Manuale C++

Il primo programma

Apriamo il programma Dev-C++

Scegliamo dal menu la voce File →Nuovo → File sorgente e digitiamo quanto segue

```
// Primo.cpp
// Il primo esempio in C++
#include <iostream.h>
using namespace std;
main()
{
  cout << "Il mio primo programma in C++";
}
```

Dal menu file scegliamo “salva come…” e gli diamo un nome, ad esempio Primo, senza estensione. Provvederà C++ ad assegnargli l’estensione .cpp

Proviamo a compilare (Menu Esegui → Compila) ed eseguire il programma (Menu Esegui → Esegui).Se non ci sono stati errori, apparirà una piccola console con la scritta:

```
Il mio primo programma in C++
```

In caso contrario, il compilatore indicherà le righe di codice in cui sono stati riscontrati dei problemi e quindi avviserà il programmatore che non è stato possibile creare l'eseguibile.

I commenti

Già questa piccola porzione di codice contiene numerose operazioni. Innanzitutto i **commenti**:

```
// Primo.cpp
// Il primo esempio in C++
```

Ogni programma ben realizzato dovrebbe includere dei commenti, per garantire una manutenzione agevole del codice: spesso capita di dover modificare codice altrui, oppure di dover riprendere in mano del vecchio codice e ricordarne il funzionamento. Per questo scrivendo un commento è utile non dare tutto troppo per scontato.

In **C++** una linea di codice commentata è preceduta dalla doppia barra `//`. Esiste, anche un secondo tipo di commento: il **commento a blocchi**, ereditato dal C. In questo caso, un commento inizia con i simboli `/*` e termina con i simboli `*/`. La differenza sostanziale è che il commento a blocchi permette di commentare più linee di codice senza dover ripetere ad ogni linea i simboli del commento stesso.

Per fare un esempio, le linee commentate del programma precedente, diverrebbero con l'uso del commento a blocchi:

```
/* Primo.cpp
   Il primo esempio in C++
 */
```

I due tipi di commenti sono equivalenti. Se c'è necessità di commentare parecchie linee di codice consecutive è conveniente ricorrere al commento a blocchi.

#include

L'istruzione:

#include <iostream.h>

rappresenta una delle funzionalità tipiche del C++, nota come istruzione per il preprocessore. Una istruzione per il preprocessore è, fondamentalmente, una istruzione di precompilazione.

#include in C contiene un richiamo ai file di intestazione (header con estensione .h) delle librerie che utilizziamo. Nello standard C++ si può omette l'estensione, per cui tra parentesi angolari si può scrivere sia "`iostream`" che "`iostream.h`"

main

Dopo l'istruzione **#include** si trova la dichiarazione della funzione **main()**:

```
main()
{
  // ... codice


}
```

Ogni programma C++ ha al suo interno una sequenza di operazioni e di funzioni. Tutti i programmi C++ devono contenere una funzione chiamata `main()`. Tale funzione rappresenta il punto di inizio dell'esecuzione del programma.

Le **parentesi graffe** `{` e `}` contengono blocchi di istruzioni. I blocchi di istruzioni possono consistere nella definizione del corpo di una funzione (come accade nel nostro caso per la funzione `main()`) oppure nel riunire più linee di codice che dipendono dalla stessa istruzione di controllo, come nel caso in cui diverse istruzioni vengono eseguite a seconda della validità o meno del test di un'istruzione.

Gli identificatori

Per poter apprezzare tutto ciò che il C++ ha da offrire sono necessari tempo e pratica. In questo paragrafo cominceremo l'esplorazione delle **strutture** del linguaggio. La grande stabilità del C++ deriva in gran parte dai tipi standard C e C++, dai modificatori e dalle operazioni che è possibile eseguire su di essi.

Gli identificatori
Gli identificatori sono i **nomi** utilizzati per rappresentare variabili, costanti, tipi, e funzioni o procedure del programma. Si crea un identificatore specificandolo nella dichiarazione di una variabile, di un tipo o di una funzione. Dopo la dichiarazione, l'identificatore potrà essere utilizzato nelle istruzioni del programma.
Un identificatore è formato da una sequenza di una o più lettere, cifre o caratteri e deve iniziare con una lettera o con un carattere di sottolineatura. Gli identificatori possono contenere un qualunque

numero di caratteri ma solo i primi 31 caratteri sono significativi per il compilatore.
Il linguaggio C++ distingue le lettere maiuscole dalle lettere minuscole. Cio' vuol dire che il compilatore considera le lettere maiuscole e minuscole come caratteri distinti. Ad esempio le variabili MAX e max saranno considerati come due identificatori distinti e, quindi, rappresenteranno due differenti celle di memoria.
Ecco alcuni esempi di identificatori:

i
MAX
max
first_name
_second_name

E' fortemente consigliato utilizzare degli identificatori che abbiano un nome utile al loro scopo. Ovvero, se un identificatore dovrà contenere l'indirizzo di una persona sarà certamente meglio utilizzare il nome indirizzo piuttosto che il nome casuale XHJOOQQO.
Sono entrati a far parte della programmazione comune gli identificatori i,j,k che vengono spesso utilizzati come *contatori* nelle iterazioni.

Le variabili
Una **variabile** non è nient'altro che un contenitore di informazione che viene utilizzata, poi, da qualche parte in un programma C++. Naturalmente, per poter conservare una determinata informazione, tale contenitore deve far uso della memoria del computer.

