Difference between OOP and POP

Oop	Pop
OOP takes a bottom-up approach in designing a program.	POP follows a top-down approach.
Program is divided into objects depending on the problem.	Program is divided into small chunks based on the functions.
Each object controls its own data.	Each function contains different data.
Focuses on security of the data irrespective of the algorithm.	Follows a systematic approach to solve the problem.
The main priority is data rather than functions in a program.	Functions are more important than data in a program.
The functions of the objects are linked via message passing.	Different parts of a program are interconnected via parameter passing.
Data hiding is possible in OOP.	No easy way for data hiding.
Inheritance is allowed in OOP.	No such concept of inheritance in POP.
Operator overloading is allowed.	Operator overloading is not allowed.
C++, Java.	Pascal, Fortran.

-
- Introduction to C++
- C++ was developed by **Bjarne Stroustrup in 1979 at Bell Labs.**
- It is a general purpose programming language which supports procedural, object-oriented and generic programming.
- The main purpose of C++ was to make writing good programs easier and more pleasant for the individual programmer.

- It is a high-level programming language, but it includes many low-level facilities to manipulate the computer's memory.
- It is the first Object-oriented programming language and most popular after C language.
- C++ is a superset of C.

Features of C++

- C++ is a **compiled, case-sensitive, free-form programming language.**
- It is used to **write device drivers and other softwares.**
- It is used for teaching and research because of **clean basic concepts.**
- It **supports object-oriented programming** including four features such as **encapsulation, data hiding, inheritance, polymorphism.**
- An **Apple Macintosh or Windows** has indirectly used C++ because the primary user interface of these operating systems are written in C++.

ANSI

- ANSI stands for **American National Standards Institution.**
- If a program is written in ANSI C++, it guarantees to run on any computer whose supporting software conforms to the standard.
- C++ includes ANSI standard as a core language and also includes extra machine-dependent features to allow smooth interaction with different computers' operating systems.
- ANSI standard ensures that C++ is portable.

Character Set of C++

- Character set is a set of valid characters.
- It is a combination of alphabets, digits, special symbols and white spaces.
-

Characters	List Included
Alphabets	A – Z, a – z
Digits	0 – 9

Special Characters	Space + - * / ^ \ () [] {} = != <> ' " $, ; : % ! & ? _ # <= >= @
Formatting Characters	Backspace, Horizontal tab, Vertical tab, Form feed and Carriage return.

Tokens

- A token is a group of characters.
- It is the smallest element of a C++ program which is meaningful to the compiler.

C++ uses the following types of Tokens:

1. Keywords
2. Identifiers
3. Data types
4. Operators

1. Keywords

- Keywords are the reserved identifiers that have special meanings.
- These reserved keywords cannot be used as identifiers in a program.
- All keywords are written in lower case.

Following are the keywords used in C++:

Asm	default	float	operator	static_cast	union
Auto	delete	for	private	struct	unsigned
Break	do	friend	protected	switch	using
Bool	double	goto	public	template	virtual
Case	dynamic	if	register	this	void

Catch	else	inline	reinterpret_cast	throw	volatile
Char	enum	int	return	true	wchar_t
Class	explicit	long	short	try	while
Const	extern	Mutable	signed	typedef	
const_cast	export	namespace	sizeof	typeid	
Continue	false	new	static	typename	

2. Identifiers

Identifier is a sequence of characters used to define various things like variables, constants, functions, classes, objects, structures, unions etc.

It follows the rules for the formation of an identifier:

- An identifier consists of alphabets, digits or underscores.
- It cannot start with a digit. It can start either with an alphabet or underscore.
- Identifier should not be a reserved word.
- C++ is case-sensitive. So, upper case and lower case letters are considered different identifiers from each other.
- Blank spaces and special symbols are not allowed except underscore.

3. Data types

- Data type is used for identifying the type of data and the memory locations are required for storing the type of data in the memory.
- Whenever a variable is declared it becomes necessary to define data type.

Following are the C++ data types:

Data type	Range	Bytes
Char	-127 to 128	1
signed char	-127 to 128	1
unsigned char	0 to 255	1
Int	-32768 to 32767	2

signed int	-32768 to 32767	2
unsigned int	0 to 65535	2
short int	-32768 to 32767	2
long int	-2147483648 to 2147483647	4
signed short int	-32768 to 32767	2
signed long int	-2147483648 to 2147483647	4
unsigned short int	0 to 65535	2
unsigned long int	0 to 4294967296	4
Float	$3.4e^{-38}$ to $3.4e^{38}$	4
Double	$1.7e^{-308}$ to $1.7e^{308}$	8
long double	$3.4e^{-4932}$ to $1.1e^{4932}$	10

4. Operators

- Operator is a symbol used to perform mathematical or logical manipulations.
- It specifies the order of operations in expressions that contain more than one operator.

It includes all C operators and has several new operators, as follows:

Operator	Description
Scope Resolution Operator (::)	It is used to identify and disambiguate identifiers used in different scopes.
Unary Operator (-)	It returns the negative value of the variable to which it precedes.
Casting Operator (())	It is used for type conversion.
Address of Operator (&)	It returns the address of a memory location.
sizeof() Operator	It returns the size of a memory location.
Comma (,) Operator	It allows grouping two statements where one is expected.
Indirection Operator or Value of Operator (*)	It defines pointer to a variable.
Ternary Operator (?:)	It is a conditional expression.
New	Creates an instance of an object type.
Delete	Deletes property of an object.

Arithmetic Operators

Operator	Description
+	Addition
–	Subtraction
*	Multiplication
/	Division
%	Modulus (Reminder)
++	Increment
—	Decrement

Comparison Operators

Operator	Description
= =	Equal To
!=	Not Equal To
<	Less Than
>	Greater Than
<=	Less Than or Equal To
>=	Greater Than or Equal To

Assignment Operators

Operator	Description
=	Simple Assignment
+=	Add and Assignment
-=	Subtract and Assignment

*=	Multiply and Assignment
/=	Divide and Assignment
%=	Modulus and Assignment

Logical Operators

Operator	Description
&&	Logical AND
||	Logical OR
!	Logical NOT

Bitwise Operators

Operator	Description
&	Bitwise AND
|	Bitwise OR
^	Bitwise XOR
~	Bitwise NOT
<<	Left Shift
>>	Right Shift

Scope Resolution Operator

- Scope Resolution Operator is denoted by **:: symbol.**
- It is used to define a function outside a class.

- When Local variable and global variable having same name then local variable gets the priority.
- C++ allows flexibility of accessing both the variables through a scope resolution operator.

Access Specifiers

- Access modifiers define the access control rules.
- It is used to set boundaries for availability of members of class.
  Following are the three access specifiers in C++:
  1. Public
  2. Private
  3. Protected

Access Specifier	Description
Public	It is accessible from anywhere outside the class but within a program.
Private	It cannot be accessed or viewed from outside the class.
Protected	It is similar to a private member but it can be accessed in child classes which are calle derived classes.

Preprocessor Directives

- Preprocessor directives contain special instructions which indicate how the program is prepared for compilation.
- Common preprocessor command is 'include' that tells the compiler to execute a program where some information is required from the specified header file.
- It starts with **'#'** character.
  It can be categorize into following directives:
  1. Inclusion Directives
  2. Macro Definition Directives
  3. Conditional Compilation Directives
  4. Other Directives

1. Inclusion Directives

- Inclusion category has only one directive called #include.

- It is used to include files into the current file.
  It can be used as follows:

Directives	Description
#include<stdio.h>	It includes **stdio.h** from include folder.
#include<iostream>	It includes cpp class library header input output stream. Stream is an object with properties that are defined by a class.
#include<my.cpp>	It includes my .cpp file from include folder.
#include"my.h"	It includes my.h file from current working folder.
#include"myfolder/abc.h"	It includes abc.h file from the myfolder which is available in current working folder.

2. Macro Definition Directives

It is used to define macros, which are one or more program statements like functions.

It includes two directives for macro definition:

1. #define : It is used to define a macro.
2. #undef : It is used to undefine a macro. The macro cannot be used after it is undefined.

3. Conditional Compilation Directives

- It is used to execute statements conditionally for debugging purpose, evaluating codes etc.

