Qualsiasi programma in C++ segue lo schema:

```
#include <iostream>  // libreria che gestisce flusso di input e output

using namespace std; // uso di librerie standard del C++

int main()
{
  // dichiarazioni delle variabili utilizzate
  ...
  // istruzioni del programma
  ...
  return 0;
}
```

Variabili

Le istruzioni racchiuse tra parentesi graffe { } costituiscono un **blocco** di elaborazione. La variabili dichiarate all'interno di un blocco sono **locali**, mentre quelle dichiarate fuori da qualsiasi blocco sono **globali**

Tipi di variabili numeriche predefiniti

int, unsigned int, long, unsigned long, float, double, long double

quanto spazio occupano in memoria:

tipo	**Bytes**	**Range**	**Precisione (decimale)**
int	2	–32768 a 32767	..
long	4	$-2x10^9$ a $2x10^9$	..
unsigned int	2	0 a 65535	..
unsigned long	4	0 a $4x\ 10^9$	..
float	4	10^{-38} a 10^{38}	7
double	8	10^{-308} a 10^{308}	15
long double	10	10^{-4932} a 10^{4932}	19

altri tipi di variabili predefiniti

char, bool (può assumere solo i 2 valori true/false)

N.B

- **const** int a = 4; (a non può più essere rassegnata)
- **typedef** float Real; (Real diventa sinonimo di float)

Operatori

+	**addizione**
-	**sottrazione**
*	**moltiplicazione**
/	**divisione**
%	**modulo**
++	**incremento**
--	**decremento**
=	**assegnazione**

assegnazione multipla: posso assegnare lo stesso valore a più variabili

var1 = var2 = var3 = valore;

assegnazione composta: si eseguono i calcoli prima dell'assegnazione:

x += y;	è come	x = x + y ;
i += 2;	è come	i = i+2;

N.B. incremento: per incrementare di 1 la variabile z si può scrivere:

z ++ oppure ++ z

cioè mettere l'operatore ++ prima o dopo del nome della variabile. Le due forme non sono sempre equivalenti. La differenza si esplicita quando si valuta una *espressione* che contiene z++ o ++z.
Scrivendo z++, il valore di z viene *prima usato poi incrementato*.
Scrivendo ++z, il valore di z viene *prima incrementato e poi usato*.
Basta tenere presente che l'ordine delle operazioni avviene sempre da sinistra verso destra.

l'operatore ?

L'operatore di assegnazione condizionata **?** ha la seguente sintassi:
espressione_logica ? espr1 : espr2
Se espressione_logica è **vera** restituisce **espr1 altrimenti** restituisce **espr2.**
Si utilizza tale operatore per assegnare, condizionatamente, un valore ad una variabile.

Esempio |x|
valore_assoluto = x>0 ? x : -x;

Tavola riassuntiva

Operatore	Esempio	Risultato
!	!a	(NOT logico) 1 se a è 0, altrimenti 0
<	a < b	1 se a<b, altrimenti 0
<=	a <= b	1 se a<=b, altrimenti 0
>	a > b	1 se a>b, altrimenti 0
>=	a >= b	1 se a>=b, altrimenti 0
==	a == b	1 se a è uguale a b, altrimenti 0
!=	a != b	1 se a non è uguale a b, altrimenti 0
&&	a && b	(AND logico) 1 se a e b sono veri, altrimenti 0: se a è falso, b non viene valutato
||	a || b	(OR logico) 1 se a è vero, (b non è valutato), 1 se b è vero, altrimenti 0

Funzioni di libreria

Richiedono tutte **#include <math.h>**

|x|	fabs(x)	
$\sqrt{x}$	sqrt(x)	
x^a	pow(x,a)	
e^x	exp(x)	
ln(x)	log(x)	
$\log_{10}(x)$	log10(x)	
sen(x)	sin(x)	
cos(x)	cos(x)	
tg(x)	tan(x)	
arcsen(x)	asin(x)	
arccos(x)	acos(x)	
arctg(x)	atan(x)	
senh(x)	sinh(x)	
cosh(x)	cosh(x)	
tgh(x)	tanh(x)	
	ceil(x)	restituisce il più piccolo intero >= x
	floor(x)	restituisce il più grande intero <= x

Funzioni create dall'utente

In C++ la funzione è il solo strumento per realizzare sottoprogrammi e procedure.
Una funzione può avere dei parametri di ingresso e, se non è definita di tipo **void**, restituire un risultato. Sintassi:

Tipo del risultato ***nome_funzione (****tipo1 inp1, tipo2 inp2, …, tipon inpn****)***

```
{
        ….
        return risultato;
}
```

Le funzioni possono essere definite prima della dichiarazione **main** oppure dopo: nel secondo caso (ed è la prassi) si deve inserire prima del main il ***prototipo*** *(prototype)* della funzione, ossia la sua intestazione.

