AISHWARYA COLLEGE OF EDUCATION

DEPARTMENT: COMPUTER SCIENCE

FACULTY NAME: SUMIT PUROHIT

Course	**BCA**
Class	**BCA 2nd YEAR**
Subject	**C++ PROGRAMMING**
Unit	**1,2,3**

TABLE OF CONTENT

1. PRINCIPLES OF OOP

The basic concepts of OOP revolve around several key terms. These key terms are:

1) Object
2) Class
3) Encapsulation
4) Abstraction
5) Inheritance
6) Polymorphism
7) Dynamic Binding
8) Reusability
9) Message Passing
10) Operator Overloading

1. Object

An object is a real-world run-time entity in OOP that has some attributes and behavior such as person, place and vehicle. It represents user-defined data types such as vectors, angles and programming constructs. An object contains data variables and functions to store and manipulate the data. For example, a student object consists of data variables such as name, rollno and marks, and functions such as get_data(), calculate_percentage and display_data. The get_data() function reads the data, the calculate_percentage function calculates the percentage and the display_data function displays the student data. The functions associated with an object are called its member functions. In an object, the data variables can only be accessed by its member functions. Figure shows the data and member functions associated with the student object.

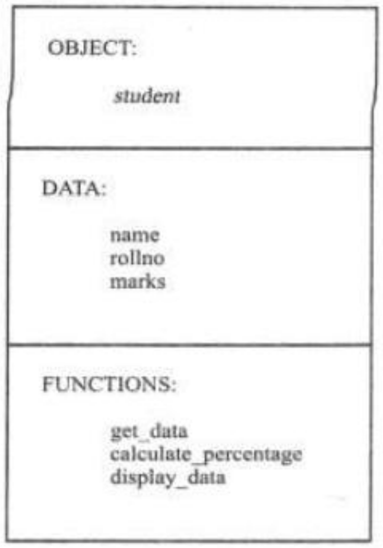


Figure: student object

2. Class

A class is a collection of similar types of objects that share common attributes and behavior. Objects are the variables of user-defined type called class. In an OOP program, you create classes to develop an application. You can categorize the real world into classes such as human being, automobiles, furniture, fruit and electrical appliances. Each class has some properties and functions. For example, the human being class has the properties such as height, weight, colour of eyes and colour of hair. The functions of human being class are walking, talking, sleeping, breathing, etc. You can take a class, Car, having properties such as model, colour and capacity. The functions that the Car class performs are speed, average, start and stop.
In technical terms, a class consists of member variables or properties and member functions. A class is also called abstract data type because it is only a template or design to be used by the objects of the class. Figure shows the class, Car.

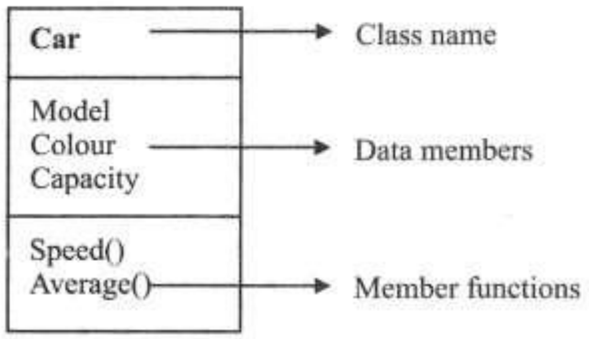


Figure: Class Structure

The Car class is divided into two parts—member variables or properties and member functions. Model, colour and capacity are the properties and speed and average are the member functions of the Car class.

3. Encapsulation

In C++, the technique of binding the data and functions together in a single unit is called encapsulation. Encapsulation helps secure the data and functions from outside interference as data is accessible only to the member functions of the object. These member functions provide an interface between the data of an object and the program. Encapsulation is also called data hiding because it protects data from direct access. To implement this technique, the data variables and methods are defined as private or public. The private data is accessible only to the member functions of the class and the public data represents the information that can be accessible to the external users of the class.

4. Abstraction

Abstraction means representing the essential features without mentioning the background details. In C++, objects use the concept of abstraction which helps represent only the important details related to an entity. Data can be accessed only by the member functions of a class. Member functions of an object act as an abstraction medium that is used to provide an interface between data and a program.

5. Inheritance

Inheritance is the process of defining new classes by extending the properties of other classes. The class that acquires the properties of other class is called derived class and the class from which the properties are inherited is called base class. The derived class shares the properties of the base class and also adds its own characteristics to create additional features. The derived class needs to define only those properties which are unique to it. Inheritance supports the concept of classification. For example, a car is a part of the four-wheeler class that, in turn, is a part of the vehicle class. It means a car has the properties of both four-wheeler and vehicle classes. It also means that a car is a subclass of the four-wheeler class that, in turn, is the subclass of the vehicle class. The vehicle class is the super class of all the classes. Figure shows the concept of inheritance in OOP.

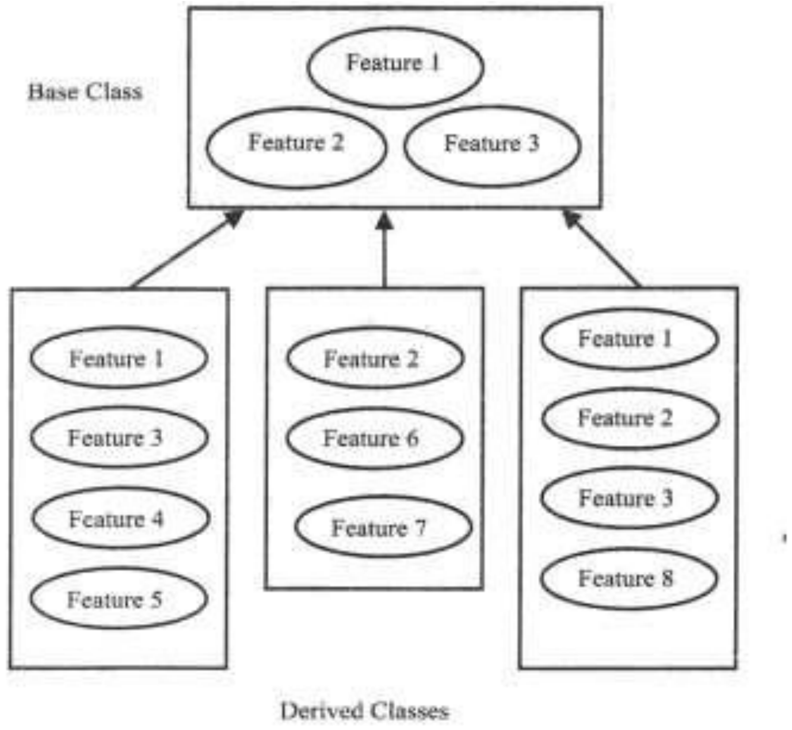


Figure Inheritance in OOP

For example, there are three base classes and from these base classes, three classes are derived. Each derived class contains its own unique features and the features of a base class also. Inheritance provides the concept of reusability which means using the existing classes to derive new classes. Using inheritance, you can add additional properties to an existing class without modifying it.

6. Polymorphism

Polymorphism means multiple forms that allow you to use a single interface for performing multiple actions in a program. Polymorphism helps reduce complexity in programming by allowing you to perform common operations using the same function name. The compiler selects the type of function that needs to be called to perform the specific task. For example, in a shape class, polymorphism enables the programmer to define different area methods for any geometrical shape such as circle, rectangle or triangle. The method name, Area 0 is same but the parameters passed in the Area 0 method are different for different shapes. Polymorphism is widely used to implement inheritance. Figure shows an example of polymorphism.

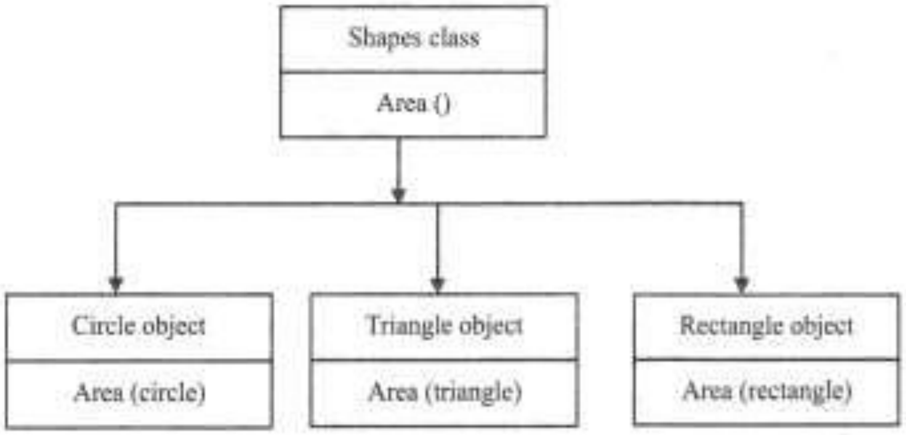


Figure: The Polymorphic Function

7. Dynamic Binding

Binding is the process of linking a function call to execute code. It means when a function is called in a program, the code is attached with it during binding. Binding is of two types—static binding and dynamic binding. Static binding means the code to be executed in response to a function call is known at the compilation time of the code. It means the function call is bound to its code when the program is being compiled. Dynamic binding resolves the code associated with a given function call at run-time. OOP supports dynamic binding. This means that OOP has the facility to link the function call to its code at run time using virtual functions.

8. Reusability

Object-oriented programming allows reusability of a code. If you have designed a class, any programmer can also use it in his program. You can add a new feature to an existing class in order to make a derived class using inheritance. Figure shows how you can implement reusability.

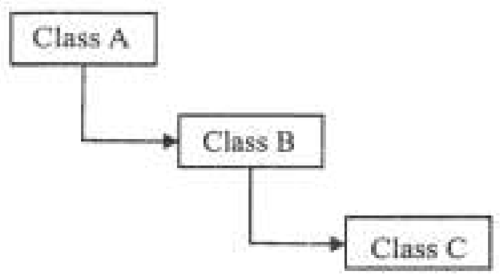


Figure Implementing Reusability

9. Message Passing

In the object-oriented programming, a program consists of objects that communicate with each other. You create classes that consist of properties and functions. Then, you create objects based on class definition and establish communication among objects. The message that the objects pass among themselves is a request for execution of a procedure. The message passing involves three things—name of the object, name of the function and information to be sent. An object can send and receive message until it is destroyed.

10. Operator Overloading

Operator overloading refers to a mechanism that you use to redefine operators for different objects by using the operator keyword. You can use operator overloading to modify the built-in operators to develop user-defined operators.

2. COMPARISON BETWEEN C AND C++

C++ is an extension of C and provides enhancements that C language does not support. Following points help compare C and C++:

1) C++ allows you to use Standard Library as well as the Standard Template Library containing various templates such as sort routine that you use to create real-world programs whereas C uses only Standard Library.

2) C++ provides memory management operator such as new and delete to allocate memory to the programs dynamically whereas C uses malloc and free memory management operators to allocate memory while writing a program.

3) C++ allows you to use stream operators such as cin and cout stored in the iostream class to perform tasks such as converting the entire input data in user readable form from machine code and vice versa whereas C doesn't provide any stream operator. C reads only a character from the entire stream of data.

4) C++ allows you to declare variables anywhere in the program whereas C allows you to declare a variable only at the beginning of a function.

5) C++ allows you to use the bool keyword to refer to booleans whereas C uses 0 and 1 for specifying false and true values of booleans.

6) C++ assigns .cpp extensions to the programs created in C++ whereas C programs are identified with .c extension.

7) C++ uses simpler language to write programs than C leading to lesser number of errors in the program.

8) C++ allows you to define a class that includes objects sharing a common behaviour whereas C does not allow you to define a class.

9) C++ allows you to use \\ and /* */ symbol to define a comment whereas C allows you to use only /* */ symbol to define a comment.

Fundamentals Of C++

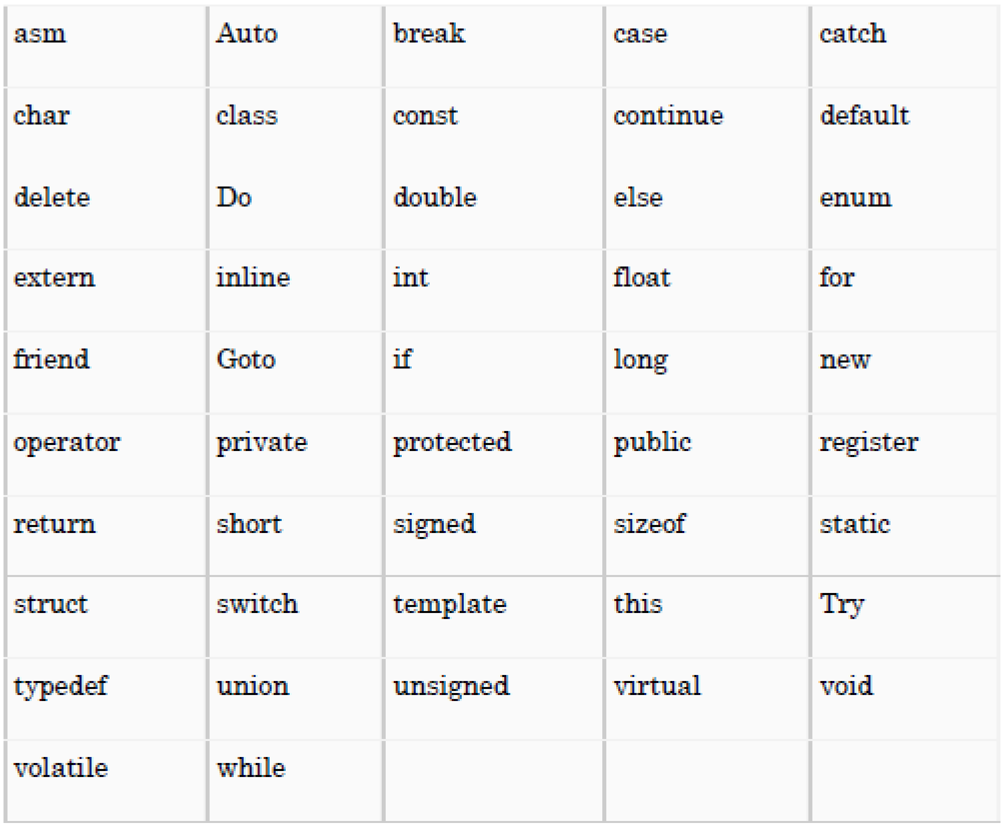
C++ language contains features of two programming languages—C that provides low-level features of programming and Simula67 that supports the concept of classes and objects. C++ is a superset of C language and therefore, most of the features of C are available in C++ also. You need to use tokens, identifiers and variables for creating a program.

