Chapter 1

Introduction to C++

1.1 Introduction to object orientation

In the past, information systems used to be defined primarily by their functionality: data and functions were kept separate and linked together by means of input and output relations. The object-oriented approach, however, focuses on objects that represent abstract or concrete things of the real world. These objects are first defined by their character and their properties which are represented by their internal structure and their attributes (data). The behaviour of these objects is described by methods (functionality).
Objects form a capsule which combines the character to the respective behaviour. Objects should enable programmers to map a real problem and its proposed software solution on a one-to-one basis.

In short, **Object-oriented** or **object-orientation** is a software engineering concept, in which concepts are represented as "objects".

The prime purpose of C++ programming was to add object orientation to the C programming language, which is in itself one of the most powerful programming languages.

The core of the pure object-oriented programming is to create an object, in code, that has certain properties and methods. While designing C++ modules, we try to see whole world in the form of objects. For example a car is an object which has certain properties such as color, number of doors, and the like. It also has certain methods such as accelerate, brake, and so on.

1.2 Difference Between Structured & Object oriented Language

Structured Programming: Structured programming takes on the top-to-bottom approach. It splits the tasks into modular forms. This makes the program simpler and easier to read with less lines and codes. This type of program accomplishes certain tasks for that a specific reason. For example, invoice printers use structured programming. This type has clear, correct, precise descriptions.

Object Oriented programming: This type of programming uses sections in a program to perform certain tasks. It splits the program into objects that can be reused into other programs. They are small programs that can be used in other software. Each object or module has the data and the instruction of what to do with the data in it. This can be reused in other software directly.
Object oriented programming supports the following concepts

>Abstraction
>Encapsulation
>Inheritance
>Polymorphism

1.3 Introduction to C++

C++ is a powerful general-purpose programming language. It can be used to create small programs or large applications. It can be used to make CGI scripts or console-only DOS programs. C++ allows you to create programs to do almost anything you need to do. The creator of C++,Bjarne Stroustrup, has put together a partial list of applications written in C++.

1.3.1 Brief History

1. During 1970 Dennis Ritchie created C programming Language.
2. In the early 1980ŝs, also at Bell Laboratories , another programming language was created which was based upon the C language.
3. New language was developed by Bjarne Stroustrup and was called C++.
4. Stroustrup states that the purpose of C++ is to make writing good program easier and more pleasant for the individual programmer.
5. C++ programming language is extension to C language.
6. In C we have already used increment operator(++). Therefore we called C++ as **"Incremented C"** means **extension to C.**

1.3.2 Versions of C++

There are several versions of C++ programming language

Visual C++
Borland C++
Turbo C++
Standardize C++ [ANSI C++]

1.4 Program structure

C++ programming language is most popular programming language after C programming language. C++ is first object oriented programming language. We have summarize structure of C++ program in the following picture

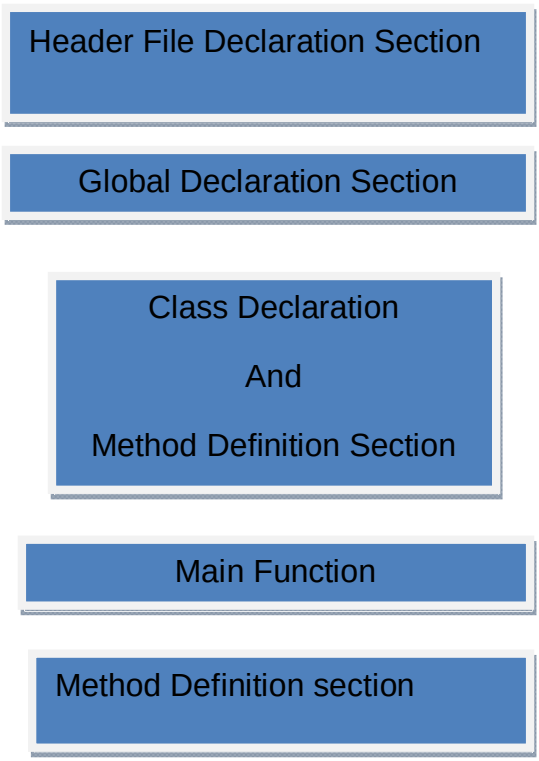


Fig 1

1.5 Program Design & Implementation Issues

A C++ program is a collection of commands, which tell the computer to do "something". This collection of commands is usually called **C++ source code**, **source code** or just **code**. Commands are either "functions" or "keywords". Keywords are a basic building block of the language, while functions are, in fact, usually written in terms of simpler functions--you'll see this in our very first program, below. (Confused? Think of it a bit like an outline for a book; the outline might show every chapter in the book; each chapter might have its own outline, composed of sections. Each section might have its own outline, or it might have all of the details written up.) Thankfully, C++ provides a

great many common functions and keywords that you can use.

But how does a program actually start? Every program in C++ has one function, always named **main**, that is always called when your program first executes. From main, you can also call other functions whether they are written by us or, as mentioned earlier, provided by the compiler.

So how do you get access to those prewritten functions? To access those standard functions that comes with the compiler, you include a header with the #include directive. What this does is effectively take everything in the header and paste it into your program. Let's look at a working program:

```
1   #include <iostream>
2
3   using namespace std;
4
5   int main()
6   {
7     cout<<"HEY, you, I'm alive! Oh, and Hello World!\n";
8     cin.get();
9   }
```

Let's look at the elements of the program. The #include is a "preprocessor" directive that tells the compiler to put code from the header called iostream into our program before actually creating the executable. By including header files, you gain access to many different functions. For example, the cout function requires iostream. Following the include is the statement, "using namespace std;". This line tells the compiler to use a group of functions that are part of the standard library (std). By including this line at the top of a file, you allow the program to use functions such as cout. The semicolon is part of the syntax of C++. It tells the compiler that you're at the end of a command. You will see later that the semicolon is used to end most commands in C++.

The next important line is int main(). This line tells the compiler that there is a function named main, and that the function returns an integer, hence int. The "curly braces" ({ and }) signal the beginning and end of functions and other code blocks. You can think of them as meaning BEGIN and END.

The next line of the program may seem strange. If you have programmed in another language, you might expect that print would be the function used to display text. In C++, however, the cout object is used to display text (pronounced "C out"). It uses the << symbols, known as "insertion operators", to indicate what to output. cout<< results in a function call with the ensuing text as an argument to the function. The quotes tell the compiler that you want to output the literal string as-is. The '\n' sequence is actually treated as a single character that stands for a newline (we'll talk about this later in more detail). It moves the cursor on your screen to the next line. Again, notice the semicolon: it is added onto the end of most lines, such as function calls, in C++.

The next command is cin.get(). This is another function call: it reads in input and

expects the user to hit the return key. Many compiler environments will open a new console window, run the program, and then close the window. This command keeps that window from closing because the program is not done yet because it waits for you to hit enter. Including that line gives you time to see the program run.

Upon reaching the end of main, the closing brace, our program will return the value of 0 (and integer, hence why we told main to return an int) to the operating system. This return value is important as it can be used to tell the OS whether our program succeeded or not. A return value of 0 means success and is returned automatically (but only for main, other functions require you to manually return a value), but if we wanted to return something else, such as 1, we would have to do it with a return statement:

```
1     #include <iostream>
2
3     using namespace std;
4
5     int main()
6     {
7       cout<<"HEY, you, I'm alive! Oh, and Hello World!\n";
8       cin.get();
9
10      return 1;
11    }
```

The final brace closes off the function. You should try compiling this program and running it.

1.6 Character set of C++

The C++ supports a group of characters as listed below:-

1. Digits 0-9
2. Alphabets
   i) Lower case letters a-z
   ii) Upper case letters A-Z
3. Special characters +, -, *, /, !, @, #, $, %, &, ǫ <, >, ?, /, \, :, ; ...

1.7 C++ Basic Elements

Programming language is a set of rules, symbols, and special words used to construct programs. There are certain elements that are common to all programming languages. Now, we will discuss these elements in brief :

1.7.1 C++ Character Set

Character set is a set of valid characters that a language can recognize.

Letters	A-Z, a-z
Digits	0-9
Special Characters	Space + - * / ^ \ () [] {} = != <> ó ñ $, ; : % ! & ? _ # <= >= @
Formatting characters	backspace, horizontal tab, vertical tab, form feed, and carriage return

1.7.2 Tokens

A token is a group of characters that logically belong together. The programmer can write a program by using tokens. C++ uses the following types of tokens.
Keywords, Identifiers, Literals, Punctuators, Operators.

1.7.2.1 Keywords

These are some reserved words in C++ which have predefined meaning to compiler called keywords..e.g. void, include etc.

1.7.2.2 Identifiers

Symbolic names can be used in C++ for various data items used by a programmer in his program. A symbolic name is generally known as an identifier. The identifier is a sequence of characters taken from C++ character set. The rule for the formation of an identifier are:

An identifier can consist of alphabets, digits and/or underscores.
It must not start with a digit

C++ is case sensitive that is upper case and lower case letters are considered different from each other.

It should not be a reserved word.

1.7.2.3 Literals

Literals (often referred to as constants) are data items that never change their value during the execution of the program. The following types of literals are available in C++.

Integer-Constants

Character-constants

Floating-constants

Strings-constants

1.7.2.4 Punctuators

The following characters are used as punctuators in C++.

Brackets [ ]	Opening and closing brackets indicate single and multidimensional array subscript.
Parentheses ( )	Opening and closing brackets indicate functions calls,; function parameters for grouping expressions etc.
Braces { }	Opening and closing braces indicate the start and end of a compound statement.
Comma ,	It is used as a separator in a function argument list.
Semicolon ;	It is used as a statement terminator.
Colon :	It indicates a labeled statement or conditional operator symbol.
Asterisk *	It is used in pointer declaration or as multiplication operator.
Equal sign =	It is used as an assignment operator.
Pound sign #	It is used as pre-processor directive.

1.7.2.5 Operators

Operators are special symbols used for specific purposes. C++ provides six types of operators. Arithmetical operators, Relational operators, Logical operators, Unary operators, Assignment operators, Conditional operators, Comma operator

.

1.8 Structure of a program

The best way to learn a programming language is by writing programs. Typically, the first program beginners write is a program called "Hello World", which simply prints "Hello World" to your computer screen. Although it is very simple, it contains all the fundamental components C++ programs have:

```
1 // my first program in C++
2 #include <iostream>
3
4 int main()
5 {
6   std::cout << "Hello World!";
7 }
```

```
Hello World!
```

The left panel above shows the C++ code for this program. The right panel shows the result when the program is executed by a computer. The grey numbers to the left of the panels are line numbers to make discussing programs and researching errors easier. They are not part of the program.

Let's examine this program line by line:

Line 1: // my first program in C++

Two slash signs indicate that the rest of the line is a comment inserted by the programmer but which has no effect on the behavior of the program. Programmers use them to include short explanations or observations concerning the code or program. In this case, it is a brief introductory description of the program.

Line 2: #include <iostream>

Lines beginning with a hash sign (#) are directives read and interpreted by what is known as the *preprocessor*. They are special lines interpreted before the compilation of the program itself begins. In this case, the directive #include <iostream>, instructs the preprocessor to include a section of standard C++ code, known as *header iostream*, that allows to perform standard input and output operations, such as writing the output of this program (Hello World) to the screen.

Line 3: A blank line.

Blank lines have no effect on a program. They simply improve readability of the code.

Line 4: int main ()

This line initiates the declaration of a function. Essentially, a function is a group of code statements which are given a name: in this case, this gives the name "main" to the group of code statements that follow. Functions will be discussed in detail in a later chapter, but essentially, their definition is introduced with a succession of a type (int), a name (main) and a pair of parentheses (()), optionally including parameters.

The function named main is a special function in all C++ programs; it is the function called when the program is run. The execution of all C++ programs begins with the main function, regardless of where the function is actually located within the code.

Lines 5 and 7: { and }

The open brace ({) at line 5 indicates the beginning of main's function definition, and the closing brace (}) at line 7, indicates its end. Everything between these braces is the function's body that defines what happens when main is called. All functions use braces to indicate the beginning and end of their definitions.

Line 6: std::cout << "Hello World!";

This line is a C++ statement. A statement is an expression that can actually produce some effect. It is the meat of a program, specifying its actual behavior. Statements are executed in the same order that they appear within a function's body.

This statement has three parts: First, std::cout, which identifies the **st**andar**d c**haracter **out**put device (usually, this is the computer screen). Second, the insertion operator (<<), which indicates that what follows is inserted into std::cout. Finally, a sentence within quotes ("Hello world!"), is the content inserted into the standard output.

Notice that the statement ends with a semicolon (;). This character marks the end of the statement, just as the period ends a sentence in English. All C++ statements must end with a semicolon character. One of the most common syntax errors in C++ is forgetting to end a statement with a semicolon.

You may have noticed that not all the lines of this program perform actions when the code is executed. There is a line containing a comment (beginning with //). There is a line with a directive for the preprocessor (beginning with#). There is a line that defines a function (in this case, the main function). And, finally, a line with a statements ending with a semicolon (the insertion into cout), which was within the block delimited by the braces ({ }) of the main function.

The program has been structured in different lines and properly indented, in order to make it easier to understand for the humans reading it. But C++ does not have strict rules on indentation or on how to split instructions in different lines.

For example, instead of

```
1 int main ()
2 {
3   std::cout << " Hello World!";
4 }
```

We could have written:

```
int main () { std::cout << "Hello World!"; }
```

all in a single line, and this would have had exactly the same meaning as the preceding

code.

In C++, the separation between statements is specified with an ending semicolon (;), with the separation into different lines not mattering at all for this purpose. Many statements can be written in a single line, or each statement can be in its own line. The division of code in different lines serves only to make it more legible and schematic for the humans that may read it, but has no effect on the actual behavior of the program.

Now, let's add an additional statement to our first program:

```
1 // my second program in C++
2 #include <iostream>
3
4 int main ()
5 {
6   std::cout << "Hello World! ";
7   std::cout << "I'm a C++ program";
8 }
```

```
Hello World! I'm a C++ program
```

In this case, the program performed two insertions into std::cout in two different statements. Once again, the separation in different lines of code simply gives greater readability to the program, since main could have been perfectly valid defined in this way:

```
int main () { std::cout << " Hello World! "; std::cout << " I'm a C++ program "; }
```

The source code could have also been divided into more code lines instead:

```
1 int main ()
2 {
3   std::cout <<
4     "Hello World!";
5   std::cout
6     << "I'm a C++ program";
7 }
```

And the result would again have been exactly the same as in the previous examples.

Preprocessor directives (those that begin by #) are out of this general rule since they

are not statements. They are lines read and processed by the preprocessor before proper compilation begins. Preprocessor directives must be specified in their own line and, because they are not statements, do not have to end with a semicolon (;).

1.8.1 Comments

As noted above, comments do not affect the operation of the program; however, they provide an important tool to document directly within the source code what the program does and how it operates.

C++ supports two ways of commenting code:

```
1 // line comment
2 /* block comment */
```

The first of them, known as *line comment*, discards everything from where the pair of slash signs (//) are found up to the end of that same line. The second one, known as *block comment*, discards everything between the /*characters and the first appearance of the */ characters, with the possibility of including multiple lines.

Let's add comments to our second program:

```
1  /* my second program in C++
2     with more comments */
3
4  #include <iostream>
5
6  int main ()
7  {
8    std::cout << "Hello World! ";     // prints Hello World!
9    std::cout << "I'm a C++ program"; // prints I'm a C++
10 program
   }
```

```
Hello World! I'm a C++ program
```

If comments are included within the source code of a program without using the comment characters combinations//, /* or */, the compiler takes them as if they were C++ expressions, most likely causing the compilation to fail with one, or several, error messages.

1.8.2 Using namespace std

If you have seen C++ code before, you may have seen cout being used instead of std::cout. Both name the same object: the first one uses its *unqualified name* (cout), while the second qualifies it directly within the *namespace* std(as std::cout).

cout is part of the standard library, and all the elements in the standard C++ library are declared within what is a called a *namespace*: the namespace std.

In order to refer to the elements in the std namespace a program shall either qualify each and every use of elements of the library (as we have done by prefixing cout with std::), or introduce visibility of its components. The most typical way to introduce visibility of these components is by means of *using declarations*:

```
using namespace std;
```

The above declaration allows all elements in the std namespace to be accessed in an *unqualified* manner (without the std:: prefix).

With this in mind, the last example can be rewritten to make unqualified uses of cout as:

```
1 // my second program in C++
2 #include <iostream>
3 using namespace std;
4
5 int main ()
6 {
7   cout << "Hello World! ";
8   cout << "I'm a C++ program";
9 }
```

```
Hello World! I'm a C++ program
```

1.9 What is meant by an object?

ÅObjects are the basic run-time entities in an object-oriented system.
Å They may represent a person, a place, , a table of data or any item that the program must handle.
ÅThey may also represent user defined data such as vectors, time and lists.
ÅWhen a program is executed, the objects interact by sending message to one another.

ÅEach object contains data and code to manipulate the data.
ÅObjects can interact without knowing having to know details of each otherôs data or code.

1.9.1 Classes

1. A class is a collection of objects of same type.
2. Once the class has been defined , we can create any number of objects belonging to that class.
3. 3.Each object is associated with the data of type class with which they are created.

1.9.2 Encapsulation

. The wrapping up of data and functions into a single unit (called class) is known as encapsulation
ÅData encapsulation is the most striking feature of a class.
The data is not accessible to the outside world and only those functions which are wrapped in the class can access it.
ÅThese functions provide the interface between the objectôs data and the program
ÅThis insulation of the data from direct access by the program is called **data hiding**

1.9.3 Inheritance.

ÅInheritance is the process by which objects of one class acquire the properties of objects of another class.
ÅIt provides the idea of reusability.
Å This is possible by deriving a new class from the existing class . The new class will have the combined features of both the classes.

1.9.4 Polymorphism

ÅIt means the ability to take more than one form.
ÅFor ex, consider the operation of addition.
ÅIf the operands are strings, then the operation would produce a third string by concatenation.
ÅIf the operands are numbers, it will generate a sum.