  It includes following directives:
  i. #if
  ii. #elif
  iii. #endif
  iv. #ifdef
  v. #ifndef

- The above macros are evaluated on compile time. Most compilers do not support the use of variables with these directives.

4. Other Directives

It includes following directives:

1. #error : If the #error directive is found, the program will be terminated.
2. #line : This directive is used to change the value _ LINE _ and _ FILE _ macros.
3. #pragma : This directive is used to allow suppression of specific error messages, manage heap and stack debugging etc.

Constants

- Constants are called as **Literals.**
- They are like variables, except that their value never changes during execution once defined.
- The **'const'** keyword is used to define constraints in C++.

Syntax:
const datatype constant_name;

Example:
const int a = 10;

- It is possible to put const either before or after the data type.

Example:
int const a = 10;

Literals

Following are the types of Literals:

Literals	Description
Integer Literal	It can be a decimal, octal or hexadecimal constant.
Floating-point Literal	It has an integral part, a decimal point, a fractional part and an exponent part.
Boolean Literal	It has two boolean literals: True and False.
Character Literal	These literals are enclosed in single quote.
String Literal	These literals are enclosed in double quotes.

Escape Codes

Escape Codes	Meaning
\n	Newline
\r	Carriage return
\t	Tab
\v	Vertical tab
\b	Backspace
\f	Form feed
\a	Alert
\'	Single quote
\"	Double quote
\?	Question mark
\\	Backslash

- Escape codes are used to present the characters which are difficult to express in the source code.
- All escape codes are start with a **blackslash (\).**

Standard Output (cout)

- The cout and insertion (<<) operator can print a message on the screen.

  Example:

```
cout<< "Welcome to C++!!!";
```

- To print the variables, there is no need to used **double quotes (“”).**

  Example:

```
int a=10;
count << a;
```

- The insertion operator **(<<)** can be used multiple times in a single statement.

  Example:

```
cout<< "Welcome" << "to" << "TC++";
```

- It is possible to combine variables and text.

Example:

```
int a = 10;
cout<< "Print number"<<a;
```

Standard Input (cin)

- The cin and extraction (>>) operator reads input from the keyboard.

Example

```
int num;
cout<< "Enter number:"
cin>>num;
cout<<num;
```

- The cin operator returns the variable type.

Manipulators

- Manipulators are used to manipulate the data by the programmer's choice of display.
- It is used to specify certain formats of the output to be displayed when using the cout statement.
- It performs certain special operations such as moving the cursor to new line, fixing the width of a data value etc.

Following are the manipulators used to display program's output properly:

Manipulators	Description
Endl	It feeds the whole line and then points the cursor to the beginning of the next line. '\n' is used instead of endl for the

	same purpose.
Setw	It sets the minimum field width on output.
Setfill	It is used after setw manipulator. It is used for filling the fields.
Setprecision	It is used to set the precision for a data after the decimal point.

Storage Classes

- Storage class is a type specifier that governs the lifetime, linkage and memory location of objects.
- It defines the scope and lifetime of variables or functions within a C++ program.

Following are the C++ storage classes:

Storage Class	Description
Auto	It is the default storage class for all local variables.
Register	This storage class is used to define local variables that should be stored in a register instead of RAM.
Static	This class is used for specifying the static variable.
Extern	This class is used to give a reference of a global variable which is available to all program files.
Mutable	This class allows a member of an object to override constness that means this class can be modified by a const membe function.

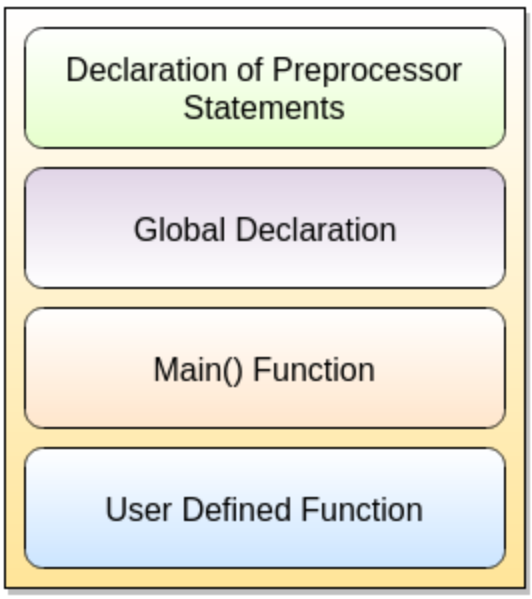


Fig. Program Structure of C++

The above diagram shows the basic program structure of C++.

Declaration section includes different library functions and header files. All preprocessor directives are written in this section.

Global declaration includes structure, class, variable. All global variables are declared here.

Main() function is an entry point for all the function. Every C++ program starts with main() function.

Example : / First C++ Program */*

```
#include <iostream>
int main()
{
    cout<<"Welcome to CPP";
    return 0;
}
```

Output:

Welcome to CPP

Following are the steps/ parts of the above C++ program:

/* First C++ Program */	**/*...*/ comments** are used for the documentation to understand the code to others. These comments are ignored by the compiler.
#include<iostream>	It is a **preprocessor directive.** It contains the contents of **iostream header file**in the program before compilation. This header file is required for input output statements.
int/void	Integer (int) returns a value. In the above program it returns value 0. Void does not return a value so there is no need to write **return** keyword.
main()	It is an entry point of all the function where program execution begins.
Curly Braces {...}	It is used to group all statements together.
std::cout	It is requried when we use **#include . Std::cout** defines that we are using a name **(cout)** which belongs to **namespace std. Namespace** is a new concept introduced by ANSI where C++ standard libraries are defined. If **using namespace std** is placed into the program then it does not required to **write std::** throughout the code. **Namespace std** contains all the classes, objects and functions of the standard C++ library.
"Welcome to TutorialRide"	The words in inverted commas are called a **String.** Each letter is called a character and series of characters that is grouped together is called a String. String always be put between inverted commas.
<<	It is the **insertion stream operator.** This operator sends the content of variables on its right to the object on its left. In the above program, right operand is the string **"Welcome to TutorialRide"**and left operand is **cout** object. So it sends the string to the **cout** object and then **cout** object displays the string as a output on the screen.

C++ Class

- Class is a collection of data and function under a single name.
- Class is similar to function with only difference that it can have functions besides data items.
- It is a blueprint for objects.

Syntax:

```
class class_name
{
    Access specifier:
        Data members;
        Member functions(){}
};
```

Example

```
class employee
{
    public:
        int empid;
        string empname;
        float salary;
};
```

- Class definition starts with the keyword **class** followed by the class name and ends with a **semicolon (;).**
- The primary purpose of a class is to hold data/information.

C++ Object

- Object is an instantiation of a class.
- It has same relationship to class as variable has to the data type.
- Object is created from a class.

Syntax:
class_name variable_name;

Example

Consider the above Employee class. Object for Employee class can be defined as:

Employee e;

Here, object **e** of **Employee** class is defined.

- It is exactly the same sort of declaration that we do for the variables of different data types.

Example : Following example demonstrates the working of Objects & Class in C++

```
#include <iostream>
using namespace std;
class Employee
{
    private:
        int empid;
        string empname;
        float salary;
    public:
        int emp_details()
        {
            empid=100;
            empname="ABC";
            salary=10000.0;
        }
        int show()
        {
```

```
            cout<<"Employee Id : "<<empid<<endl;
            cout<<"Employee Name : "<<empname<<endl;
            cout<<"Employee Salary : "<<salary<<endl;
        }
};
int main()
{
   Employee e;
   e.emp_details();
   e.show();
   return 0;
}
```

Output:

```
Employee Id : 100
Employee Name : ABC
Employee Salary : 10000
```

- In the above program, there are three data members **empid, empname & salary.** Two member functions **emp_details() & show()** are defined under **Employee** class.
- employee. Then, function **show()** for the object **e** is executed which displays details of the employee and returns it to the calling function.

following is the difference between Class and Structure:

Polymorphism

Class	Structure
Class is a reference type.	Structure is a value type.
In class, object is created on the heap memory.	In structure, object is created on the stack memory.
It supports inheritance.	It does not support inheritance.
It includes all types of constructors and destructors.	It includes only parameterized constructors.
Object can be created using **new** keyword. **For eg.** Test t = new Test();	Object can be created without using the **new**keyword. **For eg.** Test t;
The member variable of class can be initialized directly.	The member variable of structure cannot be initialized directly.

