Corso Base di Programmazione C++
(I Rudimenti)

Il manuale può essere scaricato e consultato gratuitamente, tuttavia è vietata ogni riproduzione, anche parziale, del contenuto del manuale, se non dietro autorizzazione dell'autore R. Folgieri.

Indice

Il manuale può essere scaricato e consultato gratuitamente, tuttavia è vietata ogni riproduzione, anche parziale, del contenuto del manuale, se non dietro autorizzazione dell'autore R. Folgieri.

Il manuale può essere scaricato e consultato gratuitamente, tuttavia è vietata ogni riproduzione, anche parziale, del contenuto del manuale, se non dietro autorizzazione dell'autore R. Folgieri.

1 Introduzione

Questo breve manuale, non ha certo la pretesa di voler essere un manuale completo di programmazione C++ bensì: di aiutare le persone che si affacciano per la prima volta al mondo della programmazione a comprendere le potenzialità di questo linguaggio (e della programmazione ad oggetti in generale) e di aiutare le persone che hanno già avuto a che fare in passato con il linguaggio C a rispolverare i concetti base.

Come e' facile intuire, il linguaggio C++ è un'estensione del linguaggio C. Il C++ infatti conserva tutte le caratteristiche del C: potenza, flessibilità di gestione dell'interfaccia hardware e software, possibilità di programmazione a basso livello e sinteticità delle espressioni. In più il C++ fonde i costrutti tipici dei linguaggi procedurali standard con quelli della programmazione orientata agli oggetti.

2 La storia del C++

Il linguaggio C++ nasce all'inizio degli anni Ottanta grazie a Bjarne Stroustrup dei laboratori Bell. Originariamente il C++ fu sviluppato per risolvere alcune simulazioni molto rigorose e dipendenti 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, è stato oggetto di varie evoluzioni.

Uno degli scopi principali del C++ è quello di mantenere piena compatibilità con il C conservando 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 (come ad esempio il FORTRAN) al C.

Il miglioramento più significativo del linguaggio C++ è il supporto alla 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

3 La programmazione ad Oggetti

In seguito all'evoluzione del software avvenuta in questo ultimo decennio, la programmazione ad oggetti è ormai lo standard di programmazione. 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 anche perché non è possibile scrivere del codice ad oggetti senza utilizzare i concetti di modularità.

Un **oggetto** è una singola entità che combina sia strutture dati che caratteristiche peculiari: 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 paragonati a dei mattoni che possono essere assemblati e che hanno un alto potenziale di riutilizzo. Uno dei benefici principali di tale filosofia di programmazione è infatti la riusabilità del software.

Il manuale può essere scaricato e consultato gratuitamente, tuttavia è vietata ogni riproduzione, anche parziale, del contenuto del manuale, se non dietro autorizzazione dell'autore R. Folgieri.

Riprenderemo il concetto di programmazione ad oggetti nei capitoli successivi riportando alcuni semplici esempi.

4 Differenze rispetto al C

Di seguito, vengono elencate le principali differenze tra il C ed il C++:

Commenti
In C++ è presente 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. 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 standard (questi valori dipendono dal tipo del parametro).

Gli operatori new e delete
Gli operatori new e delete permettono al programmatore di controllare l' allocazione e la deallocazione della memoria dello heap.

Il manuale può essere scaricato e consultato gratuitamente, tuttavia è vietata ogni riproduzione, anche parziale, del contenuto del manuale, se non dietro autorizzazione dell'autore R. Folgieri.

5 Gli elementi principali di un programma C

Come abbiamo detto in precedenza, Il C++ è un linguaggio derivato dal C. Sarà dunque necessario, per poter muovere i primi passi verso la programmazione in C++, conoscere i concetti ed i componenti del linguaggio C che sono alla base della programmazione in C++.

5.1 Regole fondamentali di un programma C/C++ (e non)

- I programmi elaborano informazioni provenienti da una sorgente di input.
- I programmi manipolano i dati forniti tramite una serie di istruzioni. Queste istruzioni possono appartenere a quattro categorie principali: singole istruzioni, istruzioni condizionali, cicli e funzioni.
- I programmi devono produrre in output un risultato frutto della manipolazione dei dati.
- Il codice deve essere:
    - Modulare
    - Indentato
    - Commentato in maniera esaustiva
    - I nomi scelti per la definizione di variabili, procedure o funzioni deve essere “parlanti”, deve cioè rappresentare in maniera intuitiva la loro funzione. (ex: una funzione che calcola la distanza si chiamerà “Calcolo_distanza” e non “pippo”)

5.2 Come creare e compilare un progetto in C++

Il processo di creazione e compilazione di un progetto in C++ dipende strettamente dal tipo ambiente di sviluppo che si utilizza. Poichè la maggior parte degli utenti utilizza l'ambiente Windows a 32 bit (Windows 95, Windows 98, Windows NT, etc.), illustreremo con un semplice esempio pratico quali sono i passi necessari per la creazione e la successiva compilazione di un progetto C++ utilizzando Microsoft Visual Studio.

5.2.1 Creazione di un progetto:

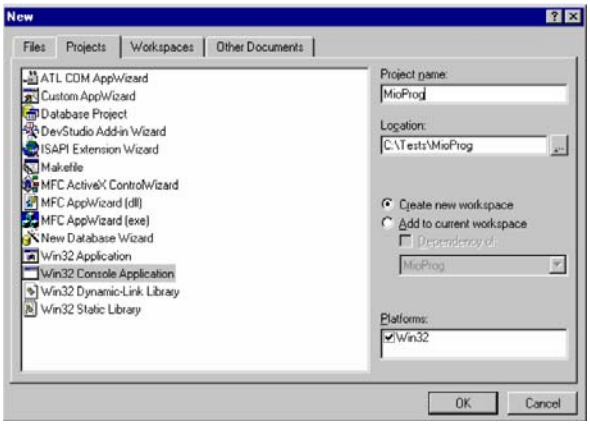
- Lanciare Microsoft Visual Studio
- Dal Menu File selezionare la voce **New**...
- Nella Finestra "New" selezionare (nella lista a sinistra, sotto la scheda **Projects)** la voce **Win32 Console Application**.
- Indicare a destra il nome del progetto e la locazione (directory del disco) in cui salvare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

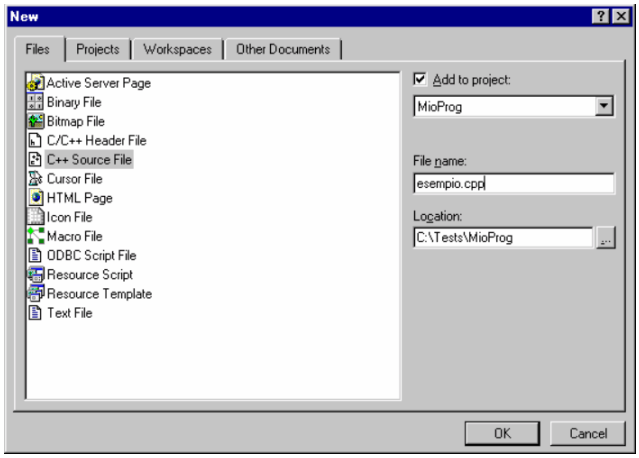
A questo punto il progetto è stato creato.
Occorre ora aggiungere al progetto uno o più files, che conterranno il codice C++, uno dei quali (e soltanto uno) deve contenere la funzione **main**. Si procede così:

- 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.

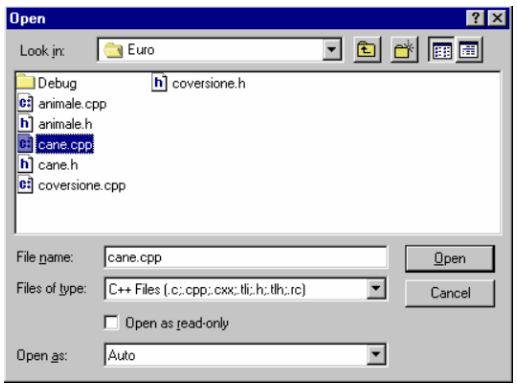
Il manuale può essere scaricato e consultato gratuitamente, tuttavia è vietata ogni riproduzione, anche parziale, del contenuto del manuale, se non dietro autorizzazione dell’autore R. Folgieri.

Riportiamo, per maggiore comprensione la sequenza grafica delle operazioni descritte:



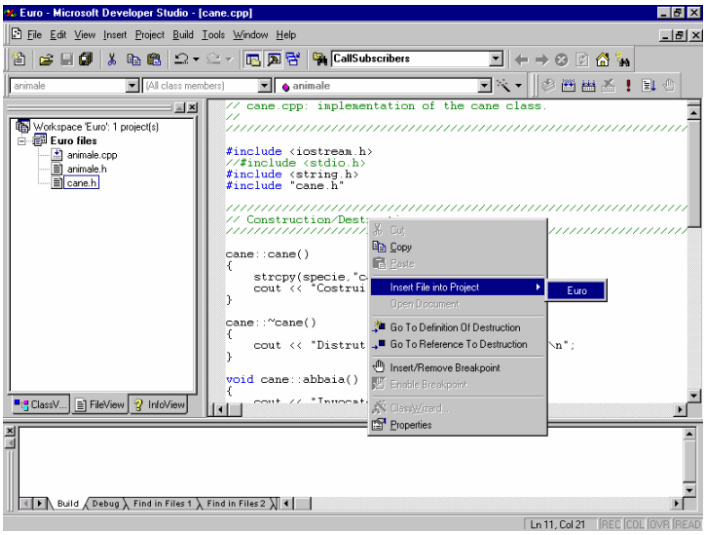


E', altresì, possibile importare dei file .cpp o .h in un progetto esistente. Per eseguire tale operazione, si procede così:



- 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.