3. TOKENS

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

1. Keywords

These are some reserved words in C++ which have predefined meaning to compiler called keywords. Some commonly used Keyword are given below:



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

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

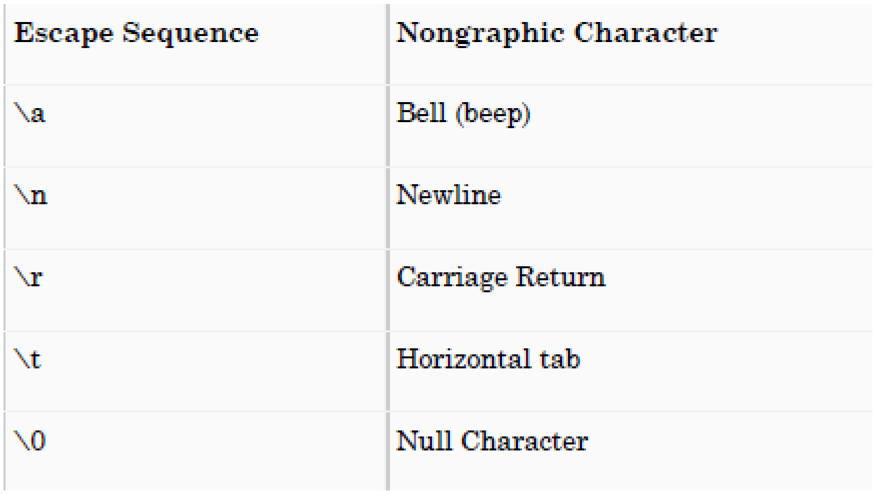
- An identifier can consist of alphabets, digits and/or underscores.
- It must not start with a digit
- C++ is case sensitive that is upper case and lower case letters are considered different from each other.
- It should not be a reserved word.

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

a. Integer-Constants
b. Character-constants
c. Floating-constants
d. Strings-constants

a. Integer Constants
Integer constants are whole number without any fractional part. C++ allows three types of integer constants.
Decimal integer constants : It consists of sequence of digits and should not begin with 0 (zero). **For example 124, - 179, +108.**
Octal integer constants: It consists of sequence of digits starting with 0 (zero). **For example. 014, 012.**
Hexadecimal integer constant: It consists of sequence of digits preceded by ox or OX.

b. Character constants
A character constant in C++ must contain one or more characters and must be enclosed in single quotation marks. For example 'A', '9', etc. C++ allows nongraphic characters which cannot be typed directly from keyboard, e.g., backspace, tab, carriage return etc. These characters can be represented by using an escape sequence. An escape sequence represents a single character. The following table gives a listing of common escape sequences.



Escape Sequence	Nongraphic Character
\a	Bell (beep)
\n	Newline
\r	Carriage Return
\t	Horizontal tab
\0	Null Character

c. Floating constants

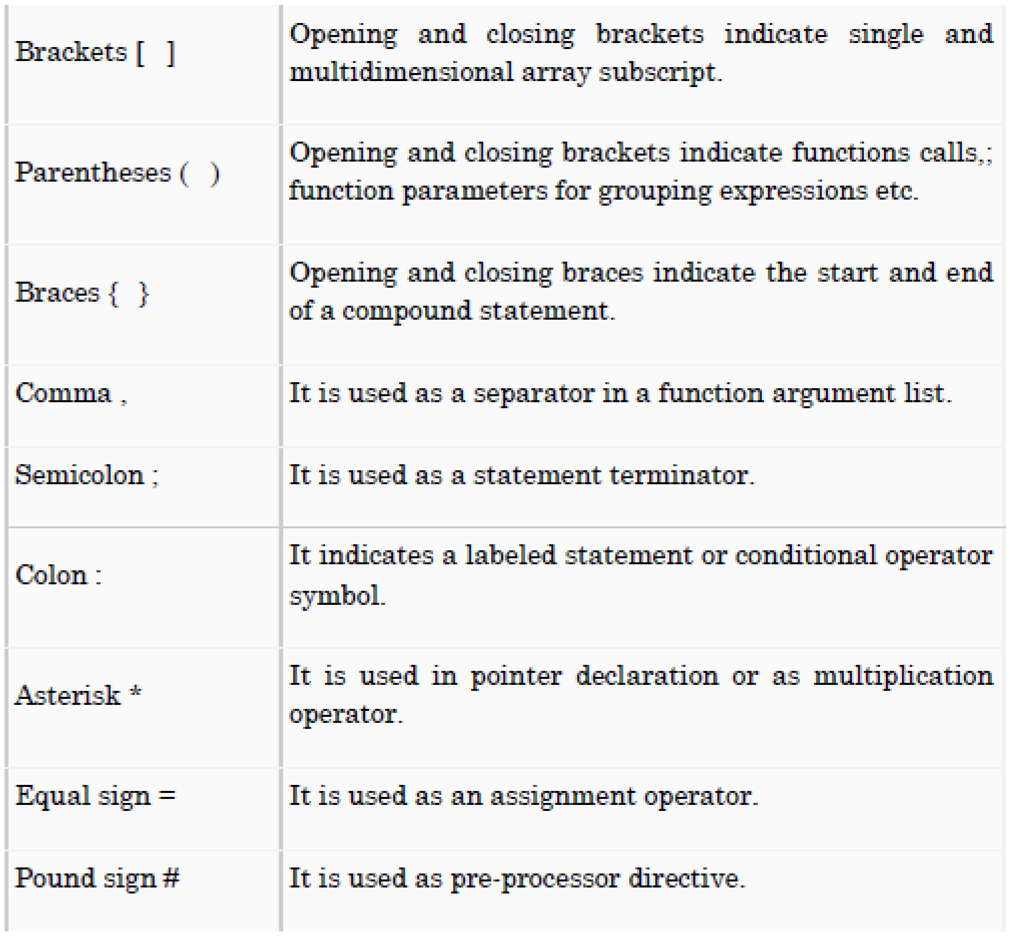
They are also called real constants. They are numbers having fractional parts. They may be written in fractional form or exponent form. A real constant in fractional form consists of signed or unsigned digits including a decimal point between digits. For example: 3.0, -17.0, -0.627 etc.

d. String Literals

A sequence of character enclosed within double quotes is called a string literal. String literal is by default (automatically) added with a special character '\0' which denotes the end of the string. Therefore the size of the string is increased by one character. For example "COMPUTER" will re represented as "COMPUTER\0" in the memory and its size is 9 characters.

4. Punctuators

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



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

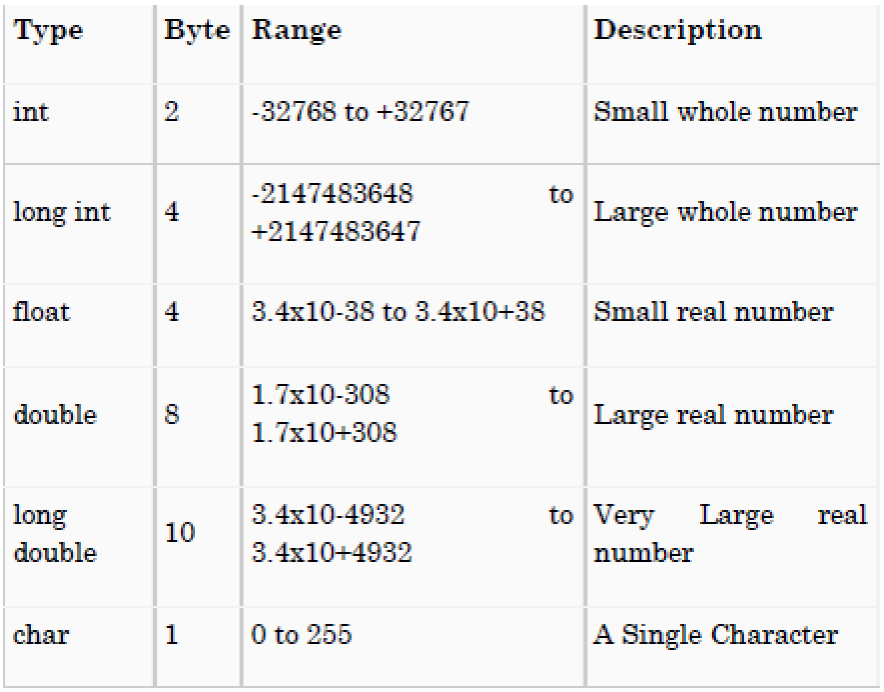
5. Operators

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

4. DATA TYPES

Basic data types

C++ supports a large number of data types. The built in or basic data types supported by C++ are integer, floating point and character. These are summarized in table along with description and memory requirement



Type	Byte	Range	Description
int	2	-32768 to +32767	Small whole number
long int	4	-2147483648 to +2147483647	Large whole number
float	4	3.4x10-38 to 3.4x10+38	Small real number
double	8	1.7x10-308 to 1.7x10+308	Large real number
long double	10	3.4x10-4932 to 3.4x10+4932	Very Large real number
char	1	0 to 255	A Single Character

Derived data types:

From the fundamental types other types can be derived by using the declaration operators. These are like: arrays, function , pointer, reference, const etc. **1. Arrays:** arrays refer to a named list of a finite number of n of similar data elements. Each of the data elements can be referenced respectively by a set of consecutive numbers, usually 0, 1, 2, 3, 4…n.

Like: ary[0], ary[1],ary[2]….. ary[n].

Array can be one two or multi dimensional.

2. Function: a function is a named part of a program that can be invoked from other parts of the program as often needed.

3. Pointer: A pointer is a variable that holds a memory address. This address is usually the location of another variable in memory. If one variable contains the address of another variable, the first variable is said to point to the second.

<Data type> *ptr;

ptr = &a;

4. Reference: A reference is an alternative name for an object. A reference variable provides an alias for a previously defined variable. A reference declaration consists of a base type, and & (ampersand). A reference variable name equated to a variable name (previously defined).

Syntax:
<Data type> & <ref-var> = <var-name>;

Example:
```
int total;
int & sum = total;
total =100;
cout<<"sum="<<sum<<"\n";
cout<<"Total="<<total<<"\n";
```

5. Constant: the keyword const can be added to the declaration of an object to make that object a constant rather than a variable. Thus, the value of the named constant can't be altered during the program run.

Syntax:
const <data type> <variable name> = <constant value>;

User defined data types:
There are some derived data types that are defined by the user. These are: **class, structure, union, and enumeration**.
1. Class: a class represents a group of similar objects. To represent classes in C++, it offers user defined data type called class. Once a class has been defined in C++, objects belonging to that class can easily be created. A class bears the same relationship to an object as a type bears to a variable.

Example:
```
Class student
{
int rollno;
char name[20];
public:
```

```
void print();
void read();
};
```

The class describes all the properties of a data type and an object is an entity created according to that description.

2. Structure: a structure is a collection of variables (of different data types) referenced under one name, providing of convenient means of keeping related information together. The keyword struct is used to construct a structure.

Example:

```
struct sturec
{
int rollno;
char name[20];
float marks;
};
```

3. Union: A union is a memory location that is shared by two or more different variables, generally of different types at different times. Defining a union is similar to defining a structure.

```
union share
{
int i;
char ch;
};
union share u1;
```

The keyword union is used for declaring and creating a union. In the union u1 both integer i and character ch share the same memory location.

4. Enumeration. An alternative method for naming integer constants is often more convenient than const. this can be achieved by creating enumeration using keyword **enum**.

```
enum { START, PAUSE, GOO};
```

This defines three integer constants called enumerators and assigns values to them. Enumerator values are by default assigned increasing from 0 therefore we can write above statement like:

```
const int START =0;
```

const int PAUSE = 1;
const int GOO = 2;
an enumeration can be named also.
enum status {START, PAUSE, GOO};

Variables
It is a location in the computer memory which can store data and is given a symbolic name for easy reference. The variables can be used to hold different values at different times during the execution of a program. To understand more clearly we should study the following statements:

Total = 20.00; //In this statement a value 20.00 has been stored in a memory location Total.

Declaration of a variable
Before a variable is used in a program, we must declare it. This activity enables the compiler to make available the appropriate type of location in the memory.
float Total;
You can declare more than one variable of same type in a single statement
int x,y;

5. INITIALIZATION OF VARIABLE
When we declare a variable it's default value is undetermined. We can declare a variable with some initial value.
int a = 20;

Input/Output (I/O)
C++ supports input/output statements which can be used to feed new data into the computer or obtain output on an output device such as: VDU, printer etc. The following C++ stream objects can be used for the input/output purpose.
cin console input
cout console output
cout is used in conjuction with << operator, known as insertion or put to operator.
cin is used in conjuction with >> operator, known as extraction or get from operator.
cout << “My first computer";Once the above statement is carried out by the computer, the message "My first computer" will appear on the screen.
cin can be used to input a value entered by the user from the keyboard. However, the get from operator>> is also required to get the typed value from cin and store it

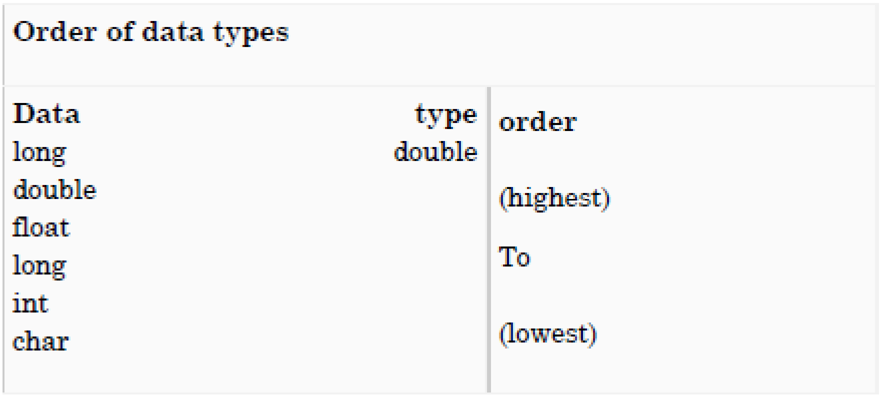
in the memory location.
Let us consider the following program segment:
int marks;
cin >> marks; In the above segment, the user has defined a variable marks of integer type in the first statement and in the second statement he is trying to read a value from the keyboard.

6. TYPE CONVERSION

The process in which one pre-defined type of expression is converted into another type is called conversion. There are two types of conversion in C++.

1. Implicit conversion
2. Explicit conversion

1. Implicit conversion
Data type can be mixed in the expression. For example
double a;
int b = 5;
float c = 8.5;
a = b * c;
When two operands of different type are encountered in the same expression, the lower type variable is converted to the higher type variable. The following table shows the order of data types.



Order of data types

Data type	**order**
long double	
double	(highest)
float	
long	To
int	
char	(lowest)

The int value of b is converted to type float and stored in a temporary variable before being multiplied by the float variable c. The result is then converted to double so that it can be assigned to the double variable a.

2. Explicit conversion

It is also called type casting. It temporarily changes a variable data type from its declared data type to a new one. It may be noted here that type casting can only be done on the right hand side the assignment statement.

T_Pay = double (salary) + bonus;

Initially variable salary is defined as float but for the above calculation it is first converted to double data type and then added to the variable bonus.

Constants

A number which does not change its value during execution of a program is known as a constant. Any attempt to change the value of a constant will result in an error message. A constant in C++ can be of any of the basic data types, const qualifier can be used to declare constant as shown below:

const float pi = 3.1415;The above declaration means that Pi is a constant of float types having a value 3.1415.

Examples of **valid constant** declarations are:

const int rate = 50;

const float pi = 3.1415;

const char ch = 'A';

7. STRUCTURE OF C++ PROGRAM

#include<header file>

main ()

{

....

....

....

}

A C++ program starts with function called main ( ). The body of the function is enclosed between curly braces. The program statements are written within the braces. Each statement must end by a semicolon ;(statement terminator). A C++ program may contain as many functions as required. However, when the program is loaded in the memory, the control is handed over to function main ( ) and it is the first function to be executed.

