Practical C++

by Rahul Batra

Preface

C++ has become a very important programming language, especially because of the use of Object Oriented Approaches in all types of software development. I've written this book as a result of my experience with C++ over the past years. Any errors, corrections, comments and suggestions should be mailed to itsmeslayer@yahoo.co.in

Rahul Batra

6/12/2005
First Edition

Acknowledgements

This book is dedicated to my parents and my sister who have helped me reach the place I am. I would also like thank my relatives and friends who have helped me along the way.

My heartfelt thanks go out to Mr Nilesh Shaw, the teacher who taught me about computers and Mrs Ghosh who taught me the C++ language. I would also like to thank Tanuj, Ashish, Mridul, Bhavana and Sonali for helping me out with this book.

Chapter 1 : Hello World Program

The compiler I am going to use is Turbo C++ Version 1.01 by Borland, now available free of cost at the Borland website. With little modifications, you can apply the same code to your favourite compiler.

So let's start with a basic program, to print the words - Hello World onto the screen.

Program Listing :-

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

void main()
{
  clrscr();
  cout<<"Hello World"<<endl;
  getch();
}
```

The first two lines include the two header files, namely iostream.h (Input/Output Stream) and conio.h (Console Input Output).

void main() is the beginning of the main function, where execution starts. clrscr() and getch() both appear in conio.h, the former is used to clear the screen, while the latter to wait until the user inputs a character.

The line which actually prints the words is the 'cout' line. Note its particular syntax. It is a statement which is going to be used quite often in C++ programming. The word 'endl' is used to specify 'end line', for jumping to a new line after the words have been printed. The usage of 'endl' is pointless here and has been shown for educational purposes.

Chapter 2 : Datatypes

The formal definition of a data type - the description of a set of values and the basic set of operations that can be applied to it. In C++, there are two kinds of data types :-

- Fundamental data types

- Derived data types

In this part we will look at only the fundamental data types, which fall under these categories :-

- Integer

- Character

- Floating point

- Boolean

- Double Precision

The three most used data types are integer, float and character.

The integer data type is used to represent integer whole numbers from -32,768 to +32,767. A variable of integer type can be declared as :-

```
int num;
```

Let us look at a program that adds, two numbers and displays their result.

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

void main()
{
  clrscr();
  int num1 = 10;
  int num2 = 5;
  cout<<"Sum = "<<num1+num2<<endl;
  getch();
}
```

The output of this program will be :-

Sum = 15

Take particular notice of the statements between 'clrscr()' and 'getch()'.

Floating point numbers are used to represent numbers which have a decimal point. For example : 3.1, 4.0099 etc.
A floating point variable can be declared as :-

float num;

where 'num' is the name of the variable.

Example : float a = 4.099;

Character data type is used to represent characters, as the name suggests. It is defined as :-

char a = 'X';

char b = '45';

The value of 'b' will be 45, but it won't be treated as an integer, means you cannot do mathematical operations like addition on it.

Chapter 3 : Input/Output

The command used for output is - cout, and the command used for input is - cin.

A stream is a sequence of bytes used in transferring data between source and destination. "cout" and "cin" statements are nothing but output and input streams for the console (screen).

```
int a=10;
cout<<a<<endl;
```

The above code snippet prints the value of variable 'a' i.e. 10, and then jumps onto the next line. We consider two more cases of the above snippet.

```
int a=10;
cout<<++a<<endl;
```

This code will output the value as 11 for variable 'a' after changing its value in the corresponding memory location.

```
int a=10;
cout<<a++<<endl;
```

The above code snippet will output the value as 10, and then change the value of variable 'a' in the memory location to 11. So the next time the value of 'a' is printed, it is going to be 11.

```
int a;
cin>>a;
```

The above code snippet is an example of taking input from the user. It is done through the "cin" statement. It is interesting to note the use of ">>" instead of "<<", as is done in the "cout" statement.

The "cin" statement can be used for anything from 'int' to 'float' to 'char'. It can also be used to take multiple inputs.

```
cin>>a>>b>>c;
```

Now let us look at a program implementing both the "cin" and the "cout" statements.

