Linguaggio C++

Introduzione

Come e' facile intuire, il linguaggio C++ è un'**estensione** del linguaggio C. In particolare, il C++ conserva tutti i **punti di forza** del C, come la potenza e la flessibilità di gestione dell'interfaccia hardware e software, la possibilità di programmare a basso livello e l'efficienza, l'economia e le espressioni, tipiche del C. Ma, in più, il C++ introduce il dinamico mondo della programmazione orientata agli oggetti che rende tale linguaggio una piattaforma ideale per l'astrazione dei problemi di alto livello.
Il C++ fonde, quindi, i costrutti tipici dei linguaggi **procedurali** standard, familiari per molti programmatori, con il modello di programmazione orientata agli oggetti, che può essere pienamente sfruttato per produrre soluzioni completamente orientate agli oggetti di un determinato problema. In pratica, una applicazione C++ riflette questa dualità incorporando sia il modello di programmazinoe procedurale che il modello di programmazione orientato agli oggetti.
Questa guida al C++ si rivolge a chi si volge allo straordinario mondo della programmazione per **la prima volta** o, anche, a chi ha desiderio di rispolverare qualche concetto non troppo chiaro. L'obiettivo, in ogni caso, non è certamente quello di fornire una descrizione approfondita di tutte le potenzialità della programmazione ad oggetti ma, più semplicemente, quello di fornire al programmatore una **panoramica** del C++ che lo renda presto in grado di scrivere applicazioni funzionanti

La storia del C++

Non dovrebbe sorprendere più di tanto il fatto che il C++ abbia un'origine simile al C. Lo sviluppo del linguaggio C++ all'inizio degli anni Ottanta è dovuto a **Bjarne Stroustrup** dei laboratori Bell (Stroustrup ammette che il nome di questo nuovo linguaggio è dovuto a Rick Mascitti). Originariamente il C++ era stato sviluppato per risolvere alcune simulazioni molto rigorose e guidate da eventi; per questo tipo di applicazione la scelta della massima efficienza precludeva l'impiego di altri linguaggi.
Il linguaggio C++ venne utilizzato all'esterno del gruppo di sviluppo di Stroustrup nel **1983** e, fino all'estate del **1987**, il linguaggio fu soggetto a una naturale evoluzione.
Uno degli **scopi principali** del C++ era quello di mantenere piena compatibilità con il C. L'idea era quella di conservare l'integrità di molte librerie C e l'uso degli strumenti sviluppati per il C. Grazie all'alto livello di successo nel raggiungimento di questo obiettivo, molti programmatori trovano la transizione al linguaggio C++ molto più semplice rispetto alla transizione da altri linguaggi (ad esempio il **FORtrAN**) al C.
Il C++ consente lo sviluppo di software su larga scala. Grazie a un maggiore rigore sul controllo dei tipi, molti degli effetti collaterali tipici del C, divengono impossibili in C++.
Il miglioramento più significativo del linguaggio C++ è il supporto della programmazione orientata agli oggetti (**Object Oriented Programming**: OOP), Per sfruttare tutti i benefici introdotti dal C++ occorre cambiare approccio nella soluzione dei problemi. Ad esempio, occorre identificare gli oggetti e le operazioni ad essi associate e costruire tutte le classi e le sottoclassi necessarie.

La programmazione ad Oggetti

La **programmazione ad oggetti** è ormai affermata e trova sempre maggiore successo nell'ambito della progettazione e dello sviluppo del software. La programmazione ad oggetti rappresenta un modo di pensare in modo astratto ad un problema, utilizzando concetti del mondo reale piuttosto che concetti legati al computer. In tal modo si riescono a comprendere meglio i requisiti dell'utente, si ottengono dei progetti più chiari e ne risultano sistemi software in cui la manutenzione è decisamente più facile. Gli oggetti costituiscono il costrutto fondamentale di tale filosofia. Un **oggetto** è una singola entità che combina sia strutture dati che comportamenti: i dati sono visti come variabili e le procedure per accedere ai dati sono viste come metodi (o funzioni).
Gli oggetti, in sostanza, possono essere visti come dei **mattoni** che possono essere assemblati e che hanno un alto potenziale di riutilizzo. Infatti uno dei benefici principali ed uno dei motivi per cui è stata sviluppata la programmazione ad oggetti è proprio quello di favorire il riutilizzo del software sviluppato.
In seguito, in questa guida, approfondiremo il concetto di programmazione orientata agli oggetti.

Miglioramenti rispetto al C

Le sezioni che seguono indicano i miglioramenti di lieve entità (che non hanno nulla a che fare con gli oggetti) rispetto al linguaggio C.
Commenti
Il C++ introduce il delimitatore di "commento fino a fine riga" //. Viene però conservato l'uso dei delimitatori /* e */.
Nome delle enumerazioni
Il nome delle enumerazioni è un nome di tipo. Questa possibilità semplifica la notazione poiché non richiede l'uso del qualificatore enum davanti al nome di un tipo enumerativo.
Nomi delle classi
Il nome di una classe è un nome di tipo. Questo costrutto non esiste in C. In C++ non è necessario usare il qualificatore class davanti ai nomi rispettivamente delle classi.
Dichiarazioni di blocchi
Il C++ consente l'uso di dichiarazioni all'interno dei blocchi e dopo le istruzioni di codice. Questa possibilità consente di dichiarare un identificatore più vicino al punto in cui viene utilizzato per la prima volta. E' perfino possibile dichiarare l'indice di un ciclo all'interno del ciclo stesso.
Operatore di qualifica della visibilità
L'operatore di qualifica della visibilità :: (detto anche operatore di scope) è un nuovo operatore utilizzato per risolvere i conflitti di nome. Ad esempio, se una funzione ha una dichiarazione locale della variabile *vettore* ed esiste una variabile globale *vettore*, il qualificatore **::vettore** consente di accedere alla variabile globale anche dall'interno del campo di visibilità della variabile locale. Non è possibile l'operazione inversa.
Lo specificatore const
Lo specificatore *const* consente di bloccare il valore di un'entità all'interno del suo campo di visibilità. E' anche possibile bloccare i dati indirizzati da una variabile puntatore, l'indirizzo di un puntatore e perfino l'indirizzo del puntatore e dei dati puntati.
Overloading delle funzioni
In C++, più funzioni possono utilizzare lo stesso nome; la distinzione si basa sul numero e sul tipo dei parametri.

Valori standard per i parametri delle funzioni
E' possibile richiamare una funzione utilizzando un numero di parametri inferiori rispetto a quelli dichiarati. I parametri mancanti assumeranno valori di standard.
Gli operatori new e delete
Gli operatori new e delete lasciano al programmatore il controllo della allocazione e della deallocazione della memoria dello heap.
gli elementi principali di un programma C
Come abbiamo detto in precedenza, Il C++ è un linguaggio derivato dal C, che conserva con questo una completa compatibilità. Sarà, dunque, necessario per poter muovere i primi passi verso la programmazione in C++ procedere alla conoscenza di quei concetti e di quegli elementi del linguaggio C che sono alla base della programmazione in C++.
Alla fine di questo capitolo si dovrebbe essere in grado di scrivere un programma breve ma funzionante.

Le cinque componenti fondamentali di un programma C/C++

- I programmi devono contenere informazioni da una sorgente di unput.
- I programmi devono decidere il modo in cui questo input deve essere manipolato e memorizzato.
- I programmi devono usare una serie di istruzioni di manipolazione dell'input. Queste istruzioni possono essere suddivise in quattro categorie principali: singole istruzioni, istruzioni condizionali, cicli e funzioni.
- I programmi devono produrre un qualche tipo di rapporto sulla manipolazione dei dati.
- Un'applicazione ben realizzata incorpora tutti gli elementi fondamentali appena elencati con un progetto modulare, con codice autodocumentante (ad esempio nomi di variabili di senso compiuto) e un rigoroso schema di indentazione.

Come creare e compilare un progetto in C++

Il processo di creazione e compilazione di un progetto in C++ dipende strettamente dal tipo di computer e di sistema operativo che si utilizza. Poichè la maggior parte dei neofiti utilizza, probabilmente, l'ambiente Windows a 32 bit (Windows 95, Windows 98, Windows NT, etc.) e, di conseguenza, il Microsoft Visual Studio, vedremo di definire con un semplice esempio pratico quali sono i passi da seguire per la creazione e la successiva compilazione di un progetto C++.
Per creare un nuovo progetto, si procede così:

- Lanciare Microsoft Visual Studio
- Dal Menù File selezionare la voce **New**...
- Apparirà la Finestra "New". Selezionare nella lista a sinistra, sotto la scheda **Projects** la voce **Win32 Console Application**. Scegliere a destra il nome del progetto e la locazione (la directory del disco) in cui salvare il progetto stesso. Quindi premere OK.

A questo punto il progetto è stato creato.
Occorre ora aggiungere al progetto uno o più files, che conterranno il codice C++, uno dei quali (e soltanto uno) deve contenere la funzione **main**. Si procede così:

- Dal menù **File**, selezionare **New...**
- Apparirà di nuovo la Finestra "New".
- Selezionare la scheda **Files**
- Selezionare la casella **Add to project** e premere Ok.
- Selezionare nella lista a sinistra la voce **C++ Source File** se si vuole creare un file con estensione .cpp oppure selezionare la voce **C/C++ Header File** se si vuole creare un file con estensione .h.
- Editare il file che si è creato aggiungendo tutte le istruzioni C++ necessarie. Quindi salvare il file stesso utilizzando la voce **Save** dal menu' **File**.
- Procedere in questo modo per tutti i file che si intende aggiungere al progetto.

Riportiamo, per maggiore comprensione la sequenza grafica delle operazioni descritte:
E', altresì, possibile importare dei file .cpp o .h in un progetto esistente. Per eseguire tale operazione, si procede così:

- Selezionare dal Menu' **File**, la voce **Open**
- Apparirà la seguente finestra:
- Selezionare il file che si desidera aprire (nell'esempio sopra è cane.cpp).
- Premere quindi il tasto **Open**.
- A questo punto il file verrà visualizzato all'interno del Visual Studio.
- Con il tasto destro del mouse cliccare all'interno della finestra che visualizza il file.

