BHARATH COLLEGE OF SCIENCE AND MANAGEMENT, THANJAVUR-5

PG DEPARTMENT OF COMPUTER SCIENCE

Title of the Paper	:	**PROGRAMMING IN C++**
Subject code	:	**16SCCCS2**
Compiled by	:	1. R.Murugesan Assistant Professor, Department of Computer Science, Bharath College of Science and Management, Thanjavur-5 2. U. Gomathi Assistant Professor, Department of Computer Science, Bharath College of Science and Management, Thanjavur-5

CORE COURSE – V

PROGRAMMING IN C++

Unit I

Basic Concepts of Object- Oriented Programming - Benefits of OOP - Object Oriented Languages - Applications of OOP – Structure of C++ Program - Tokens, Expressions and Control Structures – Functions in C++

Unit II

Classes and Objects – Constructors and Destructors – New Operator – Operator Overloading and Type Conversions

Unit III

Inheritance: Extending Classes – Pointers- Virtual Functions and Polymorphism

Unit IV

Managing Console I/O Operations – Working with Files – Templates – Exception Handling

Unit V

Standard Template Library – Manipulating Strings – Object Oriented Systems Development

Text Book

1. Balagursamy E - "Object Oriented Programming with C++", Tata McGraw Hill Publications,2006 Third edition.,[Unit-1 (Chapters - 1, 2, 3, 4) ; Unit-2 (Chapters – 5, 6,7 ,16) ; Unit-3 (Chapters – 8.9)Unit-4 (Chapters – 10, 11, 12, 13); Unit-5 (Chapters – 14, 15, 16, 17)]

Reference Books

1. Barbara Johnston, C++ Programming today, Pearson education/Prentice-Hall of India, ISBN 81-317-1079-3, 2007.

2. Steve Oualline, Practical C++ programming, O'Reilly/Shroff publishers & distributors, ISBN 81-7366-682-2.

Prepared by

R.Murugesan & U.Gomathi
AP, Dept of CS
Bharath College of Science and Managment
Thanjavur-5

CONTENTS

Prepared by

R.Murugesan & U. Gomathi
AP, Dept of CS
Bharath College of Science and Management.
Thanjavur-5

UNIT-I
2 Marks:

1. Define object oriented programming.

Object oriented Programming is defined as an approach that provides a way of modularizing programs by creating partitioned memory area for both data and functions that can be used as templates for creating copies of such modules on demand. Writing object-oriented programs involves creating classes, creating objects from those classes, and creating applications, which are stand-alone executable programs that use those objects.

1. What isC++ Programming Language?

C++ is objected oriented programming language, It is used to develop the application and system program.C++ was invented in 1979 by Bjarne Stroustrup at Bell Laboratories in New Jersey.

2. What are Basic Data types in C++?.

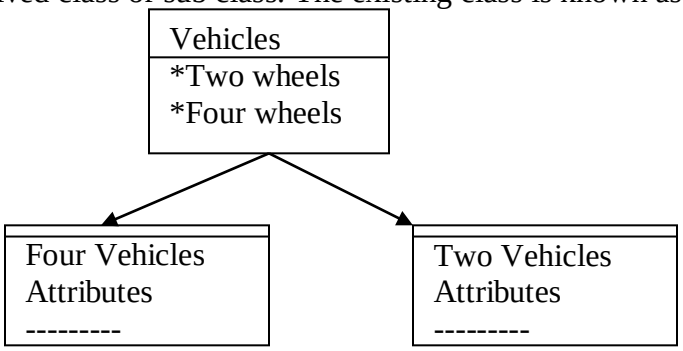
Type	Size	Type	Size
Int	2 bytes	Unsigned int	2 bytes
Char	1 bytes	Signed int	2 bytes
Float	4 bytes	Shor int	2 bytes
Double	8 bytes	Singed and unsigned char	1 bytes

3. What is Data Abstraction?

Data Abstraction is defined as a collection of Data and functions. Since the classes use the concept of data abstraction. They are known as Abstract Data Types(ADT). The only way to access the data is provided by the functions, which are wrapped in the class. Data is not accessible to the outside world.

4. What is Inheritance:

Inheritance is the process by which object of one class to acquire properties of objects of another class. The new class will have the combined feature of both the classes. The new class is know as derived class or sub class. The existing class is known as base class.



5. What is Polymorphism?

Polymorphism is the ability for a message or data to be processed in more than one from. That is the ability of a function or operator to act in different ways on different data types.

6.. What are the Benefits of OOP

The principle advantage of oops are:

1. Through inheritance, we can eliminate redundant code and extend the use of existing classes.
2. We can build programs from the standard working modules that communicate with one another, rather than having to start writing the code from scratch .This leads to saving of development time and higher productivity.
3. The principle of data hiding helps the programmer to build secure programs that Cannot be invested by code another parts of the programs.

It is possible to have a multiple object to co-exist without any interface.

5. It is possible to map objects in the problem domain to those objects in the programs.
6. It is easy to partition the work in a project based on objects.
7. Object oriented system can be easily upgraded from small to large system.
   a. Message passing techniques for communication between object makes the interface description with external system much simpler.
8. Software complexity can be easily managed.

7. What are the Application of oops Programming?

The Areas for application of oop includes

- Real-time system.
- Simulation and modeling
- Object-oriented data base.
- Hypertext ,hypermedia
- AI and expert system
- Neural networks and parallel programming
- Decision support and office automation system.
- CIM/CAD/CAM. System.

8. What are the Components of a Class?

A Class consists of two components

1. Data Members
2. Methods or Member Functions

9. What are data members and Member Functions?

Data member: The variables declared within a class , this variable is called data member..

Member function: the function declared or defined within a class, this functions is called member function. It is used to handle data members of a class.

11. What do you mean by Private member?

A member declared as Private is a Private member. We cannot access private members anywhere in the program. Private members **can be accessed only through Member function of that class.**

Private:

int k1,k2;

12. What are the Access Specifiers Used in a Class?

The Access Specifiers

1. Public Specifier
2. Private Specifier
3. Protected Specifier

13. What is Data Hiding?

The private data members can not be accessed outside of class only access within the class. This process is call data hiding.

.

Descriptive Question answer:

1. Basic Concepts of Object-Oriented Programming

These include:

- Objects
- Classes
- Data abstraction and encapsulation
- Inheritance
- Polymorphism
- Dynamic binding
- Message passing

i.Objects:

Objects are the basic run-time entities in an object oriented programming.

- It is an instance of class.
- Each instance of an object can hold its own relevant data.
- They may represent a person, a place, a bank account or any item that the program has to handle.
- Each object contains data (data members) and code (member functions) to manipulatethe data.

Object: STUDENT
DATA: Name Roll No Mark 1, Mark 2, Mark
FUNCTIONS Total Average Display

ii. Classes:

It is a representation (or) blueprint of an object.

- The entire set of data and code of an object can made a user defined data type with the help of a class.
- Classes are user-defined data types and behave like built-in data types of Programming language and they are known as Abstract Data Type.

For Example: If Fruit has been declared as a class, then the statement

Fruit mango;

will create an object **mango** belonging to the class **Fruit**

iii. Data Abstraction and Encapsulation:

Encapsulation:

The wrapping up of data and functions into a single unit is known as encapsulation**.**

- 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 interface b/n the object's data and the program. This insulation of the data from direct access by the program is called *data hiding or information hiding*.

Abstraction:

It represents the act of representing the essential features without including the background details or explanations.

- Classes use the concept of abstraction and are defined as a list of abstract attributes such as size, weight and cost and functions to operate on these attributes.
- The attributes are called data members and functions are called member functions or methods.

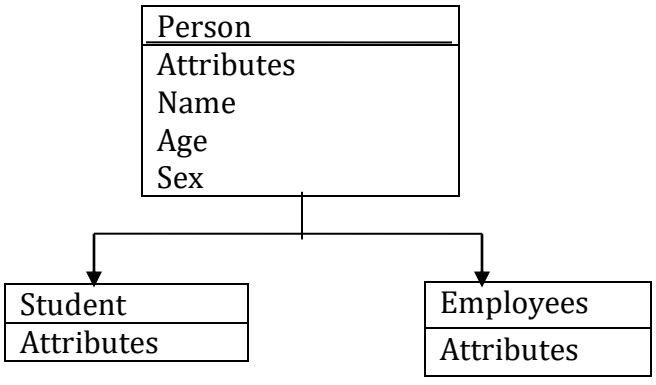
Since the classes use the concept of data abstraction, they are known as **Abstract Data Types (ADT).**

iv. Inheritance:

It is the process by which objects of one class acquire the properties of objects of another class.

- It supports the concept of hierarchical classification. For example, the 'student' and the 'employee' is a part of the class 'person'.

  This concept provides the idea of *reusability*. This means that we can add additional features to an existing class without modifying it.



Types of Inheritance

- Single inheritance

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

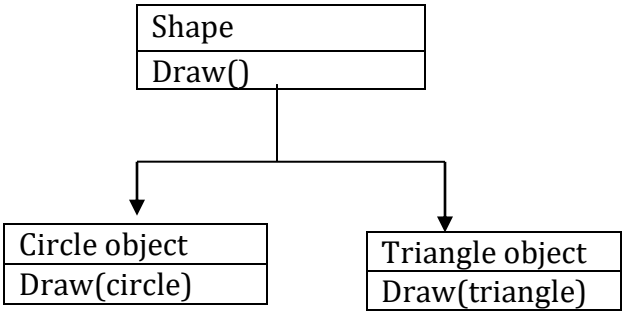
Adv:

- It offers code reusability.
- It is a technique of design and code sharing.

v. Polymorphism:

It means the ability to take more than one form. An operation may exhibit different behaviors in different instances. The behavior depends upon the types of data used in the operation.

- For example the operation addition will generate sum if the operands are numbers whereas if the operands are strings then the operation would produce a third string by concatenation. The process of making an operator to exhibit different behaviors in different instances is known as *operator overloading.*
- A single function name can be used to handle different types of tasks based on the number and types of arguments. This is known as *function overloading.*



vi. Dynamic Binding:

Binding refers to the linking of a procedure call to the code to be executed in response to the call. *Dynamic Binding (late binding)* means the code associated with the given procedure call is not known until the time of the call at run-time.

vii. Message Passing:

The process of programming in OOP involves the following basic steps:

- Creating classes that define objects and their behavior
- Creating objects from class definitions
- Establishing communication among objects

A *message* for an object is request for execution of a procedure and therefore will invoke a function (procedure) in the receiving object that generates the desired result. *Message passing* nvolves specifying the name of the object, the name of the function (message) and the information to be sent.

E.g.: employee.salary(name);

Object : employee
Message: salary
Information: name

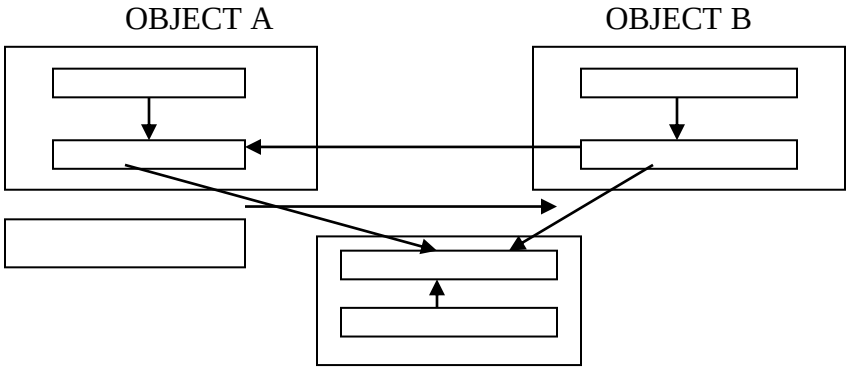
2. Object _oriented programming paradigm:-

The fundamental idea behind OOP is to combine both data and functions that operate on that data in to a single unit. Such a unit is called an object.

A OOP program consists of a number of objects. The data of an object can be accessed bit function associated with that objects. How functions of one access the functions of other objects.

Features of OOP:-

- Emphasis is on data rather than procedure.
- Programs are divided into as objects.
- Data structures are designed such that they character the objects.
- Functions that operate on the data of an object are tied together in the data structure.
- Data In hidden and cannot be accessed by the external function.
- Object may communicate with each other through function.
- New data and functions can be easily added whenever necessary.
- Follows bottom up approach in program design.



3. structure of C++ program :

Include Files
Class Definition
Member Function Definitions
Main Function

i) Include file

- In C++ number of functions, classes and variables are defined and stored in a special file called **header file**.
- There are so many header files available.
- Each header file has related functions, classes and data.
- If we want to use the predefined functions, classes and variables and data the appropriate header file should be included at the beginning of the program. E.g.: iostream.h, string.h, conio.h

General form: `#include<header_file_name.>`

ii) Class Definition

It is a user defined data type having defined functions and data.

```
Class test
  {
    }
```

iii) Member function definition

Member function can be defined inside or outside the class.

```
Void add()
        (
        }
```

iv) Main function

The execution of the program begins from void main() function. The parentheses following the word main are the distinguishing feature of the function, without the parentheses the compiler would think that main referred to a variable (or) some other program element. Where void proceeding the function main () indicates that this particular function does not have a return value.

```
. void main()
        {
        }
```

4. C++ Token

The smallest individual units in a program are called as token. C++ has the following tokens:

i.)Variable:

- Variables are the basic unit of storage in a program. It is treated as an identifier to the memory location.
- A value in a variable can change during the program execution.
- **Declaration:** type identifier; (Or) type identifier = value;
  Where type specifies the data type and identifier specify the name of the variable.
- **Example:** int a, b; float f = 12.45;

ii) Key word:

- Key word is a predefined word. Keywords can't be used as names for the program, variable or other user-defined program elements.
- **Example:** int, float, char, public, private, while, do, if, switch, return, class, etc...

iii) Identifier:

- Identifier is a name given by the programmer.
- It refers to the names of variables, functions, arrays, classes etc… created by the programmer.

Rules for Creating Identifier:

- Only alphabetic characters, digits and underscores area permitted.
- Identifier names can't starts with digits.
- Both Uppercase and Lowercase letters are allowed.
- Keywords can't be used as a variable name.

iv) Constant:

- Referred to the fixed value, that does not change during the execution of a program.
- Symbolic Constant: Any value declared as 'const' can't be modified by the program.
- Example: const float Pi = 3.1413;
- Literal Constant:

  Integer Ex: int A = 123;
  Floating point Ex: float F = 12.56;
  Characters Ex: char Ch = 'S';
  String Ex: char Ch [20] = "WELCOME";

5..C++ Operators:

a. **Arithmetic operator**
b. **Relation operator**
c. **logical operator**
d. **Assignment operator**
e. **Conditional operator**
f. Increment and Decrement Operators

An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations. C++ is rich in built-in operators. Generally, there are six type of operators: Arithmetical operators, Relational operators, Logical operators, Assignment operators, Conditional operators, Comma operator.

a) Arithmetical operators

Arithmetical operators +, -, *, /, and % are used to performs an arithmetic (numeric) operation.

