Il Linguaggio C

- Il linguaggio compreso dalla CPU è il **linguaggio macchina** ovvero una sequenza di numeri binari.
- Il linguaggio ASSEMBLER è leggermente più evoluto nel senso che le istruzioni sono formate da insieme di caratteri, ma c'è ancora una **corrispondenza biunivoca** tra istruzione assembler e istruzione linguaggio macchina.
- I linguaggi più evoluti (Fortran, Pascal, C,C++) sono **compilati:** cioè ogni istruzione è tradotta in una sequenza di istruzioni in linguaggio macchina.
- Noi tratteremo il linguaggio ANSI C (American National Standard Institute)
- Un programma scritto in ANSI C è **portabile** su qualsiasi computer dotato di compilatore C, non dipende cioè dall'architettura.

Variabili, identificatori,tipi

•Per **variabile** si intende un'area della memoria RAM nella quale può essere conservato un valore.

•Una **variabile** è identificata da un nome detto appunto **identificatore** ovvero una sequenza di "lettere (maiuscole o minuscole) + _ (underscore) + cifre".

•Un identificatore può iniziare solo con una lettera, non con una cifra (è meglio evitare anche _ come primo simbolo).

•Un identificatore di una variabile non può essere una parola chiave (vedere tabella) perchè queste hanno un particolare significato.

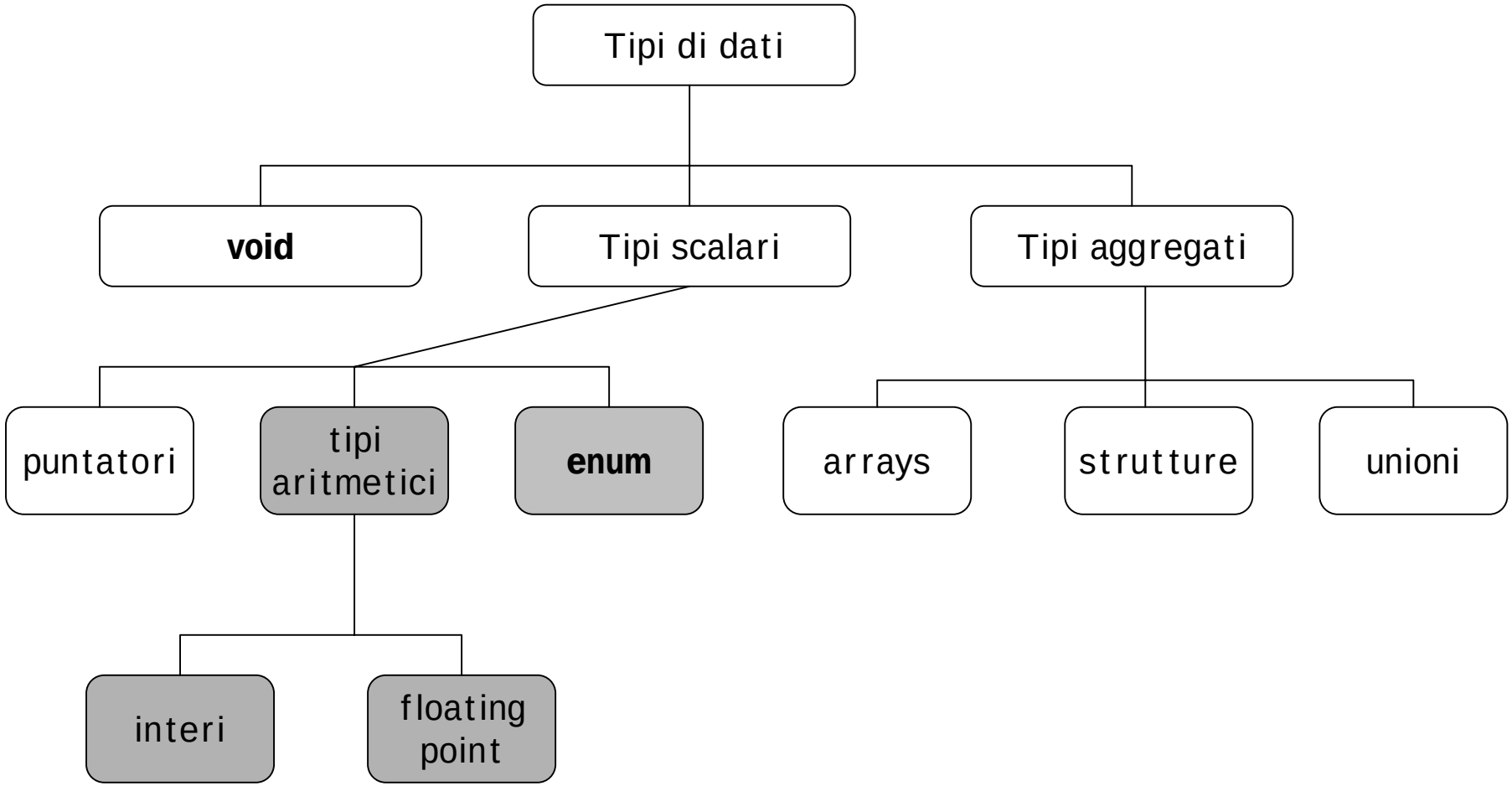
•Non è sufficiente attribuire un nome ad una variabile per poter memorizzare un valore, ma occorre definirne il **tipo.**

•Il tipo delle variabili numeriche è fondamentale nel calcolo scientifico; infatti ad ogni tipo è associato un **range** di valori rappresentabili.

Parole chiave

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

Esistono molti altre parole riservate che non possono essere usate come identificatori di variabili: funzioni delle librerie (per esempio sin, cos, etc). In caso di dubbio si possono consultare le tabelle apposite sui manuali o sui libri di testo, vale sempre la regola di personalizzare i nomi delle variabili, in modo che ricordino il significato del contenuto.


