digit
November 2008

Fast Track to C++

The All new COLOUR Fast Track

YOUR HANDY GUIDE TO EVERYDAY TECHNOLOGY

Fast Track to C++

By Team Digit

Credits

The People Behind This Book

EDITORIAL
Robert Sovereign-Smith Assistant Editor
Santanu Mukherjee, Supratim Bose, Nilay Agambagis, Bhaskar Sur Writers

DESIGN AND LAYOUT
Vijay Padaya, U Ravindranadhan Layout Design
Rohit Chandwaskar Cover Design

CONTENTS

CONTENTS

Object Oriented Programming

Object Oriented Programming (OOP) is a programming concept, involving objects and their interactions to design applications and various computer programs. The highlights included within this programming technique are encapsulation, modularity, polymorphism, and inheritance. This concept was not conventionally used in mainstream software development and predominantly came into practice in the early '90s. These days, most programming languages support OOP. The main reason for this is the need to remove the flaws encountered in the typical procedural approach used in archaic programming through the computing ages.

OOP originates way back to the sixties. Over the years, as the hardware and software evolved, quality was often compromised. Analysts and designers were soon looking for ways to address this problem. OOP focuses on data instead of processes, with programs composed of self-sufficient modules (objects) that contain all the information needed for manipulation.

'Simula' was the first language to introduce OOP to the programming world. The various terminologies it brought were objects, classes, subclasses, virtual methods, co-routines, garbage collection, and discrete event simulation. The language was also used for physical modelling. However, the first language which was labelled as an 'Object Oriented' language was 'Small Talk'.

The highlights of OOP are:
- Emphasis on data rather than the procedure.
- Programs are divided into entities known as objects.

- Data structures are designed to characterize the objects.
- Functions operating on the data of an object are tied together in the data structures.
- Data used is generally hidden and cannot be accessed by external functions.
- Functions help objects to communicate with each other.
- New functions and data can easily be added as per need.
- A bottom up approach is followed during program design.

1.1 Basic Concepts

Before delving into OOP, it is important to be familiar with its concepts. These include:

- Classes
- Objects
- Dynamic Binding
- Message passing

Classes

Classes are used to implement the concept of Abstract Data Types (ADT). A class is a combination of both properties and methods used to manipulate properties. In fact, a class is a blueprint describing the nature of the data structure. For example, consider the case of the class ‘student’. There are some common properties shared by all students, such as name, roll, class, address, and marks. Similarly, there might be some methods used to manipulate these properties. However, the values of these properties can differ depending in the student. If we want to use the class ‘student’, then we need to create instances of this class, also known as objects. Classes are user-defined data types and behave like a built-in programming language.

Objects

Objects are the basic runtime entities in an object oriented system. In fact, it is the instance of a class. An object can be a person, place, bank account or even a table of data that the program needs to handle. Objects may also represent user-defined data such as vectors, time and lists. When a program gets executed, the objects interact by sending suitable messages to one another. For example, any student will be an object in the class 'student'. Similarly, if we consider a fruit to be a class, then mango, apple, and guava will be objects in this class.

Dynamic Binding

The term binding refers to linking a procedure call to the code that needs to be executed as a response to this call. Similarly, 'Dynamic Binding' or late binding refers to the code that remains unknown until the procedure is called during run time. Dynamic binding is also associated with polymorphism and inheritance. For example, let us consider a procedure called 'calculation' declared in a class. Some classes may inherit that class. The definitions (code) associated with the procedure 'calculation' are written such that they perform different operations in each derived class. For example, in one class, the code is written for addition, while in another class for subtraction, and so on and so forth. For objects of different classes, the procedure will provide different results and will be unknown till the execution is complete.

Message Passing

Message passing is the process by which one object sends data to another, or asks the other object to invoke a method. This concept is also known as interfacing in some programming languages. In an object-oriented language, objects communicate with each other by sending and receiving various types of information. The message for an object is related to the request for execution of a procedure, and thus invokes a function in the receiving object, thereby generating the desired result.

Message passing also involves specifying the name of the object, the name of the message and the information to be sent. For example,

```
    Employee. earnings (name);
```

In the above statement, the employee is regarded as the object, earnings as the message, and name as the information.

In addition, the following needs to be mentioned:

- **Data Abstraction and Encapsulation**
- **Inheritance**
- **Polymorphism**

1.2 Data Abstraction And Encapsulation

Data abstraction and Encapsulation are fundamental to OOP. The process of wrapping up data and functions into a single unit is called 'encapsulation'. In other words, encapsulation hides the functional details of a class from objects that send messages to it. Data is not generally accessible to the outside class and only functions wrapped in the class can access it. Encapsulation is also achieved by specifying the particular class using objects of another class.

Abstraction refers to the act of representing essential features and characteristics without detail. Classes use the concept of data abstraction and are known as 'Abstract Data Types'.

1.3 Inheritance

This is another important feature of OOP. By Inheritance, classes can acquire the properties and methods of another class or classes. Inheritance supports hierarchical classification. In other words, the process by which the subclasses inherit the attributes and behaviour of the parent class is termed as 'Inheritance'.

For instance, the class 'Dog' may have sub-classes as Spitz, Alsatian, and Golden Retriever. Consider the class 'Dog' defines a method called `bark()` and a property called `furColor`. Each of its sub-classes (Spitz, Alsatian, and Golden Retriever) will inherit these members. Therefore, the programmer needs to write the code for them just once. Subclasses can also add new members. For instance, take the case of the subclass 'Alsatian'. It can add another method, say, tremble. C++ also supports multiple inheritance, where a subclass inherits properties from more than one ancestral class.

1.4 Polymorphism

A Greek term, Polymorphism is the ability to represent oneself in multiple forms. It helps the programmers to treat derived class members, just like their parent class members. By implementing this concept, one can use an operator to perform different operations depending on the operands used.

For instance, if we consider two numbers, the operation 'addition' will generate a sum. Similarly, if the operands are strings, then the operation will produce a third string by concatenation. This phenomenon of making an operator to exhibit various behaviours in different instances is termed as operator overloading.

It is also possible to use same name for different procedures or methods, but the arguments or return types should be unique for each one of them. Different codes are executed accordingly depending on the arguments or the return type.

Let us assume there are three methods of a class sharing the same name 'sum'. One takes two integers as an argument and returns an integer, the other takes three integers as an argument and returns an integer, while the third method takes two floats as an argument and returns a float value. After creating an object in the class, these functions will be called and the method that matches the arguments and return type is executed. For example, if the method is called using two integers as an argument and returns an integer value, then the first method is executed. Similarly, the other two methods are executed as a result of the corresponding argument and return type. This phenomenon is known as method overriding. The same name can be used for methods in the parent and derived class.

1.5 Applications Of OOP

In terms of benefits, OOP offers various benefits to both the designer and the user of the program. The various benefits of OOP are as follows:

- Through the process of inheritance, redundant codes can be eliminated and the use of classes can be extended.
- Programs can be built from standard working modules that communicate with each other. This saves development time and increases productivity.
- Data hiding helps the programmer to build secure programs that cannot be touched by the code of other components of the program.
- OOP allows multiple instances of an object to co-exist without any interference.
- Mapping objects in the problem domain is also possible by OOP.
- The data-centred design approach in OOP captures more detail of a model.
- It also helps in proper communication between objects by various message passing techniques, simplifying the interface description.

Beginning With C++

2.1 Introduction To C++

During the sixties, the rapid development on computers led to the evolution of several new programming languages. Among all, Algol 60, was developed as an alternative to Fortran. Algol 68 developed during this period, directly influenced the data types used in C. However, being a non specific language, it was not very popular in solving commercial tasks.

In 1963, Combined Programming language (CPL) evolved, and was more efficient in addressing concrete programming tasks as compared to Algol and Fortran. However, this was rather bulkier, and difficult to learn and implement. Four years later, in 1967, Martin Richards developed the Basic Combined Programming Language (BCPL). This was a simplified version of CPL, but was extremely abstract.

In 1970, Ken Thompson started developing UNIX at the Bell Labs and created B. It proved to be an effective simplification of CPL. Unfortunately, B too, had limitations. It compiled to a threaded, rather than executable code, thereby generating a slower code during program execution. Therefore, it was inadequate for the development of an operating system. In such an environment, Dennis Ritchie started the development of a B compiler in 1971, which was able to directly generate executable code. The resulting language was named "New B", and finally known as the "C" language.

Ritchie developed the basic structure for C in 1973. Several concepts such as arrays and pointers were incorporated in this new language, without being transformed into a high-level language.

Bjarne Stroustrap, also from Bell Labs, began development work of C++ in 1980, and published the first manual in 1985. The

ANSI committee X3J16 started developing a standard for C++ in 1990, and by 1998, C++ emerged as one of the leading programming languages, and became the preferred language to develop professional applications across all platforms. Currently, C++ development is on full swing, with a new language C++09 being developed. It is expected to be released by the end of 2009, with several new features.

2.2 Applications of C++

C++ is suitable for various programming tasks due to its versatility in handling complex and tedious programs. Various tasks such as developing an editor, a database, communication systems and various real-life application systems can be developed by this language. The reason is as follows:

- It allows you to create various hierarchy-related objects, and helps to develop special object-oriented libraries that can be used by programmers.
- Being an object-oriented language, C++ is able to effectively map real-world problems, while on the other hand, the C part of C++ gives the language the ability to define machine-level details.
- Maintenance and expansion is easy.

2.3 A Simple C++ Program

Now let us deal with a simple C++ program. The program helps to print a certain string on the screen.

Printing a particular string

```
# include <iostream.h> // a header file
int main()

{
cout<<" let us learn a wonderful language";
return 0;
}
```

The output of the program is as follows:

```
let us learn a wonderful language
```

Now let us analyse the program and its statements in detail. The first line, `# include <iostream.h>` is an integral part of the program. `#include` is the directive used in the program and causes the pre-processor to add the contents of the input-output stream file to the program. This directive also contains the declaration of the identifier cout and the operator <<.

'//' is a comment symbol. In C++, comments always starts with a //. They always terminate at the end of the line. The comment following a '//' is generally a single-line comment. int main() is the most important line in the program. Every C++ program must have a `main()` function, and the actual execution of any C++ program starts from this point. This function is called by the system and returns a value to the system if required. In the above example, it will return a value of 0. If there is no value to return, then int can be replaced by void. The parenthesis `()` is used to specify an argument. In the absence of an argument, `()` can remain blank or be replaced with void.

The next line, `cout<<"let us learn a wonderful language";` prints the output on the screen. This line introduces two new features, namely, `cout` and <<. '`cout`' is an identifier, a predefined object that resembles a standard output stream. << is known as insertion operator that sends bytes to an output stream object.

`return 0;` is the function-return statement. This statement returns a variable or value to the process called by the function. In this case, it returns 0 to the system as mentioned above.

Additional C++ Statements

In C++ programming, statements play a major role. Statements are basically program elements that control manipulations of objects, and also their order of manipulation. There can be various types of statements in C++. Some of the most important statements are listed below:

- **Expression Statements:** These evaluate an expression for various side effects, or determine its return value.
- **Null Statements:** These acts as a replacement for certain conditions when a statement is required by the C++ programming syntax, but not requiring any action.
- **Compound Statements:** Widely used in the programming language, compound statements are basically groups of statements enclosed in curly braces ({ }). These statements can be conveniently used whenever we use a single expression.
- **Selection Statements:** Various tests can be performed with the help of these statements. A particular section of code is executed if the test expression evaluated is true. On the other hand, if the test expression evaluated is false, additional sections of code are executed.
- **Iteration Statements:** These statements are among the most important statements used in C++. They perform repeated execution of blocks of code, until a certain termination criterion is met.
- **Jump Statements:** Mainly used for two purposes, either for transferring control to another location to execute a particular function, or returning control from a function.
- **Declaration Statements:** These declare the variables, methods and functions used in the program.

Now let us come to a slightly complex program based on the above statements. Our aim is to add two numbers and determine their average. As expected, we will key-in our inputs through a standard keyboard. The program is as follows:

```
# include <iostream.h>

int main()

{
float x, y;
float sum, ave;

cout<<"Enter two numbers: ";
cin>> x;
cin>> y;
sum = x+y;
ave = sum/2;
cout <<"Sum =" <<sum<<"\n";
cout<<"Average = "<<ave<<"\n";
return 0;
}
```

The output of the program is as follows:

```
 Enter two numbers: 4 6
 Sum = 10
 Average = 5
```

After the `main()` function, two variables (`x` and `y`) are declared. The variables are declared as float, which is nothing but the data type. In the next statement, another set of float type variables '`sum`' and '`ave`' are declared. These two statements can be replaced by the following statement.

```
float x,y,sum,ave;
```

The statement following tells the user to enter two numbers. The `cin` identifier and >> operator (extraction operator that is used to get bytes from input stream class) then accepts two valid numbers via the keyboard. Next the values of the variables `x` any `y` are added and stored in variable '`sum`'. The variable '`ave`' is used to store the result (sum/2), and determines the average of the two numbers. The identifier ‘cout’ and insertion operator << prints the sum and the average of the two numbers.