// This is my first program is C++

```
/* this program will illustrate different components of
a simple program in C++ */
# include <iostream.h>
int main ( )
{
cout <<"Hello World!";
return 0;
}
```

When the above program is compiled, linked and executed, the following output is displayed on the VDU screen.

```
Hello World!
```

Various components of this program are discussed below:

Comments

First three lines of the above program are comments and are ignored by the compiler. Comments are included in a program to make it more readable. If a comment is short and can be accommodated in a single line, then it is started with double slash sequence in the first line of the program. However, if there are multiple lines in a comment, it is enclosed between the two symbols /* and */

#include <iostream.h>

The line in the above program that start with # symbol are called directives and are instructions to the compiler. The word include with '#' tells the compiler to include the file iostream.h into the file of the above program. File iostream.h is a header file needed for input/ output requirements of the program. Therefore, this file has been included at the top of the program.

int main ( )

The word main is a function name. The brackets ( ) with main tells that main ( ) is a function. The word int before main ( ) indicates that integer value is being returned by the function main (). When program is loaded in the memory, the control is handed over to function main ( ) and it is the first function to be executed.

Curly bracket and body of the function main ( )

A C++ program starts with function called main(). The body of the function is enclosed between curly braces. The program statements are written within the brackets. Each statement must end by a semicolon, without which an error message in generated.

cout<<"Hello World!";

This statement prints our "Hello World!" message on the screen. cout understands that anything sent to it via the << operator should be printed on the screen.

return 0;

This is a new type of statement, called a return statement. When a program finishes running, it sends a value to the operating system. This particular return statement returns the value of 0 to the operating system, which means “everything went okay!”

```
/* This program illustrates how to declare variable, read data and display data. */
#include <iostream.h>
int main()
{
int rollno; //declare the variable rollno of type int
float marks; //declare the variable marks of type float
cout << "Enter roll number and marks :";
cin >> rollno >> marks; //store data into variable rollno & marks
cout << "Rollno: " << rollno<<"\n";
cout << "Marks: " << marks;
return 0;
}
```

OUTPUT:

```
Enter roll number and marks: 102 87.5
Rollno: 102
Marks: 87.5
```

8. REFERENCE VARIABLE

A reference is an alternative name for an object. A reference variable provides an alias for a previously defined variable. A reference declaration consists of a base type, and & (ampersand). A reference variable name equated to a variable name (previously defined).

Syntax:

<Data type> & <ref-var> = <var-name>;

Example:

```
int total;
int & sum = total;
total =100;
cout<<”sum=”<<sum<<”\n”;
cout<<”Total=”<<total<<”\n”;
```

9. FUNCTION

A function is a subprogram that acts on data and often returns a value. A program written with numerous functions is easier to maintain, update and debug than one very long program. By programming in a modular (functional) fashion, several programmers can work independently on separate functions which can be assembled at a later date to create the entire project. Each function has its own name. When that name is encountered in a program, the execution of the program branches to the body of that function. When the function is finished, execution returns to the area of the program code from which it was called, and the program continues on to the next line of code.

Creating User-Defined Functions

Declare the function.

The declaration, called the **FUNCTION PROTOTYPE**, informs the compiler about the functions to be used in a program, the argument they take and the type of value they return.

Define the function.

The function definition tells the compiler what task the function will be performing. The function prototype and the function definition must be same on the return type, the name, and the parameters. The only difference between the function prototype and the function header is a semicolon.

The function definition consists of the function header and its body. The header is EXACTLY like the function prototype, EXCEPT that it contains NO terminating semicolon.

//Prototyping, defining and calling a function

```
#include <iostream.h>
void starline(); // prototype the function
int main()
{
starline( ); // function call
cout<< "\t\tBjarne Stroustrup\n";
starline( ); // function call
return 0;
}
```

// function definition

```
void starline()
{
int count; // declaring a LOCAL variable
for(count = 1; count <=65; count++)
cout<< "*";
cout<<endl;
}
```

ARGUMENT TO A FUNCTION

Sometimes the calling function supplies some values to the called function. These are known as parameters. The variables which supply the values to a calling function called **actual parameters**. The variable which receive the value from called statement are termed **formal parameters**.

Consider the following example that evaluates the area of a circle.

```
#include<iostream.h>
void area(float);
int main()
{
float radius;
cin>>radius;
area(radius);
return 0;
}
void area(float r)
{
cout<< “the area of the circle is”<<3.14*r*r<<”\n”;
}
```

Here radius is called actual parameter and r is called formal parameter.

RETURN TYPE OF A FUNCTION

```
// Example program
#include <iostream.h>
int timesTwo(int num); // function prototype
int main()
{
int number, response;
cout<<"Please enter a number:";
cin>>number;
response = timesTwo(number); //function call
cout<< "The answer is "<<response;
return 0;
}
//timesTwo function
int timesTwo (int num)
{
int answer; //local variable
answer = 2 * num;
return (answer);
}
```

CALLING OF FUNCTION

The function can be called using either of the following methods:

i) Call by value

ii) Call by reference

CALL BY VALUE

In call by value method, the called function creates its own copies of original values sent to it. Any changes, that are made, occur on the function's copy of values and are not reflected back to the calling function.

CALL BY REFERENCE

In call be reference method, the called function accesses and works with the original values using their references. Any changes, that occur, take place on the original values are reflected back to the calling code.

Consider the following program which will swap the value of two variables.

using call by reference using call by value

```
#include<iostream.h>
void swap(int &, int &);
int main()
{
int a=10,b=20;
swap(a,b);
cout<<a<<" "<<b;
return 0;
}
void swap(int &c, int &d)
{
int t;
t=c;
c=d;
d=t;
}
#include<iostream.h>
void swap(int , int );
int main()
{
int a=10,b=20;
swap(a,b);
cout<<a<<" "<< b;
return 0;
}
void swap(int c, int d)
{
int t;
t=c;
c=d;
d=t;
}
```

output:

```
20 10
```

output:

```
10 20
```

Function With Default Arguments

C++ allows to call a function without specifying all its arguments. In such cases, the function assigns a default value to a parameter which does not have a matching arguments in the function call. Default values are specified when the function is declared. The complier knows from the prototype how many arguments a function uses for calling.

Example : float result(int marks1, int marks2, int marks3=75);a subsequent function call

average = result(60,70);passes the value 60 to marks1, 70 to marks2 and lets the function use default value of 75 for marks3.

The function callaverage = result(60,70,80);passes the value 80 to marks3.

Global Variable And Local Variable

Local Variable : a variable declared within the body of a function will be evaluated only within the function. The portion of the program in which a variable is retained in memory is known as the **scope of the variable**. The scope of the local variable is

a function where it is defined. A variable may be local to function or compound statement.
Global Variable : a variable that is declared outside any function is known as a global variable. The scope of such a variable extends till the end of the program. these variables are available to all functions which follow their declaration. So it should be defined at the beginning, before any function is defined.

Variables and storage Class
The storage class of a variable determines which parts of a program can access it and how long it stays in existence. The storage class can be classified as automatic register static external.
Automatic variable
All variables by default are auto i.e. the declarations int a and auto int a are equivalent. Auto variables retain their scope till the end of the function in which they are defined. An automatic variable is not created until the function in which it defined is called. When the function exits and control is returned to the calling program, the variables are destroyed and their values are lost. The name automatic is used because the variables are automatically created when a function is called and automatically destroyed when it returns.
Register variable
A register declaration is an auto declaration. A register variable has all the characteristics of an auto variable. The difference is that register variable provides fast access as they are stored inside CPU registers rather than in memory.
Static variable
A static variable has the visibility of a local variable but the lifetime of an external variable. Thus it is visible only inside the function in which it is defined, but it remains in existence for the life of the program.
External variable
A large program may be written by a number of persons in different files. A variable declared global in one file will not be available to a function in another file. Such a variable, if required by functions in both the files, should be declared global in one file and at the same time declared external in the second file.

10. FUNCTION OVERLOADING
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 had been declared with the same name as a previously declared declaration in the same

scope, except that both declarations have different arguments and obviously different definition (implementation).

When you call an overloaded **function** or **operator**, the compiler determines the most appropriate definition to use by comparing the argument types you 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**.

You can have multiple definitions for the same function name in the same scope.

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

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

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

11. CLASSES AND OBJECT

The mechanism that allows you to combine data and the function in a single unit is called a class. Once a class is defined, you can declare variables of that type. A class variable is called object or instance. In other words, a class would be the data type, and an object would be the variable. Classes are generally declared using the keyword class, with the following format:

```
class class_name
{
private:
members1;
protected:
members2;
public:
members3;
};
```

Where class_name is a valid identifier for the class. The body of the declaration can contain members, that can be either data or function declarations, The members of a class are classified into three categories: private, public, and protected. Private, protected, and public are reserved words and are called member access specifiers. These specifiers modify the access rights that the members following them acquire. **private members** of a class are accessible only from within other members of the same class. You cannot access it outside of the class. **protected members** are accessible from members of their same class and also from members of their derived classes.

Finally, **public members** are accessible from anywhere where the object is visible. By default, all members of a class declared with the class keyword have private access for all its members. Therefore, any member that is declared before one other class specifier automatically has private access.

Here is a complete example :class student

```
{
private :
int rollno;
float marks;
public:
void getdata()
{
```

```
cout<<"Enter Roll Number : ";
cin>>rollno;
cout<<"Enter Marks : ";
cin>>marks;
}
void displaydata()
{
cout<<"Roll number : "<<rollno<<"\nMarks : "<<marks;
}
};
```

Object Declaration

Once a class is defined, you can declare objects of that type. The syntax for declaring a object is the same as that for declaring any other variable. The following statements declare two objects of type student:student st1, st2;

Accessing Class Members

Once an object of a class is declared, it can access the public members of the class.st1.getdata();

Defining Member function of class

You can define Functions inside the class as shown in above example. Member functions defined inside a class this way are created as inline functions by default.
It is also possible to declare a function within a class but define it elsewhere.
Functions defined outside the class are not normally inline.
When we define a function outside the class we cannot reference them (directly) outside of the class. In order to reference these, we use the scope resolution operator, :: (double colon).
In this example, we are defining function getdata outside the class:void student :: getdata()

```
{
cout<<"Enter Roll Number : ";
cin>>rollno;
cout<<"Enter Marks : ";
cin>>marks;
}
```

The following program demostrates the general feature of classes. Member function initdata() is defined inside the class. Member funcitons getdata() and showdata() defined outside the class.

```
class student //specify a class
```

```
{
private :
int rollno; //class data members
float marks;
public:
void initdata(int r, int m)
{
rollno=r;
marks=m;
}
void getdata(); //member function to get data from user
void showdata();// member function to show data
};
void student :: getdata()
{
cout<<"Enter Roll Number : ";
cin>>rollno;
cout<<"Enter Marks : ";
cin>>marks;
}
void student :: showdata()
{
cout<<"Roll number : "<<rollno<<"\nMarks : "<<marks;
}
int main()
{
student st1, st2; //define two objects of class student
st1.initdata(5,78); //call member function to initialize
st1.showdata();
st2.getdata(); //call member function to input data
st2.showdata(); //call member function to display data
return 0;
}
```

12. FRIEND FUNCTION

Friend functions allow you to access the private and protected members of a class from outside the class. To access the private and protected members of a class from outside the class, you can declare functions as a friend in the class. You can create object of the class in the friend function to access the private and protected members of the class.
Consider that a class has three private data members: x, y and z. You declare the void add() function as a friend in the calculation class by applying the friend keyword in front of that function. You can define the void add 0 friend function outside the calculation class to access the private and protected data members of the class. To access the private and protected members of the class, you can create an object of the class. The object of the class helps you to calculate the sum using the private data members of the class. You can call the friend function from the main() function. The following program shows how to define friend function in a class:

```
# #include<iostream.h>
#include<conio.h>
class integer
{
int a,b;
public:
friend int enter_val(); void print_val()
cout«"Value of n1 is "«a«endl; cout«"Value of n2 is "«b;
};
int enter_val()
{
integer n;
cout«”enter value for a and b:"; cin»n.a»n.b;
return n;
}
void main()
{
clrscr();
integer n1=enter_val(); nl.print_val();
getch();
}
```

In the above program, class integer contains the enter_val function as friend function. This friend function is used to access the variables, a and b of integer type that are declared in the class integer.

13. INLINE FUNCTION

What is Inline Function?

Inline functions are functions where the call is made to inline functions. The actual code then gets placed in the calling program.

Reason for the need of Inline Function:

Normally, a function call transfers the control from the calling program to the function and after the execution of the program returns the control back to the calling program after the function call. These concepts of function save program space and memory space and are used because the function is stored only in one place and is only executed when it is called. This execution may be time consuming since the registers and other processes must be saved before the function gets called.

The extra time needed and the process of saving is valid for larger functions. If the function is short, the programmer may wish to place the code of the function in the calling program in order for it to be executed. This type of function is best handled by the inline function. In this situation, the programmer may be wondering "why not write the short code repeatedly inside the program wherever needed instead of going for inline function?". Although this could accomplish the task, the problem lies in the loss of clarity of the program. If the programmer repeats the same code many times, there will be a loss of clarity in the program. The alternative approach is to allow inline functions to achieve the same purpose, with the concept of functions.

What happens when an inline function is written?

The inline function takes the format as a normal function but when it is compiled it is compiled as inline code. The function is placed separately as inline function, thus adding readability to the source program. When the program is compiled, the code present in function body is replaced in the place of function call.

General Format of inline Function:

The general format of inline function is as follows:

inline data type function_name(arguments)

The keyword **inline** specified in the above example, designates the function as inline function. For example, if a programmer wishes to have a function named demo with return value as integer and with no arguments as inline it is written as follows:

inline int demo( )

Advantages:-
1) It does not require function calling overhead.
2) It also save overhead of variables push/pop on the stack, while function calling.
3) It also save overhead of return call from a function.
4) It increases locality of reference by utilizing instruction cache.
5) After in-lining compiler can also apply intraprocedural optmization if specified. This is the most important one, in this way compiler can now focus on dead code elimination, can give more stress on branch prediction, induction variable elimination.

Disadvantages:-
1) May increase function size so that it may not fit on the cache, causing lots of cache miss.
2) After in-lining function if variables number which are going to use register increases than they may create overhead on register variable resource utilization.
3) It may cause compilation overhead as if somebody changes code inside inline function than all calling location will also be compiled.
4) If used in header file, it will make your header file size large and may also make it unreadable.
5) If somebody used too many inline function resultant in a larger code size than it may cause thrashing in memory. More and more number of page fault bringing down your program performance.
6) It's not useful for embedded system where large binary size is not preferred at all due to memory size constraints.

