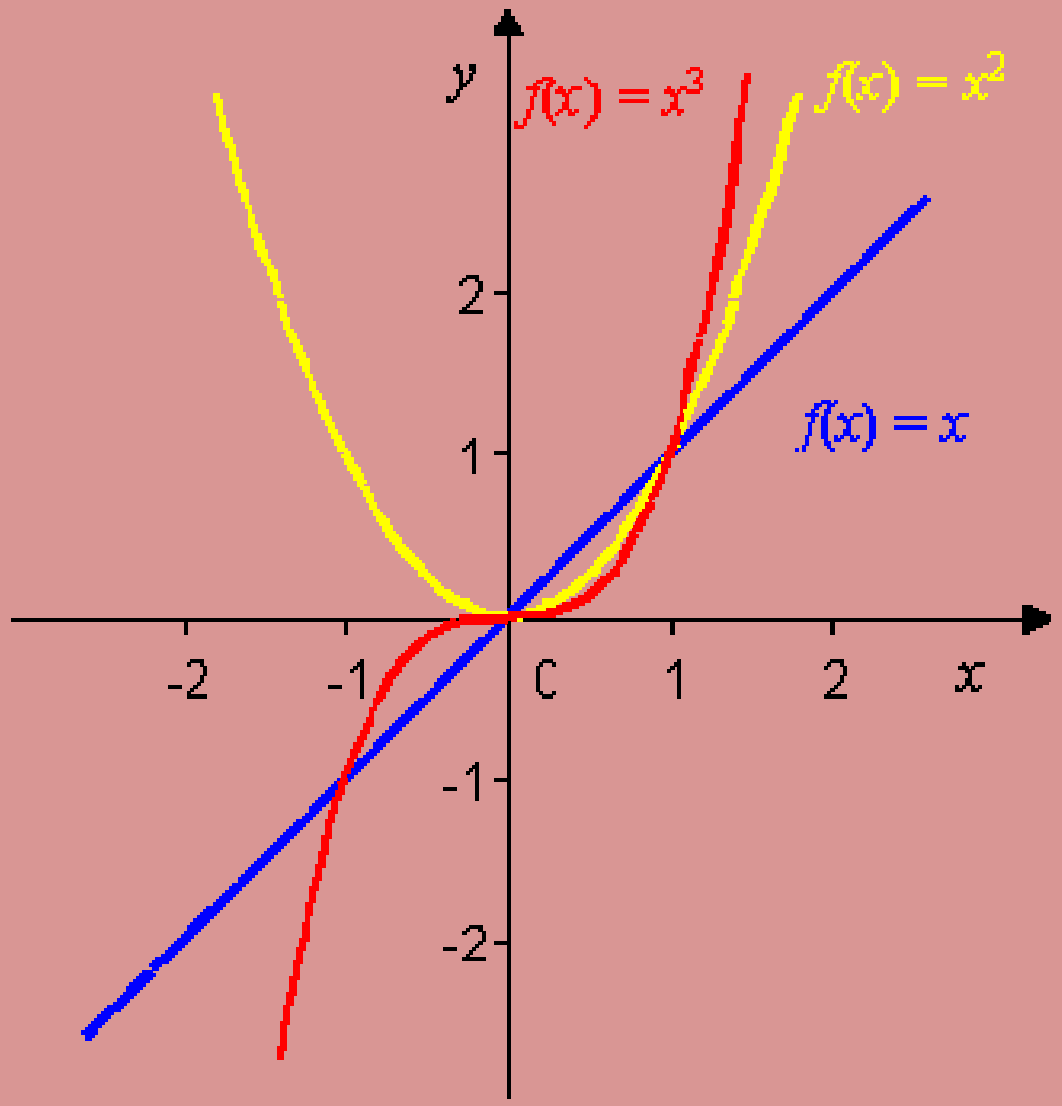
- – Funzioni Elementari
  - Passaggio dei Parametri
  - Recursione
- – Ambienti e Visibilità