Operator Meaning

+ Addition
- Subtraction
***** Multiplication
/ Division
% Modulus

You can use the operators +, -, *, and / with both integral and floating-point data types. Modulus or remainder % operator is used only with the integral data type.

b) Relational operators

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

Relational Operators

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

c) Logical operators

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

Operators Meaning

|| OR
&& AND
! NOT

d)Assignment operator

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

m = 5;

The operator takes the expression on the right, 5, and stores it in the variable on the left, m.

x = y = z = 32;

This code stores the value 32 in each of the three variables x, y, and z. In addition to standard assignment operator shown above, C++ also support compound assignment operators.

Compound Assignment Operators

Operator Example Equivalent to

+ = A + = 2 A = A + 2
- = A - = 2 A = A – 2
% = A % = 2 A = A % 2
/= A/ = 2 A = A / 2
***=** A * = 2 A = A * 2

e) Increment and Decrement Operators

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

f) Conditional operator

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

Conditional_ expression ? expression1 : expression2;

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

The comma operator

The comma operator gives left to right evaluation of expressions. When the set of expressions has to be evaluated for a value, only the rightmost expression is considered.

int a = 1, b = 2, c = 3, i; // comma acts as separator, not as an operator

i = (a, b); // stores b into i would first assign the value of a to i, and then assign value of b to variable i.

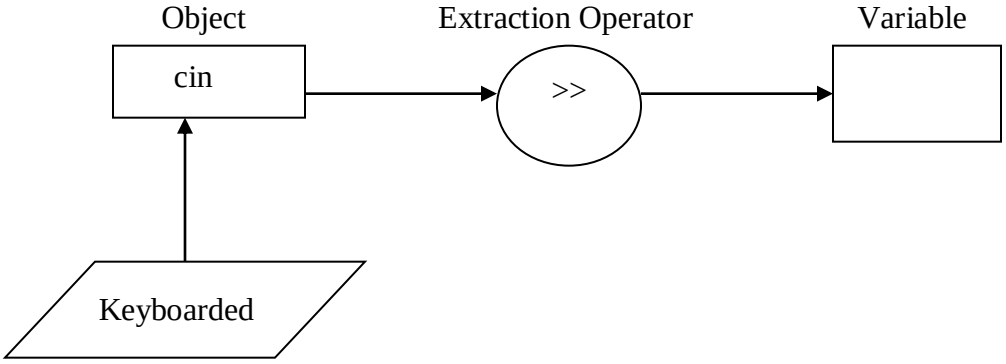
So, at the end, variable i would contain the value 2.

6. Input/ Output Operations in C++ (Cout & cin)

Input Operator:

cin>>variable;

The statement is an input statement and causes the program to wait for the user to type a data. Cin(Console Input) is an object in C++ and the operator >> is known as **extraction** or **get from operator**.



Cascading Input Operator:

When more than one extraction operators are used in a Cin statement , which is known **as cascading Input operator.**

Example:

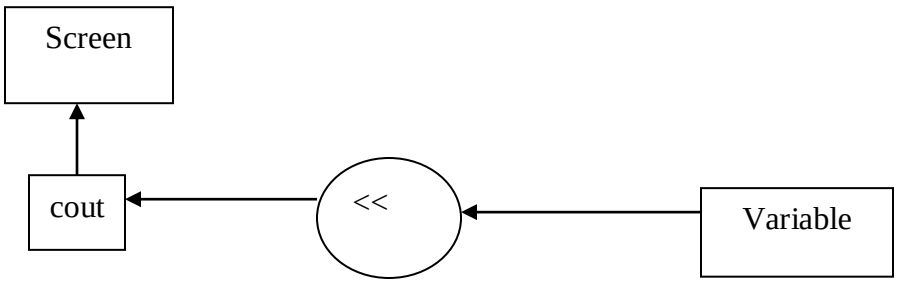
Cin>>a1>>a2>>a3;

Output Operator:

The output operator supported in C++ is

Cout<<variable ;

This operator prints the value of the variable or string on the screen.



Insertion Operator

The operator << is known as insertion operator. Cout is c++ predefined object.

Example:

Cout<<" The result is ="<<a1;

7. Data types

Data type defines size and type of values that a variable can store along with the set of operations thatcan be performed on that variable. C++ provides built-in data types that correspond to integers, characters, floating-point values, and Boolean values. There are the seven basic data types in C++ as shown below:

Data Type Names	Description	Size	Range
`char`	Single text character or small integer. Indicated with single quotes (`'a'`, `'3'`).	1 byte	signed: -128 to 127 unsigned: 0 to 255
`int`	Larger integer.	4 bytes	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
`bool`	Boolean (true/false). Indicated with the keywords `true` and `false`.	1 byte	Just `true` (1)or `false` (0).
`double`	"Doubly" precise floating point number.	8 bytes	+/-1.7e +/-308 (15 digits)

Type Meaning

a) char(character) holds 8-bit ASCII characters
b) int (Integer) represent integer numbers having no fractional part
c) float (floating point) stores real numbers in the range of about 3.4x10–38to 3.4x10 38,with a precision of seven digits.
d) Double (Double floating point) Stores real numbers in the range from 1.7x10 –308 to1.7x10308 with a precision of 15 digits.
e) bool(Boolean) can have only two possible values: true and false.
f) Void Valueless

C++ allows certain of the basic types to have modifiers preceding them. A modifier alters the meaning of the base type so that it more precisely fits the needs of various situations. The data type modifiers are: signed, unsigned, long and short This figure shows all combinations of the basic data types and modifiers along with their size and range for a 16-bit word machine

Variable

A variable is a named area in memory used to store values during program execution. Variables are run time entities. A variable has a symbolic name and can be given a variety of values. When a variable is given a value, that value is actually placed in the memory space assigned to the variable. All variables must be declared before they can be used. The general form of a declaration is:

Data type variable_list;

Here, type must be a valid data type plus any modifiers, and variable_list may consist of one or more identifier names separated by commas. Here are some declarations:

```
int i,j,l;
short int si;
unsigned int ui;
 double balance, profit, loss;
```

8. Control Structures (Statements)

a) Condition statements

b) Loop statements

a)Conditional statements

This statements check the condition first and then execute the block statements. They have two statements:

If Statement

Switch Statements

Simple If Statement.

This statements check the condition first, if the condition is ture , then execute the block statements. If false, exit from if statement.

Syntax: If(condition)

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

- Example **Program:**

```
main()
{       int i;
        printf("\nEnter the number....");
        scanf("%d",&i);
        if(i<=50)
        printf("\nThe entered number is < 50");
}
```

if.. Else Statement.

This statements check the condition , if the condition is ture , then execute the one block statements. If false, another block of statement.

Syntax: If(condition)

```
{   statement(s)-I;
 }
else
 {   statement(s)-II;
 }
```

Switch Statements

Select a block of statements to execute from the several block of statements base on the condition

Syntax:

```
switch(Exp) {
        case 1:
                statements-I;
                break;
        case 2:
```

```
            statements-II;
            break;
        default:
            statements-III;
            break;
        }
```

Example

```
    main()
    {   int a=1;
        switch(a
        {
        case 1:
            printf("Case 1 \n");
            break;
        case 2:
            printf("Case 2 Selected\n");
            break;
        default:
            printf("Default Case Selected\n");
            break;
    }
```

B) Loop Statement.

The loop statement execute the set of statements repeatedly until the condition is true. They are three statements.

- **For statement**
- **While statement**
- **Do ... while statement**

For statement

Syntax

```
For(initialization; condition; increment;)
{
Statements;
}
```

Example **Program:**

```
    main()
        {
        int i,sum=0;
        for(i=1;i<=10;i++)
            sum=sum+i;
```

```
printf("The addition of numbers upto 10 is %d", sum);
                }
```

While statement

- Syntax

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

- **Example Program:**

```
main ()
  {     int i=1,sum=0;
                while (i<=10)
                {
                sum=sum+i;
                        i++;              }
        printf("The sum of numbers upto 10 is %d",sum);
}
```

Do ... while statement

Syntax

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

- Example **Program:**

```
                main()
                   {  int i=1,sum=0;
                        do
                        {
                        sum=sum+i;
                        i++;
                        }
                        while(i<=10);
                printf("Sum of the numbers upto 10 is ...%d", sum);
}
```

9. REFERENCE VARIABLES

Reference variable is the alternate name (another name) given to an existing variable. It can be used as an argument to the copy constructor. Once a reference is initialized with a variable, either the variable name or the reference name may be used to refer to the variable.

- Reference variables are used to pass the parameter to the functions.
- Do not initialize reference variable with a constant value.
- Never return the reference variable from a function as a memory address.

- Do not assign reference variables to the variables whose memory allocated dynamically.

General form: Data type &ref-var-name = variable-name;

Example:

```
int y=10;
int &x=y;
```

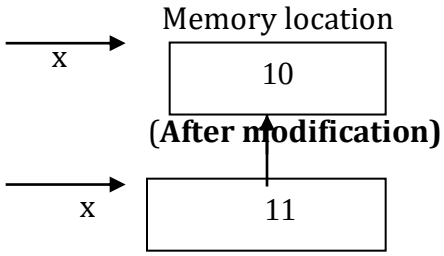
- In the above code 'y' is normally declared and 'x' is an alias for y.
- This means that both x and y share the same memory locations and any modifications made to any of them will reflect both the variables

For example

```
x++;
cout <<y;
```

The output will be 11.

(Before modification)



9.1 Scope Resolution operator

Member functions can be defined within the class definition or separately using scope resolution operator (::). Defining a member function within the class definition declares the function inline, even if you do not use the inline specifier. Defining a member function using scope resolution operator uses following declaration

```
return-type class-name::func-name(parameter- list)
{
// body of function
}
```

Here the class-name is the name of the class to which the function belongs. The scope resolution operator (::) tells the compiler that the function func-name belongs to the class class-name. That is, the scope of the function is restricted to the class-name specified.

```
Class myclass {
int a;
public:
void set_a(intnum); //member function declaration
int get_a( ); //member function declaration
};
//member function definition outside class using scope resolution operator
void myclass :: set_a(intnum)
{
a=num;
}
int myclass::get_a( ) {
```

return a;
}
Another use of scope resolution operator is to allow access to the global version of a variable. In many
situation, it happens that the name of global variable and the name of the local variable are same .In
this while accessing the variable, the priority is given to the local variable by the compiler. If we want
to access or use the global variable, then the scope resolution operator (::) is used. The syntax for accessing a global variable using scope resolution operator is as follows:-

: Global-variable-name

---

10. C++ Functions

A function is subprogram that contain set of statements. It perform a specific task.. It can be invoked from other parts of the program. Functions help to reduce the program size when same set of instructions are to be executed again and again.

Function declaration — prototype:

A function has to be declared before using it, in a manner similar to variables and constants. A function declaration tells the compiler about a function's name, return type, and parameters and how to call the function.

The general form :

return_type function_name(parameter list);

Function definition

The function definition is the actual body of the function. The function definition consists of two parts namely, function header and function body.

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

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

Here, **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.

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 a Function

To use a function, you will have to call or invoke that 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.

Example program

A c++ program calculating factorial of a number using functions

```
#include<iostream.h>
#include<conio.h>
int factorial(int n);        //function declaration
int main(){
int no, f;
cout<<"enter the positive number:-";
```

```
cin>>no;
f=factorial(no); //function call
cout<<"\nThe factorial of a number"<<no<<"is"<<f;
return 0;
}
int factorial(int n)          //function definition
{ int i , fact=1;
 for(i=1;i<=n;i++)
{  fact=fact*i;
}
return fact; }
```

9. Inline Functions

An inline function is a function that is expanded inline at the point at which it is invoked, instead of actually being called. when a function is expanded inline, none of those operations occur. Although expanding function calls in line can produce faster run times, it can also result in larger code size because of duplicated code.

A function can be defined as an inline function by prefixing the keyword inline to the function header as given below:

```
inline function header
{
function body
}
```

Example:

```
// A program illustrating inline function
#include<iostream.h>
#include<conio.h>
inline int max(int x, int y){
if(x>y)
return x;
else
return y;
}
int main( ) {
int a,b;
cout<<"enter two numbers";
cin>>a>>b;
cout << "The max is: " <<max(a,,b) << endl;
return 0;
  }
```

11.Call by reference and Call by value

a) Call by reference

Arguments can be passed to functions in one of two ways: using call-by-value or call-by-reference. When using call-by-value, a copy of the argument is passed to the function. Call-by-reference passes the address of the argument to the function. By default, C++ uses call-by-value. Provision of the reference variables in c++ permits us to pass parameter to the functions by reference.

When we pass arguments by reference, the formal arguments in the called function become aliases to the actual arguments in the calling function. This means that when the function is working with its own arguments, it is actually working on the original data.

Example

```
#include <iostream.h>
#include<conio.h>
void swap(int &x, int &y); // function declaration
int main (){
int a = 10, b=20;
cout << "Before swapping”<<endl;
cout<< “value of a :" << a <<” value of b :" << b << endl;
swap(a, b); //calling a function to swap the values.
cout << "After swapping”<<endl;
cout<<” value of a :" << a<< “value of b :" << b << endl;
return 0;
}
void swap(int &x, int &y) { //function definition to swap the values.
int temp;
temp = x;
x = y;
y = temp;
}
```

Output:

Before swapping value of a:10 value of b:20
After swapping value of a:20 value of b:10

b)Call by Value

The **call by value** method of passing arguments to a function copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.

By default, C++ uses call by value to pass arguments. In general, this means that code within a function cannot alter the arguments used to call the function. Consider the function **swap()** definition as follows.

```
// function definition to swap the values.
void swap(int x, int y)
{
int temp;
temp = x; /* save the value of x */
x = y; /* put y into x */
y = temp; /* put x into y */
return;
}
```

12.)Function Overloading

Functions having the same name but with different signature is known as function overloading.

- A signature is a combination of a function name and its parameters.
- Parameter is differentiated by

1. Number of arguments
   Sum(int, int);
   Sum(int, int, int);

2. Type of arguments
   Square(int);
   Square(double);
3. Order of arguments
   Sum(int, float);
   Sum(float, int);

- It is commonly used to perform similar task, but on different data types.
- For example many functions in math library are overloaded for different numeric data types.

Overloaded square functions:

The below example uses overloaded square functions to calculate the square of an int and the square of an double.

