PART -A

1. Define : objects and classes.

Object :

An Object is an instance of a Class. When a class is defined, no memory is allocated but when it is instantiated (i.e. an object is created) memory is allocated.

- When we define a class we define just a blueprint or we just map data members and member functions.
- The declaration of class develops a template but data members cannot be manipulated unless the object of its type is created.

Class:

The building block of C++ that leads to Object Oriented programming is a Class. It is a user defined data type, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class. A class is like a blueprint for an object.

For Example:

Consider the Class of Cars.

2. Define inline function.

Defining member function is to replace the function declaration by the actual function definition inside the class. It is treated as inline function.

The inline functions are a C++ enhancement feature to increase the execution time of a program. Functions can be instructed to compiler to make them inline so that compiler can replace those function definition wherever those are being called. What is meant by Constructors and list various types of constructors.

3. What is meant by Constructors and list various types of constructors.

1) Constructor is used for Initializing the values to the data members of the Class.

2) Constructor is that whose name is same as name of class.

3) Constructor gets Automatically called when an object of class is created.

4) Constructors never have a Return Type even void.

5) Constructor are of Default , Parameterized and Copy Constructors.

Types :

- Default constructor
- Parameterized constructor
- Copy constructor
- Destructor

4. What do you mean by type conversions?

A type cast is basically a conversion from one type to another. There are two types of type conversion:

(i) Implicit Type Conversion:

- Done by the compiler on its own, without any external trigger from the user.
- All the data types of the variables are upgraded to the data type of the variable with largest data type.

(ii) Explicit Type Conversion:

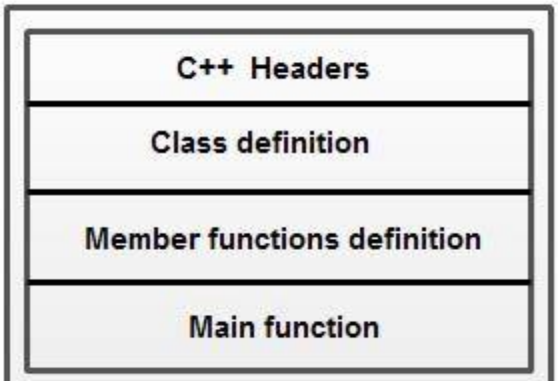
- This process is also called type casting and it is user-defined. Here the user can typecast the result to make it of a particular data type.
- By explicitly defining the required type in front of the expression in parenthesis. This can be also considered as forceful casting.

Syntax:

```
type (expression)
```

5. Give the structure of C++ program.

Programs are a sequence of instructions or statements. These statements form the structure of a C++ program. C++ program structure is divided into various sections, namely, headers, class definition, member functions definitions and main function.



Structure of a C++ Program

6. **What is scope resolution operator(::)?**

The :: (scope resolution) operator is used to get hidden names due to variable scopes so that you can still use them. The scope resolution operator can be used as both unary and binary. For example, if you have a global variable of name my_var and a local variable of name my_var, to access global my_var, you'll need to use the scope resolution operator.

7. **What is a copy constructor?**

A copy constructor is a member function which initializes an object using another object of the same class. A copy constructor has the following general function prototype:

```
ClassName (const ClassName &old_obj);
```

8. **What is meant by Destructor? Give example.**

A destructor is a special member function that works just opposite to constructor, unlike constructors that are used for initializing an object, destructors destroy (or delete) the object.

Syntax

```
~class_name()
{
  //Some code
}
```

9. List the applications of object oriented programming.

Main application areas of OOP are:

- User interface design such as windows, menu.
- Real Time Systems
- Simulation and Modeling
- Object oriented databases
- AI and Expert System
- Neural Networks and parallel programming
- Decision support and office automation systems etc.

10. Write the few words about object oriented languages.

The basic thing which are the essential feature of an object oriented programming are Inheritance, Polymorphism and Encapsulation. Any programming language that supports these feature completely are complete Object-oriented programming language.

- Inheritance is used to provide the concept of code-reusability.
- Polymorphism makes a language able to perform different task at different instance.
- Encapsulation makes data abstraction (security or privacy to data) possible. In object-oriented programming language, Encapsulation is achieved with the help of a class.

11. What is polymorphism.

The word polymorphism means having many forms. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form.

Ex:

A person at a same time can have different characteristic. Like a man at a same time is a father, a husband, a employee.

In C++ polymorphism is mainly divided into two types:

- Compile time Polymorphism
- Runtime Polymorphism

12. What is friend function.

A friend function of a class is defined outside that class' scope but it has the right to access all private and protected members of the class. Even though the prototypes for friend functions appear in the class definition, friends are not member functions.

A friend function can be:

a) A method of another class

b) A global function

13. Define Inheritance

Inheritance is one of the core feature of an object-oriented programming language. It allows software developers to derive a new class from the existing class. The derived class inherits the features of the base class (existing class).

Syntax

The basic syntax of inheritance is:

class DerivedClass : accessSpecifier BaseClass

14. Explain This pointers.

Every object in C++ has access to its own address through an important pointer called this pointer. The this pointer is an implicit parameter to all member functions. Therefore, inside a member function, this may be used to refer to the invoking object.

Friend functions do not have a this pointer, because friends are not members of a class. Only member functions have a this pointer.

1. Each object gets its own copy of the data member.

2. All access the same function definition as present in the code segment.

15. What do you mean by input stream?

C++ uses a convenient abstraction called streams to perform input and output operations in sequential media such as the screen, the keyboard or a file. A stream is an entity where a program can either insert or extract characters to/from.

If the direction of flow of bytes is from device(for example: Keyboard) to the main memory then this process is called input.

16. Give the importance of End-of-file.

C++ provides a special function, eof( ), that returns nonzero (meaning TRUE) when there are no more data to be read from an input file stream, and zero (meaning FALSE) otherwise.

Rules for using end-of-file (eof( )):

1. Always test for the end-of-file condition before processing data read from an input file stream.
   a. use a priming input statement before starting the loop
   b. repeat the input statement at the bottom of the loop body
2. Use a while loop for getting data from an input file stream. A for loop is desirable only when you know the exact number of data items in the file, which we do not know.

17. List three types of containers.

C++ contains three types of containers:

- Sequential Containers
- Associative Containers
- Unordered Containers

18. What is prototyping?

A function prototype is a declaration of the function that tells the program about the type of the value returned by the function and the number and type of arguments.

Function prototyping is one very useful feature of C++ function. A function prototype describes the function interface to the compiler by giving details such as the number and type of arguments and the type of return values.

19. Explain pointer in C++.

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.

Following are the valid pointer declaration

```
int    *ip;    // pointer to an integer
double *dp;    // pointer to a double
```

20. Define virtual function.

A virtual function a member function which is declared within base class and is re-defined (Overriden) by 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.

21. What is an iostream class?

This file defines the cin, cout, cerr and clog objects, which correspond to the standard input stream, the standard output stream, the un-buffered standard error stream and the buffered standard error stream, respectively.

22. Explain the classes for file stream operation.

C++ provides the following classes to perform output and input of characters to/from files:

- ofstream: Stream class to write on files
- ifstream: Stream class to read from files
- fstream: Stream class to both read and write from/to files.

These classes are derived directly or indirectly from the classes istream and ostream.

23. Explain containers.

Containers or container classes store objects and data. There are in total seven standard “first-class” container classes and three container adaptor classes and only seven header files that provide access to these containers or container adaptors.

- Sequence Containers
- Associative containers
- Unordered associative containers
- Container adaptors

24. What are the structured components of STL?

The Standard Template Library (STL) is a set of C++ template classes to provide common programming data structures and functions such as lists, stacks, arrays, etc.

STL has four components

- Algorithms
- Containers
- Functions
- Iterators

25. What is a pure function?

A function is called pure function if it always returns the same result for same argument values and it has no side effects like modifying an argument (or global variable) or outputting something. The only result of calling a pure function is the return value.

Examples : strlen(), pow(), sqrt() etc.

26. What is derived class?

Inheritance is one of the important feature of OOP which allows us to make hierarchical classifications of classes. In this, we can create a general class which defines the most common features. Other more specific classes can inherit this class to define those features that are unique to them. In this case, the classes which inherit from other classes, is referred as derived class.

For example, a general class vehicle can be inherited by more specific classes car and bike. The classes car and bike are derived classes in this case.

27. What are the functions for opening and closing a data file?

To open an input stream you must declare the stream to be of class ifstream. To open an output stream, it must be declared as class ofstream. A stream that will be be performing both input and output operations must be declared as class fstream.

Open a file for reading and writing purpose as follows

```
xyst.open("test",ios::in | ios::out);
```

To close a file, use the member function close().

```
xyst.close();
```

28. What is the meaning of random access?

Random file access enables us to read or write any data in our disk file without having to read or write every piece of data before it. We can Quickly search for data, Modify data, delete data in a random-access file.

In Random Access file, we can:

» We Can Read from Anywhere from the file.

» We Can Write to Anywhere in the file.

» We can Modify our record

» We can Search any Record from file

» And we can Delete Any record from file

29. Explain iterators.

An iterator is an object (like a pointer) that points to an element inside the container. We can use iterators to move through the contents of the container. They can be visualised as something similar to a pointer pointing to some location and we can access content at that particular location using them.

30. **What is the necessity for exception handling?**

C++ provides following specialized keywords for this purpose.

try: represents a block of code that can throw an exception.

catch: represents a block of code that is executed when a particular exception is thrown.

throw: Used to throw an exception. Also used to list the exceptions that a function throws, but doesn’t handle itself.

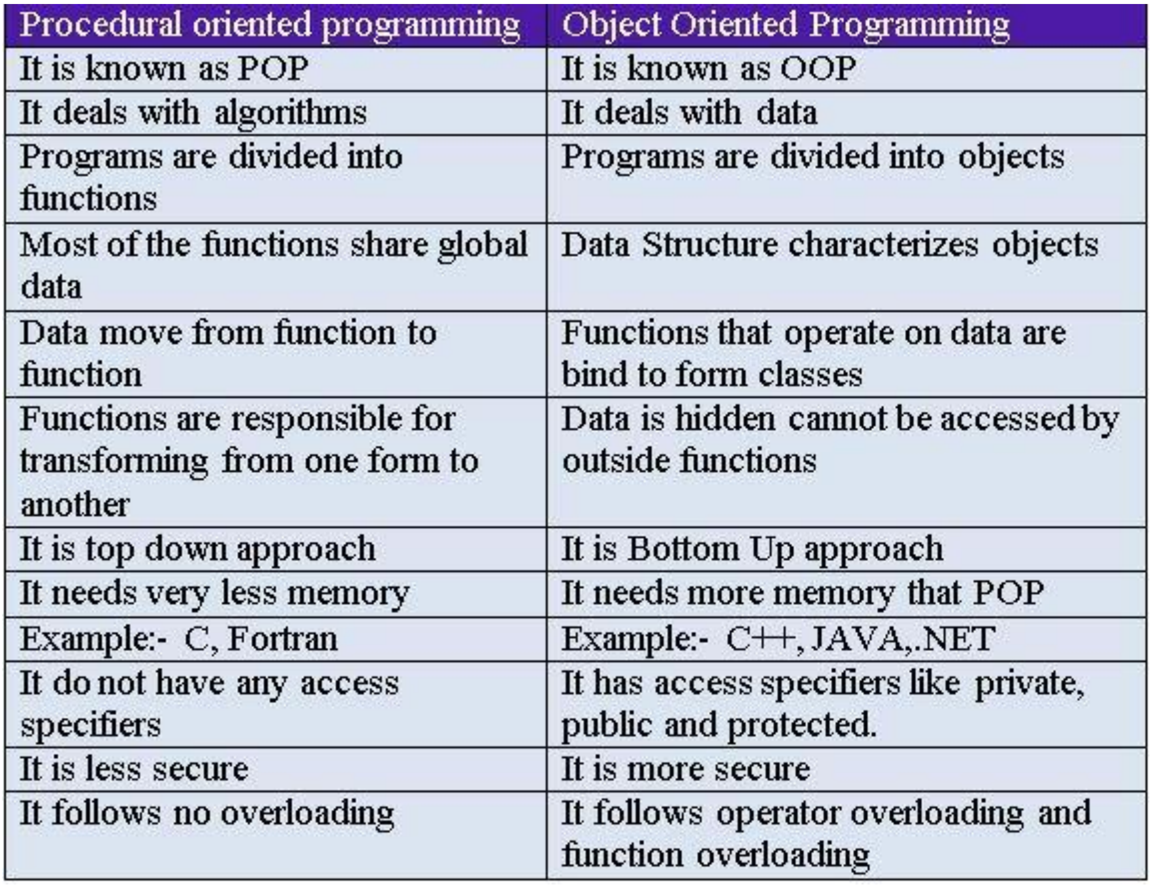
1) Separation of Error Handling code from Normal Code

2) Functions/Methods can handle any exceptions they choose

3) Grouping of Error Types

PART - B

31. a) Compare procedural and Object-oriented Programming.