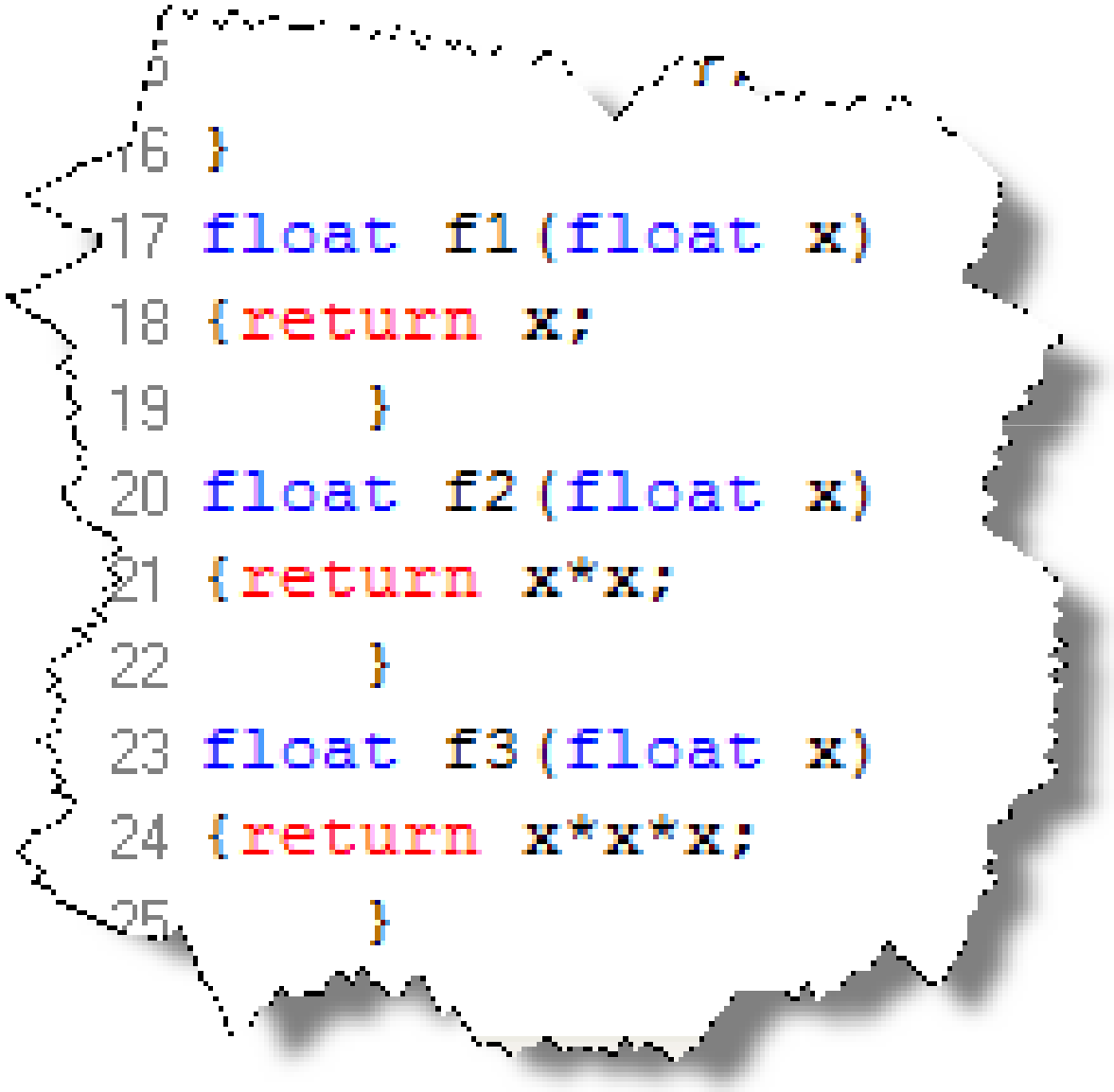
$f1(x) = x$ $f2(x) = x^2$ $f3(x) = x^3$



Le chiamate alle fi

```
float x, y1,y2,y3;
        for(I=-30;I<=30 ; I++)
        {    x=I/10.0;
             printf("%f\t",x);
             y1=f1(x);
             y2=f2(x);
             y3=f3(x);
             printf(" le tre funzioni =  %f\t%f\t%f\n", y1,y2,y3);
        };   return 0;
```