- Polymorphism word is derived from 2 greek words **Poly** and **Morphs** which means many forms.
- Polymorphism means having multiple forms.
- It is done by method overriding when both super class(Base class) and sub class (Parent class) have same member function but with different definition.
- Method overriding is called when two or more methods (functions) have exactly same method name, return type, number and types of parameters as the method in the parent class.
- Method overriding cannot be done within a class. It requires a base class and a derived class.

Example : Program demonstrating Method Overriding

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

```
class BaseClass
{
    public:
       int display()
       {
           cout <<"Super Class\t";
       }
};
class DerivedClass:public BaseClass
{
    public:
       int display()
       {
           cout <<"\n Sub Class";
       }
};
int main()
{
    BaseClass b;          //Base class object
    DerivedClass d;     //Derived class object
    b.display();         //Early Binding Occurs
    d.display();
}
```

Output:
Super Class
Sub Class

In the above program, **display()** function is overridden in the derived class. **BaseClass's object b** calling base class version of the function and **DerivedClass's object d** calling derived version of the function.

Abstraction

- Data abstraction means **hiding of data.**

- Abstraction is implemented automatically while writing the code in the form of class and object.
- It shows only important things to the user and hides the internal details.

Example : Program demonstrating Data Abstraction

```
#include<iostream>
using namespace std;
class Addition
{
   private: int a=10,b=10,c;  // Hidden data from outside world
      public:
         int add()
         {
            c=a+b;
            cout<<"Addition is : "<<c;
         }
};
int main()
{
    Addition a;
    a.add();
    return 0;
}
```

Output:

Addition is : 20

- In the above example, class **Addition** adds numbers together and returns the addition or sum. The public member **add()** function is the interface to the outside world and a user needs to know to use the class. The private member **int a,b,c** are something that the user does not need to know about, but is needed for the class to operate properly.

- Abstraction provides security for the data from the unauthorized methods and can be achieved by using class.

Encapsulation

- Encapsulation is a process of bundling the data and functions in a single unit.
- It binds the data and functions together that manipulate the data and keeps them safe from outside interference and misuse.
- It is used to secure the data from other methods. When making a data private, these data are used within the class only and not accessible outside the class.

Advantages of Encapsulation

- Encapsulation provides abstraction between an object and its users.
- It protects an object from unwanted access by unauthorized users.

Interfaces

- Interface describes the behavior of a class and contains only a **Virtual Destructor** and **Pure Virtual Functions.**
- These are the non-instantiable types which contains only function declarations.
- An object must supply definitions for all of the functions to implement an interface.
- Interfaces are implemented using **Abstract Class.**
- Abstract class provides an appropriate base class from which other classes can inherit and it cannot be used to instantiate objects and serves only as an interface.
- At least one function is required as **Pure Virtual Function** for making a class abstract. This function is specified by placing **"=0"** in its declaration as follows:

```
Class Employee
{
    public:
        virtual int emp_details() = 0;       //Pure Virtual Function
    private:
        int empid;
        string empname;
```

```
        float salary;
}
```

- Interface specifies pure virtual function declarations into a base class.

Example : Program implementing Interface

Following program demonstrates the parent class provides an interface to the base class to implement a function called area():

```
#include <iostream>
using namespace std;
class Shape  // Base class
{
    public:
        virtual float area() = 0;    // Pure Virtual Function
        float radious(float r)
        {
            ar = r;
        }
        float SetLength(float l)
        {
            length = l;
        }
        float SetBreadth(float b)
        {
            breadth=b;
        }
        float SetSideSquare(float s)
        {
            side=s;
        }
    protected:
        float ar;
```

```
        float length;
        float breadth;
        float side;
};
class Circle: public Shape   // Derived class
{
    public:
        float area()
        {
            return (3.14*ar*ar);
        }
};
class Rectangle: public Shape   //Derived class
{
    public:
        float area()
        {
            return (length*breadth);
        }
};
class SideSquare: public Shape //Derived class
{
    public:
         float area()
         {
             return (side*side);
         }
};
int main()
{
   Circle cr;
   cr.radious(3);
   cout << "Area of Circle : " << cr.area() << endl; // Print the area of circle.
```

```
    Rectangle rect;
    rect.SetLength(3);
     rect.SetBreadth(6);
    cout << "Area of Rectangle : " << rect.area() << endl;   // Print the area of
rectangle.
    SideSquare sq;
    sq.SetSideSquare(2);
    cout << "Area of Square : " << sq.area() << endl;    // Print the area of
Square.
    return 0;
}
```

Output:
Area of Circle : 28.26
Area of Rectangle : 18
Area of Square : 4

In above program, an abstract class defines an interface in terms of **area()** and three other classes **Circle, Rectangle** and **SideSquare** implement the same function but with different algorithm to calculate the area specific to the shape.

Decision Making Structure

- Conditional statement or decision making statement allows to make a decision, based on a condition.
- It specifies one or more conditions to be tested or evaluated by the programmer.

Following are the types of decision making statements:

1. If Statement
2. If . . . Else Statement
3. Nested If Statements
4. Switch Statement

1. If Statement

- If statement is the simplest way to modify the control flow of program.

- It is a conditional branching statement.
- This statement consists of a boolean expression followed by one or more statements.

Syntax:

```
if (condition)
{
    Statement 1;
    Statement 2;
    .
    .
    .
}
```

Example : Demonstrating the If Statement

```
#include <iostream>
using namespace std;
int main()
{
    int num1=10, num2=20;
    if(num1 < num2)
    {
        cout<<"Num2 is greater";
    }
    return 0;
}
```

Output:
Num2 is greater

2. If . . . Else Statement

- It is a two-way decision making statement.
- When the boolean expression becomes false then else block will be executed.
- It is used to make decisions and execute statements conditionally.

Syntax:

```
if(Condition)
{
    Statements;
}
else
{
    Statements;
}
```

Flow Diagram of If-Else Statement

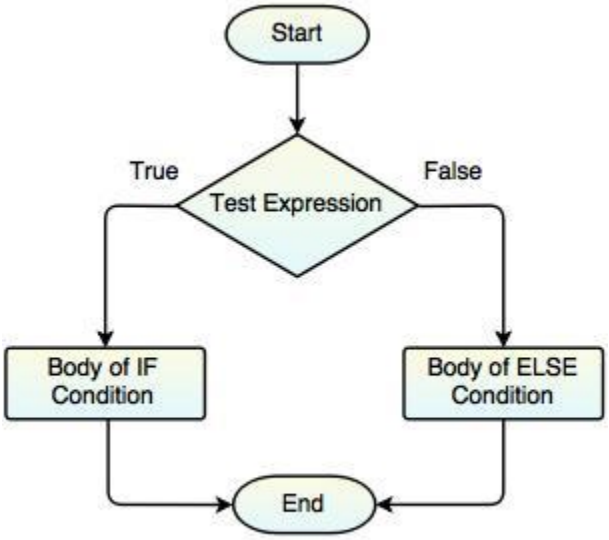


Fig. Flow Diagram of IF - ELSE Statement

Example : Demonstrating the If-Else Statement

```
#include <iostream>
using namespace std;
int main()
{
    int num1=10, num2=20;
    if(num1 > num2)
    {
        cout<<"Num2 is greater";
```

```
    }
    else
        cout<<"Num1 is smaller";
    return 0;
}
```

Output:
Num1 is smaller

3. Nested If . . . Else Statements

- Nested If . . . Else statements allow the use of one if or else if statement inside another if or else statements.
- This statement is like performing another if condition for true or false boolean value.

Syntax:

```
if (Condition)
{
    Statements;
}
else if (Condition n)
{
    Statements;
}
else
{
    Statements;
}
```

Example : Program demonstrating Nested If-Else Statements

```
#include <iostream>
using namespace std;
int main()
{
```

```
    int num1=20, num2=10;
    if(num1 > num2)
    {
        cout<<"Num2 is greater";
    }
    else if (num1 < num2)
    {
        cout<<"Num1 is smaller";
    }
    else
    {
        cout<<"Num1 and Num2 are equal";
    }
    return 0;
}
```

Output:
Num2 is greater

4. Switch Statement

- Switch statement is used to perform different actions on different conditions.
- This statement compares the same expression to several different values.