Example:

//Overloaded function

```
#include <iostream>
int square( int x)              //function square for int values

{
        Cout<< " Square of integer" << x << " is;
        return x*x;
}
                                //function square for double values
double square( double y)
{
        Cout<< " Square of double" << y << " is;
        return y*y;
}                               // end function square with double argument
int main()
{
        cout<< square(7); // calls int version
        cout<< endl;
        cout<< square(7.5); // calls double version
        cout<< endl;
        return 0;
}                       // end main
```

OUTPUT:

Square of integer 7 is 49
Square of double 7.5 is 56.25

13. Difference between C & C++ Language.

Sl. No	C Programming	C++ Programming
1	C Language is not object oriented	C++ is object oriented
2	C program may be executed in C++ Compiler	C++ program cannot be executed in C Compiler.
3.	When function takes no parameters Void is given.	When function takes no parameter Void is optional.
4.	Function prototype is optional in C	Functional prototype is must in C++
5.	Function return type is optional in C	Function return type is must in C++
6.	Local variables can be declared only at the start of a program.	Local variables can be declared anywhere before they are used.
7.	C is not supporting Boolean data type.	C++ supports Boolean data type.
8.	C is not supporting Class data type.	C++ supports Class data type.
9.	Data hiding is not possible in C	Data hiding is possible in C++
10.	Data are not secure.	Data are more secure.
11.	There is a limitation on the length of identifiers	C++ does not put any limit on the identifiers.
12.	C does not support reference variables	C++ support reference variables
13.	C does not support Scope resolution operator.	C++ Supports scope resolution operator.
14.	Printf() in C requires type of variables to be printed.	Cout in C++ does not require type of variables to be printed.
15.	C does not support passing default value to function parameter,	C++ has a facility of passing default values to function parameter.

***********END*************

UNIT-II

2 Marks:

1.What is Operator Overloading?

If a operator name performs different operation in different situation , such operator is known as Operator Overloading.

For example, + Operator performs addition between numeric data and perform concatenation between string data.

And "<<" is insertion operator when we use Cout operation and "<<" is also called Bitwise Operator.

2.What is Function Overloading?

When two or more function share the same name with different argument list are said to be function overloading.

A function call first matches the prototype having the same number of arguments and type of arguments.

Example:

average(int m1,int m2,intm3)
average(int ht1, int ht2)
average(Float a1,float a2, double a3)

3.What is Constructor function?

The constructor function is used to initialize the object of a class at the time of objects creation. The constructor function is called automatically when the object is created.

The constructor function has the ***same name of its class name*** and ***no return type***. It means,

1. Name of the constructor should be same as the name of the class.
2. The constructor cannot have any return type.
3. It is automatically invoked when the object is declared.

Example:

```
Class    Date
         { private:
            int   dd;
             int mm;
             int yy;
              public:
                   Date(); // constructor  function declared
              Void input();
       };
```

4.What is Destructor Function?

It is a member function of a class which is used to destroy an object. A destructor has a tilde character(~) followed by a class name. It is called automatically when the

```
Class    Date
         { private:
                int   dd;
                int mm;
```

```
            int yy;
public:
             Date(); // constructor  function declared
             ~Date();//  Destructor function declared
          Void input();
    };
```

5.What is Inline function?

Inline function is a small function that is commonly used with classes that will replace function call by actual function code. The Inline keyword tells the compiler to replace every function call in source code by actual code of that function.

6.What is Constructor Overloading?

Definition:

When a class has more than one constructor with same name are called constructor overloading.

Example:

```
Class Date
  {
    private:
      int dd;
      int mm;
      int yy;
    public:
      Date();
      Date(int dd1, int mm1);
      Date(int dd1, int mm1, int yy1);
    };
```

7.What is Copy Constructor?

A copy constructor is a special type of constructor which initializes all the data members of the newly created object by copying the contents of an existing object.

The syntax of calling a Copy Constructor is

Class-Name New Object(Existing Object);

Example:

Assume that we have a class called Student and object S1 of this class has been already defined. Then we can create new object S2 belonging to same class as

Student S2(S1);

\

Descriptive Questions.

1. Class and Object

a) Class

A class is a user defined data type. It is a template of an object. A class contain data member and member function.

When you define a class with declare the data and function. the data and function of a class are called members of the class. Thegeneral form of class s:

```
class class-name {
        access-specifier:
                Declaration data ;
                Declaration of functions
                // ...
                } object-list;
```

The object-list is optional. If present, it declares objects of the class. Here, access-specifier is one of these three C++ keywords:

public ,private, protected

Private member functions and data can not be accessed out side of class and only accessed within a class.

The public access_specifier allows functions or data to be accessible to other parts of your program. The protected access_specifier is needed only when
inheritance is involved.

Example:

```
#include<iostream.h>
#include<conio.h>
Class myclass
{               // class declaration
                // private members to myclass
int a;
public:
                        // public members to myclass
void set_a(intnum);
int get_a( );
};
```

b) Object

An object is a runtime entity in oops. An object is said to be an instance of a class. Defining an object is similar to defining a variable of any data type: Defining objects in this way means creating them. This is also called
instantiating them. Once a Class has been declared, we can create objects of that Class by using the class Name like any other built-in type variable as shown:

```
        className objectName
Example
void main( ) {
        myclass ob1, ob2;          //these are object of type myclass
    // ... program code
        }
```

Accessing Class Members

The main() cannot contain statements that access class members directly. Class members can be accessed only by an object of that class. To access class members, use the dot (.) operator. The dot operator links the name of an object with the name of a member. The general form of the dot operator is shown here:

object.member

Example

ob1.set_a(10);

The private members of a class cannot be accessed directly using the dot operator, but through the member functions of that class providing data hiding. A member function can call another member unction directly, without using the dot operator.

Example:

C++ program to find sum of two numbers using classes

```
#include<iostream.h>
#include<conio.h>
class A{
int a,b,c;
public:
void sum(){
cout<<"enter two numbers";
cin>>a>>b;
c=a+b;
cout<<"sum="<<c;
}
};
int main(){
A u;
u.sum();
getch();
return(0);
}
```

2. Define MEMBER FUNCTIONS of CLASSES

Class is a collection of object. The data and function is member of a class. Data is the attribute of the class and function is operation of the class. If a data is a part of a class then it is called data member and if a function is part of a class then it is called **member function**.

Defining member function

The member function of a class can be defined in two ways,

i) Inside of class Defining member function .

ii) Outside of class Defining member function .

i) Inside o f class Defining member function

Member function definition is placed inside the class definition. Dot(**.**) operator can be used to access the member function which is defined inside the class.

General Form

```
class classname
{
....
returntype functionname(args)
{
statements;
}
};
```

Example:

```
#include<iostream.h>
#include<conio.h>
class sum
{
private:
int a,b,c;
public:
void read() //member function
{
cout<<"Enter a: ";
cin>>a;
cout<<"Enter b: ";
cin>>b; }
void display()
{
c=a+b;
cout<<"A = "<<a;
cout<<"B = "<<b;
cout<<"C = "<<c;
}
};
void main()
{
clrscr();
sum s;
s.read(); //accessing member function
s.display();
getch();
}
```

ii) outside of class Defining member function

In this, member functions are declared inside the class definition and defined outside the class definition. In order to define member function outside the class, scope resolution operator (**::**) is used along with classname and function name.

General form:

```
returntype classname::functionname(args)
{
statements;
}
}
```

Example:

```
#include<iostream.h>
#include<conio.h>
class sum
{
private:
int a,b,c;
public:
void read(); //member function declaration
void display();
};
void sum::read() //member function definition
{
cout<<"Enter a: ";
cin>>a;
cout<<"Enter b: ";
cin>>b;
}
void sum::display()
{
c=a+b;
cout<<"A = "<<a;
cout<<"B = "<<b;
cout<<"C = "<<c;
}
void main()
{
clrscr();
sum s;
s.read(); //accessing member function
s.display();
getch();
}
```

3. Static Class Members

Static class members can be defined using **static** keyword. When we declare a member of a class as static it means no matter how many objects of the class are created, there is only one copy of the static member which is shared by all objects of the class.

A) Static data member b) Static member functions

a) Static data member

The member variables in a class can be declared as static. The general form is,

```
static datatype membervariable;
```

Characteristics:

- At once the object for the class are declared, the static member variables are initialized to zero.
- For each static variable separate memory locations are allocated and it is common for all objects in the class.
- Each static member variable must be defined outside the class using scope resolution operator.

```
datatype classname::membervariable;
```

- For static member variable we can give initial value.

```
datatype classname::membervariable=value;
```

Example:

```
#include<iostream.h>
#include<conio.h>
class sample
{
private:
static int st; //declaration
public:
void incre()
{
st++;
}
void print()
{
cout<<"\n Static = "<<st;
}
};
int sample::st; //definition
void main()
{
clrscr();
sample s1,s2;
s1.print();
s2.print();
s1.incre();
s2.incre();
s1.print();
s2.print();
getch();
}
```

b) Static member function

It can be declared using static. It can only access the static members declared in the same class. Static member function can be called using classname.

General form: classname::memberfunction;

Example:

```
#include<iostream.h>
#include<conio.h>
class sample
{
private:
static int st;
public:
void incre()
{
st++;
}
static void print()
{
cout<<"static = "<<st;
}
};
int sample::st;
void main()
{
clrcsr();
sample s1,s2;
sample::print();
s1.incre();
s2.incre();
sample::print();
getch();
}
```

4. Friend Function

In a class, private member variables can be accessed only by the member functions in that class. Friend function can be used to access all private and protected members of the class for which it is a friend. Friend function should be a non-member class.

General form: friend return_type function_name(args);

Characteristics:

- The keyword friend can only be used inside the class.
- More than one friend function can be declared in a class.
- A function can be friend to more than one class.

- Friend function definition should not contain the scope operator.

Example:

```
#include<iostream.h>
#include<conio.h>
class sample
{
private:
int a,b,c;
public:
void getdata();
friend void display(sample); //friend function declaration
};
void sample::getdata()
{
cin>>a>>b>>c;
}
void display(sample s)
{
cout<<s.a<<s.b<<s.c;
}
void main()
{
sample s;
s.getdata();
display(s);
getch();
}
```

5. Constructors

Constructor is a special member function with the same name as its class name. Constructor functions are invoked automatically when an object for a class is created.

Uses:

- It is used to initialize the object.
- It usually provides initial value for the data member of the object.

Syntax:

Inside the class

```
class classname
{
public:
classname(args)//constructor function
{
Statements;
}
};
```

Out side the class

```
class classname
{
public:
classname(args);
};
Classname::classname(args)
{
Statements;
}
```

Characteristics of Constructor:

- A constructor has the same name as the class itself.
- It is invoked automatically when the objects are created.
- It should be defined public.
- They can have default argument.
- It can be overload.
- It has no return type even void.
- We cannot refer to their address.
- They cannot be inherited.
- They cannot be virtual.

Types of Constructor

i) Default Constructor / Constructor with no argument
ii) Parameterized Constructor
iii) Multiple Constructor / Overloading Constructor
iv) Copy Constructor

i) Default Constructor (or) Constructor with no argument

A Constructor that accepts no parameter is called Default Constructor.

```
Constructor-name()
        {
        }
```

Example:

```
#include<iostream.h>
#include<conio.h> class sum
{
private: int a,b,c; public:
sum()          //Default Constructor
{
a=10;
b=10;
}
void add()
{
c=a+b;
cout<<c;
}};
void main()
{
sum s
s.add();
getch();
}
```

ii) Parameterized Constructor

The constructor that can take argument is called Parameterized Constructor. Initial values must be passed at the time of object creation. It can be done by two ways.

```
Constructor-name(para-list)
                {
                }
```

- By calling the constructor implicitly
- By calling the constructor explicitly

By calling the constructor implicitly	**By calling the constructor explicitly**

Constructor name has not been mentioned.
Syntax:
classname object(args);
Example:
#include<iostream.h>
#include<conio.h>
class data
{
private:
int a;
public:
data(int x)
{
a=x;
}
void print()
{
cout<<a;
}
};
void main()
{
data d1(10);
d1.print();
getch();
}
	Constructor name is explicitly provided.
	Syntax:
	classname object=constructorname(args);
	Example:
	#include<iostream.h>
	#include<conio.h>
	class data
	{
	private:
	int a;
	public:
	data(int x)
	{
	a=x;
	}
	void print()
	{
	cout<<a;
	}
	};
	void main()
	{
	data d1=data(10);
	d1.print();
	getch();
	}

iii) Multiple Constructor (or) Overloading Constructor

If a class has more than one constructor, then it is called Multiple Constructor.

Example:

```
#include<iostream.h>
#include<conio.h>
class time
{
private:
int hours;
int minutes
```

```
int seconds;
public:
time() // default constructor
{
hours=0;
minutes=0;
seconds=0;
}
time(int h) // one argument constructor
{
hours=h;
minutes=0;
seconds=0;
}
time(int h,int m) // two argument constructor
{
hours=h;
minutes=m;
seconds=0;
}
time(int h,int m,int s) // three argument constructor
{
hours=h;
minutes=m;
seconds=s;
}
void showtime()
{
cout<<hours;
cout<<minutes;
cout<<seconds;
}
};
void main()
{
clrscr();
time t1;
time t2(12);
time t3(12,15);
time t4(12,15,15);
t1.showtime();
t2.showtime();
t3.showtime();
t4.showtime();
getch();
}
```

iv) Copy Constructor

It is used to declare and initialize an object with the values from another object.

Syntax:

```
        copy constructor definition
classname(classname &reference_object)
{
statement;
}
calling copy constructor function
classname newobjectname=oldobjectname;
```

Example:

```
#include<iostream.h>
#include<conio.h>
class data
{
private:
int a;
public:
data(int x)
{
a=x;
}
data(data &ob)
{
a=ob.a;
}
void print()
{
cout<<a;
}
};
void main()
{
clrscr();
data d1(100);
d1.print();
data d2=d1;
d2.print();
getch();
}
```

4 .1 Destructors

Destructor is a member function used to deallocate the memory space allocated by the constructor. It is used to destroy the object that has been created by a constructor.

General form:

```
~destructorname()
{
Statement;
}
```

where ~ (tilde) ⟶ destructor operator

Characteristics:

- Destructor has the same name as the classname.
- No argument can be provided to a destructor.
- It won‟t return any value.
- They cannot be inherited.
- It may not be static.