Il manuale può essere scaricato e consultato gratuitamente, tuttavia è vietata ogni riproduzione, anche parziale, del contenuto del manuale, se non dietro autorizzazione dell'autore R. Folgieri.

Dopo aver inserito tutti i file necessari al progetto ed il codice desiderato, si deve procedere alla compilazione per poter ottenere un file eseguibile.

Per compilare l'intero progetto selezionare il menu **Build** e quindi la voce **Build MioProg.exe**. Nella finestra dei Messaggi in basso apparirà il risultato della compilazione. Se non si sono verificati 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.

Per poter eseguire il progetto appena compilato, dal menu **Build** selezionare la voce *Execute MioProg.exe*.

I passi indicati nell'esempio, sono relativi alla versione 5.0 di Visual Studio.Volendo utilizzare un ambiente Unix. Invece, si può utilizzare il compilatore **gcc** (Compilatore presente di default in quasi tutti i sistemi Unix).
Per compilare un file **pippo.cpp** contenente un programma sarà sufficiente digitare al prompt dei comandi:

```
gcc  pippo.cpp -o  pippo
```

Naturalmente per compilare programmi di una certa complessità, sarà necessario conoscere in maniera approfondita la sintassi e le varie opzioni di compilazione.

Il seguente programma C++ illustra le componenti fondamentali di un'applicazione C++.

```
//
// prova.CPP
// piccolo esempio in C++
//

#include <iostream.h>

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

Analizziamo quindi passo passo questa piccola porzione di codice:

i **commenti**:

```
//
// Prova.CPP
// piccolo esempio in C++
//
```

I commenti, sebbene non abbiano una parte attiva nel codice del programma (non eseguono infatti alcuna operazione) sono molto importanti, soprattutto in una fase manutentiva dell'applicazione. Chi scrive del codice, potrebbe doverlo manipolare in un secondo tempo, dovendo ricordarsi: cosa ha scritto e perché. Nel caso in cui un programma debba essere modificato da terzi, un commento preciso e puntuale del codice, consente tempi di intervento notevolmente ridotti.

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 consiste nel fatto che il commento a blocchi permette di commentare più linee di codice senza dover ripetere ad ogni linea i simboli del commento singolo.

Il manuale può essere scaricato e consultato gratuitamente, tuttavia è vietata ogni riproduzione, anche parziale, del contenuto del manuale, se non dietro autorizzazione dell'autore R. Folgieri.

Nell'esempio seguente è illustrato un piccolo esempio di commento a blocchi:

```
/*
  Prova.CPP
  primo esempio in C++
*/
```

con l'istruzione:

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

viene indicato al compilatore di includere il codice memorizzato nel file iostream.h all'interno del nostro codice. I files con estensione .h (o file header) includono di norma la definizione di classi, costanti simboliche, identificatori e prototipi di funzione. Analizzeremo di seguito questi argomenti.

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

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

Tutti i programmi C++ devono contenere una funzione chiamata main(). Tale funzione rappresenta il punto di inizio dell'esecuzione del programma e termina con l'istruzione return (0) (istruzione di fine programma).

.
Dopo l'intestazione della funzione main(), si trova il corpo della funzione stessa. Si noti la presenza dei simboli **{** e **}**. Le parentesi graffe indicano blocchi di istruzioni, e importante sottolineare che per ogni parentesi aperta { deve esistere all'interno del codice una parentesi chiusa } che indica la fine del blocco di istruzioni.

La riga:

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

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

Il manuale può essere scaricato e consultato gratuitamente, tuttavia è vietata ogni riproduzione, anche parziale, del contenuto del manuale, se non dietro autorizzazione dell'autore R. Folgieri.

5.3 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. Sebbene gli identificatori possono contenere un qualunque numero di caratteri, 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.

Ecco alcuni esempi di identificatori:

```
j
Mio
mio
first_name
Last_Name
```

E' consigliato utilizzare degli identificatori che abbiano dei nomi parlanti, vale a dire, se un identificatore dovrà contenere l'indirizzo di una persona sarà certamente meglio utilizzare il nome indirizzo piuttosto che il

5.3.1 Le variabili

Una **variabile** è paragonabile ad un contenitore di informazioni omogenee, il cui contenuto (come indicato dal nome stesso) può variare durante l'esecuzione di un programma. Il contenuto di una variabile è modificato da una istruzione di assegnamento (ex: numero= numero+5).

Per poter essere utilizzata all'interno di un programma, una variabile deve essere dichiarata. E' necessario cioè 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++ , come qualsiasi altro linguaggio di programmazione, fornisce una serie di "tipi" standard che permettono al programmatore di definire le variabili a seconda della tipologia dell'informazione da elaborare.

Il manuale può essere scaricato e consultato gratuitamente, tuttavia è vietata ogni riproduzione, anche parziale, del contenuto del manuale, se non dietro autorizzazione dell'autore R. Folgieri.

5.3.2 I tipi standard del C++

I tipi base del **C++** sono sette. E' ovvio che un utente può definire dei propri tipi di strutture e di conseguenza dei propri tipi di variabile.I tipi sono:

- 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**
- puntatori.

- Il tipo char rappresenta una qualsiasi serie di caratteri (b, F, !, ?, 6) e stringhe ("pippo"). Il tipo char occupa 8 bit per carattere. Tale tipo può anche essere anche utilizzato per rappresentare valori numerici compresi nell'intervallo tra -128 e 127.
- I valori interi sono i valori numerici compresi tra -32768 e 32767. Il tipo int occupa 16 bit. Con sistemi operativi a 32 bit (ex: Windows 98 e Windows NT) il tipo int occupa 32 bit e quindi rappresenta 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 composti da una parte intera ed una parte decimale. I numeri in virgola mobile occupano quindi 32 bit e comprendono un intervallo compreso tra, tra 3,4 x 10-38 e 3,4 x 1038 (con 7 cifre significative).
- I valori double in virgola mobile occupano 64 bit e rappresentano valori compresi fra 1,7 x 10-308 e 1,7 x 10308 (con 15 cifre significative). I valori long double consentono precisioni maggiori ed occupano 80 bit. Il loro range è compreso fra 1,18 x 10-4932 e 1,18 x 104932 (con ben 19 cifre significative).
- I tipi enumerativi sono tipi definiti dall'utente.
- Il tipo void indica valori che occupano 0 byte e quindi non hanno alcun valore.
- Il tipo puntatore rappresenta un indirizzo di memoria.

I caratteri Ogni lingua è caratterizzata una serie di caratteri: lettere dell'alfabeto, cifre e segni di punteggiatura. Il tipo char quindi, serve 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 (interpretate come caratteri e non come valori numerici) **0 1 2 3 4 5 6 7 8 9**
Uno dei seguenti simboli: **+ - * / = , . _ : ; ? \ " ' ~ | ! # % $ & ( ) [ ] { } ^ ®**

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

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

#include <stdio.h>

main()
{
        char carattere;
        char num;
         int n;
         n=0 ;
        printf(""Inserire un carattere a scelta e premere
        INVIO: ");
         while (scanf("%c",&carattere)!= “\r”)
         {
```


Il manuale può essere scaricato e consultato gratuitamente, tuttavia è vietata ogni riproduzione, anche parziale, del contenuto del manuale, se non dietro autorizzazione dell'autore R. Folgieri.

```
                printf("Il carattere inserito e': %c\n",carattere);
                n++;
                printf(""Inserire un carattere a scelta e premere
        INVIO: ");
        }
        numero=n;
        printf("Il numero inserito di caratteri e': %d\n",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.
L'opzione **%c** indica che si sta trattando un carattere mentre l'opzione **%d** è utilizzato con gli interi. Quando si scrive: scanf("%c",&carattere) si intende dire che si vuole leggere l'input fornito dalla tastiera dall'utente come carattere (quindi verrà considerato soltanto il primo elemento immesso) e memorizzare tale valore nella variabile carattere.
Similmente, quando si scrive:

```
printf("Il carattere inserito e': %c\n",carattere)
```

si sta stampando sullo schermo la stringa composta dalla concatenazione della stringa: "Il carattere inserito e': " e del carattere contenuto nella variabile carattere . Se si è inserito il carattere 'd' verrà, quindi, stampato:

```
Il carattere inserito e' d.
```

Analogamente l'ultima istruzione del programma:

```
printf("Il numero inserito di caratteri e': %d\n",num);
```

mostrerà il numero di caratteri inseriti tramite tastiera contenuto nella variabile num.

Notiamo come, sia la variabile carattere che la variabile num sono entrambi char. Nel primo caso, la variabile char carattere viene utilizzata per conservare un carattere. Nel secondo caso, la variabile char num viene utilizzata per memorizzare un valore intero, *che però non può essere più grande di 127*.

Dovendo visualizzare a video il contenuto di una variabile float o double , ricordiamo che l'opzione da specificare nel comando di stampa (printf) è %f.

Gli interi

In C++, per memorizzare valori interi, possiamo utilizzare sia il tipo Char che il tipo Int:

```
char intero che occupa un byte (8 bit)

int intero che occupa due byte (16 bit)
```

ognuno di questi tipi, può essere preceduto da tre qualificatori: unsigned, short, long, (senza segno ,corto e lungo) che ne ampliano il range.
Abbiamo detto che un char è rappresentato da un byte. Un byte è, a sua volta, rappresentato da 8 bit.
Nell'esempio seguente, è mostrata la rappresentazione binaria di alcuni numeri (ogni bit che compone un numero può assumere come valori o 0 o 1):

Il manuale può essere scaricato e consultato gratuitamente, tuttavia è vietata ogni riproduzione, anche parziale, del contenuto del manuale, se non dietro autorizzazione dell'autore R. Folgieri.