Tipi di dati
void
Tipi scalari
Tipi aggregati
puntatori
tipi aritmetici
enum
arrays
strutture
unioni
interi
floating point

TIPI aritmetici e loro RANGE

Interi

TIPO	BIT	BYTE	Contenuto	Range
char unsigned char	8 8	1 1	Carattere ASCII	-127 – 127 0 - 255
short int	16	2	intero con segno	-2^15– 2^15-1
int	32	4	intero con segno	-2^31 – 2^31-1
long int	32	4	intero con segno	-2^31 – 2^31-1
unsigned short	16	2	intero senza segno	0- 2^16-1
unsigned int	32	4	intero senza segno	0- 2^32-1
unsigned long	32	4	intero senza segno	0- 2^32-1

Floating-point

TIPO	BIT	BYTE	Valore minimo (non standard)	Valore massimo (non standard)
float	32	4	1.1755 10^-38	3.403 10^38
double	64	8	2.2251 10^-308	1.7977 10^308
long double	80	10	3.3621 10^-4932	1.1897 10^4932

Oltre ai numeri interi e ai puntatori (che vedremo in seguito) i tipi scalari comprendono anche i tipi enumerativi. Solitamente non vengono molto usati per il calcolo scientifico, ma a volte servono come supporto.

I tipi enumerativi sono utili quando si associa ad una particolare variabile un insieme di valori possibili:

```
enum {verde,giallo,rosso,blu} colore;
colore= giallo;          /* è corretto          */
colore=1 ;               /*  dà conflitto di tipo*/
```

Le costanti numeriche seguono delle regole simili a quelle delle variabili:

55 è una costante intera

55L è una costante intera di tipo Long int

55U è una costante intera di tipo unsigned int

55.; 55.0 ; 3.e-5 ; 2.e13 sono costanti di tipo floating-point (default= double)

Sono esempi errati di costanti float i seguenti numeri:

354 ; 42,100.00 ;4e ; 5e1.3

Dichiarazioni di tipo

Tutte le variabili usate in un programma vanno **dichiarate** e se è necessario possono essere **inizializzate** ad un valore.

```
int i,j;

float radius,sum=0.;

double area;

char  carattere=65,car='A';
```

Espressioni

Un' espressione è una combinazione di operatori , numeri e nomi che permette di calcolare un valore. Sono espressioni valide:

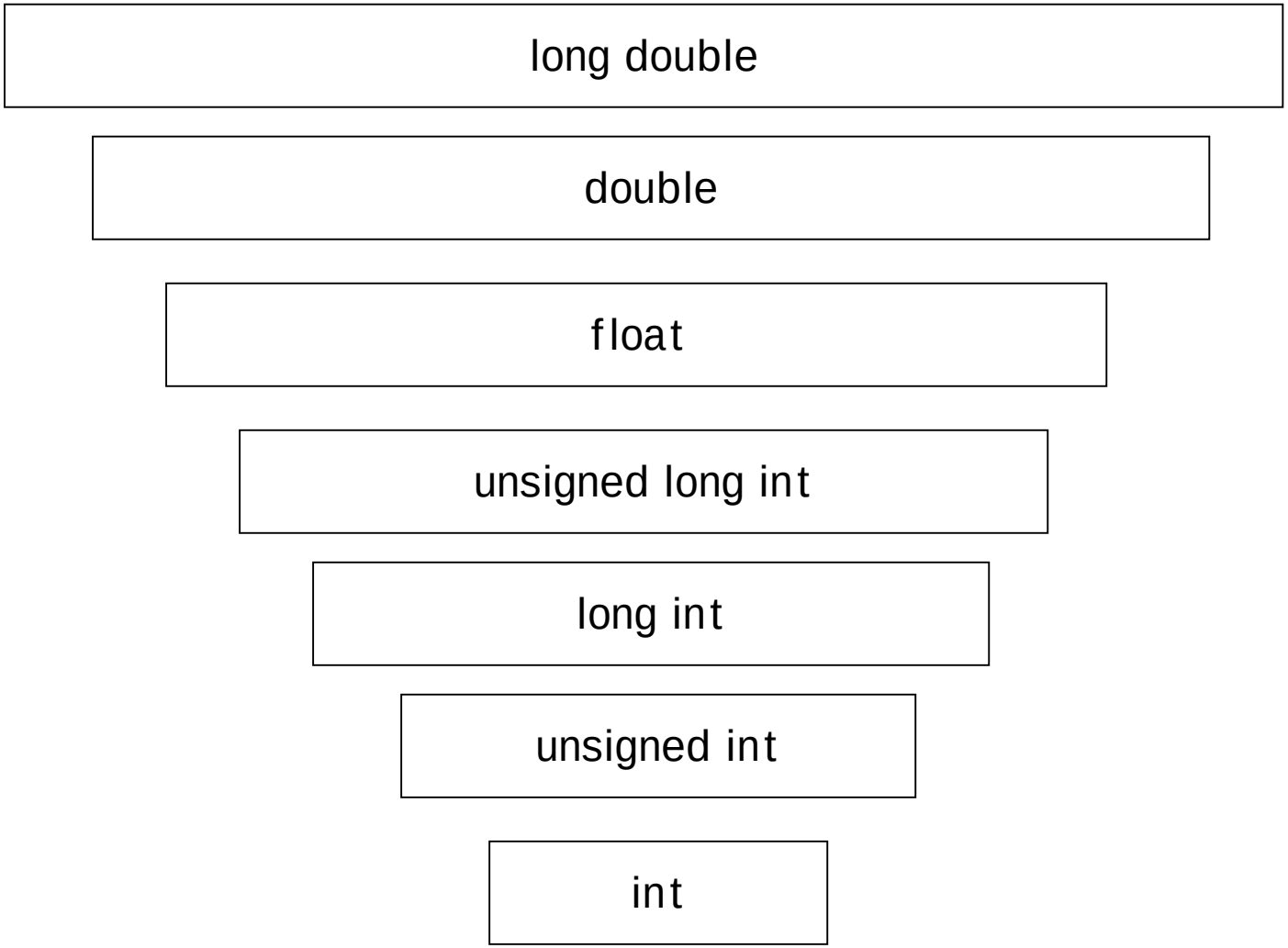
5 costante

j variabile

5+j costante + variabile

5*j+7 costante moltiplicata ad una variabile + costante

Gerarchia dei tipi di dati



Operatore di assegnamento

variabile = espressione ;

Questa istruzione ha il seguente significato: viene valutata l'espressione, il valore risultante viene memorizzato nell'area di memoria identificata dalla variabile.