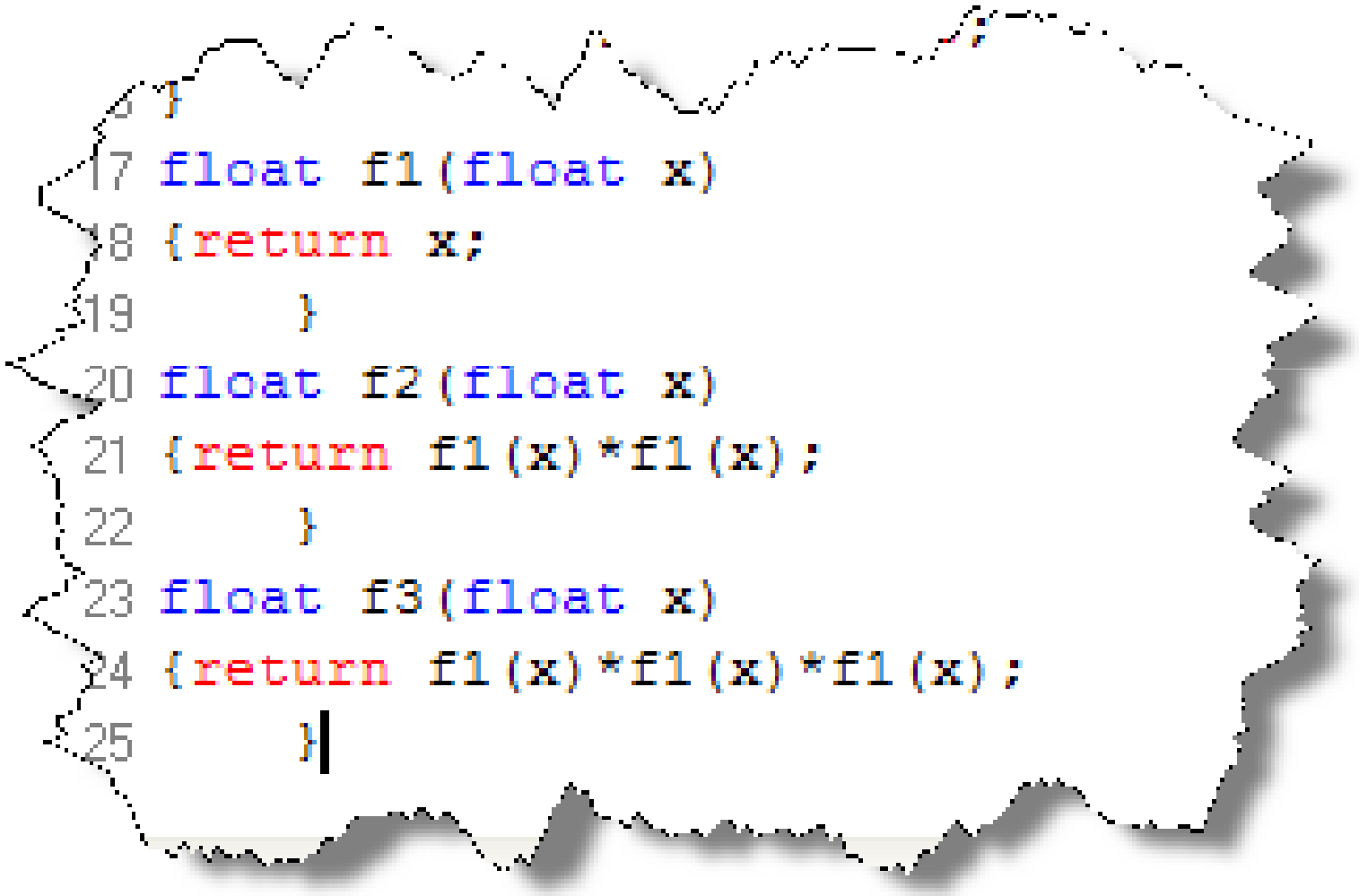
Le definizioni



Ma anche …

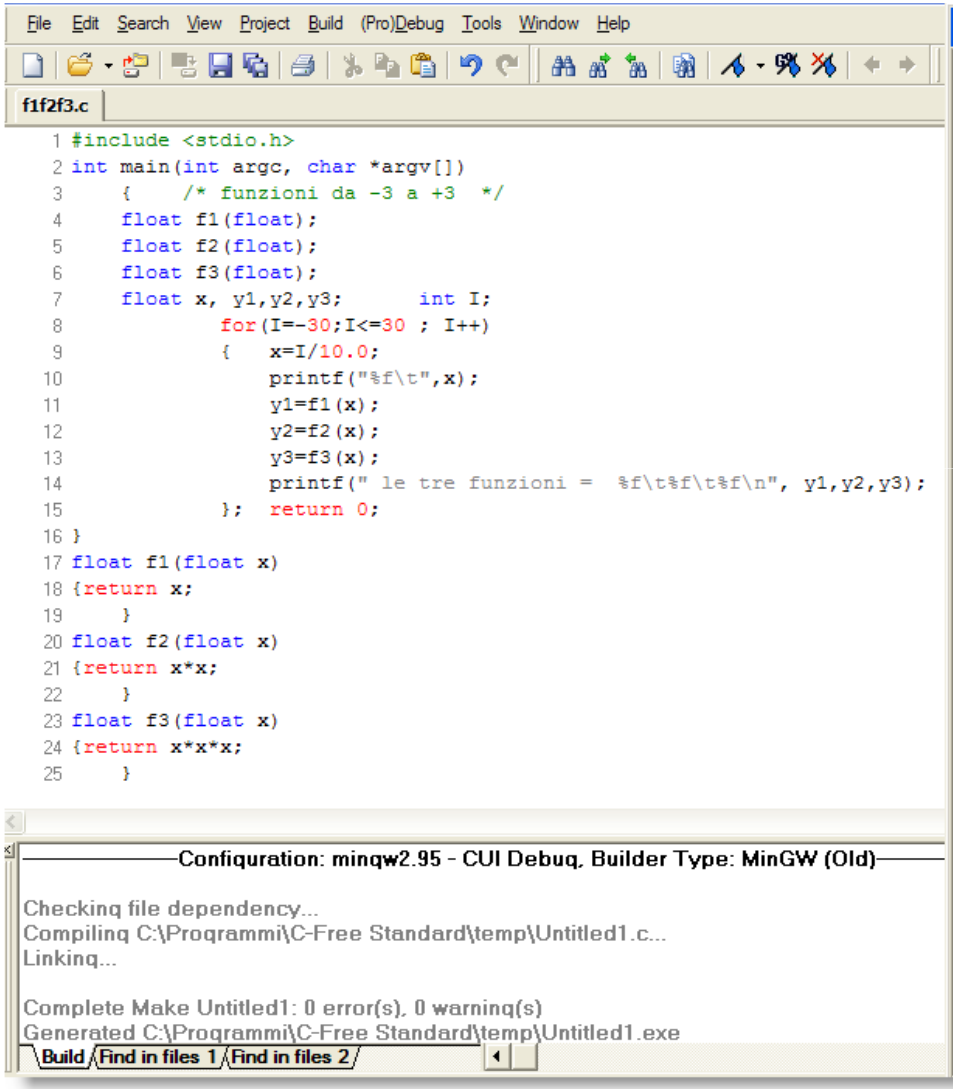

}
17 float f1(float x)
18 {return x;
19 }
20 float f2(float x)
21 {return x*f1(x);
22 }
23 float f3(float x)
24 {return x*f2(x);
}

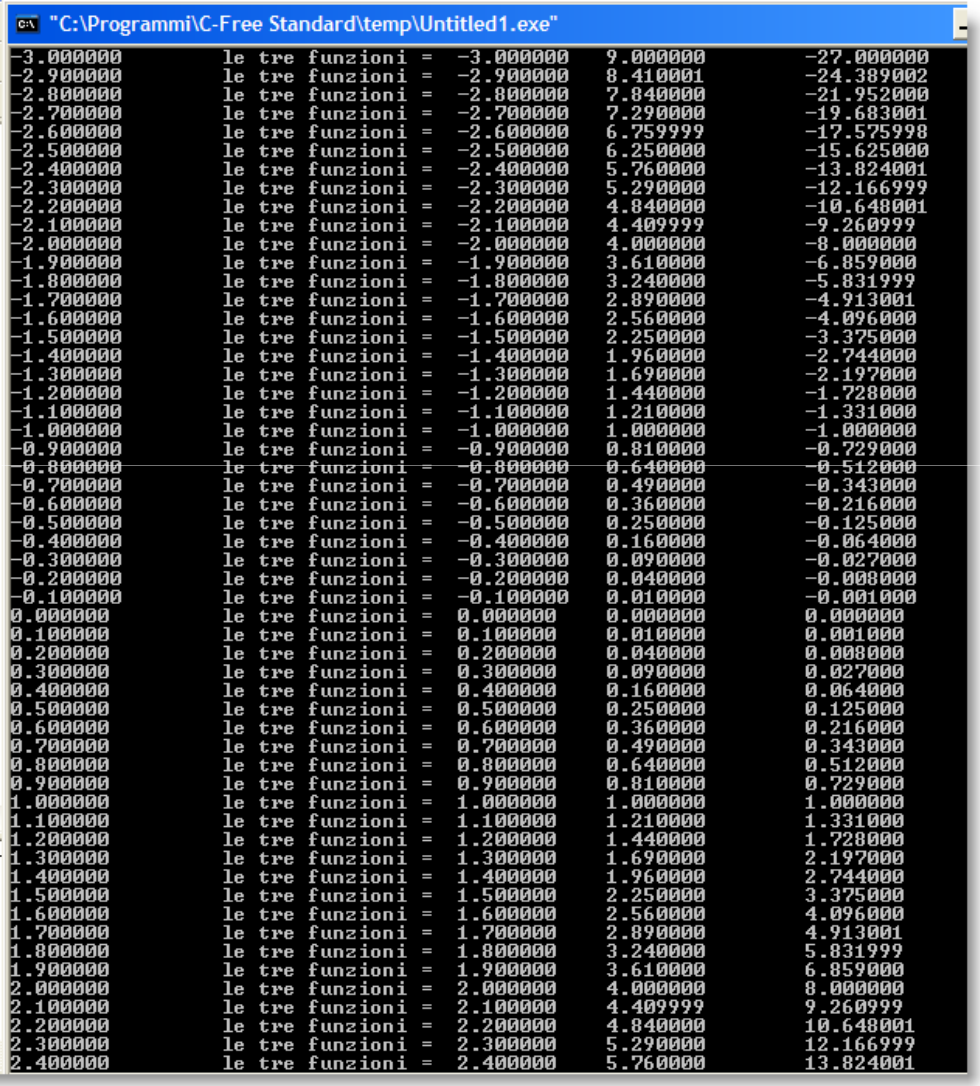
….. O ancora …



```
16 }
17 float f1(float x)
18 {return x;
19      }
20 float f2(float x)
21 {return f1(x)*f1(x);
22      }
23 float f3(float x)
24 {return f1(x)*f1(x)*f1(x);
25      }
```

