CHAPTER 1

INTRODUCTION TO C++

The 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.

The most important thing while learning C++ is to focus on concepts.

The purpose of learning a programming language is to become a better programmer that is, to become more effective at designing and implementing new systems and at maintaining old ones.

C++ supports a variety of programming styles. You can write in the style of Fortran, C, Smalltalk, etc., in any language. Each style can achieve its aims effectively while maintaining runtime and space efficiency.

- C++ is a middle-level programming language developed by Bjarne Stroustrup starting in 1979 at Bell Labs. C++ runs on a variety of platforms, such as Windows, Mac OS, and the various versions of UNIX.
- C++ is a statically typed, compiled, general-purpose, case-sensitive, free-form programming language that supports procedural, object-oriented, and generic programming.
- C++ is regarded as a **middle-level** language, as it comprises a combination of both high-level and low-level language features.
- C++ was developed by Bjarne Stroustrup starting in 1979 at Bell Labs in Murray Hill, New Jersey, as an enhancement to the C language and originally named C with Classes but later it was renamed C++ in 1983.
- C++ is a superset of C, and that virtually any legal C program is a legal C++ program.

Object-Oriented Programming

C++ fully supports object-oriented programming, including the four pillars of object-oriented development –

- Encapsulation
- Abstraction
- Polymorphism
- Static Binding

- Dynamic Binding
- Inheritance
- Message Passing
- Class
- Object

Object

This is the basic unit of object oriented programming. That is both data and function that operate on data are bundled as a unit called as object.

Class

When you define a class, you define a blueprint for an object. This doesn't actually define any data, but it does define what the class name means, that is, what an object of the class will consist of and what operations can be performed on such an object.

Abstraction

Data abstraction refers to, providing only essential information to the outside world and hiding their background details, i.e., to represent the needed information in program without presenting the details.

For example, a database system hides certain details of how data is stored and created and maintained. Similar way, C++ classes provides different methods to the outside world without giving internal detail about those methods and data.

Encapsulation

Encapsulation is placing the data and the functions that work on that data in the same place. While working with procedural languages, it is not always clear which functions work on which variables but object-oriented programming provides you framework to place the data and the relevant functions together in the same object.

Inheritance

One of the most useful aspects of object-oriented programming is code reusability. As the name suggests Inheritance is the process of forming a new class from an existing class that is from the existing class called as base class, new class is formed called as derived class.

This is a very important concept of object-oriented programming since this feature helps to reduce the code size.

Polymorphism

The ability to use an operator or function in different ways in other words giving different meaning or functions to the operators or functions is called polymorphism. Poly refers to many. That is a single function or an operator functioning in many ways different upon the usage is called polymorphism.

Overloading

The concept of overloading is also a branch of polymorphism. When the exiting operator or function is made to operate on new data type, it is said to be overloaded.

Use of C++

- C++ is used by hundreds of thousands of programmers in essentially every application domain.
- C++ is being highly used to write device drivers and other software that rely on direct manipulation of hardware under realtime constraints.
- C++ is widely used for teaching and research because it is clean enough for successful teaching of basic concepts.
- Anyone who has used either an Apple Macintosh or a PC running Windows has indirectly used C++ because the primary user interfaces of these systems are written in C++.

When we consider a C++ program, it can be defined as a collection of objects that communicate via invoking each other's methods. Let us now briefly look into what a class, object, methods, and instant variables mean.

- **Object** – Objects have states and behaviors. Example: A dog has states - color, name, breed as well as behaviors - wagging, barking, eating. An object is an instance of a class.
- **Class** – A class can be defined as a template/blueprint that describes the behaviors/states that object of its type support.

- **Methods** – A method is basically a behavior. A class can contain many methods. It is in methods where the logics are written, data is manipulated and all the actions are executed.
- **Instance Variables** – Each object has its unique set of instance variables. An object's state is created by the values assigned to these instance variables.

Applications of C++

As a middle-level language, C combines benefits of both low machine level languages and high-level developer friendly languages. Further, it is fast, structured, portable and has a rich library. These features make C a general purpose programming language, and hence, it finds application across every domain in programming world.

A super set of C, C++ is an object-oriented programming language and incorporates all the features offered by C. C++ started its journey as C with classes. Gradually, it has evolved and despite the popularity of other programming languages like C# and Java, C, C++ holds its own as one of the most widely used languages for scripting. In applications, C++ is ubiquitous.

Real-World Applications of C++

1. Games:

C++ overrides the complexities of 3D games, optimizes resource management and facilitates multiplayer with networking. The language is extremely fast, allows procedural programming for CPU intensive functions and provides greater control over hardware, because of which it has been widely used in development of gaming engines. For instance, the science fiction game Doom 3 is cited as an example of a game that used C++ well and the Unreal Engine, a suite of game development tools, is written in C++.

2. Graphic User Interface (GUI) based applications:

Many highly used applications, such as Image Ready, Adobe Premier, Photoshop and Illustrator, are scripted in C++.

3. Web Browsers:

With the introduction of specialized languages such as PHP and Java, the adoption of C++ is limited for scripting of websites and web applications. However, where speed and reliability are required, C++ is still preferred. For instance, a part of Google's back-end is coded in C++, and the rendering engine of a few open source projects, such as web browser Mozilla Firefox and email client Mozilla Thunderbird, are also scripted in the programming language.

4. Advance Computations and Graphics:

C++ provides the means for building applications requiring real-time physical simulations, high-performance image processing, and mobile sensor applications. Maya 3D software, used for integrated 3D modeling, visual effects and animation, is coded in C++.

5. Database Software:

C++ and C have been used for scripting MySQL, one of the most popular database management software. The software forms the backbone of a variety of database-based enterprises, such as Google, Wikipedia, Yahoo and YouTube etc.

6. Operating Systems:

C++ forms an integral part of many of the prevalent operating systems including Apple's OS X and various versions of Microsoft Windows, and the erstwhile Symbian mobile OS.

7. Enterprise Software:

C++ finds a purpose in banking and trading enterprise applications, such as those deployed by Bloomberg and Reuters. It is also used in development of advanced software, such as flight simulators and radar processing.

8. Medical and Engineering Applications:

Many advanced medical equipments, such as MRI machines, use C++ language for scripting their software. It is also part of engineering applications, such as high-end CAD/CAM systems.

9. Compilers:

A host of compilers including Apple C++, Bloodshed Dev-C++, Clang C++ and MINGW make use of C++ language. C and its successor C++ are leveraged for diverse software and platform development requirements, from operating systems to graphic designing applications. Further, these languages have assisted in the development of new languages for special purposes like C#, Java, PHP, Verilog etc.

C++ Program Structure

Let us look at a simple code that would print the words *Hello World*.

Program 1.1 :

```
#include <iostream.h>
// main() is where program execution begins.
 Void  main()
{
   cout << "Hello World"; // prints Hello World
   return 0;
}
```

Let us look at the various parts of the above program −

- The C++ language defines several headers, which contain information that is either necessary or useful to your program. For this program, the header <iostream.h> is needed.
- The line '// main() is where program execution begins.' is a single-line comment available in C++. Single-line comments begin with // and stop at the end of the line.
- The line void main() is the main function where program execution begins.
- The next line cout << "Hello World"; causes the message "Hello World" to be displayed on the screen.
- The next line return 0; terminates main( )function and causes it to return the value 0 to the calling process.

Semicolons and Blocks in C++

In C++, 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 three different statements −

```
x = y;
y = y + 1;
add(x, y);
```

A block is a set of logically connected statements that are surrounded by opening and closing braces. For example −

```
{
  cout << "Hello World"; // prints Hello World
  return 0;
}
```

C++ does not recognize the end of the line as a terminator. For this reason, it does not matter where you put a statement in a line. For example −

```
x = y;
y = y + 1;
add(x, y);
```

is the same as

```
x = y; y = y + 1; add(x, y);
```

Identifiers

A C++ identifier is a name used to identify a variable, function, class, module, 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. There are 46 keywords in c++.

asm	enum	public	union
auto	extern	register	unsigned
break	float	return	virtual
case	for	short	void
catch	friend	signed	volatile
char	goto	sizeof	while
class	if	static	
const	inline	struct	
continue	int	switch	
default	long	template	
delete	new	this	
do	operator	throw	
double	private	try	
else	protected	typedef	

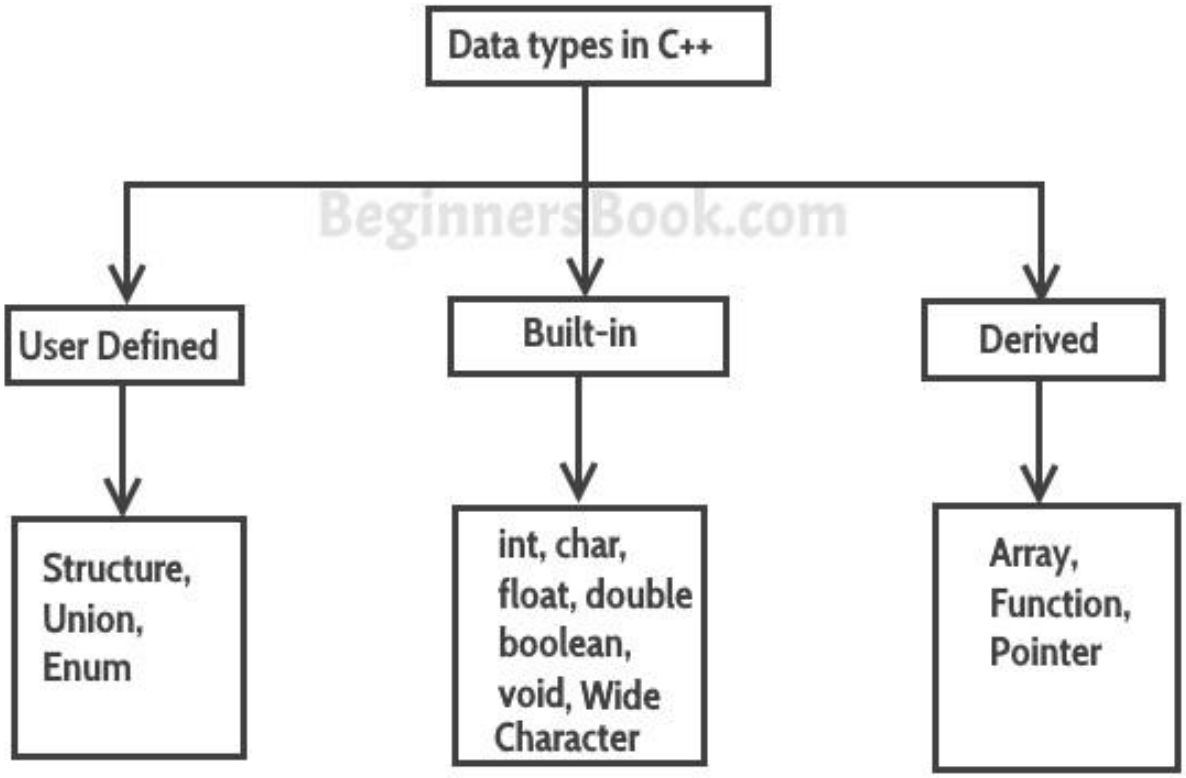
CHAPTER 2

DATA TYPES AND OPERATORS

Data Types

Data types define the type of data a variable can hold, for example an integer variable can hold integer data, a character type variable can hold character data etc.

Data types in C++ are categorised in three groups: **Built-in**, **user-defined** and **Derived**.



Built in data types

char: For characters. Size 1 byte.

char ch = 'A';

int: For integers. Size 2 bytes.

int num = 100;

float: For single precision floating point. Size 4 bytes.

float num = 123.78987;

double: For double precision floating point. Size 8 bytes.

double num = 10098.98899;

bool: For booleans, true or false.

bool b = true;

User-defined data types

We have three types of user-defined data types in C++
1. struct
2. union
3. enum

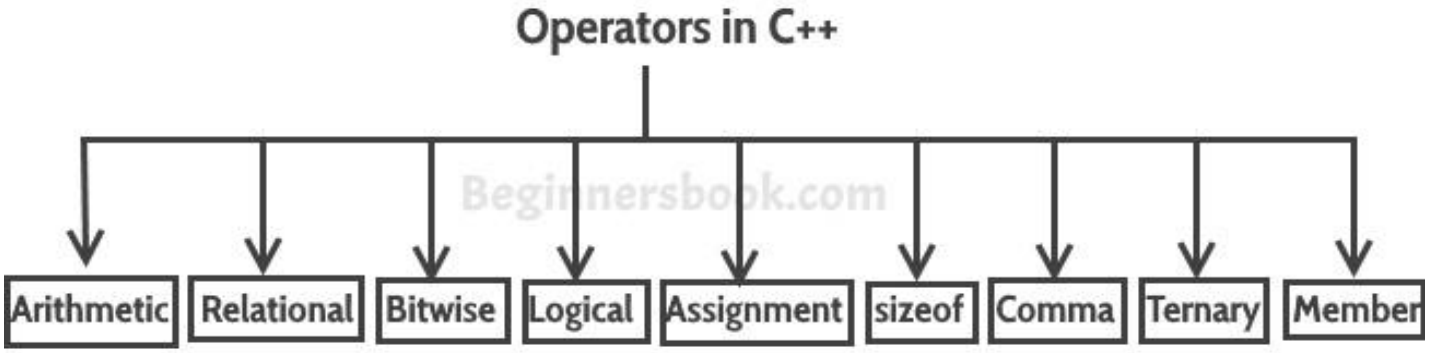
Derived data types

We have three types of derived-defined data types in C++
1. Array
2. Function
3. Pointer

Operators

Operator represents an action. For example + is an operator that represents addition. An operator works on two or more operands and produce an output. For example 3+4+5 here + operator works on three operands and produce 12 as output.