```
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 +.Normalmente quindi per rappresentare un numero, verranno usati solo 7 byte.

Con 7 bit, il valore maggiore che si riesce a raggiungere e' 128. In questo caso il range sarà –128 +128.

Se viene invece utilizzato il qualificatore **unsigned**, tutti i bit che compongono il byte vengono utilizzati per rappresentare il valore, ovviamente positivo. Il numero relativo all'ultimo byte rappresentato nell'esempio precendente varrà, in tal caso:

1 x 2E0 + 1 x 2E1 + 1 x 2E2 + 1 x 2E3 + 1 x 2E4 + 1 x 2E5 + 1 x 2E6 + 1 x 2E7 = 255

Unsigned int, short int e long int possano essere dichiarati anche più sinteticamente come unsigned, short, long in quanto mancando il tipo, si assume per default che sia int.

Numeri in virgola mobile

Con numeri in virgola mobile si identificano i tipi: float, double e long double

Il range per ciascuno di questi tipi è dato dai valori compresi tra: E-37 e 1E+37.

La maggior parte dei compilatori C comprende i tipi float e double. Successivamente è stato aggiunto un terzo tipo chiamato long double.
Vediamo ora un esempio che illustra la dichiarazione e l'uso delle variabili float.

```
/*
* calcolando dell'area di un cerchio
*/

#include <iostream.h>

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;
}
```


Il manuale può essere scaricato e consultato gratuitamente, tuttavia è vietata ogni riproduzione, anche parziale, del contenuto del manuale, se non dietro autorizzazione dell’autore R. Folgieri.

Enumerazioni

Quando si definisce una variabile di tipo enumerativo, bisogna specificare un insieme di costanti intere chiamato insieme dell'enumerazione. Le variabili possono contenere una qualsiasi delle costanti definite e possono essere utilizzate anche tramite nome. Supponendo di definire un enumeratore nel modo seguente:

```
enum sacco { VUOTO,
       MEZZO_PIENO,
       PIENO = 5} mio_sacco;
```

viene creato il tipo enum sacco. Tutte le costanti e le variabili sono di tipo int e ad ogni costante è assegnato per default un valore iniziale (a meno che non venga specificato in modo esplicito un valore diverso). Nell'esempio precedente, alla costante VUOTO viene assegnato automaticamente il valore int 0 in quanto si trova nella prima posizione e non viene fornito alcun valore specifico. Il valore di MEZZO_PIENO è 1 in quanto si trova immediatamente dopo una costante il cui valore è zero. Alla costante PIENO invece, viene assegnato il valore 5. Se, dopo la costante PIENO venisse introdotta una nuova costante, ad essa sarebbe automaticamente assegnato il valore int 6.

Dopo aver definito il tipo sacco sarà possibile definire un'altra variabile, mio_sacco nel modo seguente:

```
sacco mio_sacco;
```

oppure, in modo del tutto equivalente:

```
enum sacco tuo_sacco;
```

Potremo a questo punto utilizzare i seguenti assegnamenti:

```
mio_sacco = PIENO;
tuo_sacco = VUOTO;
```

che assegnano alla variabile mio_sacco il valore 5 ed alla variabile tuo_sacco il valore 0.

Un errore che spesso si commette è quello di pensare che sacco 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_sacco.
Poiché il nome mio_sacco è una variabile enumerativa di tipo sacco, essa potrà essere utilizzata a sinistra di un operatore di assegnamento. I valori VUOTO, MEZZO_PIENO e PIENO sono nomi di costanti; *non sono variabili e non è possibile in alcun modo cambiarne il valore (il loro valore è definito in fase di dichiarazione dell'enumeratore)*.

Dichiarazione di variabili

Come già detto, il C++ richiede che ogni variabile venga dichiarata prima di essere utilizzata dal programma. La dichiarazione avviene semplicemente indicando il tipo della variabile seguito da uno o più nomi di variabile (nel caso si necessiti dichiarare più variabili dello stesso tipo), separati da una virgola e seguiti dal punto e virgola al termine della dichiarazione.

Ad esempio, per dichiarare le variabili a e b come variabili di tipo float basta scrivere:

```
float a, b;
```

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.

Il manuale può essere scaricato e consultato gratuitamente, tuttavia è vietata ogni riproduzione, anche parziale, del contenuto del manuale, se non dietro autorizzazione dell'autore R. Folgieri.

E' possibile inizializzare una variabile, cioè assegnare alla stessa un valore contemporaneamente alla sua dichiarazione:

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

che corrisponde a scrivere:

```
float a, b;
char ch;

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

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

Conversione di variabili

Come altri linguaggi di programmazione, anche il C++ consente di effettuare certi tipi di operazioni (soprattutto quelle matematiche) tra variabili di tipo diverso (ad esempio tra una variabile dichiarata come intero ed una dichiarata come float). Qualora fosse necessario, è possibile effettuare delle conversioni di formato ricorrendo ad opportune primitive.
Per esempio, se la variabile X è di tipo int viene convertita in float nel seguente modo:

```
(float) X
```

Nell'effettuare le conversioni, è opportuno tener presente l'ordine con cui i vari operatori agiscono sugli operandi. Nell'esempio seguente si può osservare come, l'utilizzo della conversione di formato fornisca risultati differenti :

```
int X = 13, Y = 2;
float Z;

Z = X/Y;           // Si Z = 6.0

Z = (float) X/Y; // X è convertito come float e si ottiene correttamente z = 6.5

Z = (float) (a/b); // (X/Y) viene convertito e si ottiene Z = 6.0
```

Infatti:

- Nel primo caso, pur essendo Z di tipo float, la divisione tra interi non considera i decimali.
- Nel secondo caso il compilatore converte X in un valore float e poi esegue la divisione convertendo di conseguenza anche Y fornendo un risultato è corretto.

1. Nel terzo caso prima viene eseguita la divisione tra interi(che da un risultato intero) poi viene effettuata la conversione ma ormai i decimali sono persi.

Il manuale può essere scaricato e consultato gratuitamente, tuttavia è vietata ogni riproduzione, anche parziale, del contenuto del manuale, se non dietro autorizzazione dell'autore R. Folgieri.

Le costanti

Le COSTANTI, come il nome stesso suggerisce rappresentano , a differenza delle variabili,dei valori che non possono essere cambiati da alcuna codice. Questi valori, vengono definiti all'inizio del programma mediante una delle seguenti modalità:

- Con la sintassi #define
- Definendole di tipo const

La sintassi #**define** è:

```
#define <identificatore> <valore>
```

Specificando il comando #define il compilatore sostituisce alle variabili con quel nome, i valori corrispondenti. Ad esempio, indicando:

```
#include <stdio.h>
#define Contatore1 15
#define contatore2 32
.........
.........
int x,y;
x = Contatore1;
y = Contatore2;

i valori delle variabili X e Y saranno rispettivamente:
x = 10;
y = 2;
```

La sintassi dell'istruzione #define richiede che venga omesso il punto e virgola finale alla riga di dichiarazione e che non venga utilizzato l'operatore di assegnazione =

#define in C++ ha un significato molto più generale che quello di una semplice forma dichiarativa di **costante**. Mediante l'istruzione #define infatti è possibile definire delle macro. Vediamo un esempio pratico di macro:

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

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

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

La sintassi della dichiarazione **const** ha la seguente sintassi:

```
const <identificatore> = <valore>;
```

A differenza dell'istrzione #define, la riga di dichiarazione deve essere terminata col carattere ";" ed il valore

Il manuale può essere scaricato e consultato gratuitamente, tuttavia è vietata ogni riproduzione, anche parziale, del contenuto del manuale, se non dietro autorizzazione dell'autore R. Folgieri.

della costante deve essere assegnato mediante l'operatore di assegnazione =. Se il tipo non viene indicato viene assunto per default che ci si sta riferendo ad un valore intero (int).