Following are the rules for Switch statement:

1. Switch case should have at most one default label.
2. Default case is optional.
3. Case labels must be unique, end with colon, integral type and have constant expression.
4. Break statement takes control out of the switch and two or more cases may share one break statement.
5. Relational operators are not allowed in Switch statement.
6. Macro identifier and **Const** variable are allowed in switch case statement.
7. Empty switch case is allowed.
8. Nesting switch is allowed.
9. Default case can be placed anywhere in the Switch statement.

Syntax

```
switch (Expression)
{
    case condition1:
        //Statements;
        break;
    case condition2:
        //Statements;
        break;
    case condition3:
        //Statements;
        break;
        .
        .
    case condition n;
        //Statements;
        break;
    default:
        //Statement;
}
```

Flow Diagram of Switch Statement

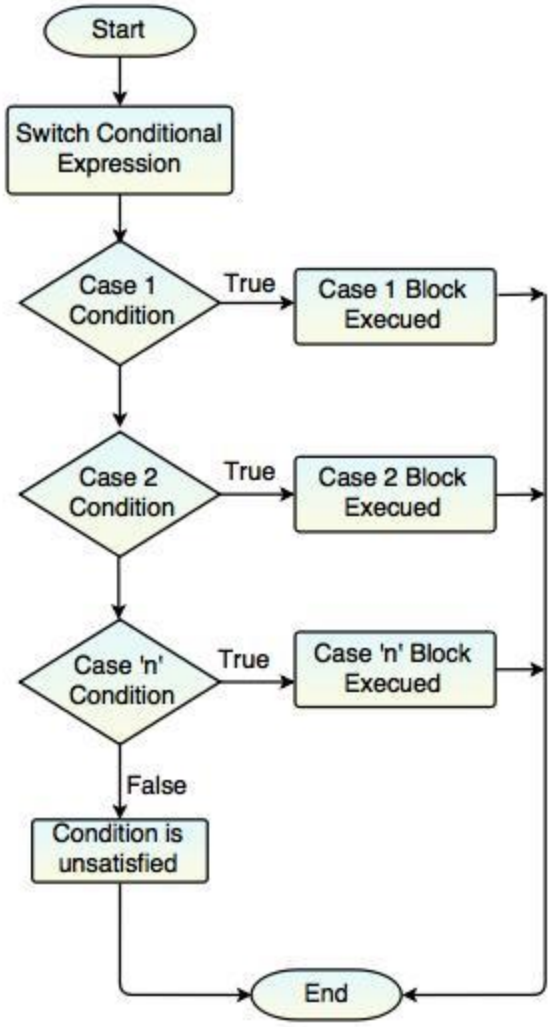


Fig. Flow Diagram of Switch Statement

Example : Demonstrating execution of Switch statement

```
#include <iostream>
using namespace std;
int main()
{
    char color='O';
    switch(color)
    {
        case 'R': cout<<"Red"<<endl;
```

```
            break;
        case 'G': cout<<"Green"<<endl;
            break;
        case 'B': cout<<"Blue"<<endl;
            break;
        case 'O': cout<<"Orange"<<endl;
            break;
        case 'P': cout<<"Pink"<<endl;
            break;
        case 'W': cout<<"White"<<endl;
            break;
        case 'Y' : cout<<"Yellow"<<endl;
            break;
        default: cout<<"Inavlid Color"<<endl;
    }
    return 0;
}
```

Looping Structure

Looping structure allows to execute a statement or group of statements multiple times.

It provides the following types of loops to handle the looping requirements:

1. While Loop
2. For Loop
3. Do . . . While Loop

1. While Loop

- While loop allows to repeatedly run the same block of code, until a given condition becomes true.
- It is called an **entry-controlled loop statement** and used for repetitive execution of the statements.
- The loop iterates while the condition is true. If the condition becomes false, the program control passes to the next line of the code.

Flow Diagram of While Loop

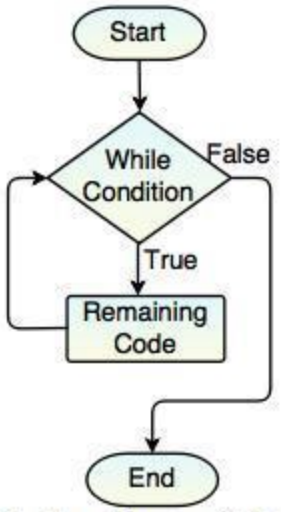


Fig. Flow Diagram of While Loop

Syntax:

```
while (Condition)
{
    //Statements;
}
```

Example : Demonstrating execution of While loop

Fibonacci series program to demonstrate execution of While loop.

```
#include <iostream>
using namespace std;
int main()
```

```
{
    int num1 = 0, num2 = 1, num3 = 0;
    cout<<"Fibonacci Series:"<<endl;
    while(num2 <= 10)
    {
        num3 = num1 + num2;
        num1 = num2;
        num2 = num3;
        cout<<num3<<endl;
    }
    return 0;
}
```

Output:

Fibonacci Series:
1
2
3
5
8
13

2. For Loop

- For loop is a compact form of looping.
- It is used to execute some statements repetitively for a fixed number of times.
- It is also called as **entry-controlled loop.**
- It is similar to while loop with only difference that it continues to process block of code until a given condition becomes false and it is defined in a single line.

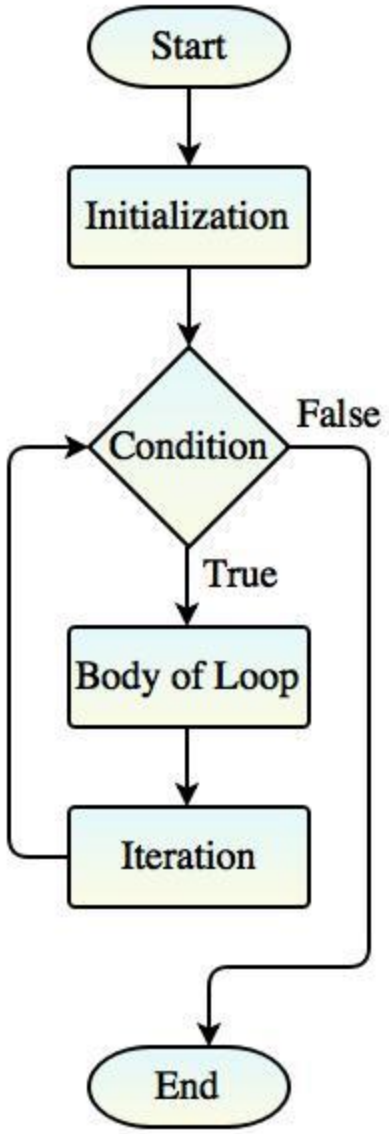
It includes three important parts:

1. Loop Initialization

2. Test Condition
3. Iteration

- All the above parts come in a single line separated by **semicolon (;).**

Flow Diagram of For Loop



Syntax:

```
for (initialization; test-condition; increment/decrement)
{
```

```
    //Statements;
}
```

Example : Demonstrating the execution of For Loop

```
#include <iostream>
using namespace std;
int main()
{
    int i = 1, j = 5;
    for(i=1; i<=j; i++)
    {
        cout<<"For Loop Execution"<<i<<endl;
    }
    return 0;
}
```

Output:

```
For Loop Execution:1
For Loop Execution:2
For Loop Execution:3
For Loop Execution:4
For Loop Execution:5
```

3. Do . . . While Loop

- Do . . . While loop is executed at least once, even if the condition is false.
- It is called as an **exit-controlled loop statement.**
- This loop does not test the condition before going into the loop. The condition will be checked after the execution of the body that means at the time of exit.
- It is guaranteed to execute the program at least one time.

Flow Diagram of Do . . . While Loop

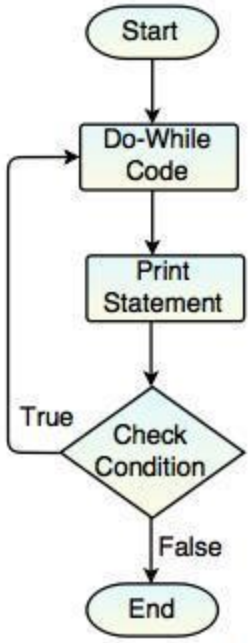


Fig. Flow Diagram of Do-While Loop

Syntax:

```
do
{
    //Statements;
}
while(Condition);
```

In **Do-while loop,** while condition should always have a semi-colon at the end.