2.4 An Example Of Class In C++

In C++, classes play a major role, and provide suitable methods for binding data together, along with functions operating on this data. Now let us consider a program involving classes.

```
// classes example

#include <iostream.h>

class Rect {
     int x, y;
  public:
     void setting_values (int,int);
     int area () {return (x*y);}
};

void Rect::setting_values (int a, int b) {
  x = a;
  y = b;
}

int main () {
  Rect rec;
        rec.setting_values (7,8);
        cout << "area: " << rec.area();
        return 0;
             }
```

The output of the program is as follows:

```
area: 56
```

In the above code, the scope resolution operator (: :) is used in the definition of `setting_values()`. The main purpose of this operator is to define a class member from outside the class definition. First a class named `Rect` is declared, with two variables (of type integer) also declared within it. These two variables are `x` and `y`, respectively. The function `area()` has been defined within the definition of the class '`Rect`'. Further, the `setting_values` method has only its prototype declared within the class but is actually defined outside of it.

Further, the scope resolution operator is used for specifying the function which is actually a member of the class '`Rect`' and only its prototype is declared within the class.

The scope resolution operator also specifies the class to which the member being declared belongs. Two parameters, `a` and `b`, of the type `int` within the `setting_values` method are passed. The values are then stored within `x` and **y** in the method declaration part as shown above.

Next we come to the main method or the main function definition part. An object of the type '`Rect`' is then instantiated. This object is named as '`rec`'. The object then accesses the `setting_values` method by the (`.`) operator and two numerical values (`7,8`) are passed. Then the method `area()` is called using the name of object and the (`.`) operator. Multiplication is done based on these two values stored in the variables `x` and `y`. According to the prototype of the method `area()`, it will return an integer value. In this case, the result of multiplication will be returned. The returned value is printed using `cout` and <<.

Basics Of C++

3.1 Program Structure

The following is a program that displays 'Hello World':

```
// This is the basic program in C++

#include <iostream.h>

int main ()
{
   cout << "Hello World!";
   return 0;
}
```

Its output will be:

```
Hello World!
```

We see the text after compiling and executing the program. Our compiler defines the compilation and editing process involved in the program. Besides, they also depend on the version and the interface (with variation in case of a Development Interface) of the compiler. This example includes most of the components in a C++ program.

In order to understand it better, let us take a closer look at it.

```
//this is the basic program in C++
```

The program starts with a double slash, indicating a comment and has no implication on the function or purpose of the program. You can add any comment to your program, but the only criterion is that it should be preceded with a double slash. These are the short notes to the source code meant for programmers for future reference.

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

The second line of our program starts with a hash (#) symbol. Visibly, these are not typical lines with expressions. `#include <iostream.h>` instructs the compiler's pre-processor to hold the `iostream.h` standard library file. In C++, the declarations of the basic standard library of input and output are included in the '`iostream.h`' standard file, which is used by the program in the later sections of the program.

```
int main ()
```

This is the beginning of the main function of the program. This point onwards, execution is independent of source code in the case of C++. That is to say, all C++ programs must begin with a 'main' function. Also, we can see two parentheses after 'main'. This is a function declaration. These parentheses indicate the difference between a function declaration and the other expressions in the source code. You can add a list of parameters within parentheses. The body of the main function follows the parentheses and enclosed in braces ({ }).

```
{
```

The next line in our program begins with a brace, indicating the beginning of the body of the function. The statements mentioned in the body of the main function define the execution process of the function. This opening brace is followed by our next line of code.

```
cout << "Hello World!";
```

`cout` is a statement in C++, and generates visual effects. It defines the standard output stream and is declared in the 'iostream.h' library file. Once executed, this statement displays the string "Hello World". Don't miss out on the semi colon that indicates the end of the statement. Missing out on this semi colon is the most common error committed by programmers.

```
return 0;
```

The main function terminates with the return statement. A return value is usually follows the return statement, in this case the value is 0. The main() function usually has a return value of 0. Missing out on this return statement results in errors and warning messages during compilation.

The closing brace indicates the end of the body of the main function, and the end of the program.

Although this example had each statement on separate lines, in C++, you can also insert a number of statements on a single line of code, separated by semi colons. As an example, consider the following piece of code:

```
int main () {  cout << "Hello World!"; return 0; }
```

You can add some more statements to the earlier example:

```
//this is the basic progam in C++

#include <iostream.h>

int main ()
{
  cout << "Hello World! ";
  cout << "I am Learning Basic Structure of C++";
  return 0;
}
```

The output of this program will be:

```
Hello World! I am Learning Basic Structure of C++
```

As you can see, this program has statements on separate lines. If typed on a single line, it would appear as follows:

```
    int main () {  cout << " Hello World! "; cout <<
" I am Learning Basic Structure "; return 0; }
```

The output will still remain the same.

3.2 Variables

A 'variable' is used to store a declared value that is used during the execution of the program. As a programmer, you must declare a variable before using it. The following is a general form of a declaration:

```
type variable_list;
```

Here, '`type`' is a valid data type or modifier. '`variable_list`' usually includes a single name for an identifier. Multiple identifier names are separated by commas.

To understand it better, take a closer look at the following lines:

```
int i,j,l;
short int si;
unsigned int ui;
double balance, profit, loss;
```

Now consider the following program,

```
// declaration of a variable

#include <iostream.h>
int main ()
{
    // declaring variables:
    int x, y;
    int total;
    // process:
    x = 5;
    y = 2;
    x = x + 1;
    total = x - y;
    // print out the result:
    cout << total;
    // terminate the program:
    return 0;
}
```

The output will be:

```
4
```

There are three ways you can assign a variable:

- Inside a function, i.e. Local variables. (also known as Automatic variables)
- In the definition of a function parameter, i.e. formal parameters.
- Outside all functions, i.e. Global variables.

Local Variable/Automatic Variable

The Local or Automatic variables are those that are usually declared within a function. Local variables lose their significance outside the blocks of functions within which they are declared.

These Local variables gain significance as long as we are in the body of the functions. Similarly, they lose their values once we exit the functions.

Consider the following program:

```
#include<iostream.h>
int main()
{
int a;
a=10;
void show_a(void); // prototype of function
show_a
show_a();
return 0;
}
void show_a(void)
{
cout<<a;
}
```

This program will generate an error on execution. Since the variable 'a' is declared in the main() function, it cannot be accessed by the function show_a() because the scope of the variable 'a' is within the main() function. Therefore, we can use same name for variables in different functions.

Let us look at the example below:

```
void call(void)
{
int A;
A = 20;
}
void cal2(void)
{
int A;
A = -299;
}
```

Here we can see that we have declared the integer variable 'A' twice. It is first declared in the function cal1() and then again in function cal2(). Here, A in cal1() has no relation with that in cal2().

Formal Parameter

We can insert Formal parameters in the function prototype as well as in the function header of the definition. We also assign values to the local variables using the argument while calling a function.

When a function involves arguments, it declares variables that accept the values of the arguments passed while calling the function. These variables are called variables with formal parameters. When they are inserted within a function then they behave as local variables.

Example:

```
   /* Return 2 if x is part of string y; 0 otherwise
*/
   #include<iostream.h>
   int main()
   {
   char *a;
   char b;
   a="jsgcjg";
   b=' s' ;
   int is_in(char *,char);
   cout<<is_in(a, b);
   return 0;
   }
   int is_in(char *y, char x)
   {
   while(*y)
   if(*y==x) return 2;
   else y++;
   return 0;
   }
```

The output of the above program will be:

```
2
```

This program uses the function is_in() that takes two arguments 'a' and 'b'. During the execution of the function, the values for 'a' and 'b' are stored in the variables 'y' and 'x', respectively. Hence, the local variables 'y' and 'x' of the function is_in(), accept the values of the argument while calling the function. The function returns '2' if any character of 'y' is similar to the value of 'x', or else it returns '0'. With the values used here, the function will return '2' and is displayed as usual.

Global Variable

Global variables are created by declaring variables outside a function. Here, all expressions are independent of the blocks of code they are inserted into. Any code can use global variables. These variables can hold the values while executing the program. In the following example, we can see that the variable 'calculate' is declared outside all functions of the program. A global variable is best used when declared at the start of the program.

Example:

```
#include <iostream.h>

int calculate; /* calculate is global */

void cal1(void);
void cal2(void);
int main(void)
{
calculate = 100;
cal1();
return 0;
}
void cal1(void)
{
int temp;
temp = calculate;
cal2();
cout<<"calculate is "<<temp; /* will print 100 */
}
void cal2(void)
{
int calculate;
for(calculate=1; calculate<10; calculate++)
cout<<' .';
}
```

The output of the program will be:

```
.calculate is 100
```

Here, '`calculate`' is declared as a global variable. This means that it can be accessed by any function in the program. '`calculate`' is initialized with 100 in the function `main()`. Next, its value is assigned to the variable '`temp`' in the function `cal1()`. Subsequently, a variable '`calculate`' is declared in the function `cal2()`, which is a local variable for the `cal2()` function.

We should always remember that C++ is a case-sensitive programming language and hence we should always be careful while naming a variable. The variable names are called identifiers. They may be a single letter, multiple letter, or even an underscore or a numeral. Spaces, punctuation mark and other symbols are not regarded as variables. Variable identifiers are significant in C++, as they distinguish one variable from the others. We should also be careful while using upper or lower case. You can name the identifier using any word. However, the following set of words is restricted in C++:

```
   asm, auto, bool, break, case, catch, char, class,
const, const_cast, continue, default, delete, do,
double, dynamic_cast, else, enum, explicit, export,
extern, false, float, for, friend, goto, if, inline,
int, long, mutable, namespace, new, operator, pri-
vate, protected, public, register, reinterpret_cast,
return, short, signed, sizeof, static, static_cast,
struct, switch, template, this, throw, true, try,
typedef, typeid, typename, union, unsigned, using,
virtual, void, volatile, wchar_t, while.
```

In addition, the following words are alternative representatives for some other operators, and hence cannot be used.

```
   and, and_eq, bitand, bitor, compl, not, not_eq,
or, or_eq, xor, xor_eq.
```

3.3 Data Types

There are various data types in the C++ programming language. These are used to hold different types of values. Here are the various types of data and the description of the data type:

Name	**Description**
`Char`	Declares characters or small integers. ASCII characters can be used with this type of data.
`short int (short)`	Declares short integers.
`int`	Declares integers and whole numbers in a program. These numbers may be positive or negative.
`long int (long)`	Declares longer numbers.
`bool`	Declares the Boolean value. Value can be either ‘true’ or ‘false’.
`float`	Declares floating point decimal numbers.
`double`	Declares double precision floating point numbers.
`long double`	Declares long double precision floating point numbers.

3.4 Constants

Constants always have a fixed value in a program. Similar to the case of variables, constants also include various data types. They are as follows:

- **Decimal Notation**
- **Octa Notation**
- **Hexadecimal Notation**
- **String Constant**
- **Back Slash Constant**

Decimal Notation: Only the numbers are represented.

Octa Notation: A number is headed by a zero.

Hexadecimal Notation: A number is headed by the characters 0x.

String Constant: Set of characters enclosed in double quotation mark.

Back Slash Constant: Character constants that are headed by a back slash, and are often referred to as the escape sequence.

Now let us look at the following back slash constants and their meanings:

Code	**Meaning**
`\b`	Backspace
`\f`	Form feed
`\n`	New line
`\r`	Carriage return
`\t`	Horizontal tab

\"	Double quote
\'	Single quote
\0	Null
\\	Backslash
\v	Vertical tab
\a	Alert
\?	Question mark
\N	Octal constant (where N is an octal constant)
\xN	Hexadecimal constant (where N is a hexadecimal constant)

Let us look at the example below:

25 represent decimal

0125 represent octal

0x125 represent hexadecimal

We can define our desired names for the commonly used constants by using the '`#define`' pre-processor directive. Let us look at the syntax of it:

```
#define identifier value
Now let us look at the example below:
#define NI 1.123456789
#define NEW '\n'
```

Here, the '`NI`' and '`NEW`' constants are defined. After defining these constants, we can use these in the remaining code just like any other regular constants.

```
// This is defined constants: determine the lim-
its

#include <iostream.h>

#define NI 1.12345
```

```
   #define NEW '\n'

   int main ()
   {
     double r=5.0;                   // This variable
is created for radius
     double circle;

     circle = 2 * NI * r;
     cout << circle;
     cout << NEW;

     return 0;
   }
```

The output of this program will be as follows:

```
11.1234
```

In the above example, '`#define`' is not a C++ directive. Rather, it is a directive for the pre-processor. Therefore, there is no need to insert a semi colon at the end of it, as it presumes the entire line as a directive.

Besides, C++ has built in constants that we can directly use in our code. These constants have fixed names, so that we can recognize these constants. The minimum value of the short integer is named as `SHRT_MIN` and the maximum value of the short integer is named as `SHRT_MAX`. Consider, the following example:

```
   #include <iostream.h>
   #include <limits.h>
   int main()
    {
    cout << "The minimum signed character: " <<
SCHAR_MIN << "\n";
    cout << "The maximum signed character: " <<
SCHAR_MAX << "\n";
```

```
    cout << "The minimum short integer is: " <<
SHRT_MIN << "\n";
    cout << "The maximum short integer is: " <<
SHRT_MAX << "\n\n";
    return 0;
    }
```

The output of this program will be:

```
The minimum signed character: -128
The maximum signed character: 127
The minimum short integer is: -32768
The maximum short integer is: 32767
```

The constants defined in the climits/limits library are as follows:

```
CHAR_BIT   INT_MAX    LONG_MAX   SCHAR_MAX  SHRT_MAX
CHAR_MAX   INT_MIN    LONG_MIN   SCHAR_MIN  SHRT_MIN
CHAR_MIN   UINT_MAX   ULONG_MAX  UCHAR_MAX  USHRT_MAX
                                            MB_LEN_MAX
```

In the cfloat library, we can get some double precision numbers provided by C++. These are as follows:

```
DBL_DIG          FLT_DIG             LDBL_DIG
DBL_EPSILON      FLT_EPSILON         LDBL_EPSILON
DBL_MANT_DIG     FLT _MANT_DIG       LDBL_MANT_DIG
DBL_MAX          FLT _MAX            LDBL_MAX
DBL_MAX_10_EXP   FLT_MAX_10_EXP      LDBL_MAX_10_EXP
DBL_MAX_EXP      FLT _MAX_EXP        LDBL_MAX_EXP
DBL_MIN          FLT _MIN            LDBL_MIN
DBL_MIN_10_EXP   FLT_MIN_10_EXP      LDBL_MIN_10_EXP
DBL_MIN_EXP      FLT _MIN_EXP        LDBL_MIN_EXP
FLT_RADIX
```

There is also a constant, called `NULL`. We can use this constant to assign a task, to which the pointer will not hold a valid value. The definition of the `NULL` constant is available in the `cstddef` library.