Come lo stesso nome suggerisce facilmente, una variabile, dopo essere stata dichiarata, può modificare il suo contenuto durante l'**esecuzione** di un programma. Il responsabile di tali eventuali modifiche altri non è che il programmatore il quale, tramite opportune istruzioni di assegnamento impone che il contenuto di una variabile possa contenere una determinata informazione. Possiamo tranquillamente dire che le variabili rappresentano l'essenza di qualunque programma di computer. Senza di esse, i computer diventerebbero totalmente inutili.

Ma cosa intendiamo quando parliamo di "**dichiarazione di una variabile**"? Diciamo che è necessario fornire al computer una informazione precisa sul tipo di variabile che vogliamo definire; per esempio, se stiamo pensando di voler assegnare ad una variabile un valore numerico dovremo fare in modo che il computer sappia che dovrà allocare una certa quantità di memoria che sia sufficiente a contenere tale informazione in modo corretto e senza possibilità di equivoci. A tale scopo, il C++ fornisce una serie di "tipi" standard che permettono al programmatore di definire le variabili in modo opportuno a seconda della tipologia dell'informazione che si vuole conservare. Vediamo dunque quali sono i tipi standard del C++.

I tipi standard del C++
Per la stragrande maggioranza dei casi, i **sette tipi base del C++** sono sufficienti per rappresentare le informazioni che possono essere manipolate in un programma. Vediamo, allora, quali sono tali tipi:
testo o **char**, intero o **int**, valori in virgola mobile o **float**, valori in virgola mobile doppi o **double**.

- Il testo (tipo char) può essere formato da serie di caratteri (b, F, !, ?, 6) e stringhe ("Quel ramo del lago di Como"). Il tipo char occupa normalmente 8 bit o, in altri termini, 1 byte per carattere. Va detto, anche che tale tipo può essere anche utilizzato per rappresentare valori numerici compresi nell'intervallo tra -128 e 127.
- I valori interi sono i valori numerici con i quali si è imparato a contare (1, 2, 43, -89, 4324, ecc.). Normalmente il tipo int occupa 16 bit (o 2 byte) e può quindi contenere valori

compresi tra -32768 e 32767. Sui sistemi operativi Windows a 32 bit (Windows 95, Windows 98 e Windows NT) il tipo int occupa invece 32 bit e quindi contiene valori compresi tra -2.147.483.648 e 2.147.483.647.

- I valori in virgola mobile sono utilizzati per rappresentare i numeri decimali. Questi numeri sono rappresentati da una parte intera ed una parte frazionale. I numeri in virgola mobile richiedono una maggiore precisione rispetto agli interi e sono quindi normalmente rappresentati da 32 bit. Il loro intervallo varia, perciò, tra ! 3,4 x 10-38 e 3,4 x 1038 (con 7 cifre significative).
- I valori double in virgola mobile sono valori molto estesi che normalmente occupano 64 bit (o 8 byte) e possono avere, quindi, un valore compreso fra ! 1,7 x 10-308 e 1,7 x 10308 (con 15 cifre significative). I valori long double sono ancora più precisi e normalmente occupano 80 bit (o 10 byte). Il loro valore è compreso fra ! 1,18 x 10-4932 e 1,18 x 104932 (con ben 19 cifre significative).

I caratteri

Qualunque lingua parlata e scritta, per costruire le proprie frasi fa uso di una serie di caratteri. Per esempio, questa guida è scritta utilizzando svariate combinazioni di lettere dell'alfabeto, cifre e segni di punteggiatura. Il tipo char, quindi, serve proprio ad identificare uno degli elementi del linguaggio. In particolare il tipo char potrà contenere:

Una delle 26 lettere minuscole dell'alfabeto:
a b c d e f g h i j k l m n o p q r s t u v w y z

Una delle 26 lettere maiuscole dell'alfabeto:
A B C D E F G H I J K L M N O P Q R S T U V X Y Z

Una delle 10 cifre (intese come caratteri e non come valori numerici)
0 1 2 3 4 5 6 7 8 9

O, infine, uno dei seguenti simboli:
+ - * / = , . _ : ; ? " ' ~ | ! # % $ & ( ) [ ] { } ^ Â®

Gli interi

Il C++ presenta una buona varietà di tipi numerici interi, poiché ai tipi fondamentali:

char intero che occupa un byte (8 bit)
int intero che occupa due byte (16 bit)

permette di aggiungere tre qualificatori unsigned, short, long, (che significano rispettivamente senza segno ,corto e lungo) che ne ampliano il significato. Vediamo come.
Abbiamo detto che un char è rappresentato da un byte. Un byte è, a sua volta, rappresentato da 8 bit. Ed il bit (che rappresenta l'acronimo di binary digit, ovvero cifra binaria) può contenere i valori 0 oppure 1. Vediamo alcuni esempi di byte:

10100010
00001100
00000000
11111111

Si definisce bit più significativo quello all'estrema sinistra mentre il bit meno significativo sarà quello all'estrema destra. Nella notazione normale, cioè senza l'uso di qualificatori, il bit più significativo viene utilizzato per il segno; in particolare se il primo bit vale 1 rappresenterà il segno

meno mentre se il primo bit vale 0 rappresenta il segno +.
Quindi, invece di 8 bit per rappresentare un valore, ne verranno usati solo 7. E con 7 bit, il valore maggiore che si riesce a raggiungere e' 128. Poiché anche lo zero deve essere considerato tra i valori possibili, allora otterremo proprio l'intervallo da 0 a 127 per i positivi e da -1 a -128 per i negativi.
Se viene, invece, utilizzato il qualificatore **unsigned**, tutti i bit che compongono il byte vengono utilizzati per contenere e comporre un valore, che in tal caso potrà essere soltanto un valore positivo. Il numero relativo all'ultimo byte rappresentato nell'esempio precendente varrà, in tal caso:

1 x 20 + 1 x 21 + 1 x 22 + 1 x 23 + 1 x 24 + 1 x 25 + 1 x 26 + 1 x 27 = 255

Questo ragionamento è valido per tutti gli altri tipi unsigned del C++..
Il qualificatore short, equivale fondamentalmente al tipo stesso. Dire int o dire short int è, in genere equivalente. Il qualificatore long è invece usato per incrementare il range di valori che solitamente un int può contenere (nel caso dei sistemi operativi Windows a 32 bit i tipi int e long int sono identici). Con tale qualificatore, vengono aggiunti due byte e si passa, dunque, dai 16 bit standard ai 32 bit.
Si noti come unsigned int, short int e long int possano essere dichiarati anche più sinteticamente come unsigned, short, long in quanto mancando l'operando, il qualificatore assume che sia sempre int.

Numeri in virgola mobile
Il C++ consente l'uso di tre tipi di numeri in virgola mobile: **float**, **double** e **long double**. Anche se lo standard ANSI C++ non definisce in modo preciso i valori e la quantità di memoria da allocare per ognuno di questi tipi, richiede però che ognuno di essi contenga come minimo i valori compresi fra 1E -37 e 1E+37.
La maggior parte dei compilatori C era già dotata dei tipi float e double. Il comitato ANSI C ha poi aggiunto un terzo tipo chiamato long double.
Vediamo un esempio che illustra la dichiarazione e l'uso delle variabili float.

```
/*
* Un semplice programma che mostra
* l'uso del tipo di dati float
* calcolando l'area di un cerchio
*/

#include <iostream>
using namespace std;
main( )
{
float raggio;
float pigreca = 3.14;  // definizione della variabile e assegnazione di un valore
float area;
cout << "Inserire il raggio: ";
cin >> raggio;
cout << endl;
area = raggio * raggio * pigreca;
cout << "L'area del cerchio e': " << area << endl;
system (“PAUSE”);
}
```

Dichiarazione di variabili
Come già detto, il C++ richiede tassativamente che ogni variabile prima di essere utilizzata dal programma venga preventivamente dichiarata. La dichiarazione avviene semplicemente indicando il tipo dell'identificatore in oggetto seguito da uno o più identificatori di variabile, separati da una virgola e seguiti dal punto e virgola al termine della dichiarazione stessa.
Per esempio per definire di tipo float le variabili a e b basta indicare:

```
float a, b;
```

Oppure per indicare di tipo int le variabili c e d:

```
int c, d;
```

E' importante sottolineare il fatto che una variabile può essere dichiarata soltanto una volta e di un solo tipo all'interno dell'intervallo di azione della variabile stessa.
E' possibile inizializzare una variabile, cioè assegnare alla stessa un valore contemporaneamente alla sua dichiarazione; ad esempio è consentita la dichiarazione:

```
float a, b = 4.6;
char ch = 't';
```

che corrisponde a scrivere:

```
float a, b;
char ch;

b = 4.6;
ch = 't';
```

cioè ad a non verrebbe per il momento assegnato alcun valore mentre b avrebbe inizialmente il valore 4.6 e ch avrebbe il carattere 't'.

Le costanti

In molti casi è utile assegnare a degli identificatori dei valori che restino costanti durante tutto il programma e che non possano essere cambiati nemmeno per errore. In C++ è possibile ottenere ciò in due modi con due risultati leggermente diversi:

- Con la direttiva al compilatore `#define`
- Con il modificatore `const`

La direttiva #define

Sintassi di #define

#define <*identificatore*> <*valore*>

Con la direttiva `#define` il compilatore in ogni punto dove incontra i simboli così definiti sostituisce ai simboli stessi i valori corrispondenti. Ad esempio, indicando:

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

```
#define MAXNUM 10
#define MINNUM 2

...

int x,y;
x = MAXNUM;
y = MINNUM;
```

automaticamente il compilatore definisce:

```
x = 10;
y = 2;
```

Volendo, in seguito modificare i valori in tutto il programma basterà modificare una volta per tutte i soli valori indicati con `#define`.

Si noti che `#define` non richiede che venga messo il punto e virgola finale e neppure l'operatore di assegnamento (=).

L'utilizzo della direttiva `#define` non soltanto diminuisce l'occupazione di memoria (non vi è infatti necessità di spazio per le costanti `MAXNUM` e `MINNUM`), ma anche rende più veloce il programma che durante l'esecuzione, quando le deve utilizzare, non deve ricercarne i valori.