Some examples of classes:

```
1  // classes example
2  #include <iostream>
3  using namespace std;
4
5  class Rectangle {
6      int width, height;
7    public:
8      void set_values (int,int);
9      int area() {return width*height;}
10 };
11
12 void Rectangle::set_values (int x, int y) {
13   width = x;
```

```
area: 12
```

```
  height = y;
}

int main () {
  Rectangle rect;
  rect.set_values (3,4);
  cout << "area: " << rect.area();
  return 0;
}
```

```
// derived classes
#include <iostream>
using namespace std;

class Polygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b;}
 };

class Rectangle: public Polygon {
  public:
    int area ()
      { return width * height; }
 };

class Triangle: public Polygon {
  public:
    int area ()
      { return width * height / 2; }
  };

int main () {
  Rectangle rect;
  Triangle trgl;
  rect.set_values (4,5);
  trgl.set_values (4,5);
  cout << rect.area() << '\n';
  cout << trgl.area() << '\n';
  return 0;
}
```

```
20
10
```

Points to remember:

Object-oriented or **object-orientation** is a software engineering concept, in which concepts are represented as "objects.

Structured programming takes on the top-to-bottom approach. It splits the tasks into modular forms.

C++ is a powerful general-purpose programming language. It can be used to create small programs or large applications. It can be used to make CGI scripts or console-only DOS programs.
C++ was developed by Bjarne Stroustrup.
A C++ program is a collection of commands, which tell the computer to do "something". This collection of commands is usually called **C++ source code**, **source code** or just **code**
Character set is a set of valid characters that a language can recognize.
A token is a group of characters that logically belong together.
Rreserved words in C++ which have predefined meaning to compiler called keywords.
Literals (often referred to as constants) are data items that never change their value during the execution of the program.
Operators are special symbols used for specific purposes. C++ provides six types of operators.

Exercise

1. **Give one word Answer of the following statements:**
    a. a software engineering concept, in which concepts are represented as "objects".
    b. Every program in C++ has one function, which is always called when your program first executes.
    c. Rreserved words in C++ which have predefined meaning to compiler.
    d. A group of characters that logically belong together.
    e. The process by which objects of one class acquire the properties of objects of another class.
2. **Short Answer type Questions**
    a. Briefly explain the structure of typical C++ program
    b. What is comment?
    c. List out C++ main character set.
    d. What is meant by an object?
    e. What is anI dentifier?
3. **Long Answer type Questions**
    a. Explain the role of Punctuators in C++ language.
    b. How C++ came in to existence ?Explain.
    c. Explain the concept of object orientation in C++.

Chapter 2

DATA TYPES, VARIABLES AND CONSTANTS

2.1 Concepts of Data Types

While doing programming in any programming language, you need to use various variables to store various information. Variables are nothing but reserved memory locations to store values. This means that when you create a variable you reserve some space in memory.

You may like to store information of various data types like character, wide character, integer, floating point, double floating point, boolean etc. Based on the data type of a variable, the operating system allocates memory and decides what can be stored in the reserved memory.

2.1.1 Primitive Built-in Types:

C++ offer the programmer a rich assortment of built-in as well as user defined data types. Following table lists down seven basic C++ data types:

Type	Keyword
Boolean	bool
Character	char
Integer	int
Floating point	float
Double floating point	double
Valueless	void
Wide character	wchar_t

Several of the basic types can be modified using one or more of these **type modifiers**:

signed

unsigned

short

long

The following table shows the variable type, how much memory it takes to store the value in memory, and what is maximum and minimum value which can be stored in such type of variables.

Type	Typical Bit Width	Typical Range
char	1byte	-127 to 127 or 0 to 255
unsigned char	1byte	0 to 255
signed char	1byte	-127 to 127
int	4bytes	-2147483648 to 2147483647
unsigned int	4bytes	0 to 4294967295
signed int	4bytes	-2147483648 to 2147483647
short int	2bytes	-32768 to 32767
unsigned short int	Range	0 to 65,535
signed short int	Range	-32768 to 32767
long int	4bytes	-2,147,483,647 to 2,147,483,647
signed long int	4bytes	same as long int
unsigned long int	4bytes	0 to 4,294,967,295
float	4bytes	+/- 3.4e +/- 38 (~7 digits)
double	8bytes	+/- 1.7e +/- 308 (~15 digits)
long double	8bytes	+/- 1.7e +/- 308 (~15 digits)
wchar_t	2 or 4 bytes	1 wide character

The sizes of variables might be different from those shown in the above table, depending on the compiler and the computer you are using.

There are following basic types of variable in C++ as explained above:

Type	Description
bool	Stores either value true or false.
char	Typically a single octet(one byte). This is an integer type.
int	The most natural size of integer for the machine.
float	A single-precision floating point value.

double	A double-precision floating point value.
void	Represents the absence of type.
wchar_t	A wide character type.

2.2 Data Type Modifiers

C++ allows the **char, int,** and **double** data types to have modifiers preceding them. A modifier is used to alter the meaning of the base type so that it more precisely fits the needs of various situations.

The data type modifiers are listed here:

signed

unsigned

long

short

The modifiers **signed, unsigned, long,** and **short** can be applied to integer base types. In addition, **signed** and **unsigned** can be applied to char, and **long** can be applied to double.
The modifiers **signed** and **unsigned** can also be used as prefix to **long** or **short** modifiers. For example, **unsigned long int**.
C++ allows a shorthand notation for declaring **unsigned, short,** or **long** integers. You can simply use the word **unsigned, short,** or **long**, without the int. The int is implied. For example, the following two statements both declare unsigned integer variables.

```
unsigned x;
unsigned int y;
```

To understand the difference between the way that signed and unsigned integer modifiers are interpreted by C++, you should run the following short program:

```
#include <iostream>
using namespace std;

/* This program shows the difference between
 * signed and unsigned integers.
*/
int main()
{
   short int i;           // a signed short integer
```

```
   short unsigned int j;  // an unsigned short integer

   j = 50000;

   i = j;
   cout << i << " " << j;

   return 0;
}
```

When this program is run, following is the output:

```
-15536 50000
```

The above result is because the bit pattern that represents 50,000 as a short unsigned integer is interpreted as -15,536 by a short.

2.3 Constants

Constants refer to fixed values that the program may not alter and they are called **literals**.

Constants can be of any of the basic data types and can be divided into Integer Numerals, Floating-Point Numerals, Characters, Strings and Boolean Values.

Again, constants are treated just like regular variables except that their values cannot be modified after their definition.

2.3.1 Integer literals:

An integer literal can be a decimal, octal, or hexadecimal constant. A prefix specifies the base or radix: 0x or 0X for hexadecimal, 0 for octal, and nothing for decimal.

An integer literal can also have a suffix that is a combination of U and L, for unsigned and long, respectively. The suffix can be uppercase or lowercase and can be in any order.

Here are some examples of integer literals:

```
212         // Legal
215u        // Legal
0xFeeL      // Legal
078         // Illegal: 8 is not an octal digit
032UU       // Illegal: cannot repeat a suffix
```

2.3.2 Floating-point literals:

A floating-point literal has an integer part, a decimal point, a fractional part, and an exponent part. You can represent floating point literals either in decimal form or exponential form.

While representing using decimal form, you must include the decimal point, the exponent, or both and while representing using exponential form, you must include the integer part, the fractional part, or both. The signed exponent is introduced by e or E.

Here are some examples of floating-point literals:

```
3.14159       // Legal
314159E-5L    // Legal
510E          // Illegal: incomplete exponent
210f          // Illegal: no decimal or exponent
.e55          // Illegal: missing integer or fraction
```

2.3.3 Boolean literals:

There are two Boolean literals and they are part of standard C++ keywords:

A value of **true** representing true.
A value of **false** representing false.

You should not consider the value of true equal to 1 and value of false equal to 0.

2.3.4 Character literals:

Character literals are enclosed in single quotes. If the literal begins with L (uppercase only), it is a wide character literal (e.g., L'x') and should be stored in **wchar_t** type of variable . Otherwise, it is a narrow character literal (e.g., 'x') and can be stored in a simple variable of **char** type.

A character literal can be a plain character (e.g., 'x'), an escape sequence (e.g., '\t'), or a universal character (e.g., '\u02C0').

There are certain characters in C++ when they are preceded by a backslash they will have special meaning and they are used to represent like newline (\n) or tab (\t). Here, you have a list of some of such escape sequence codes:

Escape sequence	Meaning
\\	\ character
\'	' character
\"	" character
\?	? character
\a	Alert or bell

\b	Backspace
\f	Form feed
\n	Newline
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab
\ooo	Octal number of one to three digits
\xhh . . .	Hexadecimal number of one or more digits

Following is the example to show few escape sequence characters:

```
#include <iostream>
using namespace std;

int main()
{
   cout << "Hello\tWorld\n\n";
   return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Hello   World
```

2.4 String literals:

String literals are enclosed in double quotes. A string contains characters that are similar to character literals: plain characters, escape sequences, and universal characters.

You can break a long line into multiple lines using string literals and separate them using whitespaces.

Here are some examples of string literals. All the three forms are identical strings.

```
"hello, dear"

"hello, \

dear"

"hello, " "d" "ear"
```

2.5 Variables

Definition
"**Variable is a memory location inC++ Programming language**"
Variable are used to store data on memory.

Why we use variables in C++ language?
Variables are used to store value and that values can be change.
The values of variables can be numeric or alphabet.
There are certain rules on choosing variable name:

- Variable name can consist of letter, alphabets and start with underscore character.
- First character of variable should always be alphabet and cannot be digit.
- Blank spaces are not allowed in variable name.
- Special characters like #, $ are not allowed.
- A single variable can only be declared for only 1 data type in a program.
- As C++ is case sensitive language so if we declare a variable name and one more NAME both are two different variables.
- C++ has certain keywords which cannot be used for variable name.
- A variable name can be consist of 31 characters only if we declare a variable more than 1 characters compiler will ignore after 31 characters

2.5.1 Declaration of Variables

C++ is a strongly-typed language, and requires every variable to be declared with its type before its first use. This informs the compiler the size to reserve in memory for the variable and how to interpret its value. The syntax to declare a new variable in C++ is straightforward: we simply write the type followed by the variable name (i.e., its identifier).

A variable declaration has the form:
type identifier-list;
type specifies the type of the variables being declared. The *identifier-list* is a list of the identifiers of the variables being declared, separated by commas.

For example:

```
1 int a;
2 float mynumber;
```

These are two valid declarations of variables. The first one declares a variable of type int with the identifier a. The second one declares a variable of type float with the

identifier mynumber. Once declared, the variables a andmynumber can be used within the rest of their scope in the program.
If declaring more than one variable of the same type, they can all be declared in a single statement by separating their identifiers with commas. For example:

```
int a, b, c;
```

his declares three variables (a, b and c), all of them of type int, and has exactly the same meaning as:

```
int a;
int b;
int c;
```

To see what variable declarations look like in action within a program, let's have a look at the entire C++ code of the example about your mental memory proposed at the beginning of this chapter:

```
// operating with variables

#include <iostream>
using namespace std;

int main ()
{
  // declaring variables:
  int a, b;
  int result;

  // process:
  a = 5;
  b = 2;
  a = a + 1;
  result = a - b;

  // print out the result:
  cout << result;

  // terminate the program:
  return 0;
}
```

```
4
```

2.5.2 Initialization of Variables

When the variables in the example above are declared, they have an undetermined value until they are assigned a value for the first time. But it is possible for a variable to have a specific value from the moment it is declared. This is called the *initialization* of the variable.

In C++, there are three ways to initialize variables. They are all equivalent and are reminiscent of the evolution of the language over the years:

The first one, known as *c-like initialization* (because it is inherited from the C language), consists of appending an equal sign followed by the value to which the variable is initialized:

type identifier = initial_value;
For example, to declare a variable of type int called x and initialize it to a value of zero from the same moment it is declared, we can write:

```
int x = 0;
```

2.6 Operators in C++

An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations. C++ is rich in built-in operators and provides the following types of operators:

Arithmetic Operators

Relational Operators

Logical Operators

Bitwise Operators

Assignment Operators

Misc Operators

Arithmetic Operators:

There are following arithmetic operators supported by C++ language:

Operator	Description	Example
+	Adds two operands	A + B will give 30

-	Subtracts second operand from the first	A - B will give -10
*	Multiplies both operands	A * B will give 200
/	Divides numerator by de-numerator	B / A will give 2
%	Modulus Operator and remainder of after an integer division	B % A will give 0
++	Increment operator, increases integer value by one	A++ will give 11
--	Decrement operator, decreases integer value by one	A-- will give 9

2.6.2 Relational Operators:

There are following relational operators supported by C++ language

Operator	Description	Example
==	Checks if the values of two operands are equal or not, if yes then condition becomes true.	(A == B) is not true.
!=	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(A != B) is true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(A > B) is not true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(A < B) is true.
>=	Checks if the value of left	(A >= B) is not true.

	operand is greater than or equal to the value of right operand, if yes then condition becomes true.	
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(A <= B) is true.

2.6.3 Logical Operators:

There are following logical operators supported by C++ language

Operator	Description	Example
&&	Called Logical AND operator. If both the operands are non-zero, then condition becomes true.	(A && B) is false.
||	Called Logical OR Operator. If any of the two operands is non-zero, then condition becomes true.	(A || B) is true.
!	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true, then Logical NOT operator will make false.	!(A && B) is true.

2.6.4 Bitwise operators (&, |, ^, ~, <<, >>)

Bitwise operators modify variables considering the bit patterns that represent the values they store.

operator	asm equivalent	description
&	AND	Bitwise AND
|	OR	Bitwise inclusive OR
^	XOR	Bitwise exclusive OR
~	NOT	Unary complement (bit inversion)

<<	SHL	Shift bits left
>>	SHR	Shift bits right

2.6.5 Assignment Operators:

There are following assignment operators supported by C++ language:

Operator	Description	Example
=	Simple assignment operator, Assigns values from right side operands to left side operand	C = A + B will assign value of A + B into C
+=	Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand	C += A is equivalent to C = C + A
-=	Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand	C -= A is equivalent to C = C - A
*=	Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand	C *= A is equivalent to C = C * A
/=	Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand	C /= A is equivalent to C = C / A
%=	Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand	C %= A is equivalent to C = C % A
<<=	Left shift AND assignment operator	C <<= 2 is same as C = C << 2
>>=	Right shift AND assignment operator	C >>= 2 is same as C = C >> 2
&=	Bitwise AND assignment operator	C &= 2 is same as C = C & 2

^=	bitwise exclusive OR and assignment operator	C ^= 2 is same as C = C ^ 2
|=	bitwise inclusive OR and assignment operator	C |= 2 is same as C = C | 2

2.6.6 Conditional ternary operator (?)

The conditional operator evaluates an expression, returning one value if that expression evaluates to true, and a different one if the expression evaluates as false. Its syntax is:

condition ? result1 : result2

If condition is true, the entire expression evaluates to result1, and otherwise to result2.

```
1 7==5 ? 4 : 3     // evaluates to 3, since 7 is not equal to 5.
2 7==5+2 ? 4 : 3   // evaluates to 4, since 7 is equal to 5+2.
3 5>3 ? a : b      // evaluates to the value of a, since 5 is greater than 3.
4 a>b ? a : b      // evaluates to whichever is greater, a or b.
```

2.7 Expressions and Statements

In Learn about Numbers we saw how to assign values to ints and floats. These assignment statements have a very simple structure.

Variable = Expression

We know what a variable is, i.e. somewhere to store the value of the expression, but just what is an expression? It's a statement that resolves to a value: either numeric or character.

What exactly is a Statement?

A statement is a building block of a program. For example the assignment statement

int a=10;

This is also an expression and has the value 10. If we write this code

int b= (a=10) ;

Both a and b will be set to the value 10.

2.8 Conditional Expression

A conditional expression is one which evaluates as true (a non zero value) or false (0). True is almost always 1 but any non zero value is also true! A zero value is false, anything else is true.

```
#include <iostream>

using namespace std;
int main()
{
   int a=10;
   int b=( a = 10) ;
   int c=( a==10) ;
   cout << "Value of b is " << b << endl ;
   cout << "Value of c is " << c << endl ;
   return 0;

}
```

Compile and run the example above. It should print out the following.

```
Value of b is 10
Value of c is 1
```

2.9 Operators Precedence in C++:

Operator precedence determines the grouping of terms in an expression. This affects how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has higher precedence than the addition operator:

For example x = 7 + 3 * 2; here, x is assigned 13, not 20 because operator * has higher precedence than +, so it first gets multiplied with 3*2 and then adds into 7.

Here, operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within an expression, higher precedence operators will be evaluated first.

Category	**Operator**	**Associativity**
Postfix	() [] -> . ++ - -	Left to right
Unary	+ - ! ~ ++ - - (type)* & sizeof	Right to left
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

Points to remember:

When you create a variable you reserve some space in memory.
C++ allows the **char, int,** and **double** data types to have modifiers preceding them.
Constants refer to fixed values that the program may not alter and they are called **literals**
Character literals are enclosed in single quotes
A floating-point literal has an integer part, a decimal point, a fractional part, and an exponent part.
The conditional operator evaluates an expression, returning one value if that expression evaluates to true, and a different one if the expression evaluates as false
Operator precedence determines the grouping of terms in an expression.

Exercise

1. Give one word answer of the following statements
    a. Reserved memory locations to store values.
    b. A building block of a program.
    c. A symbol that tells the compiler to perform specific mathematical or logical manipulations.
    d. It is used to alter the meaning of the base type so that it more precisely fits the needs of various situations.
    e. It contains characters that are similar to character literals: plain characters, escape sequences, and universal characters
2. Short Answer type questions:
    a. What are basic types of variable in C++?
    b. Explain the term type modifier.
    c. What is the difference between constant and variable?
    d. What is operator?
    e. What is conditional Expression?
3. Long answer type questions:
    a. Explain about the arithmetic operators and relational operators in detail.
    b. What do you mean by character literal? Also explain escape sequences.

.

Chapter 3

CONTROL STATEMENTS

Compound statement, is a group of statements that is treated by the compiler as if it were a single statement. Blocks begin with a { symbol, end with a } symbol, and the statements to be executed are placed in between. Blocks can be used any place where a single statement is allowed.