Il programma





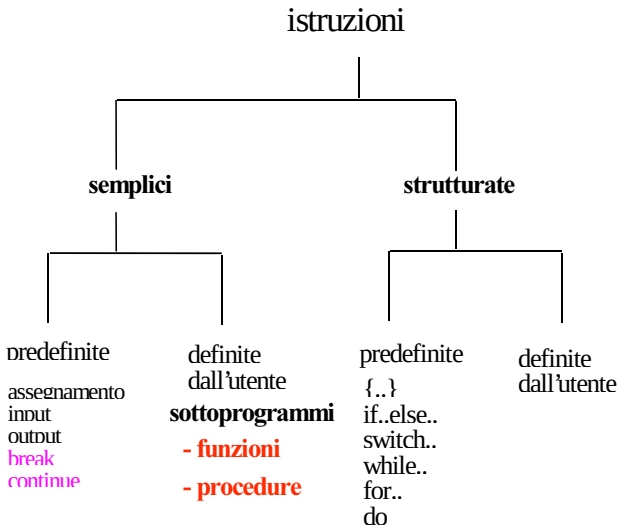
Struttura di un Programma C

Un programma C ha in linea di principio la seguente forma:

- **Direttive per il preprocessore**
- **Definizione di tipi**
- **Prototipi di funzioni**, con dichiarazione dei tipi delle funzioni e dei parametri)
- **Dichiarazione delle Variabili Globali**
- **Dichiarazione Funzioni**, dove ogni dichiarazione di una funzione ha la forma:
  Tipo NomeFunzione(Parametri)
  {
  Dichiarazione Variabili Locali
  Istruzioni C
  }

```
#include <stdio.h>

typedef struct point {
            int   x;    int   y;
            };
int       f1(void);
void    f2(int , double );


int   sum;


void main(  )
{
      int j;
      double g=0.0;
      for(j=0;j<2;j++)
            f2(j,g);

}

void f2(int i, double g)
{
      sum = sum + g*i;
}
```

Sottoprogrammi: Funzioni e Procedure



I linguaggi di alto livello permettono di definire istruzioni non primitive per risolvere parti specifiche di un problema: i **sottoprogrammi** (funzioni e procedure).

Funzioni e Procedure

Ad esempio: Ordinamento di un insieme

```
#include <stdio.h>
#define dim 10

main()
{int V[dim], i,j, max, tmp, quanti;

/* lettura dei dati */
for (i=0; i<dim; i++)
  { printf("valore n. %d: ",i);
    scanf("%d", &V[i]);
   }
/*ordinamento */
for(i=0; i<dim; i++)
  {quanti=dim-i;
    max=quanti-1;
    for( j=0; j<quanti; j++)
       if (V[j]>V[max])
          max=j;
    if (max<quanti-1)
    {  tmp=V[quanti-1];
       V[quanti-1]=V[max];
       V[max]=tmp;
    }
  }
/*stampa */
for(i=0; i<dim; i++)
  printf("Valore di V[%d]=%d\n", i, V[i]);
}
```

- Potrebbe essere conveniente scrivere lo stesso algoritmo in modo piu‘ **astratto**:

```
#include <stdio.h>
#define dim 10

main()
{
int V[dim];

/* lettura dei dati */
leggi(V, dim);

/*ordinamento */
ordina(V, dim);

/*stampa */
stampa(V,dim);
}
```

- + **`leggi(), ordina(), stampa()`** sono ***sottoprogrammi***: il main "chiama" leggi, ordina e stampa.

Vantaggi:

sintesi
leggibilita‘
possibilita‘ di riutilizzo del codice

Sottoprogrammi: *funzioni* e *procedure*

- Rappresentano nuove istruzioni che agiscono sui dati utilizzati dal programma, "nascondendo" la sequenza delle operazioni effettivamente eseguite dalla macchina.

- Vengono realizzate mediante la definizione di unita‘ di programma (***sottoprogrammi***) distinte dal programma principale (*main*).

➳ **D'ora in poi:** il programma e‘ una **collezione di unita` di programma** (tra le quali compare l’unita‘ *main*)

Tutti i linguaggi di alto livello offrono la possibilita‘ di utilizzare funzioni e/o procedure.

Cio‘ e‘ reso possibile da:

- costrutti per la **definizione** di sottoprogrammi
- meccanismi per l’**utilizzo** di sottoprogrammi (meccanismi di ***chiamata***)

Funzioni e Procedure

Definizione:

Nella fase di **definizione** di un sottoprogramma (funzione o procedura) si stabilisce:

- un **identificatore** del sottoprogramma
- si esplicita il **corpo** del sottoprogramma (cioe‘, l’insieme di istruzioni che verra‘ eseguito ogni volta che il sotto-programma verra‘ *chiamato*);
- si stabiliscono le **modalita` di comunicazione** tra l’unita‘ di programma che usa il sottoprogramma ed il sottoprogramma stesso (definizione dei **parametri formali**).