Quindi abbiamo la parte destra (**rvalue**) che è un valore e la parte sinistra (**lvalue**) che corrisponde ad un indirizzo di memoria.

x=10;

x=x+5; ← ha senso perchè si legge da sinistra a destra

x*x=3.; ← non è corretta perchè x*x non è un lvalue (ovvero un indirizzo)

Conversioni di tipo implicite e casting

Il linguaggio permette di combinare tipi aritmetici nelle espressioni con pochi vincoli, operando nel caso si trovi con costanti e/o variabili di tipo diverso una conversione automatica che va capita e sperimentata per evitare errori banali, e perdite di tempo.

Esiste una gerarchia tra i tipi aritmetici: se si combinano tipi differenti il risultato dell'espressione è convertito al tipo con gerarchia più alta. Per esempio:

float x,y;
int i;
x=y/i;

ATTENZIONE!
→

float x;
int n=256;
x=1/n; **→** **x=1.0/n;**

Questo in un certo senso può far risparmiare chiamate a funzioni come la parte intera, ma può essere pericoloso se non lo si controlla.

Effetti indesiderati di perdita di informazione si superano operando un “casting” cioè una conversione esplicita del tipo di una variabile.

```
float x;

int i=1,n=256;

x= (float) i/n;
```

Operatori di casting più utilizzati

```
(float) variabile;

(float) (espressione);

(int) variabile;

(int) (espressione);
```

Printf e scanf

Printf e scanf sono funzioni della libreria standard di input/output.

Sono utilizzate per mandare informazioni sullo standard output (video) o ricevere dati dallo standard input (tastiera).

Qui viene presentata la sintassi semplificata con alcuni esempi, per una descrizione più dettagliata potete fare riferimento a qualsiasi manuale di C.

Per utilizzare queste funzioni occorre all'inizio del file che contiene il programma mettere la seguente direttiva al pre-processsore

```
#include<stdio.h>  /*Attenzione nelle istruzioni al pre-processore non c'è il ; */
main () {
float x=100.0;
int i=2;
char c1='A';
/*printf("stringa che descrive il formato/i \n",variabile1,variabile2,...);*/
printf(" Il valore della variabile x è: %f \n",x);
printf(" valore di i (intera): %d \n",i);
printf(" carattere :%c \n",c1);
printf(" i:%d, x:%f \n",i,x);    }
```

Più delicata è la funzione scanf, infatti il valore che viene digitato deve essere registrato nell'area di memoria corrispondente alla variabile in questione.

Questa viene indirizzata ponendo il carattere & davanti all'identificatore della variabile.

```
float a,b,c;
main (){
printf("Programma per il calcolo delle radici di un'equazione di secondo grado \n");
printf(" Dammi i valori in ordine di a,b,c: "); /* non va a capo*/
scanf("%f %f %f",&a,&b,&c);
printf(" a= %f , b= %f ,c =%f \n",a,b,c); /* verifico  */
......
 }
```

se la variabili fossero invece double occorre cambiare il formato di input (%lf =long float)

```
scanf("%lf %lf %lf",&a,&b,&c);
```

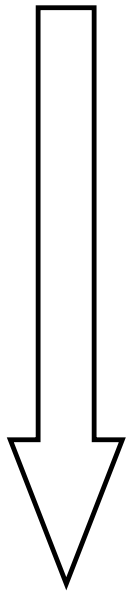
La struttura di un programma C

```
#include <stdio.h>      /* direttiva al pre-processore */
#define PI 3.14159      /* direttiva al pre-processore */

main ()
{
istruzione;
/* commento */
/* commento
su più
righe */
}
```

OPERATORI

(esistono altri operatori, guardate sui manuali)

precedenza



•moltiplicazione e divisione	***,/**
•somma e sottrazione	**+,-**
•relazionali	**<,<=,>,>=**
•uguaglianza	**==,!=**
•AND logico	**&&**
•OR logico	**||**
•NOT logico	**!**
•assegnamento (*)	**=**

Per la valutazione di una espressione vengono eseguite prima le operazioni con precedenza più alta, a parità di precedenza le espressioni che contengono questi operatori (escluso l'assegnamento) sono valutate da sinistra a destra.

Esempio:

a=b+c-d; è equivalente a **a=(b+c)-d;**

a=b=c; è invece equivalente a **b=c;**

a=b;

Notazione compatta, incremento e decremento

```
a=a+5;   →   a+=5;
a=a-5;   →   a-=5;
a=a/5;   →   a/=5;
a=a*5;   →   a*=5;
```

```
a=a+1;   →    a++;
a=a-1;   →    a--;
```

Il controllo di flusso

Le istruzioni in un programma C vengono eseguite in ordine di apparizione (programmi sequenziali).

A volte è utile eseguire blocchi di istruzioni differenti in funzione di alcune condizioni oppure ripetere un insieme di istruzioni per molte volte.

In questi casi il flusso delle istruzioni non è sequenziale e si distinguono: la selezione condizionale e i cicli.

Selezioni condizionali

Per effettuare le selezioni condizionali si usano le istruzioni:

if ... else

switch (per questa istruzione consultare i manuali)

Sintassi dell'istruzione if

```
if (espressione)
        istruzione 1;    /* eseguita se espressione diversa da 0 */
else
        istruzione 2;    /* eseguita se espressione=0 */
istruzione 3;        /* sempre eseguita */
```

Se la scelta è una solamente

```
if (espressione)
        istruzione 1;    /* eseguita se espressione diversa da 0 */
istruzione 3;        /* sempre eseguita */
```

Usando la tabulazione per incolonnare le istruzioni il codice risulta più leggibile.