3.5 Operators

Operators help us to operate variables as well as constants. Operators in C++ comprise various symbols. These are categorically divided into various groups.

● **Assignment Operator (=):**
The Assignment Operator is used in any valid expression in C++. Assume 'x' is a variable and we want to assign a value '10' to it. Here, we can use the following Assignment Operator:

```
x=10;
```

In this case, '10' is the integer value. The part on the left of the Assignment Operator is the 'lvalue' or left value and the right part of the Assignment Operator is the 'rvalue' or the right value.

Example:

```
// This is an example of assignment operator
#include <iostream.h>

int main ()
{
  int x, y;  // x:?,  y:?
  x = 10;            // x:10, y:?
  y = 4;             // x:10, y:4
  x = y;             // x:4,  y:4
  y = 7;             // x:4,  y:7

  cout << "x:";
  cout << x;
  cout << " y:";
  cout << y;
  return 0;
}
```

The output of the above program will be:

```
 x:4 y:7
```

In the above program, two variables 'x' and 'y' are declared. Here 'x' is assigned '4' and 'y' is assigned '7'. Here, we have declared x=y using the Assignment operator, hence modification of 'y' affects 'x' variable.

● **Arithmetic operators:**
There are five Arithmetical operators in C++:

Arithmetic Operators	**Function the Operators**
+	addition
-	subtraction
*	multiplication
/	division
%	modulo

● **Compound assignment** :

Symbols that are regarded as Compound assignments and are as follows: `+=, -=, *=, /=, %=, >>=, <<=, &=, ^=, |=.`

Compound assignment is the combination of two operators.

Example:

```
   // This is an example of compound assignment
operators

   #include <iostream.h>

   int main ()
   {
      int x, y=4;
      x = y;
```

```
        x+=3;                  // this compound assignment
is equivalent to x=x+3
        cout << x;
        return 0;
    }
```

The output of this program is as follows:

```
    7
```

● **Increase (++) and decrease(--) Operator:**
The increase operator (++) increases the value stored in a variable, while the decrease operator (--) performs the opposite.

● **Relational and Equality operators:**
While comparing two expressions, Relational and Equality Operators are used. Relational Operators always return a Boolean value (either True or False). The functions of various Relational and Equality Operators are as follows:

Operators	Function of the Operator
==	Equal to
!=	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

● **Logical operators:**

There are three types of Logical Operators: '!', '&&' and 'II'. We use '!' in order to perform the 'NOT' Boolean operation. '&&' (AND) and the 'II' (OR) operators are used to evaluate two expressions in order to get a single result.

'&&' returns 'TRUE' if both expressions are separated. If either of these expressions is false, then the operator returns 'FALSE'.

'||' returns if any of the expressions or both expressions returns true.

Example:

```
   ( (6 == 6) && (4 > 7) )  // This operator can
evaluate to false ( true && false ).
   ( (6 == 6) || (4 > 7) )  // This operator can
evaluate to true ( true || false ).
```

● **Conditional operator**

The symbol (?) is used as the Conditional operator. Look at the syntax of the Conditional Operator:

```
condition? result1: result2
```

This operator can evaluate an expression and return a value if that is true. If the expression is incorrect i.e. If the condition returns true then it yields a different value. In the above syntax, if the condition returns true, `result1` is executed, or else `result2` is executed.

Example:

```
// This is an example of conditional operator

#include <iostream.h>

int main ()
{
  int x,y,z;

  x=2;
  y=7;
  z = (x>y) ? x : y;

  cout << z;

        return 0;
}
```

The output of the above program is as follows:

```
7
```

Three variables of integer type are declared in the first line of the program above. Next, the value ‘2’ is assigned to ‘x’ and ‘7’ is assigned to ‘y’. Further, a conditional operator is used. If the value of ‘x’ is greater than that of ‘y’, then ‘x’ is assigned to ‘z’, else ‘y’ is assigned to ‘z’. Here, the value of ‘x’ is not greater than that of ‘y’. Therefore, the value of ‘y’ (7) is assigned to ‘z’. Finally, the value of ‘z’ is displayed on the screen as seen above.

● **Comma operator**

The comma (,) is regarded as the Comma Operator and is used to separate two expressions.

Example:

```
x = (y=3, y+2);
```

In the above line, ‘x’ and ‘y’ are two variables. Here, y=3 and y+2 are separated by a Comma Operator.

● **Bitwise Operators:**

Symbols that are regarded as the Bitwise Operators are: &, |, ^, ~, <<, >>.

These are used to modify variables that can consider bit patterns and can represent the values stored by them.

Operator	asm equivalent	Description of the Operator
&	AND	Bitwise AND
|	OR	Bitwise Inclusive OR
^	XOR	Bitwise Exclusive OR
~	NOT	Unary complement (bit inversion)
<<	SHL	Shift Left

`>>`	`SHR`	Shift Right

● **Explicit type casting operator:**
We can convert the datum of a given type to another type by using the Explicit type casting operator that accepts a single parameter. This parameter can be either a type or another variable. `sizeof()` is regarded as an Explicit type casting operator.

Example:
```
x = sizeof (char);
```

● **Precedence of operators:**
While writing a complex expression involving multiple operands, we may encounter difficulties in deciding the sequence of operands in the preferred order.

Let us look at the priority order from greatest to lowest:

Operator	Description	Grouping
`::`	scope	Left-to-right
`() [] . -> ++ -- dynamic_cast static_cast reinterpret_cast const_cast typeid`	postfix	Left-to-right
`++ -- ~ ! sizeof new delete`	unary (prefix)	Right-to-left
`* &`	indirection and reference	Right-to-left
`+ -`	unary sign operator	Right-to-left
`(type)`	type casting	Right-to-left
`.* ->*`	pointer-to-member	Left-to-right
`* / %`	multiplicative	Left-to-right
`+ -`	additive	Left-to-right
`<< >>`	shift	Left-to-right
`< > <= >=`	relational	Left-to-right
`== !=`	equality	Left-to-right
`&`	bitwise AND	Left-to-right
`^`	bitwise XOR	Left-to-right
`|`	bitwise OR	Left-to-right
`&&`	logical AND	Left-to-right

`||`	logical OR	Left-to-right
`? :`	conditional	Right-to-left
`= *= /= %= += -= >>= <<=`		
`&= ^= |=`	assignment	Right-to-left
`,`	comma	Left-to-right

3.6 Basic Input/output

By using the standard Input Output library, we can interact with users by displaying messages on the screen. Here, users can provide inputs by typing via a keyboard. C++ uses the concept of 'streams' to perform input and output operations. This is nothing but an object where programmers can either insert to, or extract characters from. In several programs used here, we have seen that the programs start with a header i.e. '`iostream.h`'. The standard C++ library comprises the header file of '`iostream.h`'.

The screen is the standard output of a program and '`cout`' is defined as the C++ stream object that is to be accessed. The programmers use '`cout`' with an insertion operator that is inserted after a double less-than sign.

The keyboard is the standard input device, and is handled by inserting the >> sign on the 'cin' stream. After the operator, a variable is inserted and this variable can store the data that is later extracted from the stream.

Example:

```
   // This is an example of basic input and output
in C++

   #include <iostream.h>
   int main ()
   {
                     int i;
```

```
            cout << "The value that is to be insert-
ed is: ";
            cin >> i;
            cout << "You have inserted this value "
<< i;
            cout << " and the double value is " <<
i*2 << ".\n";
            return 0;
    }
```

The output of this program is as follows:

```
    The value that is to be inserted is: 702
    You have inserted this value 702 and the double
value is 1404.
```

In the above program an integer variable 'i' is created. Next, a message is displayed on the screen. Next the program accepts the input from the user and the value is assigned to the variable 'i'. The fourth line displays a string as well as the value of 'i'. The last line prints a string and the value of 'i' after adding '2' with it.

Control Structures

In previous chapters, we have seen that a program is a set of statements separated by a semi colon. Also, these statements are executed sequentially (one after another) from top to bottom. However, at times, the program needs to be executed either a statement of a block at a time, depending on certain conditions. This is known as control structure. There are two concepts involved here - Branching and Looping.

4.1 Branching

The conditional execution of a statement or a group of statements is known as Branching. For these purpose C++ provides the following two methods.

4.1.1 The if Statement

The 'if' keyword is used to execute a statement or a group of statements with a specified condition.

The syntax for 'if' is as follows

```
if(condition)
  statement;
```

The statement will be executed when the condition is true, For example,

```
if(n>10)
  cout<<"Value of n ="<<n;
```

In the above code, if the value of the variable 'n' is greater than 10, then the value of 'n' will be displayed on the screen.

Enclose the statements within { } if you need to execute them as a group rather than individually. You can also use { } for each statement. The syntax for multiple statements is as follows:

```
if(condition)
{
 statement 1;
 statement 2;
 ---------------;
 statement n;
}
```

Here 'n' represents the number of statements. You can also represent this in the following way:

```
if(n>10)
{
  cout<<"Value of n=";
 cout<<n;
}
```

Sometimes there are two set of statement(s). One set is executed when the given condition is true and the other when the condition is false. Here, the 'else' keyword is used with the following syntax:

```
if(condition)
  statement;
else
  statement;
```

The statement after 'if' is executed when the condition is true. Similarly, the statement after 'else' is executed when the condition is false. For example,

```
if(a>b)
 cout<<"a is greater than b";
```

```
else
 cout<<"b is greater than a";
```

Here, when the value of 'a' is greater than 'b' (the condition is true), it displays "a is greater than b". If not, it displays "b is greater than a". In this case also, { } is required for multiple statements for both the 'if' and 'else' blocks.

The syntax is as follows:

```
if(condition)
{
 statement 1;
 statement 2;
 ---------------;
 statement n;
}
else
{
 statement 1;
 statement 2;
 ---------------;
 statement n;
}
```

Here, 'n' represents the no. of statements. For example,

```
if(a>b)
{
 cout<<"the value of a is greater than b";
 cout<<"the value is="<<a;
}
else
{
 cout<<"the value of b is greater than a";
 cout<<"the value is="<<b;
}
```

In the above example, when ‘a’ is greater than ‘b’, then it executes the statements within { } after the keyword ‘if’. Otherwise, it executes the statements within { } after the ‘else’ keyword.

One can also use the ‘if’ statement within another ‘if’ statement or within an ‘else’ statement. This is known as ‘nested if’ concept. For example, if we want to find out the greatest out of three numbers stored in three variables, the code will be as follows:

```
if(a>b)
{
 if(a>c)
 {
  cout<<"the greatest number is=";
  cout<<a;
 }
else
 {
  cout<<"the greatest number is=";
  cout<<c;
 }
}
else
{
 if(b>c)
 {
  cout<<"the greatest number is=";
  cout<<b;
 }
else
 {
  cout<<"the greatest number is=";
  cout<<c;
 }
}
```

Here, we have three integer variables - a, b and c, where the three numbers are stored. Initially, the code checks whether 'a' is greater than 'b' or not. If 'a' is greater, then it checks whether 'a' is greater 'c' or not. If this condition is true, then the greatest number will be the value of 'a'. Otherwise, the greatest number will be the value of 'c'. If the first 'if' condition returns false, then it checks whether 'b' is greater than 'c' or not, with the help of another 'if' statement.

If this condition is true, then the greatest number will be the value of 'b'. Otherwise, the greatest number will be the value of 'c'. You can combine multiple conditions in a single 'if' statement with the help of logical operators such as '&&' and '||'. Using this concept the previous code can be written as follows:

```
if(a>b && a>b)
 cout<<"The greatest number ="<<a;
if(b>a && b>c)
 cout<<"The greatest number ="<<b;
if(c>a && c>b)
 cout<<"The greatest number ="<<c;
```

Program 1:

The following program will accept any year i.e. an integer value from the user, and checks whether the year is a leap year or not. Go through the program and its explanation carefully.

```
#include<iostream.h>
int main()
{
 int year;
 cout<<"Please enter any year of you choice=>";
 cin>>year;
 if(year%100==0)
  {
  if(year%400==0)
   cout<<"You have entered "<<year<<" and this is
a leap year”;
      else
       cout<<”You have entered “<<year<<” and this is
not a leap year”;
       }
      else
       {
      if(year%4==0)
       cout<<”You have entered “<<year<<” and this is
a leap year”;
      else
       cout<<”You have entered “<<year<<” and this is
not a leap year”;
       }
   }
```

The output of this program is as follows:

```
Please enter any year of your choice => 1998
You have entered 1998 and this is not a leap year
```

Before we proceed with the explanation of the program, we need to know the criterion that decides a leap year. If a year is divisible by 400, then it is a leap year.