Utilizzo di funzioni/procedure (*chiamata*):

- Per chiamare un sottoprogramma (cioe‘, per richiedere l’esecuzione del suo corpo), si utilizza l’identificatore assegnato al sottoprogramma in fase di definizione (***chiamata*** o invocazione del sottoprogramma).

Meccanismo di Chiamata

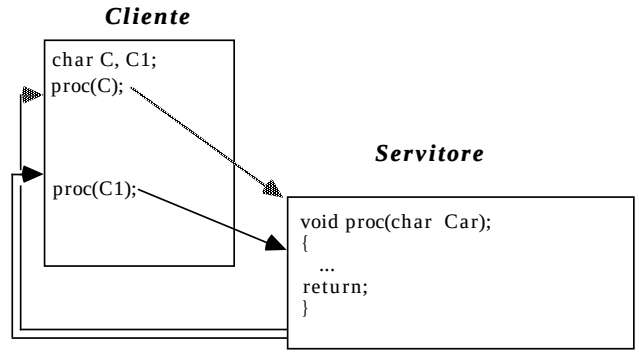
Quando si verifica una chiamata a sottoprogramma, si possono individuare due entita‘:

- l’unita‘ di programma **chiamante**;
- l’unita‘ di programma **chiamata** (il sotto-programma).

Quando avviene la chiamata, l’esecuzione dell'unita` di programma "chiamante" (quella, cioe`, che contiene l'invocazione) viene **sospesa**, ed il controllo passa al sottoprogramma chiamato (che eseguira` le istruzioni contenute nel corpo).

L'unita` chiamante funge da **cliente** dell'unita` chiamata (che svolge il ruolo di **servitore**).

Modello Cliente-Servitore



Parametri

I ***parametri*** costituiscono il mezzo di comunicazione tra unita‘ chiamante ed unita‘ chiamata.
Supportano lo scambio di informazioni tra chiamante e sottoprogramma.

parametri formali: sono quelli specificati nella definizione del sotto-programma. Sono in numero prefissato e ad ognuno di essi viene associato un tipo. Le istruzioni del corpo del sottoprogramma utilizzano i parametri formali.

parametri attuali: sono i valori effettivamente forniti dall'unita‘ chiamante al sottoprogramma all'atto della chiamata.

Parametri

- Parametri ***attuali*** (specificati nella chiamata) e ***formali*** (specificati nella definizione) devono corrispondersi in ***numero***, ***posizione*** e ***tipo***.

- All'atto della chiamata avviene il ***legame dei parametri***, cioe‘ ai parametri formali vengono associati i parametri attuali.

Come avviene l'associazione tra parametri attuali e parametri formali ?

Esistono, in generale, varie forme di legame. Ad esempio:
- legame per **valore**;
- legame per **indirizzo**;

Il significato delle due tecniche di legame dei parametri verra‘ spiegato piu‘ avanti.

Funzioni e Procedure

Vantaggi:

- **riutilizzo di codice**: sintetizzando in un sottoprogramma un sotto-algoritmo, si ha la possibilita‘ di invocarlo piu‘ volte, sia nell’ambito dello stesso programma, che nell’ambito di programmi diversi (evitando di dover replicare ogni volta lo stesso codice).
- migliore **leggibilita`**: si ha in fatti una maggiore capacita‘ di astrazione
- sviluppo **top-down**: si delega a funzioni/procedure da sviluppare in una fase successiva la soluzione di sottoproblemi.
- testo del programma piu‘ **breve**: minore probabilita‘ di errori, dimensione del codice eseguibile piu‘ piccola.

Procedure e Funzioni

In generale, i sottoprogrammi si suddividono in **procedure** e **funzioni**:

Procedura:

E` un’astrazione della nozione di ***istruzione***. E` un’istruzione non primitiva attivabile in un qualunque punto del programma in cui puo` comparire un’istruzione.

Funzione:

E’ un’astrazione del concetto di ***operatore***. Si puo` attivare durante la valutazione di una qualunque espressione e **restituisce un valore.**

Ad esempio:

```
main()
{ int Ris, N=7;
  stampa(N);/*procedura*/
  Ris=fattoriale(N)-10;/*funzione*/
};
```

Formalmente, in C i sottoprogrammi sono soltanto ***funzioni***; le procedure possono essere realizzate come funzioni che non restituiscono alcun valore (**void**).

Funzioni in C

Procedure e funzioni si definiscono seguendo regole sintattiche simili.

Definizione di funzione:

<def-funzione> ::= <intestazione>
{ <parte-dichiarazioni> <parte-istruzioni> }

Quindi, per definire una funzione, e' necessario specificare una ***intestazione*** e un ***blocco*** {..}:

Struttura dell'intestazione:

<intestazione>::=<tipo-ris> <nome> ([<lista-par-formali>])

dove:

- **<tipo-ris>**: e' un indentificatore che indica il tipo di risultato restituito(***codominio***). Il tipo restituito puo' essere predefinito o definito dall'utente. Una funzione non puo' restituire valori di tipo:
  - **vettore**
  - **funzione**
- **<nome>:** e' l'identificatore della funzione
- **<lista-par-formali>** e` la lista dei parametri formali (*dominio*). Per ciascun parametro formale viene specificato il tipo ed un identificatore che e` un nome simbolico per rappresentare il parametro all'interno della funzione (nel *blocco*). I parametri sono separati mediante virgola.

Definizione di Funzioni in C

Blocco :

- Il blocco contiene il **corpo** della funzione e, come al solito, e' strutturato in una <parte dichiarazioni> e una <parte istruzioni>:
  - la <parte dichiarazioni> contiene le dichiarazioni e definizioni *locali* alla funzione;
  - la <parte istruzioni> contiene la sequenza di istruzioni associata al corpo (rappresenta l'algoritmo eseguito dalla funzione)

- I dati riferiti nel blocco possono essere **costanti**, **variabili**, oppure **parametri formali**: all'interno del blocco, i parametri formali vengono trattati come variabili.