Here is an example of a block when writing the function main():

```
1   int main()
2   { // start a block
3
4       // multiple statements
5       int nValue = 0;
6       return 0;
7
8   } // end a block
```

Note: Nested blocks can be further discussed in this chapter

Conditional statements

A simple C++ statement is each of the individual instructions of a program, like the variable declarations and expressions seen in previous sections. They always end with a semicolon (;), and are executed in the same order in which they appear in a program.

But programs are not limited to a linear sequence of statements. During its process, a program may repeat segments of code, or take decisions and bifurcate. For that purpose, C++ provides flow control statements that serve to specify what has to be done by our program, when, and under which circumstances.

Many of the flow control statements explained in this section require a generic (sub) statement as part of its syntax. This statement may either be a simple C++ statement, - such as a single instruction, terminated with a semicolon (;) - or a compound statement. A compound statement is a group of statements (each of them terminated by its own semicolon), but all grouped together in a block, enclosed in curly braces: {} (As explained above)

{ statement1; statement2; statement3; }

The entire block is considered a single statement (composed itself of multiple sub statements). Whenever a generic statement is part of the syntax of a flow control statement, this can either be a simple statement or a compound statement.

Selection statements: if and else

The if keyword is used to execute a statement or block, if, and only if, a condition is fulfilled. Its syntax is:

if (condition) statement

Here, condition is the expression that is being evaluated. If this condition is true, statement is executed. If it is false, statement is not executed (it is simply ignored), and the program continues right after the entire selection statement.
For example, the following code fragment prints the message (x is 100), only if the value stored in the x variable is indeed 100:

```
1 if (x == 100)
2   cout << "x is 100";
```

If x is not exactly 100, this statement is ignored, and nothing is printed.

If you want to include more than a single statement to be executed when the condition is fulfilled, these statements shall be enclosed in braces ({}), forming a block:

```
1 if (x == 100)
2 {
3    cout << "x is ";
4    cout << x;
5 }
```

As usual, indentation and line breaks in the code have no effect, so the above code is equivalent to:

```
if (x == 100) { cout << "x is "; cout << x; }
```

Selection statements with if can also specify what happens when the condition is not fulfilled, by using the else keyword to introduce an alternative statement. Its syntax is:

if (condition) statement1 else statement2

where statement1 is executed in case condition is true, and in case it is not, statement2 is executed.

For example:

```
1 if (x == 100)
2   cout << "x is 100";
3 else
4   cout << "x is not 100";
```

This prints x is 100, if indeed x has a value of 100, but if it does not, and only if it does not, it prints x is not 100instead.
Several if + else structures can be concatenated with the intention of checking a range of values. For example:

```
1 if (x > 0)
2   cout << "x is positive";
3 else if (x < 0)
4   cout << "x is negative";
5 else
6   cout << "x is 0";
```

This prints whether x is positive, negative, or zero by concatenating two if-else structures. Again, it would have also been possible to execute more than a single statement per case by grouping them into blocks enclosed in braces: {}.

Nested if (Nested blocks)

Blocks can be nested inside of other blocks. As you have seen, the *if statement* executes a single statement if the condition is true. However, because blocks can be used anywhere a single statement can, we can instead use a block of statements to make the *if statement* execute multiple statements if the condition is true!

```
1     #include <iostream>
2     int main()
3     {
4         using namespace std;
5         cout << "Enter a number: ";
6         int nValue;
```

```
7       cin >> nValue;
8       if (nValue > 0)
9       { // start of nested block
10          cout << nValue << " is a positive number" << endl;
11          cout << "Double this number is " << nValue * 2 << endl;
12      } // end of nested block
13  }
```

If the user enters the number 3, this program prints:

```
3 is a positive number
Double this number is 6
```

Note that both statements inside the nested block executed when the if statement is true!

It is even possible to put blocks inside of blocks inside of blocks:

```
1   int main()
2   {
3       using namespace std;
4       cout << "Enter a number: ";
5       int nValue;
6       cin >> nValue;
7
8       if (nValue > 0)
9       {
10          if (nValue < 10)
11          {
12              cout << nValue << " is between 0 and 10" << endl;
13          }
14      }
```

```
15  }
```

There is no practical limit to how many nested blocks you can have. However, it is generally a good idea to try to keep the number of nested blocks to at most 3 (maybe 4) blocks deep. If your function has a need for more, it's probably time to break your function into multiple smaller functions!

Another selection statement: switch.

The syntax of the switch statement is a bit peculiar. Its purpose is to check for a value among a number of possible constant expressions. It is something similar to concatenating if-else statements, but limited to constant expressions. Its most typical syntax is:

```
switch (expression)
{
  case constant1:
     group-of-statements-1;
     break;
  case constant2:
     group-of-statements-2;
     break;
  .
  .
  .
  default:
     default-group-of-statements
}
```

It works in the following way: switch evaluates expression and checks if it is equivalent to constant1; if it is, it executes group-of-statements-1 until it finds the break statement. When it finds this break statement, the program jumps to the end of the entire switch statement (the closing brace).

If expression was not equal to constant1, it is then checked against constant2. If it is equal to this, it executesgroup-of-statements-2 until a break is found, when it jumps to the end of the switch.

Finally, if the value of expression did not match any of the previously specified constants (there may be any number of these), the program executes the statements included after the default: label, if it exists (since it is optional).

Both of the following code fragments have the same behavior, demonstrating the if-else equivalent of a switch statement:

switch example	if-else equivalent

switch (x) {
  case 1:
  cout << "x is 1";
  break;
  case 2:
  cout << "x is 2";
  break;
  default:
  cout << "value of x unknown";
  }
	if (x == 1) {
	  cout << "x is 1";
	}
	else if (x == 2) {
	  cout << "x is 2";
	}
	else {
	  cout << "value of x unknown";
	}

The switch statement has a somewhat peculiar syntax inherited from the early times of the first C compilers, because it uses labels instead of blocks. In the most typical use (shown above), this means that break statements are needed after each group of statements for a particular label. If break is not included, all statements following the case (including those under any other labels) are also executed, until the end of the switch block or a jump statement (such as break) is reached.

If the example above lacked the break statement after the first group for case one, the program would not jump automatically to the end of the switch block after printing x is 1, and would instead continue executing the statements in case two (thus printing also x is 2). It would then continue doing so until a break statement is encountered, or the end of the switch block. This makes unnecessary to enclose the statements for each case in braces {}, and can also be useful to execute the same group of statements for different possible values. For example:

```
1 switch (x) {
2   case 1:
3   case 2:
4   case 3:
5     cout << "x is 1, 2 or 3";
6     break;
7   default:
8     cout << "x is not 1, 2 nor 3";
9   }
```

Notice that switch is limited to compare its evaluated expression against labels that are constant expressions. It is not possible to use variables as labels or ranges, because they are not valid C++ constant expressions.

To check for ranges or values that are not constant, it is better to use concatenations of if and else if statements.

Jump statements

Jump statements allow altering the flow of a program by performing jumps to specific locations.

The break statement

break leaves a loop, even if the condition for its end is not fulfilled. It can be used to end an infinite loop, or to force it to end before its natural end. For example, let's stop the countdown before its natural end:

```
1  // break loop example
2  #include <iostream>
3  using namespace std;
4
5  int main ()
6  {
7    for (int n=10; n>0; n--)
8    {
9      cout << n << ", ";
10     if (n==3)
11     {
12       cout << "countdown aborted!";
13       break;
14     }
15   }
16 }
```

```
10, 9, 8, 7, 6, 5, 4, 3, countdown aborted!
```

The continue statement

The continue statement causes the program to skip the rest of the loop in the current iteration, as if the end of the statement block had been reached, causing it to jump to the start of the following iteration. For example, let's skip number 5 in our countdown:

```
1  // continue loop example
2  #include <iostream>
3  using namespace std;
4
5  int main ()
6  {
7    for (int n=10; n>0; n--) {
```

```
10, 9, 8, 7, 6, 4, 3, 2, 1, liftoff!
```

```
  8     if (n==5) continue;
  9     cout << n << ", ";
 10   }
 11   cout << "liftoff!\n";
 12 }
```

The goto statement

goto allows to make an absolute jump to another point in the program. This unconditional jump ignores nesting levels, and does not cause any automatic stack unwinding. Therefore, it is a feature to use with care, and preferably within the same block of statements, especially in the presence of local variables.

The destination point is identified by a *label*, which is then used as an argument for the goto statement. A *label* is made of a valid identifier followed by a colon (:).

goto is generally deemed a low-level feature, with no particular use cases in modern higher-level programming paradigms generally used with C++. But, just as an example, here is a version of our countdown loop using goto:

```
 1 // goto loop example
 2 #include <iostream>
 3 using namespace std;
 4
 5 int main ()
 6 {
 7   int n=10;
 8 mylabel:
 9   cout << n << ", ";
10   n--;
11   if (n>0) goto mylabel;
12   cout << "liftoff!\n";
13 }
```

```
10, 9, 8, 7, 6, 5, 4, 3, 2, 1, liftoff!
```

Exit() Function

Exit ends the program. The ExitCode is returned to the operating system, similar to returning a value to int main.
Example:

```
//Program exits itself
//Note that the example would terminate anyway
```

```
#include <iostream>

using namespace std;

int main()
{
   cout<<"Program will exit";
   exit(1); // Returns 1 to the operating system

   cout<<"Never executed";
}
```

Iteration Statements (C++)

Iteration statements cause statements (or compound statements) to be executed zero or more times, subject to some loop-termination criteria. When these statements are compound statements, they are executed in order, except when either the break statement or the continue statement is encountered.

Loop & Nested Loops

Loops repeat a statement a certain number of times, or while a condition is fulfilled. They are introduced by the keywords while, do, and for.

The while loop

The simplest kind of loop is the while-loop. Its syntax is:

while (expression) statement

The while-loop simply repeats statement while expression is true. If, after any execution of statement, expressionis no longer true, the loop ends, and the program continues right after the loop. For example, let's have a look at a countdown using a while-loop:

```
 1 // custom countdown using while
 2 #include <iostream>
 3 using namespace std;
 4
 5 int main ()
 6 {
 7   int n = 10;
 8
 9   while (n>0) {
10     cout << n << ", ";
```

```
10, 9, 8, 7, 6, 5, 4, 3, 2, 1, liftoff!
```

```
11    --n;
12  }
13
14  cout << "liftoff!\n";
15 }
```

The first statement in main sets *n* to a value of 10. This is the first number in the countdown. Then the while-loop begins: if this value fulfills the condition n>0 (that *n* is greater than zero), then the block that follows the condition is executed, and repeated for as long as the condition (n>0) remains being true
The whole process of the previous program can be interpreted according to the following script (beginning in main):

1. n is assigned a value
2. The while condition is checked (n>0). At this point there are two possibilities:
    - condition is true: the statement is executed (to step 3)
    - condition is false: ignore statement and continue after it (to step 5)
3. Execute statement:
   cout << n << ", ";
   --n;
   (prints the value of n and decreases n by 1)
4. End of block. Return automatically to step 2.
5. Continue the program right after the block:
   print liftoff! and end the program.

A thing to consider with while-loops is that the loop should end at some point, and thus the statement shall alter values checked in the condition in some way, so as to force it to become false at some point. Otherwise, the loop will continue looping forever. In this case, the loop includes --n, that decreases the value of the variable that is being evaluated in the condition (n) by one - this will eventually make the condition (n>0) false after a certain number of loop iterations. To be more specific, after 10 iterations, n becomes 0, making the condition no longer true, and ending the while-loop.

Note that the complexity of this loop is trivial for a computer, and so the whole countdown is performed instantly, without any practical delay between elements of the count.

The do-while loop

A very similar loop is the do-while loop, whose syntax is:

do statement while (condition);

It behaves like a while-loop, except that condition is evaluated after the execution of statement instead of before, guaranteeing at least one execution of statement, even if condition is never fulfilled. For example, the following example program echoes any text the user introduces until the user enters goodbye:

```
1 // echo machine
2 #include <iostream>
3 #include <string>
4 using namespace std;
5
6 int main ()
7 {
8   string str;
9   do {
10    cout << "Enter text: ";
11    getline (cin,str);
12    cout << "You entered: " << str << '\n';
13  } while (str != "goodbye");
14 }
```

```
Enter text: hello
You entered: hello
Enter text: who's there?
You entered: who's there?
Enter text: goodbye
You entered: goodbye
```

The do-while loop is usually preferred over a while-loop when the statement needs to be executed at least once, such as when the condition that is checked to end of the loop is determined within the loop statement itself. In the previous example, the user input within the block is what will determine if the loop ends. And thus, even if the user wants to end the loop as soon as possible by entering goodbye, the block in the loop needs to be executed at least once to prompt for input, and the condition can, in fact, only be determined after it is executed.

The for loop

The for loop is designed to iterate a number of times. Its syntax is:

for (initialization; condition; increase) statement;

Like the while-loop, this loop repeats statement while condition is true. But, in addition, the for loop provides specific locations to contain an initialization and an increase expression, executed before the loop begins the first time, and after each

iteration, respectively. Therefore, it is especially useful to use counter variables as condition.

It works in the following way:

1. initialization is executed. Generally, this declares a counter variable, and sets it to some initial value. This is executed a single time, at the beginning of the loop.
2. condition is checked. If it is true, the loop continues; otherwise, the loop ends, and statement is skipped, going directly to step 5.
3. statement is executed. As usual, it can be either a single statement or a block enclosed in curly braces { }.
4. increase is executed, and the loop gets back to step 2.
5. the loop ends: execution continues by the next statement after it.
6. Here is the countdown example using a for loop:

```
1 // countdown using a for loop
2 #include <iostream>
3 using namespace std;
4
5 int main ()
6 {
7   for (int n=10; n>0; n--) {
8     cout << n << ", ";
9   }
10  cout << "liftoff!\n";
11 }
```

```
10, 9, 8, 7, 6, 5, 4, 3, 2, 1, liftoff!
```

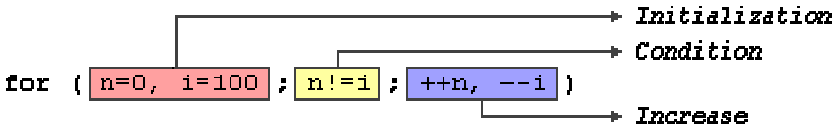
The three fields in a for-loop are optional. They can be left empty, but in all cases the semicolon signs between them are required. For example, for (;n<10;) is a loop without *initialization* or *increase* (equivalent to a while-loop); and for (;n<10;++n) is a loop with *increase*, but no *initialization* (maybe because the variable was already initialized before the loop). A loop with no *condition* is equivalent to a loop with true as condition (i.e., an infinite loop).

Because each of the fields is executed in a particular time in the life cycle of a loop, it may be useful to execute more than a single expression as any of *initialization*, *condition*, or *statement*. Unfortunately, these are not statements, but rather, simple expressions, and thus cannot be replaced by a block. As expressions, they can, however, make use of the comma operator (,): This operator is an expression separator, and can separate multiple expressions where only one is generally expected. For example, using it, it would be possible for a for loop to handle two counter variables,

initializing and increasing both:

```
1 for ( n=0, i=100 ; n!=i ; ++n, --i )
2 {
3   // whatever here...
4 }
```

This loop will execute 50 times if neither n or i are modified within the loop:



n starts with a value of 0, and i with 100, the condition is n!=i (i.e., that n is not equal to i). Because n is increased by one, and i decreased by one on each iteration, the loop's condition will become false after the 50th iteration, when both n and i are equal to 50

Nested Loops

When we write any loop statement within the another loop statement the that structure is called as nested loops

A loop can be nested inside of another loop. C++ allows at least 256 levels of nesting.

Syntax:

The syntax for a **nested for loop** statement in C++ is as follows:

```
for ( init; condition; increment )
{
   for ( init; condition; increment )
   {
      statement(s);
   }
   statement(s); // you can put more statements.
}
```

The syntax for a **nested while loop** statement in C++ is as follows:

```
while(condition)
```

```
{
   while(condition)
   {
      statement(s);
   }
   statement(s); // you can put more statements.
}
```

The syntax for a **nested do...while loop** statement in C++ is as follows:

```
do
{
   statement(s); // you can put more statements.
   do
   {
      statement(s);
   }while( condition );

}while( condition );
```

Example:

The following program uses a nested for loop to find the prime numbers from 2 to 100:

```
#include <iostream>
using namespace std;

int main ()
{
   int i, j;

   for(i=2; i<100; i++) {
      for(j=2; j <= (i/j); j++)
        if(!(i%j)) break; // if factor found, not prime
```

```
      if(j > (i/j)) cout << i << " is prime\n";
   }
   return 0;
}
```

This would produce the following result:

```
2 is prime
3 is prime
5 is prime
7 is prime
11 is prime
13 is prime
17 is prime
19 is prime
23 is prime
29 is prime
31 is prime
37 is prime
41 is prime
43 is prime
47 is prime
53 is prime
59 is prime
61 is prime
67 is prime
71 is prime
73 is prime
79 is prime
83 is prime
89 is prime
```

```
97 is prime
```

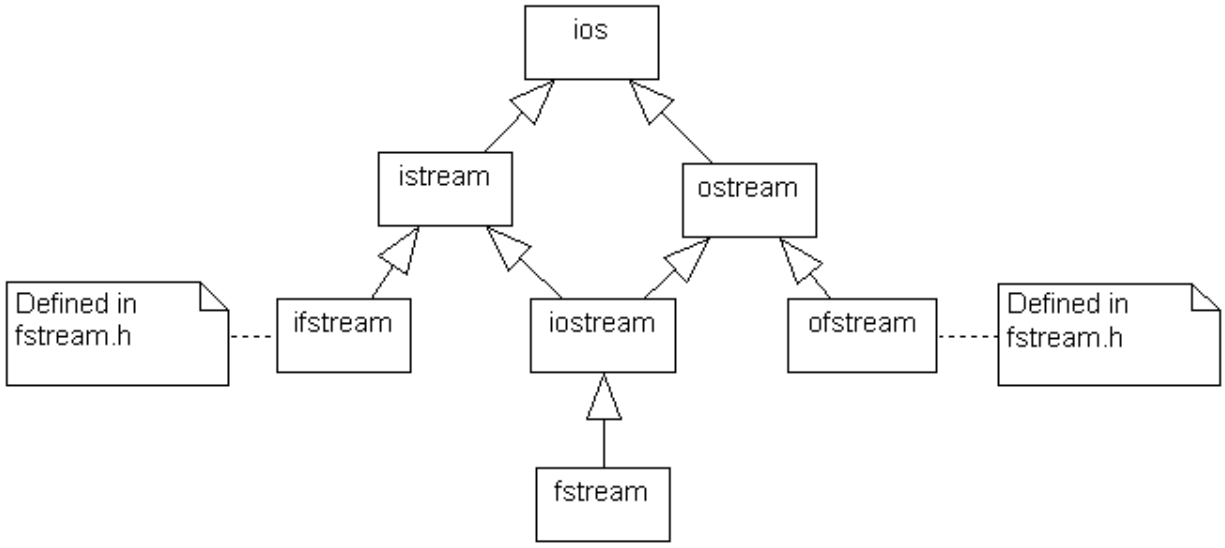
Console I/O functions

C and C++ handle I/O in very different ways

Streams:

- A stream can be thought of as a flow (or sequence) of objects (e.g., characters, bytes)
- Input involves taking objects from the stream
- Output involves adding objects to the stream

A Portion of the Class Hierarchy:



Common Practice:

- One generally includes either **<iostream>** or **<fstream>**, rather than one of their parents

Standard In and Out:

- If you only need to use "console I/O" or "standard in" and "standard out" you can use the streams **cin** and **cout**

Other Standard Output Files in **<iostream>**:

- **cerr**
- **clog**

C++ provides various formatted console I/O functions for formatting the output. They are of three types.