In una funzione possono esserci più istruzioni ***return.*** Vedi l'esempio sottostante:

```
int match (int a, int b, int c) {
  if (a == b)
        return c;
   else
        return b;
}
```

Puntatori

Una **VARIABILE PUNTATORE (pointer)** è una variabile che contiene l'indirizzo di una cella di memoria.

L'operatore & (referencing) prefisso al nome di una variabile fornisce l'indirizzo di memoria della variabile.

L'operatore * (dereferencing) prefisso ad un indirizzo fornisce il contenuto in memoria dell'indirizzo.

Il **TIPO** di un puntatore è il tipo della variabile a cui punta

Esempi

int *ptr; definisce ptr variabile puntatore che conterrà indirizzi di memoria di variabili int

float *a; definisce a variabile puntatore che conterrà indirizzi di memoria di variabili float

```
int i = 7;
int *pippo;
pippo = &i;      pippo punta all'indirizzo l'indirizzo di memoria di i
*pippo = 10;   ora la variabile i vale 10
```

Vettori

Esempi di dichiarazione:

int vet[4] = {5, 7, 1, 24}; definisce e inizializza un vettore di interi

int vet[] = {5, 7, 1, 24}; definisce e inizializza lo stesso vettore

float a[100]; definisce un vettore di 100 float, non sempre lo inizializza

float a[100] = {0}; definisce un vettore di 100 float e li inizializza a 0

sintassi generale:

tipo nome_array [num.elementi] = {valori iniziali};

Gli elementi sono memorizzati in modo contiguo uno dopo l'altro, **gli indici partono da 0.**

N.B. C++ non controlla l'accesso agli array per cui un accesso al di fuori dell'array (indice errato) non verrà segnalato e andrà a "sporcare" altre aree di memoria oppure a recuperare dati da altre aree

Alternative nella definizione:

```
const int IMAX = 100;
float a[IMAX];  // dimensiono così tutti i vettori e, se devo, cambio solo il valore di IMAX
```

oppure

```
#define IMAX 100  // sostituisce tutti gli IMAX con 100
…
float a[IMAX];
```

Puntatori e vettori (array monodimensionali)

Il nome di un vettore è un puntatore. Se definisco

float a[10] ;

a, il nome dell'array, **è l'indirizzo dove l'array**, lungo 10, **inizia.**

Esempio

Definiti **float x**, e **float a[10]** è equivalente scrivere

x = a[0];

x = *a;

in entrambi i casi x conterrà il valore del primo elemento del vettore a; così come è equivalente scrivere

x = a[i];

x = *(a+i);

in tutti i casi x conterrà il valore dell' (i+1)-esimo elemento del vettore a.

Vettori e Funzioni

input: se un vettore è un parametro di input, nella **definizione** della funzione lo indicherò seguito da parentesi quadre vuote:

float effe (…, float a[] ,)

oppure col segno di PUNTATORE

float effe (…, float *a ,)

e metterò tra i parametri di input anche la sua lunghezza. Nella **chiamata** alla funzione lo passerò semplicemente con il suo nome.

output: in C++ il ***return*** può restituire solo uno scalare; se voglio che la funzione mi renda un vettore dovrò inserire tale vettore tra i parametri di input e fare in modo che la funzione modifichi i suoi valori. Una funzione di questo genere, se non deve rendere altro che il vettore, va dichiarata ***void.***

(vedi esempio)

Matrici (array bi-dimensionali)

Dichiarazione:

int a[3][2]={ {1,2},{3,4},{5,6}}; definisce e inizializza la matrice

int a[3][2]={1, 2, 3, 4, 5, 6}; equivale alla riga precedente

float a[10][10]; definisce una matrice 10x10, non sempre la inizializza a zero

float a[10][10] = {{0}}; definisce una matrice 10x10 e la inizializza a 0

in generale:

tipo nome_array [num.righe][num.colonne] = {valori iniziali};

Gli elementi sono memorizzati in modo contiguo una riga dopo l'altra, **gli indici sulle righe e sulle colonne partono da 0.**

Alternative nella definizione (come per i vettori):

const int IMAX = 100;

float a[IMAX][IMAX]; dimensiono così tutti gli array e, se devo, cambio solo la def di IMAX.

oppure

#define IMAX 100 sostituisce tutti gli IMAX con 100

…

float a[IMAX][IMAX];

Matrici e Puntatori

A[0] è il puntatore ossia l'indirizzo in memoria di A[0][0], il primo elemento di A.

A[i] è il puntatore ossia l'indirizzo in memoria di A[i][0], il primo elemento della (i+1)esima riga di A.

Osservazione. **A[i]** oppure **A+i** è il puntatore alla riga di indice **i.**

Le istruzioni **A[i][j]** **e** ***(A[i]+j)** si equivalgono.