Selezionare quindi la voce *Insert File into Project* e selezionare, quindi, il nome del progetto a cui aggiungere il file. Ad esempio:
Dopo aver inserito tutti i file necessari al progetto si deve procedere alla compilazione per poter poi eseguire il progetto stesso.
Per compilare l'intero progetto selezionare il menù **Build** e selezionare la voce **Build MioProg.exe**.
Nella finestra dei Messaggi in basso apparirà il risultato della compilazione. Se non ci sono stati errori, verrà visualizzata la frase: **0 Errors, 0 Warnings**. 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.
Quando non vengono più riscontrati errori è possibile eseguire il progetto. Sarà sufficiente aprire il Menù **Build** e selezionare la voce Execute MioProg.exe.
Naturalmente il Visual Studio è uno strumento molto potente che consente di eseguire molte altre operazioni più complesse che però esulano da questo tutorial. Chi volesse approfondire tale argomento può fare riferimento alla guida in linea del Visual Studio stesso.
Nota: Il procedimento sopra descritto è relativo alla versione 5.0 del Microsoft Visual Studio.
Il procedimento di compilazione è, decisamente, più semplice se si preferisce utilizzare l'ambiente Unix. Infatti, il sistema operativo Unix è fornito del compilatore **gcc** (dove la lettera g sta per GNU, un progetto sviluppato per la compilazione dei programmi in ambiente Unix dalla Free Software Foundation).
Per compilare un file **test.cpp** contenente un programma sarà sufficiente digitare al prompt dei comandi:
g++ test.cpp -o test
ed ottenere automaticamente il programma eseguibile.
Naturalmente, nel caso di programmi più complessi sarà necessaria una maggiore conoscenza delle opzioni di compilazione.
Il seguente programma C++ illustra le componenti fondamentali di un'applicazione C++.

```
//
// PRIMO.CPP
// Il primo esempio in C++
//

#include

main()
{
cout << " CIAO MONDO! ";
return (0);
}
```

Si pensi che già in questa piccola porzione di codice sono contenute numerose operazioni. Innanzitutto i **commenti**:

```
//
// PRIMO.CPP
// Il primo esempio in C++
//
```

E' importantissimo dire che ogni programma ben realizzato dovrebbe includere dei commenti. Nel realizzare un commento si deve rispettare l'intelligenza dei programmatori e nel contenpo non dare tutto troppo per scontato. Si tenga presente che spesso un programmatore è costretto a "mettere le mani" nel codice scritto da altri sviluppatori e, se il codice non è ben commentato, tale operazione può rivelarsi molto più ardua del previsto. Infine, i commenti sono utili anche per l'autore del programma stesso: pensate a cosa accadrebbe se doveste riprendere il codice scritto da voi stessi tre anni fa!
Dunque 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.
Ma vediamo un esempio per comprendere meglio. Le stesse linee commentate del programma precedente, diverrebbero con l'uso del commento a blocchi:

```
/*
PRIMO.CPP
Il primo esempio in C++
*/
```

Semplice, no? I due tipi di commenti sono assolutamente intercambiabili. E' ovvio che se avete la necessità di commentare parecchie linee di codice consecutive è conveniente ricorrere al commento a blocchi.
Nota: Diversi ambienti di sviluppo (come ad esempio il Micrisoft Visual Studio) evidenziano i commenti in verde per meglio permettere al programmatore di distinguere tra codice e commenti stessi. Anche in questa guida, utilizzeremo tale convenzione.
L'istruzione successiva:

```
#include &lt;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. Nel nostro caso, l'istruzione chiede al compilatore di ricercare il codice memorizzato nel file iostream.h e di inserirlo nel punto indicato del codice sorgente. Un file come iostream.h viene chiamato file di intestazione (o file header). Tali file, di solito, includono la definizione di classi, costanti simboliche, identificatori e prototipi di funzione. In seguito vedremo bene di cosa si tratta.
Dopo l'istruzione **#include** si trova la dichiarazione della funzione **main()**:

```
main()
{
.............................................
return (0);
}
```

Ogni programma C++ è formato da chiamate a funzioni. Tutti i programmi C++ devono contenere una funzione chiamata main(). Tale funzione rappresenta il punto di inizio dell'esecuzione del programma che termina con l'istruzione return (0). E', anche, possibile utilizzare istruzioni return() senza parentesi.
Dopo l'intestazione della funzione main(), si trova il corpo della funzione stessa. Si noti la presenza dei simboli **{** e **}**. Dal punto di vista tecnico, le parentesi graffe permettono di incapsulare istruzioni composte. Tali 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 dela validità o meno del test di un'istruzione.
La riga:

```
cout << " CIAO MONDO! ";
```

rappresenta un semplice esempio di istruzione di output che stampa sullo schermo la frase:
CIAO MONDO! .

Gli identificatori

Per poter apprezzare tutto ciò che il C++ ha da offrire sono necessari tempo e pratica. In questo capitolo 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 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**, enumerazioni o **enum**, non-valori o **void** e puntatori.

- 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 tipi enumerativi sono definiti dall'utente (li vedremo meglio in dettaglio tra poco).
- Il tipo void indica valori che occupano 0 byte e non hanno alcun valore.
- Il tipo puntatore non contiene vere e proprie informazioni ma l'indirizzo di una cella di memoria contenente tali informazioni. Vedremo meglio tale concetto in seguito.

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®
Vediamo adesso un semplice programma C++ che illustra la dichiarazione e l'uso del tipo char:

```
/*
* Un semplice programma C++ che mostra
* l'uso del tipo char utilizzato sia come
* contenitore di caratteri che come
* contenitore di valori numerici interi
*/
```

```
#include

main()
{
char carattere;
char num;

printf(""Inserire un carattere a scelta e premere
INVIO: ");
scanf("%c",& carattere);
printf("Inserire un intero compreso tra 0 e 127: ");
scanf("%d",& num);
printf("Il carattere inserito e': %cn",carattere);
printf("Il numero inserito e': %dn",num);

return(0);
}
```

L'esempio appena visto utilizza le funzioni **printf e scanf** che sono derivate dal linguaggio C. La funzione printf viene qui utilizzata per visualizzare sullo schermo le informazioni desiderate, mentre la scanf è usata per leggere l'input dell'utente.
Entrambe queste funzioni sono definite nel file d'intestazione **stdio.h**, ed entrambe utilizzano dei codici di formattazione a seconda del tipo di variabile che l'utente tratta.
Il codice **%c** indica che si sta trattando un carattere mentre il codice **%d** è utilizzato con gli interi. Quando si scrive: scanf("%c",& carattere) si intende dire che si vuole leggere l'input dell'utente come carattere (quindi verrà considerato soltanto il primo elemento immesso) e memorizzare tale valore nella variabile carattere.
Invece, la riga scanf("%d",& num) viene utilizzata per leggere un intero decimale e memorizzarlo nella variabile num.
Similmente, quando si scrive:
printf("Il carattere inserito e': %cn",carattere)
si sta stampando sullo schermo la stringa: "Il carattere inserito e': " più il char identificato dalla variabile carattere e dal codice di formattazione %c.. Ovvero, se si è inserito il carattere 'c' verrà, quindi, stampato:
Il carattere inserito e' c.
Stesso discorso per l'ultima istruzione del programma:
printf("Il numero inserito e': %dn",num);
dove invece viene stampato un intero identificato dal codice %d e dalla variabile num.
Quello che va notato, è che sia la variabile carattere che la variabile num sono entrambi char. Nel primo caso, come visto, la variabile char carattere viene utilizzata per conservare un carattere. Nel secondo caso, la variabile char num viene utilizzata per conservare un intero, che però non può essere più grande di 127 (si legga la spiegazione di ciò nel paragrafo successivo).
Infine, va detto che il codice di formattazione nel caso di variabili float o double è %f.

Gli interi

Il C++ presenta una buona varietà di tipi numerici interi, poichè ai tipi fondamentali:
char intero che occupa un byte (8 bit)
int intero che occupa due byte (16 bit)
permette di aggiungerere 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

main( )
{
float raggio;
float pigreca = 3.14;
float area;
cout << "Inserire il raggio: ";
cin >> raggio;
cout << endl;
area = raggio * raggio * pigreca;
cout << "L'area del cerchio e': " << area << endl;
}
```

Enumerazioni

Quando si definisce una variabile di tipo enumerativo, ad essa viene associato un insieme di costanti intere chiamato insieme dell'enumerazione. La variabile può contenere una qualsiasi delle costanti definite, le quali possono essere utilizzate anche tramite nome. Ad esempio, la definizione:

```
enum secchio { VUOTO,
MEZZO_PIENO,
PIENO = 5} mio_secchio;
```

crea il tipo enum secchio, le costanti VUOTO, MEZZO_PIENO e PIENO e la variabile enumerativa mio_secchio. Tutte le costanti e le variabili sono di tipo int e ad ogni costante è fornito in maniera automatica un valore iniziale standard a meno che venga specificato in modo esplicito un diverso valore. Nell'esempio precedente, alla costante VUOTO viene assegnato automaticamente il valore int 0 in quanto si trova nella prima posizione e non viene fornito un valore specifico. Il valore di MEZZO_PIENO è 1 in quanto si trova immediatamente dopo una costante il cui valore è zero. La costante PIENO viene inizializzata, invece, esplicitamente al valore 5. Se, dopo la costante PIENO, si fosse introdotta una nuova costante, ad essa sarebbe stato automaticamente assegnato il valore int 6.
Sarà possibile, dopo aver creato il tipo secchio, definire un'altra variabile, tuo_secchio nel modo seguente:
secchio tuo_secchio;
Od anche, in modo del tutto equivalente:
enum secchio tuo_secchio;
Dopo tale istruzione si potranno usare i seguenti assegnamenti:
mio_secchio = PIENO;
tuo_secchio = VUOTO;
che assegnano alla variabile mio_secchio il valore 5 ed alla variabile tuo_secchio il valore 0.
Un errore che spesso si commette è quello di pensare che secchio sia una variabile. Non si tratta di una variabile ma di un tipo di dati che si potrà utilizzare per creare ulteriori variabili enum, come ad esempio la variabile tuo_secchio.
Poiché il nome mio_secchio è una variabile enumerativa di tipo secchio, essa potrà essere utilizzata a sinistra di un operatore di assegnamento e potrà ricevere un valore. Ad esempio, si è eseguito un assegnamento quando si è assegnato alla variabile il valore PIENO. I nomi VUOTO, MEZZO_PIENO e PIENO sono nomi di costanti; non sono variabili e non è possibile in alcun modo cambiarne il valore.