```
const int Massimo = 10, Minimo = 2;
const float PIGRECO = 3.1415926;
```

Si noti inoltre che l'utilizzo del tipo Const permette di definire più valori contemporaneamente (è sufficiente che siano separati da una virgola).

5.4 Visibilità delle variabili e delle costanti

Esistono **quattro** regole di visibilità delle variabili determinate dall'area cui le variabili si riferiscono. Le quattro aree 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 è stata dichiarata partendo dal punto del programma in cui è dichiarata. Una variabile dichiarata come **extern** in altri file può essere utilizzata in tutto il programma.

5.5 Le parole riservate

Le **parole riservate** sono identificatori predefiniti che hanno un particolare significato per qualsiasi. Queste parole possono essere utilizzate solo nel modo in cui sono definite (ad esempio, IF è una parola riservata e nessuna variabile di programma può avere quel nome). La seguente **tabella,** riporta l'elenco dielle parole riservate per il C/C++:

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

Il manuale può essere scaricato e consultato gratuitamente, tuttavia è vietata ogni riproduzione, anche parziale, del contenuto del manuale, se non dietro autorizzazione dell'autore R. Folgieri.

6 Gli Operatori

6.1 Operatori booleani

Operatori Booleani: true e false

Gli operatori booleani, sono degli operatori che agiscono su espressioni o variabili di tipo booleano, variabili o espressioni cioè il cui valore è Vero (1 oppure True) o Falso (0 oppure False).
Questi operatori, vengono impiegati in espressioni logiche la cui funzione è quella di stabilire se una o più condizioni sono verificate e quindi bisogna intraprendere una determinata operazione.
In C++, un operatore booleano viene definito di tipo **bool** (altri linguaggi , come ad esempio visual basic indicano tali operatori come di tipo boolean). La sintassi dela dichiarazione è 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 o &&
- Operatore di OR o ||
- Operatore di NOT o !

Ognuno degli **operatori** precedenti accetta come parametri uno o due valori booleani e restituisce in output un valore booleano.

L'**operatore di AND**, prende in input due operandi e restituisce in output un booleano, secondo la regole: se entrambi gli operatori sono true (veri) allora l'output è true; in tutti gli altri casi l'output è uguale a false (falso).

L'**operatore di OR**, prende in input due operandi e restituisce in output un booleano, secondo la seguente regola:Se almeno uno 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**, accetta in input un solo operando e restituisce in output un booleano il cui valore è l'opposto del parametro di input. Se l'operando di input è true allora l'output sarà false e vice versa.

Analizziamo alcuni esempi di utilizzo di operatori booleani.

```
// Supponiamo che Pippo abbia dei soldi
bool PippoHasoldi = true;

// Supponiamo inoltre che pippo voglia acquistare un oggetto
bool PippovuoleAcquistare = true;

// pippo può comperare l'oggetto desiderato?
Bool PippoCompra = PippoHasoldi && PippovuoleAcquistare;
// Il risultato è Vero
```

L'esempio precedente è abbastanza semplice da comprendere. La prima variabile bool (PippoHasoldi) indica che Pippo può materialmente acquistare un oggetto. La seconda variabile bool (PippovuoleAcquistare) è inizializzata a trueindica che Pippo desidera acquistare l'oggetto. La terza variabile booleana (PippoCompra) è un risultato dell'operazione di AND tra le due precedenti. Ovvero: Pippo vuole comperare l'oggetto ed ha i soldi per poterlo fare. Quindi, l'operatore AND è il più adatto in tale circostanza.

Il manuale può essere scaricato e consultato gratuitamente, tuttavia è vietata ogni riproduzione, anche parziale, del contenuto del manuale, se non dietro autorizzazione dell'autore R. Folgieri.

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

```
bool PippoHasoldi = true;
bool PippovuoleAcquistare = true;
bool  PippoCompra = PippoHasoldi || PippovuoleAcquistare;
// Il risultato è true
```

Se, ancora, si verifica la condizione:

```
PippoHaSoldi = true

bool PippoéinBolletta = !PippoHaSoldi
// PippoéinBolletta sarà uguale a false
```

Ovvero la condizione booleana *PippoéinBolletta* sarà vera se non è vera quella che identifica Pippo possiede dei soldi. 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 soldi = 1.000;
int tempo = 0;

bool shopping = soldi && tempo;
```

In questo caso la variabile shopping varrà False (0) in quanto la variabile soldi è uguale a 0.

6.2 Operatori aritmetici

Il linguaggio C++ è dotato di tutti i comuni operatori aritmetici:

- somma (+)
- sottrazione (-)
- moltiplicazione (*)
- divisione intera (/)
- 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 x = 7, y = 16, z = 0, k;

k = y % x;      // restituisce 2
k = x % y;     // restituisce 16

k = y % z;    // restituisce un messaggio
                 di errore (divisione per zero).
```


Il manuale può essere scaricato e consultato gratuitamente, tuttavia è vietata ogni riproduzione, anche parziale, del contenuto del manuale, se non dietro autorizzazione dell'autore R. Folgieri.

6.3 Operatore di assegnamento

L'**operatore di assegnamento** in C++ assegna 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 alla variabile x verrà assegnato il numero 15.

L'utilizzo di tali espressioni è però sconsigliata, non tanto dal punto di vista del risultato, ma per la scarsa leggibilità del codice. Gli unici due casi in cui operazioni di assegnamento vengono utilizzate all'interno di espressioni sono:

L'assegnazione a più variabili di uno stesso valore:

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

All'interno dei cicli while e for:

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

oppure

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

6.4 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 operatore di uguaglianza viene utilizzato per determinare il tipo di azione da intraprendere al verificarsi di certe situazioni. Nella seguente tabella, è riportato l'elenco completo degli operatori di uguaglianza:

Operatore	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

Il manuale può essere scaricato e consultato gratuitamente, tuttavia è vietata ogni riproduzione, anche parziale, del contenuto del manuale, se non dietro autorizzazione dell'autore R. Folgieri.

6.5 Regole di precedenza degli operatori

Le regole di precedenza degli operatori indicano l'ordine con cui gli operatori vengonoeseguiti nell'ambito di una espressione. 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.

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

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
```

Volendo variare questo ordine di interpretazione delle operazioni, è necessario ricorrere all'uso delle arentesi tonde, la cui funzione è di raggruppare le operazioni:

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

in questo caso l'operazione di moltiplicazione verrà effettuata dopo aver eseguito le espressioni contenute nelle due paretesi (che sono diventate gli operandi della moltiplicazione).

La tabella seguente, raggruppa in ordine crescente di priorità gli operatori:.

Operatore	Nome
!	Not
* / %	Moltiplicazione Divisione Modulo
+ -	Addizione Sottrazione
< <= > >=	Minore Minore Uguale Maggiore Maggiore Uguale
== !=	Uguale Diverso
&&	AND
||	OR
=	Assegnamento

Il manuale può essere scaricato e consultato gratuitamente, tuttavia è vietata ogni riproduzione, anche parziale, del contenuto del manuale, se non dietro autorizzazione dell'autore R. Folgieri.

7 Istruzioni Condizionali

7.1 Le istruzioni if e else

Le istruzioni **if** ed **else**, analizzando il significato in Inglese corrispondono a Se ed Allora. Intuitivamente, si utilizza questo costrutto per eseguire un gruppo di istruzioni al verificarsi di determinate condizione.

La sintassi di if è:

```
if (<condizione>)
{
   istruzione 1;
   istruzione 2;
   …
}

mentre , se esistono delle condizioni alternative la sintassi è:


if(<condizione>)
{
   istruzione 1;
   istruzione 2;
   …
}
else
{
   istruzione 3;
   istruzione 4;
   …
}
```

è possibile inoltre ottenere dei costrutti più complessi del tipo:

```
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>);
}
```

esempio:

Il manuale può essere scaricato e consultato gratuitamente, tuttavia è vietata ogni riproduzione, anche parziale, del contenuto del manuale, se non dietro autorizzazione dell'autore R. Folgieri.

```
......
if(i>=0)
{
        <istruzione 1>;
}
else
{
        <istruzione 2>;
}
........
```

L'istruzione 1 verrà eseguita solo se i >= 0, in caso contrario verrà eseguita l'istruzione 2.
I costrutti If..Else, possono ovviamente essere nidificati gli uni negli altri in quanto, ad ogni condizione, possono corrispondere ulteriori sottocondizioni. Ex:

```
if(salario < 200.000.000)
{
if(temperatura < 70.000.000)
{
        cout <<  "Aliquota = 30\%";
}
else
{
        cout << " Aliquota = 45\%";
}
}
```

Nel caso di costrutti nidificati, si ricorda di indentare il codice, in modo che quest'ultimo risulti chiaro e leggibile.

7.2 L' istruzione switch

Come accennato nel paragrafo precedente, può rendersi necessario effettuare determinate operazioni al verificarsi di una o più condizioni. Nel caso le condizioni siano molte, saremmo costretti ad utilizzare delle strutture iIF nidificate, riducendo notevolmente il grado di leggibilità del codice e rischiando di commettere degli errori di sintassi (la più comune è quella di non chiudere correttamente un gruppo di comandi). Per ovviare a questo inconveniente, è possibile ricorrere al costrutto Switch, la cui sintassi è riportata di seguito:

```
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 manuale può essere scaricato e consultato gratuitamente, tuttavia è vietata ogni riproduzione, anche parziale, del contenuto del manuale, se non dietro autorizzazione dell'autore R. Folgieri.

Il funzionamento di tale costrutto è il seguente:

- Viene valutato il valore dell'espressione passata come parametro all'istruzione switch.
- Se il valore passato come parametro, corrisponde ad uno di quelli specificati all'interno della struttura switch (case <valore>), viene eseguito il blocco di istruzioni relativo .
- 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 al parametro passsato all'istruzione switch allora viene eseguito il blocco default, se presente.

Vediamo un paio di esempi concreti:

```
char  colore[10];
switch (mio_colore)
{
case(1):
        colore= “Nero”;
        break;
case(2):
        colore= “Bianco”;
        break;
case(3):
        colore= “Rosso”;
        break;
case(4):
        colore= “Verde”;
        break;
default:
        colore= “Giallo”;
```

Nell'esempio precedente, viene valutato il valore della variabile mio_colore passato come parametro all'istruzione switch. Se tale variabile ha valore uguale a **3**, viene eseguito il blocco di istruzioni corrispondenti al **case (3)**, alla stringa colore quindi viene assegnato il valore “rosso”. Se il valore passato al costrutto fosse invece 7, non esistendo un simile valore all'interno della struttura switch, alla stringa colore verrebbe assegnato il valore di default cioè “giallo”.

7.3 L'istruzione condizionale ?

L'istruzione **condizionale ?** è un modo sintetico di valutare il risultato di un'espressione e può essere utilizzata in sostituzione dell'istruzione IF. La sintassi di un'istruzione condizionale è:

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

Nell'esempio seguente, analizziamo le due alternative possibili di codice:

```
int aliquota;
if(valore >= 60.000.000)
{
```


Il manuale può essere scaricato e consultato gratuitamente, tuttavia è vietata ogni riproduzione, anche parziale, del contenuto del manuale, se non dietro autorizzazione dell'autore R. Folgieri.