Se le istruzioni per ogni condizione sono più di una devono essere racchiuse tra parentesi graffe.

Attenzione a non confondere == con = !!!

Esempio

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h> /* serve per utilizzare le funzioni matematiche  */
main ()
{
double numero;
printf("Dammi un numero positivo\n");
scanf("%lf",&numero);
if (numero < 0)
        {
         printf(" Il numero è negativo\n");
         exit(0);
         }
printf("La radice quadrata di %lf è %lf\n",num,sqrt(num));
}
```

Cicli

I cicli (loop o iterazioni) permettono di eseguire un blocco di istruzioni fino a che non sia verificata una certa condizione.

Le istruzioni che permettono di fare ciò sono tre:

while

do ... while

for

Sintassi dell'istruzione while

```
while (espressione)
        istruzione;
```

```
while (x<y)
        x++;
```

La singola istruzione può essere sostituita da un blocco di istruzioni tra parentesi graffe.

Sintassi dell'istruzione do ...while

```
do
        istruzione;
while (espressione);
```

```
do
    x++;
while(x<y);
```

La singola istruzione può essere sostituita da un blocco di istruzioni tra parentesi graffe.

La differenza tra **while** e **do ... while** è che nella prima il controllo sull'espressione viene fatto all'inizio mentre nella seconda alla fine; quindi se l'espressione è falsa, nel primo caso l'istruzione/i non è **mai** eseguita, mentre nel secondo è eseguita almeno una volta.

Queste istruzioni sono spesso utilizzate per leggere dati da tastiera o da file, senza sapere quanti sono.

La sintassi dell'istruzione for

for (espr1;espr2;espr3)
istruzione;

for (i=0;i<100;i++)
printf("i=%d\n",i);

Attenzione l'errore più comune è mettere un **;** al termine del comando **for**, questo provoca un ciclo che non **FA NULLA.**

Anche se la sintassi dell'istruzione **for** può sembrare molto generica occorre fare alcune precisazioni:

- Esiste un indice del ciclo (in questo caso è la variabile intera **i**)
- La prima espressione inizializza l'indice del ciclo (**i=0**)
- La seconda è una espressione logica che verifica quando il ciclo termina (**i<100**)
- La terza dà la regola di incremento o decremento dell'indice di ciclo
- NON è mai opportuno MODIFICARE l'indice di ciclo.

Esempio: calcolo del fattoriale

```
int j,val;
long fact=1L;
printf("Digita un numero intero:");
scanf("%d",&val);
for (j=2;j<= val<; j++)
        fact*=j;
printf("%d\n",fact);
```

Esercizio: provate a valutare il massimo valore di **val** nel caso avessi definito la variabile **fact** del tipo **short int**.

Cicli annidati

```
/* Stampa una tavola pitagorica sfuttando i cicli annidati */
int j,k;
printf("     1  2  3  4  5  6  7  8  9  10\n");
printf("   ---------------------------\n");
for (j=1;j<=10,j++)  /*ciclo esterno */
      {
       printf("%5d|",j);
       for (k=1;k<=10;k++)  /*ciclo interno */
             printf("%5d", j*k);
       printf("\n");
       }
```

Cicli infiniti

A volte sorge la necessità di creare un ciclo per eseguire un programma “all’infinito”, ovvero fino a che l’utente lo interrompe.

A volte i cicli infiniti sono dovuti ad errori di programmazione, vediamo alcuni esempi:

```
while(1)
        istruzione;
for(;;)
        istruzione;
for (j=0;j<10;j++)
        {
        .....
        j=1;    /* si modifica l’indice del ciclo */
        }
```

Array e puntatori

Anche se gli array e i puntatori appartengono a tipi di dati differenti, i primi infatti sono un tipo di dato aggregato e i secondi sono di tipo scalare, sono strettamente collegati e costituiscono una delle caratteristiche più potenti del linguaggio C.

Per **array** si intende un insieme di variabili dello stesso tipo memorizzate consecutivamente. E' possibile accedere ad una variabile di un array, detta **elemento** attraverso una espressione detta indice (subscript). Per default l'indice 0 indica il primo elemento, l'indice 1 l'elemento successivo etc..

L'array hanno lo scopo di trattare molti dati correlati tra loro, tipicamente una serie di misure.

Dichiarazione e inizializzazione

Specificatore di tipo **nome dell'array** **[dimensione]= {valori iniziali};**

float temperature[4]={10,20,30,40};

Puntatori

Abbiamo già visto nell'istruzione **scanf,** come si identifica un indirizzo di una variabile (&nome_variabile), il valore di questo indirizzo di memoria può essere assegnato ad un variabile di tipo **puntatore.**

Esempio:

```
int j=1, *ptr; /* puntatore e dato devo avere lo stesso tipo */
printf("Il valore di j è %d\n",j);
printf("L'indirizzo di j è %p\n",&j); /* %p è il formato dei puntatori */
ptr=&j;
printf("L'indirizzo di j è %p\n",ptr);
```

L'asterisco viene usato per identificare il **contenuto** dell'area di memoria, "puntata" dal puntatore. Nel caso precedente se io assegno

```
*ptr=2;
printf("%d\n",j);  /* da' in uscita il valore 2*/
```

Se ci si ferma a queste considerazioni l'utilizzo dei puntatori sembra solo una complicazione, ma il linguaggio C definisce una aritmetica dei puntatori:

Se **ptr** è un puntatore **ptr+3** è anch'esso un puntatore che individua il terzo oggetto che segue quello individuato da **ptr.**

E' chiaro che anche in questo caso è fondamentale il tipo del puntatore, oltre ad essere coerente con il dato si possono sommare o sottrarre due puntatori solo se **entrambi** puntano allo stesso tipo di dato.

