DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

(JUNE 2016-NOV 2016)

OBJECT ORIENTED PROGRAMMING AND DESIGN

III SEMSTER

CST33-OBJECT ORIENTED PROGRAMMING AND DESIGN

SYLLABUS

UNIT – I

Introduction to Object-Oriented Programming: Evolution of programming methodologies – Disadvantages of conventional programming – programming paradigms – key concepts of object – oriented programming – advantages of OOP – usage of OOP.

Input and output in C++ : Limitations of C – Introduction to C++ – Structure of the C++ program – stream classes – formatted and unformatted data – unformatted console I/O operations – Bit fields, Manipulators – Manipulators with multiple parameter

Control structures: Decision making statements – jump statement – switch case statement – looping statements.

Classes and objects: Defining member functions – rules of inline functions – data hiding or encapsulation – classes – objects and memory – static object – array of objects – objects as function arguments, friend functions, member functions and non-member functions – overloading member functions.

Functions in C++ : Passing arguments – LValues and RValues – return by reference – default arguments – inline functions – function overloading.

UNIT – II

Constructors and Destructors: Purpose of Constructors and Destructors – overloading constructors – constructors with default arguments – copy constructors – calling constructors and destructors – dynamic initialization using constructors – recursive constructor.

Overloading Functions: Overloading unary operators – constraint on increment and decrement operators – overloading binary operators – overloading with friend functions – type conversion – one argument constructor and operator function – overloading stream operators.

Inheritance: Introduction – Types of Inheritance – Virtual base classes – constructors and destructors and inheritance – abstract classes – qualifier classes and inheritance – common constructor – pointers and inheritance – overloading member function.

UNIT - III

Pointers and arrays: Pointer to class and object – pointer to derived classes and base classes – accessing private members with pointers – address of object and void pointers –characteristics of arrays – array of classes.

Memory: Memory models – The new and delete operators – Heap consumption – Overloading new and delete opetaors – Execution sequence of constructors and destructors – specifying address of an object – dynamic objects.

Binding, Polymorphism and Virtual Functions: Binding in C++ – Pointer to derived class objects – virtual functions – Array of pointers – Abstract classes – Virtual functions in derived classes – constructors and virtual functions – virtual destructors – desctructos and virtual functions. Strings - Declaring and initializing string objects – relational operators – Handling string objects – String attributes – Accessing elements of strings – comparing and exchanging and Miscellaneous functions.

UNIT – IV

Files: File Stream classes – Checking for errors – file opening modes – file pointers and manipulators – manipulators with arguments – read and write operations – Binary and ASCII files – Random access operation – Error handling functions – command line arguments – stdstreams.

Generic Programming with Templates: Generic Functions- Need of Template – Normal function template – class template with more parameters – Function template with more parameters, overloading of function templates, class template with overloaded operators – class templates and inheritance.

Exception Handling: Fundamentals of Exception Handling – Catching Class Types – Using Multiple catch statements – Catching All Exception – Rethrowing Exception – Specifying Exception – Exceptions in constructors and destructors – controlling uncaught Exceptions – Exception and operator overloading – Exception and inheritance – Class Template and Exception handling.

UNIT – V

Object Modelling and Object Oriented Software development: Overview of OO concepts – UML – Use case model – Class diagrams – Interaction diagrams – Activity diagrams – state chart diagrams - Patterns – Types – Object Oriented Analysis and Design methodology – Interaction Modelling – OOD Goodness criteria

TEXT BOOKS

1. Ashok N.Kamthane,Object Oriented Progemmaing with ANSI and Turbo C++, Pearson Edition
2. Deitel & Deitel, C++ How to program, Prentice Hall, Eighth Edition, 2011.
3. Rajib Mall, "Fundamentals of Software Engineering". PHI Learning, Third Edition, 2013.

REFERENCES

1. 1. Eric Nagler, Learning C++ A Hands on Approach, Jaiho publishing house.
2. E Balagurusamy, *Object Oriented Programming with C++*, Tata McGraw Hill, 2nd Edition.
3. Sotter A Nicholas and Kleper J Scott, *Professional C++*, Wiley Publishing Inc.

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

SUBJECT NAME: OBJECT ORIENTED PROGRAMMING AND DESIGN SUBJECT CODE: CST33

UNIT I

Introduction to Object-Oriented Programming: Evolution of programming methodologies – Disadvantages of conventional programming – programming paradigms – key concepts of object – oriented programming – advantages of OOP – usage of OOP.

Input and output in C++ : Limitations of C – Introduction to C++ – Structure of the C++ program – stream classes – formatted and unformatted data – unformatted console I/O operations – Bit fields, Manipulators – Manipulators with multiple parameter

Control structures: Decision making statements – jump statement – switch case statement – looping statements.

Classes and objects: Defining member functions – rules of inline functions – data hiding or encapsulation – classes – objects and memory – static object – array of objects – objects as function arguments, friend functions, member functions and non-member functions – overloading member functions.

Functions in C++ : Passing arguments – LValues and RValues – return by reference – default arguments-inline functions – function overloading.

2 MARKS

1. **Mention some of the disadvantages of conventional programming.**

   Following are the drawbacks observed in monolithic, procedural, and structured programming languages:

   - Large size programs are divided into smaller programs known as functions. These functions can call one another. Hence, security is not provided.
   - Importance is not given to security of data but on doing things.
   - Data passes globally from one function to another.
   - Most functions have access to global data.

2. **List out the programming paradigms.**

   The Programming paradigms are

   - Monolithic Programming
   - Procedural Programming
   - Structured Programming
   - Object Oriented Programming

3. **Give some characteristics of procedure-oriented language.**

   - Emphasis is on doing things (algorithms).
   - Larger programs are divided into smaller programs known as functions.
   - Most of the functions share global data.
   - Data move openly around the system from function to function.
   - Employs top-down approach in program design.

4. **What are the features of Object Oriented Programming?**

   - Programs are divided into objects
   - Emphasis is on data rather than procedure
   - Data Structures are designed such that they characterize the objects
   - Functions that operate on the data of an object are tied together
   - Data is hidden and cannot be accessed by external functions
   - Objects may communicate with each other through functions

- New data and functions can easily be added whenever necessary
- Follows bottom-up approach

5. Define Object Oriented Programming(OOP)

Object Oriented programming is 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.

6. List out the key concepts of object – oriented programming.

There are several fundamental or key concepts in object-oriented programming. Some of them are

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

7. What are the advantages of OOP

OOP has the following advantages:

- Object-oriented programs can be comfortably upgraded.
- Using inheritance, redundant program codes can be eliminated and the use of previously defined classes may be continued.
- The technology of data hiding facilitates the programmer to design and develop safe programs that do not disturb code in other parts of the program.
- The encapsulation feature provided by OOP languages allows programmer to define the class with many functions and characteristics and only few functions are exposed to the user.
- All object-oriented programming languages can create extended and reusable parts of programs.

- Object-oriented programming enhances the thought process of a programmer leading to rapid development of new software in short span of time.

8. List out the usage of OOP

- Object-Oriented DBMS
- Office automation software
- AI and expert systems
- CAD/CAM software
- Network programming
- System software

9. Mention some of the limitations of C

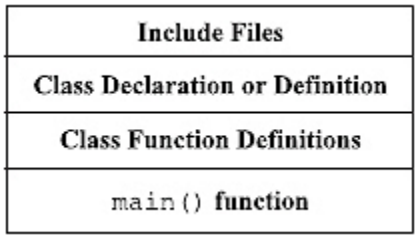
- Weak text processing capabilities
- Security and safety issues
- Weak memory management capabilities
- No built-in collections library
- No built-in support for networking, sound, graphics, etc

10. Define C++

- C++ is an object-oriented programming language developed by Bjarne Stroustrup at AT&T Bell Laboratories, Murray Hill, New Jersey (USA) in early 1980's.
- Stroustrup, a master of Simula67 and C, wanted to combine the features of both the languages into a more powerful language that could support object-oriented programming with features of C and he called the new language as ***'C with classes'.***
- However, in 1983, the name was changed to C++. The thought of C++ came from the C increment operator ++.
- Rick Mascitti coined the term C++ in 1983. Therefore, C++ is an extended version of C. C++ is a superset of C. All the concepts of C are applicable to C++ also.