Procedural oriented programming	Object Oriented Programming
It is known as POP	It is known as OOP
It deals with algorithms	It deals with data
Programs are divided into functions	Programs are divided into objects
Most of the functions share global data	Data Structure characterizes objects
Data move from function to function	Functions that operate on data are bind to form classes
Functions are responsible for transforming from one form to another	Data is hidden cannot be accessed by outside functions
It is top down approach	It is Bottom Up approach
It needs very less memory	It needs more memory that POP
Example:- C, Fortran	Example:- C++, JAVA,.NET
It do not have any access specifiers	It has access specifiers like private, public and protected.
It is less secure	It is more secure
It follows no overloading	It follows operator overloading and function overloading

31. b) Explain C++ tokens briefly.

A token is the smallest element of a program that is meaningful to the compiler. Tokens can be classified as follows:

- Keywords
- Identifiers
- Constants
- Strings

- Special Symbols
- Operators

Keyword:

Keywords are pre-defined or reserved words in a programming language. Each keyword is meant to perform a specific function in a program. C++ language supports 32 keywords which are given below:

auto
break
case
char
const
continue
default
do
double
else
enum
extern
float
for
goto
if
int
long
register
return
short
signed
sizeof
static
struct
switch
typedef
union
unsigned
void
volatile
while

Identifiers:

Identifiers are used as the general terminology for naming of variables, functions and arrays. These are user defined names consisting of arbitrarily long sequence of letters and digits with either a letter or the underscore(_) as a first character. Identifier names must differ in spelling and case from any keywords.

There are certain rules that should be followed while naming c identifiers:

- They must begin with a letter or underscore(_).
- They must consist of only letters, digits, or underscore. No other special character is allowed.
- It should not be a keyword.
- It must not contain white space.

Some examples of C++ identifiers:

_A9 Valid

Temp.var Invalid as it contains special character other than the underscore

void Invalid as it is a keyword

Constants:

Constants are also like normal variables. But, only difference is, their values cannot be modified by the program once they are defined. Constants refer to fixed values. They are also called as literals.

Constants may belong to any of the data type.

Syntax:

const data_type variable_name; (or) const data_type *variable_name;

Types of Constants:

- Integer constants – Example: 0, 1, 1218, 12482
- Real or Floating point constants – Example: 0.0, 1203.03, 30486.184
- Octal & Hexadecimal constants – Example: octal: (013)8 = (11)10, Hexadecimal: (013)16 = (19)10
- Character constants -Example: ‘a’, ‘A’, ‘z’
- String constants -Example: “Ramesh”

Strings:

Strings are nothing but an array of characters ended with a null character (‘\0’).This null character indicates the end of the string. Strings are always enclosed in double quotes. Whereas, a character is enclosed in single quotes in C and C++.

Declarations for String:

```
char string[20] = {‘r’, ’a’, ‘m’, ‘e’, ‘s’, ‘h’};
char string[20] = “ramesh”;
char string [] = “ramesh”;
```

Special Symbols:

The following special symbols are used in C having some special meaning and thus, cannot be used for some other purpose.

[] () {}, ; * = #

Operators:

Operators are symbols that triggers an action when applied to C++ variables and other objects. The data items on which operators act upon are called operands. Depending on the number of operands that an operator can act upon, operators can be classified as follows:

Unary Operators:

Those operators that require only single operand to act upon are known as unary operators.For Example increment and decrement operators

Binary Operators:

Those operators that require two operands to act upon are called binary operators. Binary operators are classified into :

- Arithmetic operators
- Relational Operators
- Logical Operators
- Assignment Operators
- Conditional Operators
- Bitwise Operators

Ternary Operators:

These operators requires three operands to act upon. For Example Conditional operator(?:).

32. a) What is Operator overloading? Explain how binary operator can be overloaded with example.

Operator Overloading :

In C++, we can make operators to work for user defined classes. This means C++ has the ability to provide the operators with a special meaning for a data type, this ability is known as operator overloading.

For example, we can overload an operator '+' in a class like String so that we can concatenate two strings by just using +.

Other example classes where arithmetic operators may be overloaded are Complex Number, Fractional Number, Big Integer, etc.

Binary operator in overloading :

Binary operator is an operator that takes two operand(variable). Binary operator overloading is similar to unary operator overloading except that a binary operator overloading requires an additional parameter.

- Binary Operators
- Arithmetic operators (+, -, *, /, %)
- Arithmetic assignment operators (+=, -=, *=, /=, %=)
- Relational operators (>, <, >=, <=, !=, ==)

Ex :

```
#include<iostream.h>
#include<conio.h>
class Rectangle
{
        int L,B;
        public:
        Rectangle()         //Default Constructor
        {
                L = 0;
                B = 0;
        }
        Rectangle(int x,int y)          //Parameterize Constructor
        {
                L = x;
                B = y;
        }



        Rectangle operator+(Rectangle Rec)
        {
                Rectangle R;
                        R.L = L + Rec.L;
                        R.B = B + Rec.B;
                return R;
        }
        void Display()
        {
                cout<<"\n\tLength : "<<L;
                cout<<"\n\tBreadth : "<<B;
        }
};
void main()
{
        Rectangle R1(2,5),R2(3,4),R3;
        cout<<"\n\tRectangle 1 : ";
        R1.Display();
        cout<<"\n\n\tRectangle 2 : ";
        R2.Display();
        R3 = R1 + R2;       Statement 1
```

```
                cout<<"\n\n\tRectangle 3 : ";
                R3.Display();
        }
```

Output :

```
        Rectangle 1 :
        L : 2
        B : 5



        Rectangle 2 :
        L : 3
        B : 4

        Rectangle 3 :
        L : 5
        B : 9
```

32. b) Describe the type conversions with examples.

A type cast is basically a conversion from one type to another. There are two types of type conversion:

1. **Implicit Type Conversion**

   It is also known as ‘automatic type conversion’. Done by the compiler on its own, without any external trigger from the user. Generally takes place when in an expression more than one data type is present. In such condition type conversion (type promotion) takes place to avoid lose of data. All the data types of the variables are upgraded to the data type of the variable with largest data type.

It is possible for implicit conversions to lose information, signs can be lost (when signed is implicitly converted to unsigned), and overflow can occur (when long long is implicitly converted to float).

Ex :

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

int main()
{
int x = 10; // integer x
char y = 'a'; // character c
x = x + y;
float z = x + 1.0;

cout << "x = " << x << endl
        << "y = " << y << endl
        << "z = " << z << endl;
return 0;
}
```

Output :

```
x = 107
y = a
z = 108
```

2. **Explicit Type Conversion:**

This process is also called type casting and it is user-defined. Here the user can typecast the result to make it of a particular data type.

In C++, it can be done by two ways:

- Converting by assignment: This is done by explicitly defining the required type in front of the expression in parenthesis. This can be also considered as forceful casting.

Syntax:

```
(type) expression
```

Ex :

```
#include <iostream>
using namespace std;
int main()
{
        double x = 1.2;
        int sum = (int)x + 1;
        cout << "Sum = " << sum;
        return 0;
}
```

Output :

```
        Sum = 2
```

- Conversion using Cast operator: A Cast operator is an unary operator which forces one data type to be converted into another data type.

C++ supports four types of casting:

- Static Cast
- Dynamic Cast
- Const Cast
- Reinterpret Cast

Ex :

```
        #include <iostream>
        using namespace std;
        int main()
        {
                float f = 3.5;
                int b = static_cast<int>(f);
                cout << b;
        }
```

33. a) Explain the basic data types in C++.

All variables use data-type during declaration to restrict the type of data to be stored. Therefore, we can say that data types are used to tell the variables the type of data it can store.

Data types in C++ is mainly divided into two types:

1. Primitive Data Types:

These data types are built-in or predefined data types and can be used directly by the user to declare variables. example: int, char , float, bool etc. Primitive data types available in C++ are:

- Integer
- Character
- Boolean
- Floating Point
- Double Floating Point
- Valueless or Void

2. Abstract or user defined data type:

These data types are defined by user itself. Like, defining a class in C++ or a structure.

This one describes primitive data types available in C++

Integer:

Keyword used for integer data types is int. Integers typically requires 4 bytes of memory space and ranges from -2147483648 to 2147483647.

Character:

Character data type

Character data type is used for storing characters. Keyword used for character data type is char. Characters typically requires 1 byte of memory space and ranges from -128 to 127 or 0 to 255.

Boolean:

Boolean data type is used for storing boolean or logical values. A boolean variable can store either true or false. Keyword used for boolean data type is bool.

Floating Point:

Floating Point data type is used for storing single precision floating point values or decimal values. Keyword used for floating point data type is float. Float variables typically requires 4 byte of memory space.

Double Floating Point:

Double Floating Point data type is used for storing double precision floating point values or decimal values. Keyword used for double floating point data type is double. Double variables typically requires 8 byte of memory space.

void:

Void means without any value. void datatype represents a valueless entity. Void data type is used for those function which does not returns a value.

DATA TYPE	**SIZE (IN BYTES)**	**RANGE**
short int	2	-32,768 to 32,767
unsigned short int	2	0 to 65,535
unsigned int	4	0 to 4,294,967,295
Int	4	-2,147,483,648 to 2,147,483,647
long int	4	-2,147,483,648 to 2,147,483,647
unsigned long int	4	0 to 4,294,967,295
long long int	8	-(2^63) to (2^63)-1

unsigned long long int	8	0 to 18,446,744,073,709,551,615
signed char	1	-128 to 127
unsigned char	1	0 to 255
Float	4	
Double	8	
long double	12	

33. b) Explain inline function with example.

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

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

To inline a function, place the keyword inline before the function name and define the function before any calls are made to the function. The compiler can ignore the inline qualifier in case defined function is more than a line.

A function definition in a class definition is an inline function definition, even without the use of the inline specifier.

Syntax :

```
inline return-type function-name(parameters)
{
   // function code
}
```

Inline functions provide following advantages:

1) Function call overhead doesn’t occur.

2) It also saves the overhead of push/pop variables on the stack when function is called.

3) It also saves overhead of a return call from a function.

4) When you inline a function, you may enable compiler to perform context specific optimization on the body of function. Such optimizations are not possible for normal function calls. Other optimizations can be obtained by considering the flows of calling context and the called context.

5) Inline function may be useful (if it is small) for embedded systems because inline can yield less code than the function call preamble and return.

Inline function disadvantages:

1) The added variables from the inlined function consumes additional registers, After in-lining function if variables number which are going to use register increases than they may create overhead on register variable resource utilization. This means that when inline function body is substituted at the point of function call, total number of variables used by the function also gets inserted. So the number of register going to be used for the variables will also get increased. So if after function inlining variable numbers increase drastically then it would surely cause an overhead on register utilization.

2) If you use too many inline functions then the size of the binary executable file will be large, because of the duplication of same code.

3) Too much inlining can also reduce your instruction cache hit rate, thus reducing the speed of instruction fetch from that of cache memory to that of primary memory.

4) Inline function may increase compile time overhead if someone changes the code inside the inline function then all the calling location has to be recompiled because compiler would require to replace all the code once again to reflect the changes, otherwise it will continue with old functionality.

5) Inline functions may not be useful for many embedded systems. Because in embedded systems code size is more important than speed.

6) Inline functions might cause thrashing because inlining might increase size of the binary executable file. Thrashing in memory causes performance of computer to degrade.

The following program demonstrates the use of use of inline function.