(1) ios class function and flags
(2) Manipulators
(3) User-defined output functions

The ios grants operations common to both input and output. The classes derived from ios are ***(istream, ostream, and iostream)*** special I/O with high-level formatting operations: The iostream ***class*** is automatically loaded in the program by the compiler.

(a) istream performs formatted input.
(b) ostream performs formatted output.
(c) iostream performs formatted input and output.

Header Files

Many programming languages and other computer files have a directive, often called **include** (as well as **copy** and **import**), that causes the contents of a second file to be inserted into the original file. These included files are called copybooks or header files. They are often used to define the physical layout of program data, pieces of procedural code and/or forward declarations while promoting encapsulation and the reuse of code.

Stdio.h: The C programming language provides many standard library functions for file input and output. These functions make up the bulk of the C standard library header <stdio.h>.The functionality descends from a "portable I/O package" written by Mike Lesk at Bell Labs in the early 1970s.[2]

Iostream.h: In the C++ programming language, **Input/output** library refers to a family of class templates and supporting functions in the C++ Standard Library that implement stream-based input/output capabilities. It is an object-oriented alternative to C's FILE-based streams from the C standard library.

getc() and putc Function

getc()

The getc() function returns the next character from the specified input stream and increment file position indicator. The character is read as an unsigned char that is converted to an integer.

Declaration:

```
int getc(FILE *stream);
```

Example:

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  int main()
4.      {
5.      FILE *fptr;
6.      char c;
7.      clrscr();
8.      if((fptr = fopen(ñTESTò,òrò))==NULL)
9.              {
10.             printf(ñCannot open file\nò);
11.             exit(1);
12.             }
13.     while((c=getc(fptr))!=EOF)
14.             putchar(c);
15.     if(fclose(fptr))
16.             pritf(ñFile close error\nò);
17.     getch();
18.     return 0;
19.     }
```

putc()

The putc() function writes the character ch to the specified stream at the current file position and then advance the file position indicator. Even though the ch is declared to be an int, it is converted by putc() into an unsigned char.

Declaration:

```
int putc(int ch, FILE *stream);
```

Example:

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  void main()
4.      {
5.      FILE *fptr;
6.      char text[100];
7.       int i=0;
8.       clrscr();
9.       printf(ñEnter a text:\nò);
10.      gets(text);
11.      if((fptr = fopen(ñTESTò,òwò))==NULL)
12.             {
13.             printf(ñCannot open file\nò);
14.             exit(1);
```

```
15.             }
16.       while(text[i]!=ð0ð)
17.       putc(text[i++],fptr);
18.       if(fclose(fptr))
19.             pritf(ñFile close error\nò);
20.       getch();
21.       }
```

gets() and puts() both are unformatted function.

gets() is used to read stdin into the character array pointed to by a string variable str until a newline character is found or end-of-file occurs. A null character is written immediately after the last character read into the array. It is defined in stdio.h header file.

gets: from standard input to memory

puts: from memory to standard input

Example :

```
#include<stdio.h>
void main( )
{
char name[10];
printf("What is your first and last name?");
gets(name);
puts(name);
}
```

Points to remember:

Compound statement, is a group of statements that is treated by the compiler as if it were a single statement.

C++ provides flow control statements that serve to specify what has to be done by our program, when, and under which circumstances.

Blocks can be nested inside of other blocks.

Jump statements allow altering the flow of a program by performing jumps to specific locations.

Break leaves a loop, even if the condition for its end is not fulfilled.

When we write any loop statement within the another loop statement the that structure is called as nested loops.

C++ provides various formatted console I/O functions for formatting the output.

gets() and puts() both are unformatted function.

Exercise

Fill in the blanks:

1. During its process, a program may _____ segments of code.
2. A compound statement is a _____.
3. _____allows to make an absolute jump to another point in the program.
4. The _____ grants operations common to both input and output
5. C++ allows at least _____ levels of nesting.

Short answer type questions:

1. What is a compound statement?
2. How many types of conditional statements can be used in C++?
3. Briefly explain the working of Break statement.
4. What is the role of Exit() function?
5. What is nested loop? Give syntax.
6. Explain I/O operations in respect with ios.
7. What is difference between gets() and puts() functions.
8. Give syntax of any one of the iteration statement.
9. Explain switch statement with example.
10. What is the role of jump statements in C++ programming?

Chapter 4

Functions

4.1 Definition of function

A function is a group of statements that together perform a task. Every C++ program has at least one function, which is **main()**, and all the most trivial programs can define additional functions.

You can divide up your code into separate functions. How you divide up your code among different functions is up to you, but logically the division usually is so each function performs a specific task.

A function **declaration** tells the compiler about a function's name, return type, and parameters. A function **definition** provides the actual body of the function.
The C++ standard library provides numerous built-in functions that your program can call. For example, function **strcat()** to concatenate two strings, function **memcpy()** to copy one memory location to another location and many more functions.

A function is knows as with various names like a method or a sub-routine or a procedure etc.

4.1.2 Defining a Function:

The general form of a C++ function definition is as follows:

```
return_type function_name( parameter list )
{
   body of the function
}
```

A C++ function definition consists of a function header and a function body. Here are all the parts of a function:

Return Type: A function may return a value. The **return_type** is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return_type is the keyword **void**.
Function Name: This is the actual name of the function. The function name and the parameter list together constitute the function signature.
Parameters: A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.
Function Body: The function body contains a collection of statements that define what the function does.

Example:

Following is the source code for a function called **max()**. This function takes two parameters num1 and num2 and returns the maximum between the two:

```
// function returning the max between two numbers

int max(int num1, int num2)
{
  // local variable declaration
  int result;

  if (num1 > num2)
    result = num1;
  else
    result = num2;

  return result;
}
```

4.2 Function Prototype

A function prototype is a declaration in C++ of a function, its name, parameters and return type. Unlike a full definition, the prototype terminates in a semi-colon.
e.g. int getsum(float * value) ;

4.2.1 Function Declaration

A function **declaration** tells the compiler about a function name and how to call the function. The actual body of the function can be defined separately.

A function declaration has the following parts:

```
return_type function_name( parameter list );
```

For the above defined function max(), following is the function declaration:

```
int max(int num1, int num2);
```

Parameter names are not importan in function declaration only their type is required, so following is also valid declaration:

```
int max(int, int);
```

Function declaration is required when you define a function in one source file and you call that function in another file. In such case, you should declare the function at the top of the file calling the function.

4.3 Objective of using function prototype

Prototypes are used in header files so that external functions in other files can be called and the compiler can check the parameters during compilation.

4.4 Accessing a function

While creating a C++ function, you give a definition of what the function has to do. To use a function, you will have to call or invoke that function.

When a program calls a function, program control is transferred to the called function. A called function performs defined task and when its return statement is executed or when its function-ending closing brace is reached, it returns program control back to the main program.

To call a function, you simply need to pass the required parameters along with function name, and if function returns a value, then you can store returned value. For example:

```
#include <iostream>
using namespace std;

// function declaration
int max(int num1, int num2);

int main ()
{
   // local variable declaration:
   int a = 100;
   int b = 200;
   int ret;

   // calling a function to get max value.
   ret = max(a, b);

   cout << "Max value is : " << ret << endl;

   return 0;
}
```

```
// function returning the max between two numbers
int max(int num1, int num2)
{
   // local variable declaration
   int result;

   if (num1 > num2)
      result = num1;
   else
      result = num2;

   return result;
}
```

While running final executable, it would produce the following result:

```
Max value is : 200
```

4.5 Passing argument to a function

If a function is to use arguments, it must declare variables that accept the values of the arguments. These variables are called the **formal parameters** of the function.

The formal parameters behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit.

While calling a function, there are two ways that arguments can be passed to a function:

Call Type	Description
Call by value	This method copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.
Call by pointer	This method copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.
Call by reference	This method copies the reference of an argument into the formal parameter. Inside the function, the reference is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.

*By default, C++ uses **call by value** to pass arguments. In general, this means that code within a function cannot alter the arguments used to call the function and above mentioned example while calling max() function used the same method.*

4.6 Default Values for Parameters:

When you define a function, you can specify a default value for each of the last parameters. This value will be used if the corresponding argument is left blank when calling to the function.

This is done by using the assignment operator and assigning values for the arguments in the function definition. If a value for that parameter is not passed when the function is called, the default given value is used, but if a value is specified, this default value is ignored and the passed value is used instead. Consider the following example:

```
#include <iostream>
using namespace std;

int sum(int a, int b=20)
{
  int result;

  result = a + b;

  return (result);
}

int main ()
{
   // local variable declaration:
   int a = 100;
   int b = 200;
   int result;

   // calling a function to add the values.
   result = sum(a, b);
   cout << "Total value is :" << result << endl;

   // calling a function again as follows.
   result = sum(a);
   cout << "Total value is :" << result << endl;

   return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Total value is :300
Total value is :120
```

4.7 THE const ARGUMENT

The constant variable can be declared using const keyword. The const keyword makes variable value stable. The constant variable should be initialized while declaring.

Syntax:

(a) const <variable name> = <value>;
(b) <function name> (const <type>*<variable name>;)
(c) int const x // in valid
(d) int const x =5 // valid

In statement ***(a),*** the const modifier assigns an initial value to a variable that cannot be changed later by the program.
For example,

const age = 40;

Any attempt to change the contents of const variable age will produce a compiler error. Using pointer, one can indirectly modify a const variable as shown below:

*(int *)&age = 45;

When the const variable is used with a pointer argument in a function's parameter list, the function cannot modify the variable that the pointer points to.

4.8 C++ function call by value

The **call by value** method of passing arguments to a function copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.
By default, C++ uses call by value to pass arguments. In general, this means that code within a function cannot alter the arguments used to call the function. Consider the function **swap()** definition as follows.

```
// function definition to swap the values.
void swap(int x, int y)
{
  int temp;

  temp = x; /* save the value of x */
  x = y;    /* put y into x */
  y = temp; /* put x into y */

  return;
}
```

Now, let us call the function **swap()** by passing actual values as in the following example:

```
#include <iostream>
```

```
using namespace std;

// function declaration
void swap(int x, int y);

int main ()
{
   // local variable declaration:
   int a = 100;
   int b = 200;

   cout << "Before swap, value of a :" << a << endl;
   cout << "Before swap, value of b :" << b << endl;

   // calling a function to swap the values.
   swap(a, b);

   cout << "After swap, value of a :" << a << endl;
   cout << "After swap, value of b :" << b << endl;

   return 0;
}
```

When the above code is put together in a file, compiled and executed, it produces the following result:

```
Before swap, value of a :100
Before swap, value of b :200
After swap, value of a :100
After swap, value of b :200
```

Which shows that there is no change in the values though they had been changed inside the function.

4.9 C++ function call by reference

The **call by reference** method of passing arguments to a function copies the reference of an argument into the formal parameter. Inside the function, the reference is used to access the actual argument used in the call. This means that changes made to the parameter affect the passed argument.
To pass the value by reference, argument reference is passed to the functions just like any other value. So accordingly you need to declare the function parameters as reference types as in the following function **swap()**, which exchanges the values of the two integer variables pointed to by its arguments.

```
// function definition to swap the values.
void swap(int &x, int &y)
```

```
{
   int temp;
   temp = x; /* save the value at address x */
   x = y;    /* put y into x */
   y = temp; /* put x into y */

   return;
}
```

For now, let us call the function **swap()** by passing values by reference as in the following example:

```
#include <iostream>
using namespace std;

// function declaration
void swap(int &x, int &y);

int main ()
{
   // local variable declaration:
   int a = 100;
   int b = 200;

   cout << "Before swap, value of a :" << a << endl;
   cout << "Before swap, value of b :" << b << endl;

   /* calling a function to swap the values using variable reference.*/
   swap(a, b);

   cout << "After swap, value of a :" << a << endl;
   cout << "After swap, value of b :" << b << endl;

   return 0;
}
```

When the above code is put together in a file, compiled and executed, it produces the following result:

```
Before swap, value of a :100
Before swap, value of b :200
After swap, value of a :200
After swap, value of b :100
```

4.10 Parameter pass by reference

Pass-by-reference means to pass the reference of an argument in the calling function to the corresponding formal parameter of the called function. The called function can modify the value of the argument by using its reference passed in.

The following example shows how arguments are passed by reference. The reference parameters are initialized with the actual arguments when the function is called.

```
#include <stdio.h>

void swapnum(int &i, int &j) {
  int temp = i;
  i = j;
  j = temp;
}

int main(void) {
  int a = 10;
  int b = 20;

  swapnum(a, b);
  printf("A is %d and B is %d\n", a, b);
  return 0;
}
```

When the function swapnum() is called, the values of the variables a and b are exchanged because they are passed by reference. The output is:

```
A is 20 and B is 10
```

4.11 Return statement

The **return** statement stops execution and returns to the calling function. When a return statement is executed, the function is terminated immediately at that point, regardless of whether it's in the middle of a loop, etc.

4.11.1 Return optional in void functions

A void function doesn't have to have a return statement -- when the end is reached, it automatically returns. However, a void function may optionally contain one or more return statements.

```
void printChars(char c, int count) {
```

```
    for (int i=0; i<count; i++) {
       cout << c;
    }//end for


    return;  // Optional because it's a void function
}//end printChars
```

4.11.2 Return required in non-void functions

If a function returns a value, it must have a return statement that specifies the value to return. It's possible to have more than one return, but the (human) complexity of a function generally increases with more return statements. It's generally considered better style to have one return at the end, unless that increases the complexity.

The max function below requires one or more return statements because it returns an int value.

```
// Multiple return statements often increase complexity.
int max(int a, int b) {
    if (a > b) {
        return a;
    } else {
        return b;
    }
}//end max
```

Here is a version of the max function that uses only one return statement by saving the result in a local variable. Some authors insist on only one return statement at the end of a function. Readable code is much more important than following such a fixed rule. The use of a single return probably improves the clarity of the max function slightly.

```
// Single return at end often improves readability.
int max(int a, int b) {
    int maxval;
    if (a > b) {
        maxval = a;
    } else {
        maxval = b;
    }
    return maxval;
```

```
}//end max
```

4.12 Passing Arrays as Function Arguments in C++

C++ does not allow to pass an entire array as an argument to a function. However, You can pass a pointer to an array by specifying the array's name without an index.

If you want to pass a single-dimension array as an argument in a function, you would have to declare function formal parameter in one of following three ways and all three declaration methods produce similar results because each tells the compiler that an integer pointer is going to be received.

Way-1

Formal parameters as a pointer as follows:

```
void myFunction(int *param)
{
.
.
.
}
```

Way-2

Formal parameters as a sized array as follows:

```
void myFunction(int param[10])
{
.
.
.
}
```

Way-3

Formal parameters as an unsized array as follows:

```
void myFunction(int param[])
{
.
.
.
}
```

Now, consider the following function, which will take an array as an argument along with another argument and based on the passed arguments, it will return average of the numbers passed through the array as follows:

```
double getAverage(int arr[], int size)
{
  int    i, sum = 0;
  double avg;

  for (i = 0; i < size; ++i)
  {
    sum += arr[i];
   }

  avg = double(sum) / size;

  return avg;
}
```

4.12.1 Calling function with array

Now, let us call the above function as follows:

```
#include <iostream>
using namespace std;

// function declaration:
double getAverage(int arr[], int size);

int main ()
{
   // an int array with 5 elements.
   int balance[5] = {1000, 2, 3, 17, 50};
   double avg;

   // pass pointer to the array as an argument.
   avg = getAverage( balance, 5 ) ;

   // output the returned value
   cout << "Average value is: " << avg << endl;

   return 0;
}
```

When the above code is compiled together and executed, it produces the following result:

```
Average value is: 214.4
```

As you can see, the length of the array doesn't matter as far as the function is concerned because C++ performs no bounds checking for the formal parameters.

4.13 C++ Variable Scope

A scope is a region of the program and broadly speaking there are three places, where variables can be declared:

Inside a function or a block which is called local variables,

In the definition of function parameters which is called formal parameters.

Outside of all functions which is called global variables.

4.13.1 Scope Rules of Functions in C C++

The scope rules of a language are the rules that govern whether a piece of code knows about or has access to another piece of code or data. Each function is a discrete block of code. A function's code is private to that function and cannot be accessed by any statement in any other function except through a call to that function. (For instance, you cannot use goto to jump into the middle of another
function.) The code that constitutes the body of a function is hidden from the rest of the program and, unless it uses global variables or data, it can neither affect nor be affected by other parts of the program. Stated another way, the code and data that are defined within one function cannot interact with the code or data defined in another function because the two functions have a different scope.
Variables that are defined within a function are called local variables. A local variable comes into existence when the function is entered and is destroyed upon exit. That is, local variables cannot hold their value between function calls. The only exception to this rule is when the variable is declared with the static storage class specifier. This causes the compiler to treat the variable as if it were a global variable for storage purposes, but limits its scope to within the function. In C (and C++) you cannot define a function within a function. This is why neither
C nor C++ are technically block-structured languages.