11. Structure of the C++ program

A Program consists of different sections as shown in below figure



12. What are the input and output operators used in C++?

- The identifier cin is used for input operation. The input operator used is >>, which is known as the extraction or get from operator.

  syntax : cin >> n1;

- The identifier cout is used for output operation. The input operator used is <<, which is known as the insertion or put to operator.

  syntax: cout << ―C++ is better than C‖;

13. What are the operators available in C++?

All operators in C are also used in C++. In addition to insertion operator << and extraction operator >>, the other new operators in C++ are:

- : : Scope resolution operator
- : : * Pointer-to-member declarator
- ->* Pointer-to-member operator
- .* Pointer-to-member operator
- endl Line feed operator
- new Memory allocation operator
- setw Field width operators.

14. What is a scope resolution operator?

- Scope resolution operator is used to uncover the hidden variables. It also allows access to global version of variables.
- C++ provides the scope resolution operator to provide access to a global variable when it has been hidden by a local variable with the same name in a local scope.
- The scope resolution operator(: :) cannot be used to access a local variable of the same name in an outer block.

Eg:

```
#include<iostream.h>
int m=10; // global variable m
void main ( )
{
    int m=20; // local variable m
    cout<<"m="<<m<<"\n";
    cout<<": : m="<<: : m<<"\n";
}
```

output:

20

10 (:: m access global m)

Scope resolution operator is used to define the function outside the class.

Syntax:

Return type <class name> : : <function name>

Eg:

void x : : getdata()

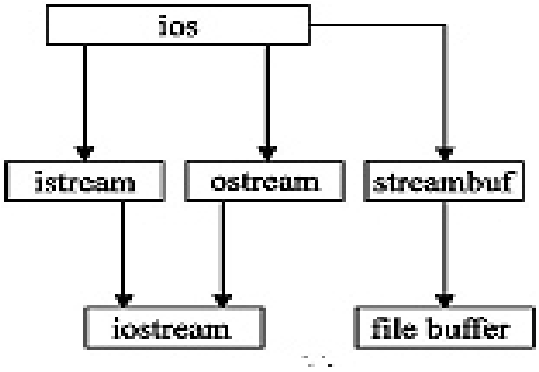
15. What is member-dereferencing operator?

C++ permits to access the class members through pointers. It provides three pointer-to-member operators for this purpose:

- : :* To declare a pointer to a member of a class.
- To access a member using object name and a pointer to the member
- ->* To access a member using a pointer to the object and a pointer to that member.

16. What are the stream classes in C++?

- C++ has a number of stream classes that are used to work with console and file operations. These classes are known as stream classes.
- All these classes are declared in the header file iostream.h.
- The file iostream.h must be included in the program if we are using functions of these classes.
- Below figure indicates the stream classes.



17. What is formatted data?

- Formatting means representation of data with different settings as per the requirement of the user.
- The various settings that can be done are number format, field width, decimal points etc.
- If the user needs to accept or display data in hexa-decimal format, manipulators with I/O functions are used.
- The data obtained or represented with these manipulators are known as formatted data.

18. What is unformatted data?

- The data accepted or printed with default setting by the I/O function of the language is known as unformatted data.
- For example, when the cin statement is executed it asks for a number. The user enters a number in decimal. For entering decimal number or displaying the number in decimal using cout statement the user won't need to apply any external setting.
- By default the I/O function represents number in decimal format. Data handled in such a way is called as unformatted data.

19. Mention some of the unformatted console I/O operations.

- Overloaded operators >> and<<
- put() and get() functions
- getline() and write() functions

20. Define Bit fields in C++

- The ios class contains the setf( ) member function. The flag indicates the format design. The syntax of the unsetf( ) function is used to clear the flags.
- The syntaxes of function are as follows.

Syntax: cout.setf(v1, v2);
Syntax: cout.unsetf(v1);

where, variable v1 and v2 are two flags. Below table describes the flag and bit-field setting that can be used with this function. The unsetf( ) accepts setting of v1 and v2 arguments.

Table Flags and Bits

Format	Flag (v1)	Bit-field (v2)
Left justification	ios::left	ios:adjustfield
Right justification	ios::right	ios:adjustfield
Padding after sign and base	ios::internal	ios:adjustfield
Scientific notation	ios::scientific	ios::floatfield

21. What are Manipulators?

- The output formats can be controlled using manipulators.
- The header file iomanip.h has a set of functions. Effect of these manipulators is the same as ios class member functions.
- Every ios member function has two formats. One is used for setting and second format is used to know the previous setting. But the manipulator does not return to the previous setting.
- The manipulator can be used with cout( ) statement as given below.

cout <<m1 <<m2 <<v1;

Here m1 and m2 are two manipulators and v1 is any valid C++ variable.

22. Define user-defined manipulators

The programmer can also define his/her own manipulator according to the requirements of the program.

The syntax for defining manipulator is given below:

```
ostream & m_name( ostream & o )
{
    statement1;
    statement2;
    return o;
}
```

The m_name is the name of the manipulator.

23. What are the Decision making statements?

Following are decision-making statements.

- The if statement
- The switch ( ) case statement

24. The if...else statement

The simple if statement executes statement only if the condition is true otherwise it follows the next statement. The else keyword is used when the expression is not true. The else keyword is optional.

The format of if..else statement is as follows:

```
if(the condition is true)
        execute the Statement1;
else
        execute the Statement2;
```

OR

Syntax of if – else statement can be given as follows:

```
if ( expression is true)
{
        statement 1;    ?    if block
        statement 2;
}
else
{
        statement 3;    ?    else block
        statement 4;
}
```

25. The nested if...else statement

- In this kind of statements, number of logical conditions are checked for executing various statements.
- If any logical condition is true, the compiler executes the block followed by if condition, otherwise it skips and executes else block.
- In if..else statement, else block is executed by default after failure of if condition.
- In order to execute the else block depending upon certain conditions we can add repetitively if statements in else block. This kind of nesting will be unlimited.

Syntax of if–else…if statement can be given as follows:

```
if ( condition)
{
      statement 1;       -> if  block
      statement 2;
}
else   if (condition)
{
      statement 3;       -> else block
      statement4;
}
else
{
```

```
        statement5;
    }
```

26. Mention the jump statements in C++

C/C++ has four statements that perform an unconditional control transfer. They are

- return( ),
- goto,
- break
- continue.

Of these,

- return( ) is used only in functions.
- The goto and return() may be used anywhere in the program but continue and break statements may be used only in conjunction with a loop statement.
- In 'switch case' 'break' is used most frequently.

27. Define goto statement

- goto statement does not require any condition.
- This statement passes control anywhere in the program without least care for any condition.
- The general format for this statement is shown below:

```
goto  label;
—
—
—
label:
```

where, label is any valid label either before or after goto. The label must start with any character and can be constructed with rules used for forming identifiers.