In the above program, the year i.e. the integer value is accepted and stored in an integer variable. The ‘`if`’ statement checks whether the value is divisible by 100. If the condition returns true, then the second ‘`if`’ statement checks whether the value is divisible by 400. If this condition is also true, then the given value i.e. the year will be a leap year and a message is displayed with the value entered by the user. If the second ‘`if`’ statement returns false, then the year will not be a leap year and a corresponding message is displayed along with the value of the year to show the result. Now if the first ‘`if`’ statement returns false, i.e. if the value is not divisible by 100, then the control moves to the ‘`else`’ part that checks whether the value is divisible by 400 or not by the second ‘`if`’ statement. If this holds true, then the year is a leap year and displays a corresponding message with the value of the year on the screen. If this is false, then the year is not a leap year. Accordingly, a corresponding message with the value of the year is displayed as before.

4.1.2 The Switch Statement

This is another concept in branching. In a situation with multiple statement(s), one of them is executed depending on the value of the expression, this concept can be used. The syntax of the switch statement is demonstrated below:

```
switch(expression)
{
 case 1:
    statement 1;
    statement 2;
    --------------- ;
    statement n;
 break;

case 2:
    statement 1;
    statement 2;
    --------------- ;
    statement n;
break;

case n:
    statement 1;
    statement 2;
    --------------- ;
    statement n;
 break;

default:
    statement 1;
    statement 2;
    --------------- ;
```

```
      statement n;
   }
```

Here, 'n' is a positive integer. Any case from '1' to 'n' is executed depending on the expression's value. If there is no such value of expression that satisfies any case, then default statement(s) are executed. There is a slight difference in constructing blocks in an 'if' and 'switch' statement. In a 'switch' statement, a label break is used to terminate a particular case instead of { } in the case of an 'if' statement. Let us go through the following program carefully.

```
# include<iostream.h>
 int main()
 {
  int a;
  cout<<"Enter your Choice=>";
  cin>>a;
  switch(a)
  {
   case 1:
          cout<<"You are in block 1";
          cout<<"\nyour choice is "<<a;

  case 2:
          cout<<" You are in block 2";
          cout<<"\nyour choice is "<<a;
                      break;
  case 3:
          cout<<" You are in block 3";
          cout<<"\nyour choice is "<<a;
          break;
  default:
          cout<<"You are in block default";
          cout<<"\nyour choice is "<<a;
 }
}
```

The output of the above program is:

```
 You are in block 1
 Your choice is 1
 You are in block 2
 Your choice is 1
```

In the above program, a break is missing in the first case and the condition satisfies the first case. Therefore, the first case statement is executed. Next, the second case statement is also executed. It then reaches a break and stops execution.

If the user enters 2 or 3, then the output will be either:

```
You are in block 2
Your choice is 2
```

or,

```
You are in block 3
Your choice is 3
```

For any other number the output will be

```
You are in block default
Your choice is x;
```

Here x is the number entered by the user.

We are bound to use constants with case labels (case 1, case 2, etc.). We cannot use variables or ranges with case labels i.e. case n (where n is a variable) or case 1 to 3 or this kinds of case labels are not allowed.

Program 2:

Now we are going to find out the grade of a student on the following conditions:

Marks **Grade**

=100	A
>=80	B
>=60	C
>=40	D
>=20	E
>20	F

Read the following program carefully.

```
# include<iostream.h>
int main()
 {
   int a,b;
   cout<<"Enter your marks=>";
   cin>>a;
   b=a/20;
   switch(b)
   {
  case 5:
           cout<<"You have obtained "<<a<<" marks";
           cout<<"\nYour grade is A";
           break;
  case 4:
           cout<<"You have obtained "<<a<<" marks";
           cout<<"\nYour grade is B";
           break;

  case 3:
           cout<<"You have obtained "<<a<<" marks";
           cout<<"\nYour grade is C";
           break;

  case 2:
           cout<<"You have obtained "<<a<<" marks";
           cout<<"\nYour grade is D";
           break;
```

```
          case 1:
               cout<<"You have obtained "<<a<<" marks";
               cout<<"\nYour grade is E";
               break;


    default:
               cout<<"You have obtained "<<a<<" marks";
               cout<<"\nyour grade is F";

    }
  }
```

In the above program, you need to enter marks obtained by a student, where the total marks is 100. Hence, the entered number should be less than or equal to 100. The marks are accepted and stored in the integer variable 'a'. Assuming the user enters '67'. The output will be as follows:

```
You have obtained 67 marks
Your grade is C
```

Since the entered number is 67, the value of b is '3'. Therefore, the statements for case 3 will be executed to provide the above output. Depending on the value entered by the user, the default statement(s) are executed and provides us the desired result.

4.2 Looping

There are instances where we need to execute a block of statements in a loop as long as a condition holds true. There are two types of loops – exit control loop and entry control loop. In an exit control, the loop condition is checked after execution of the statement(s) while in the looping block. When the condition returns true, the control enters the loop once again. The do-while statement falls under this category. Similarly, in an entry control loop, the condition is checked when the control of execution tries to enter the loop. There are two types of entry control loops — one is the '`while`' statement, and the other is the '`for`' statement.

4.2.1 do-while Statement

The syntax of do-while statement is as follows:

```
do
{
 statement 1;
 statement 2;
 --------------   ;
 statement n;
}  while(condition);
```

Here, `n` is a positive integer. A single statement can also be used within a loop, and the braces are not mandatory. Now let us see how the loop works. In an exit control loop, the body of the loop executes at least once irrespective of the condition. After execution, the condition is checked. If it holds true, then the body of the loop is executed again. This process continues till the condition returns false. For example,

```
#include<iostream.h>
int main()
{
 int roll;
 char name[ 30] ;
 float marks;
 char ch;
  do
 {
   cout<<"Enter your roll no.=>";
   cin>>roll;
        cout<<"Enter your name=>";
   cin>>name;
   cout<<"Enter your marks=>";
   cin>>marks;
   cout<<"\nYour name is "<<name;
   cout<<"\nYour roll no. is "<<roll;
   cout<<"\nYour marks is "<<marks;
```

```
      cout<<"\nWant to enter another record=>";
      cin>>ch;
    }while(ch==' y' );
  }
```

In this program, there are 3 variables - ‘roll’, ‘name’ and ‘marks’ are created to accept the values of roll no, name and marks of a student, respectively. Also, another variable ‘ch’ is created to accept the user’s choice whether he wants to enter another record or not. Initially, the variable ‘ch’ is not given any value. The do-while loop enters into the body of the loop without checking for any condition. The reason — it is an exit control loop. Next, it accepts the roll no, name and address from the user and also displays the entered information as follows:

```
Your name is=><the name entered by user>
Your roll no. is=><the roll no entered by user>
Your marks is=><the marks entered by user>
```

Another line is displayed on the screen to ask the user for another record. Next, the response from the user is checked by the ‘while’ statement. If the condition holds true, then the body of the loop is executed, or else the execution stops. The message displayed is:

```
Want to enter another record=>
```

If the user enters ‘y’, then the condition holds true, and the body of the loop is executed once more. That is, it again accepts and displays the values of roll no., name and marks. This process continues until the user enters any character other than ‘y’ (say ‘n’).

4.2.2 While Statement

This is an entry control loop. The syntax of while is as follows

```
while(condition)
 {
   statement 1;
```

```
    statement 2;
    --------------  ;
    statement n;
}
```

In this kind of loop, first the condition is checked. If the condition holds true, then the body of the loop is executed. After execution, the condition is checked and executes depending on the return value of the condition. This process continues till the condition returns false.

Program 4

Now if we want to display the first 10 integers on screen, the program will be as follows

```
#include<iostream.h>
int main()
{
  int x=1;
  while(x<=10)
   {
   cout<<"\t"<<x;
   x++;
   }
}
```

The output of the above program will be

```
1   2   3   4   5   6   7   8   9   10
```

Initially, an integer variable 'x' is created and initialized by '1'. Next, the condition checks whether the value of 'x' is less than or equal to '10'. The condition obviously holds true and the body of the loop is executed. '1' is displayed on the screen and also the value of 'x' is incremented by 1. Again the condition is checked and the body of the loop is executed. This process continues till the value of 'x' is greater than 10, i.e. 11. When the value of 'x' is 11, then the condition will return false, and the body of the loop is not executed further.

The continue statement

There may be situations where we want to skip the execution of the body of the loop for a particular condition. Yet, we do not want to completely stop the execution of the loop. Therefore, we need to use the 'continue' statement. Normally, the statement(s) within the body of the loop are executed sequentially till the last statement is reached. However, during the execution when the 'continue' statement is found, the remaining statement(s) of the loop are skipped by the compiler. This means that these statement(s) are not executed and control reach the end of the loop and again check the condition of the loop. The syntax given below will demonstrate this concept.

```
while(condition)
{
 statement 1;
 statement 2;
 --------------  ;
 if(condition)
 {
  statement 3;
  ------------  ;
  continue;
 }
 statement 4;
 -------------   ;
}
```

At first the condition of loop is checked. If this condition holds true, then the body of the loop is executed. Next, a condition is applied through the 'if' statement. When this condition holds true, the statement(s) within the 'if' block are executed. After executing some statements, the 'continue' statement is reached and the remaining statements within the loop are skipped. Finally, the condition of the loop is checked as usual.

Program 5

```
#include<iostream.h>
int main()
{
 int x=1;
 while(x<=10)
 {
  if(x==7)
   {
    cout<<"\n\tWe skipped a number\n";
         x=x+1;
    continue;
   }
   cout<<"\t"<<x;
   x=x+1;
 }
}
```

Output of the above program is

```
1  2   3   4   5   6
We skipped a number
8  9   10
```

Initially, the execution will be the same as the earlier program. Numbers from 1 to 6 will be displayed. After that, the value of 'x' will be 7. This is less than 10. Therefore, the control enters into a loop. However, the 'if' statement holds true and enters the 'if' block. Here, the message "We skipped a number" is displayed. Next, the value of 'x' is increased by '1' and reaches the 'continue' statement. Therefore, the remaining statements in the loop are not executed, and the condition for the 'while' statement is checked. Since the value of 'x' is now 8, the body of the loop is executed, thereby displaying the numbers from 8 to 10. The number 7 is not displayed because of the 'continue' statement.

The Break Statement

Sometimes we want to stop the execution of a loop before its end. This is where we use the 'break' statement. When this statement is reached, control leaves the loop and continues to execute the statements after the loop.

Program 6

```
#include<iostream.h>
int main()
{
 int x=1;
 while(x<=10)
 {
  if(x==6)
  {
   cout<<"\nExecution of loop ends here";
   break;
  }
  cout<<x<<"\t";
  x=x+1;
 }
}
```

Output of the above program will be

```
1 2       3       4       5
Execution of loop ends here
```

In this program, an integer variable is created and initialized with '0'. The 'while' loop starts, and the body of the loop is executed if the value of 'x' is less than, or equal to 0. However, when the value of 'x' becomes '6', the 'if' statement written within the loop holds true and its block is executed and displays the message "Execution of loop ends here". Finally, the 'break' statement is reached, and the control leaves the loop without checking the condition.

Program 7

The following program accepts a number from the user. If the user enters an even number, it stops execution.

```
#include<iostream.h>
int main()
{
  int x;
  char check=' y' ;
  while(check==' y' )
  {
    cout<<"Execution will stop if you enter even
number";
    cout<<"\nEnter a number=>";
    cin>>x;
    if(x%2==0)
   {
     cout<<"\nyou have entered an even number";
     check=' n' ;
   }
  }
}
```

Here, the program first creates an integer variable 'x' and a character variable 'check'. The character variable is then initialized by 'y'. If the condition of 'while' holds true, the control enters the loop. Here, it prompts the user to enter a number. If the user enters an odd number, then the 'if' condition holds false. The value of 'check' will remain 'y'. Therefore, the condition of the loop holds true and the body of the loop is executed again. This process continues till the user enters an even number. If the user enters an even number, the 'if' condition holds true and its body is executed which prints a message. The value of check becomes 'n'. Finally, the condition of the loop is checked, and the execution of the program ends once the condition is false.

4.2.3 For Statement

In this case, the condition of the loop is first checked. If it holds true, only then is the body of the loop executed. The syntax of this loop is as follows:

```
   for(initialization;condition;increment or decre-
ment)
    {
    statement 1;
    statement 2;
    --------------  ;
    statement n;
   }
```

Here, ‘n’ is a positive integer. Initialization refers to the initialization of the variable. This counts the number of times the body of the loop is executed. Condition refers to whether the body of the loop is executed or not. If this condition holds true, the body of the loop is executed. Increment or decrement is the part that increases or decreases the value of the variable discussed above. The variable should be declared earlier as other variables will be initialized by some value and then the condition will be checked. If the condition returns true, the body of the loop will be executed. Finally, before rechecking the condition, the value of the variable is increased or decreased as per requirement. The initialization takes place only once. After that, this portion will not be executed regardless the number of times the body of the loop execute.

The following program will display only the first 10 integers. We have done this before using ‘while’ loop, but this will help us to understand how a ‘for’ loop works.