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

void main()
{
  clrscr();
  int roll;
  float sal;
  cout<<"Enter the roll number of the employee : ";
  cin>>roll;
  cout<<endl<<"Enter the salary : ";
  cin>>sal;

  clrscr();
  cout<<"ROLL NO. : "<<roll<<"   SALARY : "<<sal<<endl;
  getch();
}
```

The above program takes the roll number and salary of an employee from the user, and then displays it.

Chapter 4 : Conditional Statements

The "if" statement is used to execute an instruction or block of instructions only if a condition is satisfied. The general form of an 'if' statement is,

if (condition) statement

Here <condition> implies the expression which has to be evaluated and then checked. If the result is TRUE, the <statement> is executed or processed, else the compiler just ignores the statement, and continues with the next instruction. If there is a block of statements to be executed, they are to be enclosed within curly braces { and }. Let us take an example to make the point more clear :-

```
if(a==5)
cout<<"Variable A has a value of 5"<<endl;
```

This code will check whether variable 'a' has a value equal to 5, if yes, it prints the appropriate message with the help of the corresponding 'cout' statement. If no, then the compiler ignores the 'cout' statement and continues normal execution. A point to notice here is the position of the semicolon ';'. It is placed not after the 'if' statement, but after the statement which is to be executed. Also important to notice is the '==' used for checking. In C++, '=' is used as assignment operator, and '==' is used for checking condtions.

We now look at the 'if-else' statement. Take a look at the code below :-

```
if(a==5)
cout<<"Variable A has a value of 5"<<endl;
else
cout<<"A is not equal to 5"<<endl;
```

In this code, if the value of 'a' is not 5, then the compiler executes the 'else' clause, and thus prints the corresponding statement, which states that 'A is not equal to 5'. Notice the positions of the semi-colons here also.

Another interesting mod of such conditional statements is the 'if-else-if' statement. This will be more clear if we look at the code given below :-

```
if(a==5)
cout<<"Variable A has a value of 5"<<endl;
else if(a==10)
cout<<"A is equal to 10"<<endl;
```

```
else
cout<<"A is not equal to 5 or 10"<<endl;
```

The example given is pretty much self-explanatory. Just remember, you can add as many as else-if statements as required.

We now take a look at another important statement called the 'switch'. This is very often used for making menus in applications. Take a look at the code below :-

```
cout<<"1. Add numbers"<<endl;
cout<<"2. Substract numbers"<<endl;
cout<<"Enter your choice :";
cin>>choice;
switch(choice)
{ case 1 : result = a+b;
          break;
  case 2 : result = a-b;
}
```

When the user enters his choice (1 or 2), the switch statement checks the value. If value is 1, it adds the numbers, if the value is 2, it substracts the numbers. A statement to notice here is the 'break' statement. It is used to break out of a loop, such as the 'switch'. If the 'break' is not used here, both a+b and a-b will be carried out. Hence it is imperative to use 'break' in such cases.

Chapter 5 : Loops

Loops are basically means to do a task multiple times, without actually coding all statements over and over again. For example, loops can be used for displaying a string many times, for counting numbers and of course for displaying menus.

Loops in C++ are mainly of three types :-

1. 'while' loop
2. 'do while' loop
3. 'for' loop

The 'while' loop :-
Let me show you a small example of a program which writes ABC three(3) times.

```
#include<iostream.h>

void main()
{
  int i=0;
  while(i<3)

  {
    i++;
    cout<<"ABC"<<endl;
  }
}
```

The output of the above code will be :-
ABC
ABC
ABC

A point to notice here is that, for making it more easy to understand, we could also write,

```
int i=1;
while(i<=3)
```

This would make our code more easy for a newbie, but in actuality it doesn't make a difference either way.

The 'do while' loop :-
It is very similar to the 'while' loop shown above. The only difference being, in a 'while' loop, the condition is checked beforehand, but in a 'do while' loop, the condition is checked after one execution of the loop.

Example code for the same problem using a 'do while' loop would be :-

```
void main()
{
  int i=0;
  do
  {
    i++;
    cout<<"ABC"<<endl;
  }while(i<3);
}
```

The output would once again be same as in the above example.

The 'for' loop :-
This is probably the most useful and the most used loop in C/C++. The syntax is slightly more complicated than that of 'while' or the 'do while' loop.

The general syntax can be defined as :-
for(<initial value>;<condition>;<increment>)

To further explain the above code, we will take an example. Suppose we had to print the numbers 1 to 5, using a 'for' loop. The code for this would be :-