Types of Operators in C++



1) Basic Arithmetic Operators
2) Assignment Operators
3) Auto-increment and Auto-decrement Operators
4) Logical Operators
5) Comparison (relational) operators
6) Bitwise Operators
7) Ternary Operator

1) Basic Arithmetic Operators

Basic arithmetic operators are: +, -, *, /, %
+ is for addition.

– is for subtraction.

***** is for multiplication.

/ is for division.

% is for modulo.

Program 2.1 :

Example of Arithmetic Operators

```
#include <iostream.h>
int main()
{
 int num1 = 240;
 int num2 = 40;
 cout<<"num1 + num2: "<<(num1 + num2)<<endl;
 cout<<"num1 - num2: "<<(num1 - num2)<<endl;
 cout<<"num1 * num2: "<<(num1 * num2)<<endl;
 cout<<"num1 / num2: "<<(num1 / num2)<<endl;
 cout<<"num1 % num2: "<<(num1 % num2)<<endl;
 return 0;
}
```

Output:

```
num1 + num2: 280
num1 - num2: 200
num1 * num2: 9600
num1 / num2: 6
num1 % num2: 0
```

2) Assignment Operators

Assignments operators in C++ are: =, +=, -=, *=, /=, %=

num2 = num1 would assign value of variable num1 to the variable.

num2+ =num1 is equal to num2 = num2+num1

num2- =num1 is equal to num2 = num2-num1

num2* =num1 is equal to num2 = num2*num1

num2/ =num1 is equal to num2 = num2/num1

num2% =num1 is equal to num2 = num2%num1

Program 2.2 :

Example of Assignment Operators

```
#include <iostream.h>
int main()
{
 int num1 = 240;
 int num2 = 40;
 num2 = num1;
 cout<<"= Output: "<<num2<<endl;
 num2 += num1;
 cout<<"+= Output: "<<num2<<endl;
 num2 -= num1;
 cout<<"-= Output: "<<num2<<endl;
 num2 *= num1;
 cout<<"*= Output: "<<num2<<endl;
 num2 /= num1;
 cout<<"/= Output: "<<num2<<endl;
 num2 %= num1;
 cout<<"%= Output: "<<num2<<endl;
 return 0;
}
```

Output:

```
= Output: 240
+= Output: 480
-= Output: 240
*= Output: 57600
/= Output: 240
%= Output: 0
```

3) Auto-increment and Auto-decrement Operators

++ and —
num++ is equivalent to num=num+1;

num–- is equivalent to num=num-1;

Program 2.3 :

Example of Auto-increment and Auto-decrement Operators

```
#include <iostream.h>
int main()
{
  int num1 = 240;
  int num2 = 40;
  num1++;
  num2--;
  cout<<"num1++ is: "<<num1<<endl;
  cout<<"num2-- is: "<<num2;
  return 0;
}
```

Output:

```
num1++ is: 241
num2-- is: 39
```

4) Logical Operators

Logical Operators are used with binary variables. They are mainly used in conditional statements and loops for evaluating a condition.

Logical operators in C++ are: &&, ||, !

Let’s say we have two boolean variables b1 and b2.

b1&&b2 will return true if both b1 and b2 are true else it would return false.

b1||b2 will return false if both b1 and b2 are false else it would return true.

!b1 would return the opposite of b1, that means it would be true if b1 is false and it would return false if b1 is true.

Program 2.4 : Example of Logical Operators

```
#include <iostream.h>
int main()
{
  bool b1 = true;
  bool b2 = false;
  cout<<"b1 && b2: "<<(b1&&b2)<<endl;
  cout<<"b1 || b2: "<<(b1||b2)<<endl;
  cout<<"!(b1 && b2): "<<!(b1&&b2);
  return 0;
}
```

Output:

```
b1 && b2: 0
b1 || b2: 1
!(b1 && b2): 1
```

5) Relational operators

We have six relational operators in C++: ==, !=, >, <, >=, <=

== returns true if both the left side and right side are equal

!= returns true if left side is not equal to the right side of operator.

> returns true if left side is greater than right.

< returns true if left side is less than right side.

>= returns true if left side is greater than or equal to right side.

<= returns true if left side is less than or equal to right side.

Program 2.5 :

Example of Relational operators

```
#include <iostream.h>
int main()
{
  int num1 = 240;
  int num2 =40;
  if (num1==num2) {
    cout<<"num1 and num2 are equal"<<endl;
  }
  else{
    cout<<"num1 and num2 are not equal"<<endl;
  }
  if( num1 != num2 ){
    cout<<"num1 and num2 are not equal"<<endl;
  }
  else{
    cout<<"num1 and num2 are equal"<<endl;
  }
  if( num1 > num2 ){
    cout<<"num1 is greater than num2"<<endl;
  }
  else{
    cout<<"num1 is not greater than num2"<<endl;
  }
  if( num1 >= num2 ){
    cout<<"num1 is greater than or equal to num2"<<endl;
  }
  else{
    cout<<"num1 is less than num2"<<endl;
  }
  if( num1 < num2 ){
```

```
    cout<<"num1 is less than num2"<<endl;
  }
  else{
    cout<<"num1 is not less than num2"<<endl;
  }
  if( num1 <= num2){
    cout<<"num1 is less than or equal to num2"<<endl;
  }
  else{
    cout<<"num1 is greater than num2"<<endl;
  }
  return 0;
}
```

Output:

```
num1 and num2 are not equal
num1 and num2 are not equal
num1 is greater than num2
num1 is greater than or equal to num2
num1 is not less than num2
num1 is greater than num2
```

6) Bitwise Operators

There are six bitwise Operators: &, |, ^, ~, <<, >>

```
num1 = 11; /* equal to 00001011*/
num2 = 22; /* equal to 00010110 */
```

Bitwise operator performs bit by bit processing.
num1 & num2 compares corresponding bits of num1 and num2 and generates 1 if both bits are equal, else it returns 0. In our case it would return: 2 which is 00000010 because in the binary form of num1 and num2 only second last bits are matching.

num1 | num2 compares corresponding bits of num1 and num2 and generates 1 if either bit is 1, else it returns 0. In our case it would return 31 which is 00011111

num1 ^ num2 compares corresponding bits of num1 and num2 and generates 1 if they are not equal, else it returns 0. In our example it would return 29 which is equivalent to 00011101

~num1 is a complement operator that just changes the bit from 0 to 1 and 1 to 0. In our example it would return -12 which is signed 8 bit equivalent to 11110100

num1 << 2 is left shift operator that moves the bits to the left, discards the far left bit, and assigns the rightmost bit a value of 0. In our case output is 44 which is equivalent to 00101100

Note: In the example below we are providing 2 at the right side of this shift operator that is the reason bits are moving two places to the left side. We can change this number and bits would be moved by the number of bits specified on the right side of the operator. Same applies to the right side operator.

num1 >> 2 is right shift operator that moves the bits to the right, discards the far right bit, and assigns the leftmost bit a value of 0. In our case output is 2 which is equivalent to 00000010

Program 2.6 :

Example of Bitwise Operators

```
#include <iostream.h>
int main()
{
  int num1 = 11;  /* 11 = 00001011 */
  int num2 = 22;  /* 22 = 00010110 */
  int result = 0;
  result = num1 & num2;
  cout<<"num1 & num2: "<<result<<endl;
  result = num1 | num2;
  cout<<"num1 | num2: "<<result<<endl;
  result = num1 ^ num2;
  cout<<"num1 ^ num2: "<<result<<endl;
  result = ~num1;
  cout<<"~num1: "<<result<<endl;
  result = num1 << 2;
  cout<<"num1 << 2: "<<result<<endl;
```

```
  result = num1 >> 2;
  cout<<"num1 >> 2: "<<result;
  return 0;
}
```

Output:

```
num1 & num2: 2
num1 | num2: 31
num1 ^ num2: 29
~num1: -12
num1 << 2: 44 num1 >> 2: 2
```

7) Ternary Operator

This operator evaluates a boolean expression and assign the value based on the result.
Syntax:

```
variable num1 = (expression) ? value if true : value if false
```

If the expression results true then the first value before the colon (:) is assigned to the variable num1 else the second value is assigned to the num1.

Program 2.7 :

Example of Ternary Operator

```
#include <iostream.h>
int main()
{
 int num1, num2; num1 = 99;
 /* num1 is not equal to 10 that's why
  * the second value after colon is assigned
  * to the variable num2
  */
 num2 = (num1 == 10) ? 100: 200;
 cout<<"num2: "<<num2<<endl;
```

```
 /* num1 is equal to 99 that's why
  * the first value is assigned
  * to the variable num2
  */
 num2 = (num1 == 99) ? 100: 200;
 cout<<"num2: "<<num2;
 return 0;
}
```

Output:

```
num2: 200
num2: 100
```

Miscellaneous Operators

There are few other operators in C++ such as **Comma operator** and **sizeof operator**.

Operator Precedence

This determines which operator needs to be evaluated first if an expression has more than one operator. Operator with higher precedence at the top and lower precedence at the bottom.

Unary Operators
++ – – ! ~

Multiplicative
* / %

Additive
+ –

Shift
<< >> >>>

Relational
> >= < <=

Equality
== !=

Bitwise AND
&
Bitwise XOR
^
Bitwise OR
|
Logical AND
&&

Logical OR
||

Ternary
?:

Assignment
= += -= *= /= %= > >= < <= &= ^= |=

CHAPTER 3

CONDITIONAL STATEMENT AND LOOPING STATEMENT

Conditional Statement –

1)If
2)if…else
3)Nested if…else
4)Switch…Case

If else Statement

Sometimes we need to execute a block of statements only when a particular condition is met or not met. This is called **decision making**, as we are executing a certain code after making a decision in the program logic. For decision making in C++, we have four types of control statements (or control structures), which are as follows:

a) if statement
b) nested if statement
c) if-else statement
d) if-else-if statement

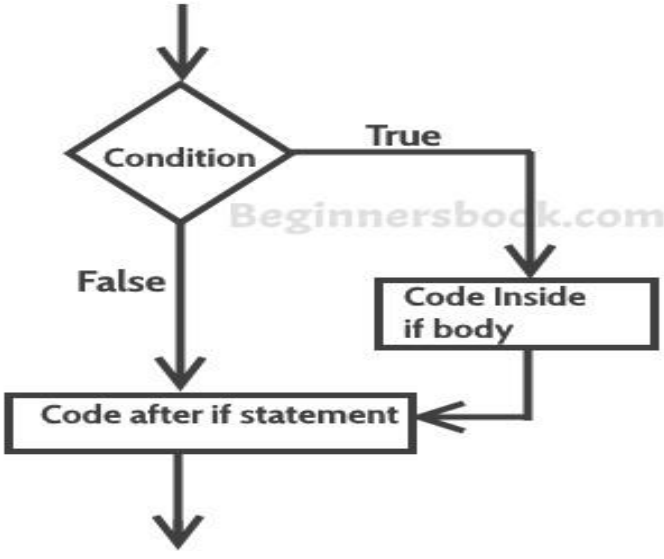
1) If statement

If statement consists a condition, followed by statement or a set of statements as shown below:

```
if(condition){
  Statement(s);
}
```

The statements inside **if** parenthesis (usually referred as if body) gets executed only when the given condition is true. If the condition is false then the statements inside if body are completely ignored.

Flow diagram of If statement



Program 3.1 :

Example of if statement

```
#include <iostream.h>
int main()
{
  int num=70;
  if( num < 100 )
  {
     cout<<"number is less than 100";
  }

  if(num > 100)
 {
     cout<<"number is greater than 100";
  }
  return 0;
}
```

Output:

```
number is less than 100
```

2) Nested if statement

When there is an if statement inside another if statement then it is called the **nested if statement**.
The structure of nested if looks like this:

```
if(condition_1) {
   Statement1(s);

   if(condition_2) {
      Statement2(s);
   }
}
```

Statement1 would execute if the condition_1 is true. Statement2 would only execute if both the conditions(condition_1 and condition_2) are true.

Program 3.2 :

Example of Nested if statement

```
#include <iostream.h>
int main()
{
  int num=90;

  if( num < 100 )
  {
    cout<<"number is less than 100"<<endl;
    if(num > 50)
   {
      cout<<"number is greater than 50";
    }
  }
  return 0;
}
```

Output:

```
number is less than 100
number is greater than 50
```

3) If else statement

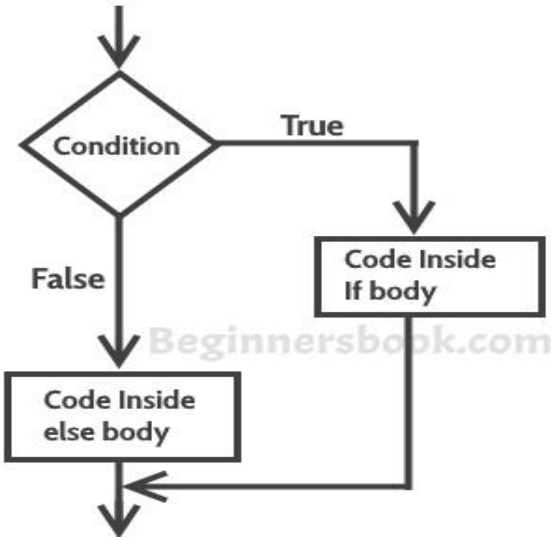
Sometimes you have a condition and you want to execute a block of code if condition is true and execute another piece of code if the same condition is false. This can be achieved in C++ using if-else statement.