Istruzione return:

Per restituire il risultato, la funzione utilizza (all'interno della parte istruzioni) l'istruzione **return**:

return [<espressione>]

Effetto:
restituisce il controllo al chiamante e assegna all'identificatore della funzione il valore dell'<espressione>.

Definitions

Function definition general form:

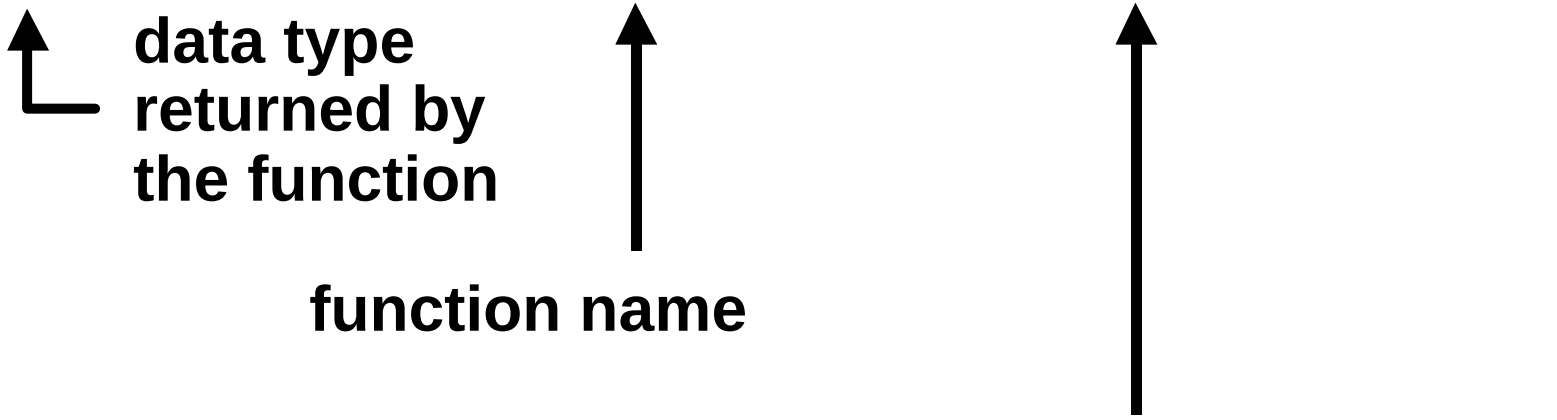
```
return-type  function-name( parameter declarations )
{
    declarations
    C statements
}
```

a function is defined
by stating the HEADER
and the BODY

Function Definitions: The Header

The HEADER includes:

return-type function-name(parameter declarations)



Function Definitions: The Body

```
return-type  function-name( parameter declarations )
{
    declarations
    C statements
}
```

BODY

The BODY is simply C statements (including declaration statements) bounded by braces

Esempio:

```
int maggioredi100 (int a) /*intest. */
{/*parte dichiarazioni: */
  const int C=100;

  /* parte istruzioni: */
  if (a>C) return 1;
  else return 0;
}
```

Esempio:

```
#define N 100

typedef   char vettore[N];

int minimo (vettore vet)
{
int i, v, min;/* def. locali a minimo */
  for (min=vet[0], i=1; i<N; i++)
    {  v=vet[i];
       if (v<min) min=v;
    }
  return min;
}
```

- i, v, min sono ***variabili locali***:
  - **tempo di vita**: esistono solo durante l'esecuzione della funzione minimo
  - **visibilita`**: sono visibili (cioe' utilizzabili) soltanto all'interno della funzione minimo.

Esempio:

```
int   read_int () /* intest. */
{
  int a
  scanf("%d", &a);
  return a;
}
```

Possono esserci ***piu` istruzioni return***:

```
int max (int a, int b) /*intest.*/
{
  if (a>b)    return a;
  else   return b;
}
```

o ***nessuna***:

```
int   print_int (int a) /* intestazione *
{
  printf("%d", a);
}
```

- In questo caso, il sottoprogramma termina in corrispondenza del simbolo } ed il valore restituito e' ***indefinito.***

Esempio:

```
/* funzione elevamento a potenza */

long power (int base, int n)
{
 int i;
 long p=1;

 for (i=1;i<=n;++i)
       p *= base; /* p = p*base */
 return p; /* ritorna il risultato */
}
```

Funzioni in C

Chiamata di funzioni:

In generale, la chiamata di una funzione compare all'interno di una espressione secondo la sintassi:

…nomefunzione(<lista parametri attuali>)…

Ad esempio:

```
main()
{
int z, x=2;
...
z=power(x,2)+power(x,3);
x=max(power(z,2), 30);
printf("%d\n", x);
}
```

Realizzazione delle Procedure in C

Una funzione puo' anche avere nessun valore (void) come risultato:

void	insieme vuoto di valori (dominio vuoto)
void fun(...)	funzione che non restituisce alcun valore

➳ **In questo modo si realizza in C il concetto di** procedura

Esempio:

```
void print_int(int a)
{
  printf("%d", a);
}
```

➳ Poiche' una procedura non restituisce alcun valore, non e' necessario prevedere l'istruzione di **return** all'interno del corpo; se si utilizza, **non si deve specificare alcun argomento**:

```
return;
```

Uso:

La procedura e' l'astrazione del concetto di istruzione:

```
main()
{ int X;
  scanf("%d", &X);
  print_int(X):
}
```

Es. funzione che somma due valori di tipo int e restituisce un int:

```
int somma(int a, int b)
{
    int sum;
    sum = a+b;
    return(sum);
}
```

La chiamata della funzione viene fatta così:

```
void main(void)
{
    int A=23; int B=-31; int risultato;
    risultato = somma(A,B);
    printf("somma= %d\n", risultato);
}
```

Es. funzione che non restituisce alcun valore:

```
void somma(int a, int b)
{
    int sum;
    sum = a+b;
    printf("somma= %d\n", sum);
    /* non serve la return */
}
```

La chiamata della funzione viene fatta così:

```
void main(void)
{
    int A=23; int B=-31;
    somma(A,B);
}
```

Esempio:

```
#include <stdio.h>

int max (int a, int b) /*def. max*/
{
  if (a>b) return a;
  else return b;
}

void print_int (int a) /* def. */
{
  printf("%d\", a);
  return;
}

void dummy() /*def. dummy */
{
printf("Ciao!\n");
}

main()
{
  int A, B;
  printf("Dammi A e B: ");
  scanf("%d %d", &A, &B);
  print_int(max(A,B));
  dummy();
}
```

➳ Se all'interno di un blocco viene utilizzata una funzione f, la definizione di f deve comparire ***prima*** del blocco che la utilizza.