```
int *ptr1,*ptr2;

int j;

ptr2=ptr1+4;

j=ptr2-ptr1;    /* j assume il valore 4 */

j=ptr1-ptr2;   /* j assume il valore  -4 */
```

Esiste il puntatore nullo (che viene utilizzato per i controlli di flusso) e si può definire:

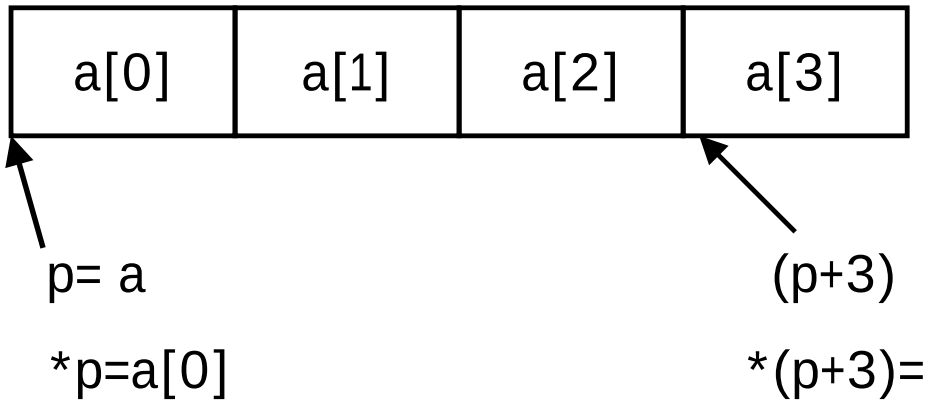
```
int *p=0;
```

Viene spontaneo pensare all'uso dei puntatori per poter accedere agli elementi di un array:

```
short a[4];

short *p;

p=&a[0];  /* quindi *p contiene il valore di a[0] */

a[1]=*(p+3); /* assegno il valore a[3] ad a[1] */
```

Inoltre il nome di un array che non è seguito da un indice viene interpretato come il puntatore all'elemento iniziale dell'array.

a è equivalente a **&a[0]** e nell'esempio sopra è equivalente a **p**



Array a più dimensioni

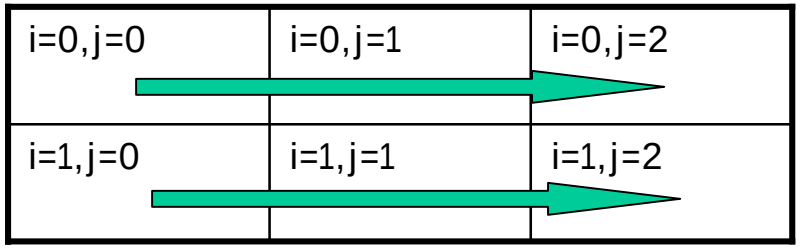
Finora abbiamo visto esempi di array ad una dimensione, in C si possono definire array a più dimensioni (senza limite sul numero di dimensioni):

float matrice[2][3];

alloca in memoria una matrice con 2 righe e 3 colonne , ogni elemento è di tipo float e viene indicato da una coppia di indici. Per mantenere efficente l'accesso alla memoria bisogna **considerare** come sono memorizzati i valori (scorre più velocemente l'indice di destra):

```
for (i=0;i<2;i++)
        for(j=0;j<3;j++)
                printf("matrice[%d][%d]=%f \n",i,j,matrice[i][j]);
```

stampa il contenuto della matrice, mantenendo l'ordine in memoria.



L'inizializzazione di un array a più dimensione si può fare in fase di dichiarazione: vediamo un quadrato "magico" (somma di ogni riga, somma di ogni colonna, somma di ogni diagonale è costante):

int magic[5][5]={{17,24,1,8,15},{23,5,7,14,16}, {4,6,13,20,22}, {10,12,19,21,3},
{11,18,25,2,9}};

ogni riga è racchiusa tra parentesi graffe.

ESERCIZIO: Verificare che l'array sopra indicato rappresenti effettivamente un quadrato magico (che valore ha la costante?)

Array multidimensionali e "puntatori a puntatori "

Un array multidimensionale è un array di array, nell'esempio sopra **magic** può essere visto come un array di 5 righe; ogni riga è un array unidimensionale di dimensione 5.

Quindi **magic[i]** è il nome del vettore unidimensionale che occupa la riga i ed è quindi un puntatore alla riga i: **magic[i][j]** equivale a ***(magic[i]+j)**

D' altra parte **magic** (nome dell'array 2d) è un puntatore a puntatore e quindi posso dire che **magic[i][j]** è equivalente a ***(*(magic +i)+j)**

Se definisco un puntatore a puntatore nel modo seguente

int **ptr;

posso assegnare al puntatore a puntatore il nome della matrice → **ptr=magic;**

Le funzioni

Uno degli strumenti più versatili e più utilizzati nel linguaggio C, sono le funzioni.

Alcune funzioni sono contenute in librerie (stdio,stdlib, math etc..) e possono essere chiamate dall'utente rispettando la loro sintassi. E' più interessante il caso in cui l'utente definisce le funzioni da utilizzare o all'interno del proprio codice o all'esterno. Consideriamo il primo caso attraverso un semplice esempio:

```
#include <stdio.h>

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

main( )
     { int ris;
       ris=somma(5,6);
       printf("risultato %d\n",ris);
     }
```

tipo_del_risultato nome_funzione(tipo_argomento nome,..)
{ *corpo della funzione;*
return (risultato); }

Le funzioni di tipo void

Una funzione può anche non restituire un valore in questo caso il tipo della funzione è void, più in generale inidca un puntatore generico.