Example : Demonstrating the execution of Do-While loop

```
#include <iostream>
using namespace std;
int main()
{
    int num=0;
    do
```

```
    {
        cout<<"Do While Loop Execution:"<<num<<endl;
        num++;
    }
    while(num<=5);
    return 0;
}
```

Output:

Do While Loop Execution:0
Do While Loop Execution:1
Do While Loop Execution:2
Do While Loop Execution:3
Do While Loop Execution:4
Do While Loop Execution:5

These control statements are used to change the normal sequence of execution of loop.

Following are the Loop control statements:

1. Break Statement
2. Continue Statement
3. Goto Statement

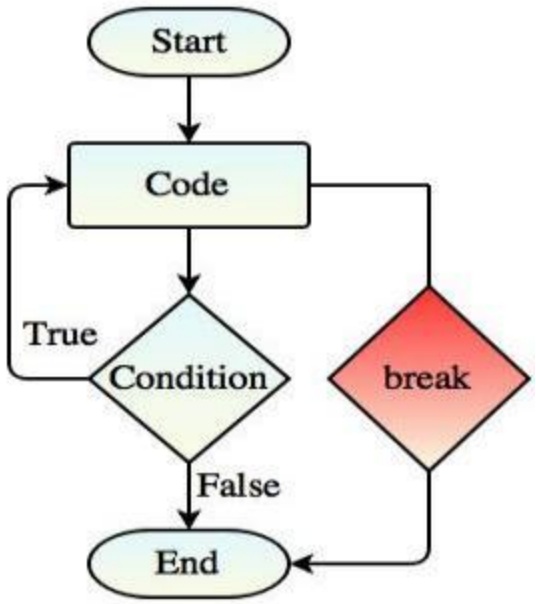
1. Break Statement

- Break statement is used to terminate loop or switch statements.
- It is used to exit a loop early, breaking out of the enclosing curly braces.

Syntax:

```
break;
```

Flow Diagram of Break Statement



Example : Demonstrating the Break statement's execution

```
#include <iostream>
using namespace std;
int main()
{
    int cnt = 0;
    do
    {
        cout<<"Value: "<<cnt<<endl;
        cnt++;
        if(cnt>5)
        {
            break;   //terminate the loop
        }
    }
    while(cnt<10);
```

```
    return 0;
}
```

Output:

Value: 0
Value: 1
Value: 2
Value: 3
Value: 4
Value: 5

2. Continue Statement

- Continue statement skips the remaining code block.
- This statement causes the loop to continue with the next iteration.
- It is used to suspend the execution of current loop iteration and transfer control to the loop for the next iteration.

  Syntax:

  continue;

- **In For Loop,** continue statement causes the conditional test and increment/ decrement statement of the loop gets executed.
- **In While Loop,** continue statement takes control to the condition statement.
- **In Do-While Loop,** continue statement takes control to the condition statement specified in the while loop.

Flow Diagram of Continue Statement

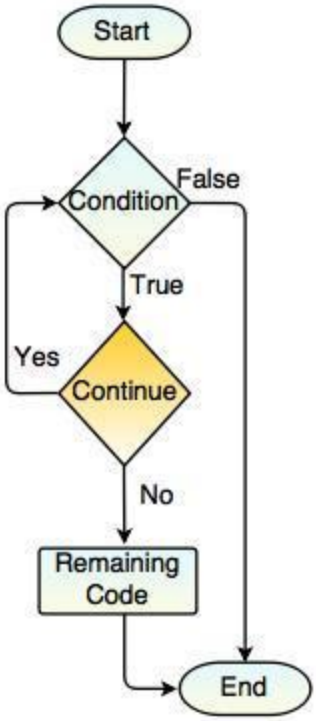


Fig. Flow Diagram of Continue Statement

Example : Demonstrating the execution of Continue statement

```
#include <iostream>
using namespace std;
int main()
{
    int cnt = 0;
    do
    {
        cnt++;
        if(cnt>5 && cnt<10)
           continue;
           cout<<"Value: "<<cnt<<endl;
    }
    while(cnt<11);
```

```
    return 0;
}
```

Output:

Value: 1
Value: 2
Value: 3
Value: 4
Value: 5
Value: 10
Value: 11

3. Goto Statement

- Goto statement transfers the current execution of program to some other part.
- It provides an unconditional jump from goto to a labeled statement in the same function.
- It makes difficult to trace the control flow of program and should be avoided in order to make smoother program.

Syntax:

```
goto label;
.
.
label: Statement;

Label is an identifier that identifies a labeled statement, followed by a colon (:).
```

- It is used to exit from deeply nested looping statements.
- If we avoid the goto statement, it forces a number of additional tests to be performed.

Flow Diagram of Goto Statement

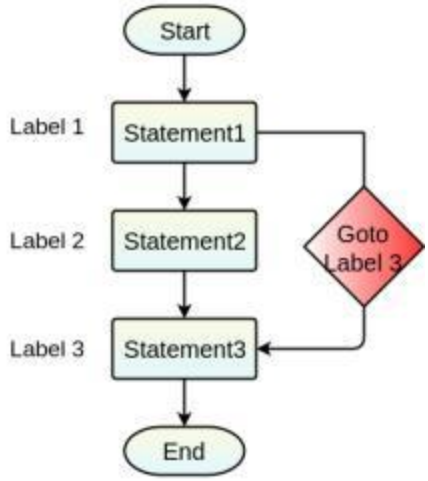


Fig. Flow Diagram of Goto Statement

Example : Demonstrating the execution of Goto Statement

```
#include <iostream>
using namespace std;
int main ()
{
    int no = 0;
    label1:do
    {
        if( no == 4)
        {
            no = no + 1;    //Skip the iteration
            goto label1;
        }
        cout << "Value : " << no << endl;
        no = no + 1;
    }while( no < 5 );
    return 0;
}
```

Output:

Value : 0
Value : 1
Value : 2
Value : 3
Value : 5

Introduction to function

- Function is a block of code that performs some operation.
- It is a self-contained block of statements that perform a task.
- Function is used to break down the complex program into the smaller chunks.
- It is useful for encapsulating common operations in a single reusable block that clearly describes what the function does.
- It defines input parameters that enable callers to pass arguments into the function and returns a value as output.

Syntax:

```
return_type function_name (argument_list)
{
    //Statements;
}
```

Following are the parts of a function:

return_type	It is a data type that function returns. It is not necessary that function will always return a value.
function_name	Function name is the actual name of the function.
argument_list / parameters	It allows passing arguments to the function from the location where it is called from. The argument list is separated by comma.
Function body	It contains a collection of statements that define what the function does.

Example : Program demonstrating the Function

```
#include <iostream>
using namespace std;
int addition(int no1, int no2);                    //Function declaration
int main()
{
    int num1=10, num2=20, result;                  //Local variable
    result = addition(num1,num2);        //Calling function. num1 & num2 are
Actual Parameters
    cout<<"Addition is : "<<result<<endl;
    return 0;
}
int addition(int no1, int no2)     //Function definition. no1 & no2 are Formal
Parameters
{
    int disp;
    disp=no1+no2;
    return disp;
}
```

Output:

```
Addition is : 30
```

Recursive Functions

- A function that calls itself is called as a **Recursive function.**
- It is useful for solving mathematical problems like calculating factorial of a number, fibonacci series etc.

Example: Program demonstrating Recursive function

Calculating the factorial of a number using recursive function.

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

```
int factor (int);
int main()
{
     int x = 5,fact;
     fact = factor(x);
     cout<<"Factorial is : "<<fact<<endl;
     return 0;
}
int factor (int y)
{
    int a;
    if (y == 1)
        return 1;
    else
        a = y*factor(y-1);
        return (a);
}
```

Output:
Factorial is : 120

Call by Value and Call by Reference

Following is the difference between Call by Value and Call by Reference:

Call by Value	Call by Reference
It is the method of passing arguments to a function that passes the actual value of an argument into the formal parameter of the function.	It is the method of passing arguments to a function that passes the reference of an argument into the formal parameter.
Original value cannot be changed or modified in call by value.	Original value is changed or modified in call by reference.