```
        aliquota = 33;
}
else
{
        aliquota = 45;
}
```

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

```
int aliquota;
aliquota = (valore>=60.000.000)  ?  45 : 33;
```

Se l'espressione_test (valore >=60.000.000) è vera (azione_true), viene assegnato ad aliquota il valore 45 altrimenti (azione_false) il valore di 33.

7.4 Il ciclo for

La scelta della sintassi da utilizzare, nel caso di istruzioni cicliche, è determinata dal numero di iterazioni che si devono effettuare. La regola è la seguente:

- I cicli **for** vengono utilizzati quando il numero delle iterazioni è un numero finito (anche se elevatissimo)
- i cicli **while** e **do-while** vengono utilizzati quando invece il numero delle iterazioni non è noto a priori.

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* imposta il valore iniziale del contatore del ciclo. La variabile *condizione_di_test* rappresenta la condizione che determina se la condizione di uscita dal ciclo.
La variabile *incremento* determina l'incremento, ad ogni iterazione, del contatore del ciclo.

Ecco un esempio (somma dei numeri che vanno da 1 a 30):

```
int totale = 0;
for (int i=1; i < 30; i++)
{
 totale = totale + i;
}
```

La variabile che funge da contatore del ciclo for (la variabile i) viene inizializzata con il valore 1. Finchè il valore di i è inferiore a 30 (i<30), vengono eseguite le istruzioni contenute tra la parentesi { e } (che indicano il corpo del ciclo FOR). Eseguito il blocco di istruzioni, la variabile i, viene incrementata di una unità (i++) e, se la condizione (i<30) è soddisfatta, il ciclo viene reiterato.

Si prosegue così finchè la variabile i resta minore di 30. In definitiva avremo eseguito l'operazione:
Totale= (1+2+3+4+5+6+7+..30).

Il manuale può essere scaricato e consultato gratuitamente, tuttavia è vietata ogni riproduzione, anche parziale, del contenuto del manuale, se non dietro autorizzazione dell'autore R. Folgieri.

7.5 Il ciclo while

L'istruzione while segue la seguente sintassi:

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

dove *condizione* rappresenta un controllo booleano che viene effettuato ad ogni iterazione.
Tutto ciò che è compreso tra la parentesi { e } viene eseguito finchè la condizione è soddisfatta

Vediamo un esempio. Supponiamo di voler scrivere un programma che stampi la somma di tutti i numeri multipli di 5 compresi nell'intervallo 100 - 250:

```
// File da includere per operazioni

// di input/output cout
#include <iostream.h>

int main()
{
// Definiamo una variabile che conterrà il valore corrente
int contatore = 100;
int totale;
 totale=0;
   while (contatore <= 250)
   {
          contatore = contatore + 5;
          totale=totale + contatore;
   }

cout << "Totale =" << totale << endl;
}
```


Il manuale può essere scaricato e consultato gratuitamente, tuttavia è vietata ogni riproduzione, anche parziale, del contenuto del manuale, se non dietro autorizzazione dell'autore R. Folgieri.

7.6 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 una volta. Il controllo della condizione viene infatti eseguito al termine di ogni ciclo. Se volessimo scrivere lo stesso esempio descritto nel paragrafo precedente otterremmo:

```
// File da includere per operazioni di

//input/output cout
#include <iostream.h>

int main()
{
// Definiamo una variabile che conterrà il valore corrente
int contatore = 100;
int totale= 0;

  do
   {
     contatore = contatore + 5;
     totale = totale + contatore;
    } while (contatore <= 250);

 cout << "Totale = " << totale << 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 contatore fosse stato 260, il programma avrebbe comunque stampato sul video il valore 260 e poi si sarebbe fermato. In tal caso, avremmo ottenuto un risultato diverso da quello desiderato in quanto il numero 260 è certamente maggiore del 250 che rappresenta il limite dell'intervallo da noi definito. Il ciclo while, invece, non avrebbe stampato alcun numero. Tale osservazione va tenuta presente quando si sceglie il tipo di costrutto da utilizzare.

Il manuale può essere scaricato e consultato gratuitamente, tuttavia è vietata ogni riproduzione, anche parziale, del contenuto del manuale, se non dietro autorizzazione dell'autore R. Folgieri.

7.7 L'istruzione break

L'istruzione break viene utilizzata per **forzare** l'uscita da un ciclo prima del tempo (prima cioè che si verifichi la condizione di uscita dal loop). Qualora l'istruzione break venga eseguita, il programma esegue il codice situato immediatamente alla fine del corpo del costrutto in cui è stato inserito (sia che si trovi all'interno di un ciclo for sia che si trovi all'interno di un costrutto switch).

Vediamo un semplice esempio:

```
// Esempio di utilizzo dell'istruzione break
// somma dei quadrati dei primi duecento numeri interi
main()
{
        int k = 1, somma = 0;

        while( k < 200)
{

        if((k*k)> 2000)or ((k*k)< 2300)
{
        break;
}
somma = somma + (k*k);
k++;
}
cout << somma
return (0);
}
```

Questa serie di istruzioni, ad ogni iterazione, incrementa il valore di k aggiungendone il quadrato alla variabile somma. Con l'istruzione if((k*k)> 2000)or ((k*k)< 2300) break indichiamo che, al verificarsi della condizione, desideriamo uscire dal loop senza attendere il verificarsi della condizione $k < 200$ che è la naturale condizione di uscita del ciclo while . L'istruzione successiva, eseguita dal calcolatore, sarà quella di stampare a video il risultato contenuto nella variabile somma.

7.8 L'istruzione continue

A differenza dell'istruzione break che interrompe prematuramente un ciclo, l'istruzione continue viene utilizzata per ignorare le istruzioni successive e passare ad una iterazione successiva. 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 break
// somma dei quadrati dei primi duecento numeri interi
main()
{
        int k = 1, somma = 0;

        while( k < 200)
{
```


Il manuale può essere scaricato e consultato gratuitamente, tuttavia è vietata ogni riproduzione, anche parziale, del contenuto del manuale, se non dietro autorizzazione dell'autore R. Folgieri.

```
        if((k*k) > 2000)or ((k*k)<2300)
{
        continue;
}
somma = somma + (k*k);
k++;
}
cout << somma
return (0);
}
```

In questo caso, alla variabile somma, non verranno addizionati tutti quei valori di k il cui quadrato è un valore compreso tra 2000 e 2300

7.9 L'istruzione exit

Durante l'esecuzione di un programma, potrebbe verificarsi la necessità di interrompere prematuramente l'elaborazione. In questi casi il C++ mette a disposizione del programmatore l'istruzione **exit()**. Tale istruzione (che in realtà corrisponde ad una funzione), accetta come argomento un valore intero (che viene restituito in output) e termina immediatamente dall'esecuzione del programma. Per convenzione, alla funzione Exit, viene passato per argomento il valore 0 quando si vuol segnalare che l'esecuzione è terminata correttamente, in caso contrario, il parametro passato sarà un qualsiasi altro valore. L'utilizzo di questi valori distinti per la funzione exit è importante per segnalare all'operatore la causa che ha determinato l'interruzione prematura dell'elaborazione.

8 Le funzioni

Finora, negli esempi che abbiamo visto, tutte le istruzioni erano composte da operazioni molto semplici, come assegnamenti, controlli booleani od operazioni aritmetiche.
Le funzioni rappresentano una componente fondamentale della programmazione in quanto permettono di eseguire con una sola linea di codice un insieme di operazioni. Le funzioni servono ad identificare, mediante un nome, una sequenza più o meno complessa di istruzioni.
L'enorme utilità delle funzioni risiede nel fatto che, grazie ad esse, è possibile scrivere il codice che esegue una particolare azione una volta sola e riutilizzarlo tutte le volte che si vuole. Il raggruppamento di sequenze di operazioni ripetitive in funzioni, sta alla base del concetto di programmazione modulare.

La sintassi di dichiarazione di una funzione è il seguente:

```
tipo_restituito nome_funzione(tipo e nome_argoment1, tipo e nome_argoment2, …)
{
    dichiarazione dei dati
     corpo della funzione
     return();
}
```

Il tipo indica quale valore verrà restituito dalla funzione. Il nome della funzione, può essere un nome qualunque scelto per descrivere lo scopo della funzione. Analogamente.anche i parametri , argomento della

Il manuale può essere scaricato e consultato gratuitamente, tuttavia è vietata ogni riproduzione, anche parziale, del contenuto del manuale, se non dietro autorizzazione dell'autore R. Folgieri.

funzione, devono essere dichiarati con il loro tipo. Se una funzione ha bisogno di più argomenti, questi dovranno essere separati da una virgola.

Per una maggiore leggibilità del codice e per una migliore manutenzione del medesimo, le funzioni vengono raggruppate per tipologia in file header (file con estensioni .h) che vengono quindi inclusi all'interno dei programmi (mediante l'istruzione #include). L'inclusione serve a rendere disponibili, le funzioni presenti all'interno del file .h. Un esempio è dato dal file header math.h che contiene tutte le funzioni utilizzate per eseguire calcoli matematici.

8.1 Visibilità di una variabile

Quando si parla di 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, globale o di classe. La visibilità di classe sarà discussa più avanti.

Una variabile locale può essere utilizzata esclusivamente all'interno della funzione in cui è stata dichiarata.

Le variabili con visibilità globale, vengono dichiarate all'esterno delle singole funzioni o classi e possono essere referenziate in qualsiasi punto del codice. Può accadere il caso in cui lo stesso nome di variabile venga usato sia come variabile globale sia all'interno di una funzione e quindi con visibilità locale; in questo caso la precedenza di riferimento 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 globale anche all'interno di una funzione in cui è presente una variabile locale con lo stesso nome. La sintassi dell'operatore è:
::variabile.

8.2 La funzione main

In C++ (come per altri linguaggi di programmazione) è possibile passare degli argomenti ad un programma nel momento in cui esso viene richiamato. Ad esempio:

PROGRAMMA Primo_parametro, Secondo_parametro, Terzo_parametro, Quarto_parametro

In tale esempio vengono passati quattro valori tramite la riga di comando. A differenza delle normali funzioni, Main() ha una sintassi particolare per quanto riguarda la dichiarazione dei suoi argomenti:

- il primo argomento è **argc**, un valore intero che contiene il numero di parametri passati al programma
- Il secondo argomento è un puntatore a stringhe chiamato **argv,** che contiene i valori passati al programma.

Tutti gli argomenti sono stringhe di caratteri e quindi argv è di tipo char* [argc].

Vediamo un semplice programma che vuole come input 2 argomenti e che stampa tali argomenti sul video:

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

main(int argc, char* argv[])
{
int i;
```


Il manuale può essere scaricato e consultato gratuitamente, tuttavia è vietata ogni riproduzione, anche parziale, del contenuto del manuale, se non dietro autorizzazione dell'autore R. Folgieri.

```
// verifica il numero di parametri passati al programma. Se diverso da 2
// viene segnalato un errore
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++)
{
// stampa dei parametri. argv[i] contiene l'i-esimo argomento
printf("Argomento %d è %s\n", i, argv[i]);
}

return (0);

}
```

8.3 L' overloading

L'overloading delle funzioni è una peculiarità del C++ che non è presente in C. Questa funzionalità permette di utilizzare lo stesso nome per una funzione più volte all'interno dello stesso programma, a patto però che gli argomenti alle singole funzioni siano differenti.

Automaticamente il programma eseguirà la funzione appropriata a seconda del tipo di argomenti passati.

Esempio:

```
#include <iostream.h>

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

main()
{
        int a = 3;
        int b = 4;
        int totale;
        float c = 9.9;
        float d  = 19.2;
        float totale1;

totale = somma(a, b);
cout  << Il risultato della somma di interi e'<< totale;
totale1= somma(c, d);
cout  << Il risultato della somma di float e'<< totale1;
return (0)}
```

L'overloading è possibile soltanto se: i tipi degli argomenti delle funzioni ed i tipi delle funzioni sono differenti.

Il manuale può essere scaricato e consultato gratuitamente, tuttavia è vietata ogni riproduzione, anche parziale, del contenuto del manuale, se non dietro autorizzazione dell'autore R. Folgieri.

8.4 Gli Array

A differenza di una variabile, che contiene un singolo valore, un array indica una variabile (o meglio un contenitore) che fa riferimento ad un insieme OMOGENEO di valori/oggetti. Ogni array ha un nome al quale viene associato un indice che individua i singoli elementi dell'array. E' possibile definire array di tipo: caratteri, interi, float, double puntatori ed array (si parla cioè di array multidimensionali).

Le proprietà fondamentali di un array sono:

- 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.

8.4.1 Dichiarazione di un array

Per dichiarare un array (analogamente ad una qualsiasi variabile) in C++ è necessario definire il tipo, seguito da un nome valido e da una coppia di parentesi quadre che racchiudono un'espressione costante rappresentante il dimensionamento massimo dell'array.

```
int array_prova[100];  // Un array di 100 interi
char array_pippo[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_PROVA_MAX 100
#define ARRAY_PIPPO_MAX 20

int array_uno[ARRAY_UNO_MAX];
char array_due[ARRAY_DUE_MAX];
```

Il C++ (e lo stesso C) non effettua nessun controllo sugli indici durante operazioni sugli array, è importante quindi effettuare dei controlli a livello di codice onde evitare di tentare di utilizzare dei valori che vanno al di fuori della dimensione massima dichiarata (ciò comporterebbe un errore a livello di runtime oppure l'accesso ad aree di memoria puramente casuali).

```
int pippo[10];
……
……      // L'array viene inizializzato
int i;
for(i=0; i < 14; i++)
{
        cout << pippo[i] << endl;
```


Il manuale può essere scaricato e consultato gratuitamente, tuttavia è vietata ogni riproduzione, anche parziale, del contenuto del manuale, se non dietro autorizzazione dell'autore R. Folgieri.

```
}
```

Nell'esempio precedente, 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). Ciò che accade quando il programma cercherà di andare a leggere i valori maggiori di 9 è assolutamente casuale. Infatti, il programma potrebbe leggere delle aree di memoria fornendo dei valori casuali oppure, nel peggiore dei casi, potrebbe tentare di assegnare dei valori ad aree di memoria riservate causando il crash del sistema operativo.

Un corretto utilizzo di un array è dato dal seguente esempio:

```
#define MAX 100

int pippo[MAX];
……
……      // L'array viene inizializzato
int i;
for (i=0; i < MAX; i++)
{
        cout << pippo[i] << endl;
}
```

8.4.2 Inizializzazione di un array

Un array può essere inizializzato in due modi:

- Esplicitamente, al momento della creazione, i valori contenuti.
- Dinamicamente, durante l'esecuzione del programma.

```
int  numeri[3] = {12, 0, 4};
char lettere[5] = {'a','k','z','o','t'};
float decimali[2] = {3.123, 4.54};
```

Per inizializzare dinamicamente un array, è necessario invece ricorrere ad un ciclo, durante il quale si inseriscono (un valore per volta) i singoli valori in corrispondenza di un determinato valore di indice. L'accesso ad un elemento di un array avviene indicando il nome dell'array e l'indice corrispondente alla posizione desiderata. Ad esempio, Pippo[2] indicherà il terzo elemento dell'array Pippo e non il secondo poichè la numerazione degli indici, come abbiamo detto, inizia da zero.

```
int numeri_Dispari[10];
int i;

for(i =0; i < 10; i++)
{
        numeri_Dispari[i] = i * 3 ;
}
```


Il manuale può essere scaricato e consultato gratuitamente, tuttavia è vietata ogni riproduzione, anche parziale, del contenuto del manuale, se non dietro autorizzazione dell'autore R. Folgieri.

Il programma precedente non fa altro che assegnare all'array i numeri pari da 0 ad 27.

8.4.3 Cenni sugli Array Multidimensionali

Come abbiamo visto, la dimensione di un array è dato dal numero di indici definito nella dichiarazione dell'array. Gli array che abbiamo visto finora sono monodimensionali e richiedono l'uso di un singolo indice. Se, nella dichiarazione di un array, fossero indicate due coppie di parentesi quadre ([ ] [ ]) avremmo a che fare con un array bidimensionale (matrice). Analogamente, è possibile definire array multidimensionali, incrementando semplicemente il numero di indici utilizzati (l'uso di indici superiore a 3 è solitamente utilizzato nei programmi di calcoli matematici complessi).

```
int Array1[10][10];
char Array2 [10][10][10];
…
```

8.4.4 Passaggio di array a funzioni

Gli array, si comportano esattamente come una qualsiasi variabile e quindi, 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 (si passa l'indirizzo del primo elemento dell'array). 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:

```
/*
*Esempio di 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);
}
```


Il manuale può essere scaricato e consultato gratuitamente, tuttavia è vietata ogni riproduzione, anche parziale, del contenuto del manuale, se non dietro autorizzazione dell'autore R. Folgieri.