```
#include <stdio.h>
void funzione(int a,int b)
        { ....   ; }
```

void nome_funzione(tipo_argomento nome,..)
{ *corpo della funzione;* }

```
main( )
      { int par1=0,par2=1;
        funzione (par1,par2);
      }
```

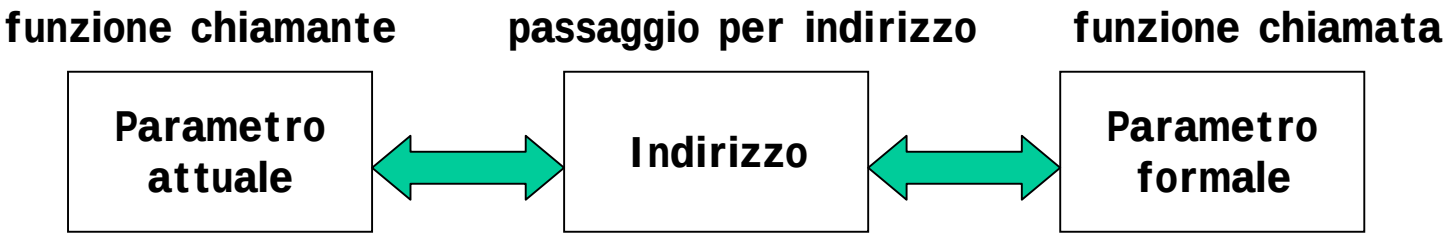
Una funzione deve essere definita prima di essere utilizzata.

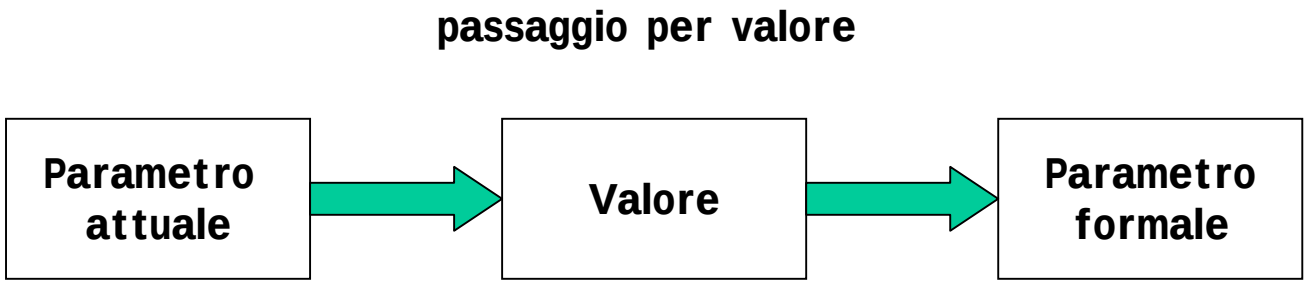
Esercizio: Convertite il vostro primo programma C, quello che stampava un messaggio, in un funzione di tipo void (oltre alla funzione main).

Osservazione: anche main è una funzione, è obbligatoria in ogni programma e può avere o non avere argomenti. Se non è specificato è di tipo intero e la scritta **main ()** equivale a **int main (void)** (in questo caso main non ha argomenti, void =puntatore generico).

Passaggio degli argomenti

Gli argomenti di una funzione rappresentano una modalità per passare dati alla funzione stessa. Molti linguaggi di programmazione passano argomenti per indirizzo (by reference), il C passa gli argomenti per valore (by value) vediamo la differenza





La differenza sostanziale è che nel primo caso vengono modificati sia il parametro attuale, sia quello formale; mentre nel secondo viene modificato solo quello formale.

Se si vuole che la funzione modifichi il parametro attuale si deve passare il puntatore alla variabile in questione.

Vediamo un esempio molto semplice di passaggio di parametri di tipo puntatore:

```
#include <stdio.h>
#include <stdlib.h>
void swap (int *x,int *y) /* funzione che non restiruisce alcun valore */
{
int temp;       /*variabile temporanea definita solo internamente alla funzione */
temp=*x;
*x=*y;
*y=temp;
 }
main () {
int a=2,b=3;
swap(&a,&b);
printf("a= %d   b=%d\n",a,b);
}
```

Passaggio di array come argomenti di funzioni

Nel linguaggio C, un nome di un array utilizzato come argomento di funzione viene interpretato, ovviamente, come l'indirizzo del primo elemento dell'array.

Nella funzione *chiamata* è necessario dichiarare l'argomento come un puntatore all'elemento iniziale del vettore e questo si può fare in due modi:

void func(float *ar)

oppure

void func(float ar[ ]) /* nome dell'array senza dimensione */

Mentre nella funzione *chiamante* si utilizzerà l'istruzione passando il parametro

che sarà stato allocato in memoria con una determinata dimensione.

float array[5]; ....

func(array);

Occorre porre attenzione che in questo modo l'informazione sulla dimensione del vettore all'interno della funzione è persa e quindi, quando è necessario, occorre passare la dimensione come ulteriore parametro.

Esempio

```
#include ....
int somma(int *a,int size) /* parametri: puntatore al vettore e sua dimensione */
{
int sum=0;
for (i=0;i<size;i++)
        sum+=a[i];
return(sum);            /* restituisce la somma degli elementi di un vettore */
 }
main() {
int vettore[3]={ 1,2,3 };
printf("%d\n",somma(vettore,3));
}
```

Nell'esempio sopra alla funzione somma occorre passare l'indirizzo del primo elemento del vettore e la sua dimensione.

Istruzione sizeof

L'istruzione **sizeof(espressione)** restituisce un intero che corrisponde al numero di byte occupati in memoria dall'espressione tra le parentesi.

L'esercizio seguente tende a chiarire il passaggio dei parametri nelle funzioni in C:

```
#include <stdio.h>
#include <stdlib.h>
void stampa_dim(float arg[ ])   funzione che stampa la dimensione dell'argomento
{ printf("La dimensione dell'argomento è: %d\n", sizeof(arg) ); }
main()
{
float vettore[10];
printf("La dimensione di vettore è: %d\n",sizeof(vettore));
stampa_dimensione(vettore);
}
La dimensione di vettore è: 40
La dimensione dell'argomento è: 4
```

Stringhe

•Una stringa è un array di caratteri terminati dal carattere nullo=**'\0'**

•Una stringa costante è definita da una serie di caratteri racchiusi tra doppi apici.

•Esempio di dichiarazione di una stringa:

char mia_stringa [ ]="esempio di testo"; /* questo array ha dimensione 17 (16+\0)*/

•Le stringhe possono essere scritte e lette con le istruzioni **printf** e **scanf** usando lo specificatore di formato **%s**

•Per quanto riguarda scanf l'argomento deve essere il puntatore ad un vettore di caratteri di dimensioni sufficienti a contenere la stringa (il carattere di terminazione viene aggiunto automaticamente).

•Per quanto riguarda printf l'argomento deve essere il puntatore ad una array di caratteri terminato dal carattere nullo. Vediamo un esempio:

```
#define max_char 80
main (){
char stringa[max_char];
printf("introdurre una stringa:");
scanf("%s",stringa);          /*legge la stringa da tastiera */
printf("%s\n",stringa);    /* La riscrive sul video */
}
```

Funzioni per la gestione delle stringhe (includere <string.h>)

•**char *strcat(char *s1, const char *s2);** *aggiunge una copia della stringa s2 alla stringa s1, sostituendo il carattere nullo di terminazione di s1 con il primo carattere di s2, restituisce il valore di s1.*

•**int strcmp(const char *s1,const char *s2);** *confronta la stringa s1 con s2, restituisce un numero positivo se s1>s2, negativo se s1<s2, se s1=s2 la funzione restituisce 0.*

•**char *strcpy(char *s1, const char *s2);** *la stringa s2 è copiata nell'array puntato da s1, la funzione restiruisce il valore di s1.*

•**size_t strlen(const char *s);** *restituisce la lunghezza in byte della stringa puntata da s (escluso il terminatore di stringa).*

Queste funzioni vengono chiamate per esempio, nel seguente modo:

```
char stringa1[3]="aa",stringa2[3]="ab";

printf("%s\n",strcat(stringa1,stringa2));

printf("%d\n",strcmp(stringa1,stringa2));

printf("%d\n",strlen(stringa1));
```

I/O da files

•Finora abbiamo visto l'input e l'output dei dati dalla tastiera (standard input) e sul video (standard output); è ovvio che questo può essere molto scomodo se si devono inserire molti dati. Le stesse operazioni di I/O si possono eseguire utilizzando files. Vediamo alcune definizioni:

•Si dice "stream" una sorgente o una destinazione di dati.

•Uno stream viene connesso ad un file o a un device tramite l'operazione di apertura

•L'operazione di chiusura termina questa connessione

•L'apertura di un file fornisce come risultato un *puntatore a file.*

```
#include <stdio.h>
main(){
file *f1,*f2;
f1=fopen("nome_del_file_input","r");     /* deve esistere r=read */
f2=fopen("nome_del_file_output","w"); /* se non esiste viene creato w=write */
/* fscanf(f1,"specificatore di formato",&variabile1,...);  legge da file
    fprintf(f2,"specificatore di formato",variabile1,...); scrive sul file  */
fclose(f1);
fclose(f2);}
```

Allocazione dinamica

Fino a questo punto abbiamo trattato le variabili e, in particolare i vettori, allocati in memoria in modo *statico*, ovvero una determinata variabile o un vettore occupavano nel corso del programma un numero fissato di bytes determinato dalla dichiarazione.

E' chiaro che in alcuni casi, quando per esempio non si sa quanti dati si deve trattare, è molto più versatile definire nel corso del programma la dimensione di una variabile.

Ciò è possibile con l'allocazione dinamica. I prototipi delle funzioni dedicate a questo scopo sono in stdlib.h e quindi per utilizzarle occorre includere questo file.

Le funzioni sono malloc() e free(). La prima alloca la variabile in memoria occupando il numero di byte passati come parametro e restituisce il puntatore, free libera la memoria allocata con malloc, quando non è più utilizzata.

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

int dim;

float *vec;  /*puntatore a vettore */  in modo statico  float vec[dim]
                                                                  ?

printf(" dammi la dimensione del vettore:");

scanf("%d",&dim);

vec=(float*)malloc(dim*sizeof(float)); /* allocazione più casting */

....  /* il vettore vec si può utilizzare nel solito modo */

free(vec);  /*libero la memoria */
```

Le strutture

Le strutture sono utilizzate per raggruppare più dati che hanno un legame logico.

Per esempio un numero complesso è rappresentato dalla sua parte reale e dalla sua parte immaginaria, un atomo è caratterizzato dal nome, dal simbolo, dal numero atomico, dalla massa atomica e dalla vita media.

•Definizione di una struttura (**struct** è la parola chiave per la definizione,atomo e complex sono i nomi di queste due strutture)

```
struct atomo {
        char *nome;
        char * simbolo;
        int  numero_atomico;
        float massa_atomica;
         };
struct complex {
         float reale;
         float imm;
         };
```

I campi delle struttura una volta dichiarati non possono essere modificati.

I dati di tipo struttura si dichiarano nel modo seguente:

struct atomo idrogeno;

struct complex numero;

idrogeno è una struttura di tipo atomo con con 4 campi. Ogni campo può essere indirizzato nel seguente modo:

idrogeno.nome= “idrogeno”;

idrogeno.simbolo=“H”;

idrogeno.numero_atomico=1;

idrogeno.massa_atomica=1.008 ;

numero è una struttura di tipo complex con 2 campi.

numero.reale=1.0;