14. DEFAULT ARGUMENT

A default argument is a value provided in a function declaration that is automatically assigned by the compiler if the caller of the function doesn't provide a value for the argument with a default value.
Following is a simple C++ example to demonstrate the use of default arguments. We don't have to write 3 sum functions, only one function works by using default values for 3rd and 4th arguments.

```
// A function with default arguments, it can be called with
// 2 arguments or 3 arguments or 4 arguments.
int sum(int x, int y, int z=0, int w=0)
{
   return (x + y + z + w);
}

/* Driver program to test above function*/
int main()
{
   cout << sum(10, 15) << endl;
   cout << sum(10, 15, 25) << endl;
   cout << sum(10, 15, 25, 30) << endl;
   return 0;
}
```

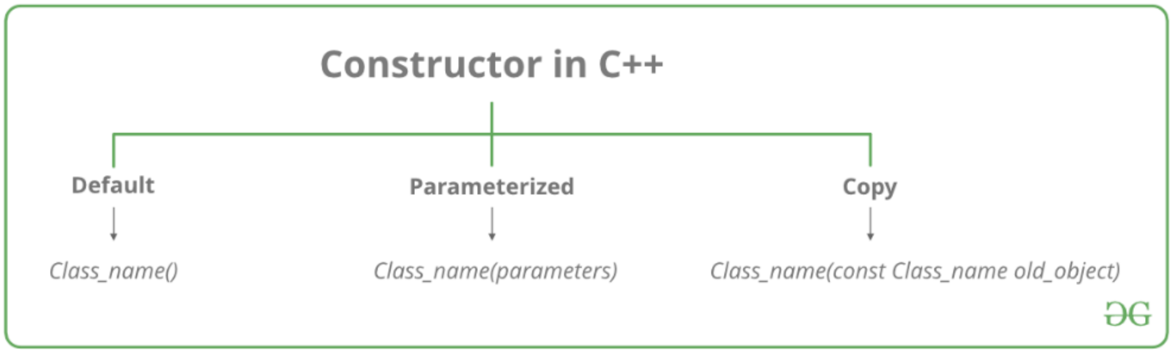
- Default arguments are different from constant arguments as constant arguments can't be changed whereas default arguments can be overwritten if required.
- Default arguments are overwritten when calling function provides values for them. For example, calling of function sum(10, 15, 25, 30) overwrites the value of z and w to 25 and 30 respectively.
- During calling of function, arguments from calling function to called function are copied from left to right. Therefore, sum(10, 15, 25) will assign 10, 15 and 25 to x, y, and z. Therefore, the default value is used for w only.
- Once default value is used for an argument in function definition, all subsequent arguments to it must have default value. It can also be stated as default arguments are assigned from right to left. For example, the following function definition is invalid as subsequent argument of default variable z is not default.

```
// Invalid because z has default value, but w after it
// doesn't have default value
int sum(int x, int y, int z=0, int w)
```

15. CONSTRUCTOR AND DESTRUCTOR

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.
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).



Types of Constructors

1. **Default Constructor:** Default constructor is the constructor which doesn't take any argument. It has no parameters.

```
class construct {
public:
   int a, b;

   // Default Constructor
   construct()
   {
      a = 10;
      b = 20;
   }
};
```

```
int main()
{
   // Default constructor called automatically
   // when the object is created
   construct c;
   cout << "a: " << c.a << endl
       << "b: " << c.b;
   return 1;
}
```

Output:

```
a: 10
b: 20
```

Note: Even if we do not define any constructor explicitly, the compiler will automatically provide a default constructor implicitly.

2. **Parameterized Constructors:** It is possible to pass arguments to constructors. Typically, these arguments help initialize an object when it is created. To create a parameterized constructor, simply add parameters to it the way you would to any other function. When you define the constructor's body, use the parameters to initialize the object.

```
class Point {
private:
   int x, y;

public:
   // Parameterized Constructor
   Point(int x1, int y1)
   {
      x = x1;
      y = y1;
   }

   int getX()
   {
      return x;
   }
   int getY()
   {
        return y;
    }
};

int main()
{
    // Constructor called
    Point p1(10, 15);

    // Access values assigned by constructor
    cout << "p1.x = " << p1.getX() << ", p1.y = " << p1.getY();

    return 0;
}
```

1. Output:

```
p1.x = 10, p1.y = 15
```

When an object is declared in a parameterized constructor, the initial values have to be passed as arguments to the constructor function. The normal way of object declaration may not work. The constructors can be called explicitly or implicitly.

```
Example e = Example(0, 50); // Explicit call

Example e(0, 50);          // Implicit call
```

Uses of Parameterized constructor:

1. It is used to initialize the various data elements of different objects with different values when they are created.
2. It is used to overload constructors.

Can we have more than one constructors in a class? Yes, It is called Constructor Overloading.

3. **Copy Constructor:** A copy constructor is a member function which initializes an object using another object of the same class. Detailed article on Copy Constructor.
   Whenever we define one or more non-default constructors(with parameters) for a class, a default constructor(without parameters) should also be explicitly defined as the compiler will not provide a default constructor in this case. However, it is not necessary but it's considered to be the best practice to always define a default constructor.

```
class point {
private:
  double x, y;

public:
  // Non-default Constructor & default Constructor
  point (double px, double py) {
    x = px, y = py;
  }
};

int main(void) {
  // Define an array of size 10 & of type point
  // This line will cause error
  point a[10];

  // Remove above line and program will compile without error
  point b = point(5, 6);
}
```

Output:

Error: point (double px, double py): expects 2 arguments, 0 provided

Destructors in C++

What is destructor?

Destructor is a member function which destructs or deletes an object.

When is destructor called?

A destructor function is called automatically when the object goes out of scope:

(1) The function ends

(2) The program ends

(3) A block containing local variables ends

(4) A delete operator is called

How destructors are different from a normal member function?

Destructors have same name as the class preceded by a tilde (~). Destructors don't take any argument and don't return anything.

```
class String
{
private:
   char *s;
   int size;
public:
   String(char *); // constructor
   ~String();      // destructor
};

String::String(char *c)
{
   size = strlen(c);
   s = new char[size+1];
   strcpy(s,c);
}

String::~String()
{
   delete []s;
}
```

16. OPERATOR OVERLOADING

Operator overloading is a very important feature of Object Oriented Programming. It is important because by using this facility we would be able to create new definitions of existing operators. In other words a single operator can take up several functions as desired by programmers depending on the argument taken by the operator by using the operator overloading facility.

We can overload operators like:

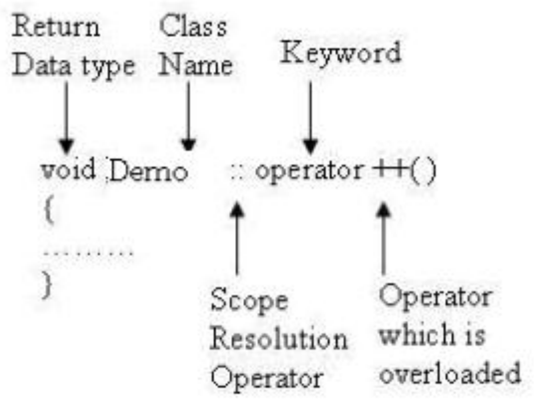
+	-	*	/	%	^	&	|	~
!	=	<	>	+=	-=	*=	/=	%=
^=	&=	|=	<<	>>	<<=	>>=	==	!=
<=	>=	&&	||	++	--	,	->*	->
( )	[ ]	new	delete	new[]	delete[]			

We can’t overload operators like:
Conditional operator (? :)
Scope resolution operator (::)
Class member access operator (.)
Pointer to member operator (. *)

Operator Overloading - Unary operators
As said before operator overloading helps the programmer to define a new functionality for the existing operator. This is done by using the keyword operator.

```
return_type classname :: operator operator_symbol(argument)
{
...
statements;
}
```

return_type - is the data type returned by the function
class name - is the name of the class
operator - is the keyword
operator symbol - is the symbol of the operator which is being overloaded or defined for new functionality
:: - is the scope resolution operator which is used to use the function definition outside the class.
Thus the above clearly specifies that operator overloading is defined as a member function by making use of the keyword operator.


Return Data type
Class Name
Keyword
void Demo :: operator ++( )
{
....
}
Scope Resolution Operator
Operator which is overloaded

Inside the class Demo the data type that is returned by the overloaded operator is defined as :

```
class Demo
{
private:
...
public:
void operator ++( );
...
};
```

Example:

```
#include <iostream>
class Demo
{
private:
int x;
public:
Demo ( ) { x=0; } //Constructor
void display();
void operator ++( );
};
void Demo :: display()
{
cout << "\nValue of x is: " << x;
}
void Demo :: operator ++( ) //Operator Overloading for operator ++ defined
{
```

```
++x;
}
void main( )
{
Demo d1,d2; //Object d1 and d2 created
cout << "\nBefore Increment";
cout << "\nObject d1: "; d1.display();
cout << "\nObject d2: "; d2.display();
++d1; //Operator overloading applied
++d2;
cout << "n After Increment";
cout << "\nObject d1: "; d1.display();
cout << "\nObject d2: "; d2.display();
}
```

Operator Overloading - Binary Operators

Binary operators, when overloaded, are given new functionality. The function defined for binary operator overloading, as with unary operator overloading, can be member function or friend function.

The difference is in the number of arguments used by the function. In the case of binary operator overloading, when the function is a member function then the number of arguments used by the operator member function is one. When the function defined for the binary operator overloading is a friend function, then it uses two arguments.

Binary operator overloading, as in unary operator overloading, is performed using a keyword operator.

Example:

```
#include <iostream>
class Demo
{
private:
int x;
int y;
public:
Demo() //Constructor
{ x=0; y=0; }
void getvalue( ) //Member Function for Inputting Values
{
cout << "n Enter value for x: ";
cin >> x;
cout << "n Enter value for y: ";
cin>> y;
}
void displayvalue( ) //Member Function for Outputting Values
{
cout << "\nvalue of x is: " << x << "; \nvalue of y is: " << y;
}
Demo operator +(Demo);
};
Demo Demo :: operator + (Demo d2)
//Binary operator overloading for + operator defined
{
Demo res; //declaring a Demo object to retain the final values
int x1 = x+ d2.x;
int y1 = y+d2.y;
res.x=x1;
res.y=y1;
return res;
}
void main( )
{
Demo d1,d2,d3; //Objects d1,d2,d3 created
cout << "\nEnter value for Object d1:";
d1.getvalue( );
cout << "\nEnter value for Object d2:";
d2.getvalue( );
d3= d1+ d2; //Binary Overloaded operator used
cout << "\nValue of d1 is: ";
d1.displayvalue();
cout << "\nValue of d2 is: " ;
d2.displayvalue();
cout << "\nValue of d3 is: ";
d3.displayvalue();
}
```

17. DYNAMIC INITIALIZATION OF VARIABLES

The process of initializing variable at the time of its declaration at run time is known as dynamic initialization of variable. Thus in dynamic initialization of variable a variable is assigned value at run time at the time of its declaration.

Example:

```
int main()
{
int a;
cout << "Enter Value of a";
cin >> a;
int cube = a * a * a;
}
```

In above example variable cube is initialized at run time using expression a * a * a at the time of its declaration.

18. OPERATORS

Operators are special symbols used for specific purposes. C++ provides six types of operators.

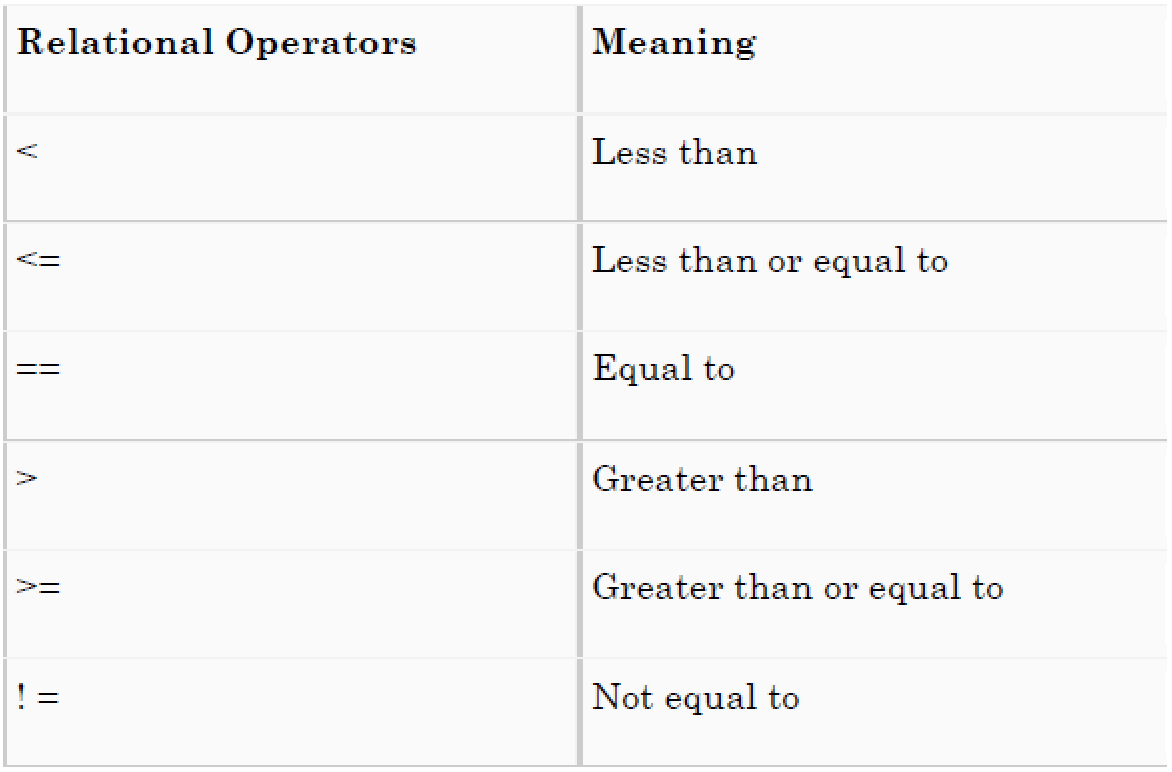
Arithmetical operators, Relational operators, Logical operators, Unary operators, Assignment operators, Conditional operators, Comma operator

Arithmetical operators

Arithmetical operators +, -, *, /, and % are used to performs an arithmetic (numeric) operation. You can use the operators +, -, *, and / with both integral and floating point data types. Modulus or remainder % operator is used only with the integral data type. Operators that have two operands are called binary operators.

Relational operators

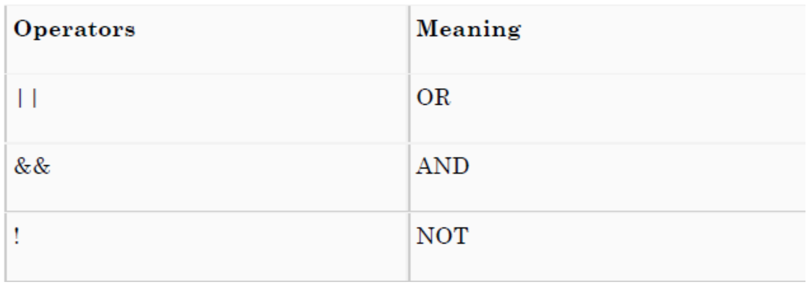
The relational operators are used to test the relation between two values. All relational operators are binary operators and therefore require two operands. A relational expression returns zero when the relation is false and a non-zero when it is true. The following table shows the relational operators.



Relational Operators	Meaning
<	Less than
<=	Less than or equal to
==	Equal to
>	Greater than
>=	Greater than or equal to
! =	Not equal to

Logical operators

The logical operators are used to combine one or more relational expression. The logical operators are



Operators	Meaning
| |	OR
&&	AND
!	NOT

Unary operators

C++ provides two unary operators for which only one variable is required.