Esempio:

```
#include <stdio.h>

int max (int a, int b)
{
  if (a>b) return a;
  else return b;
}

int sommamax(int a1, a2, a3, a4)
{  return max(a1,a2)+max(a3,a4);}

main()
{
  int A, B, C,D;
  scanf("%d%d%d%d",&A,&B,&C,&D);
  printf("%d\n", sommamax(A,B,C,D));
}
```

Dichiarazione di funzione

Regola Generale:

Prima di utilizzare una funzione e' necessario che sia gia' stata **definita oppure *dichiarata***.

Funzioni C:

- **definizione**: descrive le proprieta' della funzione (tipo, nome, lista parametri formali) e la sua realizzazione (lista delle istruzioni contenute nel blocco).
- **dichiarazione** (***prototipo***): descrive le proprieta' della funzione senza definirne la realizzazione (***blocco***) ◊ serve per "anticipare" le caratteristiche di una funzione definita successivamente.

Dichiarazione di una funzione:

La **dichiarazione** di una funzione si esprime mediante l'intestazione della funzione, seguita da "**;**":

<tipo-ris> <nome> ([<lista-par-formali>]);

Ad esempio:
Dichiarazione della funzione max:

```
int max(int a, int b);
```

Function Prototypes

return-type function-name(parameter declarations);

- generally look like the function header
- end with a semi-colon (**;**)
- come before 1st use (call) of function
- tell compiler (and programmer!) what to expect

```
double sqrt(double num);
void set_date(int, int, int);
```

Esempio:

```
#include <stdio.h>

main()
{
  int A, B;
  printf("Dammi A e B: ");
  scanf("%d %d", &A, &B);
  printf("%d\n", max(A,B));
}

int max (int a, int b) {
  if (a>b) return a;
  else return b;
}
```

- In questo caso il compilatore segnala un **errore** in corrispondenza della chiamata **max(A,B)**, perche‘ viene usato un identificatore che viene definito successivamente (dopo il main())

Soluzione:

```
#include <stdio.h>

int max(int a, int b);

main()
{
  int A, B;
  printf("Dammi A e B: ");
  scanf("%d %d", &A, &B);
  printf("%d\n", max(A,B));
}

int max (int a, int b) /*intestaz. */
{
  if (a>b) return a;
  else return b;
}
```

E le dichiarazioni di printf, scanf etc. ?

- sono contentue nel file stdio.h:

#include <stdio.h>

provoca l’inserimento del contenuto del file specificato.

Dichiarazione di Funzioni

Una funzione puo‘ essere ***dichiarata*** in punti diversi, ma e‘ ***definita*** una sola volta.

E‘ possibile inserire i prototipi delle funzioni utilizzate:

- nella parte dichiarazioni globali di un programma,
- nella parte dichiarazioni del **main**,
- nella parte dichiarazioni delle funzioni.

Ad esempio:

```
    main()
    {   long power (int base, int n);
        int X, exp;

        scanf("%d%d", &X, &exp);
        printf("%ld", power(X,exp));
    }
...
```

Struttura dei Programmi C

Spesso si strutturano i programmi in modo tale che la definizione del main compaia prima delle definizioni delle altre funzioni (per favorire la **leggibilita**‘).

Protocollo da utilizzare:

<lista dichiarazioni di funzioni>
<main>
<definizioni delle funzioni dichiarate>

Ad esempio:

Calcolo della radice intera di un numero intero letto a terminale.

```
#include <stdio.h>

/* dichiarazioni delle funzioni: */
   int   RadiceInt (int par);
   int   Quadrato (int par);

main(void)
{
   int   X;
   scanf("%d", &X);
   printf("Radice: %d\n", RadiceInt(X));
   printf("Quadrato: %d\n", Quadrato(X));
}

/* definizione funzioni: */

int   RadiceInt (int par)
{
  int cont = 0;
  while (cont*cont <= par)
    cont = cont + 1;
  return (cont-1);
}

int   Quadrato (int par)
{
  return (par*par);
}
```

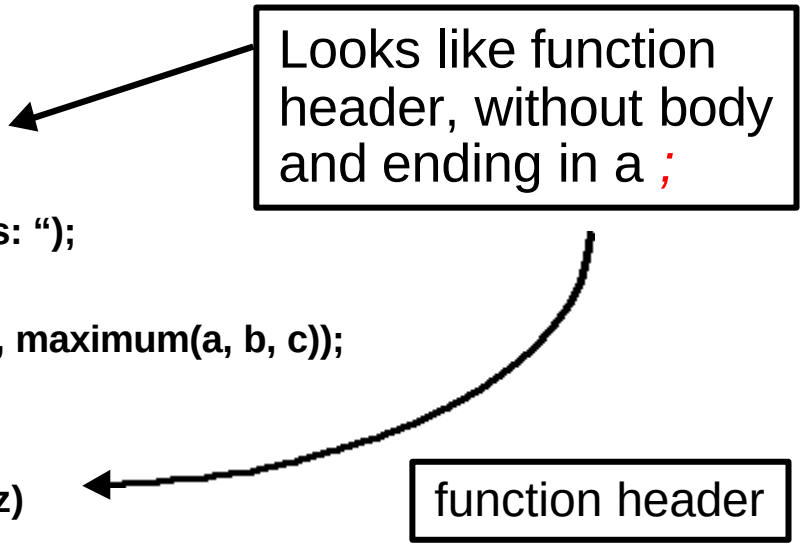
Definition Rules

- functions return max of **one** data type, if any
  - void, if none. **Default if not supplied is `int`, not void**
  - MUST have a **return** statement to return a value
  - **return** statement optional if void return data type
  - format: **return *expression*;**
  - CONTROL returns to calling routine at return statement or at ending brace of function
- functions may call other functions
  - basis for all C programs
  - recursion: a function may call itself
- parameters
  - are local variables
  - list includes data type declarations; void, if none