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:

```
loat 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'.

Riqualificazione di tipi di variabili : l'operatore di conversione (cast)

Il C++, come la stragrande maggioranza dei linguaggi di programmazione, permette alcune operazioni matematiche anche tra variabili di tipo diverso e il compilatore, se l'operazione ha significato, esegue automaticamente la riqualificazione al tipo di livello più elevato tra quelli dei due operandi; per esempio tra un operando di tipo int e l'altro di tipo float prima di eseguire l'operazione viene eseguita la riqualificazione. In alcuni casi, però, può essere utile forzarne l'utilizzo secondo le esigenze del programmatore. Ad esempio, un intero con segno considerarlo senza segno oppure considerarlo float aggiungendo decimali fittizi. In C++ questa esigenza viene soddisfatta molto semplicemente premettendo alla variabile il ripo con il quale si vuole riqualificarla, tipo che viene messo tra parentesi.
Per esempio, se la variabile a è di tipo int viene riqualificata come float nel seguente modo:
(float) a
Con il C++ la riqualificazione è molto spesso un'operazione indispensabile nella divisione tra interi. Per esempio:

```
int a = 9, b = 2;
float c;

c = a/b; // Si ottiene c = 4.0

c = (float) a/b; // a è riqualificato come float e si ottiene
// correttamente c = 4.5

c = (float) (a/b); // viene riqualificato (a/b) e si ottiene
// ancora c = 4.0
```

Infatti:

- Nel primo caso, pur essendo c di tipo float, il C++ nrl gstr la divisione tra interi non considera i decimali.
- Nel secondo caso il compilatore riqualifica a come float e poi esegue la divisione riqualificando anche b come float e il risultato è corretto.
- Nel terzo caso prima viene eseguita la divisione tra interi, perdento quindi i decimali, poi la riqualificazione ma ormai i decimali sono persi.

Vale, infine, la pena di osservare che, riqualificando una variabile, non si modifica il contenuto della variabile stessa ma soltanto il valore che interviene nell'operazione in corso.

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 sintassi di #**define** è:
#define & &
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 &
#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 assegnazione =.
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 la direttiva #define ha un utilizzo molto più generale che va oltre la definizione di una semplice **costante**. Infatti, essa permette anche la definizione delle cosidette 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 &
#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** ha, invece, la seguente sintassi:
const & = &;

Si faccia attenzione al fatto che qui viene usato sia l'operatore di assegnazione = 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.

Visibilità delle variabili e delle costanti

Sostanzialmente, vi sono **quattro** regole di visibilità delle variabili, dette anche regole di campo d'azione. I quattro campi d'azione di variabili e costanti sono: il blocco, la funzione, il file e il programma. Una variabile dichiarata in un blocco o in una funzione può essere utilizzata solo all'interno di quel blocco o di quella funzione. Una variabile dichiarata al di fuori di una funzione può essere utilizzata all'interno del file nel quale appare, a partire dal punto in cui è dichiarata. Una variabile dichiarata come **extern** in altri file può essere utilizzata in tutto il programma.

Le parole riservate

Le **parole riservate** sono identificatori predefiniti che hanno un particolare significato per il compilatore C/C++. Queste parole possono essere utilizzate solo nel modo in cui sono definite. è importante ricordare che un identificatore di programma non può essere identico a una parola riservata del C/C++ e che non è possibile, in alcun modo, ridefinire le parole riservate. Le parole riservate del C/C++ sono elencate nella seguente **tabella**:

auto	based	break
float	case	for
short	signed	char
goto	sizeof	const
static	continue	If
struct	default	Int
switch	dllexport	thread
dllimport	do	typedef
double	long	union
else	naked	unsigned
enum	void	volatile
register	while	extern
return	class	operator
virtual	delete	private
friend	protected	inline
public	new	this
argc	envp	argv
main		

Operatori booleani

Il C++ mette a disposizione del programmatore un numero superiore di operatori di quanto non facciano altri linguaggi, ma presenta anche lo svantaggio che alcuni non sono certo facilmente interpretabili perché i loro simboli non hanno un immediato riferimento mnemonico alla funzione svolta.
Vediamo i più importanti operatori del C++.
Operatori Booleani: true e false
Prima di parlare degli operatori booleani, è importante sapere cosa è un boolean.
Un boolean può dar vita ad uno dei seguenti valori: **true** o **false**. Nessun altro valore è permesso.
Gli identificatori boolean e le operazioni su di essi sono alla base della programmazione. Svariate volte, 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 nel caso in cui si verificasse il secondo caso.
Un esempio grafico particolarmente attinente alle operazioni con boolean è rappresentato dalle check box. Se una check box e' selezionata si vorrà intraprendere una determinata azione. In caso contrario non si vorrà fare nulla.
La maggior parte dei linguaggi di programmazione hanno il tipo boolean. La maggior parte dei compilatori C++ riconosce il tipo boolean con la scrittura **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 boolean sarà la seguente:
bool flag;

Gli operatori di tipo booleano, ovvero gli operatori che consentono di eseguire operazioni su elementi di tipo boolean sono 3. Essi sono:

- Operatore di AND -> &&
- Operatore di OR -> ||
- Operatore di NOT -> !

Ognuno degli **operatori** precedenti prende in input uno o due booleani e restituisce in output un altro booleano.
Vediamo in dettaglio:
L'**operatore di 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.
L'**operatore di 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.
L'**operatore di NOT**, prendi 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'e' 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. E' 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 verrà assegnato il numero 15.
E', pero', 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;
```

Il secondo tipo di utilizzo è molto frequente all'interno dei cicli while e for:

```
while ((c = getchar()) != EOF)
{
..........................................
}
```

oppure

```
for(int i = 0; i < 10; i++)
{
.......................................
}
```

Operatori di uguaglianza

Un operatore di uguaglianza è un operatore che **verifica** determinate condizioni 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 uguaglianza può essere determinante per la scelta di una determinata strada piuttosto che di un'altra. Ad esempio, se una determinata variabile supera un determinato valore potrebbe essere necessario chiamare una funzione per **controllare** altri parametri, e cosi' via.
Ma vediamo nella seguente tabella, l'elenco completo degli operatori di uguaglianza:

ratori di Uguaglianza			
Nome	**Simbolo**	**Esempio d'uso**	**Risultato**
Minore	<	bool ris = (3 < 8)	true
Maggiore	>	bool ris = (3 > 8)	false
Uguale	=	bool ris = (5 == 5)	true
Minore Uguale	<=	bool ris = (3 <= 5)	true
Maggiore Uguale	>=	bool ris = (5 >= 9)	false
Diverso	!=	bool ris = (4 != 9)	true

Livelli di precedenza degli operatori

I **livelli di precedenza** degli operatori indicano 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;
Quale sarà il **risultato** in questo caso? La risposta dipende proprio dalla precedenza degli operatori. In C++, l'operatore di moltiplicazione (*) ha precedenza rispetto all'operatore addizione (+). Ciò vuol dire che la moltiplicazione 5 * 7 avverrà prima di tutte le altre addizioni.
Avremo quindi, al termine:
risultato = 4 + 35 + 3 = 42
Come operare allora se si volesse che l'operazione precedente venisse interpretata con un **ordine diverso**? Ad esempio, se si volesse che l'operatore di moltiplicazione venisse utilizzato per moltiplicare la somma di 4 + 5 con la somma di 7 + 3, come si dovrebbe procedere?
La risposta è semplicissima: basta utilizzare le parentesi tonde. Quindi basterà scrivere:
int risultato = (4 + 5) * (7 + 3);
e la variabile risultato, in questo caso, varrà 90.
E' molto importante capire che le **parentesi** risolvono praticamente il problema di conoscere la precedenza tra gli operatori. Personalmente consiglio vivamente l'uso delle parentesi in ogni situazione in cui vi sia la presenza di più operatori contemporaneamente anche perchè la lettura del codice ne risulta certamente migliorata.
Ad ogni modo, un **livello di precedenza** tra gli operatori esiste ed è giusto descriverlo. La tabella seguente è suddivisa in gruppi. Tutti gli operatori appartenenti ad uno stesso gruppo hanno lo stesso livello di precedenza mentre gli operatori appartenenti ad un gruppo hanno maggior precedenza di tutti gli altri operatori che compaiono nei gruppi sottostanti.

Operatore	**Nome**
!	Not
* / %	Moltiplicazione Divisione Modulo
+ -	Addizione Sottrazione

< <= > >=	Minore Minore Uguale Maggiore Maggiore Uguale
== !=	Uguale Diverso
&&	AND
||	OR
=	Assegnamento

Le istruzioni if e else

Il C++ utilizza quattro istruzioni condizionali principali: if, if-else, ? condizionale e switch. Prima di affrontare tali istruzioni una per una, è bene dire che la maggior parte delle istruzioni condizionali conseente di eseguire in modo selettivo una singola riga di codice o una serie di righe di codice (che viene detto blocco di riche di codice). 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. Personalmente, comunque, per una buona lettura del codice consiglio vivamente 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 ed 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>)
}
Si faccia attenzione a non dimenticare il punto e virgola prima dell'istruzione else. Ad esempio, se deve scrivere:

......
if(i>=0)
{
<istruzione 1>;
}
else
{
<istruzione 2>;
}
....
L'istruzione 1 verrà eseguita solo se i >= 0 mentre in caso contrario viene eseguita l'istruzione 2.

istruzioni if-else nidificate

Quando si utilizzano più istruzioni if nidificate, occorre sempre essere sicuri dell'azione **else** che verrà associata ad un determinato if. Si cerchi di capire cosa accade nell'esempio seguente:
if(temperatura < 20)
if(temperatura < 10) cout << "Metti il cappotto!n";
else cout << " Basta mettere una felpan";
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 potrebbero accadere dei casi in cui appaiono tante istruzioni if-else nidificate con la conseguenza che la lettura del codice sarebbe davvero difficile per chiunque. Allora, per rendere il codice leggibile sarà buona norma fare uso sempre delle parentesi graffe e dell'indentazione del codice.
Indentare il codice altro non vuol dire che 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";
}
}
```

che 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;
```

l'istruzione condizionale ?