.

4.14 Local Variables:

Variables that are declared inside a function or block are local variables. They can be used only by statements that are inside that function or block of code. Local variables are not known to functions outside their own. Following is the example using local variables:

```
#include <iostream>
using namespace std;

int main ()
{
  // Local variable declaration:
  int a, b;
  int c;

  // actual initialization
  a = 10;
  b = 20;
  c = a + b;

  cout << c;

  return 0;
}
```

4.14.1 Global Variables:

Global variables are defined outside of all the functions, usually on top of the program. The global variables will hold their value throughout the life-time of your program.

A global variable can be accessed by any function. That is, a global variable is available for use throughout your entire program after its declaration. Following is the example using global and local variables:

```
#include <iostream>
using namespace std;

// Global variable declaration:
int g;

int main ()
{
  // Local variable declaration:
  int a, b;
```

```
  // actual initialization
  a = 10;
  b = 20;
  g = a + b;

  cout << g;

  return 0;
}
```

A program can have same name for local and global variables but value of local variable inside a function will take preference. For example:

```
#include <iostream>
using namespace std;

// Global variable declaration:
int g = 20;

int main ()
{
  // Local variable declaration:
  int g = 10;

  cout << g;

  return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
10
```

4.14.2 Initializing Local and Global Variables:

When a local variable is defined, it is not initialized by the system, you must initialize it yourself. Global variables are initialized automatically by the system when you define them as follows:

Data Type	**Initializer**
int	0
char	'\0'
float	0

double	0
pointer	NULL

It is a good programming practice to initialize variables properly, otherwise sometimes program would produce unexpected result.

Points to remember:-

A function is a group of statements that together perform a task.

A function **declaration** tells the compiler about a function's name, return type, and parameters.

A function **definition** provides the actual body of the function.

The **return_type** is the data type of the value the function returns.

While creating a C++ function, you give a definition of what the function has to do. To use a function, you will have to call or invoke that function.

The constant variable can be declared using const keyword. The const keyword makes variable value stable

The **call by value** method of passing arguments to a function copies the actual value of an argument into the formal parameter of the function.

The **call by reference** method of passing arguments to a function copies the reference of an argument into the formal parameter.

C++ does not allow to pass an entire array as an argument to a function. However, You can pass a pointer to an array by specifying the array's name without an index.

The scope rules of a language are the rules that govern whether a piece of code knows about or has access to another piece of code or data.

Variables that are declared inside a function or block are local variables.

Global variables are defined outside of all the functions, usually on top of the program.

Exercise

Fill in the blanks

1. A function is a _____ that together perform a task.
2. Variables that are declared inside a function or block are _____.
3. If a function returns a value, it must have a _____ statement that specifies the value to return.
4. The _____ keyword makes variable value stable
5. The _____ and the _____ together constitute the function signature.

Short Answer Type Questions:

1. What is a function?
2. What do you mean by function prototype?
3. How we can access a function in c++?
4. What do you mean by return type?
5. What is the use of constant arguments?
6. Define scope rules of functions and variables.
7. Explain the term ǿparametersô

Give difference:

1. Call by value and call by reference
2. Local and global variable.

Chapter 5
Array

C++ provides a data structure, **the array**, which stores a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

Instead of declaring individual variables, such as number0, number1, ..., and number99, you declare one array variable such as numbers and use numbers[0], numbers[1], and ..., numbers[99] to represent individual variables. A specific element in an array is accessed by an index.

All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.

5.1 Declaring Arrays:

To declare an array in C++, the programmer specifies the type of the elements and the number of elements required by an array as follows:

```
type arrayName [ arraySize ];
```

This is called a single-dimension array. The **arraySize** must be an integer constant greater than zero and **type** can be any valid C++ data type. For example, to declare a 10-element array called balance of type double, use this statement:

```
double balance[10];
```

5.1.1 Initializing Arrays:

You can initialize C++ array elements either one by one or using a single statement as follows:

```
double balance[5] = {1000.0, 2.0, 3.4, 17.0, 50.0};
```

The number of values between braces { } can not be larger than the number of elements that we declare for the array between square brackets [ ]. Following is an example to assign a single element of the array:

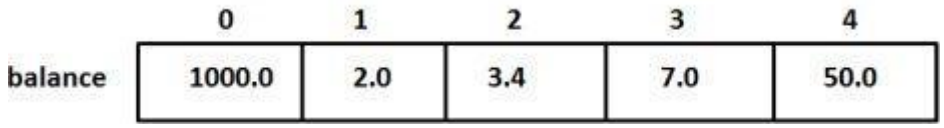
If you omit the size of the array, an array just big enough to hold the initialization is created. Therefore, if you write:

```
double balance[] = {1000.0, 2.0, 3.4, 17.0, 50.0};
```

You will create exactly the same array as you did in the previous example.

```
balance[4] = 50.0;
```

The above statement assigns element number 5th in the array a value of 50.0. Array with 4th index will be 5th, i.e., last element because all arrays have 0 as the index of their first element which is also called base index. Following is the pictorial representaion of the same array we discussed above:



5.1.2 Accessing Array Elements:

An element is accessed by indexing the array name. This is done by placing the index of the element within square brackets after the name of the array. For example:

```
double salary = balance[9];
```

The above statement will take 10th element from the array and assign the value to salary variable. Following is an example, which will use all the above-mentioned three concepts viz. declaration, assignment and accessing arrays:

```
#include <iostream>
using namespace std;

#include <iomanip>
using std::setw;

int main ()
{
  int n[ 10 ]; // n is an array of 10 integers

  // initialize elements of array n to 0
  for ( int i = 0; i < 10; i++ )
  {
    n[ i ] = i + 100; // set element at location i to i + 100
  }
  cout << "Element" << setw( 13 ) << "Value" << endl;

  // output each array element's value
  for ( int j = 0; j < 10; j++ )
  {
    cout << setw( 7 )<< j << setw( 13 ) << n[ j ] << endl;
  }
```

```
   return 0;
}
```

This program makes use of **setw()** function to format the output. When the above code is compiled and executed, it produces the following result:

```
Element      Value
      0        100
      1        101
      2        102
      3        103
      4        104
      5        105
      6        106
      7        107
      8        108
      9        109
```

5.2 One Dimensional Array

Single / One Dimensional Array is an array having a single index value to represent the arrays element.

Syntax:

```
type array_name[array_size_1]
```

Example:

```
#include <iostream.h>
void main()
{
  int i;
  float mark[6];
  cout << "Enter the marks of your 6 subjects:: \n";
  for(i=0; i<6; i++)
    {
          cin >> mark[i];
    }
  float sum=0;
  for(i=0;i<6;i++)
    {
       sum += mark[i];
    }
  float ave = sum/6;
  cout << "Average Marks is::" << ave << '\n';
}
```

Result:

```
Enter a the mark  of your 6 sujects::
 45
 56
 67
 58
 60
 59
Average Marks is::57.5
```

In the above example, the array "mark" refers the elements of an array by the index value "6". The marks entered are stored in the array using the index value of the array in a loop. The array elements are retrieved to calculate the sum of the array, then the average is found.

5.3 Nature of subscript

Array subscript is the same as the index. The number in the array that the data is being stored. For example

Array

0 1 2 3 4 5 6
a b c d e f g

Where the numbers are the index, or subscript, and the letters are the data of the subscript.

5.4 Multidimensional arrays

C++ allows multidimensional arrays. Here is the general form of a multidimensional array declaration:

```
type name[size1][size2]...[sizeN];
```

For example, the following declaration creates a three dimensional 5 . 10 . 4 integer array:

```
int threedim[5][10][4];
```

5.5 Two-Dimensional Arrays:

The simplest form of the multidimensional array is the two-dimensional array. A two-dimensional array is, in essence, a list of one-dimensional arrays. To declare a two-dimensional integer array of size x,y, you would write something as follows:

```
type arrayName [ x ][ y ];
```

Where **type** can be any valid C++ data type and **arrayName** will be a valid C++ identifier.

A two-dimensional array can be think as a table, which will have x number of rows and y number of columns. A 2-dimensional array **a**, which contains three rows and four columns can be shown as below:

	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

Thus, every element in array a is identified by an element name of the form **a[i][j]**, where a is the name of the array, and i and j are the subscripts that uniquely identify each element in a.

5.5.1 Initializing Two-Dimensional Arrays:

Multidimensioned arrays may be initialized by specifying bracketed values for each row. Following is an array with 3 rows and each row have 4 columns.

```
int a[3][4] = {
 {0, 1, 2, 3} ,   /*  initializers for row indexed by 0 */
 {4, 5, 6, 7} ,   /*  initializers for row indexed by 1 */
 {8, 9, 10, 11}   /*  initializers for row indexed by 2 */
};
```

The nested braces, which indicate the intended row, are optional. The following initialization is equivalent to previous example:

```
int a[3][4] = {0,1,2,3,4,5,6,7,8,9,10,11};
```

5.5.2 Accessing Two-Dimensional Array Elements:

An element in 2-dimensional array is accessed by using the subscripts, i.e., row index and column index of the array. For example:

```
int val = a[2][3];
```

The above statement will take 4th element from the 3rd row of the array. You can verify it in the above digram.

```
#include <iostream>
using namespace std;

int main ()
{
   // an array with 5 rows and 2 columns.
   int a[5][2] = { {0,0}, {1,2}, {2,4}, {3,6},{4,8}};

   // output each array element's value
   for ( int i = 0; i < 5; i++ )
      for ( int j = 0; j < 2; j++ )
      {
         cout << "a[" << i << "][" << j << "]: ";
         cout << a[i][j]<< endl;
      }

   return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
a[0][0]: 0
a[0][1]: 0
a[1][0]: 1
a[1][1]: 2
a[2][0]: 2
a[2][1]: 4
a[3][0]: 3
a[3][1]: 6
a[4][0]: 4
a[4][1]: 8
```

As explained above, you can have arrays with any number of dimensions, although it is likely that most of the arrays you create will be of one or two dimensions.

5.6 Array of strings

Array of strings in C++ is used to store a null terminated string which is a character array. This type of array has a string with a null character at the end of the string. Usually array of strings are declared one character long to accomodate the null character.

Example:

```
#include <iostream.h>
#include <string.h>
const int DAYS =7;
const int MAX =10;
void main()
 {
   char week[DAYS] [MAX] = {"Sunday", "Monday", "Tuesday",
   "Wednesday", "Thursday", "Friday", "Saturday" };
   for(int j=0;j<="" week[j]="" endl;="" }="" <="" pre="">
```

Result:

Sunday
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday

In the above example a two dimensional array is used as an array of strings.The first index specifies the total elements of an array, the second index the maximum length of each string. The "MAX" value is set to "10" since the string "Wednesday" has length of "9" with a null makes it "10".

Points to remember:-

An array as a collection of variables of the same type.

All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.

The **arraySize** must be an integer constant greater than zero and **type** can be any valid C++ data type.

An element is accessed by indexing the array name.

The simplest form of the multidimensional array is the two-dimensional array.

Array of strings in C++ is used to store a null terminated string which is a character array.

Array subscript is the same as the index number.

Exercise

Fill in the blanks:

1. An array as a _____ of the same type.
2. An element in 2-dimensional array is accessed by using the _____
3. Array of strings in C++ is used to store a _____ which is a character array.
4. _____is an array having a single index value to represent the arrays element.
5. If you omit _____, an array just big enough to hold the initialization is created.

Short Answer Type Questions:

1. Define an array.
2. Write a procedure to declare an array in c++.
3. What is subscript?
4. Explain two dimensional array. Give example.
5. What do you mean by array initialization?

Chapter 6

Classes and Objects

6.1 Classes

The main purpose of C++ programming is to add object orientation to the C programming language and classes are the central feature of C++ that supports object-oriented programming and are often called user-defined types.

A class is used to specify the form of an object and it combines data representation and methods for manipulating that data into one neat package. *Classes* are an expanded concept of *data structures*: like data structures, they can contain data members, but they can also contain functions as members.

An *object* is an instantiation of a class. In terms of variables, a class would be the type, and an object would be the variable.

Classes are defined using keyword class , with the following syntax:

```
class class_name {
  access_specifier_1:
    member1;
  access_specifier_2:
    member2;
  ...
} object_names;
```

Where class_name is a valid identifier for the class, object_names is an optional list of names for objects of this class. The body of the declaration can contain *members*, which can either be data or function declarations, and optionally *access specifiers*.

6.1.1Declaration of classes

To understand the declaration we will take the following example

Declares a class (i.e., a type) called Rectangle and an object (i.e., a variable) of this class, called rect. This class contains four members: two data members of type int (member width and member height) with *private access*(because private is the default access level) and two member functions with *public access*: the functions set_values and area, of which for now we have only included their declaration, but not their definition.

Here is the complete example of class Rectangle:

```
// classes example
#include <iostream>
using namespace std;

class Rectangle {
    int width, height;
  public:
    void set_values (int,int);
    int area() {return width*height;}
};

void Rectangle::set_values (int x, int y) {
  width = x;
  height = y;
}

int main () {
  Rectangle rect;
  rect.set_values (3,4);
  cout << "area: " << rect.area();
  return 0;
}
```

```
area: 12
```

After the declarations of Rectangle and rect, any of the public members of object rect can be accessed as if they were normal functions or normal variables, by simply inserting a dot (.) between *object name* and *member name*. This follows the same syntax as accessing the members of plain data structures. For example:

```
rect.set_values (3,4);
myarea = rect.area();
```

The only members of rect that cannot be accessed from outside the class are width and height, since they have private access and they can only be referred to from within other members of that same class.

This example introduces the *scope resolution operator* (::, two colons)

The scope resolution operator (::) specifies the class to which the member being declared belongs, granting exactly the same scope properties as if this function definition was directly included within the class definition. For example, the function set_values in the previous example has access to the variables width and height, which are private members of class Rectangle, and thus only accessible from other members of the class, such as this.

6.2 Access specifiers and default labels

Data hiding is one of the important features of Object Oriented Programming which allows preventing the functions of a program to access directly the internal representation of a class type. The access restriction to the class members is specified by the labeled **public, private,** and **protected** sections within the class body. The keywords public, private, and protected are called access specifiers.

A class can have multiple public, protected, or private labeled sections. Each section remains in effect until either another section label or the closing right brace of the class body is seen. The default access for members and classes is private.

```
class Base {

   public:

  // public members go here

   protected:

  // protected members go here

   private:

  // private members go here

};
```

6.3 Scope of class & its members

6.3.1 The public members:

A **public** member is accessible from anywhere outside the class but within a program. You can set and get the value of public variables without any member function as shown in the following example:

```
#include <iostream>

using namespace std;

class Line
{
   public:
      double length;
      void setLength( double len );
```

```
      double getLength( void );
};

// Member functions definitions
double Line::getLength(void)
{
    return length ;
}

void Line::setLength( double len )
{
    length = len;
}

// Main function for the program
int main( )
{
   Line line;

   // set line length
   line.setLength(6.0);
   cout << "Length of line : " << line.getLength() <<endl;

   // set line length without member function
   line.length = 10.0; // OK: because length is public
   cout << "Length of line : " << line.length <<endl;
   return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Length of line : 6
Length of line : 10
```

6.3.2 The private members:

A **private** member variable or function cannot be accessed, or even viewed from outside the class. Only the class and friend functions can access private members.
By default all the members of a class would be private, for example in the following class **width** is a private member, which means until you label a member, it will be assumed a private member:

```
class Box
{
   double width;
   public:
      double length;
```

```
   void setWidth( double wid );
   double getWidth( void );
};
```

Practically, we define data in private section and related functions in public section so that they can be called from outside of the class as shown in the following program.

```
#include <iostream>

using namespace std;

class Box
{
  public:
    double length;
    void setWidth( double wid );
    double getWidth( void );

  private:
    double width;
};

// Member functions definitions
double Box::getWidth(void)
{
   return width ;
}

void Box::setWidth( double wid )
{
   width = wid;
}

// Main function for the program
int main( )
{
  Box box;

  // set box length without member function
  box.length = 10.0; // OK: because length is public
  cout << "Length of box : " << box.length <<endl;

  // set box width without member function
  // box.width = 10.0; // Error: because width is private
  box.setWidth(10.0);  // Use member function to set it.
```

```
   cout << "Width of box : " << box.getWidth() <<endl;

   return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Length of box : 10
Width of box : 10
```

6.3.3 The protected members:

A **protected** member variable or function is very similar to a private member but it provided one additional benefit that they can be accessed in child classes which are called derived classes.

You will learn derived classes and inheritance in next chapter. For now you can check following example where I have derived one child class **SmallBox** from a parent class **Box**.

Following example is similar to above example and here **width** member will be accessible by any member function of its derived class SmallBox.

```
#include <iostream>
using namespace std;

class Box
{
   protected:
      double width;
};

class SmallBox:Box // SmallBox is the derived class.
{
   public:
      void setSmallWidth( double wid );
      double getSmallWidth( void );
};

// Member functions of child class
double SmallBox::getSmallWidth(void)
{
    return width ;
}

void SmallBox::setSmallWidth( double wid )
{
    width = wid;
}
```

```
// Main function for the program
int main( )
{
   SmallBox box;

   // set box width using member function
   box.setSmallWidth(5.0);
   cout << "Width of box : "<< box.getSmallWidth() << endl;

   return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Width of box : 5
```

6.4 Member Functions

A member function of a class is a function that has its definition or its prototype within the class definition like any other variable. It operates on any object of the class of which it is a member, and has access to all the members of a class for that object.