It passes value to the function.	It passes address of the value to the function.
Actual and formal parameters will be created at different memory location.	Actual and formal parameters will be created at sam location.
In call by value, changes made to the parameter inside the function have no effect on the argument.	In call by reference, changes made to the parameter affect the argument, because address is used to acc the actual argument.

Example
 	
#include <iostream>
using namespace std;
void swap(int a, int b)
{
int temp;
temp = a;
a = b;
b = temp;
}
int main()
{
int a = 10, b = 20;
swap(a, b); //passing value to function
cout<<"Value of a : "<<a<<endl;
cout<<"Value of b : "<<b<<endl;
return 0;
}
 	
Output
 	
Value of a : 10
Value of b : 20
	Example
	
	#include <iostream>
	using namespace std;
	void swap(int *a, int *b)
	{
	int temp;
	temp = *a;
	*a = *b;
	*b = temp;
	}
	int main()
	{
	int a = 10, b = 20;
	swap(&a, &b); //passing value to function
	cout<<"Value of a : "<<a<<endl;
	cout<<"Value of b : "<<b<<endl;
	return 0;
	}
	
	Output
	
	Value of a : 20
	Value of b : 10

Inline Function

- A function defined in the body of a class declaration is an inline function.
- Inline function is a combination of macro and function.
- It is a powerful concept in C++ programming language.
- This function increases the execution time of a program.
- Inline is a request to the compiler. It is an optimization technique used by the compiler.
- The keyword **inline** is used before the function name to make function inline.

Syntax:

```
inline function_name()
{
    //Function body
}
```

Example : Demonstrating the Inline function execution

```
#include <iostream>
using namespace std;
inline void display()
{
    cout<<"Welcome to TutorialRide";
}
int main()
{
    display();  // Call it like a normal function
}
```

Output:
Welcome to TutorialRide

Where does the Inline function not work?

- Inline function does not work if the functions are recursive.

- Inline function is not used when the function contains static variables.
- Inline function does not return any value even if the return statement is exists in the function.

Advantages of Inline Function

- Inline function does not require calling function overhead.
- It makes the program faster.
- Inline function increases locality of reference by utilizing instruction cache.
- It saves overhead of return call from a function.

Disadvantages of Inline Function

- Inline function increases function size so that it may not fit in the cache and causes lots of cache miss.
- If Inline function is used in the header files, it increases the header file size and makes it unreadable.
- Inline function is not useful for embedded system where large binary size is not preferred due to memory size constraints.

What is member function of a class?

Member function of a class is a function that must be declared inside the class.

The member functions can be defined as

1. Inside Member Function
2. Private Member Function
3. Outside Member Function

1. Inside Member Function

Inside member function class can be declared in public or private section.

Example : Program to demonstrate member function

Accessing private member of a class using member function

```
#include <iostream>
using namespace std;
class employee
{
    private:              //Private section starts
        int empid;
        float esalary;
    public:             //Public section starts
        void display()     //Member function
        {
            empid = 001;
            esalary = 10000;
            cout<<"Employee Id is : "<<empid<<endl;
            cout<<"Employee Salary is : "<<esalary<<endl;
        }
};
int main()
{
     employee e;      //Object Declaration
     e.display();       //Calling to member function
     return 0;
}
```

Output:

```
Employee Id is : 1
Employee Salary is : 10000
```

- In the above program, the member function **display()** is defined inside the class in public section.
- In **int main()** function, object **'e'** is declared. An object has permission to access the public members of the class.
- The object **'e'** invokes the public member function **display().**
- The public member function can access the private members of the same class.

- The **display()** function initializes the private member variables and displays the contents on the console.

2. Private Member Function

- It is possible to access private data member of a class using public member function.
- Private member function is invoked by public member function of the same class.

Example : Program demonstrating public member function

Accessing private member function using public member function

```
#include <iostream>
using namespace std;
class employee
{
    private:              //Private section starts
       int empid;
       float esalary;
       void addvalues()      //Private member function
       {
            empid = 001;
            esalary = 10000;
       }
    public:             //Public section starts
       void display()     //Public member function
       {
            addvalues();       //Calling to private member function
            cout<<"Employee Id is : "<<empid<<endl;
            cout<<"Employee Salary is : "<<esalary<<endl;
       }
};
int main()
```

```
{
    employee e;         //Object Declaration
    e.display();         //Calling to public member function
    //e.addvalues(); cannot be accessible
    return 0;
}
```

Output:

```
Employee Id is : 1
Employee Salary is : 10000
```

- In the above example, the public member function **display()** invokes the private member function **addvalues().**
- The private section of a class employee contains one member function **addvalues().**
- The **display()** function is defined in public section. In int **main()** function, **'e'** is an object of class employee.
- The object **'e'** cannot access the private member function. To execute the private member function, the private function must be invoked using public member function.

3. Outside Member Function

- If a function is small and inside the class, it is considered as an **Inline function.**
- If a function is large, it should be defined outside the class.
- When the member function of a class is defined outside the class they are called as **Externally defined member function.**
- **Scope Resolution (::) operator** is used to define the member function of the class outside the scope and prototype declaration of function must be declared inside the class.

Syntax:

```
Return_type Class_name :: Function_name(argument_list)
{
    //Statements;
}
```

Example : Define member function class outside the scope/class

```
#include <iostream>
using namespace std;
class employee
{
    private:                //Private section starts
        int empid;
        float esalary;
    public:                 //Public section starts
        void display(void);     //Prototype Declaration
};  //End of class
void employee :: display()     //Function Definition Outside the Class
{
    empid = 001;
    esalary = 10000;
    cout<<"Employee Id is : "<<empid<<endl;
    cout<<"Employee Salary is : "<<esalary<<endl;
}
int main()
{
    employee e;      //Object Declaration
    e.display();       //Calling to public member function
    return 0;
}
```

Output:

Employee Id is : 1

Employee Salary is : 10000

- In the above program, the prototype declaration of **display()** function is declared inside the class terminated by class definition.
- The function body of **display()** function is defined inside the class.
- The function declaration of **display()** function is, **void employee :: display()** where, **void** is a return type and **employee** is a class name. The **scope resolution (::) operator** separates the class name and function name, followed by the body of function which is defined.

Inheritance-One of the most important concepts in object-oriented programming is that of inheritance. Inheritance allows us to define a class in terms of another class, which makes it easier to create and maintain an application. This also provides an opportunity to reuse the code functionality and fast implementation time.

When creating a class, instead of writing completely new data members and member functions, the programmer can designate that the new class should inherit the members of an existing class. This existing class is called the **base**class, and the new class is referred to as the **derived** class.

The idea of inheritance implements the **is a** relationship. For example, mammal IS-A animal, dog IS-A mammal hence dog IS-A animal as well and so on.

Base and Derived Classes

A class can be derived from more than one classes, which means it can inherit data and functions from multiple base classes. To define a derived class, we use a class derivation list to specify the base class(es). A class derivation list names one or more base classes and has the form −

```
class derived-class: access-specifier base-class
```

Where access-specifier is one of **public, protected,** or **private**, and base-class is the name of a previously defined class. If the access-specifier is not used, then it is private by default.

Consider a base class **Shape** and its derived class **Rectangle** as follows −

```
#include <iostream>


using namespace std;


// Base class

class Shape {

   public:
```

```
      void setWidth(int w) {
         width = w;
      }
      void setHeight(int h) {
         height = h;
      }

   protected:
      int width;
      int height;
};

// Derived class
class Rectangle: public Shape {
   public:
      int getArea() {
         return (width * height);
      }
};

int main(void) {
   Rectangle Rect;

   Rect.setWidth(5);
   Rect.setHeight(7);

   // Print the area of the object.
   cout << "Total area: " << Rect.getArea() << endl;
```

```
   return 0;
}
```

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

```
Total area: 35
```

Access Control and Inheritance

A derived class can access all the non-private members of its base class. Thus base-class members that should not be accessible to the member functions of derived classes should be declared private in the base class.

We can summarize the different access types according to - who can access them in the following way –

Access	public	protected	private
Same class	yes	yes	yes
Derived classes	yes	yes	no
Outside classes	yes	no	no

A derived class inherits all base class methods with the following exceptions –

- Constructors, destructors and copy constructors of the base class.
- Overloaded operators of the base class.
- The friend functions of the base class.