28. Write an example program to demonstrate goto statement

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

void main( )
{
        int x;
        clrscr( );
        cout <<"Enter a Number :";
        cin>>x;
        if (x%2==0)
                goto even;
        else
                goto odd;

        even :cout<<x<<" is Even Number.";
        return;
        odd: cout<<x<<" is Odd Number.";
}
```

29. Define break statement.

- The break statement allows the programmer to terminate the loop.
- The break skips from the loop or the block in which it is defined.
- The control then automatically passes on to the first statement after the loop or the block.
- The break statement can be associated with all the conditional statements (especially switch( ) case).

30. Define continue statement.

- The continue statement works quite similar to the break statement.
- Instead of forcing the control to end of loop (as it is in case of break), continue causes the control to pass on to the beginning of the block/loop.

- In case of for loop, the continue case causes the condition testing and increment steps to be executed while rest of the statements following continue are neglected.

- For while and do-while, continue causes control to pass on to conditional tests. It is useful in programming situation when you want particular iterations to occur only up to some extent or you want to neglect some part of your code

31. Define switch case statement

- The switch statement is a multi-way branch statement and an alternative to if-else-if ladder in many situations.
- This statement requires only one argument, which is then checked with number of case options.
- The switch statement evaluates the expression and then looks for its value among the case constants.
- If the value is matched with a case constant then that case constant is executed until a break statement is found or end of switch block is reached.
- If not then simply default (if present) is executed. If default isn't present then simply control flows out of the switch block.
- Every case statement terminates with: (colon). The break statement is used to stop the execution of succeeding cases and pass the control to the switch end of block.

32. Write an example program to demonstrate the nested switch...case statement

- C/C++ supports the nesting of switch( )case.The inner switch can be part of an outer switch.
- The inner and the outer switch case constants may be the same. No conflict arises even if they are same.
- The example below demonstrates this concept

```
#include<iostream.h>
#include<constream.h>
void  main( )
{
     int x;
     clrscr( );
```

```
        cout<<"\nEnter a number :";
        cin>>x;
        switch(x)
        {
                case 0:
                        cout<<"\nThe number is 0 ";
                break;

        default:
                int y;
                y=x%2;
                switch(y)
                {
                        case 0:
                                cout<<"\nThe number is even ";
                                break;
                        case 1:
                                cout<<"\nThe number is odd ";
                                break;
                }
        }
    }
```

OUTPUT

Enter a number :5

33. What are looping statements

C/C++ provides loop structures for performing some tasks which are repetitive in nature. The C/C++ language supports three types of loop control structures. They are

- for
- while
- do....while

34. Define for loop

- The for loop allows execution of a set of instructions repeatedly for a known number of times.
- Although all programming languages provide for loops, still the power and flexibility provided by C/C++ is worth.

35. Write a program to demonstrate nested for loop

We can also nest for loops to gain more advantage in some situations. The nesting level is not restricted at all. In the body of a for loop, any number of sub for loop(s) may exist.

```
# include <iostream.h>
# include <constream.h>>
void main( )
{
        int a,b,c;
        clrscr( );
        for (a=1;a<=2;a++) /* outer loop */
        {
                for (b=1;b<=2;b++) /* middle loop */
                {
                        for (c=1;c<=2;c++) /* inner loop */
                        {
                        cout <<"\n a="<<a <<" + b="<<b <<"+ c="<<c <<" :"<<a+b+c;
                        cout <<"\n Inner  Loop Over.";
                        }
                cout<<"\n Middle Loop Over.";
                }
        cout<<"\n Outer  Loop Over.";
        }
}
```

36. Define while loop

- This is simple and entry control loop structure. Its functionality is simply to repeat statement while the condition set in expression is true
- If the condition is true, the body is executed repeatedly until the test condition becomes false
- If the value is false, the control is transferred to the next statement

Syntax:

```
while (test condition)
{
  body of the loop
}
```

37. Define do...while loop

- It is the repetitive and exit control loop structure. Its functionality is exactly the same as the while loop
- The do-while loop is evaluated after the execution of the statement instead of before, granting at least one execution of statement of statement even if condition is never fulfilled.
- The format of do-while loop in C/C++ is given below.

Syntax:

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

38. Write a program using do-while loop to print numbers and their cubes to 10

```
# include <iostream.h>
# include <constream.h>
# include <math.h>
void main( )
{
        int y, x=1;
        clrscr( );
```

```
        cout <<"\n\tNumbers and their cubes \n";
        do
        {
                y=pow(x,3);
                cout <<"\t"<<x<<"\t\t"<<y<<"\n";
                x++;
        }  while (x<=10);
}
```

39. Write the syntax for if statement?

Syntax:

```
if(condition)
{
        Statements;
}
```

If the 'if' condition is true the statements are executed else the control is skipped from if construct

40. Write the syntax for if- else statement?

- It is two way decision process

Syntax:

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

if the condition becomes true the statements in 'if' part is executed else the rest of the statements in else part is executed

41 . Write the syntax for Nested if else statement?

Syntax:

```
if(condition 1)
{
    if(condition 2)
    {
            t-statement1;
    }
    else
    {
            f-statement1;
    }
}
else
{
    if( condition 3)
    {
            t-statement2;
    }
    else
    {
            f-statement2;
    }
}
statement-x;
```

42. Write the syntax for Switch case?

- It is multiway decision making process.

  Syntax:

```
switch(exp)
{
case 1:
{
        Statements;
        break;
}
```

```
case 2:
{
        Statements;
        break;
}
.
.
case n:
{
        sts;
        break;
}
default:
{
        sts;
        break;
}
}
```

The value of expression is compared with the case values and the corresponding case statements are executed else the default statements are executed.

43. Write the syntax for for loop?

Syntax:

```
for(inilization; condition; inc/dec)
{
  Statements;
}
```

The loop executes N times so that the statements are also executed n times until the condition fails

44. Write the syntax for Do While loop ?

Syntax:

```
do
```

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

45.Write the syntax for GOTO?

To transfer the control from one part one part of the program to another.

Syntax:

```
goto  variable_name
        variable_name
        {
                Statement  1;
                Statement  2;
                Statements  n;
        }
```

46.Write the syntax for BREAK STATEMENT :

To terminate the control from any constraint

SYNTAX:

```
        break;
```

47.Write the syntax for CONTINUE :

It is used to continue the same process even if it checks errors

SYNTAX :

```
            continue;
```

48.How the member functions are defined?

Member functions can be defined in two ways

1. Outside the class definition : Member function can be defined by using scope resolution operator::

   General format is

```
        return type class_name: : function-name (argument declaration)
        {
        }
```

2. Inside the class definition: This method of defining member function is to replace the function declaration by the actual function definition inside the class. It is treated as inline function.

```
Eg:
class item
{
        int a, b;
        void getdata (int x, int y)
        {
                a=x;
                b=y;
        }
};
```

49. What is inline function?

- The inline mechanism reduces overhead relating to accessing the member function.
- It provides better efficiency and allows quick execution of functions.
- An inline member function is similar to macros.
- Call to inline function in the program, puts the function code in the caller program. This is known as inline expansion

50. Mention the rules of inline function?

- Use inline functions rarely. Apply only under appropriate circumstances.
- Inline function can be used when the member function contains few statements.
- If function takes more time to execute, then it must be declared as inline

51. Define data hiding or encapsulation?

- Data hiding is also known as encapsulation.
- It is a process of forming objects.
- An encapsulated object is often called as an abstract data type.
- We need to encapsulate data because the programmer often makes various mistakes and the data gets changed accidentally.

- Thus, to protect data we need to construct a secure and impassable wall to protect the data.
- Data hiding is nothing but making data variable of the class or struct ***private.***

52.What is class?

- A *class* is a blueprint or template or set of instructions to build a specific type of object.
- Every object is built from a class.
- It is also grouping of objects having identical properties, common behaviour, and shared relationship.

53. What is an object?

- Objects are the basic run time entities in an object-oriented system.
- They may represent a person, a place, a bank account, a table of data or any item that the program has to handle.
- They may also represent user-defined data such as vectors, time and lists.

54.How to declare an object?

- A class declaration only builds the structure of object.
- The member variables and functions are combined in the class.
- The declaration of objects is same as declaration of variables of basic data types.
- Defining objects of class data type is known as ***class instantiation.***
- When objects are created only during that moment, memory is allocated to them.

  Consider the following examples:

  - int x,y,z; // Declaration of integer variables
  - char a,b,c; // Declaration of character variables
  - item a,b, *c; // Declaration of object or class type variables

- In example (a) three variables x, y and z of integer types are declared.
- In example (b) three variables a, b and c of char type are declared.

- In the same fashion the third example declares the three objects a, b and c of class item. The object *c is pointer to class item.

55.What is an static object?

- The keyword static can be used to initialize all class data member variables to zero.
- Declaring object itself as static can do this.
- Thus, all its associated members get initialized to zero.

56. Define array of objects.

- Arrays are collection of similar data types.
- Arrays can be of any data type including user-defined data type, created by using struct, class and typedef declarations.
- We can also create an array of objects. The array elements are stored in continuous memory locations.

Consider the following example:

```
class player
{
    private:
            char name [20];
            int age;

    public:
            void input (void);
            void display (void);
};
```

In the example given above player is a user-defined data type and can be used to declare an array of object of type player. Each object of an array has its own set of data variables.

57. Defining objects as function arguments

Similar to variables, objects can be passed on to functions. There are three methods to pass an argument to a function as given below:

(a) Pass-by-value

- In this type a copy of an object (actual object) is sent to function and assigned to object of callee function (Formal object).
- Both actual and formal copies of objects are stored at different memory locations. Hence, changes made in formal objects are not reflected to actual objects.

(b) Pass-by-reference

- Address of object is implicitly sent to function.

(c) Pass-by-address

- Address of the object is explicitly sent to function.

In pass by reference and address methods, an address of actual object is passed to the function. The formal argument is reference pointer to the actual object. Hence, changes made in the object are reflected to actual object. These two methods are useful because an address is passed to the function and duplicating of object is prevented.

58.What is friend function?

- C++ allows a mechanism, in which a non-member function has access permission to the private members of the class.
- This can be done by declaring a non-member function friend to the class whose private data is to be accessed. Here friend is a keyword.

  Consider the following example:

```
class ac
{
        private:
```

```
        char name [15];
        int acno;
        float bal;
public:
        void read( );
        friend void showbal( ); };
```

59.What is const member function?

- The member functions of a class can also be declared as constant using const keyword.
- The constant functions cannot modify any data in the class.
- The const keyword is suffixed to the function prototype as well as in function definition.
- If these functions attempt to change the data, compiler will generate an error message.

60.What is recursive member function?

- Like C, C++ language also supports recursive features i.e., function is called repetitively by itself.
- The recursion can be used directly or indirectly.
- The direct recursion function calls itself till the condition is true.
- In indirect recursion, a function calls another function and then the called function calls the calling function

61. What are local classes?

- When a class is declared inside a function they are called as local classes.
- The local classes have access permission to global variables as well as static variables.
- The global variables need to be accessed using scope access operator when the class itself contains member variable with same name as global variable.
- The local classes should not have static data member and static member functions. If at all they are declared, the compiler provides an error message.

62.What are empty, static and const classes?

- The classes without any data members or member functions are called as empty classes. These types of classes are not frequently used.

- The empty classes are useful in exception handling.

The syntax of empty class is as follows:

Empty Classes

```
class nodata {  };
class vacant  {  };
```

- We can also precede the declaration of classes by the keywords static, const, volatile etc. But there is no effect in the class operations. Such declaration can be done as follows:

Classes and other keywords