Let us take previously defined class to access the members of the class using a member function instead of directly accessing them:

```
class Box
{
   public:
      double length;         // Length of a box
      double breadth;        // Breadth of a box
      double height;         // Height of a box
      double getVolume(void);// Returns box volume
};
```

Member functions can be defined within the class definition or separately using **scope resolution operator, ::**. Defining a member function within the class definition declares the function **inline**, even if you do not use the inline specifier. So either you can define **Volume()** function as below:

```
class Box
{
   public:
      double length;      // Length of a box
      double breadth;     // Breadth of a box
```

```
   double height;      // Height of a box

   double getVolume(void)
   {
     return length * breadth * height;
   }
};
```

If you like you can define same function outside the class using **scope resolution operator, ::** as follows:

```
double Box::getVolume(void)
{
   return length * breadth * height;
}
```

Here, only important point is that you would have to use class name just before :: operator. A member function will be called using a dot operator (**.**) on a object where it will manipulate data related to that object only as follows:

```
Box myBox;          // Create an object

myBox.getVolume();  // Call member function for the object
```

Let us put above concepts to set and get the value of different class members in a class:

```
#include <iostream>

using namespace std;

class Box
{
  public:
    double length;         // Length of a box
    double breadth;        // Breadth of a box
    double height;         // Height of a box

    // Member functions declaration
    double getVolume(void);
    void setLength( double len );
    void setBreadth( double bre );
    void setHeight( double hei );
};

// Member functions definitions
double Box::getVolume(void)
{
```

```
    return length * breadth * height;
}

void Box::setLength( double len )
{
    length = len;
}

void Box::setBreadth( double bre )
{
    breadth = bre;
}

void Box::setHeight( double hei )
{
    height = hei;
}

// Main function for the program
int main( )
{
   Box Box1;               // Declare Box1 of type Box
   Box Box2;               // Declare Box2 of type Box
   double volume = 0.0;     // Store the volume of a box here

   // box 1 specification
   Box1.setLength(6.0);
   Box1.setBreadth(7.0);
   Box1.setHeight(5.0);

   // box 2 specification
   Box2.setLength(12.0);
   Box2.setBreadth(13.0);
   Box2.setHeight(10.0);

   // volume of box 1
   volume = Box1.getVolume();
   cout << "Volume of Box1 : " << volume <<endl;

   // volume of box 2
   volume = Box2.getVolume();
   cout << "Volume of Box2 : " << volume <<endl;
   return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Volume of Box1 : 210
Volume of Box2 : 1560
```

6.5 Data hiding & encapsulation

All C++ programs are composed of the following two fundamental elements:

Program statements (code): This is the part of a program that performs actions and they are called functions.
Program data: The data is the information of the program which affected by the program functions.
Encapsulation is an Object Oriented Programming concept that binds together the data and functions that manipulate the data, and that keeps both safe from outside interference and misuse. Data encapsulation led to the important OOP concept of **data hiding**.
Data encapsulation is a mechanism of bundling the data, and the functions that use them and **data abstraction** is a mechanism of exposing only the interfaces and hiding the implementation details from the user.
C++ supports the properties of encapsulation and data hiding through the creation of user-defined types, called **classes**. We already have studied that a class can contain **private, protected** and **public**members. By default, all items defined in a class are private. For example:

```
class Box
{
  public:
    double getVolume(void)
    {
      return length * breadth * height;
    }
  private:
    double length;      // Length of a box
    double breadth;     // Breadth of a box
    double height;      // Height of a box
};
```

The variables length, breadth, and height are **private**. This means that they can be accessed only by other members of the Box class, and not by any other part of your program. This is one way encapsulation is achieved.
To make parts of a class **public** (i.e., accessible to other parts of your program), you must declare them after the **public** keyword. All variables or functions defined after the public specifier are accessible by all other functions in your program.

Making one class a friend of another exposes the implementation details and reduces encapsulation. The ideal is to keep as many of the details of each class hidden from all other classes as possible.

6.6 Inline Functions

C++ **inline** function is powerful concept that is commonly used with classes. If a function is inline, the compiler places a copy of the code of that function at each point where the function is called at compile time.

Any change to an inline function could require all clients of the function to be recompiled because compiler would need to replace all the code once again otherwise it will continue with old functionality.

To inline a function, place the keyword **inline** before the function name and define the function before any calls are made to the function. The compiler can ignore the inline qualifier in case defined function is more than a line.
A function definition in a class definition is an inline function definition, even without the use of the **inline**specifier.

Following is an example, which makes use of inline function to return max of two numbers:

```
#include <iostream>

using namespace std;

inline int Max(int x, int y)
{
   return (x > y)? x : y;
}

// Main function for the program
int main( )
{

   cout << "Max (20,10): " << Max(20,10) << endl;
   cout << "Max (0,200): " << Max(0,200) << endl;
   cout << "Max (100,1010): " << Max(100,1010) << endl;
   return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Max (20,10): 20
Max (0,200): 200
Max (100,1010): 1010
```

6.7 Nesting of Member Functions

A member function of a class can be called only by an object of that class using a dot operator. However, there is an exception to this. A member function can be called by using its name inside another member function of the same class. This is known as nesting of member functions.

Nesting of Member Function

```
#include
using namespace std;
class set
{
int m,n;
public:
void input(void);
void display(void);
void largest(void);
};
int set :: largest(void)
{
if(m >= n)
return(m);
else
return(n);
}
void set :: input(void)
{
cout << "Input value of m and n"<<"\n";
cin >> m>>n;
}
void set :: display(void)
{
cout << "largest value=" << largest() <<"\n";
```

```
}

int main()
{
set A;
A.input();
A.display();

return 0;
}
```

The output of program would be:

```
Input value of m and n
25   18
Largest value=25
```

6.8 Array within a Class

The array can be used as member variables in a class. The following class definition is valid.

```
const int size=10;

class array
{
int a[size];
public:
void setval(void);
void display(void);
};
```

The array variable a[] declared as private member of the class array can be used in the member function, like any other array variable. We can perform any operations on it. For instance, in the above class definition, the member function setval() sets the value

of element of the array a[], and display() function displays the values. Similarly, we may use other member functions to perform any other operation on the array values.

6.8.1 Array of objects

Arrays of variables of type "**class**" is known as "**Array of objects**". The "identifier" used to refer the array of objects is an user defined data type.

Example:

```
#include <iostream.h>
const int MAX  =100;
class Details
 {
  private:
     int salary;
     float roll;
  public:
     void getname( )
      {
         cout  << "\n Enter the Salary:";
         cin >> salary;
         cout  << "\n Enter the roll:";
         cin >> roll;
      }
    void putname( )
      {
         cout << "Employees" << salary <<
         "and roll is" << roll << '\n';
      }
  };
void main()
 {
  Details det[MAX];
  int n=0;
  char ans;
  do{
      cout  << "Enter the Employee Number::"  <<  n+1;
      det[n++].getname;
      cout << "Enter another (y/n)?: " ;
      cin >> ans;
    } while ( ans != 'n' );
  for (int j=0; j<n; j++)
    {
       cout << "\nEmployee Number is:: " << j+1;
```

```
            det[j].putname( );
        }
    }
```

Result:

```
Enter the Employee Number:: 1
Enter the Salary:20

Enter the roll:30
Enter another (y/n)?: y
Enter the Employee Number:: 2
Enter the Salary:20

Enter the roll:30
Enter another (y/n)?: n
```

In the above example an array of object "det" is defined using the user defined data type "Details". The class element "getname()" is used to get the input that is stored in this array of objects and putname() is used to display the information.

6.8.2 OBJECTS AS FUNCTION ARGUMENTS

Similar to variables, objects can be passed on to functions. There are three methods to pass an argument to a function as given below:

(a) Pass-by-valueð In this type a copy of an object (actual object) is sent to function and assigned to object of callee function (Formal object). Both actual and formal copies of objects are stored at different memory locations. Hence, changes made in formal objects are not reflected to actual objects.

(b) Pass-by-reference ð Address of object is implicitly sent to function.

(c) Pass-by-address ð Address of the object is explicitly sent to function.

In pass by reference and address methods, an address of actual object is passed to the function. The formal argument is reference pointer to the actual object. Hence, changes made in the object are reflected to actual object. These two methods are useful because an address is passed to the function and duplicating of object is prevented.

Static members of a C++ class

We can define class members static using **static** keyword. When we declare a member of a class as static it means no matter how many objects of the class are created, there is only one copy of the static member.
A static member is shared by all objects of the class. All static data is initialized to zero when the first object is created, if no other initialization is present. We can't put it in the class definition but it can be initialized outside the class as done in the following example by redeclaring the static variable, using the scope resolution operator **::** to identify which class it belongs to.

Let us try the following example to understand the concept of static data members:

```
#include <iostream>

using namespace std;

class Box
{
  public:
    static int objectCount;
    // Constructor definition
    Box(double l=2.0, double b=2.0, double h=2.0)
    {
      cout <<"Constructor called." << endl;
      length = l;
      breadth = b;
      height = h;
      // Increase every time object is created
      objectCount++;
    }
    double Volume()
    {
      return length * breadth * height;
    }
  private:
    double length;     // Length of a box
    double breadth;    // Breadth of a box
    double height;     // Height of a box
};

// Initialize static member of class Box
int Box::objectCount = 0;

int main(void)
{
  Box Box1(3.3, 1.2, 1.5);    // Declare box1
  Box Box2(8.5, 6.0, 2.0);    // Declare box2
```

```
   // Print total number of objects.
   cout << "Total objects: " << Box::objectCount << endl;

   return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Constructor called.
Constructor called.
Total objects: 2
```

6.9 Static Function Members:

By declaring a function member as static, you make it independent of any particular object of the class. A static member function can be called even if no objects of the class exist and the **static** functions are accessed using only the class name and the scope resolution operator **::**.

A static member function can only access static data member, other static member functions and any other functions from outside the class.

Static member functions have a class scope and they do not have access to the **this** pointer of the class. You could use a static member function to determine whether some objects of the class have been created or not.

Let us try the following example to understand the concept of static function members:

```
#include <iostream>

using namespace std;

class Box
{
   public:
      static int objectCount;
      // Constructor definition
      Box(double l=2.0, double b=2.0, double h=2.0)
      {
         cout <<"Constructor called." << endl;
         length = l;
         breadth = b;
         height = h;
         // Increase every time object is created
         objectCount++;
      }
```

```
    double Volume()
    {
      return length * breadth * height;
    }
    static int getCount()
    {
      return objectCount;
    }
  private:
    double length;     // Length of a box
    double breadth;    // Breadth of a box
    double height;     // Height of a box
};

// Initialize static member of class Box
int Box::objectCount = 0;

int main(void)
{

  // Print total number of objects before creating object.
  cout << "Inital Stage Count: " << Box::getCount() << endl;

  Box Box1(3.3, 1.2, 1.5);    // Declare box1
  Box Box2(8.5, 6.0, 2.0);    // Declare box2

  // Print total number of objects after creating object.
  cout << "Final Stage Count: " << Box::getCount() << endl;

  return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Inital Stage Count: 0
Constructor called.
Constructor called.
Final Stage Count: 2
```

Points To Remember:-

A class is used to specify the form of an object and it combines data representation and methods for manipulating that data into one neat package.

An *object* is an instantiation of a class.

The scope resolution operator (::) specifies the class to which the member being declared belongs, granting exactly the same scope properties as if this function definition was directly included within the class definition.

The access restriction to the class members is specified by the labeled **public, private,** and **protected** sections within the class body.

Public member is accessible from anywhere outside the class but within a program.

A **private** member variable or function cannot be accessed, or even viewed from outside the class. Only the class and friend functions can access private members.

A **protected** member variable or function is very similar to a private member but it provided one additional benefit that they can be accessed in child classes which are called derived classes.

Arrays of variables of type "**class**" is known as "**Array of objects**".

By declaring a function member as static, you make it independent of any particular object of the class.

Exercise

Fill in the blanks:

1. A static member is shared by all _____ of the class.
2. class_name is a valid _____ for the class.
3. _____ specifies the class to which the member being declared belongs,
4. _____ member is accessible from anywhere outside the class but within a program.
5. _____ is an instantiation of a class.

Questions:

1. What is a class? How we Define it?

2. Explain the working of public members with an example.
3. What is array of objects? Explain.
4. Is it possible to do the nesting of member function ?if yes,explain.
5. What is inline functions?
6. What is access specifiers? Explain
7. Define encapsulation.

Chapter 7

Constructors , Destructors And Function Overloading

7.1 Need for constructors

Suppose you are working on 100's of objects and the default value of a data member is 0. Initialising all objects manually will be very tedious. Instead, you can define a constructor which initialises that data member to 0. Then all you have to do is define object and constructor will initialise object automatically. These types of situation arises frequently while handling array of objects. Also, if you want to execute some codes immediately after object is created, you can place that code inside the body of constructor

7.1.1 Declaration and Definition of Constructors

Constructors are the special type of member function that initialises the object automatically when it is created Compiler identifies that the given member function is a constructor by its name and return type. Constructor has same name as that of class and it does not have any return type.

```
..... ... .....
class temporary
  {
     private:
       int x;
       float y;
     public:
       temporary(): x(5), y(5.5)        /* Constructor  */
```

```
        {
            /* Body of constructor */
        }
      .... ...  ....
}
int main()
  {
    Temporary t1;
    .... ... ....
  }
```

7.1.2 Working of Constructor

In the above pseudo code, temporary() is a constructor. When the object of class temporary is created, constructor is called and *x* is initialized to 5 and *y* is initialized to 5.5 automatically.

You can also initialise data member inside the constructor's function body as below. But, this method is not preferred.

```
temporary(){
  x=5;
  y=5.5;
}
/* This method is not preferred. /*
```

.

Constructor Example

```
/*Source Code to demonstrate the working of constructor in C++ Programming
*/
/* This program calculates the area of a rectangle and  displays it. */
#include <iostream>
using namespace std;
class Area
{
    private:
      int length;
      int breadth;

    public:
      Area(): length(5), breadth(2){ }   /* Constructor */
      void GetLength()
      {
          cout<<"Enter length and breadth respectively: ";
          cin>>length>>breadth;
      }
      int AreaCalculation() {  return (length*breadth);  }
      void DisplayArea(int temp)
      {
          cout<<"Area: "<<temp;
      }
};
int main()
{
    Area A1,A2;
    int temp;
    A1.GetLength();
    temp=A1.AreaCalculation();
    A1.DisplayArea(temp);
    cout<<endl<<"Default Area when value is not taken from user"<<endl;
    temp=A2.AreaCalculation();
    A2.DisplayArea(temp);
    return 0;
}
```

Explanation

In this program, a class of name *Area* is created to calculate the area of a rectangle. There are two data members *length* and *breadth*. A constructor is defined which initialises *length* to 5 and*breadth* to 2. And, we have three additional member functions GetLength(), AreaCalculation() and

DisplayArea() to get length from user, calculate the area and display the area respectively.

When, objects *A1* and A2 are created then, the length and breadth of both objects are initialized to 5 and 2 respectively because of the constructor. Then the member function GetLength() is invoked which takes the value of *length* and *breadth* from user for object *A1*. Then, the area for the object*A1* is calculated and stored in variable *temp* by calling AreaCalculation() function. And finally, the area of object *A1* is displayed. For object *A2*, no data is asked from the user. So, the value of *length*will be 5 and *breadth* will be 2. Then, the area for *A2* is calculated and displayed which is 10.

Output

Enter length and breadth respectively: 6

7

Area: 42

Default Area when value is not taken from user

Area: 10

7.2 Default Constructors

In computer programming languages the term **default constructor** can refer to a constructor that is automatically generated by the compiler in the absence of any programmer-defined constructors and is usually a nullary constructor. In other languages (e.g. in C++) it is a constructor that can be called without having to provide any arguments, irrespective of whether the constructor is auto-generated or used-defined. Note that a constructor with formal parameters can still be called without arguments if default arguments were provided in the constructor's definition.

In C++, the standard describes the default constructor for a class as a constructor that can be called with no arguments (this includes a constructor whose parameters all have default arguments) For example:

class MyClass

```
{
public:
   MyClass();  // constructor declared

private:
   int x;
};

MyClass :: MyClass()  // constructor defined
{
   x = 100;
}

int main()
{
   MyClass m;  // at runtime, object m is created, and the default constructor is called
}
```

When allocating memory dynamically, the constructor may be called by adding parenthesis after the class name. In a sense, this is an explicit call to the constructor:

```
int main()
{
   MyClass * pointer = new MyClass();  // at runtime, an object is created, and the
                                       // default constructor is called
}
```

If the constructor does have one or more parameters, but they all have default values, then it is still a default constructor. Remember that each class can have at most one default constructor, either one without parameters, or one whose all parameters have default values, such as in this case:

```
class MyClass
{
public:
   MyClass (int i = 0, std::string s = "");  // constructor declared

private:
   int x;
   int y;
   std::string z;
};

MyClass :: MyClass(int i, std::string s)     // constructor defined
{
```

```
    x = 100;
    y = i;
    z = s;
}
```

In C++, default constructors are significant because they are automatically invoked in certain circumstances; and therefore, in these circumstances, it is an error for a class to not have a default constructor:

7.3 Parameterized Constructor:

A default constructor does not have any parameter, but if you need, a constructor can have parameters. This helps you to assign initial value to an object at the time of its creation as shown in the following example:

```
#include <iostream>

using namespace std;

class Line
{
   public:
      void setLength( double len );
      double getLength( void );
      Line(double len);  // This is the constructor

   private:
      double length;
};

// Member functions definitions including constructor
Line::Line( double len)
{
    cout << "Object is being created, length = " << len << endl;
    length = len;
}

void Line::setLength( double len )
{
    length = len;
}

double Line::getLength( void )
{
```

```
    return length;
}
// Main function for the program
int main( )
{
   Line line(10.0);

   // get initially set length.
   cout << "Length of line : " << line.getLength() <<endl;
   // set line length again
   line.setLength(6.0);
   cout << "Length of line : " << line.getLength() <<endl;

   return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Object is being created, length = 10
Length of line : 10
Length of line : 6
```

7.4 Default Copy Constructor

A object can be initialized with another object of same type. Let us suppose the above program. If you want to initialise a object A3 so that it contains same value as A2. Then, this can be performed as:

```
 ....

 int main() {

    Area A1,A2(2,1);

    Area A3(A2);     /* Copies the content of A2 to A3 */

      OR,

    Area A3=A2;      /* Copies the content of A2 to A3 */
```

```
}
```

You might think, you may need some constructor to perform this task. But, no additional constructor is needed. It is because this constructor is already built into all classes.

7.5 DYNAMIC INITIALIZATION USING CONSTRUCTORS

After declaration of the class data member variables, they can be initialized at the time of program execution using pointers. Such initialization of data is called as dynamic initialization. The benefit of dynamic initialization is that it allows different initialization modes using overloaded constructors. Pointer variables are used as argument for constructors. The following example explains dynamic initialization using overloaded constructor.