This is how an if-else statement looks:

```
if(condition) {
  Statement(s);
}
else {
  Statement(s);
}
```

The statements inside "if" would execute if the condition is true, and the statements inside "else" would execute if the condition is false.

Flow diagram of if-else



Program 3.3 :

Example of if-else statement

```
#include <iostream.h>
int main()
{
  int num=66;
  if( num < 50 )
 {
    cout<<"num is less than 50";
  }
  else
 {
     cout<<"num is greater than or equal 50";
  }
  return 0;
}
```

Output:

```
num is greater than or equal 50
```

if-else-if Statement

if-else-if statement is used when we need to check multiple conditions. In this control structure we have only one "if" and one "else", however we can have multiple "else if" blocks. This is how it looks:

```
if(condition_1) {
   /*if condition_1 is true execute this*/
   statement(s);
}
else if(condition_2) {
   /* execute this if condition_1 is not met and
    * condition_2 is met
    */
   statement(s);
}
else if(condition_3) {
   /* execute this if condition_1 & condition_2 are
    * not met and condition_3 is met
    */
   statement(s);
}
.
.
.
else {
   /* if none of the condition is true
    * then these statements gets executed
    */
   statement(s);
}
```

Program 3.3 :

Example of if-else-if

```
#include <iostream.h>

int main()

{

   int num;

   cout<<"Enter an integer number between 1 & 99999: ";

   cin>>num;

   if(num <100 && num>=1)

{

      cout<<"Its a two digit number";
```

```
 }
  else if(num <1000 && num>=100)
{
    cout<<"Its a three digit number";
 }
  else if(num <10000 && num>=1000)
 {
    cout<<"Its a four digit number";
 }
  else if(num <100000 && num>=10000)
 {
    cout<<"Its a five digit number";
  }
  else
 {
    cout<<"number is not between 1 & 99999";
  }
  return 0;
}
```

Output:

```
Enter an integer number between 1 & 99999: 8976
Its a four digit number
```

4) Switch Case statement

Switch case statement is used when we have multiple conditions and we need to perform different action based on the condition. When we have multiple conditions and we need to execute a block of statements when a particular condition is satisfied. In such case either we can use lengthy if..else-if statement or switch case. The problem with lengthy if..else-if is that it becomes complex when we have several conditions. The switch case is a clean and efficient method of handling such scenarios.
The **syntax of Switch case** statement:

```
switch (variable or an integer expression)
{
    case constant:
    //C++ code
    ;
    case constant:
    //C++ code
    ;
    default:
    //C++ code
    ;
}
```

Switch Case statement is mostly used with break statement even though the break statement is optional. We will first see an example without break statement and then we will discuss switch case with break

Program 3.4 :

Example of Switch Case

```
#include <iostream.h>
  int main()
{
  int num=5;
  switch(num+2)
 {
     case 1:
       cout<<"Case1: Value is: "<<num<<endl;
     case 2:
       cout<<"Case2: Value is: "<<num<<endl;
     case 3:
       cout<<"Case3: Value is: "<<num<<endl;
     default:
       cout<<"Default: Value is: "<<num<<endl;
  }
  return 0;
}
```

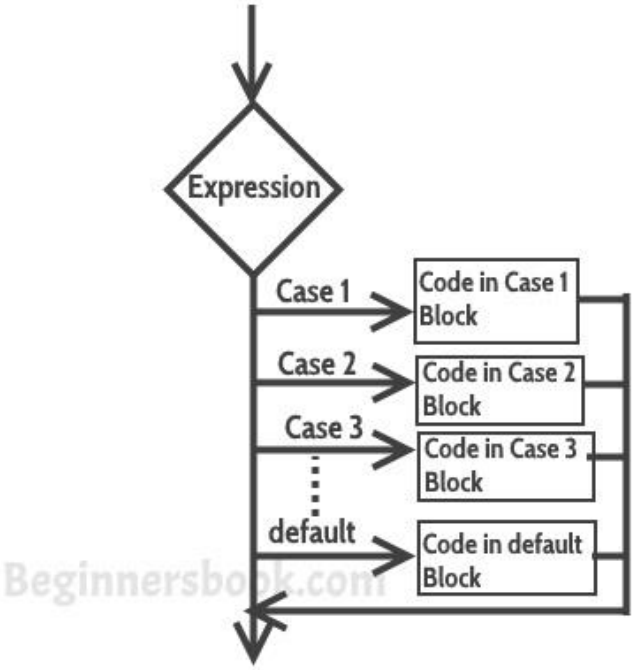
Output:

```
Default: Value is: 5
```

Explanation: In switch I gave an expression, you can give variable as well. I gave the expression num+2, where num value is 5 and after addition the expression resulted 7. Since there is no case defined with value 4 the default case got executed.

Switch Case Flow Diagram

It evaluates the value of expression or variable (based on whatever is given inside switch braces), then based on the outcome it executes the corresponding case.



Break statement in Switch Case

Before we discuss about break statement, Let's see what happens when we don't use break statement in switch case. See the example below:

Program 3.5 :

```
#include <iostream.h>
int main()
{
  int i=2;
  switch(i)
{
    case 1: cout<<"Case1 "<<endl;
```

```
    case 2: cout<<"Case2 "<<endl;
    case 3: cout<<"Case3 "<<endl;
    case 4: cout<<"Case4 "<<endl;
    default: cout<<"Default "<<endl;
  }
  return 0;
}
```

Output:

```
Case2
Case3
Case4
Default
```

In the above program, we have the variable i inside switch braces, which means whatever the value of variable i is, the corresponding case block gets executed. We have passed integer value 2 to the switch, so the control switched to the case 2, however we don't have break statement after the case 2 that caused the flow to continue to the subsequent cases till the end. However this is not what we wanted, we wanted to execute the right case block and ignore rest blocks. The solution to this issue is to use the break statement in after every case block.

Break statements are used when you want your program-flow to come out of the switch body. Whenever a break statement is encountered in the switch body, the execution flow would directly come out of the switch, ignoring rest of the cases. This is why you must end each case block with the break statement.

Program 3.6 :

```
#include <iostream.h>
int main()
{
  int i=2;
  switch(i)
  {
    case 1:
      cout<<"Case1 "<<endl;
      break;
```

```
    case 2:
      cout<<"Case2 "<<endl;
      break;
    case 3:
      cout<<"Case3 "<<endl;
      break;
    case 4:
      cout<<"Case4 "<<endl;
      break;
    default:
      cout<<"Default "<<endl;
   }
   return 0;
}
```

Output:

```
Case2
```

Looping Statement

1)For

2)While

3)do-while

1) For loop

A loop is used for executing a block of statements repeatedly until a particular condition is satisfied. For example, when you are displaying number from 1 to 100 you may want set the value of a variable to 1 and display it 100 times, increasing its value by 1 on each loop iteration.

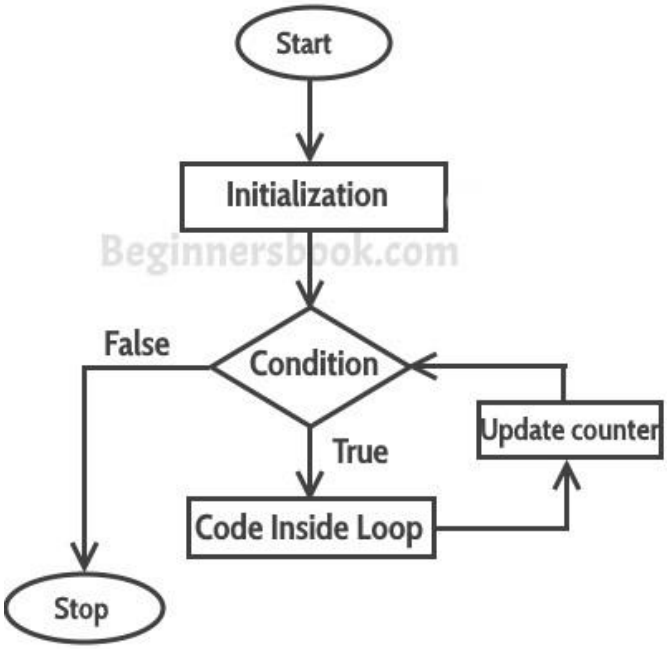
In C++ we have three types of basic loops: for, while and do-while.

Syntax of for loop

```
for(initialization; condition ; increment/decrement)
{
  C++ statement(s);
}
```

Flow of Execution of the for Loop

As a program executes, the interpreter always keeps track of which statement is about to be executed. We call this the control flow, or the flow of execution of the program.



First step: In for loop, initialization happens first and only once, which means that the initialization part of for loop only executes once.

Second step: Condition in for loop is evaluated on each loop iteration, if the condition is true then the statements inside for for loop body gets executed. Once the condition returns false, the statements in for loop does not execute and the control gets transferred to the next statement in the program after for loop.

Third step: After every execution of for loop's body, the increment/decrement part of for loop executes that updates the loop counter.

Fourth step: After third step, the control jumps to second step and condition is re-evaluated.

The steps from second to fourth repeats until the loop condition returns false.

Example of a Simple For loop

Here in the loop initialization part I have set the value of variable i to 1, condition is i<=6 and on each loop iteration the value of i increments by 1.

Program 3.7 :

```
#include <iostream.h>
int main(){
  for(int i=1; i<=6; i++)
  {
        cout<<"Value of variable i is: "<<i<<endl;
  }
  return 0;
}
```

Output:

```
Value of variable i is: 1
Value of variable i is: 2
Value of variable i is: 3
Value of variable i is: 4
Value of variable i is: 5
Value of variable i is: 6
```

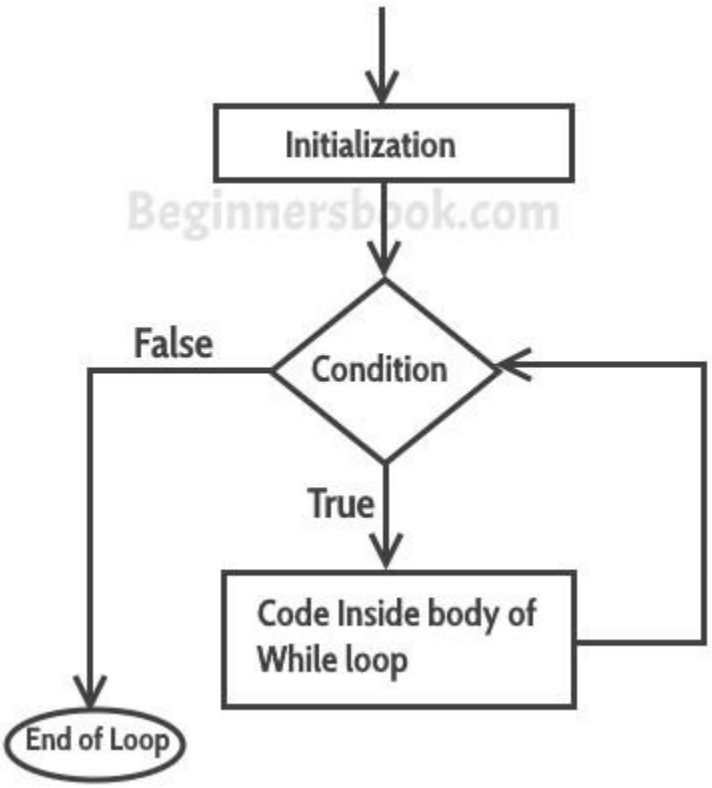
2) While loop

As discussed earlier, loops are used for executing a block of program statements repeatedly until the given loop condition returns false.

Syntax of while loop

```
while(condition)
{
  statement(s);
}
```

Flow Diagram of While loop



Program 3.8 :

While Loop example

```
#include <iostream.h>
int main()
{
  int i=1;
  while(i<=6)
 {
    cout<<"Value of variable i is: "<<i<<endl;
    i++;
  }
}
```

Output:

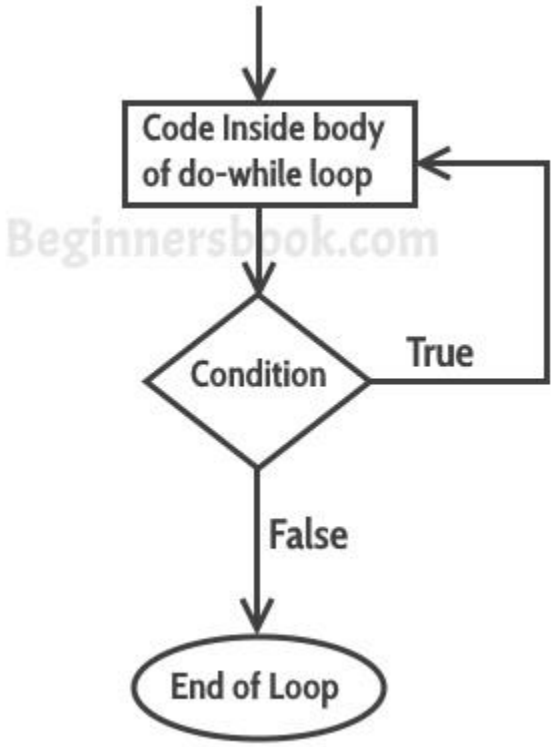
```
Value of variable i is: 1
Value of variable i is: 2
Value of variable i is: 3
Value of variable i is: 4
Value of variable i is: 5
Value of variable i is: 6
```

3)do-while loop

A loop is used for repeating a block of statements until the given loop condition returns false. do-while loop is similar to while loop, however there is a difference between them: In while loop, condition is evaluated first and then the statements inside loop body gets executed, on the other hand in do-while loop, statements inside do-while gets executed first and then the condition is evaluated.