For Example a = - 50;

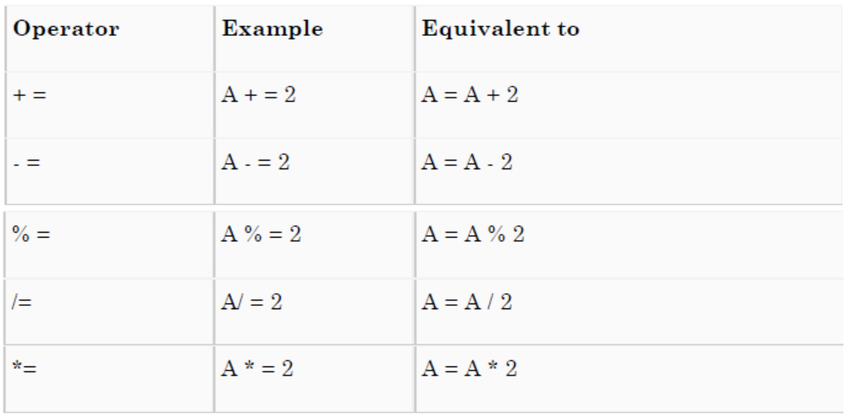
a = + 50; Here plus sign (+) and minus sign (-) are unary because they are not used between two variables.

Assignment operator

The assignment operator '=' is used for assigning a variable to a value. This operator takes the expression on its right-hand-side and places it into the variable on its left-hand-side. For example: m = 5; the operator takes the expression on the

right, 5, and stores it in the variable on the left, m.x = y = z = 32; this code stores the value 32 in each of the three variables x, y, and z.
In addition to standard assignment operator shown above, C++ also support compound assignment operators.

Compound Assignment Operators



Operator	Example	Equivalent to
+ =	A + = 2	A = A + 2
- =	A - = 2	A = A - 2
% =	A % = 2	A = A % 2
/=	A/ = 2	A = A / 2
*=	A * = 2	A = A * 2

Increment and Decrement Operators
C++ provides two special operators viz '++' and '--' for incrementing and decrementing the value of a variable by 1. The increment/decrement operator can be used with any type of variable but it cannot be used with any constant. Increment and decrement operators each have two forms, pre and post.

The syntax of the increment operator is:
Pre-increment: ++variable
Post-increment: variable++
The syntax of the decrement operator is:
Pre-decrement: —variable
Post-decrement: variable—
In Prefix form first variable is first incremented / decremented, then evaluated.
In Postfix form first variable is first evaluated, then incremented/decremented

```
int x,y;
int i=10,j=10;
x = ++i; //add one to i, store the result back in x
y= j++; //store the value of j to y then add one to j
cout<<x; //11
cout<<y; //10
```

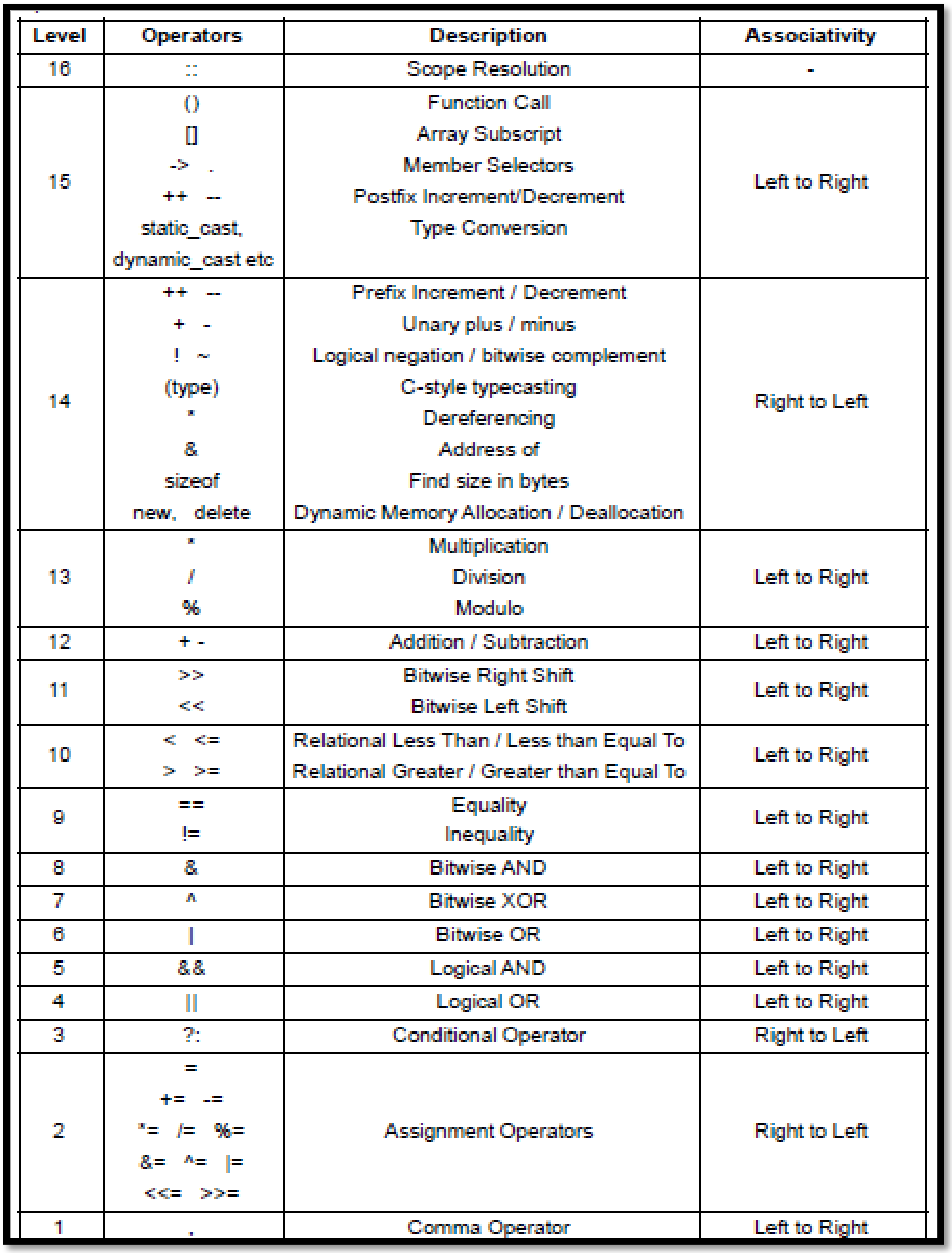
Conditional operator
The conditional operator **? :** is called ternary operator as it requires three operands. The format of the conditional operator is:

Conditional_ expression ? expression1 : expression2;

If the value of conditional expression is true then the expression1 is evaluated, otherwise expression2 is evaluated. int a = 5, b = 6;
big = (a > b) ? a : b; The condition evaluates to false, therefore big gets the value from b and it becomes 6.

The comma operator
The comma operator gives left to right evaluation of expressions. When the set of expressions has to be evaluated for a value, only the rightmost expression is considered.
int a=1, b=2, c=3, i; // comma acts as separator, not as an operator
i = (a, b); // stores b into I Would first assign the value of a to i, and then assign value of b to variable i. So, at the end, variable i would contain the value 2.
The sizeof() operator
As we know that different types of Variables, constant, etc. require different amounts of memory to store them The sizeof operator can be used to find how many bytes are required for an object to store in memory.
For example
sizeof (char) returns 1
sizeof (int) returns 2
sizeof (float) returns 4
If k is integer variable, the sizeof (k) returns 2.
The sizeof operator determines the amount of memory required for an object at compile time rather than at run time.

The order of Precedence
The order in which the Arithmetic operators (+,-,*,/,%) are used in a. given expression is called the order of precedence. The following table shows the order of precedence.



Level	Operators	Description	Associativity
16	::	Scope Resolution	-
15	() [] -> . ++ -- static_cast, dynamic_cast etc	Function Call Array Subscript Member Selectors Postfix Increment/Decrement Type Conversion	Left to Right
14	++ -- + - ! ~ (type) * & sizeof new, delete	Prefix Increment / Decrement Unary plus / minus Logical negation / bitwise complement C-style typecasting Dereferencing Address of Find size in bytes Dynamic Memory Allocation / Deallocation	Right to Left
13	* / %	Multiplication Division Modulo	Left to Right
12	+ -	Addition / Subtraction	Left to Right
11	>> <<	Bitwise Right Shift Bitwise Left Shift	Left to Right
10	< <= > >=	Relational Less Than / Less than Equal To Relational Greater / Greater than Equal To	Left to Right
9	== !=	Equality Inequality	Left to Right
8	&	Bitwise AND	Left to Right
7	^	Bitwise XOR	Left to Right
6	|	Bitwise OR	Left to Right
5	&&	Logical AND	Left to Right
4	||	Logical OR	Left to Right
3	?:	Conditional Operator	Right to Left
2	= += -= *= /= %= &= ^= |= <<= >>=	Assignment Operators	Right to Left
1	,	Comma Operator	Left to Right

Scope resolution operator: (::)

The scope resolution operator (denoted ::) in C++ is used to define the already declared member functions (in the header file with the .h extension) of a particular class. In the .cpp file one can define the usual global functions or the member functions of the class. To differentiate between the normal functions and the member functions of the class, one needs to use the scope resolution operator (::) in between the class name and the member function name i.e. vehicle::gear() where vehicle is a class and gear() is a member function of the class vehicle. The other uses of the resolution operator is to resolve the scope of a variable when the same identifier is used to represent a global variable, a local variable, and members of one or more class(es). If the resolution operator is placed between the class name and the data member belonging to the class then the data name belonging to the particular class is referenced. If the resolution operator is placed in front of the variable name then the global variable is referenced. When no resolution operator is placed then the local variable is referenced.

```
#include <iostream.h>
int n = 11; // A global variable
int main()
{
int n = 12; // A local variable
cout << ::n << '\n'; // Print the global variable: 11
cout << n << '\n'; // Print the local variable: 12
}
```

Memory management operators:

New and delete operator:

An object can be created by new and destroyed by using delete as and when required. A data object created inside a block with new will remain in existence until it is explicitly destroyed by using delete. Thus the lifetime of an object is directly under our control and is unrelated to the block structure of the program.

```
Pointer variable = new <data type>
P = new int;
Or
int *p = new int;
```

when object is no longer needed it is destroyed to release the memory space for reuse.

delete pointer variable;

What is manipulator?

Manipulators are operators used in C++ for formatting output. The data is manipulated by the programmer's choice of display. There are numerous manipulators available in C++. Some of the more commonly used manipulators are provided here below:

endl Manipulator:

This manipulator has the same functionality as the '\n' newline character
cout<<" bca 4th semester"<<endl;

setw() manipulator:

This manipulator sets the minimum field width on output. The syntax is
Setw(x)
Here setw causes the number or string that follows it to be printed within a field of x characters wide and x is the argument set in setw manipulator. The header file that must be included while using setw manipulator is.

19. CONTROL STRUCTURES

Statements

Statements are the instructions given to the computer to perform any kind of action. Action may be in the form of data movement, decision making etc. Statements form the smallest executable unit within a C++ program. Statements are always terminated by semicolon.

Compound Statement

A compound statement is a grouping of statements in which each individual statement ends with a semi-colon. The group of statements is called block.
Compound statements are enclosed between the pair of braces ({}.). The opening brace ({) signifies the beginning and closing brace (}) signifies the end of the block.

Null Statement

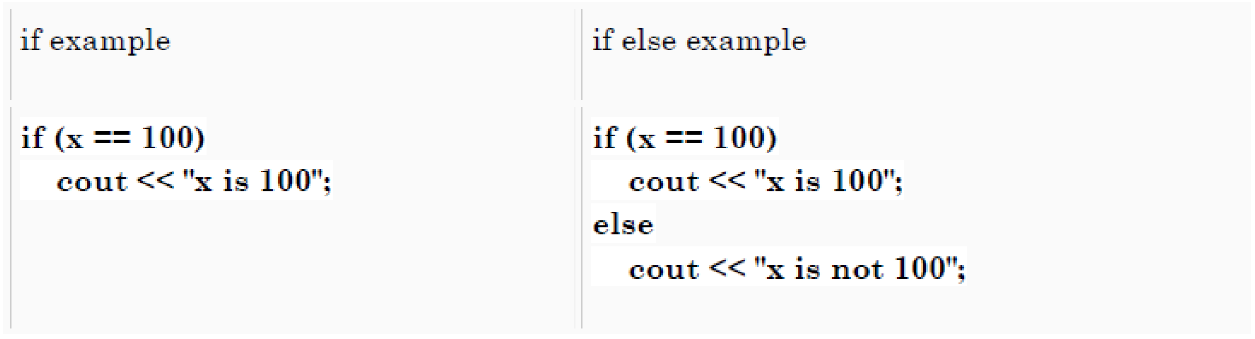
Writing only a semicolon indicates a null statement. Thus ';' is a null or empty statement. This is quite useful when the syntax of the language needs to specify a statement but the logic of the program does not need any statement. This statement is generally used in for and while looping statements.

Conditional Statements
Sometimes the program needs to be executed depending upon a particular condition. C++ provides the following statements for implementing the selection control structure.

- **if statement**
- **if else statement**
- **nested if statement**
- **switch statement**

if statement:
syntax of the if statement
if (condition)
{
statement(s);
}
From the flowchart it is clear that if the if condition is true, statement is executed; otherwise it is skipped. The statement may either be a single or compound statement.

if else statement:
syntax of the if - else statement
if (condition)
statement1;
else
statement2;
From the above flowchart it is clear that the given condition is evaluated first. If the condition is true, statement1 is executed. If the condition is false, statement2 is executed. It should be kept in mind that statement and statement2 can be single or compound statement.



if example	if else example

`if (x == 100)`
`   cout << "x is 100";`
	`if (x == 100)`
	`   cout << "x is 100";`
	`else`
	`   cout << "x is not 100";`

Nested if statement

The if block may be nested in another if or else block. This is called nesting of if or else block.

syntax of the nested if statement

```
if(condition 1)
{
if(condition 2)
{
statement(s);
}
}
```

```
if(condition 1)
statement 1;
else if (condition 2)
statement2;
else
statement3;
```

Example:

```
if(percentage>=60)
cout<<"Ist division";
else if(percentage>=50)
cout<<"IInd division";
else if(percentage>=40)
cout<<"IIIrd division";
else
cout<<"Fail" ;
```

switch statement:

The if and if-else statements permit two way branching whereas switch statement permits multiple branching. The syntax of switch statement is:

```
switch (var / expression)
{
case constant1 : statement 1;
break;
case constant2 : statement2;
break;
.
.
default: statement3;
break;
}
```

The execution of switch statement begins with the evaluation of expression. If the value of expression matches with the constant then the statements following this statement execute sequentially till it executes break. The break statement transfers control to the end of the switch statement. If the value of expression does not match with any constant, the statement with default is executed.

Some important points about switch statement-

The expression of switch statement must be of type integer or character type.

-The default case need not to be used at last case. It can be placed at any place.

-The case values need not to be in specific order.

Looping statement

It is also called a Repetitive control structure. Sometimes we require a set of statements to be executed a number of times by changing the value of one or more variables each time to obtain a different result. This type of program execution is called looping. C++ provides the following construct

1. while loop

2. do-while loop

3. for loop

1. While loop:

Syntax of while loop

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

The flow diagram indicates that a condition is first evaluated. If the condition is true, the loop body is executed and the condition is re-evaluated. Hence, the loop body is executed repeatedly as long as the condition remains true. As soon as the condition becomes false, it comes out of the loop and goes to the statement next to the 'while' loop.

2. do-while loop:

Syntax of do-while loop

```
do
{
statements;
} while (condition);
```

Note : That the loop body is always executed at least once. One important difference between the while loop and the do-while loop the relative ordering of the conditional test and loop body execution. In the while loop, the loop repetition test is performed before each execution the loop body; the loop body is not executed at all if the initial test fail. In the do-while loop, the loop termination test is Performed after each execution of the loop body. hence, the loop body is always executed least once.