```
static class boys {  };
const class data { };
volatile class area{ };
```

63.Define functions in C++?

- One of the features of C/C++ language is that a large sized program can be divided into smaller ones. The smaller programs can be written in the form of functions.
- The process of dividing a large program into tiny and handy sub-programs and manipulating them independently is known as ***modular programming.***
- This technique is similar to divide-and-conquer technique. Dividing a large program into smaller functions provides advantages to the programmer. The testing and debugging of a program with functions becomes easier.
- A function can be called repeatedly based on the application and thus the size of the program can be reduced.
- The message passing between a caller (calling function) and callee (called function) takes places using arguments.

64.Mention the parts of function

A function has the following parts:

- Function prototype declaration

- Definition of a function (function declarator)
- Function call
- Actual and formal arguments
- The return statement?

65.How you pass arguments to the function?

- The main objective of passing argument to function is message passing. The message passing is also known as communication between two functions i.e., between caller and callee functions.
- There are three methods by which we can pass values to the function. These methods are.
    - Call by value (pass by value),
    - Call by address (pass by address), and
    - Call by reference (pass by reference).

66.What is LValue?

- A lvalue is an object locator. It is an expression that points to an object. An example of an lvalue expression is *k which results to a non-null pointer.
- A changeable lvalue is an identifier or expression that is related to an object that can be accessed and suitably modified in computer memory.
- A const pointer *to a constant* is not a changeable lvalue.

67.What is return by reference?

- A reference allows creating alias for the pre existing variable.
- A reference can also be returned by the function.
- A function that returns reference variable is in fact an alias for referred variable.
- This technique of returning reference is used to establish cascade of member functions calls in operator overloading.

68.Define default arguments?

- A function is called with all the arguments as declared in function prototype declaration and function definition.

- C++ compiler lets the programmer to assign default values in the function prototype declaration/function declaratory of the function.
- When the function is called with less parameter or without parameters, the default values are used for the operations.

69. Define function overloading.

- It is possible in C++ to use the same function name for a number of times for different intentions.
- Defining multiple functions with same name is known as function overloading or function polymorphism.
- The overloaded function must be different in its argument list and with different data types.

The examples of overloaded functions are given below.
All the functions defined should be equivalent to their prototypes.

- int sqr (int);
- float sqr (float);
- long sqr (long);

70.State two principles of function overloading.

- If two functions have the similar type and number of arguments (data type), the function cannot be overloaded. The return type may be similar or void, but argument data type or number of arguments must be different
- Passing constant values directly instead of variables also result in ambiguity.

11 MARKS

1. EXPLAIN IN DETAIL ABOUT EVOLUTION OF PROGRAMMING METHODOLOGIES

Evolution of programming methodologies

- The programming languages have evolved from machine languages, assembly languages to high level languages to the current age of programming tools. While machine level language and assembly language was difficult to learn for a layman, high level languages like C, Basic, Fortran and the like was easy to learn with more English like keywords. These languages were also known as procedural languages as each and every statement in the program had to be specified to instruct the computer to do a specific job.

- The procedural languages focused on organizing program statements into procedures or functions. Larger programs were either broken into functions or modules which had defined purpose and interface to other functions.

- Procedural approach for programming had several problems as the size of the softwares grew larger and larger. One of the main problems was data being completely forgotten. The emphasis was on the action and the data was only used in the entire process. Data in the program was created by variables and if more than one functions had to access data, global variables were used.

- The concept of global variables itself is a problem as it may be accidentally modified by an undesired function. This also leads to difficulty in debugging and modifying the program when several functions access a particular data.

- The object oriented approach overcomes this problem by modeling data and functions together there by allowing only certain functions to access the required data. The procedural languages had limitations of extensibility as there was limited support for creating user defined datatypes and defining how these datatypes will be handled.

- Another limitation of the procedural languages is that the program model is not closer to real world objects . For example, if you want to develop a gaming application of car race, what data would you use and what functions you would require is difficult questions to answer in a procedural approach.

- The object oriented approach solves this further by conceptualizing the problem as group of objects which have their own specific data and functionality. In the car game example, we would create several objects such as player, car, traffic signal and so on. Some of the languages that use object oriented programming approach are C++, Java, Csharp, Smalltalk etc.

2. STATE THE LIMITATIONS OF CONVENTIONAL PROGRAMMING AND CHARACTERISTICS OF OBJECT ORIENTED PROGRAMMING

Limitations in conventional programming

- Hard to maintain code.
- Software development period is high.
- Software cost is high.
- Hard to develop new software from existing one.
- It is difficult to implement all type of problems
- If end user requirement changes a big modification is needed in software
- More stress laid on procedures not on data
- Security of data is always risk in procedural approach
- A complex program can be solved easily and its size can be reduced by bottom up approach.
- Data hiding & Encapsulation
- In structure programming functions are separate.
- Structure programming follows Top down approach.
- Debugging of programs are hard
- Not Possible to handle run time errors and future enhancements
- Code Reusability
    - Inheritance and polymorphism makes the code easily reusable and extensible.
- Lower degree of Cohesion
    - Adding functionality to a certain part of code becomes almost impossible without affecting other parts.
- Non Flexibility of Code
    - In procedural programming, parts of code are interdependent. This interdependency or degree of coupling lowers the flexibility in modification & manipulation of a particular part of the program without affecting other parts.
- Removal of Ambiguity:
    - Results better design and maintenance
    - E.g. :- what '5' ?
        - Five
        - Odd no
        - Greater than 4 and less than 6

CHARACTERSTICS OF OBJECT ORIENTED PROGRAMMING

- In object oriented programming data and the instructor for processing that data are combined into a self-sufficient "object" that can be used in other programs.
- It is a programming methodology used to solve and develop complex problems by using oops principle (i.e.) Encapsulation, Inheritance, and Polymorphism.
- The characteristics are,
    - Emphasis on data rather than procedure
    - Programs are divided into objects
    - Data structures are designed such that they characterize the objects.

- Functions that operate on the data of an object are tied together in the data structure.
- It is easy to add new data and functions.
- Data is hidden and cannot be accessed by external functions.
- Follows bottom up design approach
- We can build programs from the standard working modules, which communicate with one another.
- The principle of data hiding helps the programmer to build secure programs that cannot be invaded by code in other parts of the programs
- It is possible to have multiple instances of an object of co-exist without any interference.
- It is easy to partition the work in a project based on objects.
- Software complexity can be easily managed
- It can be easily upgrade from small to large systems.

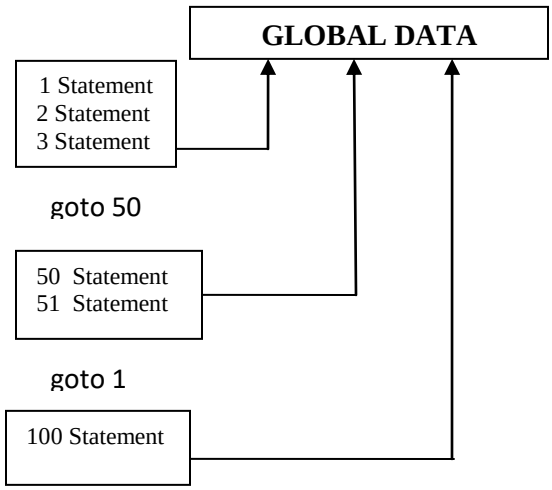
- Large size programs are divided into smaller programs known as functions. These functions can call one another. Hence, security is not provided.
- Importance is not given to security of data but on doing things.
- Data passes globally from one function to another.
- Most functions have access to global data.

3. EXPLAIN IN DETAIL ABOUT THE PROGRAMMING PARADIGMS

Programming Paradigms

(1) Monolithic Programming

- In monolithic programming languages such as BASIC and ASSEMBLY, the data variables declared are global and the statements are written in sequence.
- The program contains jump statements such as goto, that transfer control to any statement. The global data can be accessed from any portion of the program. Due to this reason the data is not fully protected.



Program in monolithic programming

- The concept of subprograms does not exist and hence useful for smaller programs.

(2) Procedural Programming

- In procedural programming languages such as FORTRAN and COBOL, programs are divided into a number of segments known as subprograms. Thus it focuses on functions apart from data.
- The control of program is transferred using unsafe goto statement.
- Data is global and all the subprograms share the same data.
- These languages are used for developing medium size applications.

(3) Structured Programming

- In structured programming languages such as PASCAL and C larger programs are developed. These programs are divided into multiple submodules and procedures.
- Each procedure has to perform different tasks.
- Each module has its own set of local variables and program code
- User-defined data types are introduced.

3. EXPLAIN IN DETAIL ABOUT THE KEY CONCEPTS IN OOPS

There are several fundamental or key concepts in object-oriented programming. Some of these are shown in below figure and are discussed below:

(1) Objects

OBJECT

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

- Objects are primary run-time entities in an object-oriented programming. They may stand for a thing that has specific application for example, a spot, a person, any data item related to program, including user-defined data types.

- Programming issues are analyzed in terms of object and the type of transmission between them. Figure 1.10 describes various objects.

- The selected program objects must be similar to actual world objects. Objects occupy space in memory. Every object has its own properties or features.

- An object is a specimen of a class. It can be singly recognized by its name. It declares the state shown by the data values of its characteristic at a specific time. The state of the object varies according to procedure used. It is known as the action of the object. The action of the object depends upon the member function defined within its class.