Example:

```
#include<iostream.h>
#include<co
nio.h> class
sample
{
public:
sample()
{
cout<<”Object is created”;
}
~sample()                    // destructor function
{
cout<<”Object is destroyed”
}
};
void main()
{
sample
s1,s2,s3,s4,s5;
getch();
}
```

6. OPERATOR OVERLOADING

The mechanism of giving special meaning to an operator is known as **Operator overloading.**

All the operators in C++ can be overloaded except the following

1.Class member access operator (., .*)
2.Scope resolution operator(::)
3.Size operator (sizeof)
4.Conditional operator(?:)

Rules for overloading operators

1. Only existing operators can be overloaded.
2. Must have at least one operand that is of user defined data-type.
3. Not able to change basic meaning of an operator
4. Some operators that cannot be overloaded
5. Some operators cannot be overloaded by friend functions. However ,can be overloaded by member functions

What is Operator Function?

Operator function is a function used to perform operator overloading. The operator function could be

1. Member Function
2. Friend Function

The general form of operator function is defined as

```
Return-type  class-name ::operator OP(argument-List)
 {
        Function  Body;
 }
```

Defining operator overloading:

Syntax:

```
Returntype classname:: operator op(arg list)
{
Function body
}
```

Example

```
 void operator+()
{
cout<<"operator";
}
```

1.What are the conditions for Member Operator Function?

The Member Operator function should have

- No argument for Unary Operators
- Only one argument for Binary Operators.

2.What are the conditions for Friend Operator Function?

The Friend Operator Function should have

- Only one argument for Unary operators
- Two argument for Binary Operators

The process of overloading involves the following steps

1.Create a class that defines the data type that is to be used in the overloading function
2.Declare the operator overloading function. It may be either **a member function** or **a friend function**
3.Define the operator overloading function

Type of operator overloading

In c++ operators are of two types

A. Unary operator
B. Binary operator

A) unary operators using Member function

The Unary Operator – (Negative) when apply to an object, it changes the sign of each of its members.

Unary operator acts on only one operand. The unary operators are

-(Negation)
++ (Increment)
--(Decrement)

For unary operators:

- Member function takes Zero arguments
- Friend function takes one argument

Example:

- Unary operator overloading = op o;
  Where op is an operator

```
#include<iostream.h>
class Minus
{
 private:
 int a, b, c ;
 public:
 Minus(int A, int B, int C)
 {
 a = A;
 b = B;
 c = C; }
 void display(void);
//********Declaration of the operator function**********
 void operator - ( );
};
void Minus :: display(void)
{
 cout << "t a = " << a << endl ;
 cout << "t b = " << b << endl ;
```

```
 cout << "t c = " << c << endl ;
}
//*Definition of the operator function**********
void Minus :: operator - ( )
{
 a = -a ;
 b = -b ;
 c = -c ;
}
//***Main Function Definition*
void main(void)
{
 Minus M(5, 10, -15) ;
 cout << "n Before activating operator - ( )n" ;
 M.display( ) ;
 -M ;
 cout << "n After activating operator - ( )n" ;
 M.display( ) ;
}
```

Output

```
 Before activating operator - ( )
a = 5
b = 10
c = -15
After activating operator - ( )
a = -5
b = -10
c = 15
```

unary operators using Friend Function

```
#include<iostream.h>
class Minus
{
 private:
 int a, b, c ;
 public:
 Minus(int A, int B, int C)
 {
 a = A;
 b = B;
 c = C;
 }

 void display(void);
//********Declaration of the operator function**********
 friend void operator - (Minus &m );
};
```

```
void Minus :: display(void)
{
 cout << "t a = " << a << endl ;
 cout << "t b = " << b << endl ;
 cout << "t c = " << c << endl ;
}
//*Definition of the operator function**********
void operator - ( )
{
 a = -a ;
 b = -b ;
 c = -c ;
}
//***Main Function Definition*
void  main(void)
{
 Minus M(5, 10, -15) ;
 cout << "n Before activating operator - ( )n" ;
 M.display( ) ;
 -M ;
 cout << "n After activating operator - ( )n" ;
 M.display( ) ;
}
```

Output

```
Before activating operator - ( )
a = 5
b = 10
c = -15
After activating operator - ( )
a = -5
b = -10
c = 15
```

B) Binary operators using Member function

Operators act on two operands are called Binary Operators.

For example, + (Addition), -(Subtraction) , * Multiplication, / Division

For Binary operators:

- Member function takes one argument
- Friend function takes two arguments

Example:

- Binary operator overloading= o op o1;
  Where op is an operator

Note 1:

When a binary operator is overloaded the left operand is passed implicitly to the operator function and the right operand is passed as an argument.

For example

S3= S1+ S2

S1 Object is passed implicitly to the function

S2 Object is passed as an argument.

Note 2:

The left side object is the object that generates the call to the operator function and is passed implicitly by "This Pointer".

1.Write a program to demonstrate Binary Operator Overloading using member function.

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

 {
  private:
    int x;
    int y;
    int z;
public:
    void getdata(int x1, int y1, int z1)
{
    x=x1;
    y=y1;
    z=z1;
 }

    void display()
{
     cout<<"X="<<x<<endl;
     cout<<"Y="<<y<<endl;
     cout<<"Z="<<z<<endl;
  }

    Space operator +(Space m1)
          {   Space m2;
              m2.x=this.x+m1.x;
              m2.y=this. y+m1.y;
              m3.y=this.z+m1.z;
              return(m2);
        }
   };
void main()
  {
  Space s1,s2,s3;
  s1.getdata(10,20,30);
  s2.getdata(50,80,100);
  s1.display();
```

```
  s2.display();
  s3=s1+s2;
 s3.display();
getch();
 }
```

Overloading Binary operators using Friend function

```
#include<iostream.h>
#include<conio.h>
class Space
  {
  private:


        int x;
        int y;
        int z;
   public:
        void getdata(int x1, int y1, int z1);
        void display();
       friend Space operator +(Space m1, Space m2);
    };
void main()
  {
      Space s1,s2,s3;
      s1.getdata(5,2,7);
      s2.getdata(6,8,4);
      s3=s1+s2;
     s1.display();
     s2.display();
     s3.display();
    getch();
   }
void Space::getdata(int x1, int y1, int z1)
   {
       x=x1;
       y=y1;
       z=z1;
   }
void Space::display()
  {
    cout<<"X="<<x<<endl;
    cout<<"Y="<<y<<endl;
    cout<<"Z="<<z<<endl;
}
Space operator +(Space m1, Space m2)
   {
    Space m3;
```

```
   m3.x=m1.x + m2.x;
  m3.y = m1.y + m2.y;
  m3.z=m1.z+m2.z;
  return(m3);
}
```

Note 1:

Implicit passing is not possible with friend function.

Note 2:

The assignment operator cannot be overloaded with friend function, but only with member function.

Program to store 10 students Rno, Name , Mark1, Mark2, Mark3, Mark4 and mark5 and display the result "Pass" or "Fail".

Program:

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

class Exam

  {
   private:
     int rno;
     char nam[20];
     int m1,m2,m3,m4,m5;
     public:
       void input();
       void output();
};
void main()
  {
    Exam s1[10];
    int i;
    for(i=0; i<=9;i++)
        s1[I].input();
    for (I=0;I<=9;I++)
      s1[I].process();
     getch();
   }
void  Exam::input()
   {
    cout<<"Give Rollno:";
    cin>>rno;
    cout<<"Give Name:";
    cin>>nam;
```

```
  cout<<"Give Five Marks";
  cin>>m1,m2,m3,m4,m5;
 }
void Exam::process()
 {
    char *res;
    int tot;
    if(m1>=40 && m2>=40 && m3>=40 && m4>=40 && m5>=40)
      res="Pass";
    else
      res="Fail";
        cout<<"Rollno:"<<rno<<endl;
        cout<<"Name :"<<nam<<endl;
        cout<<"Mark1:"<<m1<<endl;
        cout<<"Mark2:"<<m2<<endl;
        cout<<"Mark3:"<<m3<<endl;
        cout<<"Mark4:"<<m4<<endl;
        cout<<"Mark5:"<<m5<<endl;
        cout<<"Result:"<<res<<endl;
  }
```

7. New and Delete Operator.

New and Delete are unary operators that are used to allocate dynamic memory and delete the allocated memory in a better and easier way.

New Operator:

The New operator can be used to create objects of any type. It takes the following form

Pointer-Variable = new Data-type;

P1 = new int;
P2= new float;

Note:

We can also initialize the memory using the new operator. This is done as follows

Ptr→variable = new data-type(value);
P1=new int(100);
The P1 variable will have the value 100.

Note:

Array can be created by using New operator.

ptr-Variable=new data-type[Size];

P=new int[10];

Delete Operator:

Delete operator is used to delete or destroy the dynamic memory allocation.

Delete pointer-variable;

Delete p1;

For deleting array object

Delete []ptr-varaiable;

8. Passing Objects to Function:

Objects of a class can be passed to function as arguments in just the same way that other types of data are passed. The following steps are used.

Step 1:

Simply Declare the function's parameter as a class type

Step 2:

Use an object of that class as an argument when calling the function.

Program to explain object passing function.

```
#include<iostream.h>
#include<conio.h>
class Stud
  {
  private:
     int  age;
     float ht;
public:
   void input();
   void ave(Stud m1, Stud m2);
 };
void main()
  {
   Stud s1,s2,s3;
   s1.input();
   s2.input0;
   s3.ave(s1,s2);
   getch();
  }
void Stud::input()
  {
  cout<<"Give Age:";
  cin>>age;
  cout<<"Give Height:";
  cin>>ht;
 }
void Stud::ave(Stud m1, Stud m2)
 {
    float avage,avht;
    avage=(s1.age+s2.age)/2  ;
    avht=(s1.ht+s2.ht)/2 ;
    cout<<"The average Age=:"<<avage<<endl;
    cout<<"The Average Height=:"<<avht<<endl;
}
```

a) Returning Objects by Functions:

Any function can return objects from function. The function which returns value must follow the steps

Step 1:

First declare the function as returning a class type.

Step 2:

Return an object of that type using the normal return statement.

Program to explain Returning Objects from Functions

Example 1:

```
#include<iostream.h>
#include<conio.h>
class Stud
  {
     private:
          int   age;
          float ht;
     public:
        void input();
        Stud  ave(Stud m1, Stud m2);
        Void show();   };
void main()
{
   Stud s1,s2,s3,s4;
   s1.input();
   s2.input0;
   s4=s3.ave(s1,s2);
   s4.show();
   getch();
}
void Stud::input()
  {
  cout<<"Give Age:";
  cin>>age;
  cout<<"Give Height:";
  cin>>ht;
  }
void Stud::ave(Stud m1, Stud m2)
 {
    Stud m3;
    m3.age=(s1.age+s2.age)/2  ;
    m3.ht=(s1.ht+s2.ht)/2;
    return(m3);
  }
void Stud::show()
  {
          cout<<"The average Age:="<<age<<endl;
           cout<<"The average Ht :="<<ht<<endl;
  }
```

11. Write a program to demonstrate unary operator ++ overloading.

```
#include<iostream.h>
#include<conio.h>
class Sample
  {
private:
        int x;
        int y;
        int z;
  public:
     void getdata(int x1, int y1, int z1);
     void display();
     void operator ++(); // overloading negation
 };
void main()
  {
      sample s1;
     s1.getdata(10,20,30);
    s1.display();
    ++s1;
     s1.display();
}
void sample::getdata(int x1, int y1, int z1)
    {
        x=x1;
        y=y1;
        z=z1;
}
void sample::operator ++()  //  definition of operator overloading
{
   x=x+1;
   y=y+1;
   z=z+1;
 }
void sample::display()
  {
   cout<< "X="<<x<<endl;
   cout<<"Y="<<y<<endl;
    cout<<"Z="<<z<<endl;
  }
```

************END*************

UNIT-III

2 Marks:

1. What is Inheritance?

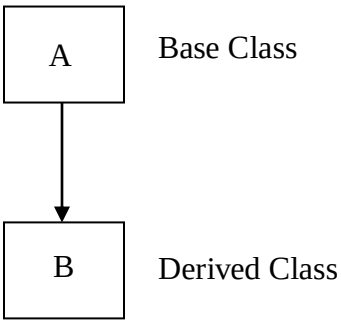
- ✓ The mechanism of deriving a new class from an old class is called **Inheritance.**
- ✓ The old class is referred as **Base Class** and the new class is known as **Derived Class.**
- ✓ The Derived class inherits some or all of the traits or properties from the base class.

2. What are the types of Inheritance?

- ✓ The different types of Inheritance are
  1. Single Inheritance
  2. Multiple Inheritance
  3. Hierarchical Inheritance
  4. Multilevel Inheritance
  5. Hybrid Inheritance

3. What is Single Inheritance?

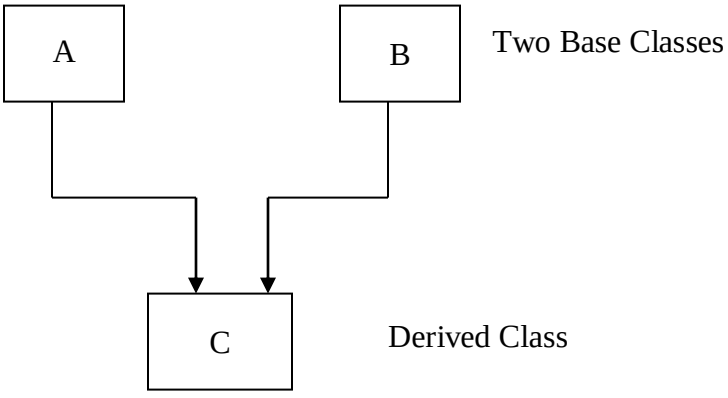
A derived class with only one Base Class is called **Single Inheritance**.



(A) Single Inheritance

4.What is Multiple Inheritance?

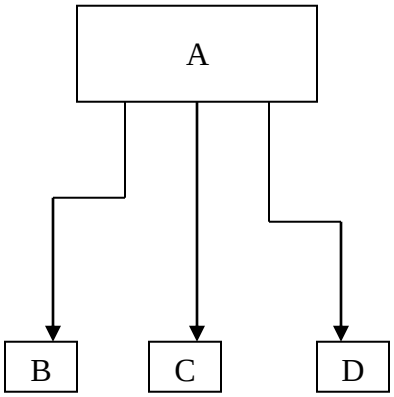
A derived class with several Base Class is called **Multiple Inheritance.**



(b) Multiple Inheritance

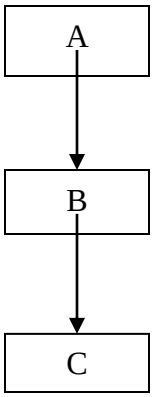
5.What is Hierarchical Class?

Several Derived classes having one base class is known as Hierarchical Inheritance.