Sample C Code

function prototype

```
#include <stdio.h>
main()
{
   int a, b, c;
   int maximum( int, int, int);

   printf("Enter three integers: ");
   scanf("%d%d%d", a, b, c);
   printf("Max value is %d\n", maximum(a, b, c));
}

int maximum( int y, int x, int z)
{
   if( x > y  &&  y > z)
      return x;
  if( y > z)  return y ; else return z;
}
```



Sample C Code

decl vs no decl

```
main()
{
   double x, y;

   x = 4.0;
   y = sqrt(x);
   printf("f\n", y);
}
```

No declaration

prints wrong answer of 16640.000000, because the double returned is presumed to be an int!

```
main()
{
   double x, y;
   double sqrt(double);

   x = 4.0;
   y = sqrt(x);
   printf("%f\n", y);
}
```

with declaration

prints correct answer of 2.000000

AMBIENTI, VISIBILITÀ DELLE VARIABILI

AMBIENTE GLOBALE DEL PROGRAMMA

- insieme di identificatori (tipi, costanti, variabili) definiti nella parte dichiarativa globale
- **regole di visibilità**: visibili a tutte le funzioni del programma

AMBIENTE LOCALE DI UNA FUNZIONE

- insieme di identificatori (tipi, costanti, variabili) definiti nella parte dichiarativa locale **e** degli identificatori definiti nella testata (parametri formali)
- **regole di visibilità**: visibili alla funzione e ai blocchi in essa contenuti

AMBIENTE DI BLOCCO

- insieme di identificatori (tipi, costanti, variabili) definiti nella parte dichiarativa locale di blocco
- **regole di visibilità**: visibili al blocco e ai blocchi contenuti

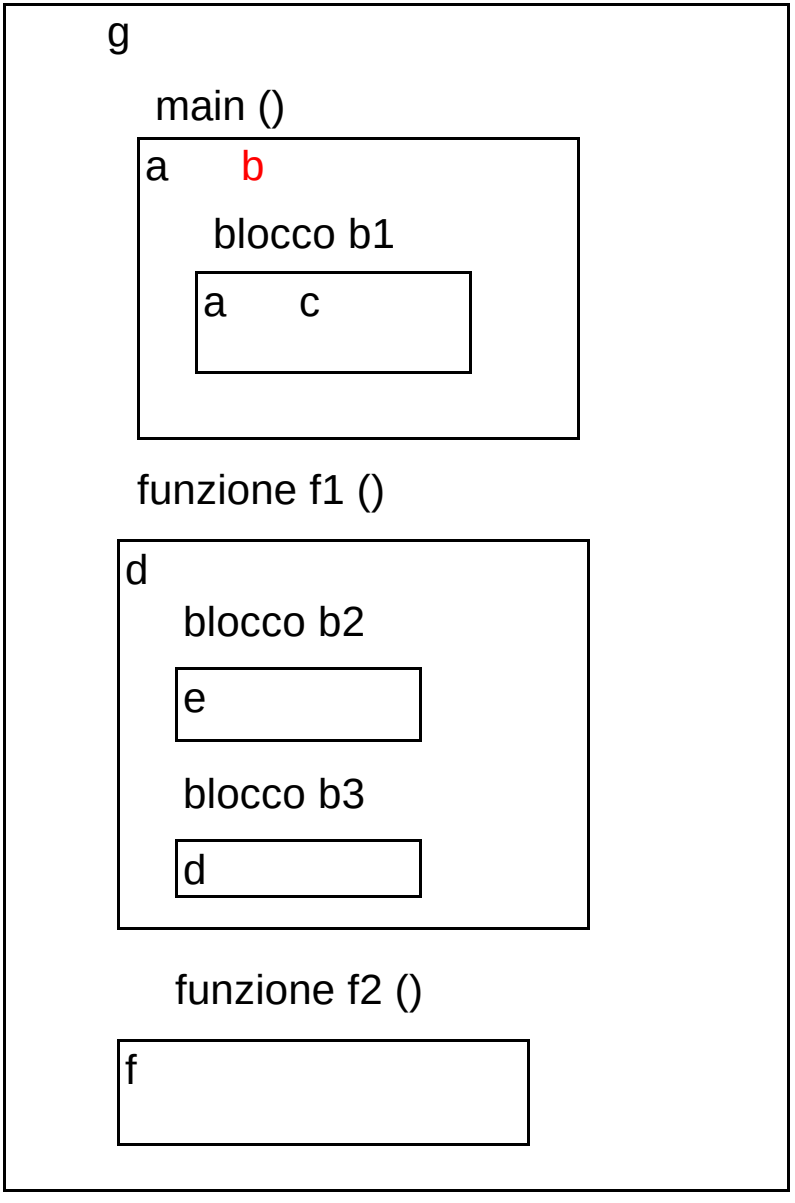
IDENTIFICATORE DI FUNZIONE:

- visibile a tutte le funzioni e a se stessa

OMONIMIA DI IDENTIFICATORI IN AMBIENTI DIVERSI

- è visibile quello dell'ambiente più «vicino»

ESEMPIO DI AMBIENTI E REGOLE DI VISIBILITÀ



main ()
- g
- a, b
- f1, f2

blocco b1
- g
- b (di main)
- a, c
- f1, f2

funzione f1()
- g
- d
- f1, f2

blocco b2
- g
- d (di f1)
- e
- f1, f2

blocco b3
- g
- d (di b3)
- f1, f2

funzione f2()
- g
- f
- f1, f2