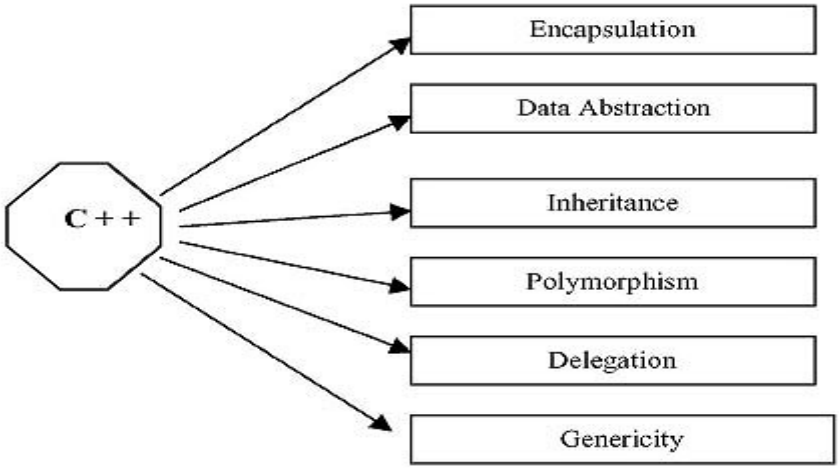


Fig. Features of object-oriented programming

Given below are some objects that we use in our daily life.

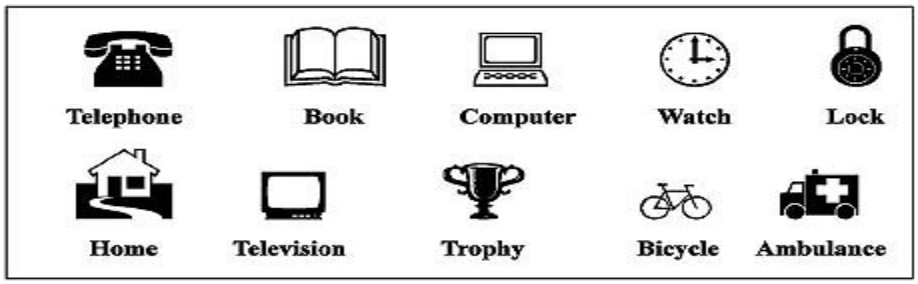


Fig. Commonly available objects

(2) Classes

CLASS

A class is grouping of objects having identical properties, common behavior, and shared relationship.

- A class is a grouping of objects having identical properties, common behaviour, and shared relationship. The entire group of data and code of an object can be built as a user-defined data type using class. Objects are nothing but variables of type class.
- Once a class has been declared, the programmer can create a number of objects associated with that class.
- The syntax used to create an object is similar to the syntax used to create an integer variable in C.
- A class is a model of the object.
- Every object has its own value for each of its member variables.
- However, it shares the property names or operations with other instances of the class. Thus, classes define the characteristics and actions of different objects.

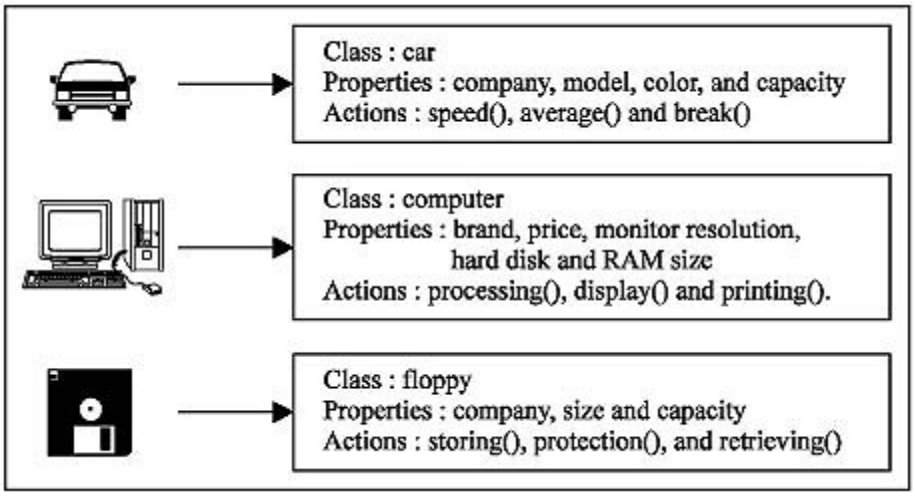


Fig. Objects and their properties

In Figure 1.11 some common objects, their properties, and the actions they perform have been described.

A car can be distinguished from another car on the basis of its properties, for example, company, model, color, capacity etc. and also on the basis of its actions for example, speed, average, break etc.

(3) Method

METHOD

An operation required for an object or entity when coded in a class is called a method.

- An operation required for an object or entity when coded in a class is called a method. The operations that are required for an object are to be defined in a class.
- All objects in a class perform certain common actions or operations. Each action needs an object that becomes a function of the class that defines it and is referred to as a method.
- In below figure, the class, its associated data members, and functions are shown in different styles. This style is frequently used while writing a class in the program.

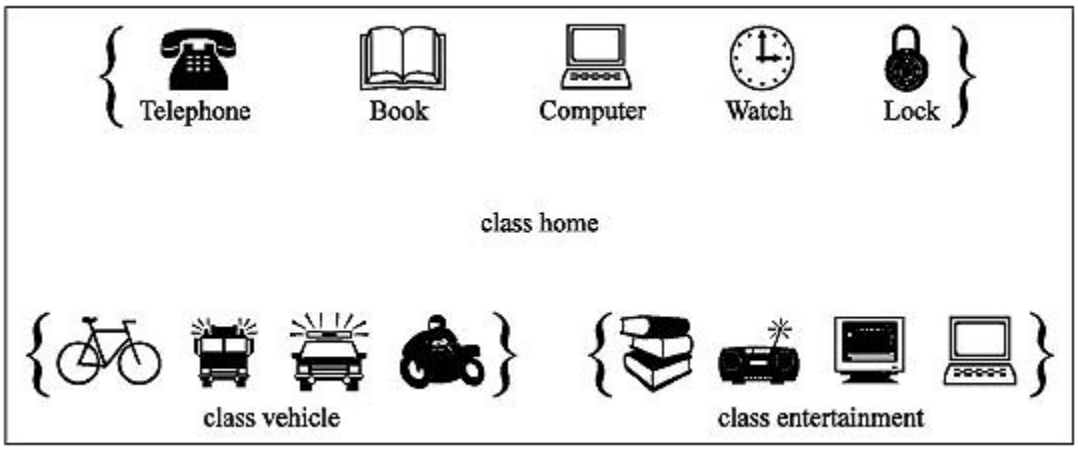


Fig. Classes and their members

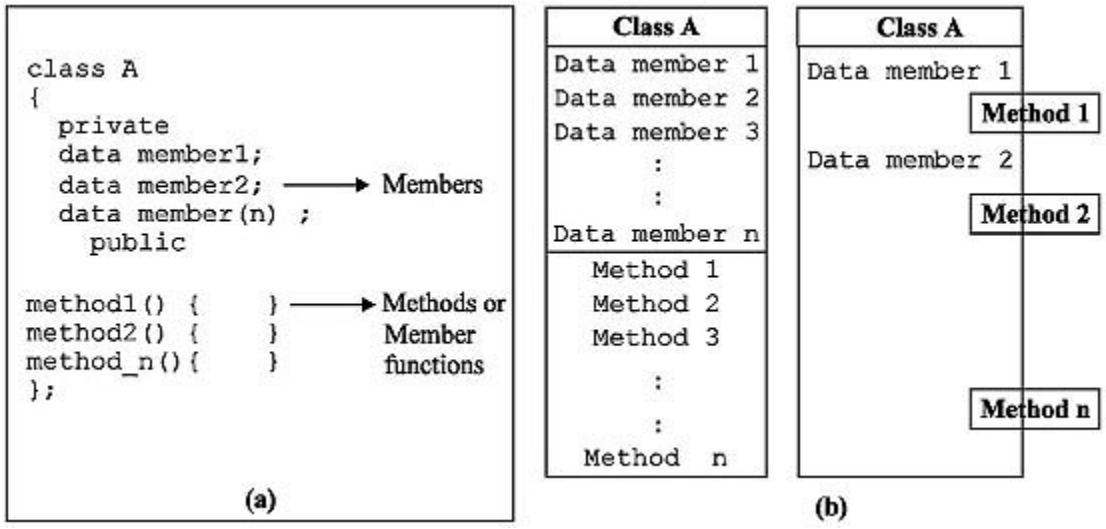


Fig. Representation of methods in different manners

(4) Data Abstraction

DATA ABSTRACTION

Abstraction directs to the procedure of representing essential features without including the background details.

- Abstraction directs to the procedure of representing essential features without including the background details. Classes use the theory of abstraction and are defined as a list of abstract properties such as size, cost, height, and few functions to operate on these properties.

- Data abstraction is the procedure of identifying properties and methods related to a specific entity as applicable to the application.

(a) Computer as a single unit (b) Different components of a computer

(5) Encapsulation

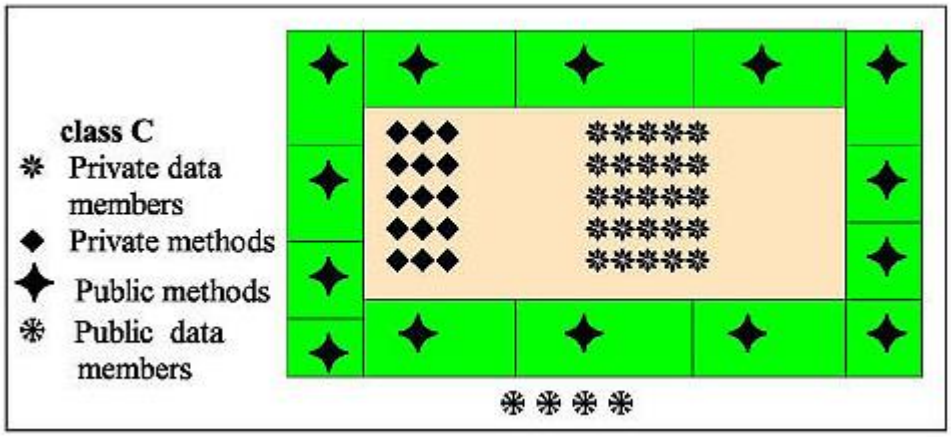
ENCAPSULATION

The packing of data and functions into a single component is known as encapsulation.

- C++ supports the features of encapsulation using classes. The packing of data and functions into a single component is known as encapsulation.

- The data is not accessible by outside functions. Only those functions that are able to access the data are defined within the class. These functions prepare the interface between the object's data and the program.

- With encapsulation data hiding can be accomplished. Data hiding is an important feature. By data hiding an object can be used by the user without knowing how it works internally.



(6) Inheritance

INHERITANCE

Inheritance is the method by which objects of one class get the properties of objects of another class.

- Inheritance is the method by which objects of one class get the properties of objects of another class.
- In object-oriented programming, inheritance provides the thought of reusability.
- The programmer can add new properties to the existing class without changing it. This can be achieved by deriving a new class from the existing one. The new class will possess features of both the classes.
- The actual power of inheritance is that it permits the programmer to reuse a class that is close to what he wants, and to tailor the class in such a manner that it does not bring any unwanted incidental result into the rest of the class.
- Thus, inheritance is the feature that permits the reuse of an existing class to make a new class. The below figure shows an example of inheritance.

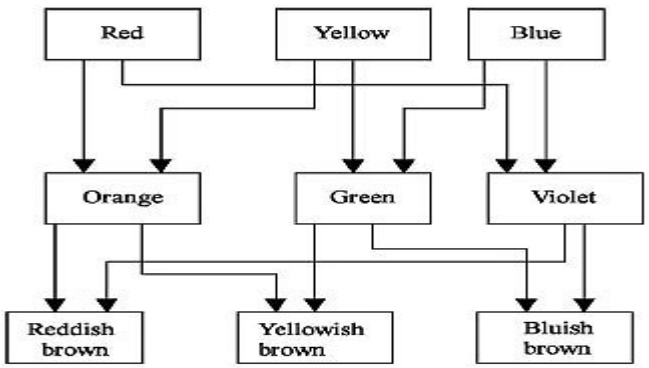


Fig. Inheritance

(7) Polymorphism

POLYMORPHISM

Polymorphism allows the same function to act differently in different classes.

- Polymorphism makes possible the same functions to act differently on different classes as shown in below figure.
- It is an important feature of OOP concept and has the ability to take more than one form.
- Polymorphism accomplishes an important part in allowing objects of different classes to share the same external interface.
- It is possible to code a non-specific (generic) interface to a set of associated actions.

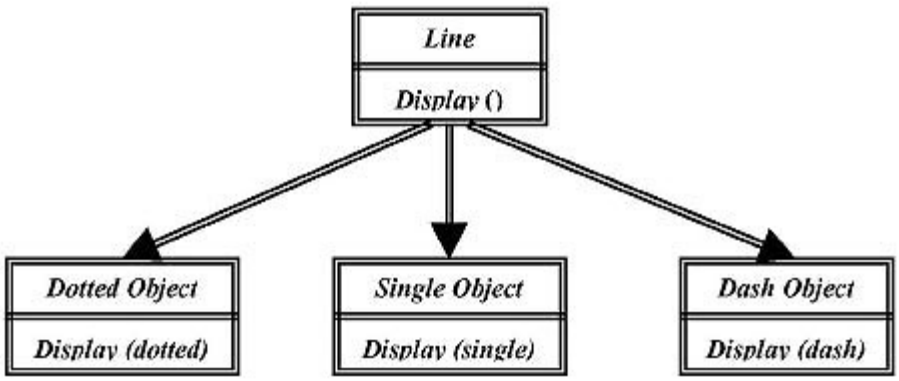


Fig. Polymorphism in OOP

(8) Dynamic Binding

DYNAMIC BINDING

Binding means connecting one program to another program that is to be executed in reply to the call.

- Binding means connecting one program to another program that is to be executed in reply to the call. Dynamic binding is also known as *late binding.*
- The code present in the specified program is unknown till it is executed. It is analogous to inheritance and polymorphism.
- polymorphism allows single object to invoke similar function from different classes. The program action is different in all the classes. During execution time, the code analogous to the object under present reference will be executed.

(9) Message Passing

Object-oriented programming includes objects which communicate with each other. Programming with these objects should be followed in steps shown below:

(1) Declaring classes that define objects and their actions.
(2) Declaring objects from classes.
(3) Implementing relation between objects.

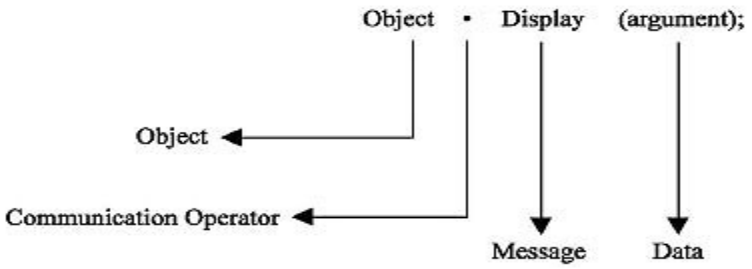


Fig. Message passing

- Data is transferred from one object to another. A message for an object is the demand for implementation of the process.

- Message passing as shown in above figure contains indicating the name of the object, function, and required data elements.

(10) Reusability

REUSABILITY

Object-oriented technology allows reusability of the classes by extending them to other classes using inheritance.

- Object-oriented technology allows reusability of classes by extending them to other classes using inheritance. Once a class is defined, the other programmer can also use it in their programs and add new features to the derived classes.

- Once verified the qualities of base classes need not be redefined.

- Thus, reusability saves time. In below figure class A is reused and class B is created. Again class B is reused and class C is created.

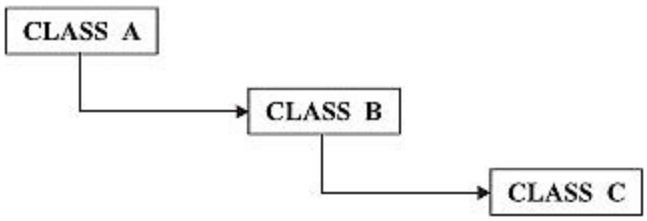


Fig. Reusability

4. STATE THE BENEFITS AND APPLICATIONS OF OOP.

Benefits of OOP

- OOP offers several benefits to both the program designer and the user. Object-Orientation contributes to the solution of many problems associated with the development and quality of software products. The new technology promises greater programmer productivity, better quality of software and lesser maintenance cost.

The principal advantages are:

- Through inheritance, we can eliminate redundant code extend the use of existing Classes.
- 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.
- The principle of data hiding helps the programmer to build secure program that cannot be invaded by code in other parts of a programs.
- It is possible to have multiple instances of an object to co-exist without any interference.
- It is possible to map object in the problem domain to those in the program.
- It is easy to partition the work in a project based on objects.
- The data-centered design approach enables us to capture more detail of a model can implemental form.
- Object-oriented system can be easily upgraded from small to large system.
- Message passing techniques for communication between objects makes to interface descriptions with external systems much simpler.
- Software complexity can be easily managed.

Applications of OOP

- Object-oriented technology is changing the style of software engineers to think, analyze, plan, and implement the software. The software developed using OOP technology is more efficient and easy to update. OOP languages have standard class library. The users can reuse the standard class library in their program. Thus, it saves lot of time and coding work.
- The most popular application of object-oriented programming is windows. There are several windowing software based on OOP technology. Following are the areas for which OOP is considered

1) Object-Oriented DBMS 2) Office automation software 3) AI and expert systems
4) CAD/CAM software 5) Network programming 6) System software

5. WRITE IN DETAIL ABOUT C++

Introduction to C++

- C++ is an object-oriented programming language. It was developed by Bjarne Stroustrup at AT&T Bell Laboratories in Murray Hill, New Jersey, USA, in the early 1980's. Stroustrup, an admirer of Simula67 and a strong supporter of C, wanted to combine the best of both the languages and create a more powerful language that could support object -oriented programming features and still retain the power and elegance of C.
- The result was C++. Therefore, C++ is an extension of C with a major addition of the class construct feature of Simula67. Since the class was a major addition to the original C language, Stroustrup initially called the new language 'C with classes'. However, later in 1983, the name was changed to C++. The idea of C++ comes from the C increment operator ++, thereby suggesting that C++ is an augmented version of C.
- C+ + is a superset of C. Almost all c programs are also C++ programs. However, there are a few minor differences that will prevent a c program to run under C++ complier. We shall see these differences later as and when they are encountered.
- The most important facilities that C++ adds on to C care classes, inheritance, function overloading and operator overloading. These features enable creating of abstract data types, inherit properties from existing data types and support polymorphism, thereby making C++ a truly object-oriented language.

Application of C++

C++ is a versatile language for handling very large programs; it is suitable for virtually any programming task including development of editors, compilers, databases, communication systems and any complex real life applications systems.

- Since C++ allow us to create hierarchy related objects, we can build special object-oriented libraries which can be used later by many programmers.
- While C++ is able to map the real-world problem properly, the C part of C++ gives the language the ability to get closed to the machine-level details.
- C++ programs are easily maintainable and expandable. When a new feature needs to be implemented, it is very easy to add to the existing structure of an object.
- It is expected that C++ will replace C as a general-purpose language in the near future.

Simple C++ Program

- Let us begin with a simple example of a C++ program that prints a string on the screen.

Printing A String