L'istruzione **condizionale ?** fornisce un modo rapido per scrivere una condizione di test. Le azioni associate all'istruzione condizionale vengono eseguite in base al valore dell'espressione (trUE oppure FALSE). Quindi l'operatore **?** può essere utilizzato per sostituire un'istruzione if-else. La sintassi di un'istruzione condizionale è:

```
espressione_test ? azione_true : azione_false;
```

L'operatore ? viene anche denominato operatore ternario in quanto richiede tre operandi. Si osservi l'istruzione seguente:

```
bool positivo;
if(valore >= 0.0)
{
positivo = true;
}
else
{
positivo = false;
}
```

Ecco come si può riscrivere la stessa sequenza di istruzioni con l'operatore condizionale:

```
bool positivo;
positivo = (valore>=0.0) ? true : false;
```

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 varibaile 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>

int 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)
{
cerr << numero_corrente << endl;
numero_corrente = numero_corrente + 2;
}

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

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.
L'output del programma precedente, sarà allora il seguente:

```
12

14

16

18

20

22

Fine del Programma!
```

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 istuzioni. 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++;
}

return (0);
}
```

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.

Le funzioni

Finora, negli esempi che abbiamo visto, tutte le istruzioni eseguivano operazioni molto semplici, come assegnamenti, controlli booleani od operazioni aritmetiche. Le funzioni rappresentano una parte fondamentale della programmazione in C++ in quanto permettono di eseguire con una sola linea di codice un insieme di operazioni. Cosa vuol dire questo? Naturalmente non bisogna pensare che ci sia qualche magia sotto o che il computer sappia da solo quali operazioni vogliamo che esso compia. Più semplicemente, le funzioni servono a racchiudere un pezzo di funzionalità del programma (composto naturalmente da linee di codice) e a far si che in ogni momento sia possibile richiamare tale funzionalità da altri punti del programma stesso. In altre parole, possiamo dire che il C++ permette di identificare una serie di istruzioni con un preciso nome: quello, appunto, di una funzione.
L'enorme utilità delle funzioni risiede nel fatto che grazie ad esse è possibile scrivere il codice che esegue una particolare azione una volta per tutte e riutilizzarlo, d'altra parte, tutte le volte che si vuole. Comodo no?

Prototipi

Secondo lo standart ANSI (American National Standard Institute), tutte le funzioni devono avere un corrispondente prototipo. E' buona norma far risiedere il prototipo in un file d'intestazione (file header, con estensione .h) anche se questa non è una regola obbligatoria. Un prototipo di funzione ha la seguente forma:

```
tipo_restituito nome_funzione
(tipo_argomento1 nome_argomento1, ......);
```

Una funzione può essere di tipo void, int, float ecc.. Il tipo restituito indica appunto il tipo del valore restituito dalla funzione. Il nome_funzione può essere un nome qualunque scelto per descriere lo scopo della funzione. Se la funzione riceve parametri, sarà necessario specificare il tipo_argomento seguito da un nome_argomento. Anche il tipo degli argomenti può essere void, int, float, ecc.. Se a una funzione si passano più argomenti, questi dovranno essere separati da una virgola.
La scrittura della funzione vera e propria è essa stessa una porzione di codice C++ che normalmente segue la definizione delle funzione main(). Una funzione può quindi assumere la seguente forma:

```
tipo_restituito nome_funzione(tipo e nome_argomenti)
{
dichiarazione dei dati e corpo della funzione
return();
}
```

Si noti che la prima riga della funzione è identica al prototipo inserito nel file header, ma con un'importante differenza: manca il punto e virgola finale. Vediamo ora un esempio che mostra un prototipo e una funzione in un programma:

```
/*
*Un semplice programma che illustra l'uso dei prototipi
*di funzioni. La funzione, esegue la moltiplicazione tra
*due interi e restituisce il risultato
* /

#include <iostream.h>

int moltiplica(int x, int y);
// Definizione del prototipo

main()
{
int a = 5;
int b = 6;
int risultato;

risultato = moltiplica(a,b);
cout << Il risultato della moltiplicazione è:
<< risultato <<endl;

return (0);
}

int moltiplica(int x, int y)
// Dichiarazione della funzione
{
int ris;
```

```
ris = x * y;
return ris;
// Valore restituito dalla funzione.
}
```

Chiamata per valore e per indirizzo

Nell'esempio precedente, gli argomenti sono stati passati alla funzione per valore. Quando il passaggio dei parametri avviene per valore, alla funzione viene in effetti passata solo una copia dell'argomento. Grazie a questo meccanismo il valore della variabile nel programma chiamante non viene modificato. Il passaggio di parametri per valore è un modo molto comune per passare informazioni a una funzione ed è anche il modo normalmente utilizzato dal C++.
In una chiamata per indirizzo, alla funzione viene passato l'indirizzo e non il valore dell'argomento. Questo approccio richiede meno memoria rispetto alla chiamata per valore ma quando si utilizza la chiamata per indirizzo le variabili del programma chiamante possono essere modificate. Per ottenere questo risultato è necessario passare alla funzione degli argomenti di tipo puntatore (Vedremo in seguito cosa sono i puntatori). Un vantaggio in più del passaggio per indirizzo rispetto a quello per valore è che la variabile può restituire più di un valore.
Nell'esempio qui di seguito riproponiamo la funzione che calcola la moltiplicazione tra due numeri ma con i parametri passati per indirizzo.

```
#include <iostream.h>

int moltiplica(int* x, int* y);
// Definizione del prototipo

main()
{
int a = 5;
int b = 6;
int risultato;

risultato = moltiplica(& a, & b);
cout << Il risultato della moltiplicazione è:
<< risultato <<endl;

return (0);
}


int moltiplica(int* x, int* y)
// Dichiarazione della funzione
{
int ris;

ris = *x * *y;
return ris;
// Valore restituito dalla funzione.
}
```

In C++ esiste ancora una terza possibilità nel passaggio degli argomenti ad una funzione: si tratta del passaggio per riferimento (o reference). Il tipo reference è un puntatore ad un indirizzo di memoria, ma non richiede l'uso di un operatore di indirizzamento. E' possibile allora utilizzare questa utilizzare questa sintassi per semplificare l'uso delle variabili puntatore nelle chiamate alle funzioni. Sarà utile confrontare la sintassi dell'esempio precedente con quello che segue:

```
#include <iostream.h>

int moltiplica(int& x, int& y);
// Definizione del prototipo

main()
{
int a = 5;
int b = 6;
int risultato;

risultato = moltiplica(a, b);
cout << Il risultato della moltiplicazione è:
<< risultato <<endl;

return (0);
}


int moltiplica(int& x, int& y)
// Dichiarazione della funzione
{
int ris;

ris = x * y;
return ris;
// Valore restituito dalla funzione.
}
```

Infine, è utile ricordare che in C++ possono tranquillamente esistere funzioni che non prendono nessun argomento e/o che non restituiscono alcun valore. Queste ultime ricordano le procedure del Pascal che servivano, appunto, per definire delle subroutine di codice che non restituivano alcun valore. In tal caso, nel C++, il tipo restituito dalla funzione sarà void.
Esempio:

```
void stampaCiao()
{
cout <<"Ciao" <<endl;
}
```

Campo di azione o visibilità

Quando si parla di campo d'azione o visibilità di una variabile utilizzata in una funzione si fa riferimento alla parte del programma in cui è visibile tale variabile. Le regole di visibilità delle variabili utilizzata nelle funzioni, sono simili in C e C++. Le variabili possono avere visibilità locale, di file o di classe. La visibilità di classe sarà discussa più avanti.
Una variabile locale può essere utilizzata esclusivamente all'interno della definizione di una funzione. La sua visibilità è quindi limitata alla funzione. La variabile si dice accessibile o visibile solo all'interno della funzione e ha quindi visibilità locale.
Le variabili con visibilità di file vengono dichiarate all'esterno delle singole funzioni o classi e possono essere utilizzate nell'intero file. Le variabili di questo tipo sono anche dette variabili globali. Può accadere il caso in cui si utilizza lo stesso nome di variabile prima all'esterno, con visibilità di file, e successivamente all'interno della definizione di una funzione e quindi con visibilità locale; in tal caso la precedenza negli accessi sarà attribuita alla variabile locale. Il C++ offre però la possibilità di utilizzare l'operatore di risoluzione del campo d'azione (::).
Questo operatore consente di accedere alla variabile con campo d'azione globale anche dall'interno del campo d'azione di una variabile locale dotata dello stesso nome. La sintassi dell'operatore è:
::variabile.

Gli argomenti della funzione main

Il C++ consente l'uso degli argomenti nella riga di commando. Gli argomenti della riga di comando vengono passati nel momento in cui il programma viene richiamato. Questo consente di passare argomenti al programma senza la necessità di presentare messaggi di richiesta aggiuntivi. Per esempio, un programma potrebbe ricevere tramite la riga di comando quattro argomenti:
PROGRAMMA Primo, Secondo, Terzo, Quarto
In tale esempio vengono passati quattro valori dalla riga di comando al programma. In realtà tali informazioni diverranno i parametri di main(). Uno degli argomenti ricevuti da main() è argc il quale è un valore intero che fornisce il numero di elementi presenti sulla riga di comando più 1. Infatti, il C++ considera come primo elemento il nome del programma stesso. Il secondo argomento è un puntatore a stringhe chiamato argv. Tutti gli argomenti sono stringhe di caratteri e quindi argv è di tipo char* [argc]. Si capirà bene tale definizione quando parleremo di puntatori e di puntatori a puntatori. Naturalmente, poichè tutti i programmi hanno un nome, argc varrà sempre almeno 1.
Vediamo la tecnica utilizzata per leggere le informazioni contenute nella riga di comando:
Vediamo un semplice programma che vuole come input 2 argomenti e che stampa tali argomenti sul video:

```
/*
* Semplice programma che legge gli argomenti
* passati nella linea di comando e li stampa
* a video
*/

include <stdio.h>
#include <process.h>
#include <iostream.h>

main(int argc, char* argv[])
{
int i;
if(argc != 2)
{
cout << "Per eseguire il programma bisogna
inserire due argomenti" << endl;
cout << "Ripetere l'operazione" << endl;
exit(0);
}


for(i = 1; i < argc; i++)
{
printf("Argomento %d è %sn", i, argv[i]);
}

return (0);

}
```

L' overloading

L'overloading delle funzioni è una funzionalità specifica del C++ che non è presente in C. Questa funzionalità permette di poter utilizzare lo stesso nome per una funzione più volte all'interno dello stesso programma, a patto però che gli argomenti forniti siano differenti. In maniera automatica, il programma eseguirà la funzione appropriata a seconda del tipo di argomenti passati.
Esempio:

```
#include <iostream.h>

int moltiplica(int x, int y);
// Definizione dei prototipi
float moltiplica (float x, float y);

main()
{
int a = 5;
int b = 6;
int risultato;
float c = 7.9;
float d = 9.2;
float risultato1;

risultato = moltiplica(a, b);
cout << Il risultato della moltiplicazione tra interi è: << risultato <<endl;

risultato1 = moltiplica (c, d);
cout << Il risultato della moltiplicazione tra float è: << risultato1 <<endl;

return (0);
}


int moltiplica(int x, int y)
// Dichiarazione della funzione
{
int ris;

ris = x * y;
return ris;
// Valore restituito dalla funzione.
}


float moltiplica(float x, float y)
// Dichiarazione della funzione
{
float ris;

ris = x * y;
return ris;
// Valore restituito dalla funzione.
}
```

Se si prova ad eseguire il precedente esempio, ci si accorgerà che il programma chiamerà in modo automatico la funzione appropriata in base al tipo di argomenti fornito.
Si noti che per poter fare l'overloading di una funzione non basta che soltanto il tipo restituito dalla funzione sia differente, ma occorre che anche gli argomenti lo siano.

Gli Array

Un array rappresenta una variabile indicizzata (ovvero contenente un indice) che viene utilizzata per contenere più elementi dello stesso tipo. Ogni array ha un nome al quale viene associato un indice che individua i singoli elementi dell'aray. In C++ è possibile creare array di qualunque tipo: caratteri, interi, float, double puntatori e anche array (ovvero array di array, detti array multidimensionali).

Le proprietà fondamentali di un array

Possiamo dire che un array ha quattro proprietà fondamentali:

- Gli oggetti che compongono l'array sono denominati elementi;
- Tutti gli elementi di un array devono essere dello stesso tipo;
- Tutti gli elementi di un array vengono memorizzati uno di seguito all'altro nella memoria del calcolatore e l'indice del primo elemento è 0;
- Il nome dell'array è un valore costante che rappresenta l'indirizzo di memoria del primo elemento dell'array stesso.

Dichiarazione di un array

Per dichiarare un array, come per ogni altra dichiarazione in C++, si deve scrivere il tipo, seguito da un nome valido e da una coppia di parentesi quadre che racchiudono un'espressione costante. Tale espressione costante definisce le dimensioni dell'array. Ad esempio:

```
int array_uno[10]; // Un array di 10 interi
char array_due[20]; // Un array di 20 caratteri
```

Si noti che all'interno delle parentesi quadre non è possibile, in fase di dichiarazione dell'array, utilizzare nomi di variabili . E' possibile, per specificare le dimensioni di un array, usare delle costanti definite, come ad esempio:

```
#define ARRAY_UNO_MAX 10
#define ARRAY_DUE_MAX 20
```

```
int array_uno[ARRAY_UNO_MAX];
char array_due[ARRAY_DUE_MAX];
```
L'utilizzo di costanti per definire le dimensioni di un array è una pratica altamente consigliata. Infatti, uno degli errori che spesso accadono nella manipolazione degli elementi di un array è quello di far riferimento ad elementi che vanno oltre il limite della dimensione dell'array precedentemente definita.
Il C++ (e lo stesso C), come magari si sarebbe portati a pensare, non effettua nessun controllo sugli indici che si utilizzano quando si eseguono operazioni sugli array. Per cui, ad esempio, se eseguiamo il seguente codice:
```
int array_tre[10];
......
...... // L'array viene inizializzato
int i;
for(i=0; i < 14; i++)
{
cout << array_tre[i] << endl;
}
```
il risultato sarà corretto fino a quando la variabile i sarà minore o uguale a 9 (infatti la dimensione dell'array è stata definita essere uguale a 10). Quello che avverrà, quando il programma cercherà di andare a leggere i valori array_tre[10], array_tre[11], array_tre[12] e array_tre[13] è assolutamente casuale. Infatti, il programma andrà a leggere delle aree di memoria che non sono state inizializzate o, addirittura, che contengono dei valori relativi ad altre variabili, stampando quindi sullo schermo risultati inaspettati. Se si fosse usata una costante per definire la dimensione dell'array questo errore sarebbe stato facilmente evitato. Infatti:
```
#define ARRAY_trE_MAX 10

int array_tre[ARRAY_trE_MAX];
......
...... // L'array viene inizializzato
int i;
for (i=0; i < ARRAY_trE_MAX; i++)
{
cout << array_tre[i] << endl;
}
```
Come si può notare facilmente in questo modo il programmatore non deve preoccuparsi di ricordarsi la dimensione dell'array array_tre poichè la costante ARRAY_trE_MAX viene utilizzata proprio per tale scopo, evitando i cosìdetti errori di "array out of bounds".

Inizializzazione di un array

Un array può essere inizializzato in due modi:

- Esplicitamente, al momento della creazione, fornendo le costanti di inizializzazione dei dati.
- Durante l'esecuzione del programma, assegnando o copiando dati nell'array.

Vediamo un esempio di inizializzazione esplicita al momento della creazione:
```
int array_quattro[3] = {12, 0, 4};
char vocali[5] = {'a','e','i','o','u'};
float decimali[2] = {1.329, 3.34};
```
Per inizializzare un array durante l'esecuzione del programma occorre accedere, generalmente con un ciclo, ad ogni elemento dell'array stesso ed assegnargli un valore. L'accesso ad un elemento di un array avviene indicando il nome dell'array e, tra parentesi quadre, l'indice corrispondente all'elemento voluto. Così, x[2] indicherà il terzo elemento dell'array x e non il secondo poichè la numerazione degli indici, come abbiamo detto, inizia da zero.
Vediamo ora un esempio di inizializzazione eseguita durante l'esecuzione del programma:
```
int numeri_pari[10];
int i;

for(i =0; i < 10; i++)
{
numeri_pari[i] = i * 2 ;
}
```
Il programma precedente non fa altro che assegnare all'array i numeri pari da 0 ad 18.

Stringhe

Il C++, contrariamente ad altri linguaggi di programmazione, non mette a disposizione del programmatore il tipo stringa. Per identificare, quindi, una stringa sarà necessario definire un array di caratteri. Si tenga presente che in un array di caratteri che rappresenti una stringa bisogna sempre inserire un elemento in più che rappresenta il carattere terminatore di stringa, ovvero: \0.Vediamo un semplice programma che mostra i tre metodi per inizializzare una stringa:
```
/*
* Un semplice programma che mostra
* l'uso delle stringhe
*
*/

include <iostream.h>

main()
{
char stringa1[6];
char stringa2[6];
char stringa3[5] = "nave";
```

```
// Inizializzazione durante la definizione
// Inizializzazione della stringa stringa1
stringa1[0] = 'p';
stringa1[1] = 'a';
stringa1[2] = 'l';
stringa1[3] = 'l';
stringa1[4] = 'a';
stringa1[5] = '\0';
```

L'output del programma sarà:
palla
pippo (Supponendo che l'utente abbia inserito la stringa pippo)
nave

Cenni sugli Array Multidimensionali

Come abbiamo visto, il termine dimensione rappresenta il numero di indici utilizzati per far riferimento a un determinato elemento dell'array. Gli array che abbiamo visto finora erano monodimensionali e richiedevano l'uso di un solo indice. E' facile determinare il numero di dimensioni di un array osservando la sua dichiarazione. Se nella dichiarazione è presente una sola coppia di parentesi quadre ([ ]) l'array è monodimensionale. Due coppie di parentesi quadre ([ ] [ ]) indicano, invece, un array bidimensionale, e così via. Il numero massimo di dimensioni normalmente solitamente non supera 3. Per immaginare la rappresentazione di un array bidimensionale si può far riferimento ad una tabella (o a una matrice) in cui il primo indice dell'array rappresenta la riga ed il secondo rappresenta la colonna.
passaggio di array a funzioniCome avviene per tutte le altre variabili, così anche gli array possono essere passati come parametro ad una funzione. Il passaggio di un array ad una funzione avviene sempre per indirizzo e mai per valore. Quando si passa un'array in effetti non si sta facendo altro che passare l'indirizzo del primo elemento dell'array stesso. Ciò vuol dire che se all'interno della funzione vengono modificati i valori dell'array, tale modifica avrà effetto anche sull'array che si è passato alla funzione. Tale nozione sarà più chiara leggendo il capitolo sui puntatori. Vediamo comunque esempio:

```
/*
*Un semplice programma che illustra il passaggio
*di un array ad una funzione
*/

#include <iostream.h>
#define size 5;

void somma(int array[ ])
{
int i;
int somma = 0;

for(i=0; i < size; i++)
{
somma = somma + array[i];
}
cout << " La somma degli elementi dell'array è " << somma << "n";
}

main( )
{
int vettore[size] = {1,2,3,4,5};