C'è da dire che `#define` ha un **utilizzo più generale** che va oltre la definizione di costanti. Essa permette anche la definizione delle cosiddette macro. Vediamo un esempio per capire bene in cosa consistono le macro. Si supponga che in un programma si debbano invertire molte volte i contenuti di alcuni identificatori. Piuttosto che ripetere ogni volta la stessa sequenza di operazioni, viene utile costruirsi un'istruzione apposita come dal seguente programma:

```
#include <stdio.h>
#define inverti (x,y,temp) (temp)=(x); (x)=(y); (y)=(temp);

main()
{
  float a= 3.0, b = 5.2, z;
  int i = 4, j = 2, k;

  inverti(a, b, z); // adesso a = 5.2 e b = 3.0
  inverti(i,j,k); // adesso i = 2 e i = 4
}
```

Il modificatore const

Sintassi di const

const <*identificatore*> = <*valore*>;

Si faccia attenzione al fatto che qui viene usato sia l'operatore di assegnamento (=) che il punto e virgola finale. Se il tipo non viene indicato il compilatore assume per default che sia intero (int). Ecco un esempio di utilizzo del modificatore const:

const int MAXNUM = 10, MINNUM = 2;

const float PIGRECO = 3.1415926;

Si noti che, contrariamente a `#define`, con un solo utilizzo di const è possible dichiarare più identificatori separandoli con la virgola. Inoltre, il compilatore assegna un valore che non può essere modificato agli identificatori utilizzati (`MAXNUM` e `MINNUM` nell'esempio precedente), valore che però è archiviato in una zona di memoria e non sostituito in fase di compilazione, come accade per `#define`.

Operatori booleani

Il C++ mette a disposizione un numero superiore di operatori rispetto ad altri linguaggi, ma alcuni di essi risultano non facilmente interpretabili perché i loro simboli non hanno un immediato riferimento mnemonico alla funzione svolta. Vediamo i più importanti operatori del C++.

Valori booleani: true e false

Prima di parlare degli operatori booleani, è importante sapere cosa è un valore "booleano". Un booleano può assumere solo due valori: `true` o `false`. Nessun altro valore è permesso.

Gli identificatori booleani e le operazioni su di essi sono molto usati. Spesso, in un programma, si rende necessario sapere se una certa condizione è vera (`true`) oppure falsa (`false`). Nel primo caso, il corso del programma prenderà una determinata direzione che, invece, non sarebbe intrapresa altrimenti.

Un esempio grafico particolarmente attinente alle operazioni con boolean è rappresentato dalle check box. Se una check box è selezionata si vorrà intraprendere una determinata azione. In caso contrario non si vorrà fare nulla.

La maggior parte dei linguaggi di programmazione contemplano il tipo booleano. La maggior parte dei compilatori C++ riconosce il tipo boolean con la parola chiave `bool`. Altri, invece, accettano la scrittura `boolean`. Diamo per buono che il compilatore riconosca il tipo `bool`. In tal caso, una dichiarazinone di una variabile booleana sarà la seguente:

bool flag;

I principali operatori di tipo booleano, ovvero gli operatori che consentono di eseguire operazioni su elementi di tipo `bool` sono 3:

Funzione	**Simbolo**	**Significato**
AND	`&&`	Congiunzione logica
OR	`||`	Disgiunzione logica
NOT	`!`	Negazione logica

Ciascun operatore prende in input uno o due booleani e restituisce in output un altro booleano.

AND, prende in input due operandi e produce in output un booleano, attenendosi al seguente comportamento: Se entrambi gli operatori sono true allora l'output è true; in tutti gli altri casi l'output è uguale a false.

OR, prende in input due operandi e produce in output un booleano, attenendosi al seguente comportamente: Se almeno undo degli operandi è uguale a true, l'output è true; altrimenti, se nessuno dei due operandi è uguale a true l'output sarà false.

NOT, prende in input un solo operando e produce in output un booleano, attenendosi al seguente comportamento: Se l'operando di input è true allora l'output sarà false. Se, invece l'operando di input è false, allora l'output sarà uguale a true. In altri termini, l'operatore di NOT prende un input e ne restituisce l'esatto contrario.

Vediamo ora alcuni esempi sull'utilizzo dei tre operatori definiti.

```
// Supponiamo che Gianni sia stanco
bool gianniStanco = true;

// Supponiamo pure che Gianni non sia costretto ad alzarsi presto
bool gianniDeveAlzarsiPresto = false;

// Andrà a letto ora Gianni?
Bool gianniSiCoricaOra = gianniStanco && gianniDeveAlzarsiPresto;

// Il risultato è false
```

L'esempio precedente è abbastanza semplice da comprendere. La prima variabile `bool` (`gianniStanco`) viene inizializzata a `true`, il che equivale a dire che Gianni è stanco. La seconda variabile bool (`gianniDeveAlzarsiPresto`) è inizializzata invece a `false`, il che vuol dire che Gianni non ha la necessità di alzarsi presto la mattina. La terza variabile booleana (`gianniSiCoricaOra`) è un risultato dell'operazione di AND tra le due precedenti. Ovvero: Gianni deve andare a letto adesso soltanto se è stanco e deve alzarsi la mattina presto. Quindi, l'operatore AND è il più adatto in tale circostanza.

Se, invece, Gianni decidesse di andare a letto se anche soltanto una delle due precondizioni fosse vera allora l'operatore da utilizzare sarebbe l'operatore OR. Avremo in tal caso:

```
bool gianniSiCoricaOra = gianniStanco || gianniDeveAlzarsiPresto;
// Il risultato è true
```

Se, ancora, si verifica la condizione:

```
gianniStanco = true

bool gianniInForma = !gianniStanco
// gianniInForma sarà uguale a false
```

Ovvero la variabile booleana `gianniInForma` sarà vera se non è vera quella che identifica Gianni stanco. Questo è un banale esempio dell'operatore NOT.