```
#include<iostream.h>

void main()
{
  for(int i=1;i<=5;i++)
  cout<<i<<endl;
}
```

The output for this code would be :-
1

2
3
4
5

Here variable 'i' is given an initial value of 1. The condition applied is till 'i' is less than or equal to 5. And for each iteration, the value of 'i' is also incremented by 1.

Notice here that if we wanted to print,
5
4
3
2
1

we would change our 'for' loop to :-
for(int i=5;i>=1;i--)

Chapter 6 : Arrays

An "array" is a group of variables having the same data type. The array grouping has one name, but can store logically related information in itself. Let us see, how we define a single-dimension array of integer type in C++,

int a[5];

The above statement makes an array of integer data type, that is, the elements stored in the array will be of integer type. The number of elements in the above array is 5,

a[0], a[1], a[2], a[3], a[4].

Let us see, how to input data into this array and then display it.

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

void main()
{
  clrscr();
  int a[5];

  cout<<"Enter the elements of the array"<<endl;
  for(int i=0;i<=4;i++)
  cin>>a[i];                    //Input of data

  clrscr();

  cout<<"The elements of the array are"<<endl;
  for(i=0;i<=4;i++)
  cout<<a[i]<<endl;             //Output of data

  getch();
}
```

Note that, if you declare an array of five elements, but supply only four initializing values, the fifth element isn't initialized, and so contains some garbage value.

Arrays do not have to be only single-dimension. They can be of any order, as long as you can program them. Let us take the example of a 2-D array, which is commonly used in representing matrices.

int mat[3][3];

Now this statement represents a 3x3 matrix, where the first element is mat[0][0] and the last element is mat[2][2]. Now we must remember that we may visualize them as a matrix with the first subscript representing rows and the second one columns, but in the memory of the computer they are stored as linear lists only.

We now focus our attention to strings. Strings ar not specific data-types as in Java, but are single-dimension character arrays. Take a look at the code snippet given below :-

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

void main()
{
  clrscr();
  char name[10];

  cout<<"Enter your name"<<endl;
  gets(name);

  clrscr();
  cout<<"Hello "<<name<<endl;

  getch();
}
```

The 'gets' function is used for entering strings. Here we can also use the 'cin' statement. Outputting strings is done using the 'cout' statement.

Chapter 7 : Functions

A function, sometimes also referred to as a procedure or a sub-routine, is a group of statements which carry out a task. Functions are an important concept in programming. In fact, Procedural Paradigm relies on nothing but functions. They are defined so that we do not have to repeat the same statements again and again.

A very important function that we have all seen is the 'main()' function. In fact, the '()' is what chracterizes 'main' as a function. We have also seen the use of 'clrscr()' and 'getch()'. Now, though we may call these functions, they are defined elsewhere, that is, in 'conio.h'.

Similarly, we shall now see, how to make a function in C++. Suppose we have an application that displays a menu to the user after every event, then we can make menu as a function. Take a look at the code below :-

```
void menu()
{
  clrscr();

  cout<<"1. Add a record"<<endl;
  cout<<"2. Delete a record"<<endl;
  cout<<"3. Modify a record"<<endl;
  cout<<"4. Exit"<<endl;

  cout<<endl<<"Enter your choice : ";

}

void main()
{
  menu();
  .
  .
  .
  menu();
}
```

Notice here that we made a function out of menu rather than writing the same output statements again and again. A point to notice here is that, before the function is called, we have defined it. What if, we want to define it after the main() but want to use it in the main() olny. Then we have to declare the

function before we use it, and can then write its deinition anywhere. For example,

```
void menu();

void main()
{
  menu();
  .
  .
  .
  menu();
}

void menu()
{
  clrscr();

  cout<<"1. Add a record"<<endl;
  cout<<"2. Delete a record"<<endl;
  cout<<"3. Modify a record"<<endl;
  cout<<"4. Exit"<<endl;

  cout<<endl<<"Enter your choice : ";

}
```

We now discuss two more important concepts in functions - arguments and return values. Arguments are parameters passed to the function for usage and return value is the value the function returns as an answer upon completion. Take a look at the program written below :-