Type of Inheritance

When deriving a class from a base class, the base class may be inherited through **public, protected** or **private** inheritance. The type of inheritance is specified by the access-specifier as explained above.

We hardly use **protected** or **private** inheritance, but **public** inheritance is commonly used. While using different type of inheritance, following rules are applied –

- **Public Inheritance** – When deriving a class from a **public** base class, **public** members of the base class become **public** members of the derived class and **protected** members of the base class become **protected** members of the derived class. A base class's **private**members are never accessible directly from a derived class, but can be accessed through calls to the **public** and **protected** members of the base class.
- **Protected Inheritance** – When deriving from a **protected** base class, **public** and **protected** members of the base class become **protected** members of the derived class.
- **Private Inheritance** – When deriving from a **private** base class, **public** and **protected** members of the base class become **private**members of the derived class.

Multiple Inheritance

A C++ class can inherit members from more than one class and here is the extended syntax –

```
class derived-class: access baseA, access baseB....
```

Where access is one of **public, protected,** or **private** and would be given for every base class and they will be separated by comma as shown above. Let us try the following example –

```
#include <iostream>


using namespace std;


// Base class Shape

class Shape {

   public:
```

```
      void setWidth(int w) {
         width = w;
      }
      void setHeight(int h) {
         height = h;
      }

   protected:
      int width;
      int height;
};

// Base class PaintCost
class PaintCost {
   public:
      int getCost(int area) {
         return area * 70;
      }
};

// Derived class
class Rectangle: public Shape, public PaintCost {
   public:
      int getArea() {
         return (width * height);
      }
};
```

```
int main(void) {

   Rectangle Rect;

   int area;


   Rect.setWidth(5);

   Rect.setHeight(7);


   area = Rect.getArea();


   // Print the area of the object.

   cout << "Total area: " << Rect.getArea() << endl;


   // Print the total cost of painting

   cout << "Total paint cost: $" << Rect.getCost(area) << endl;


   return 0;

}
```

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

```
Total area: 35
Total paint cost: $2450
```

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 in C++

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 –

```
#include <iostream>

using namespace std;


class printData {

   public:

      void print(int i) {

        cout << "Printing int: " << i << endl;

      }

      void print(double  f) {

        cout << "Printing float: " << f << endl;

      }

      void print(char* c) {

        cout << "Printing character: " << c << endl;

      }

};


int main(void) {

   printData pd;
```

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

   // Call print to print float
   pd.print(500.263);

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

   return 0;
}
```

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

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

Operators Overloading in C++

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 –

```
#include <iostream>

using namespace std;


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;

      }


      // Overload + operator to add two Box objects.

      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;
}
```

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

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

Overloadable/Non-overloadableOperators

Following is the list of operators which can be overloaded –

+	-	*	/	%	^
&	|	~	!	,	=

<	>	<=	>=	++	--
<<	>>	==	!=	&&	||
+=	-=	/=	%=	^=	&=
|=	*=	<<=	>>=	[]	()
->	->*	new	new []	delete	delete []

Following is the list of operators, which can not be overloaded –

::	.*	.	?:

Operator Overloading Examples

Here are various operator overloading examples to help you in understanding the concept.

Sr.No	Operators & Example
1	**Unary Operators Overloading**
2	**Binary Operators Overloading**
3	**Relational Operators Overloading**
4	**Input/Output Operators Overloading**
5	**++ and -- Operators Overloading**
6	**Assignment Operators Overloading**

7	**Function call () Operator Overloading**
8	**Subscripting [] Operator Overloading**
9	**Class Member Access Operator -> Overloading**

The word **polymorphism** means having many forms. Typically, polymorphism occurs when there is a hierarchy of classes and they are related by inheritance.

C++ polymorphism means that a call to a member function will cause a different function to be executed depending on the type of object that invokes the function.

Consider the following example where a base class has been derived by other two classes –

```
#include <iostream>

using namespace std;


class Shape {

   protected:

      int width, height;


   public:

      Shape( int a = 0, int b = 0){

         width = a;

         height = b;

      }

      int area() {

         cout << "Parent class area :" <<endl;
```

```
         return 0;
      }
};
class Rectangle: public Shape {
   public:
      Rectangle( int a = 0, int b = 0):Shape(a, b) { }

      int area () {
         cout << "Rectangle class area :" <<endl;
         return (width * height);
      }
};

class Triangle: public Shape {
   public:
      Triangle( int a = 0, int b = 0):Shape(a, b) { }

      int area () {
         cout << "Triangle class area :" <<endl;
         return (width * height / 2);
      }
};

// Main function for the program
int main() {
   Shape *shape;
   Rectangle rec(10,7);
   Triangle  tri(10,5);
```

```
   // store the address of Rectangle
   shape = &rec;

   // call rectangle area.
   shape->area();

   // store the address of Triangle
   shape = &tri;

   // call triangle area.
   shape->area();

   return 0;
}
```

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

```
Parent class area :
Parent class area :
```

The reason for the incorrect output is that the call of the function area() is being set once by the compiler as the version defined in the base class. This is called **static resolution** of the function call, or **static linkage** - the function call is fixed before the program is executed. This is also sometimes called **early binding** because the area() function is set during the compilation of the program.

But now, let's make a slight modification in our program and precede the declaration of area() in the Shape class with the keyword **virtual** so that it looks like this –

```
class Shape {
   protected:
      int width, height;
```

```
   public:
      Shape( int a = 0, int b = 0) {
         width = a;
         height = b;
      }
      virtual int area() {
         cout << "Parent class area :" <<endl;
         return 0;
      }
};
```

After this slight modification, when the previous example code is compiled and executed, it produces the following result –

```
Rectangle class area
Triangle class area
```

This time, the compiler looks at the contents of the pointer instead of it's type. Hence, since addresses of objects of tri and rec classes are stored in *shape the respective area() function is called.

As you can see, each of the child classes has a separate implementation for the function area(). This is how **polymorphism** is generally used. You have different classes with a function of the same name, and even the same parameters, but with different implementations.

Virtual Function

A **virtual** function is a function in a base class that is declared using the keyword **virtual**. Defining in a base class a virtual function, with another version in a derived class, signals to the compiler that we don't want static linkage for this function.

What we do want is the selection of the function to be called at any given point in the program to be based on the kind of object for which it is called. This sort of operation is referred to as **dynamic linkage**, or **late binding**.

Pure Virtual Functions

It is possible that you want to include a virtual function in a base class so that it may be redefined in a derived class to suit the objects of that class, but that there is no meaningful definition you could give for the function in the base class.

We can change the virtual function area() in the base class to the following –

```
class Shape {
   protected:
      int width, height;

   public:
      Shape(int a = 0, int b = 0) {
         width = a;
         height = b;
      }

      // pure virtual function
      virtual int area() = 0;
};
```

The = 0 tells the compiler that the function has no body and above virtual function will be called **pure virtual function**.

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

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

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

Declaring Arrays

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

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

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

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

Initializing Arrays

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

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

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

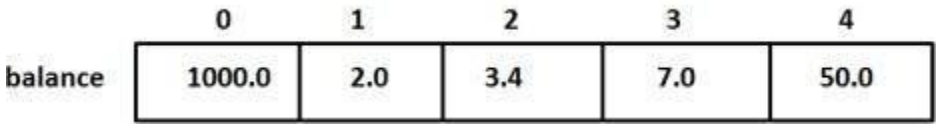
If you omit the size of the array, an array just big enough to hold the initialization is created. Therefore, if you write –

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

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

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

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



Accessing Array Elements

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

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

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

```
#include <iostream>
```

```
using namespace std;


#include <iomanip>
using std::setw;


int main () {


   int n[ 10 ]; // n is an array of 10 integers


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


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


   return 0;
}
```

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

```
Element       Value
      0         100
      1         101
      2         102
      3         103
      4         104
      5         105
      6         106
```