L'utilizzo degli operatori booleani è perfettamente lecito anche su variabili che non siano `bool`. **In C++ il valore "0" equivale a `false`** e qualunque valore diverso da zero equivale a true. Ad esempio:

```
int ore = 4;
int minuti = 21;
int secondi = 0;

bool timeIstrue = ore && minuti && secondi;
```

Poiché il risultato deriva dall'esame dei tre operandi e c'è un valore (secondi) che è uguale a zero (ovvero equivale a false) il risultato dell'espressione è false.

Operatori aritmetici

Il linguaggio C++ è dotato di tutti i comuni operatori aritmetici di somma (`+`), sottrazione (`-`), moltiplicazione (`*`), divisione (`/`) e modulo (`%`). I primi quattro operatori non richiedono alcuna spiegazione data la loro familiarità nell'uso comune.

L'operatore modulo, invece, è semplicemente un modo per restituire il resto di una divisione intera. Ad esempio:

```
int a = 3, b = 8, c = 0, d;

d = b % a; // restituisce 2
d = a % b; // restituisce 3

d = b % c; // restituisce un messaggio di errore (divisione per zero)
```

Operatore di assegnamento

L'operatore di assegnamento in C++, altro non fa che assegnare ad una variabile un determinato valore. È importante dire che un'espressione contenente un operatore di assegnamento può quindi essere utilizzata anche all'interno di altre espressioni, come ad esempio:

```
x = 5 * (y = 3);
```

In questo esempio, alla variabile `y` viene assegnato il valore 3. Tale valore verrà moltiplicato per 5 e quindi, in definitiva, alla variabile `x` sarà assegnato il numero 15.

È, però, caldamente sconsigliato l'utilizzo di tale pratica in quanto potrebbe rendere poco leggibili le espressioni. Vi sono, tuttavia, un paio di casi in cui questa possibilità viene normalmente sfruttata. Innanzitutto, si può assegnare a più variabili lo stesso valore, come in:

```
x = y = z = 4;
```

Operatori di relazione

Un operatore di relazione è un operatore che **verifica una condizione** come: "è minore di" oppure "è maggiore di" oppure ancora "è uguale a".

Un utilizzo intuitivo di tali operatori è quello che viene fatto per comparare due numeri. Un operatore di relazione può essere determinante per la scelta di una determinata strada piuttosto che

di un'altra. Ad esempio, se una certa variabile supera un valore di soglia potrebbe essere necessario chiamare una funzione per controllare altri parametri, e così via.

Ma vediamo nella seguente tabella, l'elenco completo degli operatori di relazione:

Nome	Simbolo	Esempio	Risultato
Minore	`<`	`boolris = (3 < 8)`	`true`
Maggiore	`>`	`boolris = (3 > 8)`	`false`
Uguale	`==`	`boolris = (5 == 5)`	`true`
MinoreUguale	`<=`	`boolris = (3 <= 5)`	`true`
MaggioreUguale	`>=`	`boolris = (5 >= 9)`	`false`
Diverso	`!=`	`boolris = (4 != 9)`	`true`

Precedenza tra operatori

La **precedenza tra operatori** indica l'ordine con cui gli operatori vengono valutati dal **compilatore**. Un operatore con precedenza maggiore verrà valutato per prima rispetto ad un operatore con precedenza minore, anche se quest'ultimo figura prima dell'operatore con precedenza maggiore. Ecco un esempio:

```
int risultato = 4 + 5 * 7 + 3;
```

Il risultato in questo caso dipende proprio dalla precedenza tra operatori. In C++, l'operatore di **moltiplicazione** (`*`) ha precedenza rispetto all'operatore **addizione** (`+`). Quindi la moltiplicazione `5*7` avverrà prima di tutte le altre addizioni. Ecco la sequenza della risoluzione dell'espressione precendente:

```
int risultato = 4 + 5 * 7 + 3;
risultato = 4 + 35 + 3 = 42
```

Perché le operazioni siano effettuate con ordine diverso, sarà sufficiente introdurre delle **parentesi tonde**. Ad esempio se vogliamo moltiplicare la somma `4+5` con la somma di `7+3`, basterà scrivere:

```
int risultato = (4 + 5) * (7 + 3);
```

La variabile `risultato`, in questo caso, varrà 90.

Facciamo ora un elenco di tutti gli operatori ordinati per livello di precedenza. Ad ogni riga della tabella è assegnato un livello di precendenza. L'operatore a maggior priorità avrà il valore 1 e a seguire quelli a priorità sempre più bassa

Livello di precedenza	Operatore	Nome
1	`!`	Not, negazione
2	`*`	Moltiplicazione
2	`/`	Divisione
2	`%`	Modulo
3	`+`	Addizione
3	`-`	Sottrazione
4	`<`	Minore
4	`<=`	Minore uguale

4	>	Maggiore
4	>=	Maggiore uguale
5	==	Uguale (confronto)
5	!=	Diverso
6	&&	AND
7	||	OR
8	=	Assegnamento

Le **parentesi** risolvono praticamente il problema di conoscere la precedenza tra gli operatori. Per evitare errori ed aumentare la leggibilità del codice può essere consigliabile l'uso delle parentesi in ogni situazione in cui vi sia la presenza contemporanea di più operatori e si abbiano dubbi sulle precedenze.