6. **What is Multilevel Inheritance?**

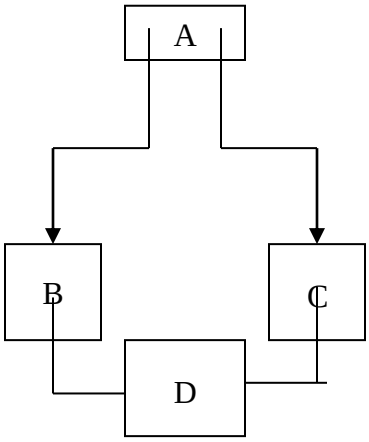
The mechanism of deriving a class from another "Derived Class" is known as Multilevel Inheritance.



D) Multilevel Inheritance

7.What is Hybrid Inheritance?

A class which is created from more than one derived class is known as Hybrid Inheritance.



6.What are the Base Class Access Control?

✓ When One class inherits another class it uses the general form

Class Derived-class-name：Access-mode Base-class-name

{

Members of Derived Class

}

- ✓ **Here the Access Mode could be**
    1. Public
    2. Private
    3. Protected
- ✓ If the access mode is not present, then it is Private by default.

Descriptive questions

1. INHERITANCE

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

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

Inheritance is the process of deriving a new class called derived class from an old class called base class. The derived class inherits some or all features of base class.

The idea of inheritance implements the is a relationship. For example, Mango is-a Fruit. Here Fruit is the base class of the derived class Mango. Likewise Mango, Orange, Banana, etc are the derived classes of the base class Fruit.

Features/Advantages of Inheritance

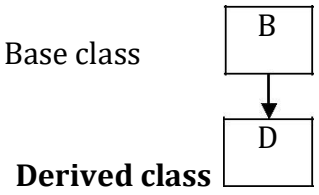
- Reusability
- Extensibility
- Saves time and effort
- Reliability
- Overriding

TYPES OF INHERITANCE

i) Single Inheritance
ii) Multiple Inheritance
iii) Multilevel Inheritance
iv) Hierarchical Inheritance
v) Hybrid Inheritance
vi) Multipath Inheritance

i) Single Inheritance

A class is derived from only one base class.



General form

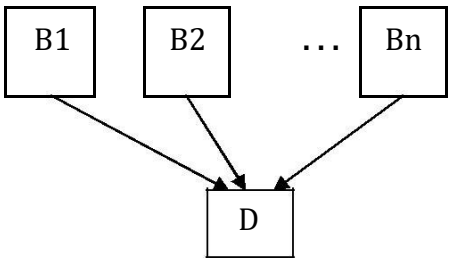
```
class base_class
{
.....
};
class derived_class:visibilitymode base_class
{
.....
};
```

Example

```
#include<iostream.h>
#include<conio.h>
class base
{
public:
        int a,b,c;
void getdata()
{
cout<<"Enter 2 values : ";
cin>>a>>b;
}
};
class derived:public base
{
public:
void add()
{
c=a+b;
cout<<"Sum = "<<c;
}
};
void main()
{
clrscr();
derived d;
d.getdata();
d.add();
getch()
}
```

ii) Multiple Inheritance

A class is derived from more than one base class.



General form

```
class base_class1
{
.....
};
class base_class2
{
.....
};
classderived_class:visibilitymode base_class1,base_class2
{
.....
};
```

Example

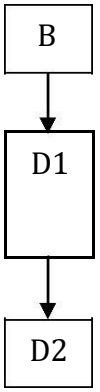
```
#include<iostream.h>
#include<coni
o.h> class
base1
{
public:
int a;
 void geta()
{
cout<<"Enter a : ";
cin>>a;
} };
class base2
{
public:
int b; void
getb()
{
cout<<"Enter b : ";
 cin>>b;
} };
```

class derived:public base1,public base2

```
{
public:
int c;
 void add()
{  c=a+b;
cout<<"Sum = "<<c;
} };
void main()
{    clrscr();
derived d;
 d.geta();
 d.getb();
 d.add(); getch();
}
```

iii) Multilevel Inheritance

A class is derived from another derived class.



- Intermediate base class

General form

```
class base_class
{
.....
};
class derived_class1:visibilitymode base_class
{
.....
};
class derived_class2:visibilitymode derived_class1
{
.....
};
```

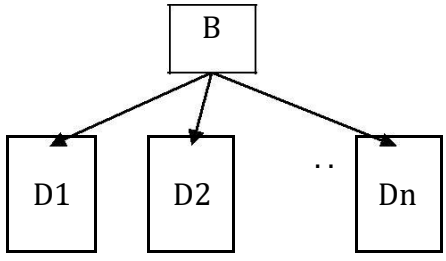
Example

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

```
class base
{
public:
int a,b,c;
void getdata()
{
cout<<"Enter 2 values : ";
cin>>a>>b;
} };
class derived1:public base
{
public: void
add()
{
c=a+b;
} };
class derived2:public derived1
{
public:
void display()
{
cout<<"Sum = "<<c;
} };
void main()
{
clrscr(); derived2 d;
d.getdata(); d.add();
d.display(); getch();
}
```

iv) Hierarchical Inheritance

Derivation of several classes from a single base class.



General form

```
class base_class
{
.....
};
```

```
class derived_class1:visibilitymode base_class
{
.....
};
class derived_class2:visibilitymode base_class
{
.....
};
class derived_classn:visibilitymode base_class
{
.....
};
```

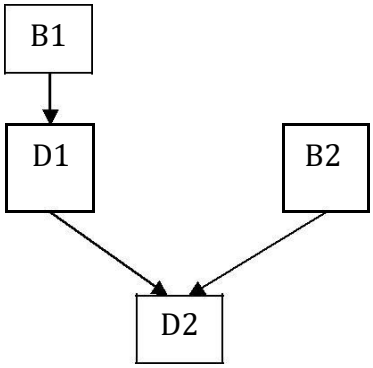
Example

```
#include<iostream.h>
#include<coni
o.h> class
base
{
public:
int a,b,c;
void getdata()
{
cout<<"Enter 2 values : ";
cin>>a>>b;
} };
class derived1:public base
{
public:
void add()
{
c=a+b;
cou<<"Sum = "<<c;
} };
class derived2:public base
{
public:
void sub()
{
c=a-b;
cou<<"Diff = "<<c;
}  };
void main()
{
```

```
clrscr(); derived1 d1;
d1.getdata();
d1.add(); derived2
d2; d2.getdata();
d2.sub(); getch();
}
```

v) Hybrid Inheritance

Derivation of a class involving more than one form of inheritance is known as Hybrid Inheritance.



General form

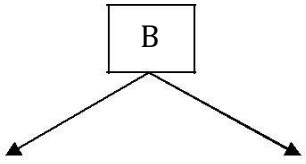
```
class base1
{
.....
};
class derived1:visibilitymode base1
{
.....
};
class base2
{
.....
};
class derived2:visibilitymode derived1,visibilitymode base2
{
....
};
```

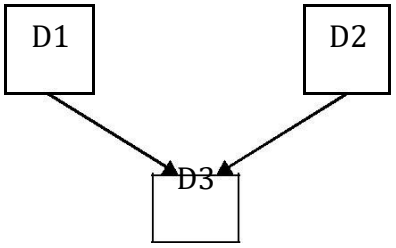
Example

```
#include<iostream.h>
#include<conio.h>
 class base1
{
public:
```

```
 int a,b,c;
void getdata()
{
cout<<"Enter 2 values : ";
cin>>a>>b;
} };
```
class derived1:public base1
```
{
public:
void add()
{
c=a+b;
cout<<"Sum = "<<c;
}  };
```
class base2
```
{
public: int d;
void getd()
{
d=10;
}  };
```
class derived2:public derived1,public base2
```
{
public: int e; void mul()
{
e=c*d;
cout<<"Product = "<<e;
}
};
void main()
{
clrscr(); derived2
d; d.getdata();
d.add(); d.getd()
d.mul();
getch();
}
```
vi) Multipath Inheritance

Derivation of a class from other derived classes, which are derived from the same base class.





General form

```
class base
{
.....
};
class derived1:visibilitymode base
{
.....
};
class derived2:visibilitymode base
{
.....
};
class derived3:visibilitymode derived1,visibilitymode derived2
{
.....};
```

Example

```
#include<iostream.h>
#include<conio.h>
 class base
{
public:
 int a; };
class derived1:public base
{
public:
int b;
};

class derived2:public base
{
Public:
Int d
}
class derived3:public derived1,public derived2
{
```

```
Public:
Int d
}
void display()
{
cout<<"Values are "<<a<<b<<C<<d;
}};
void main()
{
derived3 d3;
d3.a=10;
//error
d3.b=20;
d3.c=30;
d3.d=40;
d3.display();
}
```

2.Virtual Base Class

The duplication of inherited members due to multipath avoided by making the common base class as virtual base class. When a class is made as virtual base class, then only one copy of that class will be inherited.

General form

```
class base
{
.....
};
class derived1:virtual visibilitymode base
{
.....
};
class derived2:virtual visibilitymode base
{
.....
};
class derived3:visibilitymode derived1,visibilitymode derived2
{
..... };
```

Example

```
#include<iostream.h>
#include<conio.h>
class base
{
```

```
Public:
  int a;
}
class derived1:virtual public base
{
Public:
  int b;
};
class derived2:virtual public base
{
Public:
  int c;
}
class derived3:public derived1,public derived2
{
Public:
  int d;
}
void display()
{
cout<<"Values are "<<a<<b<<C<<d;
}
};
void main()
{
derived3 d3;
d3.a=10;
 d3.b=20;
 d3.c=30;
d3.d=40;
 d3.display();
}
```

3.) Pointer and pointer to object

A pointer is a variable that contains a memory address. Very often this address is the location of another object, such as a variable. For example, if x contains the address of y, then x is said to "point to" y. Pointer variables must be declared as such. The general form of a pointer variable declaration is type *var-name;

Here, type is the pointer's base type. The base type determines what type of data the pointer will be pointing to. var-name is the name of the pointer variable.

To use pointer:

- We define a pointer variable.
- Assign the address of a variable to a pointer.
- Finally access the value at the address available in the pointer variable. This is done by using unary operator * that returns the value of the variable located at the address specified by its operand.

Example:
int a=10; //normal variable
int*p; //declare pointer
p = &a; // Assign the address of a variable "a" to a pointer "p"
cout<<"a="<<*p; //prints a=10

3.1 pointers to object.

Objects can be accessed via pointers. When a pointer to an object is used, the object's members are referenced using the arrow(→)operator instead of the dot(.) operator.

Note 1:

When an object pointer is incremented , it points to the next object.

Note 2:

When object pointer is decremented , it points to the previous object.

Example:

```
#include<iostream.h>
#include<conio.h>
class Sample
  {
    private:
          int a,b,c;
    public:
      void input();
      void output();
};
void main()
 {
   Sample s1,s2;
   Sample *p;
   P=&s1;
   p→input();
   p→output();
  (p+1)→input();
  (p+1)→output();
  getch();
  }
 void Sample::input()
  {
   cout<<"Give Data :";
    cin>>a>>b>>c;
  }
void Sample::output()
 {
  cout<<a<<b<<c;
 }
```

3.2 "This Pointer" ?

"**This pointer**" is a pointer that is automatically passed to any member function when it is called and it is a pointer to the object, that generates a call.

Or

"This" is a pointer and points to the object for which this function was called. Each time a member function is invoked it is automatically passed a pointer called "this" to the object that has invoked it.

Example:

```
#include<iostream.h>
#include<conio.h>
Class Example
 {
  private:
       int a,b;
   public:
     void input();
     void output();
};
void main()
   {
 Example s1;
   s1.input():
   s1.output();
}
void Example::input()
{
    cout<<"Give value for a & b";
    cin>>this→a>>this→b;
}
void Example::output()
   {
     cout<<this→a<<this→b;
   }
```

4. Virtual functions (Polymorphism)

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

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

Virtual function

- When same function name used in both base and derived classes, the function in base class is declared as virtual using the keyword virtual at its beginning of declaration.
- Virtual function support late binding (or) dynamic binding. When a function is made virtual, C++ determines which function is use at run time based on the type of object pointed by base pointer.
- By making the base pointer to point to different objects, we can execute different version of virtual function. The derived class inherits the virtual function from the base class.

Example

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

```
#include<conio.h>
class base
{
public:
void display()
{
cout<<"Display Base";
}
virtual void show()
{
cout<<"Show Base";
}
};
class derived:public base
{
public:
void display()
{
cout<<"Display Derived";
}
void show()
{
cout<<"Show Derived";
}
};
void main()
{
clrscr();
base b;
derived d;
base *ptr;
cout<<"Pointer points to Base";
ptr=&b;
ptr->display();
ptr->show();
cout<<"Pointer points to Derived";
ptr=&d;
ptr->display();
ptr->show();
getch();
}
```

Rules

- The virtual functions must be members of some class.
- They cannot be static members.
- They are accessed by using object pointers.
- A virtual function can be a friend of another class.
- Virtual function in a base class must be defined even though it may not be used.

- The prototype of base class version of a virtual function and all the derived class versions must be identical.
- We cannot have virtual constructors, but we can have virtual destructors.
- Base pointer can point to any type of derived object. But we cannot use a pointer of a derived class to access an object of base type.
- Incrementing and decrementing of the base pointer will not point to next object of derived class.
- It will get incremented or decremented only relative to its base type.
- If a virtual function is defined in the base class, it not be redefined in the derived class, then calls will invoke the base function.

4.1. Pure Virtual Function

A pure virtual function is a virtual function declared in a base class that has no definition relative to the base class. Such functions are called do nothing functions.
Pure virtual function is a virtual function with no definition. They start with virtual keyword
and ends with =0

Example

```
#include<iostream.h>    #include<conio.h>
class base
{
public:
virtual void display()=0;
};
class derived1:public base
{
public:
void display()
{
cout<<"Derived 1";
} };
class derived2:public base
{
public:
void display()
{
cout<<"Derived 2";
} };
void main()
{
base *b;
derived d1;
derived d2;
b=&d1;
b->display();
b=&d2;
b->display();
}
```

5. Abstract Base Class

Abstract class is a class which contains at least one pure virtual function in it. Abstract classes are used to provide an interface for its sub classes. Classes inheriting an Abstract class must provide definition to the pure virtual function; otherwise they will also become abstract class.

Characteristics

- It cannot be instantiated, but pointers and references of Abstract class type can be created.
- It can have normal functions and variables along with a pure virtual function.
- They are mainly used for Up casting, 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.