Program 8

```
#include<iostream.h>
int main()
{
 int x;
 for(x=1;x<=10;x++)
  cout<<x<<"\t";
}
```

The output of the above program will be:

```
1 2    3    4    5    6    7    8    9    10
```

Initially, an integer variable 'x' is created. Then, it enters the 'for' loop for the first time. The variable is then initialized by 1. Then the condition is checked. As it holds true, the body of the loop is executed and the value of 'x', i.e. 1 is printed. Next, the value of 'x' is increased by 1. Further, the condition of the loop is checked. If this holds true, the previous process continues. Here, the condition will holds true till the value of 'x' becomes greater than 10, i.e. it becomes 11.

Program 9

This program finds the factorial of a number.

```
#include<iostream.h>
int main()
{
 int x,y,fact;
 cout<<"Enter a number=>";
 cin>>x;
 fact=1;
 for(y=1;y<=x;y++)
  {
  fact=fact*y;
  }
 cout<<"\nFactorial of "<<x<<" is "<<fact;
}
```

In the above program, there are three integer variables 'x', 'y' and 'fact'. The number entered by the user is accepted and stored in the variable 'x'. The program calculates the factorial of the value of 'x'. Assuming the user entered the value '3'. The variable 'fact' is then initialized by '1'. Next, the 'for' loop starts. Here, the variable 'y' is initialized by '1'. As the value of 'y' is now less than the value of 'x', the loop block is executed. Now the value of 'fact' is 1. The value of 'y' is now increased by '1'. The condition is then checked to verify whether it holds true. After execution of the body of the loop, the value of 'fact' will be 2, and 'y' is again increased by '1'. The loop is executed as mentioned above. The value of 'fact' will now be '6', and the value of 'y' will be '4'. This time, the condition holds false, and the statement after the loop is executed. The output on the screen is:

```
Factorial of 3 is 6.
```

This program is executed in the same way, irrespective of the value entered by the user. However, the user must enter a positive integer to obtain the correct answer.

Note:

The above program will provide incorrect results if the value of 'x' or 'fact' goes beyond the range of integers variable.

The 'continue' and 'break' statements can be used in case of all the loops. Since these statements were described in case of 'while' statement, we did not mention them for other loops.

Functions

5.1 Main Function

Functions are vital in programming development. For the sake of convenience and simplicity, a program is divided into smaller units called functions. This is one of the major principles of sequential structural programming. However, the advantage of using functions in a program is to reduce the size of the program. The functions are called at different phases of the program to simplify its execution.

The `main()` function generally returns a value of type `int`. The language defines the `main()` method and matches one of the following prototypes. These are as follows:

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

It is to be noted that functions returning a value should always use the return statement. Hence, whenever we declare and define a function, it should be declared as follows:

```
int main()
{
...... statements ..................
..........statements............

return 0;

}
```

By default, the return functions is of type `int` and is considered optional in the `main()` header of a program. For any function that has a return value, there should be a `return` statement within the definition of the function. Normally this is supposed to be the last statement of the function, else the C++ compiler will generate errors or a warning shall be issued in the absence of a return statement.

5.2 Function Prototyping

This is one of the most important features in C++. The prototype mainly ensures that whenever a function is called, it is used properly with its right parameters. It also describes the function interface to the complier by giving details about the type of arguments and return values. This helps to conveniently call the function rather the entire definition. It is usually defined as a declaration statement in the calling program. The main form of declaring a function prototype is as follows:

```
    Return type name (argument_type1, argument_type2,
...);
```

`Return type` refers to the data type returned by the function. `name` is related to the name of the function, while `argument_type1`, `argument_type2`, etc. refer to parameters passed to the function when it is called.

Consider the following example:

```
float volume (int m, float n, float o);
```

Here, `volume` is the name of the function that returns a float value. Parameters that are passed within this float function include `m` (`int`), n (`float`) and `o` (`float`). The prototype described above is identical to a function definition, except that this is not related to the function body. The parameters that are passed and enumerated don't need to include identifiers, but only the type specifiers are included. In the prototype declaration, it is optional to include a name for each parameter. For instance let us declare a function called Example which accepts two parameters of type `integer`. This is shown below:

```
int Example (int p, int q);
int Example (int, int);
```

Functions help us in structuring various programs in a much more modular way. In simple terms, it can be best described as a group of executable statements that can be executed when called at some point in the program.

The following format best explains this:

```
type name ( parameter1, parameter2, ...) { state-
ments }
```

Here, `type` is the data type specifier that specifies the type of data returned by the function. `name` is the identifier that mainly calls the function. The third and the most important factor is the parameter. Each parameter passed, consists of a data type specifier followed by an identifier. The parameter also allows arguments to pass to that particular function when it is called at a certain phase in the program. Parameters passed must always be separated by commas.

Consider two programs where this concept is implemented.

Program1

```
// function example

#include <iostream.h >
int add(int x, int y)      // a function add is
declared with two variables x and y as parameters
{
  int z;
  z=x+y;
  return (z);
}
int main ()
{
        int m;
        m = add(10,9); // the function add is called in
the main method
        cout << "The result is " <<m;
        return 0;
    }
      Output:    The result is 19
```

Here, in the main function of the above program, first a variable called `m` is declared of the type `int`. The next line of the program calls a function named `add` which is defined above. The result of this function is stored in the variable `m`. Two values `10` and `9` are passed as values within the function that correspond to the `int x` and `int y` parameters declared for function addition. The value of both the arguments that are passed in the function are subsequently copied to the local variables `x` and `y`. Now note in the function declaration, a function called add is declared with two parameters `x` and of the type `int`. Another variable called `z` is then declared of the type `int` which stores the result of the addition.

`return (z);` is one of the main statements of the program. This statement returns the control to that part of the program that called the function and also returns the value passed by it. In this case, since the function `add` is called by the variable `m` in the main function, it will return the control to the statement `m = add(10,9);` and value of `z` will be assigned to `m`. The value of `m` is then displayed.

Program2

```
#include <iostream.h>
int sub(int m, int n)
{
    int k;
    k=m-n;
    return (k);
}
```

```
  int main ()
  {
    int x=5, y=3, v;
    v = sub(7,2);
    cout << "The first result is " << v << '\n';
    cout << "The second result is " << sub(7,2) <<
'\n';
    cout << "The third result is " << sub(x,y) <<
'\n';
    v= 4 + sub(x,y);
    cout << "The fourth result is " <<v << '\n';
    return 0;
  }
```

The output of this program is:

```
The first result is 5
The second result is 5
The third result is 2
The fourth result is 6
```

Here, within the `main()` function, a variable `v` of the type `int` is declared. Another two variables `x` and `y` are declared and initialised with values `5` and `3`, respectively. In the next line, the function `sub` is called by the variable `v`. Two values, `7` and `2`, are passed to this function. Then the function sub is executed. The values `7` and `2` is assigned to the local variables `m` and `n`, respectively, and the result of subtraction is stored in another local variable `k`. Hence, the value of `k` will be `5`. The function `sub` returns the value of `k`. This return value is stored in the variable `v` used to call the function `sub`. Next, `cout` displays the value of `v`. Next, the function `sub` is again called by the same values. However, this time the return value will be displayed directly by `cout` without assigning it to any variable. After that the function `sub` is again called, but this time two variables mentioned above are passed to it. So the function `sub` will be executed with the values of the variables `x` and y and the value returned by the function sub will be 2 in this case is displayed. In the next line the function sub is again called by the variables x and y. This time the return value is stored in the variable v after adding 4 to it. So the value of v will be 6 and is displayed by the next line.

5.3. Call By Reference

Before understanding Call by reference let us discuss what happens when arguments are passed by value. Generally, any argument passed to the function, is passed by value. Therefore, whenever a function is called, along with its parameters a copy of the values of the variables is passed, but not the actual variables. Consider the following example.

```
int m=5, y=3, k;
k= adding (m,y);
```

Here, the function adding is called and the values of the respective variables m and y, 5 and 3 but the original variables are never passed. However, in some situations, the need arises to manipulate the value of a certain variable from within a function. This can be explained with the help of the following program.

```
// passing parameters by reference

#include <iostream.h>
void duplet(int& m, int& n, int& o)
{
  m* =2;
  n* =2;
  o* =2;
}
int main ()
{
  int x=2, y=7, z=9;
```

```
      duplet (x, y, z);
      cout << "x=" << x << ", y=" << y << ", z="
<< z;
      return 0;
   }
```

```
          The output of this program is:

     x=4, y=14, z=18
```

Here, the main part is the declaration of the function `duplet`. The type of parameters passed are followed by `&`. This specifies that the arguments that are passed should be passed by reference and not by value, that is, we associate the variables `m`, `n` and `o` with the arguments passed to the function. This method of passing arguments by reference is different from that of passing arguments by address in C. While passing an argument by address, the process involves passing the address of the variable rather the variable itself. If the above program is written using the concept of passing arguments by address in C, then the program will be as follows:

```
   #include <iostream.h>
   void duplet(int* m, int* n, int* o)
   {
      *m*=2;
      *n*=2;
      *o*=2;
   }
   int main ()
   {
      int x=2, y=7, z=9;
      duplet (&x, &y, &z);
      cout << "x=" << x << ", y=" << y << ", z="
<< z;
      return 0;
   }
```

Here, the addresses of the variables x, y and z are passed as argument and not variables. Then in the function definition those addresses are assigned to the integer pointers m, n and o, respectively. Here in function definition the variables are accessed with a * operator with the pointers mentioned above.

5.4. Return By Reference

Besides the call by reference method, a function can also return a reference. Consider the following example.

```
int & minimum ( int&  x, int& y)
{

if (x<y)
return x;
else
return y;
}
```

Here a function minimum is declared which is of the return type int&. Due to this, a reference to the variables x and y is returned. Hence, if another function called min is declared, then it will refer to the variables declared inside the minimum function.

5.5 Inline Functions

With a concept known as 'inline function', the time required for calling small functions is drastically reduced. This reduces execution time. In general terms an inline function is a function that is expanded in line when it is invoked. It is expanded at compile time. The main syntax for declaring an inline function is as follows:

```
   inline type name ( arguments ...) {  instructions
... }
```

Here, `type` is the data type of the inline function, while `name` refers to the name of that particular inline function.
For example consider the following:

```
inline float  area(float a, float b)
{

return(a*b);
}
```

5.6 Function Overloading

The process by which functions having the same name can be used for performing different tasks is known as 'function overloading'. With the help of this concept, different functions can be created which will have a common name but different parameter lists. We shall demonstrate this concept with the help of two examples.

Example 1
———————

```
#include <iostream.h>
int calculate (int a, int b)
{
   return (a*b);
}
float calculate (float a, float b)
{
   return (a/b);
}
int main ()
{
   int k=5,l=2;
   float o=5.0,p=2.0;
   cout << calculate  (k,l);
   cout << "\n";
   cout << calculate  (o,p);
   cout << "\n";
```

```
     return 0;
   }
```

The output of the program is as follows:

```
10
2.5
```

Here, we declared and defined two functions having the same name - calculate. However, the major difference between the declarations is that one of the functions accepts parameters of type integer, while the other function accepts parameters of the type float. In the first function call, the two parameters passed are of the type int and the result is the product of the two integers that are passed. Similarly, while calling the second function, the result is division of two numbers. It is apparent from the above that although the names of the functions are the same, the tasks they are performing are different. This means the function is overloaded.

Example 2

```
   #include <iostream.h>
   int vol (int);                     // first function,
with int as parameter type
   double  vol(double, int);          // second function
with int and double as parameter type
   long vol(long, int, int)           // third function
with long,int and int as parameter

   int main()
   {
   cout << vol(20) <<"\n";
   cout << vol(3.5,6) <<"\n";
   cout << vol(150l,50,45) <<"\n";
   return 0;
   }
```

```
// defining the functions  declared above
int vol(int x)
{
return(x*x*x);
}

double vol( double r, int y)
{
return(3.15*r*r*y);
}

long vol(long  m,int n, int o)
{
return(m*n*o);
}
```

The output of this program is:

```
8000
231.525
337500
```

This is an example of an overloaded function. Here the function `vol` is declared and defined thrice with three different types of parameters. Three different types of operations or tasks are performed, although the name of the function remains the same. The first function deals with the volume of a cube with side `x`. The second is related to the volume of a cylinder with height as `y` and radius `r`. The last one deals with volume of a body having length `m`, breadth `n` and height `o`.

Classes And Objects

A class is used for binding data and the functions that work on them. In other words, a class may also be considered as an expanded form of a data structure. It has the unique feature of holding data and functions together. While creating a class, we use the keyword class. Classes can also be instantiated and the object is an instantiation of the class. While declaring objects, assume class as the user-defined data type, and object as the variable.

A class is declared as follows:

```
class class_name {
access_specifier_1:
member 1;
access_specifier_2:
member_2;
} object_names;
```

Here, `class_name` is the name of the particular class or a valid identifier of it. Access specifiers determine the visibility of the members. An access specifier can be either private, protected or public. A member when declared as private can be accessible only by the other members of the same class, while a member declared as protected can be accessed by members of the same class, as well as members from their derived classes. Similarly, if we declare a member as `public`, it can be accessed from any part of the program where it is visible.

However, by default all data members of a class that are declared are private in nature. `member 1`, `member 2`, etc. represent data members or methods of the class declared. `Object_names` represents the name of the object. If we want to create multiple objects, then their names should be separated by commas and a semicolon, else the declaration is terminated with a semicolon after the closing curly brace (}). Objects can also be created within the `main()` function. The syntax is as follows:

Class_name object_name;

Here, `class` refers to the class whose object is to be created. `object_name` represents the name of the object.