Syntax of do-while loop

```
do
{
  statement(s);
} while(condition);
```



Program 3.9 :

do-while loop

```
#include <iostream.h>
int main()
{
  int num=1;
  do
 {
    cout<<"Value of num: "<<num<<endl;
    num++;
  }
  while(num<=6);
  return 0;
}
```

Output:

```
Value of num: 1
Value of num: 2
Value of num: 3
Value of num: 4
Value of num: 5
Value of num: 6
```

Continue Statement

Continue statement is used inside loops. Whenever a continue statement is encountered inside a loop, control directly jumps to the beginning of the loop for next iteration, skipping the execution of statements inside loop's body for the current iteration.

Syntax of continue statement

```
continue;
```

Example: continue statement inside for loop

As you can see that the output is missing the value 3, however the for loop iterate though the num value 0 to 6. This is because we have set a condition inside loop in such a way, that the continue statement is encountered when the num value is equal to 3. So for this iteration the loop skipped the cout statement and started the next iteration of loop.

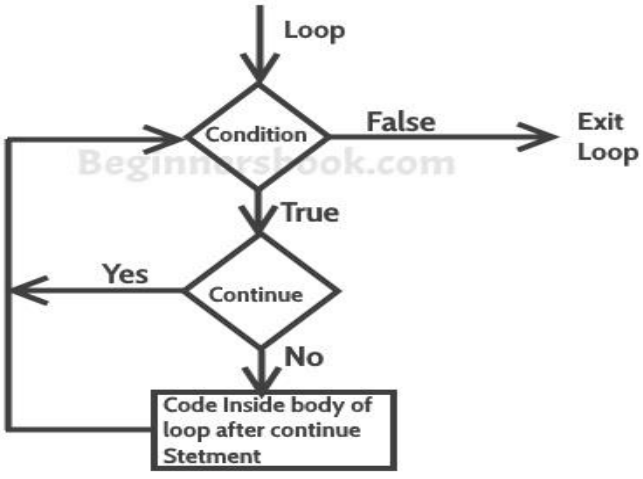
Program 3.10 :

```
#include <iostream.h>
int main()
{
  for (int num=0; num<=6; num++)
   {
     if (num==3)
    {
        continue;
     }
     cout<<num<<" ";
  }
  return 0;
}
```

Output:

```
0 1 2 4 5 6
```

Flow Diagram of Continue Statement



Program 3.11 :

Example: Use of continue in While loop

```
#include <iostream.h>
int main()
{
  int j=6;
  while (j >=0)
  {
    if (j==4)
   {
      j--;
      continue;
    }
    cout<<"Value of j: "<<j<<endl;
    j--;
  }
  return 0;
}
```

Output:

```
Value of j: 6
Value of j: 5
Value of j: 3
Value of j: 2
Value of j: 1
Value of j: 0
```

Program 3.12 :

Example of continue in do-While loop

```
#include <iostream.h>
int main()
{
  int j=4;
  do
   {
     if (j==7)
     {
       j++;
       continue;
     }
     cout<<"j is: "<<j<<endl;
     j++;
  }
  while(j<10);
  return 0;
}
```

Output:

```
j is: 4
j is: 5
j is: 6
j is: 8
j is: 9
```

Break statement

The **break statement** is used in following two scenarios:

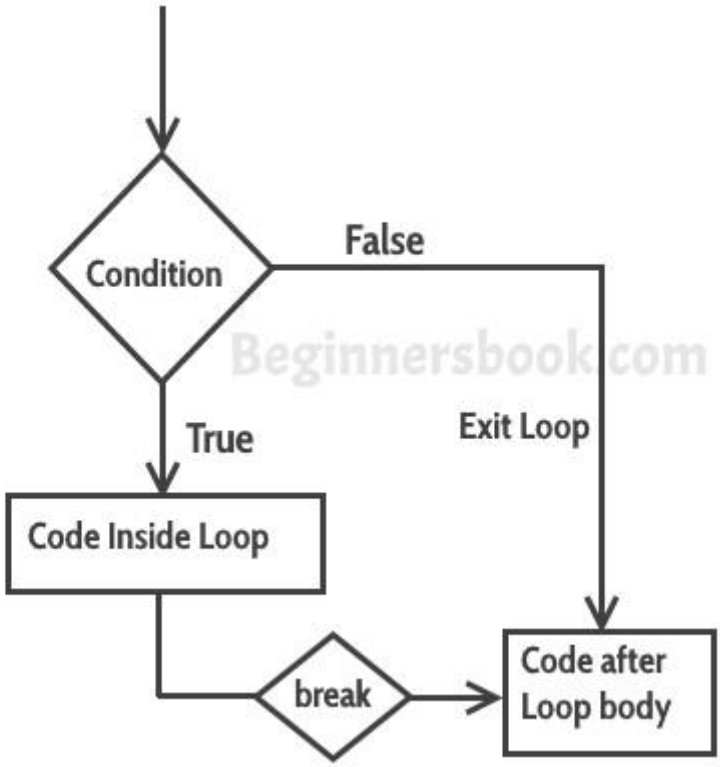
a) Use break statement to come out of the loop instantly. Whenever a break statement is encountered inside a loop, the control directly comes out of loop terminating it. It is used along with if statement, whenever used inside loop(see the example below) so that it occurs only for a particular condition.

b) It is used in switch case control structure after the case blocks. Generally all cases in switch case are followed by a break statement to avoid the subsequent cases (see the example below) execution. Whenever it is encountered in switch-case block, the control comes out of the switch-case body.

Syntax of break statement

```
break;
```

break statement flow diagram



Example – Use of break statement in a while loop

In the example below, we have a while loop running from 10 to 200 but since we have a break statement that gets encountered when the loop counter variable value reaches

12, the loop gets terminated and the control jumps to the next statement in program after the loop body.

Program 3.13 :

```
#include <iostream.h>
int main()
{
  int num =10;
  while(num<=200)
 {
    cout<<"Value of num is: "<<num<<endl;
    if (num==12)
   {
      break;
    }
    num++;
  }
  cout<<"Hey, I'm out of the loop";
  return 0;
}
```

Output:

```
Value of num is: 10
Value of num is: 11
Value of num is: 12
Hey, I'm out of the loop
```

Program 3.14 :

Example: break statement in for loop

```
#include <iostream.h>
int main()
 {
   int var;
   for (var =200; var>=10; var --)
   {
     cout<<"var: "<<var<<endl;
     if (var==197)
    {
       break;
     }
  }
  cout<<"Hey, I'm out of the loop";
  return 0;
}
```

Output:

```
var: 200
var: 199
var: 198
var: 197
Hey, I'm out of the loop
```

Program 3.15 :

Example: break statement in Switch Case

```
#include <iostream.h>
int main()
 {
   int num=2;
   switch (num)
 {
      case 1: cout<<"Case 1 "<<endl;
      break;
      case 2: cout<<"Case 2 "<<endl;
      break;
      case 3: cout<<"Case 3 "<<endl;
      break;
      default: cout<<"Default "<<endl;
   }
   cout<<"Hey, I'm out of the switch case";
   return 0;
}
```

Output:

```
Case 2
Hey, I'm out of the switch case
```

goto statement

The goto statement is used for transferring the control of a program to a given label. The syntax of goto statement looks like this:

```
goto label_name;
```

Program structure:

```
label1:
...
...
goto label2;
...
..
label2:
...
```

In a program we have any number of goto and label statements, the goto statement is followed by a label name, whenever goto statement is encountered, the control of the program jumps to the label specified in the goto statement.

goto statements are almost never used in any development as they are complex and makes your program much less readable and more error prone. In place of goto, you can use continue and break statement.

Program 3.16 :

Example of goto statement

```
#include <iostream.h>
int main()
{
  int num;
  cout<<"Enter a number: ";
  cin>>num;
  if (num % 2==0)
 {
    goto print;
  }
  else
 {
    cout<<"Odd Number";
  }
  print:
  cout<<"Even Number";
```

```
  return 0;
}
```

Output:

```
Enter a number: 42
Even Number
```

CHAPTER 4

FUNCTIONS

Functions

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 such that 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 known with various names like a method or a sub-routine or a procedure etc.

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.

Following is the source code for a function called max(). This function takes two parameters num1 and num2 and return the biggest of both −

```
// 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;
}
```

Function Declarations

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 important 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.

Calling 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 it's 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 –

Example 4.1 :

```
#include <iostream.h>

// 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;
}
```

max() function along with main() function and compiled the source code. While running final executable, it would produce the following result −

```
Max value is : 200
```

Inline Function

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 –

Example 4.2 :

```
#include <iostream>
 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;
}
```

Output:

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

Overloading (Operator and Function)

C++ allows you to specify more than one definition for a function name or an operator in the same scope, which is called **function overloading** and **operator overloading** respectively.

An overloaded declaration is a declaration that is 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).

When you call an overloaded **function** or **operator**, the compiler determines the most appropriate definition to use, by comparing the argument types you have used to call the function or operator with the parameter types specified in the definitions. The process of selecting the most appropriate overloaded function or operator is called **overload resolution**.

Function Overloading

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 cannot 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 –

Example 4.3 :

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

Output:

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

Operators Overloading

You can redefine or overload most of the built-in operators available in C++. Thus, a programmer can use operators with user-defined types as well.

Overloaded operators are functions with special names: the keyword "operator" followed by the symbol for the operator being defined. Like any other function, an overloaded operator has a return type and a parameter list.

```
Box operator+(const Box&);
```

declares the addition operator that can be used to add two Box objects and returns final Box object. Most overloaded operators may be defined as ordinary non-member functions or as class member functions. In case we define above function as non-member function of a class then we would have to pass two arguments for each operand as follows −

```
Box operator+(const Box&, const Box&);
```

Following is the example to show the concept of operator over loading using a member function. Here an object is passed as an argument whose properties will be accessed using this object, the object which will call this operator can be accessed using this operator as explained below –

Example 4.4 :

```
#include <iostream.h>
class Box
{
  public:
    double getVolume(void)
   {
        return length * breadth * height;
    }
    void setLength( double len )
   {
        length = len;
    }
    void setBreadth( double bre )
   {
        breadth = bre;
    }
    void setHeight( double hei )
   {
        height = hei;
    }

      Box operator+(const Box& b)
  {
      Box box;
      box.length = this->length + b.length;
      box.breadth = this->breadth + b.breadth;
      box.height = this->height + b.height;
      return box;
    }
```

```
private:
    double length;      // Length of a box
    double breadth;     // Breadth of a box
    double height;      // Height of a box
};

// Main function for the program
int main()
{
  Box Box1;              // Declare Box1 of type Box
  Box Box2;              // Declare Box2 of type Box
  Box Box3;              // Declare Box3 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;

  // Add two object as follows:
```

```
  Box3 = Box1 + Box2;

  // volume of box 3
  volume = Box3.getVolume();
  cout << "Volume of Box3 : " << volume <<endl;
  return 0;
}
```

Output:

```
Volume of Box1 : 210
Volume of Box2 : 1560
Volume of Box3 : 5400
```

CHAPTER 5

CLASSES AND OBJECTS

C++ Class Definitions

When you define a class, you define a blueprint for a data type. This doesn't actually define any data, but it does define what the class name means, that is, what an object of the class will consist of and what operations can be performed on such an object.

A class definition starts with the keyword class followed by the class name; and the class body, enclosed by a pair of curly braces. A class definition must be followed either by a semicolon or a list of declarations. For example, we defined the Box data type using the keyword class as follows –

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

The keyword public determines the access attributes of the members of the class that follows it. A public member can be accessed from outside the class anywhere within the scope of the class object. You can also specify the members of a class as private or protected which we will discuss in a sub-section.

Define C++ Objects

A class provides the blueprints for objects, so basically an object is created from a class. We declare objects of a class with exactly the same sort of declaration that we declare variables of basic types. Following statements declare two objects of class Box –

```
Box Box1;        // Declare Box1 of type Box
Box Box2;        // Declare Box2 of type Box
```

Both of the objects Box1 and Box2 will have their own copy of data members.