```
#include <iostream>
using namespace std;
class operation
{
        int a,b,add,sub,mul;
        float div;
public:
        void get();
        void sum();
        void difference();
        void product();
        void division();
};
inline void operation :: get()
{
        cout << "Enter first value:";
        cin >> a;
        cout << "Enter second value:";
        cin >> b;
}

inline void operation :: sum()
{
        add = a+b;
        cout << "Addition of two numbers: " << a+b << "\n";
}

inline void operation :: difference()
{
        sub = a-b;
        cout << "Difference of two numbers: " << a-b << "\n";
}
int main()
{
        cout << "Program using inline function\n";
        operation s;
        s.get();
        s.sum();
        s.difference();
        return 0;
}
```

34. a) Explain the constructor with an example.

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

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.

Types of Constructor:

1. **Default Constructors:**

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

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.

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.

Syntax:

```
class class-name
{

   Access Specifier :
   Member - Variables
   Member - Functions
public:
   class-name() {
      // Constructor code
   }

   //... other Variables & Functions
}
```

Ex:

```
include<conio.h>
using namespace std;
class Example {
   int a, b;
public:
   Example()
{
      a = 10;
      b = 20;
      cout << "I am Constructor\n";
   }
   void Display() {
      cout << "Values :" << a << "\t" << b;
   }
};
```

```
int main()
{
   Example Object;
   Object.Display();
   getch();
   return 0;
}
```

Ouptut:

```
    I am Constructor
    Values :10      20
```

34. b) Explain Classes and Objects with example.

Class:

The building block of C++ that leads to Object Oriented programming is a Class. It is a user defined data type, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class. A class is like a blueprint for an object.

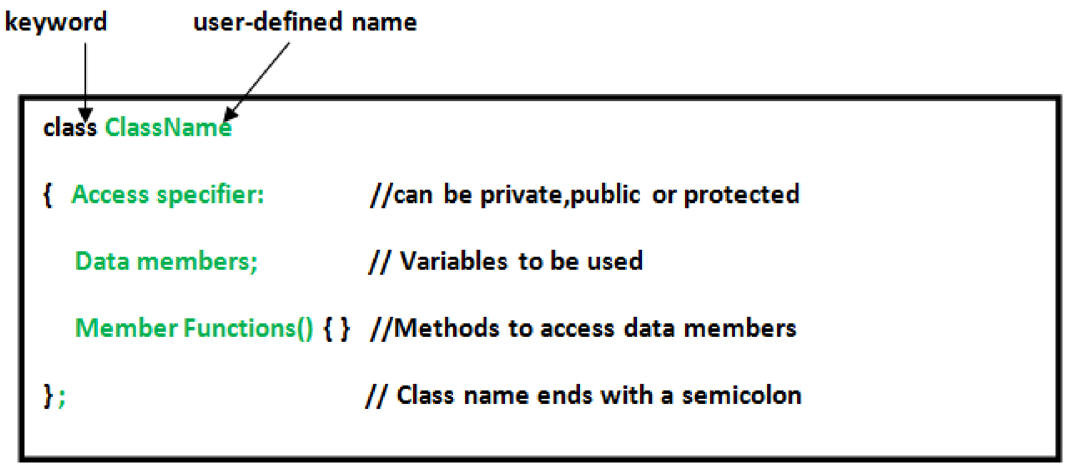
A Class is a user defined data-type which has data members and member functions. Data members are the data variables and member functions are the functions used to manipulate these variables and together these data members and member functions defines the properties and behavior of the objects in a Class.

Object :

An Object is an instance of a Class. When a class is defined, no memory is allocated but when it is instantiated (i.e. an object is created) memory is allocated.

Defining Class and Declaring Objects

A class is defined in C++ using keyword class followed by the name of class. The body of class is defined inside the curly brackets and terminated by a semicolon at the end.



```
class ClassName
{  Access specifier:          //can be private,public or protected
   Data members;              // Variables to be used
   Member Functions() { }     //Methods to access data members
};                            // Class name ends with a semicolon
```

Declaring Objects:

When a class is defined, only the specification for the object is defined; no memory or storage is allocated. To use the data and access functions defined in the class, you need to create objects.

Syntax:

ClassName ObjectName;

Accessing data members and member functions: The data members and member functions of class can be accessed using the dot('.') operator with the object.

Accessing Data Members

The public data members are also accessed in the same way given however the private data members are not allowed to be accessed directly by the object. Accessing a data member depends solely on the access control of that data member.

This access control is given by Access modifiers in C++. There are three access modifiers: public, private and protected.

Ex:

```
#include <iostream.h>
using namespace std;
class test
{
        public:
        string myname;
        void printname()
        {
        cout << "My name is: " << myname;
        }
};
int main() {

        test obj1;
        obj1.myname = "ram";
        obj1.printname();
        return 0;
}
```

Output:

```
My name is : ram
```

35. a) Describe the different styles of function prototypes.

Prototype of a function is the function without its body. Prototyping of a function makes the program easier to understand and makes the program short and more convenient.

Function prototype are usually written at the beginning of a program, ahead of any programmer-defined functions including main(). We can add as many parameter to the functions as much programmer needs

Making function prototyping and using includes 3 steps :-

1) Function Prototype;

2) Function Definition;

3) Function Call.

Syntax :

```
<ReturnType> <FunctionName> (type 1 arg. 1, type 2 arg. 2,...type n arg n);
```

Define function :

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

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

Return Type − A function may return a value. The return_type is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return_type is the keyword void.

Function Name − This is the actual name of the function. The function name and the parameter list together constitute the function signature.

Parameters − A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.

Function Body − The function body contains a collection of statements that define what the function does.

Calling function

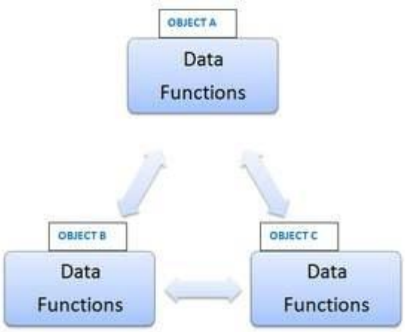
To call a function, you simply need to pass the required parameters along with function name, and if function returns a value, then you can store returned value.

Ex:

```
#include <iostream>
using namespace std;
int max(int num1, int num2);
int main ()
{
  int a = 100;
  int b = 200;
  int ret;
  ret = max(a, b);
  cout << "Max value is : " << ret << endl;
  return 0;
}

int max(int num1, int num2)
{
  int result;
  if (num1 > num2)
    result = num1;
  else
    result = num2;

  return result;
}
```

35. b) List some of the striking features of OOP.

Object oriented programming(OOP), treats data as a critical element in the program and does not allow it to flow freely around the system. It ties data more closely to the functions that operate on it to flow freely around the system. It allows decomposition of a problem into a number of entities called Objects. The data of an object can be accessed only by the functions associated with the object.



***OOP Characteristics*:**

- Objects
- Classes
- Data Abstraction
- Encapsulation
- Inheritance
- Polymorphism
- Dynamic Binding
- Message Passing

***Objects*:**

Objects are the basic run-time entities in an object-oriented programming. They may represents a person, a place, a bank account, a table of data or any item that the program has to handle. They may also represents user-defined data such as vector, time and lists. When the program is executed, the object interact by sending message to one another.

***Classes*:**

Objects contain data, and code to manipulate that data. The entire set of data and code of an object can be made a user-defined data type with the help of class. In fact, objects are variable of the type class. Once a class has been defined, we can create a number of objects belonging to that class. A class is a collection of objects of similar type.

Ex. Fruit mango;

***Data Abstraction*:**

Abstractions refer to the act of representing essential features without including background details or explanation. They are commonly known as Abstraction Data Type(ADT).

***Encapsulation*:**

The wrapping up of data and functions into single unit is known as *encapsulation*. Data encapsulation is a striking feature of a class. The data is not accessible to the outside world, and only those functions which are wrapped in the class can access it. These functions provide the interface between the object's data and the program.

***Inheritance*:**

Inheritance is the process by which objects of one class acquire the properties of object of another class. The class whose members are inherited is called the Base class and the class that inherits those members is called Derived class. It supports class of hierarchical classification.

***Polymorphism*:**

Polymorphism is another OOP concept. Polymorphism means the ability to take more than one form. An operation may exhibit different behaviors at different instances.

36. a) Explain about multiple constructor with example in C++.

Definition

In C++, Constructor is automatically called when an object(an instance of the lass) create. It is the special member function of the class. Which constructor has arguments is called Parameterized Constructor.

One Constructor overload another constructor is called Constructor Overloading . It has the same name of the class.

- It must be a public member.
- No Return Values.
- Default constructors are called when constructors are not defined for the classes.

Syntax:

```
class class_name
{
   Access Specifier :
   Member_Variables
   Member_Functions
public:
   class_name()
{
// Constructor code
   }
   class_name(variables)
 {
// Constructor code
   }
   ... other Variables & Functions
}
```

Ex:

```
#include<iostream>
#include<conio.h>
using namespace std;
class Example
{
    int a, b;
public:
    Example()
{
        a = 50;
        b = 100;
        cout << "\nIm Constructor";
    }
    Example(int x, int y)
{
        a = x;
        b = y;
        cout << "\nIm Constructor";
    }
    void Display()
 {
        cout << "\nValues :" << a << "\t" << b;
    }
};

int main()
{
    Example Object(10, 20);
    Example Object2;
    Object.Display();
    Object2.Display();
    getch();
    return 0;
}
```

36. b) Explain pointers to members with example.

Pointer to Data Members of Class

We can use pointer to point to class's data members (Member variables).

Syntax :

```
datatype class_name :: *pointer_name;
```

Syntax for Assignment :

```
pointer_name = &class_name :: datamember_name;
```

Both declaration and assignment can be done in a single statement too.

```
datatype class_name::*pointer_name = &class_name::datamember_name ;
```

Using Pointers with Objects

For accessing normal data members we use the dot . operator with object and -> with pointer to object. But when we have a pointer to data member, we have to dereference that pointer to get what its pointing to, hence it becomes,

```
Object.*pointerToMember
```

and with pointer to object, it can be accessed by writing,

```
ObjectPointer->*pointerToMember
```

Example:

```
class Data
{
   public:
   int a;
   void print()
   {
      cout << "a is "<< a;
   }
};
int main()
{
   Data d, *dp;
   dp = &d;     // pointer to object
```

```
    int Data::*ptr=&Data::a;   // pointer to data member 'a'
    d.*ptr=10;
    d.print();
    dp->*ptr=20;
    dp->print();
}
```

Output :

```
a is 10
a is 20
```

37. a) What is meant by Inheritance? Write the syntax with example.

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.

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.

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
};
```

Ex:

```
// C++ program to demonstrate implementation of Inheritance
#include <bits/stdc++.h>
using namespace std;
class Parent
{
        public:
        int id_p;
};
class Child : public Parent
{
        public:
        int id_c;
};
int main()
{
                Child obj1;
                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’.

37. b) What is Virtual function? Write the syntax with example.

A virtual function a member function which is declared within base class and is re-defined (Overriden) by 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.

Syntax :

```
class Base {
public:
  virtual arbitrary_return_type virt0( /*...arbitrary params...*/ );
  virtual arbitrary_return_type virt1( /*...arbitrary params...*/ );
  //...
};
```

Ex:

```
// CPP program to illustrate  concept of Virtual Functions
#include<iostream>
using namespace std;
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;
        bptr->print();
        bptr->show();
}
```

Output:

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

38. a) Write short notes on function template.

Function templates are special functions that can operate with generic types. This allows us to create a function template whose functionality can be adapted to more than one type or class without repeating the entire code for each type.

In C++ this can be achieved using template parameters. A template parameter is a special kind of parameter that can be used to pass a type as argument: just like regular function parameters can be used to pass values to a function, template parameters allow to pass also types to a function. These function templates can use these parameters as if they were any other regular type.

The format for declaring function templates with type parameters is:

```
template <class identifier> function_declaration;
template <typename identifier> function_declaration;
```

The only difference between both prototypes is the use of either the keyword class or the keyword typename. Its use is indistinct, since both expressions have exactly the same meaning and behave exactly the same way.

To use this function template we use the following format for the function call:

```
function_name <type> (parameters);
```

Example:

```
// function template
#include <iostream>
using namespace std;
template <class T>
T GetMax (T a, T b)
{
  T result;
  result = (a>b)? a : b;
  return (result);
}
int main ()
 {
  int i=5, j=6, k;
  long l=10, m=5, n;
 k=GetMax<int>(i,j);
 n=GetMax<long>(l,m);
 cout << k << endl;
 cout << n << endl;
 return 0;
}
```

38. b) Explain exception handling with syntax and example.

An exception is a problem that arises during the execution of a program. A C++ exception is a response to an exceptional circumstance that arises while a program is running, such as an attempt to divide by zero.

C++ exception handling is built upon three keywords:

- Try
- catch and
- throw.

throw − A program throws an exception when a problem shows up. This is done using a throw keyword.

catch − A program catches an exception with an exception handler at the place in a program where you want to handle the problem. The catch keyword indicates the catching of an exception.

try − A try block identifies a block of code for which particular exceptions will be activated. It's followed by one or more catch blocks.

The syntax for using try/catch as follows

```
try
```

```
{
   // protected code
}
 catch( ExceptionName e1 )
 {
   // catch block
}
 catch( ExceptionName e2 )
 {
   // catch block
}
 catch( ExceptionName eN )
 {
   // catch block
}
```

You can list down multiple catch statements to catch different type of exceptions in case your try block raises more than one exception in different situations.

Throwing Exceptions

Exceptions can be thrown anywhere within a code block using throw statement. The operand of the throw statement determines a type for the exception and can be any expression and the type of the result of the expression determines the type of exception thrown.

Catching Exceptions

The catch block following the try block catches any exception. You can specify what type of exception you want to catch and this is determined by the exception declaration that appears in parentheses following the keyword catch.

Ex:

```
#include <iostream>
```

```
using namespace std;
double division(int a, int b)
 {
   if( b == 0 )
{
      throw "Division by zero condition!";
   }
   return (a/b);
}
int main ()
 {
   int x = 50;
   int y = 0;
   double z = 0;
   try
{
      z = division(x, y);
      cout << z << endl;
   }
 catch (const char* msg)
{
     cerr << msg << endl;
   }   return 0;
}
```

Result : Division by zero condition!

39. a) Write short notes on Standard template library(STL).

The Standard Template Library (STL) is a set of C++ template classes to provide common programming data structures and functions such as lists, stacks, arrays, etc. It is a library of container classes, algorithms, and iterators. It is a generalized library and so, its components are parameterized. A working knowledge of template classes is a prerequisite for working with STL.

STL has four components

- Algorithms
- Containers
- Functions
- Iterators

Algorithms

The header algorithm defines a collection of functions especially designed to be used on ranges of elements.They act on containers and provide means for various operations for the contents of the containers.

- Algorithm
  - Sorting
  - Searching
  - Important STL Algorithms
  - Useful Array algorithms
  - Partition Operations
- Numeric
  - valarray class

Containers

Containers or container classes store objects and data. There are in total seven standard “first-class” container classes and three container adaptor classes and only seven header files that provide access to these containers or container adaptors.

- **Sequence Containers**: implement data structures which can be accessed in a sequential manner.
  - vector
  - list
  - deque
  - arrays
  - forward_list
- **Container Adaptors** : provide a different interface for sequential containers.
  - queue
  - priority_queue
  - stack
- **Associative Containers** : implement sorted data structures that can be quickly searched.
- **Unordered Associative Containers** : implement unordered data structures that can be quickly searched

Functions

The STL includes classes that overload the function call operator. Instances of such classes are called function objects or functors. Functors allow the working of the associated function to be customized with the help of parameters to be passed.

- Functors

Iterators

As the name suggests, iterators are used for working upon a sequence of values. They are the major feature that allow generality in STL.

- Iterators

39. b) Write short notes on application of container classes.

There are three most popular containers namely,

1) Vector
2) List
3) Map

1. **Vector :**

Vectors are same as dynamic arrays with the ability to resize itself automatically when an element is inserted or deleted, with their storage being handled automatically by the container. Vector elements are placed in contiguous storage so that they can be accessed and traversed using iterators. In vectors, data is inserted at the end. Inserting at the end takes differential time, as sometimes there may be a need of extending the array. Removing the last element takes only constant time because no resizing happens. Inserting and erasing at the beginning or in the middle is linear in time.

Certain functions associated with the vector are:

Iterators

begin() – Returns an iterator pointing to the first element in the vector.

end() – Returns an iterator pointing to the theoretical element that follows the last element in the vector.

rbegin() – Returns a reverse iterator pointing to the last element in the vector (reverse beginning). It moves from last to first element.

rend() – Returns a reverse iterator pointing to the theoretical element preceding the first element in the vector .

cbegin() – Returns a constant iterator pointing to the first element in the vector.

cend() – Returns a constant iterator pointing to the theoretical element that follows the last element in the vector.

crbegin() – Returns a constant reverse iterator pointing to the last element in the vector (reverse beginning). It moves from last to first element.

crend() – Returns a constant reverse iterator pointing to the theoretical element preceding the first element in the vector .

2. **List**

Lists are sequence containers that allow non-contiguous memory allocation. As compared to vector, list has slow traversal, but once a position has been found, insertion and deletion are quick. Array and Vector are contiguous containers, i.e they store their data on continuous memory, thus the insert operation at the middle of vector/array is very costly (in terms of number of operaton and process time) because we have to shift all the elements, linked list overcome this problem. Linked list can be implemented by using the list container.

Syntax for creating a new linked list using list template is:

```
#include <iostream>

#include <list>

int main()

{

   std::list<int> l;

}
```

Functions used with List:

front() – Returns the value of the first element in the list.

back() – Returns the value of the last element in the list .

push_front(g) – Adds a new element ‘g’ at the beginning of the list .

push_back(g) – Adds a new element ‘g’ at the end of the list.

pop_front() – Removes first element of the list, and reduces size of the list by 1

pop_back() – Removes the last element of the list, and reduces size of the list by 1

list::begin() and list::end() in C++ STL– begin() function returns an iterator pointing to the first element of the list

end()– end() function returns an iterator pointing to the theoretical last element which follows the last element.

3. Maps

Maps are associative containers that store elements in a mapped fashion. Each element has a key value and a mapped value. No two mapped values can have same key values.

Some basic functions associated with Map:

begin() – Returns an iterator to the first element in the map.

end() – Returns an iterator to the theoretical element that follows last element in the map.

size() – Returns the number of elements in the map.

max_size() – Returns the maximum number of elements that the map can hold

empty() – Returns whether the map is empty.

pair insert(keyvalue, mapvalue) – Adds a new element to the map.

erase(iterator position) – Removes the element at the position pointed by the iterator.

erase(const g)– Removes the key value ‘g’ from the map.

clear() – Removes all the elements from the map.

40. a) Explain the uses of Pointers in C++.

1. To pass arguments by reference. Passing by reference serves two purposes
   (i) To modify variable of function in other. Example to swap two variables;
   (ii) For efficiency purpose. Example passing large structure without reference would create a copy of the structure.
2. For accessing array elements. Compiler internally uses pointers to access array elements.
3. To return multiple values.
   Example returning square and square root of numbers.
4. Dynamic memory allocation : We can use pointers to dynamically allocate memory. The advantage of dynamically allocated memory is, it is not deleted until we explicitly Delete it.
5. To implement data structures.
   Example linked list, tree, etc. We cannot use C++ references to implement these data structures because references are fixed to a location .
6. To do system level programming where memory addresses are useful.
   For example shared memory used by multiple threads. For more examples, see IPC through shared memory, Socket Programming in C/C++, etc.
7. Reduces the storage space and complexity of the program.
8. Reduces the execution time of the program.

9. Pointers help us to build complex data structures like linked lists, queues, stacks, trees, graphs etc.

40. b) Write a suitable program explain the working of inheritance.

Inheritance is one of the feature of Object Oriented Programming System(OOPs), it allows the child class to acquire the properties (the data members) and functionality (the member functions) of parent class.

```
// C++ program to demonstrate implementation  of Inheritance
#include <bits/stdc++.h>
using namespace std;
class Parent
{
        public:
        int id_p;
};
class Child : public Parent
{
        public:
        int id_c;
};
int main()
{
                Child obj1;
                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
```

41. a) Write short notes on class template.

A class implementation that is same for all classes, only the data types used are different. Normally, you would need to create a different class for each data type or create different member variables and functions within a single class.

This will unnecessarily bloat your code base and will be hard to maintain, as a change is one class/function should be performed on all classes/functions.However, class templates make it easy to reuse the same code for all data types.

Declaration :

```
template <class T>
class className
{
   ... .. ...
public:
   T var;
   T someOperation(T arg);
   ... .. ...
};
```

In the above declaration, T is the template argument which is a placeholder for the data type used. Inside the class body, a member variable var and a member function someOperation() are both of type T.

create a class template object?

```
className<dataType> classObject;
```

Example for simple calculator using class template

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

```
template <class T>
class Calculator
{
private:
    T num1, num2;
public:
    Calculator(T n1, T n2)
    {
        num1 = n1;
        num2 = n2;
    }
    void displayResult()
    {
        cout << "Numbers are: " << num1 << " and " << num2 << "." << endl;
        cout << "Addition is: " << add() << endl;
        cout << "Subtraction is: " << subtract() << endl;
        cout << "Product is: " << multiply() << endl;
        cout << "Division is: " << divide() << endl;
    }
    T add() { return num1 + num2; }
    T subtract() { return num1 - num2; }
    T multiply() { return num1 * num2; }
    T divide() { return num1 / num2; }
};
int main()
{
    Calculator<int> intCalc(2, 1);
    Calculator<float> floatCalc(2.4, 1.2);
    cout << "Int results:" << endl;
    intCalc.displayResult();
    cout << endl << "Float results:" << endl;
```

```
        floatCalc.displayResult();
        return 0;
}
```

Output

```
        Int results:
        Numbers are: 2 and 1.
        Addition is: 3
        Subtraction is: 1
        Product is: 2
        Division is: 2

        Float results:
        Numbers are: 2.4 and 1.2.
        Addition is: 3.6
        Subtraction is: 1.2
        Product is: 2.88
        Division is: 2
```

41. b) Explain about unformatted input output function with suitable example.

Unformatted Input/Output is the most basic form of input/output. Unformatted input/output transfers the internal binary representation of the data directly between memory and the file.

The following are operations of unformatted consol input / output operations:

A) void get()

It is a method of cin object used to input a single character from keyboard. But its main property is that it allows wide spaces and newline character.

Syntax:

```
char c=cin.get();
```

B) void put()

It is a method of cout object and it is used to print the specified character on the screen or monitor.

Syntax:

```
cout.put(variable / character);
```

C) getline(char *buffer,int size)

This is a method of cin object and it is used to input a string with multiple spaces.

Syntax:

```
char x[30];
cin.getline(x,30);
```

D) write(char * buffer, int n)

It is a method of cout object. This method is used to read n character from buffer variable.

Syntax:

```
cout.write(x,2);
```

E) cin

It is the method to take input any variable / character / string.

Syntax:

```
cin>>variable / character / String / ;
```

F) cout

This method is used to print variable / string / character.

Syntax:

```
cout<< variable / charcter / string;
```

Ex:

```
//C++ program to demonstrate getline() and write() functions
#include <iostream>
using namespace std;
int main()
{
    char str[30];
    cin.getline(str, 10);
    cout.write(str, 10);
    return 0;
}
```

Input: C++ rocks

Output of the above program is:

```
C++ rocks
```

42. a) Explain the steps in Object Oriented Design.

Object Oriented Design is the concept that forces programmers to plan out their code in order to have a better flowing program. After the analysis phase, the conceptual model is developed further into an object-oriented model using object-oriented design (OOD).

In OOD, the technology-independent concepts in the analysis model are mapped onto implementing classes, constraints are identified, and interfaces are designed, resulting in a model for the solution domain. In a nutshell, a detailed description is constructed specifying how the system is to be built on concrete technologies.

The stages for object–oriented design can be identified as

- Definition of the context of the system
- Designing system architecture
- Identification of the objects in the system
- Construction of design models
- Specification of object interfaces

Object Oriented Design is defined as a programming language that has 5 conceptual tools to aid the programmer. These programs are often more readable than non-object oriented programs, and debugging becomes easier with locality.

Encapsulation

A tight coupling or association of data structures with the methods or functions that act on the data. This is called a class, or object (an object is often the implementation of a class).

Data Protection

The ability to protect some components of the object from external entities. This is realized by language keywords to enable a variable to be declared as private or protected to the owning class.

Inheritance

The ability for a class to extend or override functionality of another class. The so called child class has a whole section that is the parent class and then it has it's own set of functions and data.

Interface

A definition of functions or methods, and their signatures that are available for use to manipulate a given instance of an object.

Polymorphism

The ability to define different functions or classes as having the same name but taking different data types.

42. b) Explain the steps in Object Oriented Analysis.

The three analysis techniques that are used in conjunction with each other for object-oriented analysis are object modelling, dynamic modelling, and functional modelling.

Object Modelling

Object modelling develops the static structure of the software system in terms of objects. It identifies the objects, the classes into which the objects can be grouped into and the relationships between the objects. It also identifies the main attributes and operations that characterize each class.

The process of object modelling can be visualized in the following steps −

- Identify objects and group into classes
- Identify the relationships among classes
- Create user object model diagram
- Define user object attributes
- Define the operations that should be performed on the classes
- Review glossary

Dynamic Modelling

After the static behavior of the system is analyzed, its behavior with respect to time and external changes needs to be examined. This is the purpose of dynamic modelling.

Dynamic Modelling can be defined as "a way of describing how an individual object responds to events, either internal events triggered by other objects, or external events triggered by the outside world".

The process of dynamic modelling can be visualized in the following steps −

- Identify states of each object
- Identify events and analyze the applicability of actions
- Construct dynamic model diagram, comprising of state transition diagrams
- Express each state in terms of object attributes
- Validate the state–transition diagrams drawn

Functional Modelling

Functional Modelling is the final component of object-oriented analysis. The functional model shows the processes that are performed within an object and how the data changes as it moves between methods. It specifies the meaning of the operations of object modelling and the actions of dynamic modelling. The functional model corresponds to the data flow diagram of traditional structured analysis.

The process of functional modelling can be visualized in the following steps −

- Identify all the inputs and outputs
- Construct data flow diagrams showing functional dependencies
- State the purpose of each function
- Identify constraints
- Specify optimization criteria

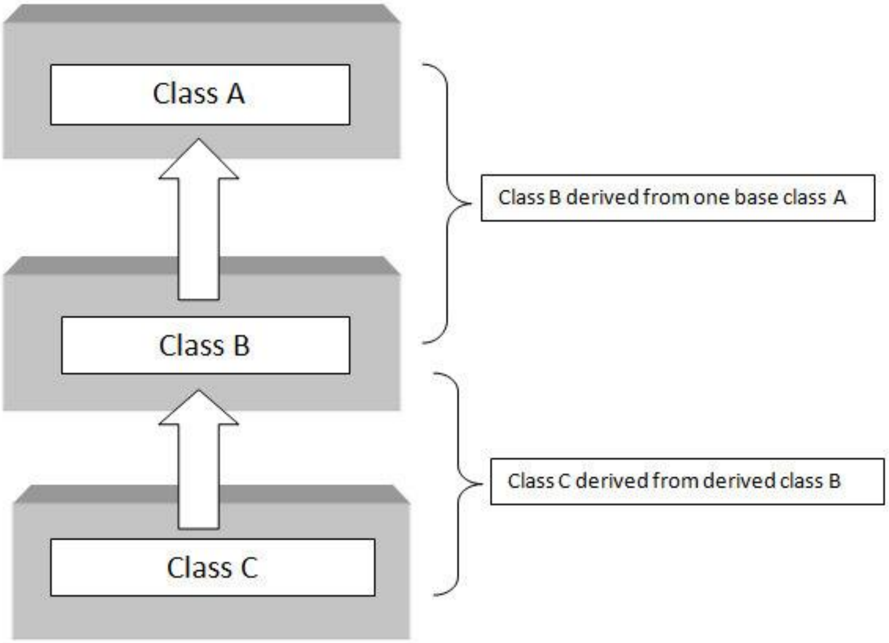
43. a) Explain multilevel inheritances with example.

When a class is derived from a class which is also derived from another class, i.e. a class having more than one parent classes, such inheritance is called Multilevel Inheritance. The level of inheritance can be extended to any number of level depending upon the relation. Multilevel inheritance is similar to relation between grandfather, father and child.

Syntax:

```
class base_classname
```

```
{
   properties;
   methods;
};

class intermediate_classname:visibility_mode base_classname
{
   properties;
   methods;
};

class child_classname:visibility_mode intermediate_classname
{
   properties;
   methods;
};
```


Class A
Class B
Class C
Class B derived from one base class A
Class C derived from derived class B

```
#include <iostream>
#include <conio.h>
using namespace std;

class person
{
    char name[100];
    public:
        void getdata()
        {
            cout<<"Name: ";
            fflush(stdin);
            gets(name);
        }
        void display()
        {
            cout<<"Name: "<<name<<endl;
        }
};

class employee: public person
{
    float salary;
    public:
        void getdata()
        {
            person::getdata();
            cout<<"Salary: Rs.";
            cin>>salary;
        }
        void display()
        {
            person::display();
            cout<<"Salary: Rs."<<salary<<endl;
        }
};

class programmer: public employee
{
    int number;
    public:
        void getdata()
        {
            employee::getdata();
            cout<<"Phone number : ";
            cin>>number;
        }
        void display()
        {
            employee::display();
            cout<<"Phone number is : "<<number;
        }
};

int main()
{
    programmer p;
    p.getdata();
    p.display();
    getch();
    return 0;
}
```

43. b) Explain pure virtual function with syntax and example.

- A pure virtual function is a function which has no definition in the base class. Its definition lies only in the derived class i.e it is compulsory for the derived class to provide definition of a pure virtual function. Since there is no definition in the base class, these functions can be equated to zero.
- The general form of pure virtual function is :

  ```
  virtual type func-name(parameter-list) = 0;
  ```

- Consider the following example of base class Shape and classes derived from it viz Circle, Rectangle, Triangle etc.

Ex:

```
class Shape
{
   int x, y;
   public:
       virtual void draw() = 0;
};
```

```
class Circle: public Shape
{
  public:
      draw()
      {
        //Code for drawing a circle
      }
};
class Rectangle: public Shape
{
  Public:
      void draw()
      {
        //Code for drawing a rectangle
      }
};
class Triangle: public Shape
{
  Public:
      void draw()
      {
        //Code for drawing a triangle
      }
};
```

Thus, base class Shape has pure virtual function draw(); which is overridden by all the derived classes.

44. a) Discuss about how to open and close files in C++.

Opening of files can be achieved in the following two ways :

- Using the constructor function of the stream class.
- Using the function open().

The first method is preferred when a single file is used with a stream.

Opening File Using Constructors

We know that a constructor of class initializes an object of its class when it is being created. Same way, the constructors of stream classes (ifstream, ofstream, or fstream) are used to initialize file stream objects with the filenames passed to them. This is carried out as explained here:

To open a file named myfile as an input file, we shall create a file stream object of input type i.e., ifstream type.

Example:

```
ifstream fin("myfile", ios::in) ;
```

The above given statement creates an object, fin, of input file stream. After creating the ifstream object fin, the file myfile is opened and attached to the input stream, fin. This stream object will be used using the getfrom operator (">>").

Example:

```
char ch;
fin >> ch ;
float amt ;
fin >> amt ;
```

When you want a program to write a file i.e., to open an output file.

- creating ofstream object to manage the output stream
- associating that object with a particular file

Example,

```
ofstream fout("secret" ios::out) ;
```

Now, to write something to it, you can use << (put to operator) in familiar way.

Example,

```
int code = 2193 ;
fout << code << "xyz" ;
```

The connections with a file are closed automatically when the input and the output stream objects expires i.e., when they go out of scope. Also, we can close a connection with a file explicitly by using the close() method :

- fin.close() ;
- fout.close() ;

Closing such a connection does not eliminate the stream; it just disconnects it from the file.

Opening Files Using Open() Function

There may be situations requiring a program to open more than one file. If the situation requires simultaneous processing of two files, then you need to create a separate stream for each file. However, if the situation demands sequential processing of files then you can open a single stream and associate it with each file in turn. To use this approach, declare a stream object without initializing it, then use a second statement to associate the stream with a file.

For example,

```
ifstream fin;
fin.open("Master.dat", ios::in);
fin.close();
fin.open("Tran.dat", ios::in);
fin.close();
```

Closing a File in C++

A file is closed by disconnecting it with the stream it is associated with. The close() function accomplishes this task and it takes the following general form :

```
stream_object.close();
```

For example, if a file Master is connected with an ofstream object fout.

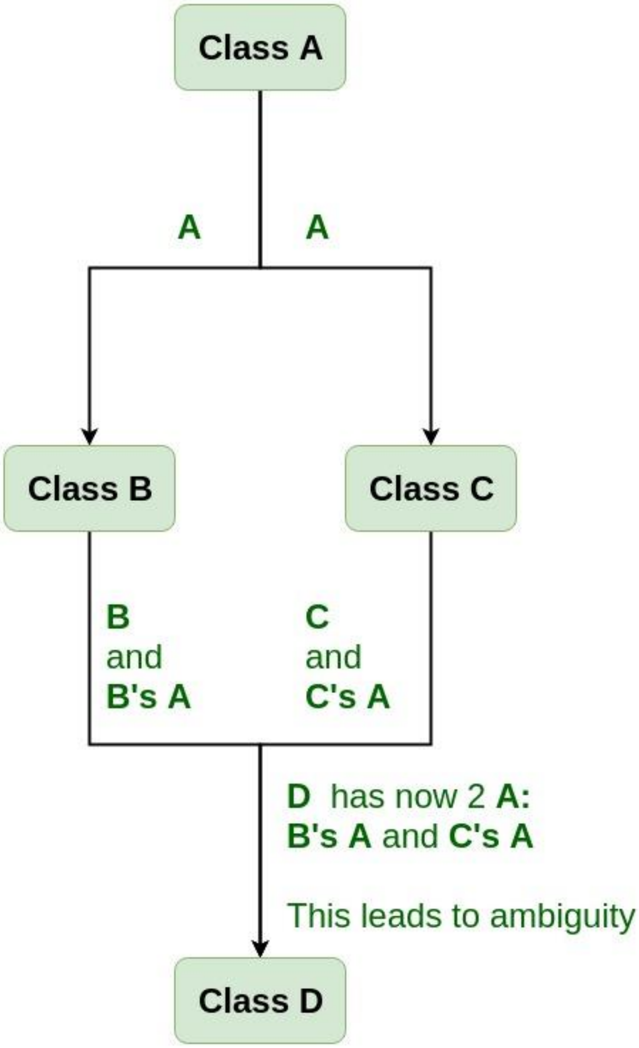
```
fout.close() ;
```

44. b) Discuss briefly about virtual base class with example.

Virtual base classes are used in virtual inheritance in a way of preventing multiple "instances" of a given class appearing in an inheritance hierarchy when using multiple inheritances.

Need for Virtual Base Classes:

Consider the situation where we have one class **A** .This class is **A** is inherited by two other classes **B** and **C**. Both these class are inherited into another in a new class **D** as shown in figure below.



As we can see from the figure that data members/function of class **A** are inherited twice to class **D**. One through class **B** and second through class **C**. When any data / function member of class **A** is accessed by an object of class **D**, ambiguity arises as to which data/function member would be called? One inherited through **B** or the other inherited through **C**. This confuses compiler and it displays error.

Example :

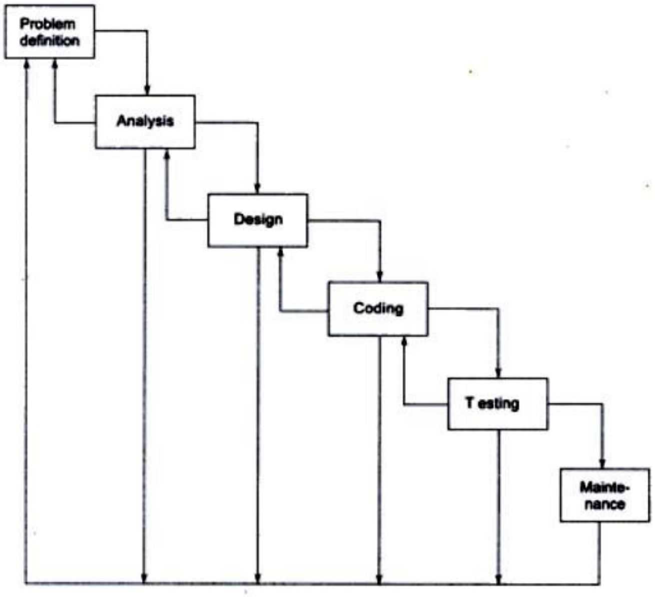
```
#include <iostream>
using namespace std;
class A
{
public:
        int a;
        A() // constructor
        {
                a = 10;
        }
};

class B : public virtual A
{
};

class C : public virtual A
{
};

class D : public B, public C
{
};

int main()
{
        D object; // object creation of class d
        cout << "a = " << object.a << endl;
        return 0;
}
```

45. a) Explain about procedure oriented paradigm.

Software development is usually categorized by a series of stages depicting the various tasks involved in the development process.

The classic life cycle model is based on an underlying model, commonly referred to as the “Water-fall” Model. This model attempts to break up the identifiable activities into series of actions, each of which must be completed before the next begins. The activities include

- Problem definition
- Requirement analysis
- Design
- Coding
- Testing
- And Maintenance



Further refinements to this model include iteration back to the previous stages in order to incorporate any changes or missing links.

Problem Definition: This activity requires a precise definition of the problem in user terms. A clear statement of the problem is crucial to the success of the software. It helps not only the developer but also the user to understand the problem better.

Analysis: this covers a detailed study of both the user and software. This activity is basically concerned with what of the system such as

- What are the inputs to the system?
- What are the processes required?
- What are the outputs expected?
- What are the constraints?

Design: the design phase with various concepts of system design such as data structure, software architecture, and algorithms, this phase translates the requirements into a representation of the software. This stage answers the questions of how.

Coding : Coding refers to the translation of the design into machine – readable form. The more detailed the design, the easier is the coding, and better its reliability.

Testing: after the software has been installed, it may undergo some changes, this may occur due to a change in the users requirements, a change in the operating environment, or an error in the software that has not been fixed during the testing. Maintenance ensures that these changes are incorporated wherever necessary.

Each phase of the life cycle has its own goals and outputs. The output of one phase acts as an input to the next phase.

45. b) Write a C++ program to compare two strings.