3. for loop:

It is a count controlled loop in the sense that the program knows in advance how many times the loop is to be executed.

syntax of for loop :

```
for (initialization; decision; increment/decrement)
{
statement(s);
}
```

The flow diagram indicates that in for loop three operations take place:

- **Initialization of loop control variable**
- **Testing of loop control variable**
- **Update the loop control variable either by incrementing or decrementing.**

Operation (i) is used to initialize the value. On the other hand, operation (ii) is used to test whether the condition is true or false. If the condition is true, the program executes the body of the loop and then the value of loop control variable is updated. Again it checks the condition and so on. If the condition is false, it gets out of the loop.

Jump Statements

The jump statements unconditionally transfer program control within a function.

1. goto statement

2. break statement

3. continue statement

4. exit

1. The goto statement
goto allows to make jump to another point in the program.goto pqr;
pqr:pqr is known as label. It is a user defined identifier. After the execution of goto statement, the control transfers to the line after label pqr.

2. The break statement
The break statement, when executed in a switch structure, provides an immediate exit from the switch structure. Similarly, you can use the break statement in any of the loop. When the break statement executes in a loop, it immediately exits from the loop.

3. The continue statement
The continue statement is used in loops and causes a program to skip the rest of the body of the loop.

```
while (condition)
{
Statement 1;
If (condition)
continue;
statement;
}
```

The continue statement skips rest of the loop body and starts a new iteration.

4. The exit ( ) function
The execution of a program can be stopped at any point with exit ( ) and a status code can be informed to the calling program. The general format is
exit (code) ;
where code is an integer value. The code has a value 0 for correct execution. The value of the code varies depending upon the operating system.

20. 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.

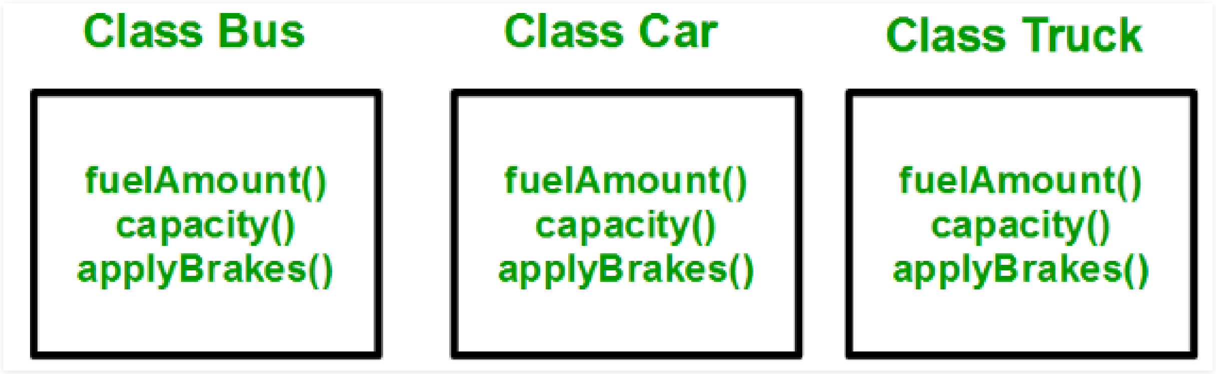
When creating a class, instead of writing completely new data members and member functions, the programmer can designate that the new class should inherit the members of an existing class. This existing class is called the **base class**, and the new class is referred to as the **derived class**.

Sub Class: The class that inherits properties from another class is called Sub class or Derived Class.

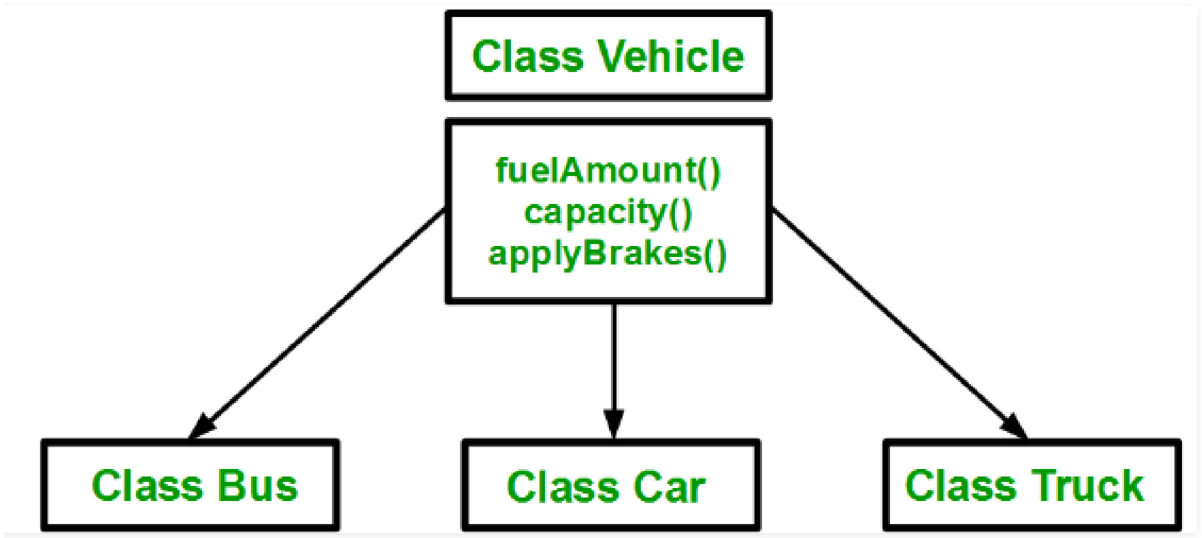
Super Class: The class whose properties are inherited by sub class is called Base Class or Super class.

Why and when to use inheritance?

Consider a group of vehicles. We need to create classes for Bus, Car and Truck. The methods fuelAmount(), capacity(), applyBrakes() will be same for all of the three classes. If we create these classes avoiding inheritance then we have to write all of these functions in each of the three classes as shown in below figure:



We can clearly see that above process results in duplication of same code 3 times. This increases the chances of error and data redundancy. To avoid this type of situation, inheritance is used. If we create a class Vehicle and write these three functions in it and inherit the rest of the classes from the vehicle class, then we can simply avoid the duplication of data and increase re-usability. Look at the below diagram in which the three classes are inherited from vehicle class:



Using inheritance, we have to write the functions only one time instead of three times as we have inherited rest of the three classes from base class(Vehicle).

Implementing inheritance in C++: For creating a sub-class which is inherited from the base class we have to follow the below syntax.
Syntax:

```
class subclass_name : access_mode base_class_name
{
 //body of subclass
};
```

Here, **subclass_name** is the name of the sub class, **access_mode** is the mode in which you want to inherit this sub class for example: public, private etc. and **base_class_name** is the name of the base class from which you want to inherit the sub class.

Note: A derived class doesn't inherit ***access*** to private data members. However, it does inherit a full parent object, which contains any private members which that class declares.

```
//Base class
class Parent
{
   public:
     int id_p;
};

// Sub class inheriting from Base Class(Parent)
class Child : public Parent
{
  public:
    int id_c;
};

//main function
int main()
  {

      Child obj1;

      // An object of class child has all data members
      // and member functions of class parent
      obj1.id_c = 7;
      obj1.id_p = 91;
      cout << "Child id is " <<  obj1.id_c << endl;
      cout << "Parent id is " <<  obj1.id_p << endl;

      return 0;
  }
```

Output:

```
Child id is 7
Parent id is 91
```

In the above program the ‘Child’ class is publicly inherited from the ‘Parent’ class so the public data members of the class ‘Parent’ will also be inherited by the class ‘Child’.

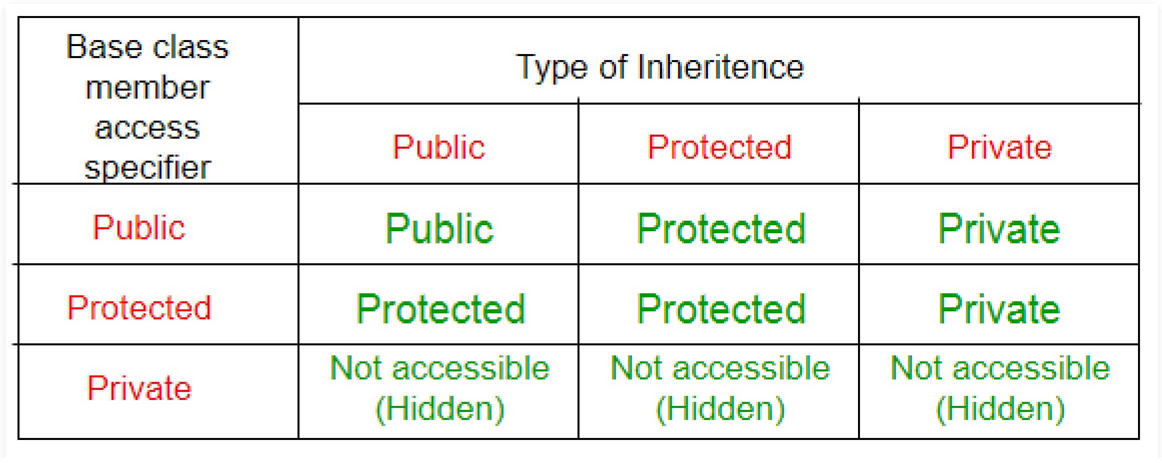
Modes of Inheritance

1. **Public mode**: If we derive a sub class from a public base class. Then the public member of the base class will become public in the derived class and protected members of the base class will become protected in derived class.
2. **Protected mode**: If we derive a sub class from a Protected base class. Then both public member and protected members of the base class will become protected in derived class.
3. **Private mode**: If we derive a sub class from a Private base class. Then both public member and protected members of the base class will become Private in derived class.

Note : The private members in the base class cannot be directly accessed in the derived class, while protected members can be directly accessed. For example, Classes B, C and D all contain the variables x, y and z in below example. It is just question of access.

```
class A
{
public:
   int x;
protected:
   int y;
private:
   int z;
};

class B : public A
{
   // x is public
   // y is protected
   // z is not accessible from B
};

class C : protected A
{
   // x is protected
   // y is protected
   // z is not accessible from C
};

class D : private A    // 'private' is default for classes
{
   // x is private
   // y is private
   // z is not accessible from D
};
```

The below table summarizes the above three modes and shows the access specifier of the members of base class in the sub class when derived in public, protected and private modes:



Base class member access specifier	Type of Inheritence		
	Public	Protected	Private
Public	Public	Protected	Private
Protected	Protected	Protected	Private
Private	Not accessible (Hidden)	Not accessible (Hidden)	Not accessible (Hidden)

Types of Inheritance in C++

1. Single inheritance

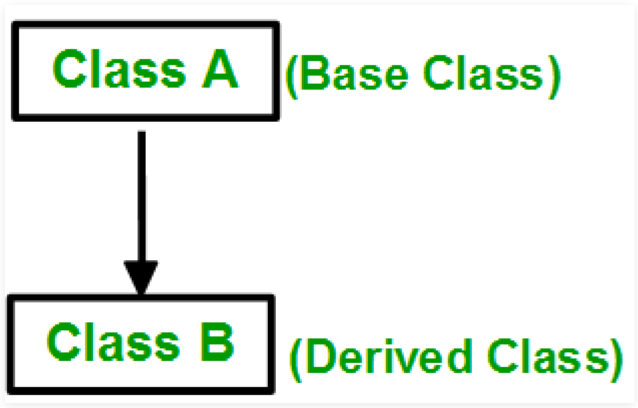
2. Multiple inheritance

3. Multilevel inheritance

4. Hierarchical inheritance

5. Hybrid (virtual) inheritance

1. Single Inheritance: In single inheritance, a class is allowed to inherit from only one class. i.e. one sub class is inherited by one base class only.



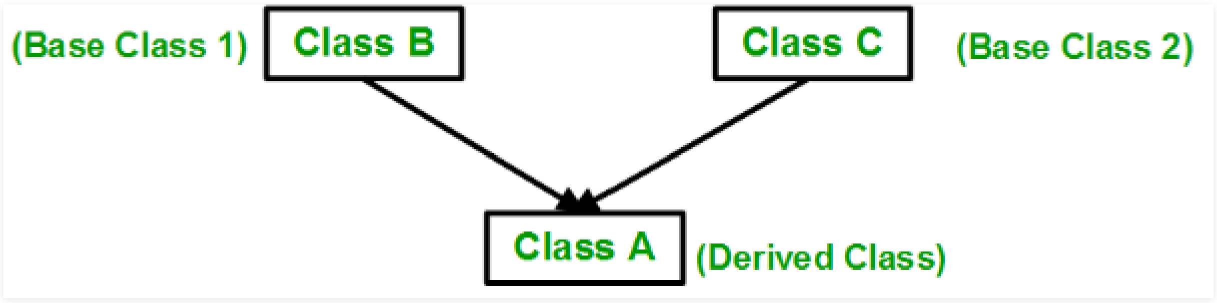
Syntax:

```
class subclass_name : access_mode base_class
{
  //body of subclass
};
class Vehicle {
  public:
    Vehicle()
    {
      cout << "This is a Vehicle" << endl;
    }
};

// sub class derived from two base classes
class Car: public Vehicle{

};

// main function
int main()
{
    // creating object of sub class will
    // invoke the constructor of base classes
    Car obj;
    return 0;
}
```

Output:

This is a vehicle

2. Multiple Inheritance: Multiple Inheritance is a feature of C++ where a class can inherit from more than one classes. i.e one **sub class** is inherited from more than one **base classes**.



.