Example

```
#include<iostream.h>
#include<conio.h>
class base
{
public:
virtual void display()=0;
};
class derived1:public base
{
public:
void display()
{
cout<<"Derived 1";
}
};
class derived2:public base
{
public:
void display()
{
cout<<"Derived 2";
}
};
void main()
{
base *b;
derived d1;
derived d2;
b=&d1;
b->display();
b=&d2;
b->display();
getch();
}
```

UNIT-IV

2 Marks:

1. **What is File?**
   File is the collection of related records stored in a disk.
2. **What are the types of files?**
   There are three types :
   - Sequential File
   - Random File
   - Indexed Sequential file
3. **What is sequential file?**
   The records of the sequential files can be accessed one by one or sequentially.
4. **What is indexed Sequential file?**
   The records of the index sequential file can be access randomly as well as sequentially .
5. **What is the name of Header file supports file classes?**
   The header file <fstream.h> contains all file supporting classes such as
   ifstream in // input
   ofstream out // output
   fstream io // input and output
6. **What is Random Access?**
   Random Access is the method of accessing a record from the file in any order.

7. **What are the functions used for random accessing?**
   The random accessing can be done using the functions
   - Seekg()
   - Seekp()

8. **What is the purpose of seekg() function?**
   The seekg() function is used to move get pointer(input) to a specified location.
   Example:
   Infile.seekg(10);
   It moves the file pointer to the byte 10.

9. **What is the purpose of seekp() function?**
   The seekp() is used to move the put pointer(output) to a specified location.
   Example:
   Outfile.seekp(m) where m is the bytes.

10. **What is Exception Handling?**

- ✓ Exception Handing is a built-in mechanism to handle error during run time. It si more useful to manage and respond run time errors.
- ✓ The exception handling is built upon three keywords
  a. Try
  b. Catch
  c. Throw

Syntax:-

```
Try {
```

```
      //  try block
          }
 catch(type1 arg) {
        // catch blcok
          }
 catch(type2 arg) {
     // catch block
 }
.
.
.catch(typen arg) {
  //  catch block
 }
```

Descriptive Questions

1. C++ Streams

The C++ I/O system operates through streams. A stream is logical device that either produces or consumes information. A stream is linked to a physical device by the C++ I/O system. All streams behave in the same manner, even if the actual physical devices they are linked to differ. Because all streams act the same, the I/O system presents the programmer with a consistent interface.

Two types of streams:

Output stream: a stream that takes data from the program and sends (writes) it to destination.

Input stream: a stream that extracts (reads) data from the source and sends it to the program.

C++ provides both the formatted and unformatted IO functions. In formatted or high-level IO, bytes are grouped and converted to types such as int, double, string or user-defined types. In unformatted or low-level IO, bytes are treated as raw bytes and unconverted. Formatted IO operations are supported via overloading the stream insertion (<<) and stream extraction (>>) operators, which presents a consistent public IO interface.

C++ provides supports for its I/O system in the header file< iostream>. Just as there are differentkinds of I/O (for example, input, output, and file access), there are different classes depending on the type of I/O. The following are the most important stream classes:

Class istream :- Defines input streams that can be used to carry out formatted and unformatted input operations. It contains the overloaded extraction (>>) operator functions. Declares input functionssuch **get(), getline() and read().**

Class ostream :- Defines output streams that can be used to write data. Declares output functions putand **write().**The ostream class contains the overloaded insertion (<<) operator function

When a C++ program begins, these four streams are automatically opened:

Stream Meaning Default Device

cin Standard input Keyboard

cout Standard output Screen

cerr Standard error Screen

clog Buffer version of cerr Screen

2. Unformatted Input/ Output Functions

Functions get() and put()

The get function receives one character at a time. There are two prototypes available in C++ for get as given below:

```
get (char *)
get ()
```

Their usage will be clear from the example below:

```
char ch ;
cin.get (ch);
```

In the above, a single character typed on the keyboard will be received and stored in the character variable ch.

Let us now implement the get function using the other prototype:

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

This is the difference in usage of the two prototypes of get functions.

The complement of get function for output is the put function of the ostream class. It also has two forms as given below:

```
cout.put (var);
```

Here the value of the variable var will be displayed in the console monitor. We can also display a specific character directly as given below:

```
cout.put ('a');
```

getline() and write() functions

C++ supports functions to read and write a line at one go. The getline() function will read one line at atime. The end of the line is recognized by a new line character, which is generated by pressing the Enter key. We can also specify the size of the line.

The prototype of the getline function is given below:

```
cin.getline (var, size);
```

When we invoke the above statement, the system will read a line of characters contained in variable var one at a time. The reading will stop when it encounters a new line character or when the required number (size-1) of characters have been read, whichever occurs earlier. The new line character will be received when we enter a line of size less than specified and press the Enter key. The Enter key or Return key generates a new line character. This character will be read by the function but converted into a NULL character and appended to the line of characters.

Similarly, the write function displays a line of given size. The prototype of the write function is given below:

```
write (var, size) ;
```

where var is the name of the string and size is an integer.

Unformatted Binary I/O:

- ✓ The binary data format will be known as unformed binary data stream if we store unformatted binary data on disk it consumer very less disk memory.
- ✓ If we want to write records in Binary format we have to open the file in binary mode by using the I/O operations read and write.
- ✓ **Read function:**

  This function is used to read record in a binary form from a file.

  Syntax :

  ```
  Ifstream read(char * buf, streamsize num);
  ```

  The read() reads num characters from the stream and put them in the buffer pointed to by

buf.

- ✓ **Write( ):**

  This function is used to write the Binary form data on file.

  Syntax :

  Ofstream write((char*buf, stream size num).

3. Formatted I/O via manipulators

The C++ I/O system allows you to format I/O operations. For example, you can set a field width,specify a number base, or determine how many digits after the decimal point will be displayed. I/O manipulators are special I/O format functions that can occur within an I/O statement.

Manipulator	Purpose	Input/Ouput
boolalpha	Turns on boolaphaflag	Input/Output
dec	Turns on decflag	Input/Output
endl	Outputs a newline character and flushes the stream	Output
ends	Outputs a null	Output
flush	Flushes a stream	Output
hex		Input/Output
setw(int w)	Sets the field width to w	Output
showbase	Turns on showbaseflag	Output
showpoint	Turns on showpointflag	Output

The following program demonstrates several of the I/O manipulators:

```
#include<iostream>
#include<iomanip>
using namespacestd;
int main( ) {
cout<< hex << 100 << endl;
cout<< oct<< 10 << endl;
cout<< setfill('X') << setw(10);
cout<< 100 << " hi " << endl;
return0;
}
```

This program displays the following:

```
64
13
XXXXXXX144 hi
```

4. File concepts and related functions in C++

I/O File concepts

A file is a bunch of bytes stored on some storage devices like hard disk, floppy disk etc. File I/O and console I/O are closely related. In fact, the same class hierarchy that supports console I/O also supports the file I/O. To perform file I/O, you must include <fstream> in your program. It defines several classes, including **ifstream, ofstream and fstream**. In C++, a file is opened by linking it to a stream. There are three types of streams: input, output and input/output. Before you can open a file, you must first obtain a stream.

To create an input stream, declare an object of type ifstream.
To create an output stream, declare an object of type ofstream.
To create an input/output stream, declare an object of type fstream.
For example, this fragment creates one input stream, one output stream and one stream capable of both input and output:

ifstream in; // input;
fstream out; // output;
fstream io; // input and output

- ✓ To perform file I/O operation. We must include a header file <fstream.h> which contains all classes supports file operations.
- ✓ **Ifstream class: (input file operation)**
  This calss provides input operations or methods which contains open(), getline(), get(), read(), tellg(), seekg().
  Example :
  Ifstrem f1;
  Fstream infile;
  f1.open("employee.dat");
- ✓ **ofstream class: (output file operation)**
  This class provides output operations it supports the following methods.
  Open( ) tell( )
  Put( ) write( ) Seeekp( )
  Example :
  Ofstream outfile;
  Outfile open ("student-dat");
- ✓ **fstream class:**
  It provides all function supports I/O operation.
  File operation:
  1. Opening the file
  2. Checking the file
  3. Reading / writing the file
  4. Processing the records
  5. Closing the records
  6. Closing the file.
     - ➢ **Opening the file:**

       File can be opened using the open() method.
       Syntax
       Filestream-class object name;
       Object-name.open("File name");
       Example
       Ofstream f1;
       f1.open("student.Dat");
     - ➢ **Checking the file:**
       The file is checked using a method called eof().
       It returns true, if the end of file is encountered. Otherwise it returns zero.
       Example

```
Ifstream f1;
f1.open("numbers.data");
f1.eof()
while (!f1.eof())
{
        file processing;
}
```

- **Writing a file: -**
  The data can be written in a file using.
  File.object<<data1<<end1<<data2<<end1<<data3<<endl;
  Example :
  ```
  ofstream f1;
  f1.open("student.Dat");
  f1<<a1<<endl<<a2<<endl<<a3<<endl;
  ```
- **Reading file**
  The data can be read from the file using
  File object>>data1>>data2>>data3;
  Example
  ```
  f1>>a1>>a2;
  ```
- **Processing the records:**
  Records can be processed by using any mathematical and logical operations.
- **Closing the file:**
  ```
  File object.close();
  ```

5. Sequential Input and Output Operations:

The file stream classes support a number of member functions for performing the input and output operations on files.One pairs of functions,**put()** and **get()** are designed for handling a single character at a time.Another pair of functions, **write(),read()** are designed to write and read blocks of binary data.

5.1 put() and get() Functions:

The function **put()** writes a single character to the associated stream.Similarly,the function **get()** reads a single character from the associated stream.The following program illustrates how the functions work on a file.The program requests for a string.On receiving the string,the program writes it,character by character,to the file using the **put()** function in a for loop.The length of string is used to terminate the for loop.

The program then displays the contents of file on the screen.It uses the function **get()** to fetch a character from the file and continues to do so until the end –of –file condition is reached.The character read from the files is displayed on screen using the operator <<.

PROGRAM 6.3 I/O OPERATIONS ON CHARACTERS

```
#include <iostream.h>   #include <fstream.h>
#onclude<string.h>
int main() {
char string[80];
cout<<"enter a string \n";
cin>>string;
```

```
int len =strlen(string);
fstream file;
file.open("TEXT". Ios::in | ios::out);
for (int i=o;i<len;i++)
file.put(string[i]);
file .seekg(0);
char ch;
while(file)
{
file.get(ch);
cout<<ch;
}  return }
```

5.2 write() and read () functions:

The functions **write()** and **read(),**unlike the functions **put()** and **get()** ,handle the data in binary form.This means that the values are stored in the disk file in same format in which they are stored in the internal memory.An int character takes two bytes to store its value in the binary form,irrespective of its size.But a 4 digit int will take four bytes to store it in the character form.The binary input and output functions takes the following form:

infile.read ((char *) & V,sizeof (V));
outfile.write ((char *) & V ,sizeof (V));
These functions take two arguments.The first is the address of the variable V, and the second is the length of that variable in bytes.The address of the variable must be cast to type char*(i.e pointer to character type).The following program illustrates how these two functions are used to save an array of floats numbers and then recover them for display on the screen.

PROGRAM 6.4 // I/O OPERATIONS ON BINARY FILES

```
#include <iostream.h>
#include <fstream.h>
#include <iomanip.h>
const char * filename ="Binary";
int main()
{
float height[4] ={ 175.5,153.0,167.25,160.70};
ofstream outfile;
outfile.open(filename);
outfile.write((char *) & height,sizeof(height));
outfile.close();
for (int i=0;i<4;i++)
height[i]=0;
ifstream infile;
infile.open(filename);
infile.read ((char *) & height,sizeof (height));
for (i=0;i<4;i++)
{
cout.setf(ios::showpoint);
cout<<setw(10)<<setprecision(2)<<height[i];
}
infile.close();
return 0; }
```

6, Random Access File

File pointers

C++ also supports file pointers. A file pointer points to a data element such as character in the file.The pointers are helpful in lower level operations in files. There are two types of pointers:

get pointer

put pointer

The get pointer is also called input pointer. When we open a file for reading, we can use the get pointer. The put pointer is also called output pointer. When we open a file for writing, we can use put pointer.

These pointers are helpful in navigation through a file. When we open a file for reading, the get pointer will be at location zero and not 1. The bytes in the file are numbered from zero. Therefore, automatically when we assign an object to ifstream and then initialize the object with a file name, the get pointer will be ready to read the contents from 0th position. Similarly, when we want to write we will assign to an ofstream object a filename. Then, the put pointer will point to the 0th position of the given file name after it is created. When we open a file for appending, the put pointer will point to the 0th position. But, when we say write, then the pointer will advance to one position after the last
character in the file.

File pointer functions

There are essentially four functions, which help us to navigate the file as given below Functions
Function Purpose
tellg() Returns the current position of the get pointer
seekg() Moves the get pointer to the specified location
tellp() Returns the current position of the put pointer
seekp() Moves the put pointer to the specified location

```
//To demonstrate writing and reading- using open
#include<fstream.>
#include<iostream>
int main(){ //Writing
ofstream outf;
outf.open(“Temp2.txt”);
outf<<“Working with files is fun\n”;
outf<<“Writing to files is also fun\n”;
outf.close();
char buff[80];
ifstream inf;
inf.open(“Temp2.txt”); //Reading
while(inf){
inf.getline(buff, 80);
cout<<buff<<“\n”;
}
inf.close();
return 0;
}
```

7. Exception Handling

Two common types of error in a program are:
1) **Syntax error** (arises due to missing semicolon, comma, and wrong prog. constructs etc)
2) **Logical error** (wrong understanding of the problem or wrong procedure to get the solution)

Exceptions

Exceptions are the errors occurred during a program execution. Exceptions are of two types:
_ Synchronous (generated by software i.e. division by 0, array bound etc).
_ Asynchronous (generated by hardware i.e. out of memory, keyboard etc).

Exception handling mechanism

_ C++ exception handling mechanism is basically built upon three keywords namely, try, throw and catch.
_ Try block hold a block of statements which may generate an exception.
_ When an exception is detected, it is thrown using a throw statement in the try block.