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

8.5 Stringhe

Il C++, contrariamente ad altri linguaggi di programmazione non è definito il tipo stringa. Le stringhe quindi vengono assimilate ad un array di caratteri. La differenza tra un array di caratteri ed una stringa, consiste nel fatto che una stringa è un array di caratteri terminato da un carattere speciale '\0' detto appunto "terminatore di stringa". 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];

// assegnamento del valore della stringa3 in fase di dichiarazione
char stringa3[5] = "nave";


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

// immissione da tastiera del valore della stringa stringa2

cin >> stringa2;

cout << "La stringa stringa1 è: " << stringa1 <<  "\n";
cout << "La stringa stringa2 è: " << stringa2  << "\n";
cout << "La stringa stringa3 è: " << stringa3 <<  "\n";

       return(0);
}
```

L'output del programma sarà:

```
palla
pippo   (Supponendo che l'utente abbia inserito la stringa pippo)
nave
```


Il manuale può essere scaricato e consultato gratuitamente, tuttavia è vietata ogni riproduzione, anche parziale, del contenuto del manuale, se non dietro autorizzazione dell'autore R. Folgieri.

8.6 I puntatori

Nonostante possano essere assimilati a delle variabili, i puntatori sono delle variabili particolari. La particolarità consiste nel fatto che i valori contenuti si riferiscono ad indirizzi di memoria. Per poter capire a cosa serve un puntatore, è necessario conoscere come è gestita che la memoria del calcolatore:

- Lo Stack
- L'Heap

Nello stack vengono immagazzinate tutte le variabili definite in un programma. Ad esempio quando si definisce:

```
int i = 0;
```

Il computer riserverà due byte di memoria dello Stack per la variabile i. Utilizzando tali variabili non è possibile in alcun modo deallocare la memoria utilizzata dalla variabile. Questo vincolo, può rappresentare un grosso problema soprattutto quando si manipolano grosse quantità di informazioni in quanto potrebbe accadere che lo stack si riempia e non vi sia più spazio per poter allocare nuove variabili (**stack overflow**).

Per permettere ovviare questo inconveniente, esiste 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 (è opportuno infatti, liberare la memoria occupata quando non la si utilizza più).

Questa gestione tuttavia, è un'operazione molto delicata , il cui scorretto utilizzo può portare al verificarsi di erori incontrollati e difficilmente individuabili.

La manipolazione dell'Heap avviene tramite i puntatori.

8.6.1 Che cos'è una variabile puntatore

Una variabile puntatore, come abbiamo già detto, è una variabile che contiene l'indirizzo di memoria di un'altra variabile. Ad esempio, si supponga di avere una variabile intera chiamata pippo ed un'altra variabile (quella puntatore appunto) chiamata diPippo 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:

```
diPippo = &Pippo;
```

In genere quindi, una variabile che contiene un indirizzo, come ad esempio diPippo, viene chiamata variabile puntatore o, più semplicemente, puntatore.

Supponiamo che la variabile intera Pipposia allocata all'indirizzo di memoria 1200. La variablediPippo, ovvero la variabile puntatore, inizialmente non contiene alcun valore (se non inizializzati opportunamente, i puntatori possono contenere indirizzi di memoria casuali e possono puntare ad aree riservate). Nel momento in cui viene eseguita l'istruzione:

```
DiPippo = &Pippo;
```


Il manuale può essere scaricato e consultato gratuitamente, tuttavia è vietata ogni riproduzione, anche parziale, del contenuto del manuale, se non dietro autorizzazione dell'autore R. Folgieri.

la variabile puntatore conterrà l'indirizzo della variabile Pippo cioè 1200.

Per accedere al contenuto della cella il cui indirizzo è memorizzato in diPippo è sufficiente far precedere alla variabile puntatore il carattere asterisco (*).

Ad esempio, se si prova ad eseguire le due istruzioni seguenti:

```
diPippo = &Pippo;
*diPippo = 20;
```

il valore della cella Pippo sarà 20. Si noti che se diPippo contiene l'indirizzo di Pippo, entrambe le istruzioni che seguono avranno lo stesso effetto, ovvero entrambe memorizzeranno nella variabile Pippo il valore 20:

```
Pippo = 20;
*diPippo = 20;
```

8.6.2 Dichiarazione di variabili puntatore

Anche le variabili puntatore, come tutte le altri variabili, richiedono di essere definite. Per definire un puntatore all'indirizzo di una variabile di tipo intero, la sintassi corretta sarà:

```
int* diPippo
```

Si potrebbe vedere tale dichiarazione come formata da due parti. Il tipo di Pippo:

```
int*
```

e l'identificatore della variabile:

```
diPippo
```

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;
float* f;
double* d;
```

E' molto importante ricordare che, se in un programma viene definito un puntatore di un determinato tipo e poi lo si utilizza per puntare ad un oggetto di un altro tipo, si potranno ottenere errori di esecuzione e warning

Il manuale può essere scaricato e consultato gratuitamente, tuttavia è vietata ogni riproduzione, anche parziale, del contenuto del manuale, se non dietro autorizzazione dell’autore R. Folgieri.

in fase di compilazione. Anche se il C++ permette simili operazioni è comunque buona norma di programmazione evitarle.

8.6.3 Inizializzazione di una variabile puntatore

Anche per le variabili puntatore devono essere inizializzate nel momento della loro definizione. Ad esempio:

```
int x;
int* Y = &x;
char c ;
char* Z = &c;
```

Per evitare gli inconvenienti, accennati nei paragrafi precedenti (puntatore che fa riferimento ad aree di memoria casuale), è possibile utilizzare un valore particolare, che indica (come indica il nome stesso) “nessuna area di memoria “. Questo valore è universalmente conosciuto come **NULL** . Ad esempio:

```
int* Y = NULL;
char* Z = NULL;
```

Quando però si inizializza un puntatore a NULL, si deve tener presente che non è possible utilizzare detto puntatore per effettuare alcuna operazione finchè non viene “inizializzato” con un indirizzo valido. Se provassimo a scrivere:

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

otterremmo un errore in esecuzione in quanto la variabile puntatore Y non punta nessuna variabile intera che contenga un valore valido. E’ corretto invece scrivere:

```
int x = 20;
int* Y = NULL;
…….
…….
…….
Y = &x;
cout << *Y << endl;
```

Oltre al tipo NULL, è possibile far ricorso ad un altro metodo per inizializzare un puntatore utilizzando l'operatore **new** (C++).

```
int x = 20;
int* Y = NULL;
……..
…….

Y = new int;
*Y  = x;
```


Il manuale può essere scaricato e consultato gratuitamente, tuttavia è vietata ogni riproduzione, anche parziale, del contenuto del manuale, se non dietro autorizzazione dell’autore R. Folgieri.

Nonostante il risultato sia apparentemente simile a quanto visto negli esempi precedenti, in realtà l'istruzione:

```
 Y = new int
```

alloca una quantità di memoria necessaria per contenere un valore di tipo intero (ovvero 2 byte) ed assegnare l'indirizzo del primo byte di memoria alla variabile Y. Questo indirizzo è 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:

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

per assegnare al puntatore Y il valore 20.

Come già accennato, le variabili puntatore fanno riferimento alla memoria dinamica (heap). Diversamente dalla memoria statica, tutte le variabili puntatore allocate dal programmatore devono essere poi deallocate quando non servono più per evitare inutile occupazione di memoria.

L'istruzione del C++ che serve per distruggere una variabile puntatore è: **delete**. Nel caso dell'esempio precedente, avremmo:

```
int x = 20;
int* Y = NULL;
……..
…….

Y = new int;
*Y  = x;
…….
…….
delete Y;
// La variabile Y,  non essendo più utilizzata, può essere deallocarla
............
............
```

Questa operazione è importante quando si manipolano strutture dati molto grosse che di conseguenza occupano molta memoria. Fate molta attenzione nel caso si abbia la necessità di allocare dinamicamente dei puntatori all'interno di cicli. Qualora la condizione di uscita del ciclo non dovesse essere soddisfatta, si corre il rischio di saturare la memoria della macchina causando crolli del sistema e conseguente perdita di dati.

8.6.4 Puntatori ad array

Il presente paragrafo è stato introdotto per sottolineare una certa analogia tra un puntatore ed il nome di una variabile di tipo array. Il nome di un array infatti è una variabile che fa riferimento (punta) al primo valore contenuto nell'array

Supponendo di aver dichiarato:

```
#define MAX 30
```


Il manuale può essere scaricato e consultato gratuitamente, tuttavia è vietata ogni riproduzione, anche parziale, del contenuto del manuale, se non dietro autorizzazione dell'autore R. Folgieri.

```
int mio_array[MAX];
int* puntatore;
```

È lecito quindi effettuare il seguente assegnamento

```
puntatore = mio_array;
```

o, equivalentemente:

```
puntatore = &mio_array[0];
```

Da questo esempio risulta quindi evidente che è possibile assegnare una stringa ad un puntatore. Una stringa infatti altri non è che un array di caratteri terminato dal carattere speciale “\0”.

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

oppure:

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

Ricordiamo che, a stringa è stato assegnato l'indirizzo della stringa e non il suo contenuto (ovvero *stringa punta alla lettera "H").

8.7 Il tipo reference

Il C++ mette a disposizione del programmatore un tipo particolare, diverso dai puntatori per la modalità di utilizzo, 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 nessuna operazione di allocazione o deallocazione.

La sintassi dichiarativa di una variabile di tipo reference è il seguente:

```
int valore = 45;
int& rif_valore = valore;
```

L'esempio precedente definisce la variabile reference rif_valore e le assegna la variabile valore. Poiché entrambe le variabili puntano allo stesso indirizzo di memoria, ogni assegnamento effettuato a rif_valore interesserà anche la variabile valore e viceversa.