Le istruzioni if e else

Le istruzioni condizionali ci consentono di far eseguire in modo selettivo una singola riga di codice o una serie di righe di codice (che viene detto **blocco di istruzioni**).

Il C++ fornisce quattro istruzioni condizionali:

- **if**
- **if-else**
- **?**
- **switch**

Prima di affrontare tali istruzioni una per una, è bene dire che, quando all'istruzione condizionale è associata una sola riga di codice, non è necessario racchiudere l'istruzione da eseguire fra **parentesi graffe**. Se invece all'istruzione condizionale è associata una serie di righe di codice, il blocco di codice eseguibile dovrà essere racchiuso fra parentesi graffe.

Per evitare il rischio di errori (specie quando si è neofiti) e per una buona lettura del codice è consigliabile l'uso delle parentesi graffe anche quando l'istruzione da eseguire è soltanto una.

Le istruzioni if e else

Le istruzioni in oggetto sono molto intuitive se si considera che le parole inglesi **if** ed **else** corrispondono rispettivamente alle italiane *se* e *altrimenti*. La sintassi di `if` è:

```
if (<condizione>)
{
  (<istruzioni da svolgere se la condizione è vera>);
}
```

mentre se `if` è accompagnata da altre istruzioni alternative la sintassi è:

```
if(<condizione>)
{
  (<istruzioni da svolgere se la condizione è vera>);
}
else
{
  (<istruzioni da svolgere se la condizione è falsa>);
}
```

oppure se è necessario condizionare anche le istruzioni rette da else:

```
if(<condizione 1>)
{
  <istruzioni da svolgere solo se la condizione 1 è vera>);
}
else if(<condizione 2>)
{
  (<istruzioni da svolgere solo se la condizione 1 è falsa e la condizione 2 è
vera>);
}
```

All'interno delle parentesi le istruzioni vanno completate sempre con il punto e virgola, mentre se non utilizziamo blocchi, ma semplici istruzioni facciamo attenzione a non dimenticare il punto e virgola prima dell'istruzione `else`. Ad esempio:

```
if(i>=0)
  <istruzione 1>; // singola istruzione
else
{                 // blocco di istruzioni
  <istruzione 2>;
  <istruzione 3>;
}
```

L'istruzione 1 verrà eseguita solo se `i >= 0` mentre in caso contrario viene eseguito il blocco con le istruzioni 2 e 3.

Istruzioni if-else nidificate

Quando si utilizzano più **costrutti if-else** nidificati, occorre sempre essere sicuri dell'azione **else** che verrà associata ad un determinato **if**. Vediamo cosa accade nell'esempio seguente:

```
if(temperatura < 20)
if(temperatura < 10) cout << "Metti il cappotto!n";
else cout << " Basta mettere una felpa";
```

Come si vede, il codice non è allineato in maniera ben leggibile ed è facile che il programmatore poco esperto possa confonderne il flusso di esecuzione, non capendo bene a quale istruzione if fa riferimento l'istruzione **else** finale.

La regola base, in questi casi, dice che un'istruzione **else** è sempre riferita all'ultima istruzione **if** che compare nel codice. Ma, è bene non confondersi troppo le idee utilizzando tale regola, poichè ci sono casi in cui appaiono tante istruzioni if-else annidate con la conseguenza che la lettura del codice sarebbe davvero difficile per chiunque.

Per rendere il codice leggibile e per confondersi il meno possibile, è buona norma fare uso sempre delle parentesi graffe e dell'**indentazione** del codice.

Indentare il codice significa inserire degli spazi prima della scrittura di una riga in modo da facilitarne la comprensione e la leggibilità. L'esempio precedente, indentato e corretto con l'uso delle parentesi diventa:

```
if(temperatura < 20)
{
  if(temperatura < 10)
  {
    cout << "Metti il cappotto!";
  }
  else
  {
    cout << " Basta mettere una felpa";
  }
}
```

Oppure, per chi ama le forme abbreviate (ma più insidiose):

```
if(temperatura < 20)
  if(temperatura < 10) cout << "Metti il cappotto!";
  else cout << "Basta mettere una felpa";
```

Ora è decisamente più comprensibile. Adesso, infatti, diventa chiaro che l'istruzione `else` è riferita alla istruzione `if(temperatura < 10)`.

le istruzioni switch

Le istruzioni switch permettono di evitare noiose sequenze di if. Esse sono particolarmente utili quando in un programma si deve dare all'utente la possibilità di scegliere tra più opzioni. La sintassi di tali istruzioni è la seguente:

switch(<espressione intera>)
{
case (<valore costante 1>):
...(<sequenza di istruzioni>)
break;
case (<valore costante 2>)
...(<sequenza di istruzioni>)
break;
....
....
default:
// è opzionale
...(<sequenza di istruzioni>)
}

Il costrutto precedente esegue le seguenti operazioni:

- Valuta il valore dell'espressione intera passata come parametro all'istruzione switch.
- Rimanda lo svolgimento del programma al blocco in cui il parametro dell'istruzione case ha lo stesso valore di quello dell'istruzione switch.
- Se il blocco individuato termina con un'istruzione break allora il programma esce dallo switch.
- Altrimenti, vengono eseguiti anche i blocchi successivi finchè un'istruzione break non viene individuata oppure non si raggiunge l'ultimo blocco dello switch.
- Se nessun blocco corrisponde ad un valore uguale a quello dell'istruzione switch allora viene eseguito il blocco default, se presente.