Accessing the Data Members

The public data members of objects of a class can be accessed using the direct member access operator (.). Let us try the following example to make the things clear –

Program 5.1 :

```
#include <iostream>

class Box

{

  public:

    double length;   // Length of a box

    double breadth;  // Breadth of a box

    double height;   // Height of a box

};

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.height = 5.0;

  Box1.length = 6.0;

  Box1.breadth = 7.0;


  // box 2 specification

  Box2.height = 10.0;
```

```
   Box2.length = 12.0;

   Box2.breadth = 13.0;



   // volume of box 1

   volume = Box1.height * Box1.length * Box1.breadth;

   cout << "Volume of Box1 : " << volume <<endl;



   // volume of box 2

   volume = Box2.height * Box2.length * Box2.breadth;

   cout << "Volume of Box2 : " << volume <<endl;

   return 0;

}
```

Output :

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

It is important to note that private and protected members can not be accessed directly using direct member access operator (.). We will learn how private and protected members can be accessed.

Class Access Modifiers

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

};
```

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 –

Program 5.2 :

```
#include <iostream>

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;
}
```

Output:

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

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.

Program 5.3 :

```
#include <iostream>

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;
}
```

Output:

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

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.

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

Program 5.4 :

```
#include <iostream>
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;
}
```

Output:

```
Width of box : 5
```

Difference between Class and Structure

	Class	Structure
Definition	A class in C++ can be defined as a collection of related variables and functions encapsulated in a single structure.	A structure can be referred to as a user defined data type possessing its own operations.
Keyword for the declaration	Class	Struct
Default access specifier	Private	Public

Example	class myclass
	{
	private:
	int data;
	public:
	myclass(int data_):
	data(data_)
	{}
	virtual void foo()=0;
	virtual ~class()
	{}
	};
		struct myclass
		{
		private:
		int data;
		public:
		myclass(int data_):
		data(data_)
		{}
		virtual void foo()=0;
		virtual ~class()
		{}
		};

Purpose	Data abstraction and further inheritance	Generally, grouping of data
Type	Reference	Value

Usage	Generally used for large amounts of data.	Generally used for smaller amounts of data.

Object as an argument

As we know that, we can pass any type of arguments within the member function and there are any numbers of arguments.

In C++ programming language, we can also pass an object as an argument within the member function of class.

This is useful, when we want to initialize all data members of an object with another object, we can pass objects and assign the values of supplied object to the current object. For complex or large projects, we need to use objects as an argument or parameter.

Program 5.5 :

Consider the given program:

```
#include <iostream.h>

class Demo
{
        private:
                int a;


        public:
                void set(int x)
                {
                        a = x;
                }

                void sum(Demo ob1, Demo ob2)
                {
                        a  = ob1.a + ob2.a;
                }
```

```
        void print()
        {
                cout<<"Value of A :  "<<a<<endl;
        }
};

int main()
{
        //object declarations
        Demo d1;
        Demo d2;
        Demo d3;

        //assigning values to the data member of objects
        d1.set(10);
        d2.set(20);

        //passing object d1 and d2
        d3.sum(d1,d2);

        //printing the values
        d1.print();
        d2.print();
        d3.print();

        return 0;
}
```

Output

```
Value of A :  10
Value of A :  20
Value of A :  30
```

Static Members

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 –

Program 5.6 :

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

Output:

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

Static Member Function

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.

Program 5.7 :

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

  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



  cout << "Final Stage Count: " << Box::getCount() << endl;

  return 0;
}
```

Output:

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

friend Function and friend Classes

One of the important concepts of OOP is data hiding, i.e., a nonmember function cannot access an object's private or protected data.

But, sometimes this restriction may force programmer to write long and complex codes. So, there is mechanism built in C++ programming to access private or protected data from non-member functions.

This is done using a friend function or/and a friend class.

Friend Function

If a function is defined as a friend function then, the private and protected data of a class can be accessed using the function.

The complier knows a given function is a friend function by the use of the keyword **friend**.

For accessing the data, the declaration of a friend function should be made inside the body of the class (can be anywhere inside class either in private or public section) starting with keyword friend.

Declaration of friend function

```
class class_name

{

   ... .. ...

   friend return_type function_name(argument/s);

   ... .. ...

}
```

Now, you can define the friend function as a normal function to access the data of the class. No friend keyword is used in the definition.

```
class className
```

```
{
   ... .. ...
   friend return_type functionName(argument/s);
   ... .. ...
}

return_type functionName(argument/s)
{
   ... .. ...
   // Private and protected data of className can be accessed from
   // this function because it is a friend function of className.
   ... .. ...
}
```

Program 5.8 :

```
#include <iostream.h>
class Distance
{
   private:
      int meter;
   public:
      Distance(): meter(0) { }
```

```
        //friend function
        friend int addFive(Distance);
};

// friend function definition
int addFive(Distance d)
{
   //accessing private data from non-member function
   d.meter += 5;
   return d.meter;
}

int main()
{
   Distance D;
   cout<<"Distance: "<< addFive(D);
   return 0;
}
```

Output

```
Distance: 5
```

Here, friend function addFive() is declared inside Distance class. So, the private data meter can be accessed from this function.

CHAPTER 6

CONSTRUCTORS AND DESTRUCTORS

Constructors

What is constructor?

A constructor is a member function of a class which initializes objects of a class. In C++, Constructor is automatically called when object(instance of class) create. It is special member function of the class.

The process of creating and deleting objects in C++ is a vital task. Each time an instance of a class is created the constructor method is called. Constructors is a special member function of class and it is used to initialize the objects of its class. It is treated as a special member function because its name is the same as the class name. These constructors get invoked whenever an object of its associated class is created. It is named as "constructor" because it constructs the value of data member of a class. Initial values can be passed as arguments to the constructor function when the object is declared.

How constructors are different from a normal member function?

- A constructor is different from normal functions in following ways:
- Constructor has same name as the class itself
- Constructors don’t have return type
- A constructor is automatically called when an object is created.
- If we do not specify a constructor, C++ compiler generates a default constructor for us (expects no parameters and has an empty body).

This can be done in two ways:

- By calling constructor explicitly
- By calling constructor implicitly

Special characteristics of Constructors:

- They should be declared in the public section
- They do not have any return type, not even void
- They get automatically invoked when the objects are created
- They cannot be inherited though derived class can call the base class constructor
- Like other functions, they can have default arguments
- You cannot refer to their address
- Constructors cannot be virtual

Types of Constructors

These are three types of constructors

1. Default constructor
2. Parameterized constructor
3. Copy constructor

Default constructor

The default constructor is the constructor which doesn't take any argument. It has no parameter but a programmer can write some initialization statement there.

Syntax:

```
class_name()
{
 // Constructor Definition ;
}
```

Program 6.1 :

```
#include <iostream.h>
class Calc
{
        int val;
public:
 Calc()
 {
        val = 20;
 }
};
int main()
{
 Calc c1;
 cout <<"val="<< c1.val;
}
```

Output:

```
Val=20
```

A default constructor is very important for initializing object members, that even if we do not define a constructor explicitly, the compiler automatically provides a default constructor implicitly.

Parameterized Constructor

A default constructor does not have any parameter, but programmers can add and use parameters within a constructor if required. This helps programmers to assign initial values to an object at the time of creation.

Program 6.2 :

```
#include <iostream.h>
class Calc
 {
          int val2;
  public:
  Calc(int x)
  {
   val2=x;
 } };
 int main()
 {
  Calc c1(10);
  Calc c2(20);
  Calc c3(30);
  cout <<"val="<< c1.val2;
  cout <<"val="<< c2.val2;
  cout <<"val="<< c3.val2;
}
```

Output:

```
val=10
val=20
val=30
```

Copy Constructor

C++ provides a special type of constructor which takes an object as an argument and is used to copy values of data members of one object into another object. In this case, copy constructors are used to declaring and initializing an object from another object.

```
Calc C2(C1);

Or

Calc C2 = C1;
```

The process of initializing through a copy constructor is called the copy initialization.

Syntax:

```
class-name (class-name &)

{

 . . .

}
```

Program 6.3 :

```
#include <iostream.h>

class CopyCon

{

          int a, b;

public:

 CopyCon(int x, int y)

 {

 a = x;

 b = y;

 cout << "\nHere is the initialization of Constructor";
```

```
 }
 void Display()
 {
 cout << "\nValues : \t" << a << "\t" << b;
 }
};
void main()
{
 CopyCon Object(30, 40);
 //Copy Constructor
 CopyCon Object2 = Object;
 Object.Display();
 Object2.Display();
}
```

Output:

```
Values : 30   40
Values : 30   40
```

Constructor Overloading

In C++, We can have more than one constructor in a class with same name, as long as each has a different list of arguments.This concept is known as Constructor Overloading and is quite similar to function overloading.

- Overloaded constructors essentially have the same name (name of the class) and different number of arguments.
- A constructor is called depending upon the number and type of arguments passed.

- While creating the object, arguments must be passed to let compiler know, which constructor needs to be called.

Program 6.4 :

```
#include <iostream.h>
class construct
{
 public:
   float area;
      // Constructor with no parameters
   construct()
   {
      area = 0;
   }
     // Constructor with two parameters
   construct(int a, int b)
   {
      area = a * b;
   }
     void disp()
   {
      cout<< area<< endl;
   }
};
 int main()
```

```
{
   // Constructor Overloading
   // with two different constructors
   // of class name
   construct o;
   construct o2( 10, 20);

   o.disp();
   o2.disp();
   return 1;
}
```

Output:

```
0
200
```

What are Destructors?

As the name implies, destructors are used to destroy the objects that have been created by the constructor within the C++ program. Destructor names are same as the class name but they are preceded by a tilde (~). It is a good practice to declare the destructor after the end of using constructor. Here's the basic declaration procedure of a destructor:

~Cube()

```
{        }
```

The destructor neither takes an argument nor returns any value and the compiler implicitly invokes upon the exit from the program for cleaning up storage that is no longer accessible.

Destructor Rules

1) Name should begin with tilde sign(~) and must match class name.

2) There cannot be more than one destructor in a class.

3) Unlike constructors that can have parameters, destructors do not allow any parameter.

4) They do not have any return type, just like constructors.

5) When you do not specify any destructor in a class, compiler generates a default destructor and inserts it into your code.

When does the destructor get called?

A destructor is **automatically called** when:

1) The program finished execution.

2) When a scope (the { } parenthesis) containing local variable ends.

3) When you call the delete operator.

Program 6.5 :

```
#include <iostream.h>

class ABC

{

public:

       ABC () //constructor defined

      {

            cout << "Hey look I am in constructor" << endl;

      }

      ~ABC() //destructor defined

      {

           cout << "Hey look I am in destructor" << endl;
```

```
    }
};
int main()
{
    ABC cc1; //constructor is called
    cout << "function main is terminating...." << endl;
    /*....object cc1 goes out of scope ,now destructor is being called...*/
    return 0;
}  //end of program
```

Output

```
Hey look I am in constructor
function main is terminating....
Hey look I am in destructor
```

Explanation

In the above program, when constructor is called *"Hey look I am in constructor"* is printed then following it *"Function main is terminating….."* is printed but after that the object cc1that was created before goes out of scope and to de-allocate the memory consumed by cc1destructor is called and *"Hey I am in destructor"* is printed.

Program 6.6 :

```
#include <iostream.h>
class HelloWorld{
public:
```

```
 //Constructor
  HelloWorld(){
    cout<<"Constructor is called"<<endl;
  }
 //Destructor
  ~HelloWorld(){
    cout<<"Destructor is called"<<endl;
   }
  //Member function
   void display(){
     cout<<"Hello World!"<<endl;
   }
};
int main()
{
   //Object created
   HelloWorld obj;
   //Member function called
   obj.display();
   return 0;
}
```

Output:

```
Constructor is called
Hello World!
Destructor is called
```

CHAPTER 7

INHERITANCE

C++ Inheritance

In C++, inheritance is a process in which one object acquires all the properties and behaviors of its parent object automatically. In such way, you can reuse, extend or modify the attributes and behaviors which are defined in other class.

In C++, the class which inherits the members of another class is called derived class and the class whose members are inherited is called base class. The derived class is the specialized class for the base class.

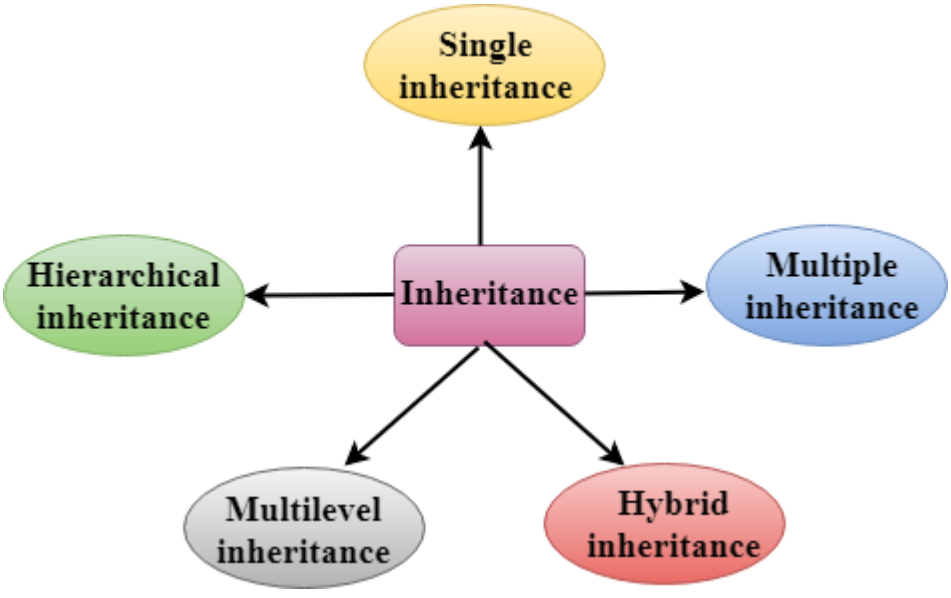
Advantage of C++ Inheritance

Code reusability: Now you can reuse the members of your parent class. So, there is no need to define the member again. So less code is required in the class.