E’ importante sottolineare che:

- Una variabile di tipo Reference deve essere inizializzata. Se infatti nella dichiarazione scrivessimo semplicemente int& ref_risultato; otterremmo un errore.

Il manuale può essere scaricato e consultato gratuitamente, tuttavia è vietata ogni riproduzione, anche parziale, del contenuto del manuale, se non dietro autorizzazione dell’autore R. Folgieri.

- Una variabile di tipo Reference, non può essere inizializzata col valore NULL

9 La programmazione orientata agli oggetti

Quanto esposto nei capitoli precedenti, riguarda un tipo di programmazione di tipo tradizionale. Il corpo del programma cioè è composto da una funzione principale ed una serie di funzioni secondarie richiamate dalla funzione principale (main).

In questo capitolo invece, verranno affrontati i vari aspetti legati alla programmazione ad oggetti.

9.1 Concetti base della programmazione ad oggett

Il concetto che sta alla base della programmazione ad oggetti è quello della **classe**. Una classe 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 dato dall'istanza di una classe.

Vediamo un esempio, rifacendoci al concetto matematico di insieme :

Consideriamo la **classe animale**. Essa può essere vista come un contenitore generico di proprietà che ne identificano le caratteristiche (es: il nome, la specie, ecc.) e le azioni associate ai suoi componenti (es: mangiare, dormire, ecc.). 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. Tale funzione appartiene strettamente a quella classe e può agire solo sugli oggetti appartenenti a quella classe o sulle classi 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, private). Tale approccio, nonostante sia ammesso, è caldamente sconsigliato per una semplice comprensione del codice.
- Possibilità di overloading delle funzioni (si veda la definizione data nei capitoli precedenti).
- **Ereditarietà**. E' la possibilità per un oggetto di acquisire le caratteristiche (attributi e funzioni membro) della classe di appartenenza.
- **Polimorfismo**. Rappresenta la possibilità di utilizzare uno stesso identificatore per definire dati (attributi) o operazioni (funzioni membro) diverse.Tornando all'esempio citato in precedenza, animali appartenenti ad una stessa classe possono assumere aspetto diverso in relazioni all'ambiente in cui vivono

Il manuale può essere scaricato e consultato gratuitamente, tuttavia è vietata ogni riproduzione, anche parziale, del contenuto del manuale, se non dietro autorizzazione dell'autore R. Folgieri.

9.2 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 Anagrafica
{
        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 anagrafica.h e creiamo un nuovo file, che chiamiamo anagrafica.cpp, in cui andiamo a scrivere l'implementazione dei metodi. Il file anagrafica.cpp sarà così fatto:

Il manuale può essere scaricato e consultato gratuitamente, tuttavia è vietata ogni riproduzione, anche parziale, del contenuto del manuale, se non dietro autorizzazione dell'autore R. Folgieri.

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

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

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

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


main( )
{
        anagrafica 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 anagrafica.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:

Il manuale può essere scaricato e consultato gratuitamente, tuttavia è vietata ogni riproduzione, anche parziale, del contenuto del manuale, se non dietro autorizzazione dell'autore R. Folgieri.

```
main( )
{
        anagrafica * cliente;
        cliente = new anagrafica( );
        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;
}
```

L'operazione *cliente = new anagrafica( );* presente nel main, definisce un oggetto della classe *anagrafica* servendosi, però, di una variabile puntatore. Quando, si fa riferimento agli attributi o ai metodi di un oggetto definito tramite una variabile puntatore, la sintassi è diversa:

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

Qualora non si definiscano, nella dichiarazione di classe, i tipi delle funzioni o degli attributi, per default si assume che essi siano di tipo 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.

Il manuale può essere scaricato e consultato gratuitamente, tuttavia è vietata ogni riproduzione, anche parziale, del contenuto del manuale, se non dietro autorizzazione dell'autore R. Folgieri.

Si presti molta 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. 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.

9.3 Costruttori e distruttori

Con costruttore e distruttore, si indicano delle funzioni membro di una classe.

I costruttori sono utilizzati per inizializzare le variabili della classe o per allocare aree di memoria.E' importante ricordare che il costruttore di una classe deve sempre avere lo stesso nome della classe in cui è definito.Creare un oggetto appartenente ad una classe, significa eseguire il costruttore di quella classe. Qualora il costruttore non venisse dichiarato esplicitamente nella classe, esso verrà generato automaticamente dal compilatore (costruttore di default).

I distruttori , al contrario dei costruttori, sono utilizzati per deallocare la memoria di sistema occupata da un oggetto. Il distruttore, come il costruttore, ha sempre lo stesso nome della classe nella quale è definito ma è preceduto dal carattere tilde (~).

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.

Analogamente ai costruttori, anche i distruttori, quando non vengono definiti esplicitamente, vengono generati automaticamente dal compilatore.

Vediamo un semplice programma di esempio:

```
/*
* Convertitore Lire / Euro
* File Conversione.h
*/

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

        long valore_lira;
        float valore_euro;

        void ottieni_valore();
```


Il manuale può essere scaricato e consultato gratuitamente, tuttavia è vietata ogni riproduzione, anche parziale, del contenuto del manuale, se non dietro autorizzazione dell'autore R. Folgieri.

```
        float converti_lira_in_euro( );
};



/*
* File Conversione.cpp
*/

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

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

~Conversione( )
{
cout << "Fine della conversione\n";
}


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.

Il manuale può essere scaricato e consultato gratuitamente, tuttavia è vietata ogni riproduzione, anche parziale, del contenuto del manuale, se non dietro autorizzazione dell'autore R. Folgieri.

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 (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 conversione\n";
        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 conversione\n";
delete valore_lira;
}
```


Il manuale può essere scaricato e consultato gratuitamente, tuttavia è vietata ogni riproduzione, anche parziale, del contenuto del manuale, se non dietro autorizzazione dell'autore R. Folgieri.

```
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);
}
```

Nell'esempio corrente, la variabile valore_lira è stata definita come puntatore, poter essere utilizzata è quindi necessario che essa venga allocata. Così come le inizializzazioni, anche le allocazioni di memoria devono (per buona regola)essere definite all'interno del costruttore della classe.

Dopo aver allocato della memoria, occorre ricordarsi di restituire al sistema la memoria non più utilizzata dal programma. Tutte le deallocazioni di aree di memoria dinamica, devono essere effettuate all'interno del distruttore della classe.

Il manuale può essere scaricato e consultato gratuitamente, tuttavia è vietata ogni riproduzione, anche parziale, del contenuto del manuale, se non dietro autorizzazione dell'autore R. Folgieri.

9.4 Uso del puntatore this

La parola chiave **this** identifica un puntatore che fa riferimento alla classe in oggetto . Non occorre che venga dichiarato poiché la sua dichiarazione è implicita nella classe. Il suo utilizzo è comodo quando, all'interno del codice di definizione di una classe, si vuol far riferimento a variabili della classe stessa.

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

Ad esempio: Data la seguente definizione di classe:

```
class nome_classe
{
        char ch;
        public:
                void imposta_char( char k);
                char ritorna_char ();
};
```

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

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

char nome_classe::ritorna_char( )
{
        return this-->chr;  //uso del puntatore this
}
```

In questo caso, il puntatore this consente di accedere alla variabile chr definita all'interno della classe stessa, (una variabile privata della classe). This, può essere utilizzato solo per far riferimento alla classe ed ai metodi/variabili della classe all'interno della quale è utilizzato.

9.5 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

Il manuale può essere scaricato e consultato gratuitamente, tuttavia è vietata ogni riproduzione, anche parziale, del contenuto del manuale, se non dietro autorizzazione dell'autore R. Folgieri.

può modificare i privilegi d'accesso , aggiungere nuove funzioni membro o modificare tramite overloading le funzioni membro esistenti.

9.5.1 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 causare errori perché dipende dallo standard della classe base. In pratica, gli attributi 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.

In dettaglio avremo:

- **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.

Il manuale può essere scaricato e consultato gratuitamente, tuttavia è vietata ogni riproduzione, anche parziale, del contenuto del manuale, se non dietro autorizzazione dell'autore R. Folgieri.

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 animale\n";
        }

     ~animale ( )
     {
        cout << "Distruttore della classe animale\n";
     }

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

    void animale::beve ( )
    {
        cout << "Invocato il metodo beve\n";
    }
```


Il manuale può essere scaricato e consultato gratuitamente, tuttavia è vietata ogni riproduzione, anche parziale, del contenuto del manuale, se non dietro autorizzazione dell'autore R. Folgieri.

```
    void animale::ottieni_dati ( )
 {
        cout << "Inserire l'eta' dell'animale: ";
        cin >> eta;
        cout <<< cout " F  o M Inserire il sesso dell'animale"
         cin >> sesso;
        cout <

}
…
….


// File cane.h


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









// File cane.cpp


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

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

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

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


void cane::stampa_dati()
```


Il manuale può essere scaricato e consultato gratuitamente, tuttavia è vietata ogni riproduzione, anche parziale, del contenuto del manuale, se non dietro autorizzazione dell'autore R. Folgieri.

```
{
        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

Il manuale può essere scaricato e consultato gratuitamente, tuttavia è vietata ogni riproduzione, anche parziale, del contenuto del manuale, se non dietro autorizzazione dell’autore R. Folgieri.