Vediamo un paio di esempi per comprendere meglio quanto detto:

bool freddo, molto_freddo, caldo, molto_caldo;
switch (temperatura)
{
case(10):
freddo = true;
molto_freddo = false;
caldo = false;
molto_caldo = false

```
break;
case(2):
freddo = true;
molto_freddo = true;
caldo = false;
molto_caldo = false;
break;
case(28):
freddo = false;
molto_freddo = false;
caldo = true;
molto_caldo = false;
break;
case(40):
freddo = false,
molto_freddo = false;
caldo = true;
molto_caldo = true;
break;
default:
freddo = false;
molto_freddo = false;
caldo = false;
molto_caldo = false;
```

Nell'esempio precedente, viene valutato il valore costante temperatura passato come parametro a switch. A secondo del valore che tale costante assume viene poi eseguito il relativo blocco di istruzioni. Se, invece, nessuno dei blocchi ha un valore uguale a quello passato a switch, verrà eseguito il blocco default.
Si noti, che nel caso seguente:

```
.....

case(28):
freddo = false;
molto_freddo = false;
caldo = true;
molto_caldo = false;
case(40):
freddo = false,
molto_freddo = false;
caldo = true;
molto_caldo = true;
break;
.....
```

se il valore della temperatura fosse di 28 gradi, verrebbe giustamente eseguito il blocco corrispondente, ma poichè manca l'istruzione break alla fine di tale blocco, il programma avrebbe continuato ad eseguire anche le istruzioni del blocco successivo (quello con la temperatura uguale a 40 gradi) e quindi il valore finale delle variabili sarebbe stato:

```
freddo = false;
molto_freddo = false;
caldo = true;
molto_caldo = true;
```

il ciclo for

Il linguaggio C++ è dotato di tutte le istruzioni di controllo ripetitive presenti negli altri linguaggi: cicli **for, while e do-while**. La differenza fondamentale fra un ciclo for e un ciclo while o do-while consiste nella definizione del numero delle ripetizioni. Normalmente, i cicli for vengono utilizzati quando il numero delle operazioni richieste è ben definito mentre i cicli while e do-while vengono utilizzati quando invece il numero delle ripetizioni non è noto a priori.

Il ciclo for
Il ciclo for ha la seguente sintassi:

```
for(valore_iniziale, condizione_di_test, incremento)
{
(<istruzioni da eseguire all'interno del ciclo>)
}
```

La variabile *valore_iniziale* setta la il valore iniziale del contatore del ciclo. La variabile *condizione_di_test* rappresenta la condizione da testare ad ogni passo del ciclo per vedere se è necessario continuare oppure uscire dal ciclo stesso.

La variabile *incremento* descrive la modifica che viene apportata al contatore del ciclo ad ogni esecuzione.

Ecco un esempio:

```
// Il codice seguente esegue la somma
// dei primi 10 numeri

// Questa variabile contiene la somma totale

int totale=0;

// Il ciclo seguente aggiunge i numeri
//da 1 a 10 alla variabile totale

for (int i=1; i < 11; i++)
{
totale = totale + i;
}
```

Quindi, nella porzione di codice precedente accade che:

il nostro valore_iniziale è un intero che vale 0.
La condizione_di_test verifica che i sia minore di 11 per continuare il ciclo L'incremento aggiunge 1 ad ogni passo del ciclo al valore_iniziale (l'istruzione i++ serve per incrementare di un'unità un intero).

Subito dopo l'esecuzione iniziale del primo loop la variabile intera i viene settata a 1, viene eseguita l'istruzione totale = totale + i e quindi il valore della variabile totale diviene uguale ad 1. A questo punto viene eseguito l'incremento e quindi i diventa uguale a 2. Viene eseguita la condizione di test e quindi ancora incrementata la variabile i al valore 2. totale ora varrà allora 1 + 2 = 3.
Si prosegue così finchè la variabile i resta minore di 11. In definitiva avremo eseguito l'operazione: 1+2+3+4+5+6+7+8+9+10 = 55.

il ciclo while

L'istruzione while segue la seguente sintassi:

```
while(condizione)
{
// Istruzioni da eseguire
}
```

dove *condizione* rappresenta un controllo booleano che viene effettuato ogni volta al termine del blocco di istruzioni contenuto tra le parentesi graffe.
Se la condizione restituisce true allora il ciclo sarà eseguito ancora mentre se la condizione ritorna un valore uguale a false il ciclo sarà terminato.
Vediamo un esempio. Supponiamo di voler scrivere un programma che stampi sullo schermo tutti i numeri pari tra 11 e 23:

```
// File da includere per operazioni
// di input/output cout
#include <iostream.h>

main()
{
// Definiamo una variabile che conterrà il valore corrente
int numero_corrente = 12;

// ciclo while che stampa sullo schermo tutti i numeri pari
// tra 11 e 23
while (numero_corrente < 23)
{
cout << numero_corrente << endl;
numero_corrente = numero_corrente + 2;
}

cout << "Fine del Programma!" << endl;

system(“PAUSE”);
}
```