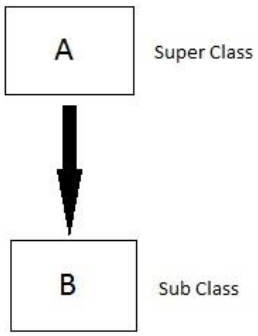
Types Of Inheritance

C++ supports five types of inheritance:

- Single inheritance
- Multiple inheritance
- Hierarchical inheritance
- Multilevel inheritance
- Hybrid inheritance



Single Inheritance



If a single class is derived from one base class then it is called **single inheritance**. In single inheritance base and derived class exhibit one to one relation.

Program 7.1 :

```
#include <iostream.h>

class base    //single base class

{

  public:

    int x;

  void getdata()

  {

    cout << "Enter the value of x = "; cin >> x;

  }

};

class derive : public base    //single derived class

{

  private:

   int y;

  public:

  void readdata()

 {
```

```
    cout << "Enter the value of y = "; cin >> y;
  }
  void product()
  {
    cout << "Product = " << x * y;
  }
};
int main()
{
   derive a;     //object of derived class
   a.getdata();
   a.readdata();
   a.product();
   return 0;
}        //end of program
```

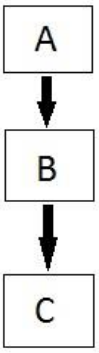
Output

```
Enter the value of x = 3
Enter the value of y = 4
Product = 12
```

Multilevel Inheritance

If a class is derived from another derived class then it is called **multilevel inheritance**. So in C++ multilevel inheritance, a class has more than one parent class.

For example, if we take animals as a base class then mammals are the derived class which has features of animals and then humans are the also derived class that is derived from sub-class mammals which inherit all the features of mammals.



Program 7.2 :

```
#include <iostream.h>

class base //single base class

{

        public:

        int x;

        void getdata()

        {

        cout << "Enter value of x= "; cin >> x;

        }

};

class derive1 : public base // derived class from base class

{

        public:

        int y;

        void readdata()
```

```
        {
           cout << "\nEnter value of y= "; cin >> y;
        }
};
class derive2 : public derive1   // derived from class derive1
{
        private:
        int z;
        public:
        void indata()
        {
        cout << "\nEnter value of z= "; cin >> z;
        }
        void product()
        {
           cout << "\nProduct= " << x * y * z;
        }
};
int main()
{
    derive2 a;      //object of derived class
    a.getdata();
```

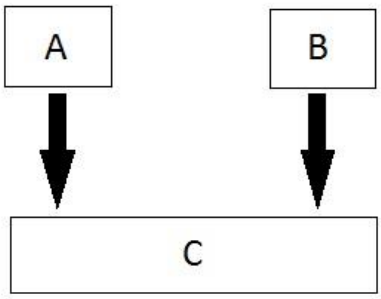
```
    a.readdata();

    a.indata();

    a.product();

    return 0;

}                   //end of program
```

Output

```
Enter value of x= 2



Enter value of y= 3



Enter value of z= 3



Product= 18
```

Multiple Inheritance

If a class is derived from two or more base classes then it is called multiple inheritance. In **multiple inheritance** a derived class has more than one base class.



How does multiple inheritance differ from multilevel inheritance?

Though but multiple and multilevel sounds like same but they differ hugely in meaning. Inmultilevel inheritance, we have multiple parent classes whereas in in multiple inheritance we have multiple base classes.

To put it in simple words, in multilevel inheritance, a class is derived from a class which is also derived from another base class. And these levels of inheritance can be extended. On the contrary, in multiple inheritance, a class is derived from two different base classes.

For example

- **Multilevel inheritance** : Inheritance of characters by a child from father and father inheriting characters from his father (grandfather)
- **Multiple inheritance** : Inheritance of characters by a child from mother and father

Program 7.3 :

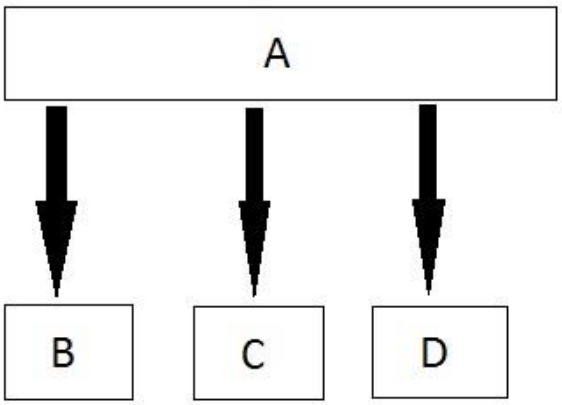
```
#include <iostream.h>

class A

{

        public:

        int x;

        void getx()

   {

          cout << "enter value of x: "; cin >> x;

   }

};

class B

{

        public:

        int y;

        void gety()

        {
```

```
            cout << "enter value of y: "; cin >> y;
        }
};
class C : public A, public B   //C is derived from class A and class B
{
        public:
        void sum()
        {
            cout << "Sum = " << x + y;
        }
};

int main()
{
         C obj1; //object of derived class C
         obj1.getx();
         obj1.gety();
         obj1.sum();
         return 0;
}        //end of program
```

Output

```
enter value of x: 5

enter value of y: 4

Sum = 9
```

Hierarchical Inheritance



When several classes are derived from common base class it is called **hierarchical inheritance**.

In **C++ hierarchical inheritance**, the feature of the base class is inherited onto more than one sub-class.

For example, a car is a common class from which Audi, Ferrari, Maruti etc can be derived.

Following block diagram highlights its concept.

Program 7.4 :

```
#include <iostream.h>

class A //single base class

{

   public:

         int x, y;

         void getdata()

         {

            cout << "\nEnter value of x and y:\n"; cin >> x >> y;

         }
```

```
};
class B : public A //B is derived from class base
{
   public:
        void product()
        {
           cout << "\nProduct= " << x * y;
        }
};
class C : public A //C is also derived from class base
{
   public:
        void sum()
        {
      cout << "\nSum= " << x + y;
        }
};
int main()
{
   B obj1;         //object of derived class B
   C obj2;         //object of derived class C
   obj1.getdata();
```

```
    obj1.product();

    obj2.getdata();

    obj2.sum();

    return 0;

}  //end of program
```

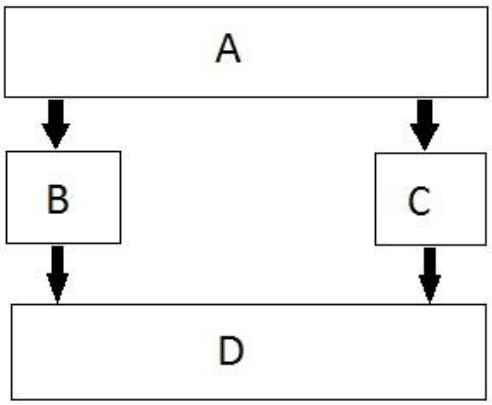
Output

```
Enter value of x and y:

2

3

Product= 6

Enter value of x and y:

2

3

Sum= 5
```

Hybrid Inheritance

The inheritance in which the derivation of a class involves more than one form of any inheritance is called **hybrid inheritance**. Basically **C++ hybrid inheritance** is combination of two or more types of inheritance. It can also be called multi path inheritance.

Following block diagram highlights the concept of hybrid inheritance which involves singleand multiple inheritance.



Program 7.5 :

```
#include <iostream.h>

class A

{

        public:

        int x;

};

class B : public A

{

        public:

        B()      //constructor to initialize x in base class A

        {

          x = 10;

        }

};

class C

 {

        public:

        int y;

        C()   //constructor to initialize y

        {
```

```
          y = 4;

     }

};

class D : public B, public C   //D is derived from class B and class C

{

        public:

        void sum()

        {

           cout << "Sum= " << x + y;

        }

};



int main()

{

       D obj1;         //object of derived class D

        obj1.sum();

        return 0;

}               //end of program
```

Output

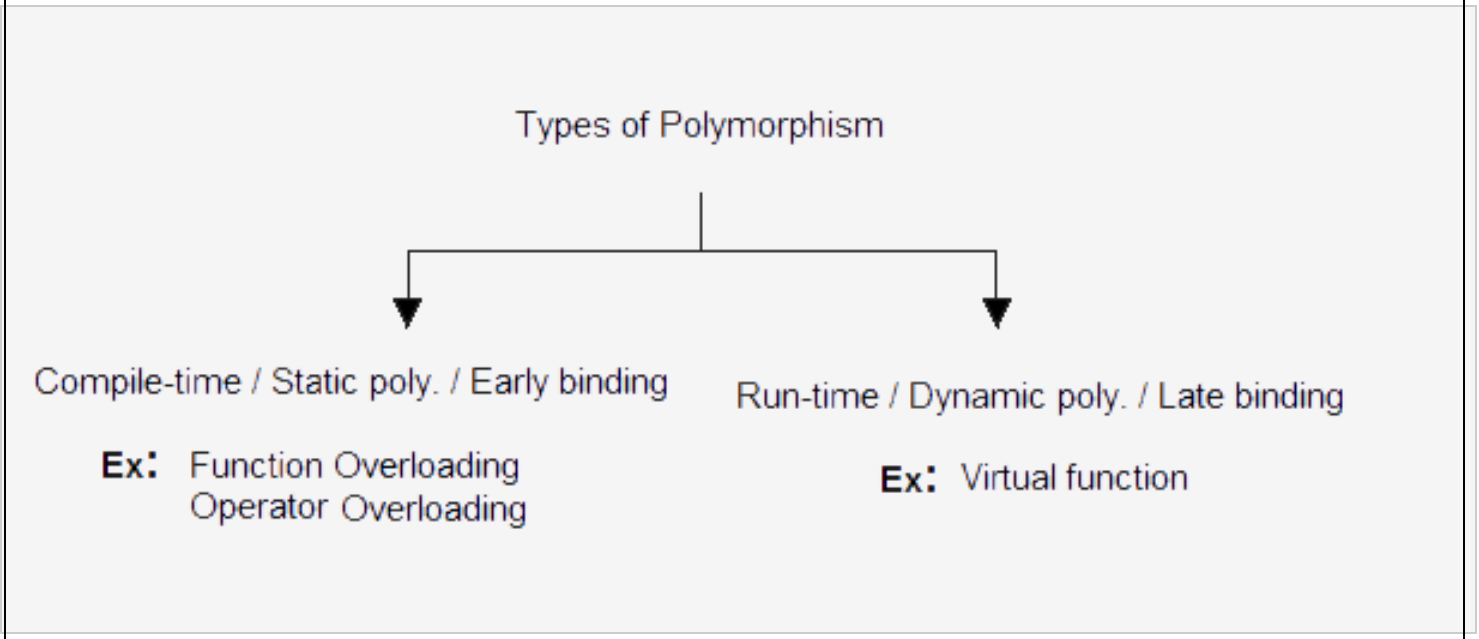
```
Sum= 14
```

CHAPTER 8

POLYMORPHISM VIRTUAL FUNCTIONS AND POINTERS

Polymorphism means more than one function with same name, with different working. Polymorphism can be static or dynamic. In static polymorphism memory will be allocated at compile-time. In dynamic polymorphism memory will be allocated at run-time. Both function overloading and operator overloading are an examples of static polymorphism. Virtual function is an example of dynamic polymorphism.

- Static polymorphism is also known as early binding and compile-time polymorphism.
- Dynamic polymorphism is also known as late binding and run-time polymorphism.



- **Function Overloading :** Function overloading is an example of static polymorphism. More than one function with same name, with different signature in a class or in a same scope is called function overloading.
- **Operator Overloading :** Another example of static polymorphism is Operator overloading. Operator overloading is a way of providing new implementation of existing operators to work with user-defined data types.
- **Virtual function :** Virtual function is an example of dynamic polymorphism. Virtual function is used in situation, when we need to invoke derived class function using base class pointer. Giving new implementation of derived class method into base class and the calling of this new implemented function with base class's object is done by making base class function as virtual function. This is how we can achieve "Runtime Polymorphism".

Virtual function

- A C++ virtual function is a member function in the base class that you redefine in a derived class. It is declared using the virtual keyword.
- It is used to tell the compiler to perform dynamic linkage or late binding on the function.
- There is a necessity to use the single pointer to refer to all the objects of the different classes. So, we create the pointer to the base class that refers to all the derived objects. But, when base class pointer contains the address of the derived class object, always executes the base class function. This issue can only be resolved by using the 'virtual' function.
- A 'virtual' is a keyword preceding the normal declaration of a function.
- When the function is made virtual, C++ determines which function is to be invoked at the runtime based on the type of the object pointed by the base class pointer.

Late binding or Dynamic linkage

In late binding function call is resolved during runtime. Therefore compiler determines the type of object at runtime, and then binds the function call.