somma(vettore);
return(0);
}
```

Come è facile intuire, l'output del programma sarà:
La somma degli elementi dell'array è 15

I puntatori

puntatori rappresentano uno dei punti di forza più rappresentativi del C++. Al tempo stesso, però, spesso sono causa di confusione per i neofiti a causa del loro meccanismo leggermente complesso.
Per poter capire a cosa serve un puntatore, è necessario sapere che la memoria del calcolatore in cui possono risiedere le variabili è suddivisa in due parti:

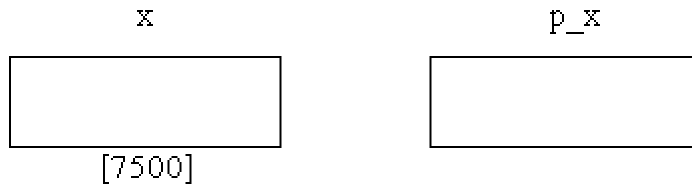
- Lo Stack
- L'Heap

Nello stack vengono immagazzinate tutte le variabili del programma che abbiamo visto finora. Ad esempio quando si definisce:
int i = 0;
Il computer riserverà due byte di memoria dello Stack per la variabile i. Con l'utilizzo di tali variabili non è possibile in alcun modo ottenere più variabili di quelle dichiarate nè l'applicazione è in grado di deallocare la memoria di una variabile. Quest'ultima osservazione può rappresentare un problema quando si manipolano grosse quantità di informazioni in quanto potrebbe succedere che lo stack si riempia e quando si tenta di allocare altra memoria si verifichi il cosìdetto stack overflow.
Per permettere al programmatore di utilizzare la memoria in maniera "intelligente" è possibile allora utilizzare la memoria Heap, detta anche memoria dinamica. Il termine dinamica sta proprio ad indicare che è data la possibilità al programmatore di allocare e deallocare la memoria a suo piacimento.
Questa, tuttavia, è un'operazione molto delicata che, se compiuta in modo errato può portare ad errori spesso difficili da trovare che si

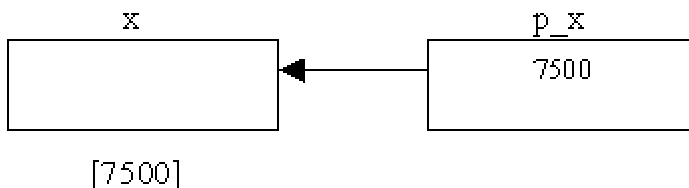
verificano in fase di esecuzione.
La manipolazione dell'Heap avviene proprio tramite i puntatori.

Che cos'è una variabile puntatore

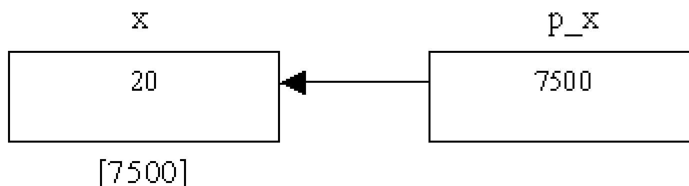
Una variabile puntatore è una variabile che contiene l'indirizzo di memoria di un'altra variabile. Ad esempio, si supponga di avere una variabile intera chiamata x ed un'altra variabile (quella puntatore appunto) chiamata p_x che è in grado di contenere l'indirizzo di una variabile di tipo int. In C++, per conoscere l'indirizzo di una variabile, è sufficiente far precedere al nome della variabile l'operatore **&**. La sintassi per assegnare l'indirizzo di una variabile ad un'altra variabile è la seguente:
p_x = & x;
In genere, quindi, una variabile che contiene un indirizzo, come ad esempio p_x, viene chiamata variabile puntatore o, più semplicemente, puntatore. Per chiarire ulteriormente questo concetto, guardiamo la figura seguente:



Supponiamo che la variabile intera x, raffigurata nel riquadro di sinistra, sia allocata all'indirizzo di memoria 7500. La variabile p_x, ovvero la variabile puntatore, inizialmente non contiene alcun valore (ovvero nessun indirizzo).
Nel momento in cui viene eseguita l'istruzione:
p_x = & x;
la variabile puntatore conterrà l'indirizzo della variabile x. Cioè:



La freccia viene tracciata dalla cella che conserva l'indirizzo verso la cella di cui si conserva l'indirizzo.
Per accedere al contenuto della cella il cui indirizzo è memorizzato in p_x è sufficiente far precedere alla variabile puntatore il carattere asterisco (*). In tal modosi è deindirizzato il puntatore p_x.
Ad esempio, se si prova ad eseguire le due istruzioni seguenti:
p_x = & x;
*p_x = 20;
il valore della cella x sarà 20. Graficamente otterremo:



Si noti che se p_x contiene l'indirizzo di x, entrambe le istruzioni che seguono avranno lo stesso effetto, ovvero entrambe memorizzeranno nella variabile x il valore 20:
x = 20;
*p_x = 20;

Dichiarazione di variabili puntatore

Naturalmente, anche le variabili puntatore come tutte le altre richiedono una definizione per ogni variabile. L'istruzione che segue definisce una variabile puntatore (p_x) che è in grado di contenere l'indirizzo di una variabile int:
int* p_x
Si potrebbe vedere tale dichiarazione come formata da due parti. Il tipo di x:
int*
e l'identificatore della variabile:
p_x
L'asterisco che segue il tipo int può essere interpretato come: "puntatore a" ovvero il tipo:
int*
dichiara una variabile di tipo "puntatore a int".
Naturalmente, è possibile definire variabili puntatori per ogni tipo. Così, potremo scrivere:
char* p_c;
float* p_f;
double* p_d;
E' molto importante tenere presente che se un programma definisce un puntatore di un determinato tipo e poi lo utilizza per puntare ad un oggetto di un altro tipo, si potranno ottenere errori di esecuzione e avvertimenti in fase di compilazione. Anche se il C++ permette simili operazioni è comunque buona norma di programmazione evitarle.

Inizializzazione di una variabile puntatore

Anche per le variabili puntatore esiste la possibilità di ricorrere all'inizializzazione nel momento della loro definizione. Ad esempio:

```
int x;
int* p_x = & x;
char c ;
char* p_c = & c;
```

Il C++ mette a disposizione un tipo particolare di inizializzazione di un puntatore. Infatti è possibile, ed anzi è buona norma farlo, inizializzare una variabile puntatore a NULL (ovvero non la si fa puntare a nessun indirizzo di memoria) .
Ad esempio:

```
int* p_x = NULL;
char* p_c = NULL;
```

Quando però si inizializza un puntatore a NULL, si deve tener presente che non è possible in alcun modo utilizzare la variabile puntatore per nessuna operazione finchè non la si inizializzi con un indirizzo valido.
Infatti, se provassimo a scrivere:

```
int* p_x = NULL;
cout << *p_x << endl;
// Errore!!
```

otterremmo un errore in esecuzione in quanto la variabile puntatore p_x non punta nessuna variabile intera che contenga un valore valido.
Occorrerà far puntare p_x ad una variabile intera:

```
int x = 20;
int* p_x = NULL;
......
......
......
p_x = & x;
cout << *p_x << endl;
// Adesso è corretto
```

Esiste ancora un'altra possibilità per inizializzare una variabile puntatore: utilizzando l'operatore **new** messo a disposizione dal C++.
Quando si scrive:

```
int x = 20;
int* p_x = NULL;
......
......

p_x = new int;
*p_x = x;
```

quello che si ottiene è un risultato apparentemente simile a quello visto negli esempi precedenti, ma in realtà abbastanza diverso.
L'istruzione:

```
p_x = new int
```

altro non fa che allocare una quantità di memoria necessaria per contenere un int (ovvero 2 byte) ed assegnare l'indirizzo del primo byte di memoria alla variabile p_x. Questo indirizzo è certamente diverso da quello in cui è contenuta la variabile x stessa, per cui in tal caso avremo in realtà due variabili intere differenti che contengono lo stesso valore.
E' anche possibile usare l'istruzione:

```
p_x = new int(20);
```

per inizializzare direttamente il valore *p_x a 20.
Si faccia ora attenzione. Abbiamo detto che le variabili puntatore fanno riferimento alla memoria dinamica (heap). Diversamente dalla memoria statica, tutte le variabili puntatore allocate dal programmatore devono essere poi distrutte quando non servono più, per evitare i cosìdetti "memory leaks", ovvero per evitare che la memoria allocata resti tale anche dopo la terminazione del programma.
L'istruzione del C++ che serve per distruggere una variabile puntatore è: **delete**.
Nel caso dell'esempio precedente, avremmo:

```
int x = 20;
int* p_x = NULL;
......
......

p_x = new int;
*p_x = x;
......
......
delete p_x;
// La variabile p_x non serve più. Possiamo deallocarla
......
......
```

Per capire cosa è un memory leak, vediamo l'esempio seguente:

```
int x = 20;
int* p_x = NULL;
......
......

p_x = new int;
p_x = & x; // Attenzione!! Memory leak!!
*p_x = 20;
......
```

Perchè quando si esegue l'istruzione p_x = new int si sta creando un memory leak? La risposta è semplice. La variabile p_x è stata innanzitutto inizializzata utilizzando l'istruzione p_x = new int.
Tale istruzione ha fatto sì che venissero allocati due byte ed assegnato l'indirizzo del primo di questi byte alla variabile p_x. Quando poi si esegue l'istruzione p_x = & x la variabile p_x viene fatta puntare all'indirizzo in cui è contenuta la variabile x ma in nessun modo viene deallocata la memoria che era stata allocata in precedenza. Questo causa il famoso memory leak.

Bisogna stare molto attenti ai memory leak. Spesso non sono semplici da riconoscere e possono causare problemi ai programmi causando una graduale diminuzione delle risorse di sistema.
Per evitare il memory leak di prima avremmo dovuto scrivere:

```
int x = 20;
int* p_x = NULL;
......
......

p_x = new int;
delete p_x; // Deallocazione della memoria allocata da p_x
p_x = & x; // Corretto!
*p_x = 20;
......
```

Puntatori ad array

I puntatori e gli array sono argomenti strettamente correlati. Ricordiamo, infatti, che il nome di un array è in realtà un valore costante che rappresenta l'indirizzo del suo primo elemento. Per tale motivo, il valore del nome dell'array non può essere modificato da nessuna istruzione.
Date le dichiarazioni:

```
#define MAX_size 20

float f_array[MAX_size];
float* f_array2;
```

Il nome dell'array f_array è una costante il cui valore è l'indirizzo del primo elemento dell'array di 20 float. L'istruzione seguente assegna l'indirizzo del primo elemento dell'array alla variabile puntatore f_array2:

```
f_array2 = f_array;
```

Equivalentemente sarà possibile scrivere:

```
f_array2 = & f_array[0];
```

Puntatori a stringhe

Una costante stringa, come ad esempio "Hello World" viene, come si è già detto, memorizzata come array di caratteri con l'aggiunta di un carattere nullo alla fine della stringa. Poiché un puntatore a char può contenere l'indirizzo di un char, è possibile eseguire la definizione e l'inizializzazione in un'unica istruzione. Ad esempio:

```
char* stringa = "Hello World";
```

definisce la variabile puntatore a char stinga e la inizializza assegnandole l'indirizzo del primo carattere della stringa. Inoltre viene allocata memoria per tutta la stringa stessa. L'istruzione precedente si sarebbe potuta scrivere anche come:

```
char* stringa;
stringa = "Hello Word";
```

Anche in questo caso occorre capire che a stringa è stato assegnato l'indirizzo della stringa e non il suo contenuto (ovvero *stringa che punta alla lettere "H").

Il tipo reference

Il C++ fornisce una forma di chiamata per indirizzo che è persino più semplice da utilizzare rispetto ai puntatori: il tipo *reference*. Allo stesso modo di una variabile puntatore, il tipo reference fa riferimento alla locazione di memoria di un'altra variabile, ma come una comune variabile, non richiede nessun operatore specifico di deindirizzamento. La sintassi della variabile reference è la seguente:

```
int risultato = 6;
int& ref_risultato = risultato;
```

L'esempio precedente definisce la variabile reference ref_risultato e le assegna la variabile rusultato. Adesso è possibile fare riferimento alla stessa locazione in due modi: tramite risultato e tramite rif_risultato. Poiché entrambe le variabili puntano alla stessa locazione di memoria, esse rappresentano dunque la stessa variabile. E' questo il motivo per cui una variabile reference viene anche chiamata variabile alias.
Quindi, ogni assegnamento fatto su ref_risultato si rifletterà anche su risultato e viceversa.
Attenzione. E' sempre necessario inizializzare una variabile di tipo reference. Ad esempio non si sarebbe potuto scrivere semplicemente:

```
int& ref_risultato; // Errore!
```

Il tipo reference, infatti, ha una restrizione che lo distingue dalle variabili puntatore: bisogna sempre definire il valore del tipo refererence al momento della dichiarazione e tale associazione non può essere più modificata durante tutta l'esecuzione del programma.
Si tenga inoltre presente che non è possibile assegnare ad una variabile reference il valore NULL, utilizzato invece per i puntatori.
Il vantaggio, facilmente visibile, del tipo reference rispetto ai puntatori è rappresentato dal fatto che una variabile reference, dopo la sua definizione, va trattata esattamente allo stesso modo di una variabile normale e non necessita degli operatori di indirizzamento e deindirizzamento utilizzati dai puntatori.

La programmazione orientata agli oggetti

Tutto ciò che abbiamo visto finora si è basato sull'utilizzo delle tecniche di programmazione tradizionale e strutturata. In generale, in un programma tradizionale esiste una funzione principale ed una serie di funzioni secondarie richiamate dalla stessa funzione principale. Tale tipo di approccio si chiama top-down, in quanto l'esecuzione va dall'alto verso il basso (ovvero parte dall'inizio della funzione principale e termina alla fine della stessa funzione).
Nella programmazione procedurale il codice e i dati restano sempre distinti. Le funzioni definiscono quello che deve accadere ai dati ma tali due elementi, codice e dati, non diventano mai una cosa sola. Uno degli svantaggi principali della programmazione procedurale è rappresentato dalla manutenzione del programma: spesso, per aggiungere o modificare parti di un programma è necessaria la rielaborazione

di tutto il programma stesso. Questo approccio richiede un enorme quantità di tempo e di risorse che non è certamente un fattore trascurabile.
Un programma orientato agli oggetti funziona in maniera molto diversa. Vi sono, fondamentalmente, tre vantaggi per un programmatore: il primo è la facilità di manutenzione del programma. I programmi risultano più semplici da leggere e da comprendere. Il secondo vantaggio è costituito dalla possibilità di modificare il programma (aggiungendo nuove funzionalità o cancellando operazioni non più necessarie). Per eseguire tali operazioni basta aggiungere o cancellare i relativi oggetti. I nuovi oggetti erediteranno le proprietà degli oggetti da cui derivano; sarà solo necessarioaggiungere o cancellare li elementi differenti.
Il terzo vantaggio è dovuto al fatto che gli oggetti possono essere utilizzati più volte.
In definitiva, la programmazione ad oggetti rappresenta un modo di interpretare i concetti proprio come una serie di oggetti. Utilizzando, dunque gli oggetti, è possibile rappresentare le operazioni che devono essere eseguite e le loro eventuali interazioni.

Concetti base della programmazione ad oggetti

Il concetto che sta alla base della programmazione ad oggetti è quello della **classe**. Una classe C++ rappresenta un tipo di dati astratto che può contenere elementi in stretta relazione tra loro e che condividono gli stessi attributi.
Un oggetto, di conseguenza, è semplicemente un'istanza di una classe.
Vediamo un esempio per chiarire questo concetto:
Consideriamo la **classe animale**. Essa può essere vista come un contenitore generico di dati i quali identificano le caratteristiche (es: il nome, la specie, ecc.) e le azioni (es: mangiare, dormire, ecc.) comuni a tutti gli animali. In particolare le caratteristiche della classe vengono denominate **proprietà** o **attributi**, mentre le azioni sono dette **metodi o funzioni membro**. Una istanza della classe animale è rappresentata, ad esempio, dall'oggetto cane. Il cane è un animale con delle caratteristiche e delle azioni particolari che specificano in modo univoco le proprietà ed i metodi definiti nella classe animale.
Possiamo riassumere i concetti base della programmazione ad oggetti nel seguente modo:

- Possibilità di dichiarare una funzione come funzione membro di una determinata classe, che significa che tale funzione appartiene strettamente a quella classe e può agire solo sui membri di quella classe o su quelle da essa derivate.
- Possibilità di dichiarare i singoli attributi e funzioni membro della classe come:
  private: accessibili solo alle funzioni membro appartenenti alla stessa classe
  protected: accessibili alle funzioni membro appartenenti alla stessa classe e alle classi da questa derivate (vedremo tra poco cosa significa classe derivata).
  public: accessibili da ogni parte del programma entro il campo di validità della classe in oggetto.
- Possibilità di definire in una stessa classe, dati (attributi e/o funzioni membro) con lo stesso identificatore ma con diverso campo di accessibilità (public, protected, public). Tuttavia, si sconsiglia di usare questo approccio per non rendere difficilmente leggibile il programma.
- Possibilità di overloading delle funzioni (si veda la definizione data nei capitoli precedenti).
- **Incapsulamento**. Con tale termine si definisce la possibilità offerta dal C++ di collegare strettamente i dati contenuti in una classe con le funzioni che la manipolano. L'oggetto, è proprio l'entità logica che deriva dall'incapsulamento.
- **Ereditarietà**. E' la possibilità per un oggetto di acquisire le caratteristiche (attributi e funzioni membro) di un altro oggetto.
- **Polimorfismo**. Rappresenta la possibilità di utilizzare uno stesso identificatore per definire dati (attributi) o operazioni (funzioni membro) diverse allo stesso modo di come, per esempio, animali appartenenti ad una stessa classe possono assumere forme diverse in relazioni all'ambiente in cui vivono.

La sintassi e le regole delle classi C++

La definizione di una classe C ++ inizia con la parola riservata class. Di seguito si deve specificare il nome della classe (ovvero il nome del tipo).

```
class tipo
{
public:
tipo var1;
tipo var2;
tipo var3;
funzione membro 1
funzione membro 2

protected:

tipo var4;
tipo var5;
tipo var6;
funzione membro 3;
funzione membro 4;

private:
tipo var7;
tipo var8;
tipo var9;
funzione membro 5;
funzione membro 6;
};
```

E' buona norma di programmazione inserire la definizione di una classe in un file header (i file intestazione, con estensione ".h") anche se non è indispensabile. Tutte le implementazioni dei metodi della classe andranno, invece, inseriti nel file con estensione cpp.
Vediamo un esempio di una semplice classe:

```
// Semplice esempio di una classe C++
class Cliente
```

```
{
public:
char nome[20];
char cognome[20];
char indirizzo[30];
void inserisci_nome( );
void inserisci_cognome( );
void inserisci_indirizzo( );
};
```

Nella classe precedente sia gli attributi che le funzioni membro della classe sono tutti public, ovvero sono accessibili da ogni punto del programma.
Adesso salviamo la classe appena definita in un file chiamata cliente.h e creiamo un nuovo file, che chiamiamo cliente.cpp, in cui andiamo a scrivere l'implementazione dei metodi. Il file cliente.cpp sarà così fatto:

```
#include <iostream.h>
include "cliente.h"

void Cliente::inserisci_nome( )
{
cout << Inserire il nome del dipendente: ";
cin >> nome;
cout << endl;
}

void Cliente::inserisci_cognome( )
{
cout << Inserire il cognome del dipendente: ";
cin >> cognome;
cout << endl;
}

void Cliente::inserisci_indirizzo( )
{
cout << Inserire l' indirizzo del dipendente: ";
cin >> indirizzo;
cin >> get(newline); //elimina il Carriage Return
}


main( )
{
Cliente cliente;
cliente.inserisci_nome( );
cliente.inserisci_cognome( );
cliente.inserisci_indirizzo( );
cout << "Il nome del cliente inserito è:
" << cliente.nome << endl;

cout << "Il cognome del cliente inserito è:
" << cliente.cognome << endl;

cout << "L' indirizzo del cliente inserito è:
" << cliente.indirizzo << endl;
}
```

Avrete certamente notato la particolare sintassi utilizzata nel file Cliente.cpp relativamente alla implementazione delle funzioni membro. Essa segue la regola:

Tipo_restituito nome_classe::nome_metodo(eventuali parametri)

dove l'operatore "::" viene denominato operatore di scope.
Inoltre, come si può vedere facilmente dalla funzione main, ogni volta che si fa riferimento ad un attributo o ad una funzione membro di un oggetto, va utilizzata la sintassi:

```
oggetto.attributo
oggetto.metodo ()
```

Si supponga, adesso di creare lo stesso programma, ma cambiando il main nel seguente modo:

```
main( )
{
Cliente* cliente;
cliente = new Cliente( );
cliente->inserisci_nome( );
cliente->inserisci_cognome( );
cliente->inserisci_indirizzo( );
cout << "Il nome del cliente inserito è:
" << cliente->nome << endl;

cout << "Il cognome del cliente inserito è:
" << cliente->cognome << endl;

cout << "L' indirizzo del cliente inserito è:
```

```
" << cliente->indirizzo << endl;

delete cliente;
}
```

Adesso, come si può notare, nella funzione main si sta definendo un oggetto della classe Cliente servendosi, però, di una variabile puntatore. Quando, si fa riferimento agli attributi o ai metodi di un oggetto che è stato definito tramite una variabile puntatore, la sintassi è diversa:

```
oggetto->attributo
oggetto->metodo( )
```

Quando il programmatore non dovesse specificare nessun modificatore per un metodo o per un attributo all'interno di una classe, questi si intendono automaticamente private.
Ad esempio:

```
class Cliente
{
char nome[20];
char cognome[20];
char indirizzo[30];

public:
void inserisci_nome( );
void inserisci_cognome( );
void inserisci_indirizzo( );
};
```

gli identificatori: nome, cognome ed indirizzo, che si trovano subito dopo la dichiarazione del nome della classe, saranno attributi private e, come tali, utilizzabili soltanto all'interno della classe stessa.
Si faccia attenzione quando si definisce la visibilità dei metodi e degli attributi di una classe. Se, infatti, avessimo scritto:

```
class Cliente
{
char nome[20];
char cognome[20];
char indirizzo[30];
void inserisci_nome( );
void inserisci_cognome( );
void inserisci_indirizzo( );
};
```

la classe Cliente sarebbe stata una classe assolutamente inutilizzabile dall'esterno visto che tutti i suoi dati sono privati (non è stato infatti specificato alcun modificatore di accesso). Questo è chiaramente un errore. Se si provasse ad eseguire il programma precedente con tutti gli attributi private, il compilatore darebbe un errore in quanto non sarebbe in grado di accedere agli attributi e ai metodi della variabile cliente.

Costruttori e distruttori

Abbiamo visto la sintassi che consente di creare una semplice classe C++. Ma le classi hanno anche altre funzionalità che non sono state evidenziate da questa semplice sintassi.
Un costruttore è una funzione membro di una classe. I costruttori sono utili per inizializzare le variabili della classe o per allocare aree di memoria. E' importante sottolineare il fatto che il costruttore di una classe deve sempre avere lo stesso nome della classe in cui è definito.
I costruttori sono molto versatili: possono accettare argomenti e possono essere modificati tramite overloading.
Ogni volta che viene creato un oggetto di una determinata classe, viene sempre eseguito il costruttore di quella classe. Se non viene dichiarato esplicitamente nella dichiarazione della classe, il costruttore viene generato automaticamente dal compilatore (costruttore di default).
Un distruttore è una funzione membro di una classe normalmente utilizzata per restituire al sistema la memoria allocata da un oggettto. Il distruttore, come il costruttore, ha sempre lo stesso nome della classe nella quale è definito ma è preceduto dal carattere tilde (~). In pratica, i distruttori hanno una funzione opposta a quella dei costruttori.
Il distruttore viene richiamato automaticamente quando si applica l'operatore delete ad un puntatore alla classe oppure quando un programma esce dal campo di visibilità di un oggetto della classe. A differenza dei costruttori, i distruttori non possono accettare argomenti e non possono essere modificati tramite overloading.
Infine, anche i distruttori, quando non vengono definiti esplicitamente, vengono creati automaticamente dal compilatore (distruttore di default).
Vediamo un semplice programma che fa uso del costruttore e del distruttore.