```
#include <iostream.h>

int add(int a, int b)
{
  int result;
  result=a+b;
  return (result);
}

int main ()
{
```

```
  int r;
  r = add(5,8);
  cout<<"The result is "<<r;
  return 0;
}
```

Notice here that 'a' and 'b' are passed the value 5 and 8 respectively, hence are the arguments. The value of 'result' calculated is 13 and serves as a return value to the main() function. This value is put into 'r' and then is printed out on the screen. Notice that both add() and main() are of 'int' type. This depends upon what data type is their return value. In this program, both the functions return integer type values.

Chapter 8 : Structures

Structures are a way of combining many different variables of different types under the same name. A general format of a structure is :-

```
struct <name> {

  type1 element1;
  type2 element2;
  type3 element3;

}<structure variable>;
```

Here <name> stands for the name you want to give to the structure, whereas <structure variable> is the variable type or object of the structure. Remember that it is not necessary to write the <structure variable> immediately after the definition of the structure. Let us take a program which shows the use of structures with the above mentioned concepts,

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

struct database {
  int id;
  int age;
  float salary;
};

void main()
{
  clrscr();
  database person;

  person.age = 20;
  person.id = 123;
  person.salary = 8000;

  getch();
}
```

In the above code we make a structure called 'database' with three members - id, age and salary. We then define an object 'person' of the type 'database'. Notice how we assign the values to the different members using the dot (.) operator. The advantage of using structures is that it makes the program more

modular. Structures can also be nested within one another, that is, a structure can have another structure as its data member.

There is also another similar concept called 'unions'. They are similar to structures except for the fact that all the variables defined in the structure share the same memory. When a union is declared the compiler allocates enough memory for the largest data-type in the union.

A lengthy discussion of structures or unions is not important in C++, as these concepts are more important to C. C++ uses 'classes' instead of structures, which we will see later on.

Chapter 9 : Pointers

Pointers are variables that point to an area in memory. They do not contain actual data, but point to memory locations which hold the data.

To define a pointer, you just have to prefix an asterisk (*) before the variable name, for example,

int *ptr;

This will define a pointer 'ptr' which will hold the address of a memory location holding an integer number. Now suppose we have a integer number say 'num'. To make the pointer 'ptr' point to 'num', we have to write the following statement :-

ptr = #

The ampersand (&) sign should be read as 'the address of', and causes the address in memory of a variable to be returned, instead of the variable itself. Therefore now, 'ptr' starts pointing to the memory location of 'num'. The asterisk (*) should be read as 'the value pointed by'. So if, you directly want to change the value of 'num', you can do so with the help of the pointer 'ptr'. Just write,

*ptr = 8;

To get all these concepts clearer, we will now write the complete code :-

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

void main()
{
   clrscr();
   int num;
   int *ptr;


   num = 5;
   ptr = &num;

   cout<<"The value is : "<<num<<endl;

   *ptr = 8;
```

```
    cout<<"The new value is : "<<num<<endl;

    getch();
}
```

The output of the above code will be :-

```
The value is : 5
The new value is : 8
```

For the advanced programmer, a conceptual understanding of pointers is absolutely essential. They form the basis of resizeable arrays, dynamic memory allocation concepts and all of data strucutres including linked lists, stacks and queues. To allocate memory dynamically in you program, use the follwing lines of code :-

```
int *ptr;
ptr = new int;
```

The keyword 'new' is used to initialize pointers with memory from free store. Thus 'ptr' now points to its own exclusive memory location, of the size of an 'int' data type. And at the end of the program, or whenever required, we must free up the memory we have allocated using 'new'. This is done using the keyword 'delete', as shown,

```
delete ptr;
```

Now at this point we have to be very careful about freeing the memory space. Always pass a valid pointer to 'delete', otherwise it may lead undesirable circumstances like crashing.

Chapter 10 : Classes

A major change in C++ from C was the Object Oriented Programming approach (OOP). Instead of having structures, C++ has defined a new kind of data type called 'class'. A variable of the type class is called its 'object'. The classical definition of an object is that it is a runtime instance of a class. These definitions are derived from real-life objects and the classes of objects in the world surrounding us.

A class has the same format as that of a structure with one exception, it also has functions inside the class definition itself. The general format of a class is represented as,

```
class <name>
{
  data member;
  data member;

  public :

  member function;
  member function;