```
#include<iostream.h>
#include<string.h>
using namespace std;
int main ()
{
    char str1[50], str2[50];
    cout<<"Enter string 1 : ";
    gets(str1);
    cout<<"Enter string 2 : ";
    gets(str2);
    if(strcmp(str1, str2)==0)
        cout << "Strings are equal!";
    else
        cout << "Strings are not equal.";
    return 0;
}
```

Output:

Case 1 :

```
Enter string 1 : hi
Enter string 2 : hello
Strings are not equal.
```

Case 2 :

```
Enter string 1 : 12345
Enter string 2 : 12345
Strings are equal!
```

PART – C

46. Describe the various control structures in C++ with suitable examples.

Control Statements, Looping and Iteration

(i) Conditional structure: if and else
(ii) For Loop
(iii) While Loop
(iv) Do While
(v) Switch Statement and Break

(i) Conditional structure: if and else

- The if statement executes based test expression inside the braces.
- If statement expression is to true, If body statements are executed and Else body statements are skipped.
- If statement expression is to false If body statements are skipped and Else body statements are executed.
- Simply, Block will execute based on If the condition is true or not.

if and else Syntax

```
if (expression) // Body will execute if expression is true or non-zero
{
        //If Body statements
}else
{
        //Else Body statements
}
```

Ex:

```
        if (i == 3)
 {
                doSomething();
        }
        else
```

```
{
                doSomethingElse();
        }
```

Ex:

```
#include<iostream>
#include<conio.h>
using namespace std;
int main()
{
        int a;

        cout<<"Enter the Number :";
        cin>>a;
        if(a > 10)
        {
          cout<<a <<" Is Greater than 10";
        }
        else
        {
          cout<<a<<" Is Less than/Equal to 10";
        }
        getch();
        return 0;
}
```

The for statement

The C++ for statement has the following form:

Syntax:

```
for  (expression1;Condition;expression2)

        statement;

for  (expression1;Condition;expression2) {

        block of statements

}
```

expression1 initialises; expression2 is the terminate test; expression3 is the modifier;

Ex:

```
#include<iostream>
#include<conio.h>
using namespace std;
int main()
{
   int x=3;
         for (x=3;x>0;x--)
         {
                  cout<<"x="<<x<<endl;
         }
    getch();
    return 0;
}
```

The while statement

The while statement is similar to those used in other languages although more can be done with the expression statement.

Syntax:

```
while (expression)
         statement;

while (expression)
         block of statements
}
```

Ex:

```
#include<iostream>

#include<conio.h>

using namespace std;

//Main Function

int main()

{

    int x=3;

            while (x>0)

            {

                        cout<<"x="<<x<<endl;

                        x--;

            }

     getch();

     return 0;

}
```

The do-while statement

Syntax:

```
do

{

   statement;

}

while (expression);
```

Ex:

```
#include<iostream>
```

```
#include<conio.h>

using namespace std;

int main()

{

    int x=3;

             do {

               cout<<"x="<<x<<endl;

               x--;

             }while (x>0);

     getch();

     return 0;

}
```

Switch Case Statement

- We can use switch statements alternative for an if..else ladder.
- The switch statement is often faster than nested if...else Ladder.
- Switch statement syntax is well structured and easy to understand.

Syntax :

```
            switch ( <expression> or <variable> )
             {
            case value1:
              break;

            case value2:
              break;
            ...
            default:
              Code to execute for not match case
```

```
  break;
 }
```

47. What is Class in C++? Explain with syntax and example program.

Class:

The building block of C++ that leads to Object Oriented programming is a Class. It is a user defined data type, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class. A class is like a blueprint for an object.

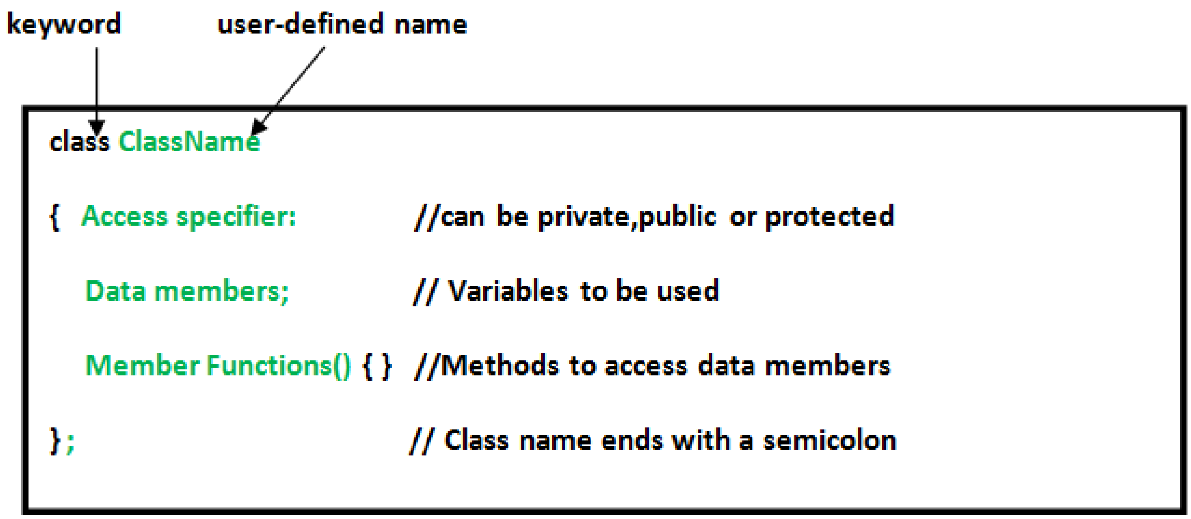
For Example:

Consider the Class of Cars. There may be many cars with different names and brand but all of them will share some common properties like all of them will have 4 wheels, Speed Limit, Mileage range etc. So here, Car is the class and wheels, speed limits, mileage are their properties.

A Class is a user defined data-type which has data members and member functions. Data members are the data variables and member functions are the functions used to manipulate these variables and together these data members and member functions defines the properties and behavior of the objects in a Class.

In the above example of class Car, the data member will be speed limit, mileage etc and member functions can be apply brakes, increase speed etc.

A class is defined in C++ using keyword class followed by the name of class. The body of class is defined inside the curly brackets and terminated by a semicolon at the end.



```
class ClassName

{   Access specifier:          //can be private,public or protected

    Data members;              // Variables to be used

    Member Functions() { }   //Methods to access data members

};                             // Class name ends with a semicolon
```

Accessing Data Members

The public data members are also accessed in the same way given however the private data members are not allowed to be accessed directly by the object. Accessing a data member depends solely on the access control of that data member.

This access control is given by Access modifiers in C++. There are three access modifiers :

- public,
- private and
- protected.

```
#include <iostream.h>
using namespace std;
class test
{
        public:
        string myname;
        void printname()
        {
        cout << "My name is: " << myname;
        }
};
int main() {

        test obj1;
        obj1.myname = "ram";
        obj1.printname();
        return 0;
}
```

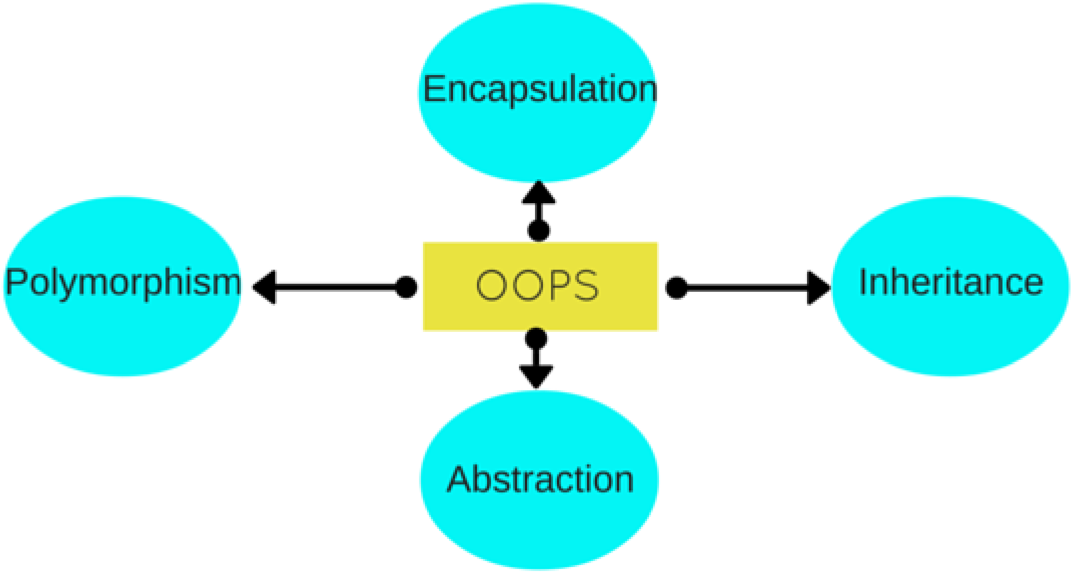
Output:

```
My name is : ram
```

48. Explain OOPs concept with its benefits.

Object Oriented Programming

Object Oriented programming is a programming style that is associated with the concept of Class, Objects and various other concepts revolving around these two, like Inheritance, Polymorphism, Abstraction, Encapsulation etc.



There are a few principle concepts that form the foundation of object-oriented programming –

Object

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

Class

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

Abstraction

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

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

Encapsulation

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

Inheritance

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

Polymorphism

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

Overloading

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

Benefits of OOPs:

1. **Data Re-usability**:- “Write a once and use multiple time” we can achieve this by using class. Once We Write a class we can bus it number of time by creating the object for class.
2. **Data Redundancy**:- Inheritance is the good feature for data redundancy if you need a same functionality in multiple class you can write a common class for the same functionality and inherit that class to sub class.
3. **Easy Maintenance**:- It is easy to maintain and modify existing code as new objects can be created with small differences to existing ones.
4. **Data hiding**:- Implementation details are hidden from other modules and other modules has a clearly defined interface.
5. **Security**:- Using data hiding and abstraction we are providing necessary data only it mean we are maintaining security.

49. What is friend function? Explain the friend function with example and also list its characteristics.

Definition :

A friend function of a class is defined outside that class' scope but it has the right to access all private and protected members of the class. Even though the prototypes for friend functions appear in the class definition, friends are not member functions.

A friend function can be:

a) A method of another class

b) A global function

Characteristics :

- A friend function is not in the scope of the class, in which it has been declared as friend.
- It cannot be called using the object of that class.
- It can be invoked like a normal function without any object.
- Unlike member functions, it cannot use the member names directly.
- It can be declared in public or private part without affecting its meaning.
- Usually, it has objects as arguments.

Ex:

//program to find the greatest number among two numbers using the friend function

```
#include <iostream.h>
#include <conio.h>
class Num
{
     int x,y;
  public:
    void num()
    {
        cout<<"Enter the first number = "<<endl;
       cin>>x;
       cout<<"Enter the second number = "<<endl;
       cin>>y;
       cout<<"The entered numbers are "<< x <<" and "<<y<<endl;
    }
    friend void max(Num m);
};
void max(Num m)
    {
       if(m.x>m.y)
      {
         cout<<"The greatest number among "<<m.x<<" and "<<m.y<<" is 
"<<m.x<<endl;
      }
      else
      {
         cout<<"The greatest number among "<<m.x<<" and "<<m.y<<" is 
"<<m.y<<endl;
      }
    }
void main()
{
   Num a;
  a.num();
  max(a);
  getch();
}
```

Output :

```
        Enter the first number =3
        Enter the second number =5
        The greatest number among 3 and 5 is 5
```

50. Explain the concept of function in C++ with syntax and example.

Function :

In C++, a function is a group of statements that is given a name, and which can be called from some point of the program. The most common syntax to define a function is:

type name (parameter1, parameter2, ...) { statements }

Where:

- type is the type of the value returned by the function.
- name is the identifier by which the function can be called.
- parameters (as many as needed): Each parameter consists of a type followed by an identifier, with each parameter being separated from the next by a comma.

Ex:

```
#include <iostream>

using namespace std;

int max(int num1, int num2);

int main ()

{

  int a = 100;

  int b = 200;

  int ret;

  ret = max(a, b);

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

  return 0;

}

int max(int num1, int num2)

{

  int result;

  if (num1 > num2)
```

```
      result = num1;
   else
      result = num2;

   return result;
}
```

Functions with no type. The use of void

Syntax:

```
type name ( argument1, argument2 ...) { statements }
```

Requires the declaration to begin with a type.

Ex:

```
#include <iostream>
using namespace std;
void printmessage ()
{
  cout << "I'm a function!";
}
int main ()
{
  printmessage ();
}
```

The return value of main

The return type of main is int, but most examples in this and earlier chapters did not actually return any value from main.

If the execution of main ends normally without encountering a return statement the compiler assumes the function ends with an implicit return statement:

```
return 0;
```

Arguments passed by value and by reference

Arguments have always been passed by value. This means that, when calling a function, what is passed to the function are the values of these arguments on the moment of the call, which are copied into the variables represented by the function parameters. For example

```
include <iostream>

using namespace std;

void duplicate (int& a, int& b, int& c)

{

  a*=2;

  b*=2;

  c*=2;

}

int main ()

{

  int x=1, y=3, z=7;

  duplicate (x, y, z);

  cout << "x=" << x << ", y=" << y << ", z=" << z;

  return 0;

}
```

const references

Calling a function with parameters taken by value causes copies of the values to be made. This is a relatively inexpensive operation for fundamental types such as int, but if the parameter is of a large compound type, it may result on certain overhead. For example, consider the following function:

```
string concatenate (string a, string b)
{
  return a+b;
}
```

This function takes two strings as parameters (by value), and returns the result of concatenating them. By passing the arguments by value, the function forces a and b to be copies of the arguments passed to the function when it is called. And if these are long strings, it may mean copying large quantities of data just for the function call.

But this copy can be avoided altogether if both parameters are made references:

```
string concatenate (string& a, string& b)
{
  return a+b;
}
```

Inline function

Defining member function is to replace the function declaration by the actual function definition inside the class. It is treated as inline function.

The inline functions are a C++ enhancement feature to increase the execution time of a program. Functions can be instructed to compiler to make them inline so that compiler can replace those function definition wherever those are being called. What is meant by Constructors and list various types of constructors.

For example, the concatenate function above may be declared inline as:

```
inline string concatenate (const string& a, const string& b)
{
  return a+b;
}
```

51. Describe the various types of constructors in C++.

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.

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.

Types of Constructors:

- ➢ Default constructor
- ➢ Parameterized constructor
- ➢ Copy constructor
- ➢ Destructor

Default Constructors:

Default constructor is the constructor which doesn’t take any argument. It has no parameters.

Ex:

```
#include <iostream>
using namespace std;
class construct
 {
public:
    int a, b;
    construct()
    {
        a = 10;
```

```
            b = 20;
    }
};
int main()
{
    construct c;
    cout << "a: " << c.a << endl
            << "b: " << c.b;
    return 1;
}
```

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.

Ex:

```
#include <iostream>
using namespace std;
class Point
 {
private:
    int x, y;
public:
    Point(int x1, int y1)
    {
            x = x1;
            y = y1;
    }
    int getX()
    {
            return x;
    }
    int getY()
    {
            return y;
    }
};
int main()
{
    Point p1(10, 15);
    cout << "p1.x = " << p1.getX() << ", p1.y = " << p1.getY();
    return 0;
}
```

Copy Constructor:

A copy constructor is a member function which initializes an object using another object of the same class. A copy constructor has the following general function prototype:

ClassName (const ClassName &old_obj);

Ex:

```
#include<iostream>
using namespace std;
class Point
{
private:
    int x, y;
public:
    Point(int x1, int y1)
 {
 x = x1;
 y = y1;
}
Point(const Point &p2)
{
x = p2.x;
 y = p2.y;
}
     int getX()
      {
 return x;
 }
     int getY()
      {
 return y;
 }
};
int main()
{
Point p1(10, 15); // Normal constructor is called here
Point p2 = p1; // Copy constructor is called here
cout << "p1.x = " << p1.getX() << ", p1.y = " << p1.getY();
cout << "\np2.x = " << p2.getX() << ", p2.y = " << p2.getY();
return 0;
}
```

Destructor:

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

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

Ex:

```
#include<iostream>
#include<conio.h>
using namespace std;
class BaseClass
{
public:
  BaseClass()
 {
    cout << "Constructor of the BaseClass : Object Created"<<endl;
  }
  ~BaseClass()
{
    cout << "Destructor of the BaseClass : Object Destroyed"<<endl;
  }
};
int main ()
{
    BaseClass des;
    return 0;
}
```

52. What are abstract classes? Explain their uses.

C++ Abstract Class

Abstract class is used in situation, when we have partial set of implementation of methods in a class. For example, consider a class have four methods. Out of four methods, we have an implementation of two methods and we need derived class to implement other two methods. In these kind of situations, we should use abstract class.

The purpose of an abstract class (often referred to as an ABC) is to provide an appropriate base class from which other classes can inherit. Abstract classes cannot be used to instantiate objects and serves only as an interface. Attempting to instantiate an object of an abstract class causes a compilation error.

A virtual function will become pure virtual function when you append "=0" at the end of declaration of virtual function.

A class with at least one pure virtual function or abstract function is called abstract class.

Characteristics of Abstract Class

- Abstract class cannot be instantiated, but pointers and refrences of Abstract class type can be created.
- Abstract class can have normal functions and variables along with a pure virtual function. Abstract classes are mainly used for upcasting, so that its derived classes can use its interface.
- Classes inheriting an Abstract Class must implement all pure virtual functions, or else they will become Abstract too.

We can't create an object of abstract class because it has partial implementation of methods.

Example:

```
#include<iostream.h>
#include<conio.h>
class BaseClass       //Abstract class
{
       public:
        virtual void Display1()=0;     //Pure virtual function or abstract function
        virtual void Display2()=0;     //Pure virtual function or abstract function
        void Display3()
        {
              cout<<"\n\tThis is Display3() method of Base Class";
        }
};
class DerivedClass : public BaseClass
{
       public:
        void Display1()
        {
              cout<<"\n\tThis is Display1() method of Derived Class";
        }

        void Display2()
        {
              cout<<"\n\tThis is Display2() method of Derived Class";
        }
};
void main()
{
        DerivedClass D;
        D.Display1();           // This will invoke Display1() method of Derived Class
        D.Display2();           // This will invoke Display2() method of Derived Class
        D.Display3();           // This will invoke Display3() method of Base Class
    }
```

Output :

```
        This is Display1() method of Derived Class

        This is Display2() method of Derived Class

        This is Display3() method of Base Class
```

53. What is exception handling? Explain the basics of exception handling with example.

Exceptions allow a method to react to exceptional circumstances and errors (like runtime errors) within programs by transferring control to special functions called handlers. For catching exceptions, a portion of code is placed under exception inspection. Exception handling technology offers a securely integrated approach to avoid the unusual predictable problems that arise while executing a program.

There are two types of exceptions:

- Synchronous exceptions
- Asynchronous exceptions

Errors such as: out of range index and overflow fall under the category of synchronous type exceptions. Those errors that are caused by events beyond the control of the program are called asynchronous exceptions.

The main motive of the exceptional handling concept is to provide a means to detect and report an exception so that appropriate action can be taken. This mechanism needs a separate error handling code that performs the following tasks:

- Find and hit the problem (exception)
- Inform that the error has occurred (throw exception)
- Receive the error information (Catch the exception)
- Take corrective actions (handle exception)

The error handling mechanism basically consists of two parts. These are:

- To detect errors
- To throw exceptions and then take appropriate actions

Exception handling in C++ is built on three keywords:

- Try
- catch and
- throw.

try

throw: A program throws an exception when a problem is detected which is done using a keyword "throw".

catch: A program catches an exception with an exception handler where programmers want to handle the anomaly. The keyword catch is used for catching exceptions.

The Catch blocks catching exceptions must immediately follow the try block that throws an exception. The general form of these two blocks is as follows:

```
try

{

   throw exception;

}

catch(type arg)

{

   //some code

}
```

If the try block throws an exception then program control leaves the block and enters into the catch statement of the catch block. If the type of object thrown matches the argument type in the catch statement, the catch block is executed for handling the exception. Divided-by-zero is a common form of exception generally occurred in arithmetic based programs.

Ex:

```
#include<iostream>
using namespace std;
double division(int var1, int var2)
{
   if (var2 == 0)
{
      throw "Division by Zero.";
   }
   return (var1 / var2);
}
int main()
{
   int a = 30;
   int b = 0;
   double d = 0;
   try
{
      d = division(a, b);
      cout << d << endl;
   }
   catch (const char* error)
{
      cout << error << endl;
   }
   return 0;
}
```

Output :

```
        Division by zero
```

54. **Explain the phases in object oriented development.**

We know that the Object-Oriented Modelling (OOM) technique visualizes things in an application by using models organized around objects. Any software development approach goes through the following stages −

- Analysis,
- Design, and
- Implementation.

In object-oriented software engineering, the software developer identifies and organizes the application in terms of object-oriented concepts, prior to their final representation in any specific programming language or software tools.

Phases in Object-Oriented Software Development

The major phases of software development using object–oriented methodology are object-oriented analysis, object-oriented design, and object-oriented implementation.

Object–Oriented Analysis

In this stage, the problem is formulated, user requirements are identified, and then a model is built based upon real–world objects. The analysis produces models on how the desired system should function and how it must be developed. The models do not include any implementation details so that it can be understood and examined by any non–technical application expert.

Object–Oriented Design

Object-oriented design includes two main stages, namely, system design and object design.

System Design

In this stage, the complete architecture of the desired system is designed. The system is conceived as a set of interacting subsystems that in turn is composed of a hierarchy of interacting objects, grouped into classes. System design is done according to both the system analysis model and the proposed system architecture. Here, the emphasis is on the objects comprising the system rather than the processes in the system.

Object Design

In this phase, a design model is developed based on both the models developed in the system analysis phase and the architecture designed in the system design phase. All the classes required are identified. The designer decides whether −

- new classes are to be created from scratch,
- any existing classes can be used in their original form, or
- new classes should be inherited from the existing classes.

Class: A class is a collection of similar objects. It is a template where certain basic characteristics of a set of objects are defined. The class defines the basic attributes and the operations of the objects of that type. Defining a class does not define any object, but it only creates a template. For objects to be actually created instances of the class are created as per the requirement of the case.

Abstraction: Classes are built on the basis of abstraction, where a set of similar objects are observed and their common characteristics are listed. Of all these, the characteristics of concern to the system under observation are picked up and the class definition is made. The attributes of no concern to the system are left out. This is known as abstraction.

The abstraction of an object varies according to its application. For instance, while defining a pen class for a stationery shop, the attributes of concern might be the pen color, ink color, pen type etc., whereas a pen class for a manufacturing firm would be containing the other dimensions of the pen like its diameter, its shape and size etc.

Inheritance: Inheritance is another important concept in this regard. This concept is used to apply the idea of reusability of the objects. A new type of class can be defined using a similar existing class with a few new features. For instance, a class vehicle can be defined with the basic functionality of any vehicle and a new class called car can be derived out of it with a few modifications. This would save the developers time and effort as the classes already existing are reused without much change.

The associations between the identified classes are established and the hierarchies of classes are identified. Besides, the developer designs the internal details of the classes and their associations, i.e., the data structure for each attribute and the algorithms for the operations.

Object–Oriented Implementation and Testing

In this stage, the design model developed in the object design is translated into code in an appropriate programming language or software tool. The databases are created and the specific hardware requirements are ascertained. Once the code is in shape, it is tested using specialized techniques to identify and remove the errors in the code. There are all the benefits of using the Object Orientation. Some of these are:

Reusability - The classes once defined can easily be used by other applications. This is achieved by defining classes and putting them into a library of classes where all the classes are maintained for future use. Whenever a new class is needed the programmer looks into the library of classes and if it is available, it can be picked up directly from there.

Inheritance - The concept of inheritance helps the programmer use the existing code in another way, where making small additions to the existing classes can quickly create new classes.

Data Hiding - Encapsulation is a technique that allows the programmer to hide the internal functioning of the objects from the users of the objects. Encapsulation separates the internal functioning of the object from the external functioning thus providing the user flexibility to change the external behaviour of the object making the programmer code safe against the changes made by the user.

55. Describe the role of keywords try, catch and throw in exception handling.

Exceptions provide a way to transfer control from one part of a program to another. C++ exception handling is built upon three keywords: try, catch, and throw.

try − A try block identifies a block of code for which particular exceptions will be activated. It's followed by one or more catch blocks.

Syntax of try statement

```
try
{
statement 1;
statement 2;
}
```

throw − A program throws an exception when a problem shows up. This is done using a throw keyword.

Syntax of throw statement

```
throw (excep);
throw excep;
throw // re-throwing of an exception
```

catch − A program catches an exception with an exception handler at the place in a program where you want to handle the problem. The catch keyword indicates the catching of an exception.

Syntax of catch statement

```
try
{
Statement 1;
Statement 2;
}
```

```
catch ( argument)
{
statement 3; // Action to be taken
}
```

When an exception is found, the catch block is executed. The catch statement contains an argument of exception type, and it is optional. When an argument is declared, the argument can be used in the catch block. After the execution of the catch block, the statements inside the blocks are executed. In case no exception is caught, the catch block is ignored, and if a mismatch is found, the program is terminated.

Ex:

```
#include <iostream>
using namespace std;
double division(int a, int b)
 {
  if( b == 0 )
{
    throw "Division by zero condition!";
  }
  return (a/b);
}
int main ()
 {
  int x = 50;
  int y = 0;
  double z = 0;
  try
{
    z = division(x, y);
    cout << z << endl;
  }
 catch (const char* msg)
{
    cerr << msg << endl;
  }   return 0;
}
```

Result : Division by zero condition!

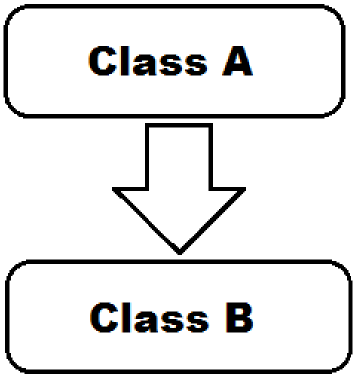
56. Discuss about different types of inheritance with example.

There are different types of inheritance:

1. Single Inheritance
2. Multiple Inheritance
3. Multilevel Inheritance
4. Hierarchical Inheritance
5. Hybrid (Virtual) Inheritance

1. Single Inheritance

Single inheritance represents a form of inheritance when there is only one base class and one derived class.

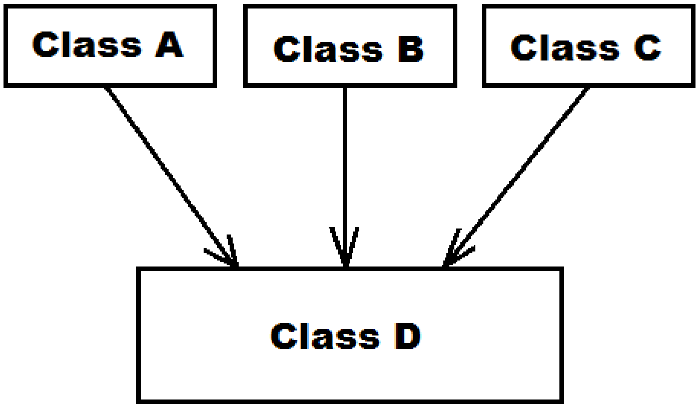


Ex:

```
#include <iostream>
using namespace std;
class base
{
  public:
    int x;
  void getdata()
  {
   cin >> x;
  }
 };
class derive : public base
 {
  private:
   int y;
  public:
  void readdata()
  {
   cin >> y;
  }
  void product()
  {
    cout << x * y;
  }
 };

 int main()
 {
   derive a;
   a.getdata();
   a.readdata();
   a.product();
   return 0;
 }
```

2. Multiple Inheritance

Multiple inheritance represents a kind of inheritance when a derived class inherits properties of **multiple** classes. For example, there are three classes A, B and C and derived class is D as shown below:

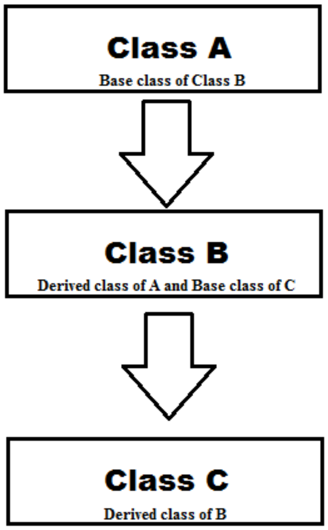


If you want to create a class with multiple base classes, you have to use following syntax:

Class DerivedClass: accessSpecifier BaseClass1, BaseClass2, …, BaseClassN

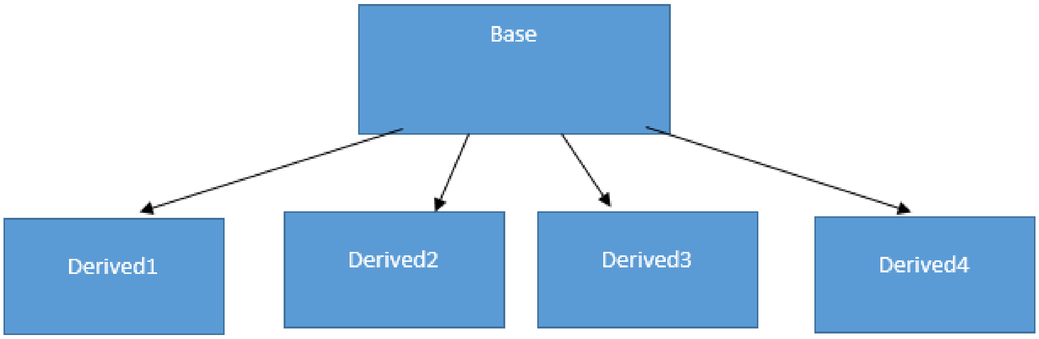
3. Multilevel Inheritance

Multilevel inheritance represents a type of inheritance when a Derived class is a base class for another class. In other words, deriving a class from a derived class is known as multi-level inheritance. Simple multi-level inheritance is shown in below image where Class A is a parent of Class B and Class B is a parent of Class C



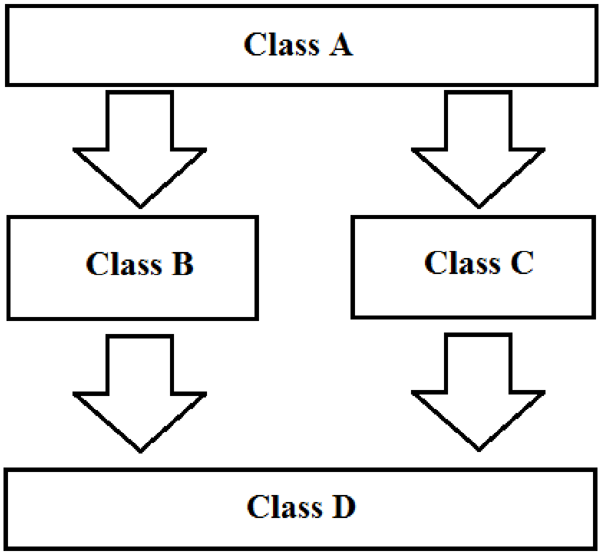
4. Hierarchical Inheritance

When there is a need to create multiple Derived classes that inherit properties of the same Base class is known as Hierarchical inheritance.



5. Hybrid (Virtual) Inheritance

Combination of Multi-level and Hierarchical inheritance will give you Hybrid inheritance.



57. Describe about manipulating strings in C++.

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

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

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

```
#include <iostream>
#include <cstring>
using namespace std;
int main () {
char str1[10] = "Hello";
char str2[10] = "World";
char str3[10];
int  len ;
strcpy( str3, str1);
cout << "strcpy( str3, str1) : " << str3 << endl;
strcat( str1, str2);
cout << "strcat( str1, str2): " << str1 << endl;
len = strlen(str1);
cout << "strlen(str1) : " << len << endl;
return 0;
}
```

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

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

The String Class in C++

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

```
#include <iostream>
#include <string>
using namespace std;
int main ()
{
  string str1 = "Hello";
  string str2 = "World";
  string str3;
  int  len ;
  str3 = str1;
  cout << "str3 : " << str3 << endl;
```

```
   str3 = str1 + str2;
   cout << "str1 + str2 : " << str3 << endl;
   len = str3.size();
   cout << "str3.size() :  " << len << endl;
   return 0;
 }
```

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

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

58. What is pointer to a function? Explain the situations in which such a concept is used.

In normal function call (call by value), the parameters of a function are xerox copies of the arguments passed to the function. It is like we are passing xerox copies. So altering them won't affect the real values.

But in call by referance, we pass the address of variables to the function. Passing address is like passing original 'x' and 'y'. Altering the parameters will alter the real values also.

Ex:

swap two numbers i.e., interchange the values of two numbers.

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

Output :

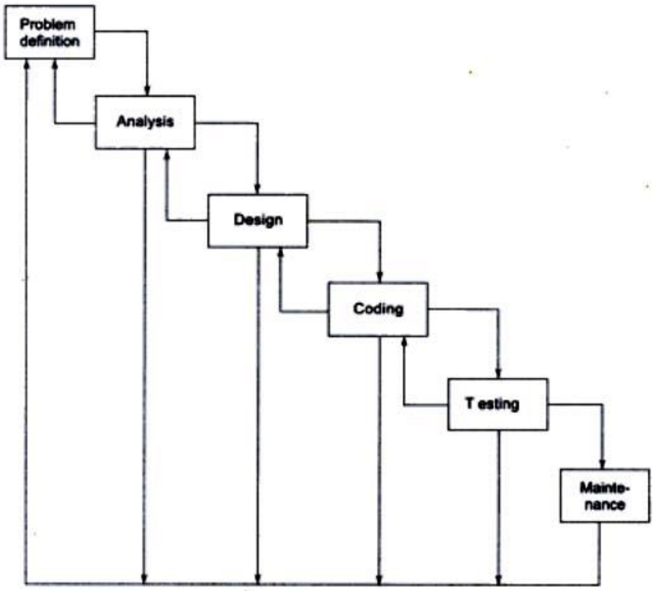
```
Enter first number
2
Enter second number
4
First number = 4
Second number = 2
```

59. Discuss in detail procedure oriented Paradigms and Object Oriented Paradigms.

Software development is usually categorized by a series of stages depicting the various tasks involved in the development process.

The classic life cycle model is based on an underlying model, commonly referred to as the "Water-fall" Model. This model attempts to break up the identifiable activities into series of actions, each of which must be completed before the next begins. The activities include

- Problem definition
- Requirement analysis
- Design
- Coding
- Testing
- And Maintenance



Further refinements to this model include iteration back to the previous stages in order to incorporate any changes or missing links.

Problem Definition: This activity requires a precise definition of the problem in user terms. A clear statement of the problem is crucial to the success of the software. It helps not only the developer but also the user to understand the problem better.

Analysis: this covers a detailed study of both the user and software. This activity is basically concerned with what of the system such as

- What are the inputs to the system?
- What are the processes required?

- What are the outputs expected?
- What are the constraints?

Design: the design phase with various concepts of system design such as data structure, software architecture, and algorithms, this phase translates the requirements into a representation of the software. This stage answers the questions of how.

Coding : Coding refers to the translation of the design into machine – readable form. The more detailed the design, the easier is the coding, and better its reliability.

Testing: after the software has been installed, it may undergo some changes, this may occur due to a change in the users requirements, a change in the operating environment, or an error in the software that has not been fixed during the testing. Maintenance ensures that these changes are incorporated wherever necessary.

Each phase of the life cycle has its own goals and outputs. The output of one phase acts as an input to the next phase.

Object Oriented Paradigm

Object-oriented programming (OOP) is a programming paradigm based upon objects (having both data and methods) that aims to incorporate the advantages of modularity and reusability. Objects, which are usually instances of classes, are used to interact with one another to design applications and computer programs.

The important features of object–oriented programming are −

- Bottom–up approach in program design
- Programs organized around objects, grouped in classes
- Focus on data with methods to operate upon object's data
- Interaction between objects through functions
- Reusability of design through creation of new classes by adding features to existing classes

Some examples of object-oriented programming languages are C++, Java, Smalltalk, Delphi, C#, Perl, Python, Ruby, and PHP.

60. **Discuss about sequential input and output operations on files in C++.**

The file stream classes support a number of member functions for performing the input and output operations on files. The functions get() and put() are capable of handling a single character at a time. The function getline() lets you handle multiple characters at a time. Another pair of functions i.e., read() and write() are capable of reading and writing blocks of binary data.

The get(), getline() and put() Functions

The functions get() and put() are byte-oriented. That is, get() will read a byte of data and put() will write a byte of data. The get() has many forms, but the most commonly used version is shown here, along with put() :

```
    istream & get(char & ch) ;

    ostream & put(char ch) ;
```

The get() function reads a single character from the associated stream and puts that value in ch. It returns a reference to the stream. The put() writes the value of ch to the stream and returns a reference to the stream.

The following program displays the contents of a file on the screen. It uses the get() function :

```
/* C++ Sequential Input/Output Operations on Files */

#include<iostream.h>

#include<stdlib.h>

#include<fstream.h>

#include<conio.h>

void main()

{

    char fname[20], ch;

    ifstream fin;      // create an input stream

    clrscr();

    cout<<"Enter the name of the file: ";

    cin.get(fname, 20);

    cin.get(ch);
```

```
        fin.open(fname, ios::in);
        if(!fin)
        {
          cout<<"Error occurred in opening the file..!!\n";
          cout<<"Press any key to exit...\n";
          getch();
          exit(1);
        }
        while(fin)
        {
          fin.get(ch);
          cout<<ch;
        }
        cout<<"\nPress any key to exit...\n";
        fin.close();
        getch();
}
```

when the end-of-file is reached, the stream associated with the file becomes zero. Therefore, when fin reaches the end of the file, it will be zero causing the while loop to stop.

Other Forms of get() Function

In addition to the form shown earlier, there are several other forms of get() function. Two most commonly used form shave the following prototypes :

istream & get(char ∗ buf, int num, char delim = '\n') ;

int get() ;

The getline() Function

Another member function that performs input is getline() whose prototype is :

```
istream & getline(char * buf, int num, char delim = '\n') ;
```

The function getline() also reads characters from input stream and puts them in the array pointed to by buf until either num character have been read, or the character specified by delim is encountered. If not mentioned, the default value of delim is newline character.

The read() and write() Functions

Another way of reading and writing blocks of binary data is to use C++'s read() and write() functions. Their prototypes are :

```
istream & read( (char *) & buf, int sizeof(buf) ) ;

ostream & write( (char *) & buf, int sizeof(buf) ) ;
```

The read() function reads sizeof(buf) (it can be any other integer value also) bytes from the associated stream and puts them in the buffer pointed to by buf. The write() function writes sizeof(buf) (it can be any other integer value also) bytes to the associated stream from the buffer pointed to by buf.