Rules of Virtual Function

- Virtual functions must be members of some class.
- Virtual functions cannot be static members.
- They are accessed through object pointers.
- They can be a friend of another class.
- A virtual function must be defined in the base class, even though it is not used.
- The prototypes of a virtual function of the base class and all the derived classes must be identical. If the two functions with the same name but different prototypes, C++ will consider them as the overloaded functions.
- We cannot have a virtual constructor, but we can have a virtual destructor
- Consider the situation when we don't use the virtual keyword.

Program 8.1 :

```
#include <iostream.h>
{
 public:
virtual void display()
{
        cout << "Base class is invoked"<<endl;
}
};

class B:public A
{
public:
 void display()
{
         cout << "Derived Class is invoked"<<endl;
}
};

int main()
{
        A* a;    //pointer of base class
         B b;     //object of derived class
         a = &b;
        a->display();   //Late Binding occurs
}
```

Output:

```
Derived Class is invoked
```

Dynamic Binding - OOPs

In OOPs Dynamic Binding refers to linking a procedure call to the code that will be executed only at run time. The code associated with the procedure in not known until the program is executed, which is also known as late binding.

Program 8.2 :

```
#include <iostream.h>
int Square(int x)
{
  return x*x;
}
int Cube(int x)
{
  return x*x*x;
}

int main()
{
  int x =10;
  int choice;
 do
{
   cout << "Enter 0 for square value, 1 for cube value: ";
   cin >> choice;
}
while (choice < 0 || choice > 1);
int (*ptr) (int);
```

```
switch (choice)
{
  case 0: ptr = Square;

  break;
  case 1: ptr = Cube;

  break;
}

cout << "The result is: " << ptr(x) << endl;
return 0;

}
```

Output :

```
Enter 0 for square value, 1 for cube value:0
The result is:100
```

Function Overloading vs Function Overriding

Function Overloading (achieved at compile time)

It provides multiple definitions of the function by changing signature i.e changing number of parameters, change datatype of parameters, return type doesn't play anyrole.

- It can be done in base as well as derived class.
  Example:

```
void area(int a);
void area(int a, int b);
```

Program 8.3 :

```
#include <iostream.h>
 // overloaded functions
void test(int);
void test(float);
void test(int, float);
int main()
{
   int a = 5;
   float b = 5.5;

   test(a);
   test(b);
   test(a, b);

   return 0;
}
void test(int var)
{
   cout << "Integer number: " << var << endl;
}
void test(float var)
{
   cout << "Float number: "<< var << endl;
}
void test(int var1, float var2)
```

```
{
   cout << "Integer number: " << var1;
   cout << " and float number:" << var2;
}
```

Output:

```
Integer number: 5
Float number: 5.5
Integer number: 5 and float number: 5.5
```

Function Overriding (achieved at run time)

It is the redefinition of base class function in its derived class with same signature i.e return type and parameters.

- It can only be done in derived class.

Example:

```
Class a
{
public:
     virtual void display(){ cout << "hello"; }
}

Class b:public a
{
public:
      void display(){ cout << "bye";};
}
```

Program 8.4 :

```
#include<iostream.h>
 class BaseClass
{
public:
   virtual void Display()
   {
      cout << "\nThis is Display() method"
            " of BaseClass";
   }
   void Show()
   {
      cout << "\nThis is Show() method "
           "of BaseClass";
   }
};

class DerivedClass : public BaseClass
{
public:
   void Display()
   {
      cout << "\nThis is Display() method" " of DerivedClass";
   }
};
```

```
// Driver code
int main()
{
   DerivedClass dr;
   BaseClass &bs = dr;
   bs.Display();
   dr.Show();
}
```

Output:

```
This is Display() method of DerivedClass
This is Show() method of BaseClass
```

Function Overloading VS Function Overriding

1. **Inheritance:** Overriding of functions occurs when one class is inherited from another class. Overloading can occur without inheritance.
2. **Function Signature:** Overloaded functions must differ in function signature ie either number of parameters or type of parameters should differ. In overriding, function signatures must be same.
3. **Scope of functions:** Overridden functions are in different scopes; whereas overloaded functions are in same scope.
4. **Behavior of functions:** Overriding is needed when derived class function has to do some added or different job than the base class function. Overloading is used to have same name functions which behave differently depending upon parameters passed to them.

Pure Virtual Function

A virtual function will become pure virtual function when you append "=0" at the end of declaration of virtual function. Pure virtual function doesn't have body or implementation. We must implement all pure virtual functions in derived class.

Pure virtual function is also known as abstract function.

A class with at least one pure virtual function or abstract function is called abstract class. We can't create an object of abstract class. Member functions of abstract class will be invoked by derived class object.

Program 8.5 :

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

class BaseClass       //Abstract class
{

      public:
       virtual void Display1()=0;     //Pure virtual function or abstract function
       virtual void Display2()=0;     //Pure virtual function or abstract function

       void Display3()
       {
            cout<<"\n\tThis is Display3() method of Base Class";
       }

};

class DerivedClass : public BaseClass
{

      public:
       void Display1()
       {
            cout<<"\n\tThis is Display1() method of Derived Class";
       }
```

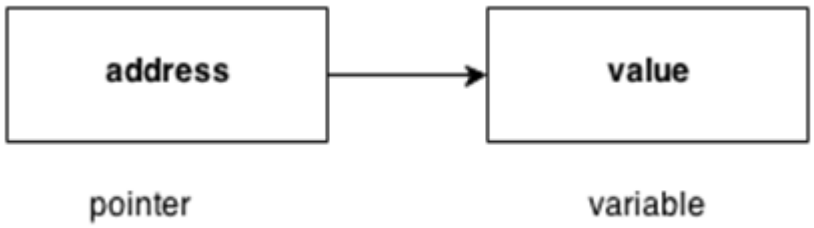
```
        void Display2()
        {
            cout<<"\n\tThis is Display2() method of Derived Class";
        }

};

void main()
{

    DerivedClass D;

    D.Display1();       // This will invoke Display1() method of Derived Class
    D.Display2();       // This will invoke Display2() method of Derived Class
    D.Display3();       // This will invoke Display3() method of Base Class

}
```

Output :

```
    This is Display1() method of Derived Class
    This is Display2() method of Derived Class
    This is Display3() method of Base Class
```

Pointers

The pointer in C++ language is a variable, it is also known as locator or indicator that points to an address of a value.



Advantage of pointer

1) Pointer reduces the code and improves the performance, it is used to retrieving strings, trees etc. and used with arrays, structures and functions.

2) We can return multiple values from function using pointer.

3) It makes you able to access any memory location in the computer's memory.

Usage of pointer

There are many usage of pointers in C++ language.

1) Dynamic memory allocation

In c language, we can dynamically allocate memory using malloc() and calloc() functions where pointer is used.

2) Arrays, Functions and Structures

Pointers in c language are widely used in arrays, functions and structures. It reduces the code and improves the performance.

Symbols used in pointer:

Symbol	Name	Description
& (ampersand sign)	Address operator	Determine the address of a variable.
* (asterisk sign)	Indirection operator	Access the value of an address.

Declaring a pointer

The pointer in C++ language can be declared using ∗ (asterisk symbol).

int ∗ a; //pointer to int

char ∗ c; //pointer to char

Program 8.6 :

```
#include <iostream.h>

int main()

{

int number=30;

int ∗ p;

p=&number;   //stores the address of number variable

cout<<"Address of number variable is:"<<&number<<endl;

cout<<"Address of p variable is:"<<p<<endl;

cout<<"Value of p variable is:"<<*p<<endl;

  return 0;

}
```

Output :

```
Address of number variable is:0x7ffccc8724c4

Address of p variable is:0x7ffccc8724c4

Value of p variable is:30
```

Pointer Program to swap 2 numbers without using 3rd variable

Program 8.7 :

```
#include <iostream.h>
int main()
{
int a=20,b=10,*p1=&a,*p2=&b;
cout<<"Before swap: *p1="<<*p1<<" *p2="<<*p2<<endl;
*p1=*p1+*p2;
*p2=*p1-*p2;
*p1=*p1-*p2;
cout<<"After swap: *p1="<<*p1<<" *p2="<<*p2<<endl;
return 0;
}
```

Output :

```
Before swap: *p1=20 *p2=10
After swap: *p1=10 *p2=20
```

Program 8.8 :

```
#include <iostream.h>
int main()
{
      int var1 = 3;
      int var2 = 24;
      int var3 = 17;
```

```
    cout << &var1 << endl;
    cout << &var2 << endl;
    cout << &var3 << endl;
}
```

Output :

```
0x7fff5fbff8ac

0x7fff5fbff8a8

0x7fff5fbff8a4
```

Program 8.9 :

```
#include <iostream.h>

int main()

{

   int *pc, c;

   c = 5;

   cout << "Address of c (&c): " << &c << endl;

   cout << "Value of c (c): " << c << endl << endl;

   pc = &c;    // Pointer pc holds the memory address of variable c

   cout << "Address that pointer pc holds (pc): "<< pc << endl;

   cout << "Content of the address pointer pc holds (*pc): " << *pc << endl << endl;

   c = 11;    // The content inside memory address &c is changed from 5 to 11.

   cout << "Address pointer pc holds (pc): " << pc << endl;
```

```
   cout << "Content of the address pointer pc holds (*pc): " << *pc << endl << endl;
*pc = 2;

   cout << "Address of c (&c): " << &c << endl;

   cout << "Value of c (c): " << c << endl << endl;

return 0;

 }
```

Output :

```
Address of c (&c): 0x7fff5fbff80c

Value of c (c): 5

Address that pointer pc holds (pc): 0x7fff5fbff80c

Content of the address pointer pc holds (*pc): 5

Address pointer pc holds (pc): 0x7fff5fbff80c

Content of the address pointer pc holds (*pc): 11

Address of c (&c): 0x7fff5fbff80c

Value of c (c): 2
```

CHAPTER 9

INPUT/OUTPUT STREAMS

File Handling using File Streams

File represents storage medium for storing data or information. Streams refer to sequence of bytes. In Files we store data i.e. text or binary data permanently and use these data to read or write in the form of input output operations by transferring bytes of data. So we use the term File Streams/File handling. We use the header file `<fstream.h>`

ofstream: It represents output Stream and this is used for writing in files.

ifstream: It represents input Stream and this is used for reading from files.

fstream: It represents both output Stream and input Stream. So it can read from files and write to files.

What is Stream?

- A stream is an abstraction. It is a sequence of bytes.
- It represents a device on which input and output operations are performed.
- It can be represented as a source or destination of characters of indefinite length.
- It is generally associated to a physical source or destination of characters like a disk file, keyboard or console.
- C++ provides standard iostream library to operate with streams.
- The iostream is an object-oriented library which provides Input/Output functionality using streams.

I/O Stream	Meaning	Description
istream	Input Stream	It reads and interprets input.
ostream	Output stream	It can write sequences of characters and represents other kinds of data.
ifstream	Input File Stream	The ifstream class is derived from fstreambase and istream by multiple inheritance. This class accesses the member functions such as get(), getline(), seekg(), tellg() and read().

		It provides open() function with the default input mode and allows input operations.
ofstream	Output File Stream	The ofstream class is derived from fstreambase and ostream classes. This class accesses the member functions such as put(), seekp(), write() and tellp(). It provides the member function open() with the default output mode.
fstream	File Stream	The fstream allows input and output operations simultaneous on a filebuf. It invokes the member function istream::getline() to read characters from the file. This class provides the open() function with the default input mode.
fstreambase	File Stream Base	It acts as a base class for fstream, ifstream and ofstream. The open() and close() functions are defined in fstreambase.

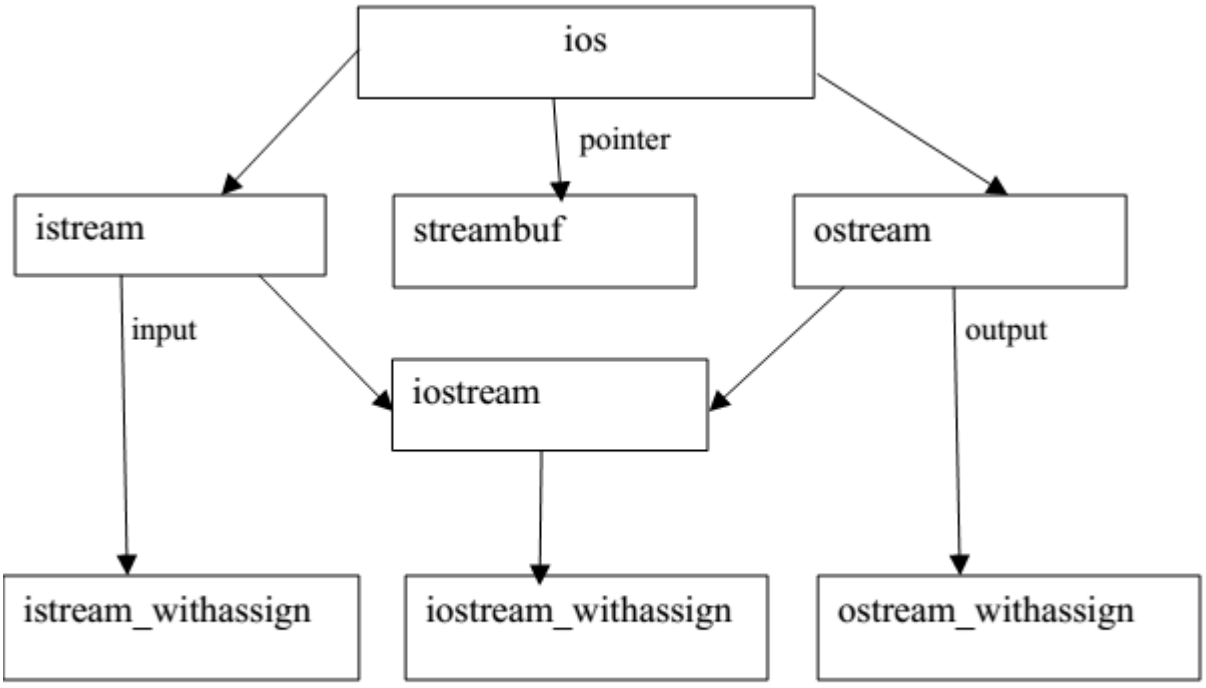
Advantages of Stream Classes

Stream classes have good error handling capabilities.

These classes work as an abstraction for the user that means the internal operation is encapsulated from the user.

These classes are buffered and do not uses the memory disk space.

These classes have various functions that make reading or writing a sequence of bytes easy for the programmer.



Stream classes for console I/O operations

Special operations in a File

There are few important functions to be used with file streams like:

tellp() - It tells the current position of the put pointer.

Syntax: filepointer.tellp()

tellg() - It tells the current position of the get pointer.

Syntax: filepointer.tellg()

seekp() - It moves the put pointer to mentioned location.

Syntax: filepointer.seekp(no of bytes,reference mode)

seekg() - It moves get pointer(input) to a specified location.

Syntax: filepointer.seekg((no of bytes,reference point)

put() - It writes a single character to file.

get() - It reads a single character from file.

Note: For seekp and seekg three reference points are passed:
ios::beg - beginning of the file
ios::cur - current position in the file
ios::end - end of the file

Program 9.1 :