Syntax:

```
class subclass_name : access_mode base_class1, access_mode base_class2, ....
{
  //body of subclass
};
```

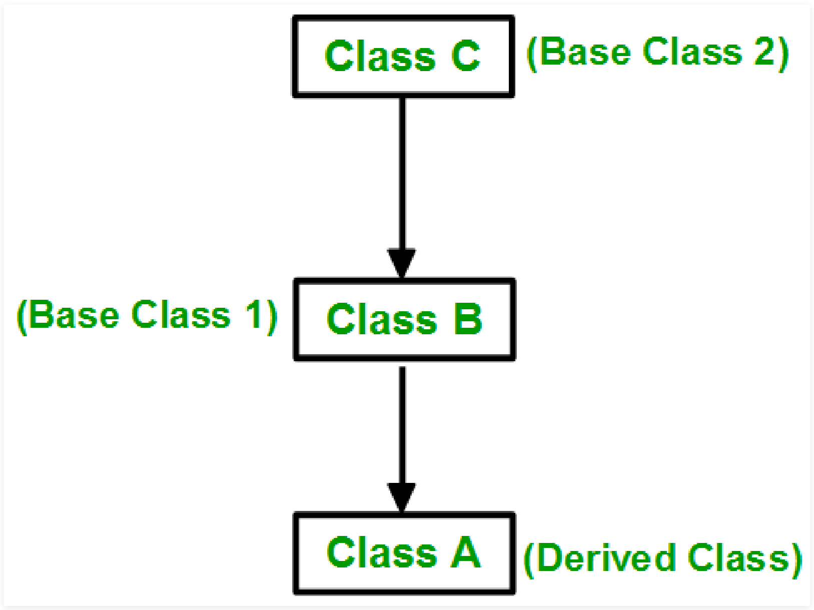
Here, the number of base classes will be separated by a comma (‘, ‘) and access mode for every base class must be specified.

```
class Vehicle {
  public:
    Vehicle()
    {
      cout << "This is a Vehicle" << endl;
    }
};

// second base class
class FourWheeler {
  public:
    FourWheeler()
    {
      cout << "This is a 4 wheeler Vehicle" << endl;
    }
};

// sub class derived from two base classes
class Car: public Vehicle, public FourWheeler {

};

// main function
int main()
{
    // creating object of sub class will
    // invoke the constructor of base classes
    Car obj;
    return 0;
}
```

Output:

```
This is a Vehicle
This is a 4 wheeler Vehicle
```

3. Multilevel Inheritance: In this type of inheritance, a derived class is created from another derived class.



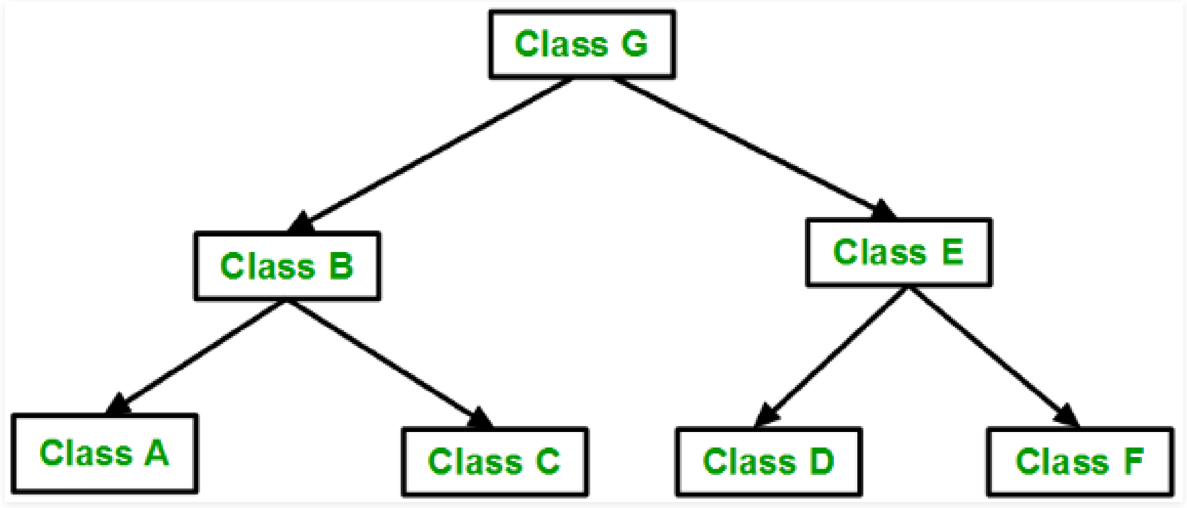
```
class Vehicle
{
  public:
   Vehicle()
   {
     cout << "This is a Vehicle" << endl;
   }
};
class fourWheeler: public Vehicle
{  public:
   fourWheeler()
   {
     cout<<"Objects with 4 wheels are vehicles"<<endl;
   }
};
// sub class derived from two base classes
class Car: public fourWheeler{
   public:
     car()
     {
       cout<<"Car has 4 Wheels"<<endl;
     }
};

// main function
```

```
int main()
{
   //creating object of sub class will
   //invoke the constructor of base classes
   Car obj;
   return 0;
}
```

Output:

```
This is a Vehicle
Objects with 4 wheels are vehicles
Car has 4 Wheels
```

4. Hierarchical Inheritance: In this type of inheritance, more than one sub class is inherited from a single base class. i.e. more than one derived class is created from a single base class.



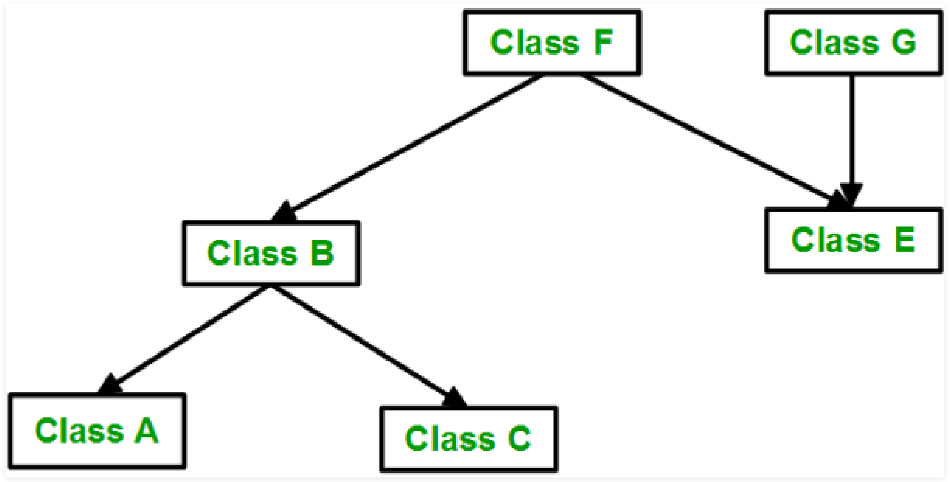
```
class Vehicle
{
  public:
   Vehicle()
   {
     cout << "This is a Vehicle" << endl;
   }
};


// first sub class
class Car: public Vehicle
{
};

// second sub class
class Bus: public Vehicle
{

};

// main function
int main()
{
   // creating object of sub class will
   // invoke the constructor of base class
   Car obj1;
   Bus obj2;
   return 0;
}
```

Output:

```
This is a Vehicle
This is a Vehicle
```

5. Hybrid (Virtual) Inheritance: Hybrid Inheritance is implemented by combining more than one type of inheritance. For example: Combining Hierarchical inheritance and Multiple Inheritance.

Below image shows the combination of hierarchical and multiple inheritance:



```
class Vehicle
{
  public:
    Vehicle()
    {
      cout << "This is a Vehicle" << endl;
    }
};

//base class
class Fare
{
    public:
    Fare()
    {
        cout<<"Fare of Vehicle\n";
    }
};

// first sub class
class Car: public Vehicle
{

};

// second sub class
class Bus: public Vehicle, public Fare
{

};

// main function
int main()
{
    // creating object of sub class will
    // invoke the constructor of base class
    Bus obj2;
    return 0;
}
```

Output:

```
This is a Vehicle
Fare of Vehicle
```

21. 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.

Implementation

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 –

```
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;
}
```

NULL pointer

The NULL pointer is a constant with a value of zero defined in several standard libraries, including iostream. Consider the following program −

```
int main ()
{
   int  *ptr = NULL;
   cout << "The value of ptr is " << ptr ;

   return 0;
}
```

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

```
The value of ptr is 0
```

On most of the operating systems, programs are not permitted to access memory at address 0 because that memory is reserved by the operating system. However, the memory address 0 has special significance; it signals that the pointer is not intended to point to an accessible memory location. But by convention, if a pointer contains the null (zero) value, it is assumed to point to nothing.

To check for a null pointer you can use an if statement as follows −

```
if(ptr)     // succeeds if p is not null
if(!ptr)    // succeeds if p is null
```

Thus, if all unused pointers are given the null value and you avoid the use of a null pointer, you can avoid the accidental misuse of an uninitialized pointer. Many times, uninitialized variables hold some junk values and it becomes difficult to debug the program.

Pointer Arithmetic

To understand pointer arithmetic, let us consider that **ptr** is an integer pointer which points to the address 1000. Assuming 32-bit integers, let us perform the following arithmetic operation on the pointer −

ptr++

the **ptr** will point to the location 1004 because each time ptr is incremented, it will point to the next integer. This operation will move the pointer to next memory location without impacting actual value at the memory location. If ptr points to a character whose address is 1000, then above operation will point to the location 1001 because next character will be available at 1001.

Incrementing a Pointer

We prefer using a pointer in our program instead of an array because the variable pointer can be incremented, unlike the array name which cannot be incremented because it is a constant pointer. The following program increments the variable pointer to access each succeeding element of the array −

```
const int MAX = 3;

int main ()
{
  int  var[MAX] = {10, 100, 200};
  int  *ptr;

  // let us have array address in pointer.
  ptr = var;

  for (int i = 0; i < MAX; i++)
{
    cout << "Address of var[" << i << "] = ";
    cout << ptr << endl;

    cout << "Value of var[" << i << "] = ";
    cout << *ptr << endl;

    // point to the next location
    ptr++;
  }
```

```
   return 0;
}
```

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

```
Address of var[0] = 0xbfa088b0
Value of var[0] = 10
Address of var[1] = 0xbfa088b4
Value of var[1] = 100
Address of var[2] = 0xbfa088b8
Value of var[2] = 200
```

Decrementing a Pointer

The same considerations apply to decrementing a pointer, which decreases its value by the number of bytes of its data type as shown below −

```
const int MAX = 3;

int main () {
   int  var[MAX] = {10, 100, 200};
   int  *ptr;

   // let us have address of the last element in pointer.
   ptr = &var[MAX-1];

   for (int i = MAX; i > 0; i--) {
      cout << "Address of var[" << i << "] = ";
      cout << ptr << endl;

      cout << "Value of var[" << i << "] = ";
      cout << *ptr << endl;

      // point to the previous location
      ptr--;
   }

   return 0;
}
```

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

```
Address of var[3] = 0xbfdb70f8
```

```
Value of var[3] = 200
Address of var[2] = 0xbfdb70f4
Value of var[2] = 100
Address of var[1] = 0xbfdb70f0
Value of var[1] = 10
```

Pointers v/s Array

Pointers and arrays are strongly related. In fact, pointers and arrays are interchangeable in many cases. For example, a pointer that points to the beginning of an array can access that array by using either pointer arithmetic or array-style indexing. Consider the following program –

```
const int MAX = 3;

int main () {
  int  var[MAX] = {10, 100, 200};
  int  *ptr;

  // let us have array address in pointer.
  ptr = var;

  for (int i = 0; i < MAX; i++) {
    cout << "Address of var[" << i << "] = ";
    cout << ptr << endl;

    cout << "Value of var[" << i << "] = ";
    cout << *ptr << endl;

    // point to the next location
    ptr++;
  }

  return 0;
}
```

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

```
Address of var[0] = 0xbfa088b0
Value of var[0] = 10
Address of var[1] = 0xbfa088b4
```

```
Value of var[1] = 100
Address of var[2] = 0xbfa088b8
Value of var[2] = 200
```

However, pointers and arrays are not completely interchangeable. For example, consider the following program −

```
#include <iostream.h>
 const int MAX = 3;
 int main ()
{
   int  var[MAX] = {10, 100, 200};

   for (int i = 0; i < MAX; i++) {
      *var = i;    // This is a correct syntax
      var++;       // This is incorrect.
   }

   return 0;
}
```

It is perfectly acceptable to apply the pointer operator * to var but it is illegal to modify var value. The reason for this is that var is a constant that points to the beginning of an array and can not be used as l-value.

Because an array name generates a pointer constant, it can still be used in pointer-style expressions, as long as it is not modified. For example, the following is a valid statement that assigns var[2] the value 500 −

*(var + 2) = 500;

Above statement is valid and will compile successfully because var is not changed.

Array of pointers

Before we understand the concept of array of pointers, let us consider the following example, which makes use of an array of 3 integers –

```
const int MAX = 3;

int main ()
{
   int  var[MAX] = {10, 100, 200};

   for (int i = 0; i < MAX; i++) {
     cout << "Value of var[" << i << "] = ";
     cout << var[i] << endl;
   }

   return 0;
}
```

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

```
Value of var[0] = 10
Value of var[1] = 100
Value of var[2] = 200
```

There may be a situation, when we want to maintain an array, which can store pointers to an int or char or any other data type available. Following is the declaration of an array of pointers to an integer −

```
int *ptr[MAX];
```

This declares **ptr** as an array of MAX integer pointers. Thus, each element in ptr, now holds a pointer to an int value. Following example makes use of three integers which will be stored in an array of pointers as follows −

```
const int MAX = 3;
int main () {
  int  var[MAX] = {10, 100, 200};
  int *ptr[MAX];

  for (int i = 0; i < MAX; i++) {
    ptr[i] = &var[i]; // assign the address of integer.
  }

  for (int i = 0; i < MAX; i++) {
    cout << "Value of var[" << i << "] = ";
    cout << *ptr[i] << endl;
  }

  return 0;
}
```

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

```
Value of var[0] = 10
Value of var[1] = 100
Value of var[2] = 200
```

You can also use an array of pointers to character to store a list of strings as follows −

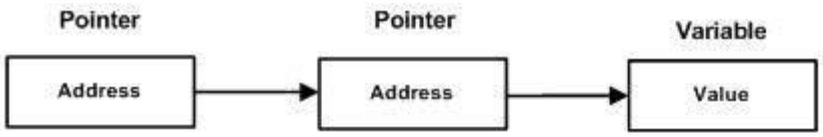
```
const int MAX = 4;
int main () {
const char *names[MAX] = { "Sumit", "Naveen", "Giriraj", "Shailendra" };

  for (int i = 0; i < MAX; i++) {
    cout << "Value of names[" << i << "] = ";
    cout << (names + i) << endl;
  }

  return 0;
}
```

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

```
Value of names[0] = 0x7ffd256683c0
Value of names[1] = 0x7ffd256683c8
Value of names[2] = 0x7ffd256683d0
Value of names[3] = 0x7ffd256683d8
```

Pointer to pointer

A pointer to a pointer is a form of multiple indirection or a chain of pointers. Normally, a pointer contains the address of a variable. When we define a pointer to a pointer, the first pointer contains the address of the second pointer, which points to the location that contains the actual value as shown below.



A variable that is a pointer to a pointer must be declared as such. This is done by placing an additional asterisk in front of its name. For example, following is the declaration to declare a pointer to a pointer of type int −

```
int **var;
```

When a target value is indirectly pointed to by a pointer to a pointer, accessing that value requires that the asterisk operator be applied twice, as is shown below in the example −

```
int main ()
{
  int  var;
  int  *ptr;
  int  **pptr;
  var = 3000;

  // take the address of var
  ptr = &var;

  // take the address of ptr using address of operator &
  pptr = &ptr;

  // take the value using pptr
  cout << "Value of var :" << var << endl;
  cout << "Value available at *ptr :" << *ptr << endl;
  cout << "Value available at **pptr :" << **pptr << endl;

  return 0;
}
```

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

```
Value of var :3000
Value available at *ptr :3000
Value available at **pptr :3000
```

Passing pointers to functions

C++ allows you to pass a pointer to a function. To do so, simply declare the function parameter as a pointer type.

An example that will swap two numbers i.e., interchange the values of two numbers.

```
void swap( int *a, int *b )
{
        int t;
        t = *a;
        *a = *b;
        *b = t;
}

int main(){
        int num1, num2;
        cout << "Enter first number" << endl;
        cin >> num1;
        cout << "Enter second number" << endl;
        cin >> num2;
        swap( &num1, &num2);
        cout << "First number = " << num1 << endl;
        cout << "Second number = " << num2 << endl;
        return 0;
}
```

Swapping means to interchange the values.
void swap(int *a, int *b) - It means our function 'swap' is taking two pointers as argument. So, while calling this function, we will have to pass the address of two integers (**call by reference**).

int t; t = *a; We took any integer t and gave it a value '*a'.