6.1. Specifying A Class

A declared class has two types of specifications:
1. Class declaration
2. Definition of class function

Consider the following class example:

```
class QRect
{
int m,n;
public:
void setter(int, int);
int area(void); } rect;
```

Here, a class `QRect` is declared with two private variables `m` and `n`. Next, two methods are declared - `setter` and `area`. The variables declared are data members, while methods are known as member functions. `rect` is the object of class `QRect`.

Consider the following programs.

Program 1

```
#include <iostream.h>
class Calculation {
     int x, y;
```

```
    public:
       void set_values (int,int);
       int area () {return (x*y);}
 };

 void Calculation::set_values (int a, int b) {
    x = a;
    y = b;
 }

 int main () {
    Calculation r1, r2;
    r1.set_values (6,7);
    r2.set_values (8,8);
    cout << "r1 area: " << r1.area() << endl;
    cout << "r2.area: " << r2.area() << endl;
    return 0;
 }
```

The output of this program is:

```
r1 area: 42
r2.area: 64
```

Here, a class called calculation is declared. It has two private members, x and y. Next, two public member functions are declared - set_values and area. In the main method, two instances or objects of the class Calculation are declared. They are r1 and r2, respectively. Both r1 and r2 access the set_value and area methods with the help of the . operator.

Program 2

```
// classes example

#include <iostream.h>
class C1{
     int x, y;
```

```
    public:
      void s1(int,int);
      int area () {return (x*y);}
};
void C1::s1(int a, int b) {
    x = a;
    y = b;
}

int main () {
    C1 r5;
    r5.s1(6,5);
    cout << "area: " << r5.area();
    return 0;
}
```

The output of this program is:

```
area: 30
```

Here, a class called C1 is declared. Next, within the `class` declaration, two private variables `x` and `y` are declared. Another two public member functions are declared - `s1` and `area`. Also, note the scope (::) resolution operator that is used to define a class member from outside the class definition. The function `s1` that is declared within the class definition, has only its prototype declared within the class itself. Similarly, the function area is properly defined. Next, the scope resolution operator is used to define that the function `s1` that is a member of the class `C1`. Within the main method, an object `r5` is created that is an instance of the class `C1`. This object `r5` calls the member function s1 using dot operator and proper arguments.

6.2 Defining Member Functions

As we can see, member functions can be defined in two places - outside, and within the class definition. Consider the case where the member function is defined outside the class definition. A member function should always be defined separately outside a class, if its prototype is declared inside a class. There is an important difference between a member function and a normal function. A member function has an identity label in its header that tells the compiler which class the function belongs to.

Normally, a member function is defined as follows:

```
     return-type class-name :: function-name (argu-
ment declaration)
        {
        function body
         }

         Consider the following section of code.

        void  OurClass :: get_data( int k, int n)
  {
    number = k;
    cost = n;
    }

        void  OurClass :: put_ data( void)
  {
    cout <<" Number : " << number <<"\n";
    cout << "cost : "<<cost <<"\n";
    }
```

Here, `get_data` and `put_data` are two member functions of the class `OurClass`. Neither return any value, and hence their return type is `void`. `::` is known as the scope resolution operator as discussed above. A member function can also be defined by replacing the function declaration with the actual function definition within the class.

6.3 A C++ Program With Class

In this section, let us consider a program where the above concepts are implemented.

```
      #include<iostream.h>
      class i1
      {
      int number;
      float cost;

   public:
     void get_value(int x, float y);
     void put_value (void)
   {
   cout<< "number :" <<number<<"\n";
   cout <<"cost:" << cost <<"\n";
   }
   };

   void i1 :: get_value (int x, float y)
   {
   number = x;
   cost = y; }

   int main()
   {
    i1 i2;
   cout <<"\n object i2 "<<"\n";

   i2.get_value(378,78.6);
   i2.put_value();

   i1 i3;
   cout <<"\n object i3 "<<"\n";

   i3.get_value(37,8.6);
   i3.put_value();

   return 0;
  }

   The output of this program is as follows:

               object i2
               number: 378
                cost :78.6

                object i3
                number: 37
               cost:8.6
```

Now let us come to the explanation part of the program. First a class `i1` is defined. It contains two private variables, and two public functions. The private variable is a number and cost, while the two public member functions are `get_value` and the `put_value`. Within the class declaration, only the prototype of the function `get_value` is declared. This function is then defined outside the class and provides value to both the variables. Therefore, member functions can have direct access to private data members. Similarly, the `put_value method` is defined inside the class. This means that the function `put_value` behaves as an inline function. This function displays the values of the two private variables - number and cost. Within the `main()` function two objects `i2` and `i3` are created and the methods described above are called by these objects with proper argument.

6.4 Nesting Of Member Functions

A member function of a class is an object of that class by using the dot operator. It can also be called using its name inside another function. This feature is known as `nesting of member functions`. The following program illustrates this feature:

```
// A program showing nesting of member function.
#include<iostream.h>
class s1
{
int m,o;
public:
void i1(void);
void d1(void);
int  l1(void);
};

int s1 :: l1(void)
{
if(m>= o)
return(m);
else
return(o);
}
void s1 :: i1(void)
{
cout<<"input values of m and o"<<"\n";
cin>>m>>o;
}

void s1 :: d1(void)
{
cout << "largest value= " <<l1()<<"\n";   // the
member function l1() is called
}
```

```
int main()
{
s1 s2;
s2.i1();
s2.d1();
return 0;
}
```

The output of this program is:
Input values of m and o
67 78

[These values are entered by the user from keyboard during execution of the program]

```
largest value=78
```

A class `s1` is first declared with two private variables `m` and other public member functions are declared within the class and these are `i1`, `d1` and `l1`, respectively. The next part of the program defines these functions one after the other. The function `l1` deals with an `if` statement that checks which is greater among the two variables. The function `i1` helps to take input values for these variables. It is to be noted that when the function `d1` is defined, it calls the member function `l1`, a case of nesting of member functions.

6.5 Static Member Functions

This type of variable is declared using the keyword static followed by the data type and name of the variable. This is shown in the program given below. The static member variable has some special features. They are initialized with 0 when the first object is created. Only one copy of the variable is created, irrespective of the number of objects declared. The same copy is shared by all the objects. A member function is called static if it has the following properties:

- While calling a static member function, the class name should be used instead of its objects:
  `class-name :: function-name`

- A function declared static can have access to only other static members - functions and variables declared within the same class.

This is highlighted in the following program.

```
#include <iostream.h>
class t1
{

int x;
static int y;  // This is a static member vari-
able
public:
  void set(void)
{
        x= ++y;
}
  void show(void)
{
cout<< "object number: " <<x <<"\n";
}
static void s1(void)  // static member function
{
cout<<"count: "<<y<<"\n";
}
};

int  t1 :: y;

int main()
{
        t1  m1,m2;
        m1.set();
        m2.set();

        t1 :: s1();
        t1  m3;
        m3.set();

        m1.show();
        m2.show();
        m3.show();
        return 0;

}
```

The output of the program is as follows:

```
count: 2
object number: 1
object number: 2
object number: 3
```

Here, a class `t1` is declared. Two member variables `x` and `y` are declared. Here, `y` is the static member variable. The next three public methods to be declared are `set`, `show` and `s1`. `s1` is the static member function that displays the number of objects created till that moment. Similarly, the static member variable `y` maintains the count of the number of objects that is created. The show() function displays the code number for each object. Consider the statement `x = ++y;`

This statement is executed whenever the set() function is called and the current value of the variable y is assigned to the code.

6.6 Friendly Functions

Consider a case where two classes, `engineer` and `chemist` are defined and we want a function `behavior()` to operate on the objects of these classes. C++ allows the common function to be friendly to both the classes. This allows the particular function to have access to the private data of these classes.

For making a function friendly to a class, we must declare the function as a `friend` of the class as follows;

```
class MN
{
…………………….
……………………..
public:
……………………..
……………………….
friend void pqr(void)
};
```

Here, the function `pqr` is declared as a friend function. It should be noted that the function declaration should be preceded by the keyword `friend`. The friend function has some important features as follows:

1. It can be invoked like a normal function without taking the help of any object
2. It normally has the objects as arguments.

3. It cannot access the member names directly, and uses an object name and dot operator with each member name

An illustration is shown below.

```
# include <iostream.h>
class s1
{
int a;
int b;
public:
  void set() {  a=50; b=40;}
  friend float m1(s1  s);
};
float m1(s1  s)
{
return float(s.a + s.b)/2.0;
}

int main()
{
  s1  k;
  k.set();
  cout<<" Mean value = " <<m1(k) <<"\n";
  return 0;
 }
```

The output of this program is:

```
Mean value = 45
```

Here, `m1()` is declared as a friend function. The function accesses both the variables `a` and `b` with the help of the dot operator and the object passed to it. The function calls `m1(k)` within the main function passes the object `k` by value to the friend function. The friend function `m1()` that is declared inside the class `s1` is defined outside the class. This function finds the mean of the two values assigned to the variables `a` and `b`.

Besides this, a friend function can also act as a bridge between two classes. This is demonstrated in the following program.

```
# include<iostream.h>

class MN1;
class MN2
{

  int x;
  public:
            void set(int i) {x=i;}
            friend void m1(MN2,MN1);

  };

  class MN1
  {
  int a;
  public:
            void set(int i) {a=i;}
            friend void m1(MN2,MN1);
  };
void m1(MN2 m, MN1 n) // The friend function m1
is defined
  {
  if(m.x  >=  n.a)
  cout << m.x;
  else
  cout <<n.a;
}

int main()
{
 MN1 mn1;
mn1.set(15);
MN2  mn2;
mn2.set(25);
m1(mn2,mn1);
return 0;
}
```

The output of this program is:

```
25
```

Here, the function `m1()` has arguments from both the classes `MN2` and `MN1`. When the function `m1` is declared as a friend function in the class `MN2` for the first time, the compiler does not acknowledge the presence of the class `MN1`, unless we declare its name at the beginning of the program as `class MN1;` This is known as the 'forward declaration'.

Constructors And Destructors

7.1 Introduction

Member variables can be initialised while creating the objects by using constructors. We can also destroy the objects when they are not required using destructors.

Classes have a very complicated structure. We can use constructors and destructors to initialise member variables of a class or destroy class objects. Besides, initialisations for objects construction also indicate memory allocation for the objects. Similarly, besides cleaning up objects, destructors de-allocation of memory used by the objects.

Constructors and destructors are usually declared within the declaration of a class. Here, you can define them either inline or external to the class declaration. Default arguments can be included in the constructors. Constructors and destructors also have some limitations.

Features of return values and return types are not found in constructors and destructors. Destructors don't take any argument. Programmers cannot use references and pointers in constructors. The keyword `virtual` cannot be used while declaring a constructor. Class objects that include constructors and destructors cannot be inserted in the unions. Constructors and destructors always maintain the access rules of member functions. The compiler where the whole program is run can automatically call constructors while defining class objects. Similarly, the compiler can automatically call destructors where the class objects turn to be insignificant.

7.2 Constructors

The task of a constructor is to set up the object in order to make it usable. These are special members of functions in a class. It can only build an object that belongs to its class. Constructors maintain the same name as that of a class. You can insert any number of overloaded constructors in a class. However, there should be a different set of parameters.

Constructors do not return any values. These are not created between base and the derived classes. If we do not provide a Constructor then the compiler creates a default constructor that does not include any parameter. This is so as there must be a constructor and it can be empty or a default constructor. No default constructor will be created if a constructor with parameter is supplied by the programmer. Constructors cannot be virtual. You can define the multiple constructors for the same class.

Here is an example of a constructor.

```
Syntax:
class dealer
{
  private:
          int person_identity ;
          float daily_sales ;
  public:
          dealer() //default constructor
          {
                  person_identity = 0 ;
                  daily_sales = 0.00 ;
          }
};
```

Now let us look at the following example:

```
//here we can get class with a constructor
class integer
```

```
   {
         Int x, y;
   Public:
        Integer(void) ;                         //here
the constructor is declared
         ………
         ………
   } ;
   integer   ::   integer(void)                 //here
the constructor is defined
   {
            x = 0; y = 0;
   }
```

Here, we can see that a class includes a constructor, and is initialised automatically when an object is created. Look at the declaration below:

```
   Integer int2;        // here an object int2 is cre-
ated
```

This declaration creates the object `intl2` as the type `integer`. At the same time, it initialises the data members `x` and `y` to `0`.

Exception before the completion of a constructor causes difficulties. In such a case, a destructor for cleaning the object will not appear. Here, the most common problem is the allocation of resources in constructors. The destructor will not get scope for the de-allocation of resources if any exception appears in the constructor. This problem often happens in case of `'naked'` pointers. Let us go through the following example:

Example:

```
   //: problems if exception is thrown in the con-
structor before completion

   // Naked pointers
   #include <fstream.h>
       ofstream out("nudep.out");
       class Rat {
public:
Rat() {  cout << "Rat()" << endl; }
~Rat() {  cout << "~Rat()" << endl; }
};
class Frog {
public:
void* operator new(size_t sz) {
cout << "allocating a Frog" << endl;
throw int(47);
}
void operator delete(void* p) {
cout << "deallocating a Frog" << endl;
::delete p;
}
};
class UseResources {
Rat* bp;
Frog* op;
public:
UseResources(int count = 1) {
cout << "UseResources()" << endl;
bp = new Rat[ count];
op = new Frog;
}
~UseResources() {
cout << "~UseResources()" << endl;
delete []bp; // Array delete
delete op;
}
};
int main() {
try {
UseResources ur(3);
} catch(int) {
cout << "inside handler" << endl;
}
} ///:~
```

The output of this program is:

```
UseResources()
Rat()
Rat()
Rat()
allocating a Frog
inside handler
```

Here we have entered the `UseResources` constructor. The `Rat` constructor is also completed successfully for the array objects. We can see that an exception `Frog::operator` is also used. The problem here is that it ends inside the handler without calling the `UseResources` destructor. The `UseResources` could not be finished. It indicates that the `Cat` object was created successfully and was not destroyed.