```
#include<iostream.h>

int main()

{

	int count=0;

	char c;

	cout<<”INPUT TEXT\n”;

	cin.get(c);

	while(c!=’\n’)

	{

		cout.put(c);

		count++;

		cin.get(c);

	}

	cout<<”\n number of characters=”<<count<<”\n”;

	return 0;

}
```

Output:

```
C++ programming

Number of characters=14
```

Program 9.2 :

```
#include <iostream.h>
#include <string.h>
int main ()
{
  string mystr;
  cout << "What's your name? ";
  getline (cin, mystr);
  cout << "Hello " << mystr << ".\n";
  cout << "What is your favorite team? ";
  getline (cin, mystr);
  cout << "I like " << mystr << "     too!\n";
  return 0;
}
```

Output :

```
What's your name?
Homer Simpson
Hello Homer Simpson.
What is your favorite team? The Isotopes
I like The Isotopes too!
```

C++ Manipulators - endl, setw, setprecision, setf

Formatting output using manipulators

Formatted output is very important in development field for easily read and understand.

C++ offers the several input/output manipulators for formatting, commonly used manipulators are given below.

Manipulator	Declaration in
endl	iostream.h
setw	iomanip.h
setprecision	iomanip.h
setf	iomanip.h

endl

endl manipulator is used to Terminate a line and flushes the buffer.

Difference b/w '\n' and endl

When writing output in C++,you can use either std::endl or '\n' to produce a newline, but each has a different effect.

- std::endl sends a newline character '\n' and flushes the output buffer.
- '\n' sends the newline character, but does not flush the output buffer.

The distinction is very important if you're writing debugging messages that you really need to see immediately, you should always use std::endl rather than '\n' to force the flush to take place immediately.

The following is an example of how to use both versions, although you cannot see the flushing occuring in this example.

Program 9.3 :

```
#include <iostream.h>
int main()
{
        cout<<"USING '\\n' ...\n";
        cout<<"Line 1 \nLine 2 \nLine 3 \n";
        cout<<"USING end ..."<< endl;
        cout<< "Line 1" << endl << "Line 2" << endl << "Line 3" << endl;
        return 0;
}
```

Output

```
USING '\n' ...
Line 1
Line 2
Line 3
USING end ...
Line 1
Line 2
Line 3
```

setw() and setfill() manipulators

setw manipulator sets the width of the filed assigned for the output.

The field width determines the minimum number of characters to be written in some output representations. If the standard width of the representation is shorter than the field width, the representation is padded with fill characters (using setfill).

setfill character is used in output insertion operations to fill spaces when results have to be padded to the field width.

Syntax

```
setw([number_of_characters]);

setfill([character]);
```

Program 9.4 :

```
#include <iostream.h>
#include <iomanip.h>
int main()
{
        cout<<"USING setw() .............\n";
        cout<< setw(10) <<11<<"\n";
        cout<< setw(10) <<2222<<"\n";
```

```
        cout<< setw(10) <<33333<<"\n";
        cout<< setw(10) <<4<<"\n";

        cout<<"USING setw() & setfill() [type- I]...\n";
        cout<< setfill('0');
        cout<< setw(10) <<11<<"\n";
        cout<< setw(10) <<2222<<"\n";
        cout<< setw(10) <<33333<<"\n";
        cout<< setw(10) <<4<<"\n";

        cout<<"USING setw() & setfill() [type-II]...\n";
        cout<< setfill('-')<< setw(10) <<11<<"\n";
        cout<< setfill('*')<< setw(10) <<2222<<"\n";
        cout<< setfill('@')<< setw(10) <<33333<<"\n";
        cout<< setfill('#')<< setw(10) <<4<<"\n";
        return 0;
}
```

Output

```
USING setw() ..............
        11
      2222
     33333
         4
USING setw() & setfill() [type- I]...
0000000011
0000002222
0000033333
0000000004
USING setw() & setfill() [type-II]...
--------11
******2222
```

```
@@@@@33333
#########4
```

setf() and setprecision() manipulator

setprecision manipulator sets the total number of digits to be displayed, when floating point numbers are printed.

Syntax

```
setprecision([number_of_digits]);

cout<<setprecision(5)<<1234.537;

// output will be : 1234.5
```

On the default floating-point notation, the precision field specifies the maximum number of meaningful digits to display in total counting both those before and those after the decimal point. Notice that it is not a minimum and therefore it does not pad the displayed number with trailing zeros if the number can be displayed with less digits than the precision.

In both the fixed and scientific notations, the precision field specifies exactly how many digits to display after the decimal point, even if this includes trailing decimal zeros. The number of digits before the decimal point does not matter in this case.

Syntax

```
setf([flag_value],[field bitmask]);
```

field bitmask	flag values
adjustfield	left, right or internal
basefield	dec, oct or hex
floatfield	scientific or fixed

Program 9.5 :

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

```
#include <iomanip.h>
int main()
{
        cout<<"USING fixed .....................\n";
        cout.setf(ios::floatfield,ios::fixed);
        cout<< setprecision(5)<<1234.537<< endl;

        cout<<"USING scientific .................\n";
        cout.setf(ios::floatfield,ios::scientific);
        cout<< setprecision(5)<<1234.537<< endl;
        return 0;
}
```

Output

```
USING fixed .....................
1234.53700
USING scientific .................
1234.5
```

Program 9.6 :

```
#include <iostream.h>
#include <iomanip.h>
int main()
{
        int num=10;

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

        cout.setf(ios::basefield,ios::oct);
        cout<<"Octal value is :"<< num << endl;
```

```
        cout.setf(ios::basefield,ios::hex);
        cout<<"Hex value is :"<< num << endl;
        return 0;
}
```

Adjusting the width of output.

Program 9.7 :

```
 #include <iostream.h>
int main()
{
   cout << "Start >";
   cout.width(25);
   cout << 123 << "< End\n";

  cout << "Start >";
  cout.width(25);
  cout << 123<< "< Next >";
  cout << 456 << "< End\n";

  cout << "Start >";
  cout.width(4);
  cout << 123456 << "< End\n";

return 0;
}
```

Output:

```
Start >             123< End
Start >             123< Next >456< End
Start >123456< End
```

CHAPTER 10

FILES

Many real-life scenarios are there that handle a large number of data, and in such situations, you need to use some secondary storage to store the data. The data are stored in the secondary device using the concept of files. Files are the collection of related data stored in a particular area on the disk. Programs can be written to perform read and write operations on these files.

Working with files generally requires the following kinds of data communication methodologies:

Data transfer between console units

Data transfer between the program and the disk file

So far we have learned about iostream standard library which provides cin and cout methods for reading from standard input and writing to standard output respectively. In this chapter, you will get to know how files are handled using C++ program and what are the functions and syntax used to handle files in C++.

Here are the lists of standard file handling classes

- **Ofstream:** This file handling class in C++ signifies the output file stream and is applied to create files for writing information to files
- **Ifstream:** This file handling class in C++ signifies the input file stream and is applied for reading information from files
- **Fstream:** This file handling class in C++ signifies the file stream generally, and has the capabilities for representing both ofstream and ifstream

All the above three classes are derived from fstreambase and from the corresponding iostream class and they are designed specifically to manage disk files.

If programmers want to use a disk file for storing data, they need to decide about the following things about the file and its intended use. These points that are to be noted are:

- A name for the file
- Data type and structure of the file
- Purpose (reading, writing data)
- Opening method
- Closing the file (after use)

Files can be opened in two ways. They are:

- Using constructor function of the class
- Using member function open of the class

Opening a File

The first operation generally performed on an object of one of these classes to use a file is the procedure known as to opening a file. An open file is represented within a program by a stream and any input or output task performed on this stream will be applied to the physical file associated with it. The syntax of opening a file in C++ is:

```
open (filename, mode);
```

There are some mode flags used for file opening. These are:

- ios::app: append mode
- ios::ate: open a file in this mode for output and read/write controlling to the end of the file
- ios::in: open file in this mode for reading
- ios::out: open file in this mode for writing
- ios::trunk: when any file already exists, its contents will be truncated before file opening

Closing a file

When any C++ program terminates, it automatically flushes out all the streams releases all the allocated memory and closes all the opened files. But it is good to use the close() function to close the file related streams and it is a member of ifsream, ofstream and fstream objects.

The structure of using this function is:

```
void close();
```

General functions used for File handling

- open(): To create a file
- close(): To close an existing file
- get(): to read a single character from the file
- put(): to write a single character in the file
- read(): to read data from a file
- write(): to write data into a file

Reading from and writing to a File

While doing C++ program, programmers write information to a file from the program using the stream insertion operator (<<) and reads information using the stream extraction operator (>>). The only difference is that for files programmers need to use an ofstream or fstream object instead of the cout object and ifstream or fstream object instead of the cin object.

Operations in File Handling:

- Creating a file: open()
- Reading data: read()
- Writing new data: write()
- Closing a file: close()

Creating/Opening a File

We create/open a file by specifying new path of the file and mode of operation. Operations can be reading, writing, appending and truncating. Syntax for file creation: FilePointer.open("Path",ios::mode);

Example of file opened for writing: st.open("E:\studytonight.txt",ios::out);

Example of file opened for reading: st.open("E:\studytonight.txt",ios::in);

Example of file opened for appending: st.open("E:\studytonight.txt",ios::app);

Example of file opened for truncating: st.open("E:\studytonight.txt",ios::trunc);

Program 10.1 :

```
#include<iostream.h>
#include<conio.h>
#include <fstream.h>
int main()
{
   fstream st; // Step 1: Creating object of fstream class
   st.open("study.txt",ios::out);  // Step 2: Creating new file
   if(!st) // Step 3: Checking whether file exist
   {
      cout<<"File creation failed";
   }
   else
   {
      cout<<"New file created";
      st.close(); // Step 4: Closing file
   }
   getch();
   return 0;
}
```

Writing to a File

Program 10.2 :

```
#include <iostream.h>
#include<conio.h>
#include <fstream.h>
int main()
{
   fstream st; // Step 1: Creating object of fstream class
   st.open("study.txt",ios::out);  // Step 2: Creating new file
   if(!st) // Step 3: Checking whether file exist
   {
      cout<<"File creation failed";
   }
   else
   {
      cout<<"New file created";
      st<<"Hello";    // Step 4: Writing to file
      st.close(); // Step 5: Closing file
   }
   getch();
   return 0;
}
```

Here we are sending output to a file. So, we use ios::out. As given in the program, information typed inside the quotes after "FilePointer <<" will be passed to output file.

Reading from a File

Program 10.3 :

```
#include <iostream.h>
#include<conio.h>
#include <fstream.h>
int main()
{
   fstream st; // step 1: Creating object of fstream class
   st.open("study.txt",ios::in);   // Step 2: Creating new file
   if(!st) // Step 3: Checking whether file exist
   {
      cout<<"No such file";
   }
   else
   {
      char ch;
      while (!st.eof())
      {
         st >>ch;  // Step 4: Reading from file
         cout << ch;   // Message Read from file
      }
      st.close(); // Step 5: Closing file
   }
   getch();
```

```
   return 0;

}
```

Here we are reading input from a file. So, we use ios::in. As given in the program, information from the output file is obtained with the help of following syntax "FilePointer >>variable".

Close a File

Program 10.4 :

It is done by FilePointer.close().

```
#include <iostream.h>

#include<conio.h>

#include <fstream.h>

int main()

{

   fstream st; // Step 1: Creating object of fstream class

   st.open(“study.txt",ios::out);  // Step 2: Creating new file

   st.close(); // Step 4: Closing file

   getch();

   return 0;

}
```