***a = *b** - Now, *a is *b. This means that now the values of *a and *b will be equal to that of *b.
***b = t;** - Since't' has an initial value of '*a', therefore, '*b' will also contain that initial value of '*a'. Thus, we have interchanged the values of the two variables.
Since we have done this swapping with pointers (we have targeted on address), so, this interchanged value will also reflect outside the function and the values of 'num1' and 'num2' will also get interchanged.
In the above example, we passed the address of the two variables (num1 and num2) to the swap function. The address of num1 is stored in 'a' pointer and that of num2 in 'b' pointer. In the swap function, we declared a third variable 't' and the values of 'a' and 'b' (and thus that of num1 and num2) gets swapped.
In the swapping example also, we used call by reference in which we passed the address of num1 and num2 as the arguments to the function. The function parameters 'a' and 'b' point to the address of num1 and num2 respectively. So, any change in the parameters 'a' and 'b' changes the value of num1 and num2 also.

22. VIRUTAL FUNCTION

A virtual function is a member function which is declared within a base class and is re-defined (Overriden) by a derived class. When you refer to a derived class object using a pointer or a reference to the base class, you can call a virtual function for that object and execute the derived class's version of the function.

- Virtual functions ensure that the correct function is called for an object, regardless of the type of reference (or pointer) used for function call.
- They are mainly used to achieve Runtime polymorphism

- Functions are declared with a **virtual** keyword in base class.
- The resolving of function call is done at Run-time.

Rules for Virtual Functions

1. Virtual functions cannot be static and also cannot be a friend function of another class.
2. Virtual functions should be accessed using pointer or reference of base class type to achieve run time polymorphism.
3. The prototype of virtual functions should be same in base as well as derived class.
4. They are always defined in base class and overridden in derived class. It is not mandatory for derived class to override (or re-define the virtual function), in that case base class version of function is used.
5. A class may have virtual destructor but it cannot have a virtual constructor.

```
class base
{
public:
   virtual void print()
   {
      cout << "print base class" << endl;
   }

   void show()
   {
      cout << "show base class" << endl;
   }
};

class derived : public base {
public:
   void print()
   {
      cout << "print derived class" << endl;
   }

   void show()
   {
      cout << "show derived class" << endl;
   }
};

int main()
{
   base* bptr;
   derived d;
   bptr = &d;

   // virtual function, binded at runtime
   bptr->print();

   // Non-virtual function, binded at compile time
   bptr->show();
}
```

Output:

```
print derived class
show base class
```

Explanation: Runtime polymorphism is achieved only through a pointer (or reference) of base class type. Also, a base class pointer can point to the objects of base class as well as to the objects of derived class. In above code, base class pointer 'bptr' contains the address of object 'd' of derived class.
Late binding(Runtime) is done in accordance with the content of pointer (i.e. location pointed to by pointer) and Early binding(Compile time) is done according to the type of pointer, since print() function is declared with virtual keyword so it will be bound at run-time (output is *print derived class* as pointer is pointing to object of derived class) and show() is non-virtual so it will be bound during compile time(output is *show base class* as pointer is of base type).

NOTE: If we have created a virtual function in the base class and it is being overridden in the derived class then we don't need virtual keyword in the derived class, functions are automatically considered as virtual functions in the derived class.

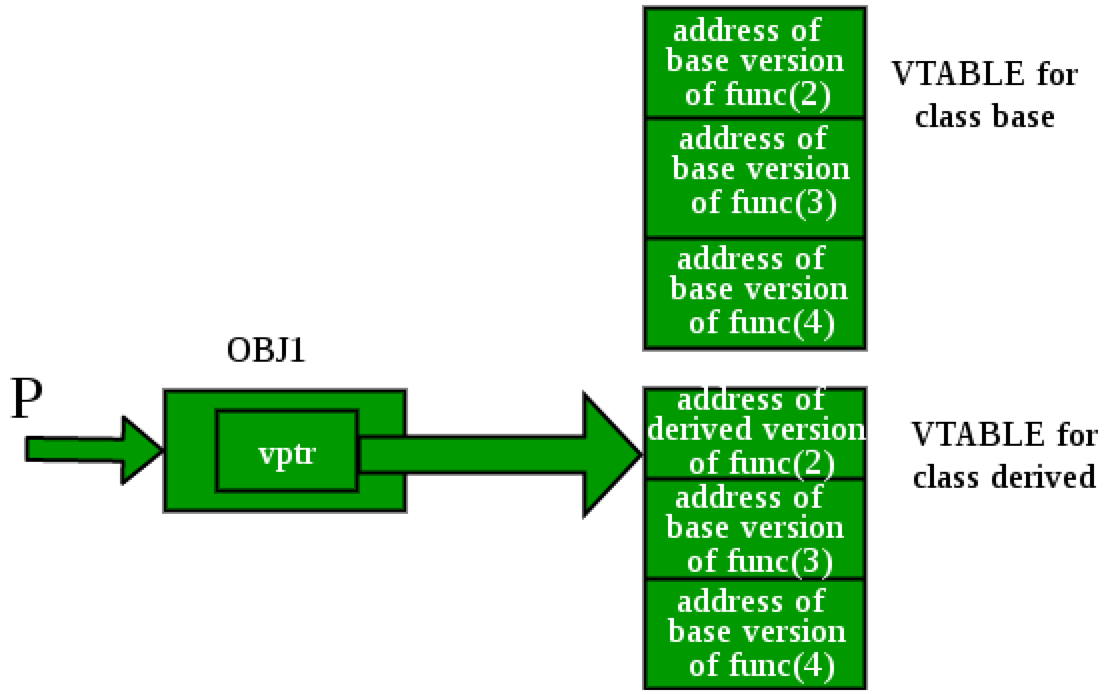
Working of virtual functions(concept of VTABLE and VPTR)

If a class contains a virtual function then compiler itself does two things:

1. If object of that class is created then a **virtual pointer(VPTR)** is inserted as a data member of the class to point to VTABLE of that class. For each new

object created, a new virtual pointer is inserted as a data member of that class.

2. Irrespective of object is created or not, **a static array of function pointer called VTABLE** where each cell contains the address of each virtual function contained in that class.


address of base version of func(2)
address of base version of func(3)
address of base version of func(4)
VTABLE for class base
OBJ1
P
vptr
address of derived version of func(2)
address of base version of func(3)
address of base version of func(4)
VTABLE for class derived

Abstract class and Pure virtual Function

The goal of object-oriented programming is to divide a complex problem into small sets. This helps understand and work with problem in an efficient way.

Sometimes, it's desirable to use inheritance just for the case of better visualization of the problem.

In C++, you can create an abstract class that cannot be instantiated (you cannot create object of that class). However, you can derive a class from it and instantiate object of the derived class.

Abstract classes are the base class which cannot be instantiated.

A class containing **pure virtual function** is known as **abstract class**.

Pure Virtual Function

A virtual function whose declaration ends with =0 is called a pure virtual function. For example,

```
class Weapon
{
   public:
     virtual void features() = 0;
};
```

Here, the pure virtual function is

virtual void features() = 0

And, the class Weapon is an abstract class.

```
class Shape
{
   protected:
      float l;
   public:
      void getData()
      {
         cin >> l;
      }

      // virtual Function
      virtual float calculateArea() = 0;
};

class Square : public Shape
{
   public:
      float calculateArea()
      {   return l*l;  }
};

class Circle : public Shape
{
   public:
      float calculateArea()
      { return 3.14*l*l; }
};
```

```
int main()
{
   Square s;
   Circle c;

   cout << "Enter length to calculate the area of a square: ";
   s.getData();
   cout<<"Area of square: " << s.calculateArea();
   cout<<"\nEnter radius to calculate the area of a circle: ";
   c.getData();
   cout << "Area of circle: " << c.calculateArea();

   return 0;
}
```

Output

```
Enter length to calculate the area of a square: 4
Area of square: 16
Enter radius to calculate the area of a circle: 5
Area of circle: 78.5
```

In this program, pure virtual function virtual float area() = 0; is defined inside the Shape class.

One important thing to note is that, you should override the pure virtual function of the base class in the derived class. If you fail the override it, the derived class will become an abstract class as well.

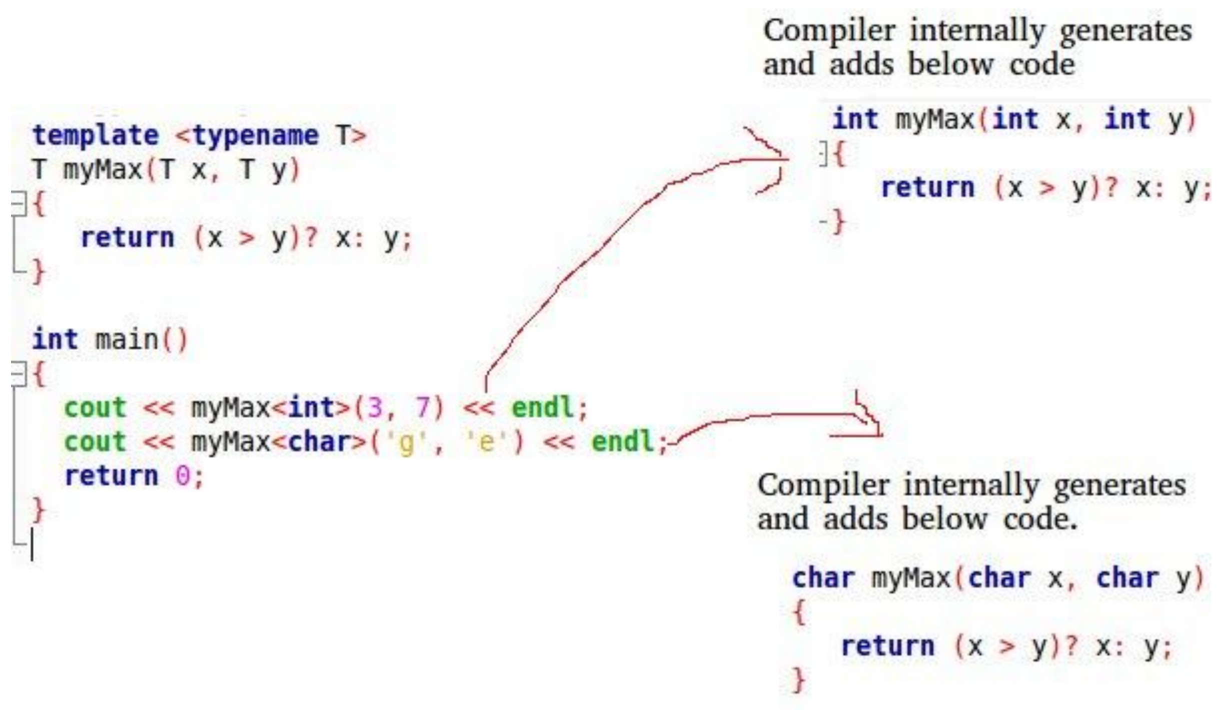
23. TEMPLATES

A template is a simple and yet very powerful tool in C++. The simple idea is to pass data type as a parameter so that we don't need to write the same code for different data types. For example, a software company may need sort() for different data types. Rather than writing and maintaining the multiple codes, we can write one sort() and pass data type as a parameter.

C++ adds two new keywords to support templates: *'template'* and *'typename'*. The second keyword can always be replaced by keyword 'class'.

How templates work?

Templates are expanded at compiler time. This is like macros. The difference is, compiler does type checking before template expansion. The idea is simple, source code contains only function/class, but compiled code may contain multiple copies of same function/class.



```
template <typename T>
T myMax(T x, T y)
{
   return (x > y)? x: y;
}

int main()
{
  cout << myMax<int>(3, 7) << endl;
  cout << myMax<char>('g', 'e') << endl;
  return 0;
}
```

Compiler internally generates and adds below code

```
int myMax(int x, int y)
{
   return (x > y)? x: y;
}
```

Compiler internally generates and adds below code.

```
char myMax(char x, char y)
{
   return (x > y)? x: y;
}
```

Function Templates: We write a generic function that can be used for different data types. Examples of function templates are sort(), max(), min(), printArray().

```
// One function works for all data types.  This would work
// even for user defined types if operator '>' is overloaded
template <typename T>
T myMax(T x, T y)
{
  return (x > y)? x: y;
}

int main()
{
 cout << myMax<int>(3, 7) << endl;  // Call myMax for int
 cout << myMax<double>(3.0, 7.0) << endl; // call myMax for double
```

```
  cout << myMax<char>('g', 'e') << endl;   // call myMax for char

  return 0;
}
```

Output:

```
7
7
g
```

24. NAMING SPACE

Namespaces allow us to group named entities that otherwise would have *global scope* into narrower scopes, giving them *namespace scope*. This allows organizing the elements of programs into different logical scopes referred to by names.

- Namespace is a feature added in C++ and not present in C.
- A namespace is a declarative region that provides a scope to the identifiers (names of the types, function, variables etc) inside it.
- Multiple namespace blocks with the same name are allowed. All declarations within those blocks are declared in the named scope.

A namespace definition begins with the keyword **namespace** followed by the namespace name as follows:

```
namespace namespace_name
{
   int x, y; // code declarations where
            // x and y are declared in
            // namespace_name's scope
}
```

- Namespace declarations appear only at global scope.
- Namespace declarations can be nested within another namespace.
- Namespace declarations don’t have access specifiers. (Public or private)
- No need to give semicolon after the closing brace of definition of namespace.
- We can split the definition of namespace over several units.

```
namespace ns1
{
   int value()    { return 5; }
}
namespace ns2
{
  const double x = 100;
  double value() {  return 2*x; }
}

int main()
{
  // Access value function within ns1
  cout << ns1::value() << '\n';

  // Access value function within ns2
  cout << ns2::value() << '\n';

  // Access variable x directly
  cout << ns2::x << '\n';

  return 0;
}
```

Output:

```
5
200
100
```

Short Questions (2-2 marks)

Explain following terms-

1. Object
2. Class
3. Inheritance
4. Polymorphism
5. Dynamic binding
6. Operator overloading
7. Tokens
8. Encapsulation
9. Data types
10. Type conversion
11. What are reference variables?
12. What are operators?
13. Differentiate between unary and binary operators.
14. Differentiate between relational and logical operators.
15. List various types of jump statements in C++.
16. What do you understand by looping?
17. Differentiate between entry control loop and exit control loop.
18. List various types of looping constructs in C++.
19. What are pointers?
20. Define naming space.
21. Define templates.
22. Differentiate between virtual function and pure virtual function.
23. Define virtual function.

Short notes (4-4 marks)

1. Type casting
2. Tokens
3. Structure of C++ program
4. Reference variable in C++
5. Looping in C++
6. Operators
7. Scope resolution operator
8. Memory allocation operators
9. Dynamic initialization of variables.
10. Pointers in C++
11. Virtual function

12. **Templates**
13. **Hybrid inheritance**
14. **Pure virtual function**
15. **Naming space in C++**

Differentiate between following:

- a. **Overloading and overriding**
- b. **Early binding and late binding**
- c. **Compile time and run time polymorphism**
- d. **Virtual function and pure virtual function**
- e. **Reference variable and pointer**
- f. **Constructor and destructor**
- g. **Static binding and dynamic binding**
- h. **Class and structure**
- i. **C and C++**
- j. **POP and OOP**
- k. **Top down and bottom up approach**

Questions (8-8 marks)

1. **Explain various principles of OOP.**
2. **Differentiate between C and C++.**
3. **What are data types? Explain various data types available in C++.**
4. **What do you understand by operators? Explain various types of operators available in C++.**
5. **What are control structures? Explain various control structures available in C++.**
6. **Differentiate between compile time and run time polymorphism? Explain run time polymorphism with suitable example.**
7. **What is inheritance? Explain its various types with suitable examples.**