7.3 Types of Constructors

There are various types of constructors - default constructors, copy constructors and dynamic constructors.

A Default Constructor is a special category of constructor that does not accept any parameter. For example, we can say that if `marketing` is a class, then `marketing::marketing()` is a default constructor because it doesn't need any parameter. The compiler will provide the default constructor. Constructors are not particularly defined in a class. It must not have any argument. The default constructor that is provided by the compiler does not have any special activities. What it can do is initialise data members that include a dummy value.

Consider the following example:

```
   #include <iostream.h>
   class Square
   {
     private:
                         float span ;
                                 float breath ;
     public:
                         S    q    u    a    r    e    (    )
//this is the Default Constructor, without any argu-
ment
                         {                  }
                         Square(float    I,    float    b)
//this is the Constructor with two argument
                         {
                  span =I;
                  breath = b ;
                          }
                          void Insert_Ib(void)
                         {
                  cout  <<  "\n\t  Insert  the  span  of  the
Square: " ;
                  cin >> span ;
                                        cout << "\t Insert
the breath of the Square: " ;
                  cin >> breath ;
                         }
                         void View_area(void)
                    {
                  cout << "\n\t The area of the Square = "
<< span*breath ;
                  }
   };
     //this is the end of the class definitions
   void main(void)
   {
```

```
                Square q1 ;
//here the first Constructor without any argument

is invoked
                cout << "\n First Square----------'\n"
;
                q1.Insert_Ib() ;
                q1.View_area() ;
                cout << "\n\n Second Square---------
'\n" ;
                Square q2 (5.6, 3.5) ;
      //here the second Constructor with two

arguments is invoked
                q2.View_area() ;
    }
```

```
    The output of this program is:

    First Square---------'

                Insert the span of the Square:    4
                Insert the breadth of the Square:5

                The area of the Square = 20

    Second Square-------'
                The area of the Square = 19.6
```

Another type of constructor is a copy constructor. Let us look at the form of the copy constructor:

```
    class name (class name &).
```

While initialising an instance, the copy constructor is used by the compiler. Here, the values of the other instance of the same type are used.

Now let us look at the example below:

```
#include <iostream.h>
class Model
{
  private:
           int x,y;
  public:
           Model(int m, int n)         //this    is
the constructor with two argument
           {
           x=m;
           y=n;
             cout<<"\nHere  the  Parameterized  con-
structor is working\n";
    }
    Model(Model &p)                //copy Constructor
    {
                    y=p.y;
                    x=p.x;
             cout<<"\nHere the Copy constructor is
working\n";
    }
           void publish()
           {
                    cout <<x<<"\n"<<y;
           }                           //End of the
class definition
};
void main()
{
           Model m1(53,63) ;           //Invokes
the constructor
           Model m2(m1) ;
//Invokes the copy constructor
           m2.publish() ;
}
```

```
The output of this program is:
```

Here the Parameterized constructor is working

Here the Copy constructor is working
53
63

Dynamic Constructors are nothing but the process of allocating memory to objects while constructing them. These constructors are used to allocate memory when the objects are created. This is extremely helpful for the allocation of adequate memory size for the objects of varying size. The new operator helps memory allocation.

Now let us look at the following example:

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

class S1
{
        char *z1;
        int l1;
  public:
                S1()        //this is the con-
structor-1
{
        l1 = 0;
        z1 = new char[ l1 +1];
}
S1(char *x)          //constructor-2
{
        l1 = strlen(x);
        z1 = new char[ l1 +1];      //one  addi-
tional
```

```
                                                    //character
for \0
                    strcpy(z1, x);
    }
    void display(void)
    { cout <<z1 << "\n";}
                    void include(S1 &m, S1 &n);
    };
    void S1 :: include(S1 &m, S1 &n)
    {
                    l1 = m.l1 +n.l1;
                    delete z1;
                    z1 = new char[ l1+1] ;          //dynamic
allocation

                    strcpy(z1, m.z1);
                    strcat(z1, n.z1);
    };

    int  main()
    {
                    char *initial = "Jack ";
                    S1    naming1(initial),     naming2("Ram
"),naming3("Raj"),t1,t2;

                    t1.include(naming1,naming2);
                    t2.include(t1, naming3);
                    naming1.display();
                    naming2.display();
                    naming3.display();
                    t1.display();
                    t2.display();

                    return 0;
    }
```

The output of this program is:

```
Jack
Ram
Raj
Jack Ram
Jack Ram Raj
```

7.4 Constructing Two Dimensional Arrays

We can name multidimensional arrays as "Arrays of Arrays". Two dimensional arrays can be regarded as a two dimensional table which is made of various elements. These elements always maintain a uniform data type.

Consider the following example, of a matrix variable that is constructed using the objects of class types.

Example:

```
#include <iostream.h>
class matrix
{
        int  **q;           //this   is   the
pointer to matrix
        int  z1,z2;  //these are the dimensions
  public:
        matrix(int v, int u);
        void  getting_element(int a, int b, int
cost)
        { q[ a][ b] =cost;}
        int  &  insert_element(int  a,  int  b)
        { return   q[ a][ b] ;}
};
matrix :: matrix(int v, int u)
{
        int a;
        z1=v;
```

```
            z2=u;
            q= new int *[ z1] ;           //Here   the
an array  pointer is created
            for (a = 0;a<z1;a++)
                   q[ a]  = new int[ z2] ;  //Here space
for each row is created
   }

   int main()
   {
            int l, p;

            cout<<"Here you can insert the size of
matrix: ";
            cin>>l>>p;
            matrix A(l,p);                //matrix
object A constructed

            cout<<"Here you can enter the matrix ele-
ments row by row  \n";
            int a,b,cost;

            for(a=0;a<l;a++)
            {
             for(b=0;b<p;b++)
              {
                   cin>>cost;
                   A.getting_element(a,b,cost);
             }
            }
     cout<<"\n";
     cout<<A.insert_element(2,3);
     return  0;
   }
```

The output of this program is:

```
Here you can insert size of matrix: 3  4
Here you can enter the matrix elements row by row
as shown below

11      22      33      44
55      66      77      88
99      111     222     333

After that the element corresponding to third row
and fourth column will be displayed as below.

333
```

7.5 Destructors

Destructors are not as complicated, and are called automatically. There is only one destructor per object. A destructor has a single name, its class and is headed by a tilde (˜) operator.

Let us look at the example below:

```
performer::~performer() {
   potency = 0;
   nimbleness = 0;
   fitness = 0;
}
```

Consider the following example of both constructor and destructor:

Example:

```
    //example including both constructor and
destructor

        #include <iostream.h>
        class Plant {
```

```
        int tallness;
        public:
   Plant(int initialTallness); // a Constructor is
inserted here
   ~Plant(); // a Destructor appears with a tilde
sign
   void growth(int years);
   void copysize();
   };
   Plant::Plant(int initialTallness) {
   tallness = initialTallness;
   }
   Plant::~Plant() {
   cout << "inside Plant destructor" << endl;
   copysize();
   }
   void Plant::growth(int years) {
   tallness += years;
   }
   void Plant::copysize() {
   cout << "Plant tallness is " << tallness << endl;
   }
   int main() {
   cout << "before opening brace" << endl;
   {
   Plant t(12);
   cout << "after Plant creation" << endl;
   t.copysize();
   t.growth(4);
   cout << "before closing brace" << endl;
   }
   cout << "after closing brace" << endl;
   } ///:~
```

The output of this program is:

```
before opening brace
```

```
after Plant creation
Plant tallness is 12
before closing brace
inside Plant destructor
Plant tallness is 16
after closing brace
```

Here we can see that the destructor is automatically called at the ending brace of the scope where it is enclosed. While creating an object by a constructor, a few resources such as memory space are usually allocated for use and can be allocated to the data members.

Here we make free the memory space before the destruction of the object. We have already discussed that a destructor cannot take any argument and it cannot return any value. The compiler that we use will automatically call the destructor.

Consider the following example:

```
   #include <iostream.h>
   class S1
   {
          private:
               int m,n;
         public:
                S1(int x, int y)   //this is the con-
structor with two arguments
    {
     m=x;
                   n=y;
     cout<<"\nConstructor in work\n";
                  }
              void print()
                  {
                   cout<<"\nThe object value of S1
class\n";
```

```
                    cout <<m <<"\n"<<n<<"\n";
                      }
                ~S1()
                     {
                   cout<<"\nCalling Destructor";
                     }
          };         //This is the ending of the class
definition
   void main()
   {
                                    S1      k(8,9);;
//Here the constructor is invoked
         k.print() ;
   }
```

The output of this program is:

```
Constructor in work
The object value of S1 class
8
9
Calling Destructor
```

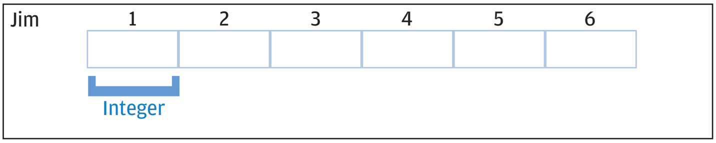
Compound Data Types

8.1 Arrays

Arrays are defined as a series of elements of the same type, placed in contiguous memory locations. They are referenced individually, by adding an index to a unique identifier. An array can also be defined as a data structure, allowing a collective name to be given to a group of elements having the same type. The individual element of an array is identified by its own unique index. This is also referred as the subscript of the element.

If we want to store six values of the type integer, but don't want to declare a variable with a different identifier for each of them, we can do it by using arrays.

This is shown as follows:



Here, the array Jim is of type `integer` that contains six integer values and is represented as shown above. Each blank panel shown above represents an individual element of the array `Jim`. The elements of the array are numbered from 0 to 5. These numbers are called index of the array element. Note that in arrays, the first index is always `0`. An array is like a regular variable and should be declared before it can be used. A normal array declaration is as follows:

```
type name [elements];
```

Here, `type` refers to the data type such as `int` and `float`. `name` refers to a valid identifier, while the `elements` field indicates the numbers of elements in the array. You can declare the above case as:

```
    int Jim[6];
```

While declaring a regular array, if you don't specify the initial values of the elements, the elements will not be initialised to any value by default The elements of global and static arrays are initialised automatically with their default values, i.e. `0`. When an array is declared, you can assign initial values to each element by enclosing the values within braces`{}`. For example,

```
int Jim[6] = {12,34,56,78,12,32};
```

The number of values inside the braces should never be larger than the number of elements declared for the array in square `[]` brackets. C++ also allows the square brackets to remain empty of initialisation values for certain situations. For this situation, the compiler assumes the size of an array that matches the number of values included within the braces. This is shown below:

```
  int Jim[] = {212,234,456,781,10,312};
```

At any point in a program, where an array is visible, the value of any individual elements can be accessed, as if it's a normal variable. The syntax for this is:

```
     name [index]
```

For example, consider the case in which an array named Jim has six elements and each of those elements is of type `int`. If we want to store the value `60` in the third element of the array `Jim`, we can write the following statement:

```
Jim[ 2]  =  60;
```

If we want to pass the value of the third element of the array to a variable x, we can write:

```
x  =  Jim[ 2] ;
```

The third element of the array Jim is specified as Jim[2] as the first element is denoted as Jim[0] , second as Jim[1] and so on.

There are two different uses of the [] brackets related to arrays. One of the tasks of this bracket is to specify the size of an array when they are declared and the other is to specify indices for array elements. Some valid operations with arrays are as follows:

```
Jim[ 0]  =  x;
Jim[ a]  =  7;
b  =  Jim [ a+2] ;
```

where x and b are two integer variables.

Consider a program where the above concepts are implemented.

```
 // arrays example
#include <iostream.h>
```

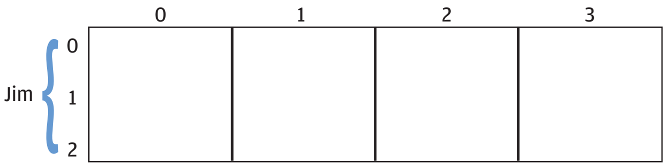
int Jim[] = {16, 223, 77, 401, 12}; // an array Jim is declared and values are assigned to its elements.

```
int  x,  res=0;
int  main  ()
{
   for  (  x=0  ;  x<5  ;  x++  )
   {
      res  +=  Jim[ x] ;   //  implies  res=  res+Jim[ x]
```

```
    }
    cout << res;
    return 0;
}

The output of this program is:
729
```

Here, an array `Jim` is declared and its elements are initialised with some values. Then, by using the `for` loop, the values of the elements are added and the result is stored in an integer variable `res`. The value of `res` is then displayed.



Multidimensional arrays

A multidimensional array is an "array of arrays". This can be represented as follows:

In the above example, `Jim` represents a bi-dimensional array of 3 x 4 elements of type `integer`. This is declared as:

```
int Jim[ 3][ 4] ;
```

In C++, in order to accept arrays as parameters while declaring a function, it is required to specify the element type of the array in its parameters - an identifier and a pair of void brackets [].
Consider the following statement:

```
void p1(int  a1[ ])
```

Here, `p1` is the function that accepts an array as parameter of type `int` called `a1`. You can understand this from the following program.