```
/*
* Un semplice programma che converte le lire in Euro
* File Conversione.h
*/

class Conversione
{
public:
Conversione();
~Conversione();

long valore_lira;
float valore_euro;

void ottieni_valore();
float converti_lira_in_euro( );
};

/*
* File Conversione.cpp
```

```
*/

#include <iostream.h>
#inclde "Conversione.h"

Conversione( )
{
cout << "Inizio della conversionen";
valore_lira = 0;
// Inizializzazione della variabile valore_lira
valore_euro = 0.0;
// Inizializzazione della variabile valore_euro
}

~Conversione( )
{
cout << "Fine della conversionen";
}

void Conversione::ottieni_valore( )
{
cout << "Inserire il valore in lire: " ;
cin >> valore_lira;
cout << endl;
}

float Conversione:: converti_lira_in_euro( )
{
float risultato;
risultato = ((float) valore_lira) / (float) 1936.27;
return risultato;
}

main( )
{
Conversione conv;
conv.ottieni_valore( );
conv.valore_euro = conv.converti_lira_in_euro( );
cout << conv.valore_lira <<
" in lire, vale " << conv.valore_euro << " Euro."
cout << endl;

return(0);
}
```

Se si prova ad eseguire il programma precedente, si noterà che la prima cosa che viene stampata sullo schermo è:
Inizio della conversione.
Che corrisponde all'istruzione eseguita all'interno del costruttore della classe Conversione. Il costruttore della classe viene invocato non appena si costruisce un'istanza della classe stessa, ovvero quando nel main viene eseguita l'istruzione:

Conversione conv ;

Dentro il costruttore sono contenute due istruzioni di inizializzazione di variabili della classe. E' buona norma inserire tutte le inizializzazioni sempre all'interno del costruttore della classe.
L'ultima stampa sullo schermo sarà:
Fine della conversione.
Che corrisponde alla invocazione del distruttore della classe Conversione. La chiamata del distruttore viene effettuata in modo automatico non appena l'oggetto costruito precedentemente esce dallo scopo del programma (in tal caso proprio alla fine del programma stesso).
Vediamo ora come sia possibile effettuare l'allocazione e la deallocazione di memoria rispettivamente nel costruttore e nel distruttore.
Modifichiamo leggermente l'esempio precedente:

```
/*
* Un semplice programma che converte le lire in Euro
* File Conversione.h
*/

class Conversione
{
public:
Conversione();
~Conversione();

long* valore_lira;
float valore_euro;

void ottieni_valore();
float converti_lira_in_euro( );
};

/*
```

```
* File Conversione.cpp
*/

#include <iostream.h>
#inclde "Conversione.h"

Conversione( )
{
cout << "Inizio della conversionen";
valore_lira = new long (0);
// Inizializzazione e allocazione della variabile valore_lira
valore_euro = 0.0;
// Inizializzazione della variabile valore_euro
}

~Conversione( )
{
cout << "Fine della conversionen";
delete valore_lira;
}

void Conversione::ottieni_valore( )
{
cout << "Inserire il valore in lire: " ;
cin >> *valore_lira;
cout << endl ;
}

float Conversione:: converti_lira_in_euro( )
{
float risultato;
risultato = ((float) *valore_lira) / (float) 1936.27;
return risultato;
}

main( )
{
Conversione conv;
conv.ottieni_valore( );
conv.valore_euro = conv.converti_lira_in_euro( );
cout << *(conv.valore_lira) <<
" in lire, vale " << conv.valore_euro <<
" Euro."
cout << endl;

return(0);
}
```

Adesso, come si può vedere, la variabile long valore_lira è stata definita come variabile puntatore. Per tale ragione è necessario che essa venga allocata per poter essere utilizzata. Tale allocazione, è buona norma eseguirla nel costruttore della classe.
Dopo aver allocato della memoria occorre ricordarsi di restituire al sistema la memoria utilizzata e che non occorre più al programma stesso.
Il distruttore rappresenta il punto ideale per deallocare la memoria dinamica utilizzata per le variabili come valore_lira.
uso del puntatore thisLa parola chiave this identifica un puntatore che fa riferimento alla classe. Non occorre che venga dichiarato poichè la sua dichiarazione è implicita nella classe.

```
nome_classe* this;
// dove nome_classe è il tipo della classe
```

Il puntatore this punta all'oggetto per il quale è stata richiamata la funzione membro. Ecco un esempio:
Data la seguente definizione di classe:

```
class nome_classe
{
char ch;

public:
void setta_char( char k);
char ritorna_char ();
};
```

ecco l'utilizzo del puntatore this all'interno del metodo ritorna_char():

```
void nome_classe::setta_char(char k)
{
chr = k;
}

char nome_classe::ritorna_char( )
{
return this-->chr;
}
```

In questo caso, il puntatore this consente di accedere alla variabile membro chr, una variabile privata della classe. Naturalmente il puntatore this può avere tanti altri utilizzi, ma è importante sapere che esse rappresenta un puntatore alla classe che si sta utilizzando.

Classi Derivate

Come abbiamo detto, una classe derivata può essere considerata un'estensione di una classe oppure una classe che eredita le proprietà e i metodi da un'altra classe. La classe originaria viene denominata classe base mentre la classe derivata viene anche chiamata sottoclasse (o classe figlia).
Fondamentalmente, una classe derivata consente di espandere o personalizzare le funzionalità di una classe base, senza costringere a modificare la classe base stessa.
E' possibile derivare più classi da una singola classe base.
La classe base può essere una qualsiasi classe C++ e tutte le classi derivate ne rifletteranno la descrizione. In genere, la classe derivata aggiunge nuove funzionalità alla classe base. Ad esempio, la classe derivata può modificare i privilegi d'accesso , aggiungere nuove funzioni membro o modificare tramite overloading le funzioni membro esistenti.

La sintassi di una classe derivata

Per descrivere una classe derivata si fa uso della seguente sintassi:

```
class classe-derivata : <specificatore d'accesso> classe base
{
......
......
};
```

Ad esempio:

```
class pesce_rosso : public pesce
{
......
......
};
```

In tal caso, la classe derivata si chiama pesce_rosso. La classe base ha visibilità pubblica e si chiama pesce.
Il meccanismo di ereditarietà, pur essendo abbastanza semplice, richiede una certa attenzione per non cadere in errori perchè dipende dallo standard della classe base. In pratica, gli attibuti ed i membri che vengono ereditati dalla classe base possono cambiare la loro visibilità nella classe figlia, in base allo specificatore d'accesso con il quale si esegue l'ereditarietà stessa. Precisamente:

- Se lo specificatore d'accesso è public:

i **public** restano public
i **protected** restano protected
I private non possono essere trasferiti

- Se lo specificatore d'accesso è private:

i **public** diventano private
i **protected** diventano private
i **private** non possono essere trasferiti
Si ricordi, comunque, che durante l'ereditarietà un accesso può restringersi o restare uguale ma mai ampliarsi.
Vediamo ora un esempio che fa uso di una classe derivata:

```
/*
* Definizione della classe base animale.
* File animale.h
*/
class animale
{
public:
animale( );
~animale( );

protected:
char specie[20];
int eta;
char sesso;

void mangia ( );
void beve ( );

public:
void ottieni_dati ( );
};

/*
* Implementazione del file animale.cpp
*/

#include <iostream.h>
#include <animale.h>

animale ( )
{
strcpy(specie," ");
cout << "Costruttore della classe animalen";
}
```

```
~animale ( )
{
cout << "Distruttore della classe animalen";
}

void animale::mangia( )
{
cout << "Invocato il metodo mangian";
}

void animale::beve ( )
{
cout << "Invocato il metodo beven";
}

void animale::ottieni_dati ( )
{
cout << "Inserire l'eta' dell'animale: ";
cin >> eta;
cout <<endl;

cout << "Inserire il sesso dell'animale (M o F): ";
cin >> sesso;
cout <<endl;
}

/*
* Definizione della classe derivata cane
* File cane.h
*/

#include "animale.h"

class cane : public animale
{
public:
cane( );
~cane ( );
void esegui_azioni ( );
void stampa_dati ( );

private:
void abbaia ( );
};

/*
* Implementazione del file cane.cpp
*/

#include <iostream.h>
#include <string.h>
#include <cane.h>

cane ( )
{
cout << "Costruito un oggetto di tipo canen";
strcpy(specie,"cane");}

~cane( )
{
cout << "Distrutto un oggetto di tipo canen";
}

void cane::abbaia ( )
{
cout << "Invocato il metodo abbaian";
}

void cane::stampa_dati()
{
cout << "La specie dell'animale e': "<< specie << endl;
cout << "L' età dell'animale e': "<< eta << endl;
cout << "Il sesso dell'animale e' :" << sesso << endl;
}

void cane::esegui_azioni()
{
mangia();
```

```
beve();
abbaia();
}

main ( )
{
cane c;
c.ottieni_dati ( );
c.stampa_dati();
c.esegui_azioni();
return (0);
}
```

Si provi a studiare il programma precedente per capire il funzionamento delle classi derivate. Come si può osservare, si è creata una classe base, animale, che contiene alcune informazioni di base (metodi e attributi) comuni a tutti gli animali. Poi, abbiamo costruito una classe figlia, cane, che eredita tutte le informazioni della classe animale (si osservi che sono state dichiarate protected) ed in più implementa un nuovo metodo, abbaia( ), comune soltanto alla specie cane.
Si noti ,infine,anche l'utilizzo della funzione strcpy, implementata nella libreria standard string.h, che serve per copiare il valore di una stringa sorgente ad una destinazione.