Write a program to initialize member variables using pointers and constructors.

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

class city
{
  char city[20];
  char state[20];
  char country[20];

public:

city( ) {    city[0]=state[0]=country[0]=NULL; }

void display( char *line);

city(char *cityn)
{

strcpy(city, cityn);
state[0]=NULL;
}

city(char *cityn,char *staten)
{
  strcpy(city,cityn);
  strcpy(state,staten);
  country[0]=NULL;
}
```

```
city(char *cityn,char *staten, char *countryn)
{
  _fstrcpy(city,cityn);
  _fstrcpy(state,staten);
  _fstrcpy(country,countryn);
}
};

void city:: display (char *line)
{
  cout <<line<<endl;
  if (_fstrlen(city))     cout<<"City    : "<<city<<endl;
  if (strlen(state))    cout <<"State   : "<<state <<endl;
  if (strlen(country))  cout <<"Country : "<<country <<endl;
}

void main( )
{
  clrscr( );
  city c1("Mumbai"),
       c2("Nagpur","Maharashtra"),
       c3("Nanded","Maharashtra","India"),
       c4('\0','\0','\0');

  c1.display("=========*============");
  c2.display("=========*============");
....
```

7.6 The Class Destructor:

7.6.1 Definition and characteristics

A **destructor** is a special member function of a class that is executed whenever an object of it's class goes out of scope or whenever the delete expression is applied to a pointer to the object of that class.

A destructor will have exact same name as the class prefixed with a tilde (~) and it can neither return a value nor can it take any parameters. Destructor can be very useful for releasing resources before coming out of the program like closing files, releasing memories etc.

Following example explains the concept of destructor:

```
#include <iostream>
```

```
using namespace std;

class Line
{
   public:
      void setLength( double len );
      double getLength( void );
      Line();   // This is the constructor declaration
      ~Line();  // This is the destructor: declaration

   private:
      double length;
};

// Member functions definitions including constructor
Line::Line(void)
{
    cout << "Object is being created" << endl;
}
Line::~Line(void)
{
    cout << "Object is being deleted" << endl;
}

void Line::setLength( double len )
{
    length = len;
}

double Line::getLength( void )
{
    return length;
}
// Main function for the program
int main( )
{
   Line line;

   // set line length
   line.setLength(6.0);
   cout << "Length of line : " << line.getLength() <<endl;

   return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Object is being created
Length of line : 6
Object is being deleted
```

7.7 Function overloading in C++:

An overloaded declaration is a declaration that had been declared with the same name as a previously declared declaration in the same scope, except that both declarations have different arguments and obviously different definition (implementation).

You can have multiple definitions for the same function name in the same scope. The definition of the function must differ from each other by the types and/or the number of arguments in the argument list. You can not overload function declarations that differ only by return type.

Following is the example where same function **print()** is being used to print different data types:

```
#include <iostream>
using namespace std;

class printData
{
   public:
      void print(int i) {
        cout << "Printing int: " << i << endl;
      }

      void print(double  f) {
        cout << "Printing float: " << f << endl;
      }

      void print(char* c) {
        cout << "Printing character: " << c << endl;
      }
};

int main(void)
{
   printData pd;

   // Call print to print integer
   pd.print(5);
   // Call print to print float
   pd.print(500.263);
```

```
   // Call print to print character
   pd.print("Hello C++");

   return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Printing int: 5
Printing float: 500.263
Printing character: Hello C++
```

7.8 Steps involved in finding the best match

The process of creating and deleting objects in C++ is not a trivial task. Every time an instance of a class is created the constructor method is called. The constructor has the same name as the class and it doesn't return any type, while the destructor's name it's defined in the same way, but with a '~' in front:

Even if a class is not equipped with a constructor, the compiler will generate code for one, called the implicit default constructor. This will typically call the default constructors for all class members, if the class is using virtual methods it is used to initialize the pointer to the virtual table, and, in class hierarchies, it calls the constructors of the base classes. Both constructors in the above example use initialization lists in order to initialize the members of the class.

The construction order of the members is the order in which they are defined, and for this reason the same order should be preserved in the initialization list to avoid confusion.

To master developing and debugging C++ applications, more insight is required regarding the way constructors and destructors work.

Points to remember:

Constructors are the special type of member function that initializes the object automatically when it is created Compiler identifies that the given member function is a constructor by its name and return type.
Default constructor can refer to a constructor that is automatically generated by the compiler in the absence of any programmer-defined constructors.
The constructor has the same name as the class and it doesn't return any type, while the destructor's name it's defined in the same way, but with a '~' in front:

You can have multiple definitions for the same function name in the same scope.
A **destructor** is a special member function of a class that is executed whenever an object of it's class goes out of scope or whenever the delete expression is applied to a pointer to the object of that class.
A object can be initialized with another object of same type.

Exercise

Fill in the blanks:

1. Constructors are the special type of member function that _____ the object automatically.
2. _____can refer to a constructor that is automatically generated by the compiler in the absence of any programmer-defined constructors.
3. When allocating memory dynamically, the constructor may be called by adding _____ after the class name.
4. _____ of the members is the order in which they are defined.
5. _____allows different initialization modes using overloaded constructors.

Question Answers:

1. What do you mean by constructors? What is the need for define a constructor in a class?
2. Explain parameterized constructors.
3. Describe the process of dynamic initialization of constructors.
4. What are Destructors?
5. What do you mean by Function overloading?

Chapter 8

Inheritance

8.1 Inheritance:Extending Classes

This lesson discusses about inheritance, the capability of one class to inherit properties from another class as a child inherits some properties from his/her parents. The most important advantage of inheritance is code reusability. Once a base class is written and debugged, it can be used in various situations without having to redefine it or rewrite it. Reusing existing code saves time, money and efforts of writing the code again. Without redefining the old class, you can add new properties to desired class and redefine an inherited class member function.

8.2 Need for Inheritance

Inheritance is one of the important concepts of object-oriented language. There are several reasons why this concept was introduced in object oriented language. Some major reasons are:

(i) The capability to express the inheritance relationship which ensures the closeness with the real world model.

(ii) Idea of reusability, i.e., the new class can use some of the features of old class.

(iii) Transitive nature of inheritance, i.e., it can be passed on further.

8.3 Defining Derived Class

A derived class is defined by specifying its relationship with the base class using visibility mode.

The general form of defining a derived class is:

class derived_class : visibility_mode base_class

{

_____

_____ // members of derived class.

};

(inherits some property) from base_class.

The base class(es) name(s) follow(s) the colon (:). The names of all the base classes of a derived class follow : (colon) and are separated by comma. The visibility-mode can be either private or public or protected. If no visibility mode is specified, then by default the visibility mode is considered as private.

Following are some examples of derived class definitions:

class Marksheet : public student / / public derivation

{

// members of derived class

};

class Marksheet : private student / / private derivation

// members of derived class

};

class Marksheet : protected student // protected derivation

{

// members of protected class

};

In the above definitions, Marksheet is the derived class of student base class. The visibility mode public indicates that student is a public base class. Similarly, the visibility modes private or protected indicates that student is private base class or protected base class respectively.Inheritance Extending Classes :: 135

When we say that members of a class are inheritable, it means that the derived class can access them directly. However, the derived class has access privilege only to the non-private members of the base class. Although the private members of the base class cannot be accessed directly, yet the objects of derived class are able to access them through the non-private inherited members.

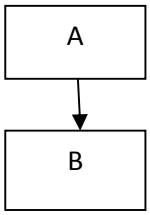
8.4 Different Forms of Inheritance

The mechanism of deriving a new class from an old one is called inheritance (or derivation). The old class is referred to as the base class and new one is called the derived class. There are various forms of inheritance.

8.4.1 Single inheritance

ð > A derived class with only one base class is called single

inheritance.



Example:

```
#include <iostream.h>
class Value
  {
    protected:
      int val;
    public:
      void set_values (int a)
        { val=a;}
  };
class Cube: public Value
  {
    public:
     int cube()
       { return (val*val*val); }
```

```
  };
int main ()
 {
   Cube cub;
   cub.set_values (5);
   cout << "The Cube of 5 is::" << cub.cube() << endl;
   return 0;
 }
```

Result:

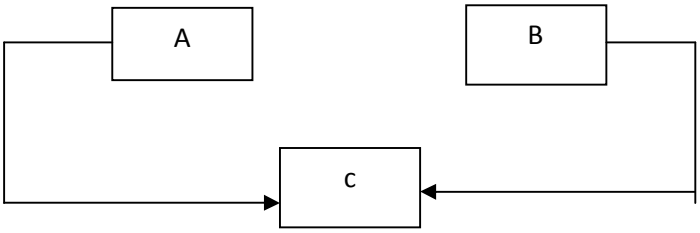
The Cube of 5 is:: 125

In the above example the derived class "Cube" has only one base class "Value". This is the single inheritance OOP's concept.

8.4.2 Multiple inheritance

ð > A derived class with several base classes is called

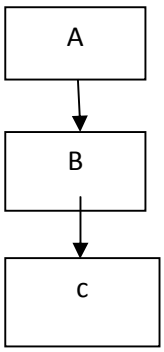
multiple inheritance.



8.4.3 Multilevel inheritance

ð > The mechanism of deriving a class from another

derived class is called multilevel inheritance.



Example:

```
#include <iostream.h>
class mm
  {
    protected:
      int rollno;
    public:
      void get_num(int a)
        { rollno = a; }
      void put_num()
        { cout << "Roll Number Is:\n"<< rollno << "\n"; }
  };
class marks : public mm
  {
    protected:
      int sub1;
      int sub2;
    public:
      void get_marks(int x,int y)
        {
          sub1 = x;
          sub2 = y;
        }
      void put_marks(void)
        {
          cout << "Subject 1:" << sub1 << "\n";
          cout << "Subject 2:" << sub2 << "\n";
        }
   };
class res : public marks
   {
     protected:
       float tot;
     public:
       void disp(void)
         {
           tot = sub1+sub2;
           put_num();
           put_marks();
           cout << "Total:"<< tot;
         }
   };
int main()
```

```
{
  res std1;
  std1.get_num(5);
  std1.get_marks(10,20);
  std1.disp();
  return 0;
}
```

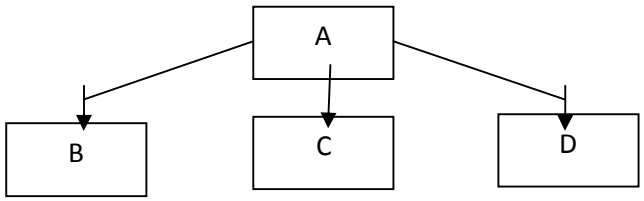
Result:

```
Roll Number Is:
5
Subject 1: 10
Subject 2: 20
Total: 30
```

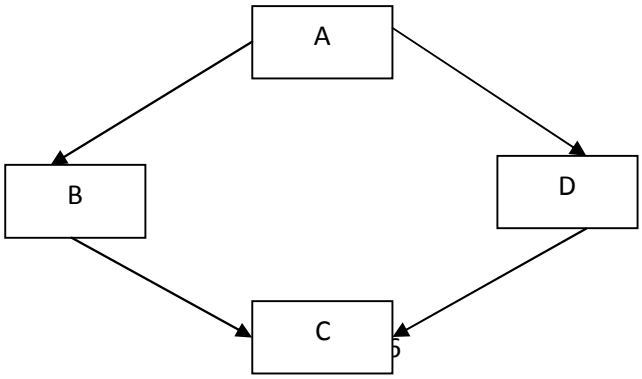
In the above example, the derived function "res" uses the function "put_num()" from another derived class "marks", which just a level above. This is the multilevel inheritance OOP's concept in C++.

8.4.4 Hierarchical inheritance ð > One class may be inherited by more than one

classes. This process is known as hierarchical inheritance.



8.4.5 Hybrid inheritance ð > It is a combination of hierarchical and multiple inheritance.



8.5 Visibility Modes

It can be public, private or protected. The private data of base class cannot be inherited.

(i) If inheritance is done in public mode, public members of the base class become the public members of derived class and protected members of base class become the protected members of derived class.

(ii) In inheritances is done in a private mode, public and protected members of base class become the private members of derived class.

(iii) If inheritance is done in a protected mode, public and protected members of base class become the protected members of derived class.

The following table shows the three types of inheritance:136 :: Certificate in Computer Science

Base class Access specifier	Derived class		
	public	private	protected
public	public	private	protected
private	Not inherited	Not inherited	Not inherited
protected	protected	protected	protected

8.5.1The Public Visibility mode

The following example and figure illustrate the public derivation in classes.

class student

```
{
private :
int x;
void getdata ( );
public:
int y;
void putdata ( );
protected:
int z;
void check ( );
};
class marks : public student
{
private :
int a ;
void readdata ( );
public :
int b;
void writedata ( );
protected :Inheritance Extending Classes :: 137
int c;
void checkvalue ( );
};
```

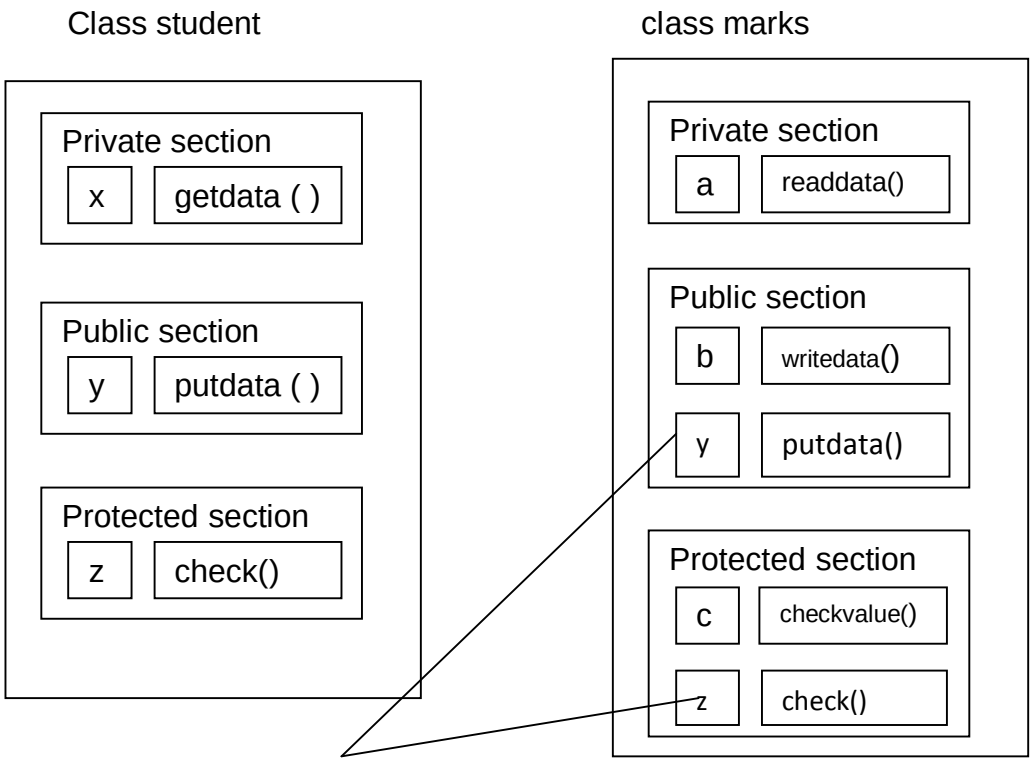


Fig. public derivation of a class

The public derivation does not change the access specifiers for inherited members in the derived class. The private data of base class student cannot be inherited.

8.5.2 The Private Visibility Mode

We are using the same example, but the derivation is done in private mode.

```
Class student
{
// same as in previous example138 :: Certificate in Computer Science
};
class marks : private student
{
//
};
```

The following figure illustrates the private derivation in the classes.

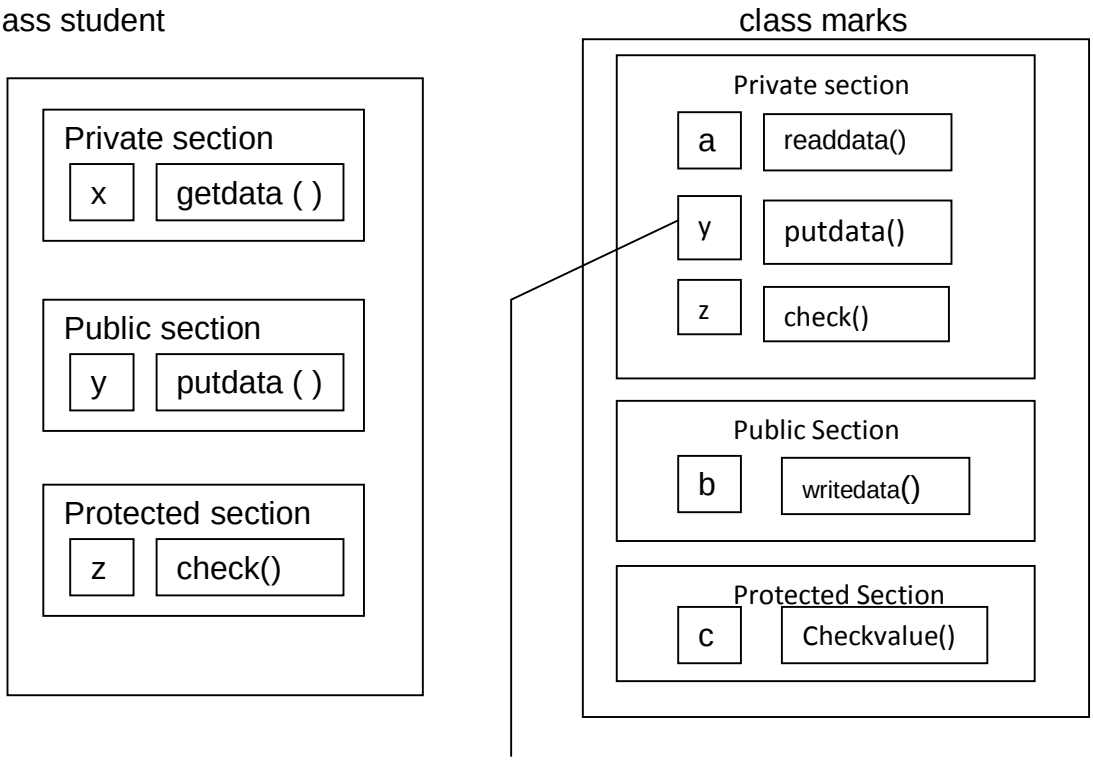


Fig .Private derivation of a class

As it is clear from the figure that the data present in public and protected section of base class become the private members of derived class. The data in private section of base class cannot be inherited.

8.5.3 The Protected visibility mode

We are using the same example but the derivation is done in protected mode.