- ✓ The exception handling is built upon three keywords
  - **a. Try**
  - **b. Catch**
  - **c. Throw**

Try block

Detects and throws exception

Catch block

Catches and handles the exception

A try block can be followed by any number of catch blocks.
The general form of **try** and **catch** block is as follows:

Syntax:-

```
Try   {
            //  try block
        }
 catch(type1 arg) {
                              // catch blcok
                    }
 catch(type2 arg) {
                              // catch block
                    }
.catch(typen arg) {
                              //  catch block
                 }
```

Write a program to find x/y, where x and y are given from the keyboard and both are integers.

```
#include<iostream.h>
void main()
{
int x, y;
cout<<"enter two number"<<endl;
cin>>x>>y;
try
```

```
{
if(y!=o)
{
z=x/y;
cout<<endl<<z;
}
else
{
throw(y);
}
}
catch(int y)
{
cout<<"exception occurred: y="<<y<<endl;
}
}
```

Output:

```
Enter two number
6 o
exception occurred:y=o
```

---

a. Catching Class Types

An exception can be of any type, including class types that you create. Actually, in real-world programs, most exceptions will be class types rather than built-in types. Perhaps the most common reason that you will want to define a class type for an exception is to create an object that describes the error that occurred. This information can be used by the exception handler to help it process the error. The following example demonstrates this.

```
#include <iostream.h>
#include <cstring.h>
class MyException
{
public:
char str_what[80];
int what;
MyException() { *str_what = 0; what = 0; }
MyException(char *s, int e)
{
strcpy(str_what, s);
what = e;
}
};
void main()
{
int i;
try {
cout << "Enter a positive number: ";
cin >> i;
if(i<0)
throw MyException("Not Positive", i);
```

```
}
catch (MyException e) { // catch an error
cout << e.str_what << ": ";
cout << e.what << "\n";
}
}
```

Output:

Enter a positive number: -4
Not Positive: -4

The program prompts the user for a positive number. If a negative number is entered, an object of the class **MyException** is created that describes the error. Thus, **MyException** encapsulates information about the error. This information is then used by the exception handler. In general, you will want to create exception classes that will encapsulate information about an error to enable the exception handler to respond effectively.

---

8. Illustrate some of an File Handling Problems.

Write a program to create file number dat on disk to store 10 records which contains tow numeric numbers.

```
#include<iostream.h>
#include<fstream.h>
#include<conio.h>
void main()
  {
    ofstream f1;
    f1.open("number.dat");

     int i;
     int a1;
     int a2;
    for(i=1;i<=5;i++)
     {
       cout<<"Give Data1";
       cin>>a1;
       cout<<"Give Data2:";
       cin>>a2;
       f1<<a1<<endl<<a2<<endl;
     }
       f1.close();
       getch();
    }
```

Write a program to create a file called “student.dat” to store rno, name, m1, m2, m3, m4,m5 for ten students.

```
#include<iostream.h>
#include<fstream.h>
#include<conio.h>
void main( )
{
ofstream f1
```

```
f1.open("student.dat");
int rno;
char name[15];
int m1, m2, m3, m4, m5;
for (i=1; i<=10; i++)
{
cout<<"Give rno<<"Give name"<<endl;
cout<<"Give marks" <<endl;
cin>>rno>>name>>endl;
cin>>m1>>m2>>m3>>m4>>m5;
f1<<m1 <<m2<<m3<<m4<<m5<<endl;
}
f1.close();
}
```

Write a program to read a file "number.dat" which contains records of two numbers and find the sum.

```
#include<iostream.h>
#include<fstream.h>
#include<conio.h>
void main()
  {
    clrscr();
    ifstream f1;
    f1.open("number.dat");
    int a1, a2, c1;
   while (!f1.eof( ))
    {
      f1>>a1>>a2;
      c1=a1+a2;
      cout<<"The output:"<<a1<<"  "<<a2<<c1<<endl;
      }
      f1.close();
      getch();
      }
```

Write a program to create a file "payroll.dat" to store 10 employees informations such as eno, ename, basicpay, da, & hra.

```
#include<iostream.h>
#include<fstream.h>
#include<conio.h>
void main()
  {
   clrscr();
   ofstream f1;
   f1.open("payroll.dat");
   int eno;
   char nam[10];
   float basic;
   float da;
```

```
  float hra;
  int i;
  for(i=1;i<=10;i++)
   {
    cout<<"Give emp NO";
    cin>>eno;
    cout<<"Give Name:";
    cin>>nam;
    cout<<"Give Baic [ay:";
    cin>>basic;
    cout<<"Give Da:";
    cin>>da;
    cout<<"Give Hra:";
    cin>>hra;
    f1<<eno<<endl<<nam<<endl<<basic<<endl<<da<<endl<<hra<<endl;
    }
    f1.close();
    getch();
   }
```

Write a program to read the "payroll.dat" file to find gross pay for each employee.

```
#include<iostream.h>
#include<fstream.h>
#include<conio.h>
void main()
 {
  clrscr();
  ifstream f1;
  f1.open("payroll.dat");
  int eno;
  char nam[10];
  float basic;
  float da;
  float hra;
 float gross
 while(!f1.eof())
  {
   f1>>eno>>nam>>basic>>da>>hra;
    gross=basic+da+hra
   cout<<endl;
   cout<<"Employee No:"<<eno<<endl;
   cout<<"Employee Name:"<<nam<<endl;
   cout<<"Basic Pay    :"<<basic<<endl;
   cout<<"Hra          :"<<hra<<endl;
   cout<<"DA           :"<<da<<endl;
   cout<<"Gross pay          :"<<gross<<endl;
   }
   f1.close();
```

```
 getch();
}
```

Program to create a binary file to store ten records of students and read them .

```
#include<iostream.h>
#include<fstream.h>
#include<conio.h>
void main()
  {
    clrscr();
    struct student
          {  int rno;
             char nam[15];
             int age;
             float ht;
   };
    struct student stud;
    ofstream f1;
    f1.open("student.dat",ios::binary);
    for(int i=1;i<=5;i++)
     {
         cout<<"Give Roll No:";
         cin>>stud.rno;
         cout<<"Give Name   :";
         cin>>stud.nam;
         cout<<"Give Age    :";
         cin>>stud.age;
         cout<<"Give Ht     :";
         cin>>stud.ht;
         f1.write((char *) &stud,sizeof(struct student));
      }
         f1.close();
   }
```

Binary File Reading:

```
include<iostream.h>
#include<fstream.h>
#include<conio.h>
void main()
  {
    clrscr();
    struct student
    {
            int rno;
            char nam[15];
            int age;
```

```
        float ht;
};
 struct student stud;

 int sum=0;
 float aage;
 ifstream f1;
 f1.open("student.dat",ios::binary);
while(!f1.eof())
{
      f1.read((char*) &stud,sizeof(struct student));
      cout<<"The Roll No:"<<stud.rno<<endl;
      cout<<"The Name   :"<<stud.nam<<endl;
      cout<<"The Age    :"<<stud.age<<endl;
      cout<<"The Height :"<<stud.ht<<endl<<endl;
      sum=sum+stud.age;
 }
f1.close();
aage=sum/10;
cout<<"The Average Age:"<<aage;
getch();
}
```

8 . Templates

- **a. Function Template**
- **b. Class Template**

Templates are the foundation of generic programming, which involves writing code in a way that is independent of any particular type.

A template is a blueprint or formula for creating a generic class or a function. The library containers like iterators and algorithms are examples of generic programming and have been developed using template concept.

There is a single definition of each container, such as **vector**, but we can define many different kinds of vectors for example, **vector <int>** or **vector <string>**.

You can use templates to define functions as well as classes, let us see how they work:

a) Function Template

The general form of a template function definition is shown here:

```
template <class type> ret-type func-name(parameter list)
{
// body of function
}
```

Here, type is a placeholder name for a data type used by the function. This name can be used within the function definition.

The following is the example of a function template that returns the maximum of two values:

```
#include <iostream>
#include <string>
using namespace std;
template <typename T>
inline T const& Max (T const& a, T const& b)
{
return a < b ? b:a;
}
int main ()
{
int i = 39;   int j = 20;
cout << "Max(i, j): " << Max(i, j) << endl;
double f1 = 13.5;
double f2 = 20.7;
cout << "Max(f1, f2): " << Max(f1, f2) << endl;
string s1 = "Hello";
string s2 = "World";
cout << "Max(s1, s2): " << Max(s1, s2) << endl;
return 0;
}
```

If we compile and run above code, this would produce the following result:

```
Max(i, j): 39
Max(f1, f2): 20.7
Max(s1, s2): World
```

b). Class Template

Just as we can define function templates, we can also define class templates. The general form of a generic class declaration is shown here:

```
template <class type> class class-name
{
.
.
}
```

Here, **type** is the placeholder type name, which will be specified when a class is instantiated. You can define more than one generic data type by using a comma-separated list.

Following is the example to define class Stack<> and implement generic methods to push and pop the elements from the stack:

```
#include <iostream>
#include <vector>
#include <cstdlib>
#include <string>
#include <stdexcept>
using namespace std;
template <class T>
```

```
class Stack {
private:
vector<T> elems; // elements
public:
void push(T const&); // push element
void pop(); // pop element
T top() const; // return top element
bool empty() const{ // return true if empty.
return elems.empty();
} };
template <class T>
void Stack<T>::push (T const& elem)
{                                          // append copy of passed element
elems.push_back(elem);
}
template <class T>
void Stack<T>::pop ()
{
if (elems.empty()) {
throw out_of_range("Stack<>::pop(): empty stack");
}                                  // remove last element
elems.pop_back();
}
template <class T>
T Stack<T>::top () const
{
if (elems.empty()) {
throw out_of_range("Stack<>::top(): empty stack");
}                                  // return copy of last element
return elems.back();
}
int main()
{
try {
Stack<int> intStack; // stack of ints
Stack<string> stringStack; // stack of strings
// manipulate int stack
intStack.push(7);
cout << intStack.top() <<endl;
// manipulate string stack
stringStack.push("hello");
cout << stringStack.top() << std::endl;
stringStack.pop();
stringStack.pop();
}
catch (exception const& ex)
{ cerr << "Exception: " << ex.what() <<endl;
return -1;
```

```
}}
```
If we compile and run above code, this would produce the following result: 7
hello Exception: Stack<>::pop(): empty stack

************END**************

UNIT-V

2Mark:

1.Define STL.

- ✓ A set of general-purpose templatized classes for data structures and algorithms that could be used as a standard approach for storing and processing of data.
- ✓ The colletion of these generic classes and functions is called the Standard Template Library.

2.Name the Components of STL.

- ✓ The STL contains several components. These omponents work in conjunction with one another to provide support to a variety of programming solutions.They are:
- ✓ A)Containers
  B)Algorithms and
  C)Iterators

3.Define Containers.

- ✓ A container is a way to store data, whether the data consists of built-in types such as int and float, or of class objects.
- ✓ The STL makes seven basic kinds of containers available, as well as three more that are derived from the basic kinds.
- ✓ Containers in the STL fall into two main categories: ***sequence*** **and** ***associative*****.**

4.Define Object-Oriented Analysis.

1.Understanding the problem.
2.Drawing the specification of requirement of the user and the software.
3.Identifying the objects and their attributes.
4.Identifying the services that each object is expected to provide (interface).
5.Establishing inter-connections (collaborations) between the objects in terms of services

5. Write the role of Software engineers.

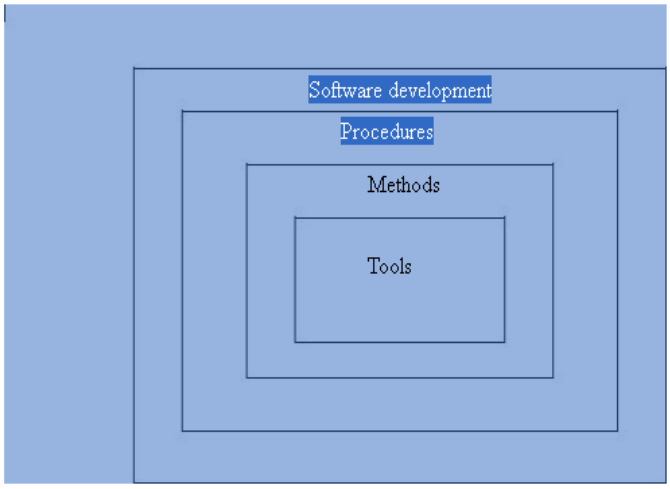
- ✓ Software engineers have been trying various tools, methods, and procedures to control the process of software development in order to build high quality software with improved productivity.
- ✓ The methods provide "how to s" for building the software while the tools provide automated or semi-automated support for the methods.
- ✓ They are used in all the stages of software development process, namely, planning, analysis, design, development and maintenance.
- ✓ The software development procedures integrate the methods and tools together and enable rational and timely development of software systems.

Descriptive Questions :

1. **Components of Software development.**

- ✓ Software engineers have been trying various tools, methods, and procedures to control the process of software development in order to build high quality software with improved productivity.
- ✓ The methods provide "how to s" for building the software while the tools provide automated or semi-automated support for the methods.
- ✓ They are used in all the stages of software development process, namely, planning, analysis, design, development and maintenance.

- ✓ The software development procedures integrate the methods and tools together and enable rational and timely development of software systems.



- ✓ They provide guidelines as to apply the methods and tools, how to produce the deliverables at each stage, what controls to apply, and what milestones to use to assess the progress.
- ✓ There exist a number of software development paradigms, each using a different set of methods and tools.
- ✓ The selection of particular paradigms depends on the nature of the application, the programming language used, and the controls and deliverables required.
- ✓ The development of a successful system depends not only on the use of the appropriate methods and techniques but also on the developer's commitment to the objectives of the systems.
- ✓ A successful system must:
    1. satisfy the user requirements,
    2. be easy to understand by the users and operators,
    3. be easy to operate,
    4. have a good user interface,
    5. be easy to modify, Tools
    6. be expandable,
    7. have adequate security controls against misuse of data,
    8. handle the errors and exceptions satisfactorily, and
    9. Be delivered on schedule within the budget.

2. Procedure-Oriented Paradigm.

- ✓ Software development is usually characterized by a series of stages depicting the various asks involved in the development process.
- ✓ The classic life cycle 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 development but also the user to understand the problem better.
- ✓ **Analysis:** This covers a detailed study of the requirements of both the user and the software. The activity is basically concerned with what of the system such as
  •What are the inputs to the systems?
  •What are the processes required?
  •What are the outputs expected?
  •What are the constraints?

- ✓ **Design:** the design phase deals 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:** once the code is written, it should be tested rigorously for correctness of the code and results. Testing may involve the individual units and the whole systems. It requires a detailed plan as to what, when and how to test.
- ✓ **Maintenance:** After the software has been installed, it may undergo some changes. This may occur due to a change in the user's requirement, a change in the operating environment, or an error in the software that has been fixed during the testing. Maintenance ensures that these changes are incorporated wherever necessary.