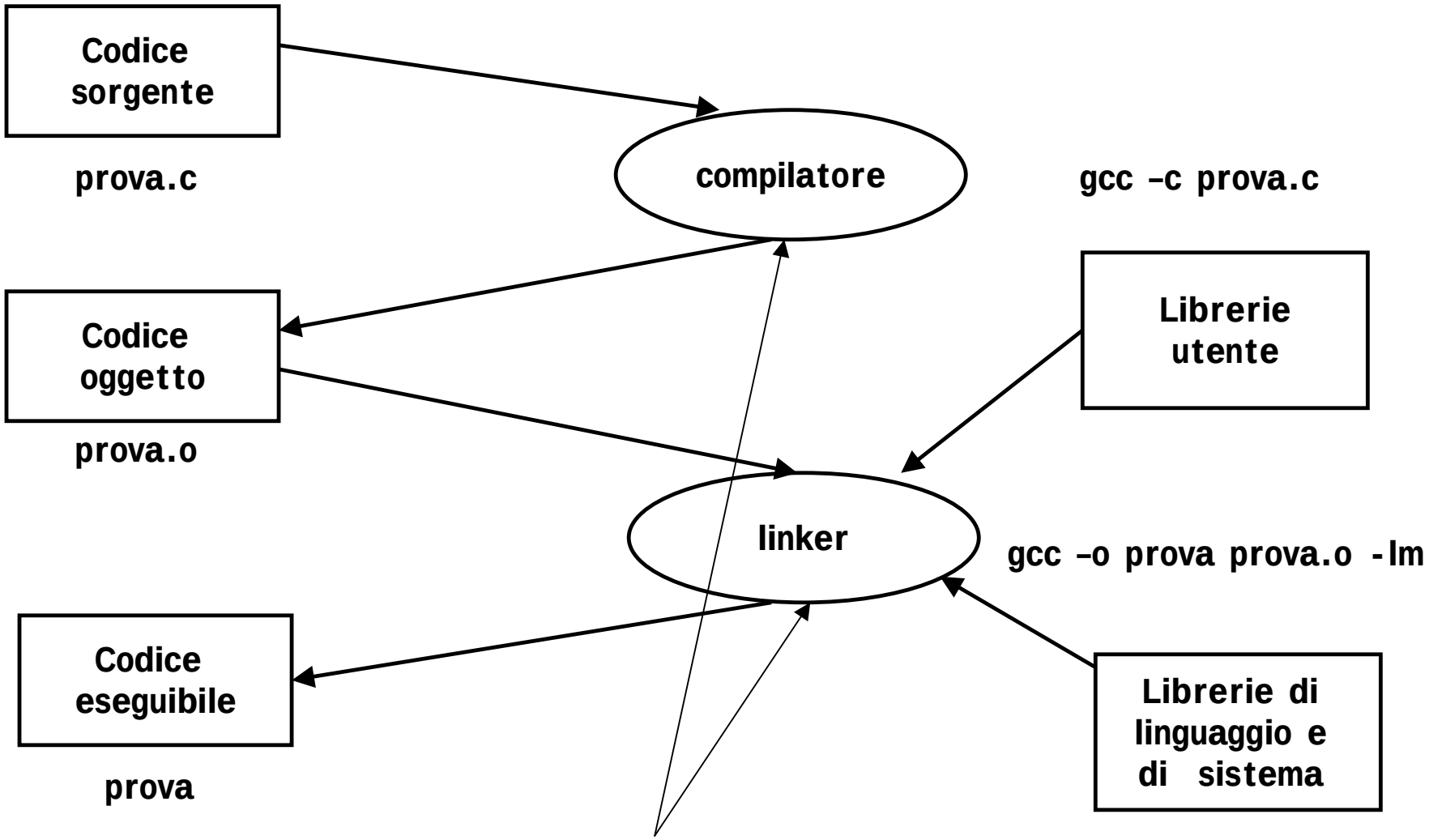
numero.imm=2.0;

definisce il numero complesso 1 + 2 i

Esercizio: Create alcune funzioni che eseguono le operazioni aritmetiche sui numeri complessi.

```
#include <stdio.h>
#include <stdlib.h>
 struct complex {
    float re;
    float im;
 };
struct complex somma(struct complex a, struct complex b)
{  struct complex somma;
     somma.re=a.re+b.re;
     somma.im=a.im+b.im;
     return (somma);}
main ()
{  struct complex c1,c2,c3;
  c1.re=1.0;    /* non è possibile inizializzare una struttura nella dichiarazione */
  c1.im=0.;
  c2.re=0.;
  c2.im=2.;
   c3=somma(c1,c2);
   printf("%f %f \n",c3.re, c3.im);
 }
```

La compilazione



Appendice: la libreria matematica (math.h)

```
#define M_E             2.7182818284590452354
#define M_LOG2E         1.4426950408889634074
#define M_LOG10E        0.43429448190325182765
#define M_LN2           0.69314718055994530942
#define M_LN10          2.30258509299404568402
#define M_PI            3.14159265358979323846
#define M_TWOPI         (M_PI * 2.0)
#define M_PI_2          1.57079632679489661923
#define M_PI_4          0.78539816339744830962
#define M_3PI_4         2.35619449019234483700E0
#define M_SQRTPI        1.77245385090551602792981
#define M_1_PI          0.31830988618379067154
#define M_2_PI          0.63661977236758134308
#define M_2_SQRTPI      1.12837916709551257390
#define M_SQRT2         1.41421356237309504880
#define M_SQRT1_2       0.70710678118654752440
#define M_LN2LO         1.90821492927058770002E-10
#define M_LN2HI         6.93147180369123816490E-1
#define M_SQRT3         1.73205080756887719000
#define M_IVLN10        0.43429448190325182765 /* 1 / log(10) */
#define M_LOG2_E        0.69314718055994530942
#define M_INVLN2        1.44269504088896338700E0  /* 1 / log(2) */
```

double sin(double x);	**seno (argomento in radianti)**
double sinh(double x);	**seno iperbolico**
double asin(double x);	**arcoseno: il risultato è nell'intervallo [-pi/2,pi/2]**
double cos (double x);	**coseno**
double cosh (double x);	**coseno iperbolico**
double acos (double x);	**arcoseno: il risultato è nell'intervallo [-pi/2,pi/2]**
double tan (double x);	**tangente**
double tanh (double x);	**tangente iperbolica**
double atan (double x);	**arcotangente:il risultato è nell'intervallo [-pi/2,pi/2]**
double exp (double x);	**esponenziale di x**
double log (double x);	**logaritmo in base e di x**
double log10 (double x);	**logaritmo in base 10 di x**
double pow (double x,double y);	**x** y**
double sqrt (double x);	**radice quadrata non negativa**
double fabs (double x);	**valore assoluto di x**
double floor (double x);	**più grande intero non maggiore di x**
double ceil (double x);	**più piccolo intero non minore di x**