```
class student
{Inheritance Extending Classes :: 139
// same as in previous example
};
class marks : protected student
{
```

};

The following figure illustrates the protected derivation in the classes.

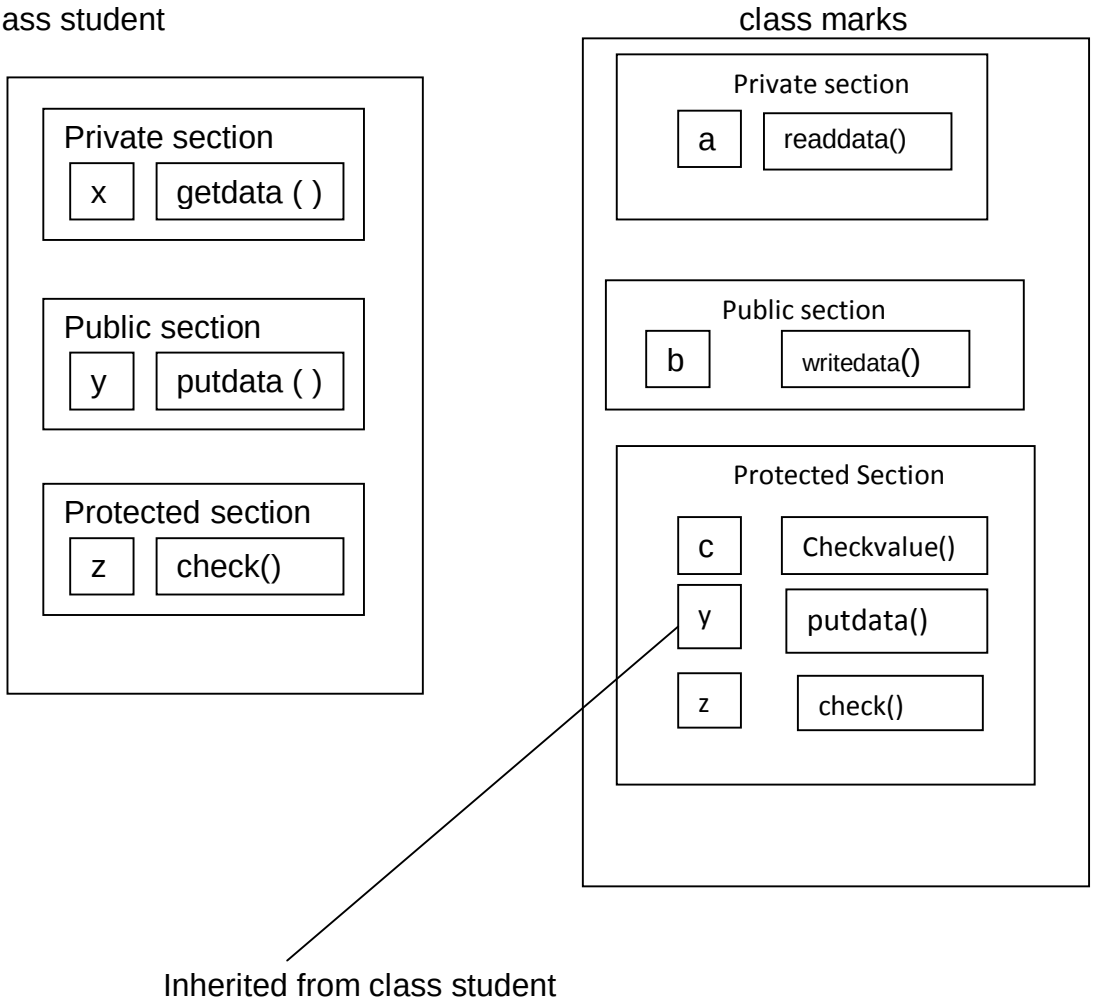


Fig .Protected derivation of a class

The data present in private section of base class cannot be inherited. The difference between private and protected section is that data present in protected section can be inherited. Otherwise both the section cannot be accessed by the object of the class.

8.6 Inherit private members of base class

C++ inheritance is very similar to a parent-child relationship. When a class is inherited all the functions and data member are inherited, although not all of them will be accessible by the member functions of the derived class. But there are some exceptions to it too.

Some of the exceptions to be noted in C++ inheritance are as follows.

1.The constructor and destructor of a base class are not inherited
2. The assignment operator is not inherited
3. The friend functions and friend classes of the base class are also not inherited.
There are some points to be remembered about C++ inheritance. The protected and public variables or members of the base class are all accessible in the derived class. But a private member variable not accessible by a derived class.

It is a well known fact that the private and protected members are not accessible outside the class. But a derived class is given access to protected members of the base class.

Points to remember:

C++ inheritance is very similar to a parent-child relationship.

A derived class is defined by specifying its relationship with the base class using visibility mode.

The private and protected members are not accessible outside the class.

The old class is referred to as the base class and new one is called the derived class.

A derived class with several base classes is called multiple inheritance.

The public derivation does not change the access specifiers for inherited members in the derived class

EXERCISE

Question Answers:

1. What is inheritance? What are base class and derived class ?

2. What are the different forms of inheritance?

3. What are the three modes of inheritance?

5. What is the difference between private and protected sections ?

6. What are the needs of inheritance?

Fill in the blanks:

(a) The base class is also called .....

(b) The derived class can be derived from base class in ..... or ..... way.

(c) By default base class is visible as ..... mode in derived class.

(d) When a derived class is derived from more than one base class then the inheritance is called ..... inheritance.

State whether the following are True or False.

(a) Inheritance means child receiving certain traits from parents.

(b) The default base class is visible as public mode in derived class.

(c) When a derived class is derived from more than one base class then the inheritance is called hierarchical inheritance.

(d) The base class is called abstract class

(e) Private data of base class can be inherited

Revision of class XI

Introduction To C Programming Language

C is a general-purpose, high-level language that was originally developed by Dennis M. Ritchie to develop the UNIX operating system at Bell Labs. C was originally first implemented on the DEC PDP-11 computer in 1972.

In 1978, Brian Kernighan and Dennis Ritchie produced the first publicly available description of C, now known as the K&R standard.

The UNIX operating system, the C compiler, and essentially all UNIX applications programs have been written in C. The C has now become a widely used professional language for various reasons.

Easy to learn

Structured language

It produces efficient programs.

It can handle low-level activities.

It can be compiled on a variety of computer platforms.

Facts about C

C was invented to write an operating system called UNIX.

C is a successor of B language which was introduced around 1970

The language was formalized in 1988 by the American National Standard Institute (ANSI).

The UNIX OS was totally written in C by 1973.

Today C is the most widely used and popular System Programming Language.

Most of the state-of-the-art softwares have been implemented using C.

Today's most popular Linux OS and RBDMS MySQL have been written in C.

Why to use C?

C was initially used for system development work, in particular the programs that make-up the operating system. C was adopted as a system development language because it produces code that runs nearly as fast as code written in assembly language. Some examples of the use of C might be:

Operating Systems

Language Compilers

Assemblers

Text Editors

Print Spoolers

Network Drivers

Modern Programs

Databases

Language Interpreters

Utilities

C Programs

Before we study basic building blocks of the C programming language, let us look a bare minimum C program structure so that we can take it as a reference in upcoming chapters.

C Hello World Example

A C program basically consists of the following parts:

Preprocessor Commands

Functions

Variables

Statements & Expressions

Comments

Let us look at a simple code that would print the words "Hello World":

```
#include <stdio.h>

int main()
{
   /* my first program in C */
   printf("Hello, World! \n");
```

```
   return 0;
}
```

Let us look various parts of the above program:

1. The first line of the program *#include <stdio.h>* is a preprocessor command, which tells a C compiler to include stdio.h file before going to actual compilation.
2. The next line *int main()* is the main function where program execution begins.
3. The next line /*...*/ will be ignored by the compiler and it has been put to add additional comments in the program. So such lines are called comments in the program.
4. The next line *printf(...)* is another function available in C which causes the message "Hello, World!" to be displayed on the screen.
5. The next line **return 0;** terminates main()function and returns the value 0.

Compile & Execute C Program:

Lets look at how to save the source code in a file, and how to compile and run it. Following are the simple steps:

1. Open a text editor and add the above-mentioned code.
2. Save the file as *hello.c*
3. Open a command prompt and go to the directory where you saved the file.
4. Type *gcc hello.c* and press enter to compile your code.
5. If there are no errors in your code the command prompt will take you to the next line and would generate *a.out* executable file.
6. Now, type *a.out* to execute your program.
7. You will be able to see *"Hello World"* printed on the screen.out

```
Hello, World!
```

You have seen a basic structure of C program, so it will be easy to understand other basic building blocks of the C programming language.

Tokens in C

A C program consists of various tokens and a token is either a keyword, an identifier, a constant, a string literal, or a symbol. For example, the following C statement consists of five tokens:

```
printf("Hello, World! \n");
```

The individual tokens are:

```
printf
(
"Hello, World! \n"
)
;
```

Semicolons ;

In C program, the semicolon is a statement terminator. That is, each individual statement must be ended with a semicolon. It indicates the end of one logical entity.

For example, following are two different statements:

```
printf("Hello, World! \n");
return 0;
```

Comments

Comments are like helping text in your C program and they are ignored by the compiler. They start with /* and terminates with the characters */ as shown below:

```
/* my first program in C */
```

You cannot have comments within comments and they do not occur within a string or character literals.

Identifiers

A C identifier is a name used to identify a variable, function, or any other user-defined item. An identifier starts with a letter A to Z or a to z or an underscore _ followed by zero or more letters, underscores, and digits (0 to 9).

C does not allow punctuation characters such as @, $, and % within identifiers. C is a **case sensitive**programming language. Thus, *Manpower* and *manpower* are two different identifiers in C. Here are some examples of acceptable identifiers:

```
mohd       zara    abc   move_name  a_123
myname50   _temp   j     a23b9      retVal
```

Keywords

The following list shows the reserved words in C. These reserved words may not be used as constant or variable or any other identifier names.

auto	else	long	switch
break	enum	register	typedef
case	extern	return	union
char	float	short	unsigned
const	for	signed	void
continue	goto	sizeof	volatile
default	if	static	while
do	int	struct	_Packed
double			

Whitespace in C

A line containing only whitespace, possibly with a comment, is known as a blank line, and a C compiler totally ignores it.

Whitespace is the term used in C to describe blanks, tabs, newline characters and comments. Whitespace separates one part of a statement from another and enables the compiler to identify where one element in a statement, such as int, ends and the next element begins. Therefore, in the following statement:

```
int age;
```

There must be at least one whitespace character (usually a space) between int and age for the compiler to be able to distinguish them. On the other hand, in the following statement:

```
fruit = apples + oranges;   // get the total fruit
```

No whitespace characters are necessary between fruit and =, or between = and apples, although you are free to include some if you wish for readability purpose.

Control flow

C provides two styles of flow control:

Branching
Looping

Branching is deciding what actions to take and looping is deciding how many times to take a certain action.

Branching:

Branching is so called because the program chooses to follow one branch or another.

if statement

This is the most simple form of the branching statements.

It takes an expression in parenthesis and an statement or block of statements. if the expression is true then the statement or block of statements gets executed otherwise these statements are skipped.

NOTE: Expression will be assumed to be true if its evaulated values is non-zero.

if statements take the following form:

```
if (expression)
  statement;

or

if (expression)
  {
    Block of statements;
  }
```

```
or

if (expression)
  {
    Block of statements;
  }
else
  {
    Block of statements;
  }

or

if (expression)
  {
    Block of statements;
  }
else if(expression)
  {
    Block of statements;
  }
else
  {
    Block of statements;
  }
```

? : Operator

The ? : operator is just like an if ... else statement except that because it is an operator you can use it within expressions.

? : is a ternary operator in that it takes three values, this is the only ternary operator C has.

? : takes the following form:

```
if condition is true ? then X return value : otherwise Y value;
```

switch statement:

The switch statement is much like a nested if .. else statement. Its mostly a matter of preference which you use, switch statement can be slightly more efficient and easier to read.

```
switch( expression )
    {
       case constant-expression1:   statements1;
       [case constant-expression2:  statements2;]
       [case constant-expression3:  statements3;]
       [default : statements4;]
    }
```

Using break keyword:

If a condition is met in switch case then execution continues on into the next case clause also if it is not explicitly specified that the execution should exit the switch statement. This is achieved by using *break* keyword.

Try out given example Show Example

What is default condition:

If none of the listed conditions is met then default condition executed.

Try out given example Show Example

Looping

Loops provide a way to repeat commands and control how many times they are repeated. C provides a number of looping way.

while loop

The most basic loop in C is the while loop.A while statement is like a repeating if statement. Like an If statement, if the test condition is true: the statments get executed. The difference is that after the statements have been executed, the test condition is checked again. If it is still true the statements get executed again.This cycle repeats until the test condition evaluates to false.

Basic syntax of while loop is as follows:

```
while ( expression )
{
   Single statement
   or
   Block of statements;
```

```
}
```

for loop

for loop is similar to while, it's just written differently. for statements are often used to proccess lists such a range of numbers:

Basic syntax of for loop is as follows:

```
for( expression1; expression2; expression3)
{
   Single statement
   or
   Block of statements;
}
```

In the above syntax:

expression1 - Initialisese variables.
expression2 - Condtional expression, as long as this condition is true, loop will keep executing.
expression3 - expression3 is the modifier which may be simple increment of a variable.

do...while loop

do ... while is just like a while loop except that the test condition is checked at the end of the loop rather than the start. This has the effect that the content of the loop are always executed at least once.

Basic syntax of do...while loop is as follows:

```
do
{
   Single statement
   or
   Block of statements;
}while(expression);
```

break and continue statements

C provides two commands to control how we loop:

break -- exit form loop or switch.
continue -- skip 1 iteration of loop.

You already have seen example of using break statement. Here is an example showing usage of**continue** statement.

```
#include

main()
{
   int i;
   int j = 10;

   for( i = 0; i <= j; i ++ )
   {
      if( i == 5 )
      {
         continue;
      }
      printf("Hello %d\n", i );
   }
}
```

This will produce following output:

```
Hello 0
Hello 1
Hello 2
Hello 3
Hello 4
Hello 6
Hello 7
Hello 8
Hello 9
Hello 10
```

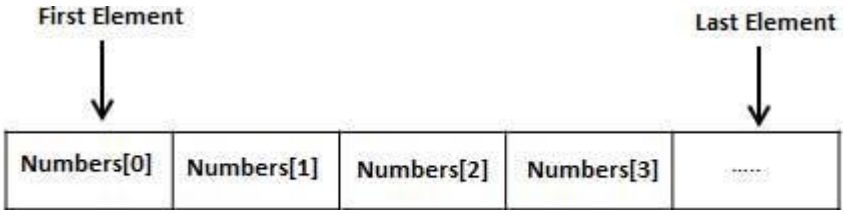
Arrays

C programming language provides a data structure called **the array**, which can store a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

Instead of declaring individual variables, such as number0, number1, ..., and number99, you declare one array variable such as numbers and use numbers[0], numbers[1], and

..., numbers[99] to represent individual variables. A specific element in an array is accessed by an index.

All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.



Declaring Arrays

To declare an array in C, a programmer specifies the type of the elements and the number of elements required by an array as follows:

```
type arrayName [ arraySize ];
```

This is called a *single-dimensional* array. The **arraySize** must be an integer constant greater than zero and **type** can be any valid C data type. For example, to declare a 10-element array called **balance** of type double, use this statement:

```
double balance[10];
```

Now *balance* is a variable array which is sufficient to hold upto 10 double numbers.

Initializing Arrays

You can initialize array in C either one by one or using a single statement as follows:

```
double balance[5] = {1000.0, 2.0, 3.4, 7.0, 50.0};
```

The number of values between braces { } can not be larger than the number of elements that we declare for the array between square brackets [ ].

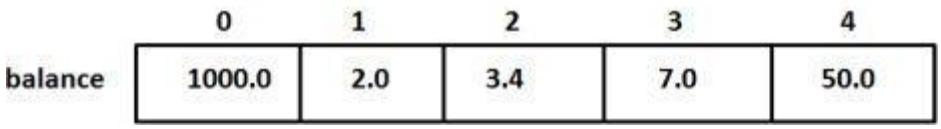
If you omit the size of the array, an array just big enough to hold the initialization is created. Therefore, if you write:

```
double balance[] = {1000.0, 2.0, 3.4, 7.0, 50.0};
```

You will create exactly the same array as you did in the previous example. Following is an example to assign a single element of the array:

```
balance[4] = 50.0;
```

The above statement assigns element number 5th in the array with a value of 50.0. All arrays have 0 as the index of their first element which is also called base index and last index of an array will be total size of the array minus 1. Following is the pictorial representation of the same array we discussed above:



Accessing Array Elements

An element is accessed by indexing the array name. This is done by placing the index of the element within square brackets after the name of the array. For example:

```
double salary = balance[9];
```

The above statement will take 10th element from the array and assign the value to salary variable. Following is an example which will use all the above mentioned three concepts viz. declaration, assignment and accessing arrays:

```
#include <stdio.h>

int main ()
{
   int n[ 10 ]; /* n is an array of 10 integers */
   int i,j;

   /* initialize elements of array n to 0 */
   for ( i = 0; i < 10; i++ )
   {
      n[ i ] = i + 100; /* set element at location i to i + 100 */
   }

   /* output each array element's value */
   for (j = 0; j < 10; j++ )
   {
      printf("Element[%d] = %d\n", j, n[j] );
   }

   return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Element[0] = 100
```

```
Element[1] = 101
Element[2] = 102
Element[3] = 103
Element[4] = 104
Element[5] = 105
Element[6] = 106
Element[7] = 107
Element[8] = 108
Element[9] = 109
```