Il funzionamento del programma precedente è molto semplice: viene definita una variabile numero_corrente che useremo per conservare il valore che di volta in volta il ciclo while modificherà. Inizialmente tale variabile viene inizializzata a 12 (che è il primo valore pari dell'intervallo). A questo punto si entra nel ciclo while e viene testata la condizione che si chiede se numero_corrente è minore di 23. Ovviamente la condizione viene soddisfatta e così viene eseguito

il blocco all'interno del ciclo while. La prima istruzione del blocco stampa il valore della variabile numero_corrente (che è attualmente 12) mentre la seconda istruzione incrementa il valore della stessa variabile di 2 unità. Adesso numero_corrente vale 14.
Si ricomincia di nuovo: si testa la condizione, questa è ancora soddisfatta, si stampa il valore della variabile (adesso è 14) e si incrementa numero_corrente di 2 unità (ora vale 16). E così via fino a quando la condizione non è più verificata, ovvero quando accadrà che numero corrente varrà 24.

il ciclo do-while

Il ciclo do-while ha la seguente sintassi:

```
do
{
(<istruzioni da eseguire all'interno del ciclo>)
}while (condizione)
```

La differenza fondamentale tra il ciclo **do-while** e i cicli **for** e **while**è che il **do-while** esegue l'esecuzione del ciclo almeno per una volta. Infatti, come si vede dalla sintassi, il controllo della condizione viene eseguito al termine di ogni loop. Se volessimo scrivere lo stesso esempio della visualizzazione dei numeri pari compresi tra 11 e 23 otterremmo:

```
// File da includere per operazioni di
//input/output cout
#include <iostream.h>

int main()
{
// Definiamo una variabile che conterrà il valore corrente
int numero_corrente = 12;

// ciclo do-while che stampa sullo
//schermo tutti i numeri pari
// tra 11 e 23
do
{
cout << numero_corrente << endl;
numero_corrente = numero_corrente + 2;
} while (numero_corrente < 23);

cerr << "Fine del Programma!" << endl;
}
```

Il funzionamento del programma precedente è molto simile a quello visto per il ciclo while con la differenza, come detto, che la condizione viene verificata al termine dell'esecuzione del blocco di istruzioni. Se, per esempio, il valore iniziale della variabile numero_corrente fosse stato 26, il programma avrebbe comunque stampato sul video il valore 26 e poi si sarebbe fermato. In tal caso, avremmo ottenuto un risultato diverso da quello desiderato in quanto il numero 26 è certamente maggiore del 23 che rappresenta il limite dell'intervallo da noi definito. Il ciclo while, invece, non avrebbe stampato alcun numero.
Tale osservazione va tenuta presente per capire che non sempre le tre istruzioni di ciclo sono intercambiabili.

l'istruzione break

Abbiamo già visto, esaminando l'istruzione switch, un utilizzo pratico dell'istruzione break. Diciamo, in generale, che l'istruzione break può essere utilizzata per uscire da un ciclo prima che la condizione di test divenga false.
Quando si esce da un ciclo con un'istruzione break, l'esecuzione del programma continua con l'istruzione che segue il ciclo stesso. Vediamo un semplice esempio:

```
// Esempio di utilizzo dell'istruzione break
main()
{
int i = 1, somma = 0;

while( i < 10)
{
  somma = somma + i;
  if(somma > 20)
    {
      break;
    }
    i++;
}
}
```

Quello che avviene è che il ciclo while ad ogni passo ricalcola il valore di somma ed incrementa i. Così, per esempio, al primo passo avremo che somma = 0 + 1 e i viene incrementata a 2.
Quindi, al secondo passo, somma vale 1 + 2 = 3 ed i sarà incrementata a 3.
Se non fosse presente l'istruzione break, l'ultimo passo del ciclo while sarebbe: somma = 36 + 9 = 45, con i = 9 e successivamente incrementata ad 10.
Ma con la presenza dell'istruzione break, quando somma raggiunge un valore maggiore di 20 avviene l'uscita immediata dal ciclo while. Nel nostro esempio, quindi somma varrà 21.

l'istruzione continue

Vi è una piccola ma significativa differenza tra l'istruzione break e l'istruzione continue. Come si è visto nell'ultimo esempio, break provoca l'uscita immediata dal ciclo. L'istruzione continue invece, fa in modo che le istruzioni che la seguono vengano ignorate ma non impedisce l'incremento della variabile di controllo o il controllo della condizione di test del ciclo. In altri termini, se la variabile di controllo soddisfa ancora la condizione di test, si continuerà con l'esecuzione del ciclo.
Modifichiamo leggermente l'esempio visto prima cambiando l'istruzione break con l'istruzione continue e inserendo l'incremento della variabile i come prima istruzione del blocco while.
Otterremo:

```
// Esempio di utilizzo dell'istruzione continue
main()
{
int i = 0, somma = 0;

while ( i < 10)
{
i++;
```

```
if (i == 5)
{
continue;
}
somma = somma + i;

}
return (0);
}
```

Cosa accadrà adesso? Semplicemente, quando la variabile i varrà 5, l'istruzione somma = somma + i verrà saltata ed il ciclo riprenderà dal passo successivo. Dunque, quello che si otterrà e:

somma = 0 + 1 + 2 + 3 + 4 + 6 + 7 + 8 + 9 = 50

l'istruzione exit

Può accadere, in alcuni casi, che il programma debba terminare molto prima che vengano esaminate o eseguite le sue istruzioni. In questi casi, il C++ mette a disposizione l'istruzione **exit()**. Tale istruzione (che in realtà corrisponde ad una funzione), prende un argomento intero come parametro ed esce immediatamente dall'esecuzione del programma. In generale, l'argomento 0 indica una terminazione del programma regolare.