}<object name>;
```

Here 'public' means that the member functions written in the public mode can be called by the object directly, using the dot (.) operator. Whereas the private data members cannot be accessed directly by the object, but have to accessed with the help of member functions. This particluar OOP concept is called 'data hiding'. Take a look at the program given below :-

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

class car
{
  int num;
  char color[10];
  char model[15];
  float mileage;

  public :

  void inputdata()
```

```
  {
    clrscr();
    cout<<"Enter the car id number"<<endl;
    cin>>num;

    cout<<"Enter the color and model of the car"<<endl;
    cin>>color>>model;

    cout<<"Enter the mileage in km/l"<<endl;
    cin>>mileage;
  }

  void outputdata()
  {
    clrscr();
    cout<<"Car ID = "<<num<<endl;
    cout<<"Color = "<<color<<endl;
    cout<<"Model = "<<model<<endl;
    cout<<"Mileage = "<<mileage<<endl;
    getch();
  }
};

void main()
{
  clrscr();
  car obj;

  obj.inputdata();
  obj.outputdata();

  getch();
}
```

The above code is pretty much self explanatory. But for the heck of it, the 'obj' variable is the object of type 'car'. It has four data members and two member functions which access the data members and perform necessary manipulation. Take particular care about the public functions. Also note the semi colon at the end of the class definition.

Chapter 11 : Inheritance

Inheritance simply means one class deriving or inheriting the properties of another class. Here the parent class is known as 'base class' whereas the child class which gets the properties is known as 'derived class'. The properties that the derived class gets is the public data members and member functions.

Let's take an example to better understand the concept of inheritance :-

```
class automobile
{
  public :

  float price;
  int tyres;

  void purchase();
};

class car : public automobile
{
  int seats;

  public :

  void paint;
}

class truck : public automobile
{
  int capacity;

  public :

  void build();
}
```

The base class 'automobile' has two data members - price and tyres, since every automobile has these two (unless its a hovercraft ;-)). The two derived classes will inherit both the data members and the member function 'purchase ()'. Notice that all these are in public mode. Anything in private mode doesn't get inherited.

If you want to inherit certain members but also want them to be hidden (as in private), there is a separate mode called 'protected' mode. Members in the protected mode cannot be directly accessed by the object but can be inherited. Let us take an example to clarify this :-

```
class abc
{
  private :
  int a;

  protected :
  int b;

  public :
  int c;
}

class xyz : public xyz
{
  public :
  int x;
}
```

Now derived class 'xyz' has three members - x, b anc c. 'b' in this case will remain protected in the derived class also.

This leads to the concept of inheritance mode. So far while inheriting classes we used "class derived : public base", which means a public inheritance mode. In this mode all members which are inherited remain of the same visibility in the derived class. For example, 'b' in the above code remained protected. In private inheritance mode, all inherited members become private members of the derived class. Similarly, in protected inheritance mode, they become protected members of the derived class

Chapter 12 : File I/O

File I/O means reading and writing into files stored on secondary storage media like hard disks. An extensive knowledge of file I/O is absolutely essential for making and handling databases. A header file called 'fstream.h' has to be included for file operations.

For reading/writing to a file, it is necessary to open it first. The two most common methods of opening a file are - input mode which allows only read operations and output mode which allows only write operations. The syntax for opening a file are,

```
ifstream f1("file.dat");   // For input mode

ofstream f1("file.dat");   // For output mode
```

Here 'f1' is the stream name and 'file.dat' is the file required to be read or written to. After the read and write operations have been performed, it is very important to close the files. We do this using,

```
f1.close();
```

To understand the file handling operations, we take an example :-

```
#include<fstream.h>
#include<conio.h>

class person
{
  int age;
  char name[10];

  public :

  void input()
  {
    clrscr();
    cout<<"Enter the name and age of the person"<<endl;
    cin>>name>>age;
  }

  void output()
  {
    clrscr();
```

```
    cout<<"Name = "<<name<<endl;
    cout<<"Age = "<<age<<endl;
    getch();
  }

}obj;

void main()
{
  clrscr();

  obj.input();
  ofstream f1("abc.txt");
  f1.write((char *)&obj, sizeof(obj));
  f1.close();

  ifstream f2("abc.txt");
  f2.read((char *)&obj, sizeof(obj));
  f2.close();
  obj.output();

  getch();
}
```

Note the particular syntax of the read() and write() operations. The function sizeof() is used to calculate the size of the object to be written. Notice here that we do not include 'iostream.h' since that is already included in 'fstream.h'. Instead of the above read() and write() operations you can also write the following code,

```
f1<<obj;    // For writing object

f2>>obj;    // For reading object
```