```
7          107
8          108
9          109
```

Arrays in C++

Arrays are important to C++ and should need lots of more detail. There are following few important concepts, which should be clear to a C++ programmer –

Sr.No	Concept & Description
1	**Multi-dimensional arrays** C++ supports multidimensional arrays. The simplest form of the multidimensional array is the two-dimensional array.
2	**Pointer to an array** You can generate a pointer to the first element of an array by simply specifying the array name, without any index.
3	**Passing arrays to functions** You can pass to the function a pointer to an array by specifying the array's name without an index.
4	**Return array from functions** C++ allows a function to return an array.

C++ provides following two types of string representations –

- The C-style character string.
- The string class type introduced with Standard C++.

The C-Style Character String

The C-style character string originated within the C language and continues to be supported within C++. This string is actually a one-dimensional array of characters which is terminated by a **null** character '\0'. Thus a null-terminated string contains the characters that comprise the string followed by a **null**.

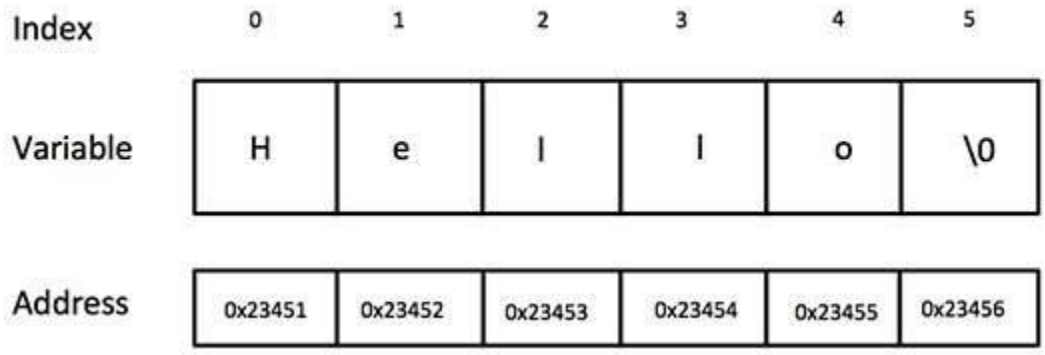
The following declaration and initialization create a string consisting of the word "Hello". To hold the null character at the end of the array, the size of the character array containing the string is one more than the number of characters in the word "Hello."

```
char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

If you follow the rule of array initialization, then you can write the above statement as follows –

```
char greeting[] = "Hello";
```

Following is the memory presentation of above defined string in C/C++ –



Actually, you do not place the null character at the end of a string constant. The C++ compiler automatically places the '\0' at the end of the string when it initializes the array. Let us try to print above-mentioned string –

```
#include <iostream>

using namespace std;

int main () {

   char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};

   cout << "Greeting message: ";
   cout << greeting << endl;
```

```
   return 0;
}
```

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

```
Greeting message: Hello
```

C++ supports a wide range of functions that manipulate null-terminated strings –

Sr.No	Function & Purpose
1	**strcpy(s1, s2);** Copies string s2 into string s1.
2	**strcat(s1, s2);** Concatenates string s2 onto the end of string s1.
3	**strlen(s1);** Returns the length of string s1.
4	**strcmp(s1, s2);** Returns 0 if s1 and s2 are the same; less than 0 if s1<s2; greater than 0 if s1>s2.
5	**strchr(s1, ch);** Returns a pointer to the first occurrence of character ch in string s1.
6	**strstr(s1, s2);** Returns a pointer to the first occurrence of string s2 in string s1.

Following example makes use of few of the above-mentioned functions –

```cpp
#include <iostream>
#include <cstring>

using namespace std;

int main () {

   char str1[10] = "Hello";
   char str2[10] = "World";
   char str3[10];
   int  len ;

   // copy str1 into str3
   strcpy( str3, str1);
   cout << "strcpy( str3, str1) : " << str3 << endl;

   // concatenates str1 and str2
   strcat( str1, str2);
   cout << "strcat( str1, str2): " << str1 << endl;

   // total lenghth of str1 after concatenation
   len = strlen(str1);
   cout << "strlen(str1) : " << len << endl;

   return 0;
}
```

When the above code is compiled and executed, it produces result something as follows –

```
strcpy( str3, str1) : Hello
strcat( str1, str2): HelloWorld
strlen(str1) : 10
```

The String Class in C++

The standard C++ library provides a **string** class type that supports all the operations mentioned above, additionally much more functionality. Let us check the following example –

```
#include <iostream>

#include <string>


using namespace std;


int main () {


   string str1 = "Hello";

   string str2 = "World";

   string str3;

   int  len ;


   // copy str1 into str3

   str3 = str1;

   cout << "str3 : " << str3 << endl;


   // concatenates str1 and str2

   str3 = str1 + str2;

   cout << "str1 + str2 : " << str3 << endl;
```

```
   // total length of str3 after concatenation
   len = str3.size();
   cout << "str3.size() :  " << len << endl;

   return 0;
}
```

When the above code is compiled and executed, it produces result something as follows –

```
str3 : Hello
str1 + str2 : HelloWorld
str3.size() :  10
```

C++ pointers are easy and fun to learn. Some C++ tasks are performed more easily with pointers, and other C++ tasks, such as dynamic memory allocation, cannot be performed without them.

As you know every variable is a memory location and every memory location has its address defined which can be accessed using ampersand (&) operator which denotes an address in memory. Consider the following which will print the address of the variables defined –

```
#include <iostream>

using namespace std;
int main () {
   int  var1;
   char var2[10];

   cout << "Address of var1 variable: ";
   cout << &var1 << endl;
```

```
   cout << "Address of var2 variable: ";

   cout << &var2 << endl;



   return 0;

}
```

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

```
Address of var1 variable: 0xbfebd5c0
Address of var2 variable: 0xbfebd5b6
```

What are Pointers?

A **pointer** is a variable whose value is the address of another variable. Like any variable or constant, you must declare a pointer before you can work with it. The general form of a pointer variable declaration is –

```
type *var-name;
```

Here, **type** is the pointer's base type; it must be a valid C++ type and **var-name** is the name of the pointer variable. The asterisk you used to declare a pointer is the same asterisk that you use for multiplication. However, in this statement the asterisk is being used to designate a variable as a pointer. Following are the valid pointer declaration –

```
int    *ip;    // pointer to an integer
double *dp;    // pointer to a double
float  *fp;    // pointer to a float
char   *ch     // pointer to character
```

The actual data type of the value of all pointers, whether integer, float, character, or otherwise, is the same, a long hexadecimal number that represents a memory address. The only difference between pointers of different data types is the data type of the variable or constant that the pointer points to.

Using Pointers in C++

There are few important operations, which we will do with the pointers very frequently. **(a)** We define a pointer variable. **(b)** Assign the address of a variable to a pointer. **(c)** Finally access the value at the address available in the pointer variable. This is done by using unary operator * that returns the

value of the variable located at the address specified by its operand. Following example makes use of these operations −

```
#include <iostream>

using namespace std;

int main () {
   int  var = 20;   // actual variable declaration.
   int  *ip;        // pointer variable

   ip = &var;       // store address of var in pointer variable

   cout << "Value of var variable: ";
   cout << var << endl;

   // print the address stored in ip pointer variable
   cout << "Address stored in ip variable: ";
   cout << ip << endl;

   // access the value at the address available in pointer
   cout << "Value of *ip variable: ";
   cout << *ip << endl;

   return 0;
}
```

When the above code is compiled and executed, it produces result something as follows −

```
Value of var variable: 20
Address stored in ip variable: 0xbfc601ac
Value of *ip variable: 20
```

Pointers in C++

Pointers have many but easy concepts and they are very important to C++ programming. There are following few important pointer concepts which should be clear to a C++ programmer –

Sr.No	Concept & Description
1	**Null Pointers** C++ supports null pointer, which is a constant with a value of zero defined in several standard libraries.
2	**Pointer Arithmetic** There are four arithmetic operators that can be used on pointers: ++, --, +, -
3	**Pointers vs Arrays** There is a close relationship between pointers and arrays.
4	**Array of Pointers** You can define arrays to hold a number of pointers.
5	**Pointer to Pointer** C++ allows you to have pointer on a pointer and so on.
6	**Passing Pointers to Functions** Passing an argument by reference or by address both enable the passed argument to be changed in the calling function by the called function.
7	**Return Pointer from Functions** C++ allows a function to return a pointer to local variable, static variable and dynamically allocated memory as well.