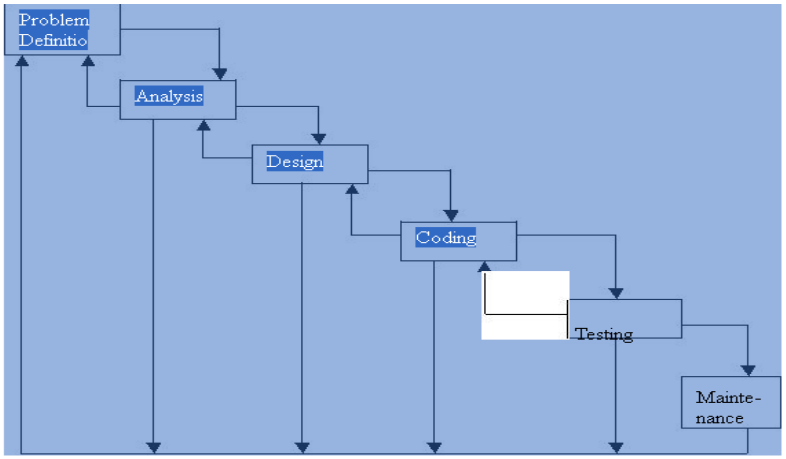


Fig. classic software development life cycle (Embedded 'water-fall' model)

---

2. Object-Oriented Analysis

- ✓ Object-oriented analysis (OOA) refers to the methods of specifying requirements of the software in the terms of real-world objects, their behavior, and their interactions.
- ✓ Object- oriented design (OOD), on the other hand, turns the software requirements into specifications for objects and derives class hierarchies from which the objects can be created.
- ✓ Finally, object-oriented programming (OOP) refers to the implementation of the program using objects, in an object-oriented programming language such as C++.

- ✓ By developing specifications of the objects found in the problem space, a clear and well-organized statement of the problem is actually built into application.
- ✓ These objects form a high-level layer of definitions that are written in terms of the problem space. During the refinement of the definitions and the implementation of the application objects, other objects and identified.
- ✓ All the phases in the object-oriented approach work more closely together because of the commonality of the object model. In one phase, the problem domain objects are identified, while in the next phase additional objects required for a particular solution are specified. The design process is repeated for these implementation-level objects.
- ✓ Object-oriented analysis provides us with simple, yet powerful, mechanism for identifying objects, the building block of the software to be developed. The analysis is basically concerned with the decomposition of a problem into its component parts and establishing a logical model to describe the system functions.
- ✓ The object-oriented analysis (OOA) approach consists of the following steps:
  1.Understanding the problem.
  2.Drawing the specification of requirement of the user and the software.
  3.Identifying the objects and their attributes.
  4.Identifying the services that each object is expected to provide (interface).
  5.Establishing inter-connections (collaborations) between the objects in terms of services required and services rendered.

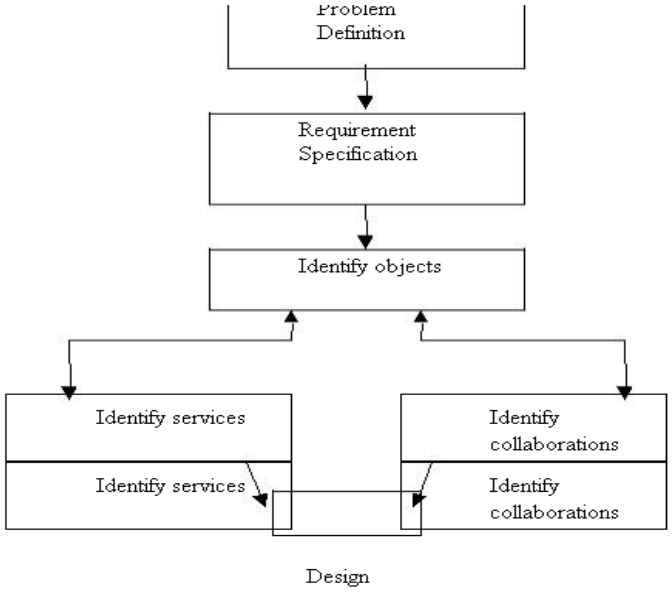


Fig. Activities of object-oriented Analyses

4.)Types of Containers.

Introduction:

- ✓ A container is a way to store data, whether the data consists of built-in types such as int and float, or of class objects.
- ✓ The STL makes seven basic kinds of containers available, as well as three more that are derived from the basic kinds.
- ✓ Containers in the STL fall into two main categories: ***sequence*** **and** ***associative*****.**
- ✓ The sequence containers are ***vector, list,*** **and** ***deque.***

- ✓ The associative containers are ***set, multiset, map,*** **and** ***multimap*****.** In addition, several specialized containers are derived from the sequence containers. These are *stack*, *queue*, and *priority queue*.

Sequence Containers:

- ✓ A sequence container stores a set of elements in what you can visualize as a line, like houses on a street. Each element is related to the other elements by its position along the line. Each element is preceded by one specific element and followed by another.
- ✓ The STL provides the *vector* container to avoid these difficulties.This can be very time-consuming.
- ✓ The STL provides the *list* container, which is based on the idea of a linked list.
- ✓ The third sequence container is the *deque*, which can be thought of as a combination of a stack and a queue. Both input and output take place on the top of the stack.
- ✓ A queue, on the other hand, uses a first-in-first-out arrangement: data goes in at the front and comes out at the back, like a line of customers in a bank.
- ✓ A deque combines these approaches so you can insert or delete data from either end. The word deque is derived from Double-Ended QUEue. It's a versatile mechanism that's not only useful in its own right, but can be used as the basis for stacks and queues.

Associative Containers

- ✓ An associative container is not sequential; instead it uses *keys* to access data. The keys, typically numbers or stings, are used automatically by the container to arrange the stored elements in a specific order.
- ✓ It's like an ordinary English dictionary, in which you access data by looking up words arranged in alphabetical order and the container converts this key to the element's location in memory.
- ✓ There are two kinds of associative containers in the STL: ***sets*** **and** ***maps*****.**
- ✓ These both store data in a structure called a ***tree***, which offers fast searching, insertion, and deletion.
- ✓ Sets and maps are thus very versatile general data structures suitable for a wide variety of applications. However, it is inefficient to sort them and perform other operations that require random access.
- ✓ Sets are simpler and more commonly used than maps. A set stores a number of items which contain *keys*. The keys are the attributes used to order the items.
- ✓ For example, a set might store objects of the person class, which are ordered alphabetically using their name attributes as keys. In this situation, you can quickly locate a desired person object by searching for the object with a specified name.
- ✓ If a set stores values of a basic type such as int, the key is the entire item stored.
- ✓ A map stores pairs of objects: a key object and a value object.
- ✓ The *map* and *set* containers allow only one key of a given value to be stored. This makes sense in, say, a list of employees arranged by unique employee numbers. On the other hand, the *multimap* and *multiset* containers allow multiple keys.

---

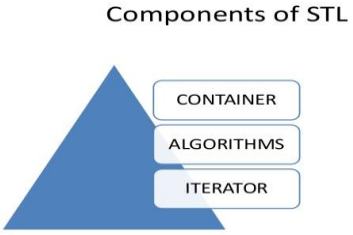
5. Components of Standard Template Library.

Introduction to the STL

- ✓ The STL contains several kinds of entities. The three most important are
  - i) containers
  - ii) algorithms,and
  - iii) iterators.
- ✓ A ***container*** **i**s a way that stored data is organized in memory. The STL containers are

implemented by template classes, so they can be easily customized to hold different kinds of data.

- ✓ ***Algorithms*** in the STL are procedures that are applied to containers to process their data in various ways.For example, there are algorithms to sort, copy, search, and merge data. Algorithms are represented by template functions. These functions are not member functions of the container classes.
- ✓ ***Iterators*** are a generalization of the concept of pointers: they point to elements in a container. You can increment an iterator, as you can a pointer, so it points in turn to each element in a container. Iterators are a key part of the STL because they connect algorithms with containers.



Containers

- ✓ A container is a way to store data, whether the data consists of built-in types such as int and float, or of class objects.
- ✓ The STL makes seven basic kinds of containers available, as well as three more that are derived from the basic kinds.
- ✓ Containers in the STL fall into two main categories: ***sequence*** **and** ***associative*****.**
- ✓ The sequencecontainers are ***vector, list,*** **and** ***deque.***
- ✓ The associative containers are ***set, multiset, map,*** **and** ***multimap*****.**
- ✓ In addition,several specialized containers are derived from the sequence containers. These are *Stack, Queue and Priority Queue.*

Algorithms

- ✓ An algorithm is a function that does something to the items in a container (or containers).
- ✓ We noted, algorithms in the STL are not member functions or even friends of container classes, as they are in earlier container libraries, but are standalone template functions.
- ✓ Suppose you create an array of type int, with data in it:

```
int arr[8] = {42, 31, 7, 80, 2, 26, 19, 75};
```

- ✓ You can then use the STL sort() algorithm to sort this array by saying

```
sort(arr, arr+8);
```

where arr is the address of the beginning of the array, and arr+8 is the past-the-end address(one item past the end of the array).

Iterators

- ✓ Iterators are pointer-like entities that are used to access individual data items (which are usually called *elements*), in a container.
- ✓ Often they are used to move sequentially from element to element, a process called *iterating* through the container.
- ✓ You can increment iterators with the ++ operator so they point to the next element, and dereference them with the * operator to obtain the value of the element they point to.
- ✓ In the STL an iterator is represented by an object of an iterator class.
- ✓ Different classes of iterators must be used with different types of container.
- ✓ There are three major classes of iterators: forward, bidirectional, and random access.

- ✓ A *forward iterator* can only move forward through the container, one item at a time. Its ++ operator accomplishes this. It can't move backward and it can't be set to an arbitrary location in the middle of the container.
- ✓ A *bidirectional iterator* can move backward as well as forward, so both its ++ and -- operators are defined. A *random access iterator*, in addition to moving backward and forward, can jump to an arbitrary location.

6.St*ring*s and Basic String Operations in C++

Putting aside any string-related facilities inherited from C, in C++, strings are not a built-in data type, but rather a Standard Library facility. Thus, whenever we want to use strings or string manipulation tools, we must provide the appropriate **#include** directive, as shown below:

```
#include <string>
using namespace std;    // Or using std::string;
```

We now use **string** in a similar way as built-in data types, as shown in the example below, declaring a variable **name**:

```
string name;
```

Unlike built-in data types (**int**, **double**, etc.), when we declare a **string** variable without initialization (as in the example above), we *do have* the guarantee that the variable will be initialized to an empty string — a string containing zero characters.

C++ strings allow you to directly initialize, assign, compare, and reassign with the intuitive operators, as well as printing and reading (e.g., from the user), as shown in the example below:

```
string name;
cout << "Enter your name: " << flush;
cin >> name;
   // read string until the next separator
   // (space, newline, tab)

   // Or, alternatively:
getline (cin, name);
   // read a whole line into the string name

if (name == "")
{
   cout << "You entered an empty string, "
        << "assigning default\n";
   name = "John";
}
else
{
   cout << "Thank you, " << name
        << "for running this simple program!"
        << endl;
}
```

String Operations.:

C++ strings also provide many string manipulation facilities. The simplest string manipulation that we commonly use is concatenation, or addition of strings. In C++, we can use the + operator to concatenate (or "add") two strings, as shown below:

```
string result;
string s1 = "hello ";
string s2 = "world";
result = s1 + s2;
   // result now contains "hello world"
```

Notice that both **s1** and **s2** remain unchanged! The operation reads the values and produces a result corresponding to the concatenated strings, but doesn't modify any of the two original strings.

The += operator can also be used. In that case, one string is appended to another one:

```
string result;
string s1 = "hello";
  // without the extra space at the end
string s2 = "world";
result = s1;
result += ' ';    // append a space at the end
result += s2;
```

After execution of the above fragment, **result** contains the string **"hello world"**.

You can also use two or more + operators to concatenate several (more than 2) strings. The example below shows how to create a string that contains the full name from first name and last name (e.g., **firstname = "John"**,**lastname = "Smith"**, **fullname = "Smith, John"**).

```
string firstname, lastname, fullname;
cout << "First name: ";
getline (cin, firstname);
cout << "Last name: ";
getline (cin, lastname);
fullname = lastname + ", " + firstname;
cout << "Fullname: " << fullname << endl;
```

Of course, we didn't need to do that; we could have printed it with several << operators to concatenate to the output. The example intends to illustrate the use of strings concatenation in situations where you need to store the result, as opposed to simply print it.

Now, let's review this example to have the full name in format **"SMITH, John"**. Since we can only convert characters to upper case, and not strings, we have to handle the string one character at a time. To do that, we use the square brackets, as if we were dealing with an array of characters, or a vector of characters.

For example, we could convert the first character of a string to upper case with the following code:

```
str[0] = toupper (str[0]);
```

The function **toupper** is a Standard Library facility related to character processing; this means that when using it, we have to include the **<cctype>** library header:

```
#include <cctype>
```

If we want to change all of them, we would need to know the length of the string. To this end, strings have a method**length**, that tells us the length of the string (how many characters the string has).

Thus, we could use that method to control a loop that allows us to convert all the characters to upper case:

```
for (string::size_type i = 0; i < str.length(); i++)
{
   str[i] = toupper (str[i]);
}
```

Notice that the subscripts for the individual characters of a string start at zero, and go from **0** to **length−1**.

Notice also the data type for the subscript, **string::size_type**; it is recommended that you always use this data type, provided by the **string** class, and adapted to the particular platform. All string facilities use this data type to represent positions and lengths when dealing with strings.

The example of the full name is slightly different from the one shown above, since we only want to change the first portion, corresponding to the last name, and we don't want to change the string that holds the last name — only the portion of the full name corresponding to the last name. Thus, we could do the following:

```
fullname = lastname + ", " + firstname;
for (string::size_type i = 0; i < lastname.length(); i++)
{
   fullname[i] = toupper (fullname[i]);
}
```

C++ supports functions that manipulate null-terminated strings.

_____strcpy(str1, str2): Copies string str2 into string str1.

- strcat(str1, str2): Concatenates string str2 onto the end of string str1.
- strlen(str1): Returns the length of string str1.
- strcmp(str1, str2): Returns 0 if str1 and str2 are the same; less than 0 if str1<str2; greater than 0 if str1>str2.
- strchr(str1, ch): Returns a pointer to the first occurrence of character ch in string str1.
- strstr(str1, str2): Returns a pointer to the first occurrence of string str2 in string str1.

Important functions supported by String Class

- append(): This function appends a part of a string to another string
- assign():This function assigns a partial string
- at(): This function obtains the character stored at a specified location
- begin(): This function returns a reference to the start of the string
- capacity(): This function gives the total element that can be stored
- compare(): This function compares a string against the invoking string
- empty(): This function returns true if the string is empty
- end(): This function returns a reference to the end of the string
- erase(): This function removes character as specified
- find(): This function searches for the occurrence of a specified substring
- length(): It gives the size of a string or the number of elements of a string
- swap(): This function swaps the given string with the invoking one

Important Constructors obtained by String Class

- String(): This constructor is used for creating an empty string
- String(const char *str): This constructor is used for creating string objects from a null-terminated string
- String(const string *str): This constructor is used for creating a string object from another string object

**************END*************