```
// arrays as parameters
#include <iostream.h>

void p1(int a1[], int l1) {
  for (int x=0; x<l1; x++)
    cout << a1[x] << " ";
  cout << "\n";
}
int main ()
{
  int first[] = {15, 22, 32};
  int sec[] = {21, 41, 61, 81, 100};
  p1(first,3);
  p1(sec,5);
  return 0;
}
```

The output of this program is:

```
15  22  32
21 41 61 81 100
```

Here, a function `p1` is declared and two parameters are passed. The first parameter `int a1[]` accepts an array with elements of type `int` while the second parameter is a variable of type `int`. This parameter is included to determine the length of each array passed to the function. The `for` loop helps print out the array. Within the main function, two arrays are declared - `first[]` and `sec[]` and values are assigned to their elements.

8.2 Character Sequences

Since strings are mainly sequences of characters they can be represented as plain arrays of `char` elements. For example, the statement,

```
char x1[ 32] ;
```

implies that `x1` is an array that can store up to 32 elements of type `char`.

Initialising Null-terminated Character Sequences

Since character arrays are regarded as ordinary arrays, they follow a simple rule. If we want to initialise an array of characters with some character sequences, we can initialise it as any other normal array, as shown below:

```
char  m1[ ]  = { 'I','n','d','i','a','\0'} ;
```

This shows an array named `m1` declared of type `char` with six elements that form the word "India", and an additional null '\0' character at the end. Besides this method, there is another method to initialise the values of the char elements of an array - using the string literals. Double quoted strings are considered as literal constants whose data type is in a null-terminated array of characters. In general, string literals enclosed in double quotes have a null character ('\ 0') which is always added to its end. Thus, whenever we want to in initialise the elements of an array of type `char`, you can use two methods.

```
   1.        Normal Method
(char f1[ ] ={ 'I','n','d','i','a','\0'} ;)
   2.        By using string literals
(char f1[ ] =  "India";)
```

In both cases, the array f1[] is declared with six elements of type `char`. Out of the six characters, five are for the word "India", while the last null character is for specifying the end of the sequence. In the second case, by using string literals, the null character ('\ 0') is automatically inserted at the end.

Applying Null-terminated Sequence Of Characters

Null terminated sequence of characters is an effective way of treating strings. They can also be used in procedures. For example, the extraction and insertion operators `cin` and `cout` support these sequences and can be used directly to extract strings of characters or to insert them. The following program illustrates this.

```
   // null-terminated sequences of characters
   #include <iostream.h>
   int main ()
   {
     char q1[ ] = "Please, enter your first name:
";
      char g1[ ]  = "Hello, ";
      char y1[ 80] ;
      cout << q1;
      cin >> y1;
      cout << g1 << y1 << "!";
      return 0;
   }
```

The output of this program is:

```
Please, enter your first name: Harry
Hello, Harry
```

In the above program, three arrays of type `char` are declared. They are q1, g1 and y1, respectively. The first two arrays q1 and g1 are initialised with string literals constant, while the third one y1 is left uninitialised. For the first two arrays q1 and g1, the size was implicitly defined, while for the last one, the size is explicitly defined - 80 characters. When we run the program, it prompts you to enter your name. The name entered is assigned to the array y1. Now if the number of characters entered is less than the number of elements specified during declaration of y1, then \0 will append just after the last entered character. If some one enters Harry as shown above, y1 will be as follows

y1	H	a	r	r	y	/0		

The name displayed with the value of array g1 is Hello.

8.3 Pointers

Identifiers also help us to refer to our variables. When a variable is declared, it is assigned a specific memory address.

Reference variable

Each variable has a specified address in the memory known as the reference to that variable. With the help of this address, a certain variable can be located within the memory. A reference to this variable is obtained by putting the & symbol before the identifier. For example, Jim=&Jack; assigns the address of the variable Jack to the variable Jim. Here, & is the reference operator that precedes the variable Jack. Consider the following code segment:

```
a1 = 52;
f1 = a1;
t1 = &a1;
```

Here, the value 52 is assigned to a1. If a1 has a memory address of 1778, the next statement that follows, copies the content of the variable a1 to a new variable f1. The last statement on the other hand does not copy the value contained in a1, but only a reference to it – its address 1778 which we have assigned to the variable t1. This is due to the presence of the address of operator `&` that precedes the identifier `a1`. Thus a pointer may be defined as a variable that stores the reference to another variable.

Pointers are special variables that store the reference of a variable of its type. That means an integer pointer can store the address of an integer variable, a float pointer can store the address of a float variable and so on. In the above examples, all variables used to store reference of other variables are pointers. Pointers play a vital role and have uses in various applications. Generally a pointer is defined as follows:

```
type * name;
```

Here, `type` refers to the data type of the variable that the pointer points to. Let us take another example that is shown below.

```
int * n1;
char * c1;
float * g1;
```

Here, three pointers have been declared. Each pointer that is declared points to a different data type, but all of them occupy the same amount of space in the memory. However, the data to which the pointers point to, are of different types - the first pointer points to an `int`, the second points to a `char` and last one points to a `float`. Now let us look at a program which is based on pointers.

```
 // Program 1
// Application of pointers
#include <iostream.h>
int main ()
{
  int f1, s1;
  int * m1;
```

```
    m1 = &f1;
    *m1 = 56;
    m1 = &s1;
    *m1 = 78;
    cout << "first value is " << f1 << endl;
    cout << "second value is " << s1<< endl;
    return 0;
}
```

The output of this program is:

```
first value is 56
second value is 78
```

Here, two variables `f1` and `s1` and one pointer `m1` of type `int` are declared. It is to be noted that values are not directly set to the either `f1` or `s1`, but receive values indirectly with the help of the pointer `m1`. Now coming back to the program, first a reference to the variable `f1` is assigned to the pointer `m1`. Next, a value `56` is assigned to the memory location pointed by `m1`. Next, we again assign the reference to the variable `s1` to the pointer `m1` and then the value `78` is assigned to the memory location pointed by `m1`. Here, an operator `*` is used. This operator refers to the value of the variable it points. This operator is called "`value pointed by`".

The following is a program which deals with the application of pointers.

```
// Program 2
// Application of pointers

#include <iostream.h>
int main ()
{
    int f1 = 5, s1 = 15;
    int * p3, * p4;
```

```
     p3 = &f1;                  // p3 = address of
f1
     p4 = &s1;                  // p4 = address of
s1
     *p3 = 60;            // value pointed by p3 =
60
     *p4 = *p1;           // value pointed by p4 =
value pointed by p3
     p3 = p4;                // p3 = p4 (value of
pointer is copied)
     *p3 = 20;              // value pointed by p3
= 20
     cout << "first value is " << f1 << endl;
     cout << "second value is " << s1 << endl;
     return 0;
  }
```

The output of this program is:

```
first value is 60
second value is 20
```

Initially, two variables, `f1` and `s1` of the type `int` are declared. Then, two pointers `p3` and `p4` are declared. These are also of type `int`. Initial values are assigned to the variables `f1` and `s1`. Next, a reference to the variables `f1` and `s1` is assigned to the pointers `p3` and `p4`. The next statement `*p3=60` assigns the value `60` to the memory location pointed by the pointer `p3`.

Pointers and arrays

Arrays are similar to pointers. Consider the two declarations below:

```
   int  n1[ 15] ;          // n1 is an array of type
int
   int  *  x;              // x is a pointer of type
int
```

Based on the above statements, we can make a valid statement as follows:

```
 x = n1; //
```

The above statement implies that `x` and `n1` are equivalent and will have the same properties. The only difference lies in the fact that the value of the pointer `x` can be changed, while the pointer `n1` will always point to the first of the 15 elements of type `int`.

Similarly, if we declare a statement `n1= x;`, the statement will be considered invalid as `n1` is an array and operates as a constant pointer. Since we cannot assign values to constants, the above statement will not work. Now let us consider another program based on pointers.

```
   # Program 3: A program to show various expres-
sions related to pointers
   #include <iostream.h>

   int main ()
   {
     int n1[ 5] ;   // an array n1 declared of type
int
     int * q;           // a pointer declared  of
type int
     q = n1;
   *q = 1;
     q++;
   *q = 2;
     q = &n1[ 2] ;
   *q = 3;
     q = n1+ 3;
   *q = 4;
     q = n1;
   *(q+4) = 5;
     for (int x=0; x<5; x++)
       cout << n1[ x] << ", ";
```

```
    return 0;
}
```

The output of this program is:

```
 1,2,3,4,5
```

Dynamic Memory

Dynamic memory is the memory to be used in a program during its run time. C++ provides two effective operators — `new` and `delete`, which are used for this purpose.

The `new` And `new[ ]` Operators

The `new` operator is used for dynamic memory allocation. This operator should be followed by a data type. If more than one element is required, specify it within brackets `[ ]`. The general form of using this operator is as follows:

```
p1 = new type
p1 = new type [ number_of_elements]
```

Here, the first expression `p1 = new type` is used for allocating memory so that one single element of type `type` can be contained. Similarly, the second expression is used for assigning an array of elements of type `type`. Here, `number_of_elements` is an integer value. Consider the statements:

```
int * b;
b = new int [ 6] ;
```

In the second statement, the new operator dynamically assigns space for six elements of type `integer` and returns a pointer to the first element of the sequence assigned to `b`. Therefore, the pointer `b` points to a valid block of memory with space for six elements of type `integer`.

The `delete` And `delete []` Operators

Dynamic memory is used for specified tasks, and is freed once the task is completed so that the memory is available for other tasks/requests. This is done by the `delete` operator. The format of the `delete` operator is as follows:

```
        delete m;
delete [] m;
```

Here, the first statement deletes the memory allocated for a single element, while the second statement allocates memory for arrays of elements.

8.4 Other Data Types

Defined Data Types `typedef`

There is a provision for defining types based on other existing data types. One way is by using the keyword `typedef`. The syntax is as follows:

```
typedef existing_type new_type_name ;
```

In the above expression, `existing_type` is a fundamental data type and `new_type_name` is the name related to the new type.

Consider the following statements:

```
typedef char x;
typedef unsigned int  y;
typedef char * p1;
typedef char f1[ 50] ;
```

We have declared `x,y,p1` and `f1` as `char`, `unsigned int`, `char*` and `char[ 50]` , respectively, so that these can be used suitably in later declarations. Consider the following statements:

```
x         mx,  xm,  *pt2;
y         y1;
p1        y2,y3;
f1         fr  ft;
```

`typedef` does not create different types. Rather, it only creates synonyms of the existing ones. `typedef` is also used to define data types with a long and confusing name.

Unions

The next important data type that is often used is `unions`. This data type is used to access the same memory accessed by other data types. The declaration of a union is similar to that of a structure, but in functionality both are different. A union is declared as follows:

```
union union_name {

  member_type1 member_name1;
  member_type2 member_name2;
  member_type3 member_name3;
  .
  .
}  object_names;
```

Note that in an union declaration, all the elements occupy the same physical space in memory. For example consider the following union.

```
union m1 {
  char v;
  int y;
  float z;
  }  my;
```

Here, three elements are declared — `v`, `y` and `z`, each having a different data type. However, all of them refer to the same memory location. Hence, if the value of one of the element gets altered, it affects the value of the other elements.

Anonymous Unions

There is also a provision for declaring a union without any name. This is known as anonymous union declarations. The members of this union can be accessed directly by using the names of its members. Consider the following two structure declarations:

```
      First declaration:
   struct {
     char t1[ 15] ;
     char a1[ 15] ;
     union {
       float d1;
       int y1;
     } p1; // name of the union
   } b1;

   Second  declaration
    struct {
     char t1[ 15] ;
     char a1[ 15] ;
     union {
       float d1;
       int y1;
     } ;  // union name is missing
   } b1;
```

In the above two declarations, the only difference is that in the first one, a name has been given to the union (`p1`). Similarly, in case of the second declaration, the name is missing. The difference can be properly explained while accessing the members `d1` and `y1`. In the case of an object belonging to the first type we can access it by the statement:

```
b1.p1.y1
b1.p1.d1
```

On the other hand, for an object belonging to the second type we can access it by the statement:

```
b1.y1
b1.d1
```

Enumerations `enum`

An enumeration is a data type that can be given a finite set of named values, each of which has a meaningful name. It can be declared as follows:

```
enum enumeration_name {
  value1,
  value2,
  value3,
  .
  .
} object_names;
```

For instance, consider the following declaration. Here, a new type called `t1` is created for storing different values.

```
enum t1 {tx, ty, tz, tn, th, tu, ti, td};
```

Note that no fundamental type has been included in the declaration part. In fact, a completely new data type has been created. The values that are present within the braces are the new constant values.

After the enumeration `t1` has been declared, we can declare the following expressions based on it. They are as follows:

```
            t1 tq;
tq = tj;
```

The constants of the enumeration are assigned an integer numerical value internally. It is to be noted that the first enumerator has a value `0` and each successive enumerator is one greater than the previous one. Hence in the previous declaration (where t1 was created), `tx` would be 0, `ty` would be `1`, and so on. Also, an integer value can be specified explicitly for any of the constant values of an enumerated type.

Notes

Notes