```
#include<iostream>
Using namespace std;
int main()
{
cout<<" c++ is better than c \n";
 return 0;
}
```

- This simple program demonstrates several C++ features.

Program feature

- Like C, the C++ program is a collection of function. The above example contain only one function **main().** As usual execution begins at main(). Every C++ program must have a **main()**. C++ is a free form language. With a few exception, the compiler ignore carriage return and white spaces. Like C, the C++ statements terminate with semicolons.

Comments

- C++ introduces a new comment symbol // (double slash). Comment start with a double slash symbol and terminate at the end of the line. A comment may start anywhere in the line, and whatever follows till the end of the line is ignored. Note that there is no closing symbol.

- The double slash comment is basically a single line comment. Multiline comments can be written as follows:

// This is an example of
// C++ program to illustrate
// some of its features

- The C comment symbols /*,*/ are still valid and are more suitable for multiline comments. The following comment is allowed:

/* This is an example of C++ program to illustrate some of its features*/

Output operator

- The only statement in program 1.10.1 is an output statement. The statement

Cout<<"C++ is better than C.";

- Causes the string in quotation marks to be displayed on the screen. This statement introduces two new C++ features, cout and <<. The identifier cout(pronounced as C out) is a predefined object that represents the standard output stream in C++. Here, the standard output stream represents the screen. It is also possible to redirect the output to other output devices. The operator << is called the insertion or put to operator.

The iostream File

- We have used the following #include directive in the program:

#include <iostream>

- The #include directive instructs the compiler to include the contents of the file enclosed within angular brackets into the source file. The header file **iostream.h** should be included at the beginning of all programs that use input/output statements.

Namespace

- Namespace is a new concept introduced by the ANSI C++ standards committee. This defines a scope for the identifiers that are used in a program. For using the identifier defined in the **namespace** scope we must include the using directive, like

  Using namespace std;

- Here, std is the namespace where ANSI C++ standard class libraries are defined. All ANSI C++ programs must include this directive. This will bring all the identifiers defined in std to the current global scope. **Using** and **namespace** are the new keyword of C++.

Return Type of main()

- In C++, main () returns an integer value to the operating system. Therefore, every main () in C++ should end with a return (0) statement; otherwise a warning an error might occur. Since main () returns an integer type for main () is explicitly specified as **int.** Note that the default return type for all function in C++ is **int.** The following main without type and return will run with a warning:

```
main ()
{
  ……..
  …….
}
```

More C++ Statements

Let us consider a slightly more complex C++ program. Assume that we should like to read two numbers from the keyboard and display their average on the screen. C++ statements to accomplish this is shown in this program

AVERAGE OF TWO NUMBERS

```
#include<iostream.h> // include header file

Using namespace std;

Int main()

{

    Float number1, number2,sum, average;
    Cin >> number1;       // Read Numbers
    Cin >> number2;       // from keyboard
    Sum = number1 + number2;
    Average = sum/2;
    Cout << ”Sum = “ << sum << “\n”;
    Cout << “Average = “ << average << “\n”;
```

```
        Return 0;

    } //end of example
```

The output would be:
Enter two numbers: 6.5 7.5
Sum = 14
Average = 7

Variables

The program uses four variables number1, number2, sum and average. They are declared as type float by the statement.

float number1, number2, sum, average;

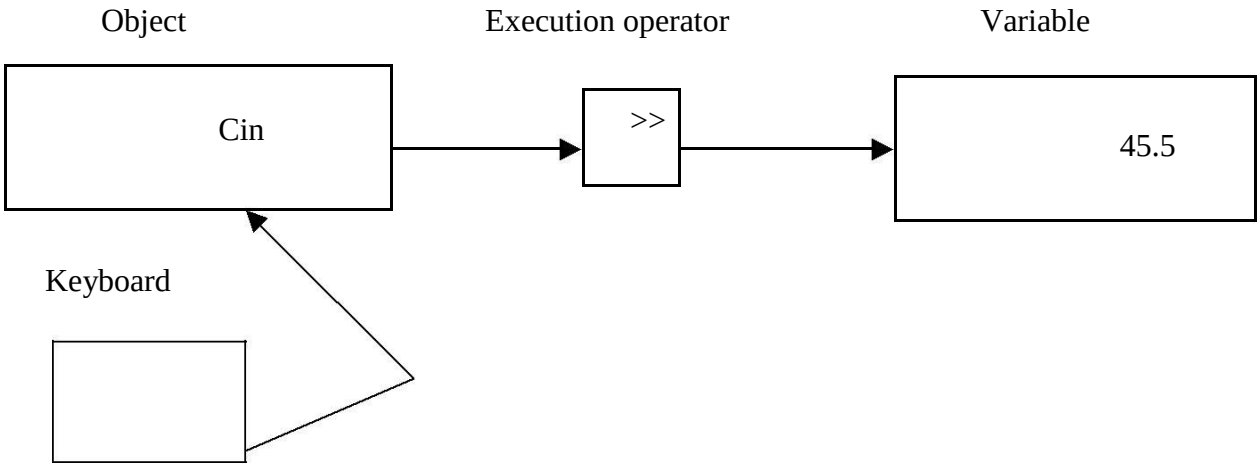
All variable must be declared before they are used in the program.

Input Operator

The statement

cin >> number1;

- Is an input statement and causes the program to wait for the user to type in a number. The number keyed in is placed in the variable number1. The identifier cin (pronounced 'C in') is a predefined object in C++ that corresponds to the standard input stream. Here, this stream represents the keyboard.
- The operator >> is known as extraction or get from operator. It extracts (or takes) the value from the keyboard and assigns it to the variable on its right fig 1.8. This corresponds to a familiar scanf() operation. Like <<, the operator >> can also be overloaded.



1.8 Input using extraction operator

Cascading of I/O Operators

- We have used the insertion operator << repeatedly in the last two statements for printing results.
- The statement **Cout << "Sum = " << sum << "\n";**
- First sends the string "Sum = " to cout and then sends the value of sum. Finally, it sends the newline character so that the next output will be in the new line. The multiple use of << in one statement is called cascading. When cascading an output operator, we should ensure necessary blank spaces between different items. Using the cascading technique, the last two statements can be combined as follows:

  Cout << "Sum = " << sum << "\n"
  << "Average = " << average << "\n";

- This is one statement but provides two line of output. If you want only one line of output, the statement will be:

  Cout << "Sum = " << sum << ","
  << "Average = " << average << "\n";

- *The output will be*: Sum = 14, average = 7
- We can also cascade input iperator >> as shown below**: Cin >> number1 >> number2;**
- The values are assigned from left to right. That is, if we key in two values, say, 10 and 20, then 10 will be assigned to munber1 and 20 to number2.

An Example with Class

- One of the major features of C++ is classes. They provide a method of binding together data and functions which operate on them. Like structures in C, classes are user-defined data types.
- This shows the use of class in a C++ program.

USE OF CLASS

```
#include<iostream.h> // include header file

using namespace std; class person
{

        char name[30]; Int age;

        public:
            void getdata(void); void
            display(void);
};
```

```
void person :: getdata(void)
{
        cout << "Enter name: "; cin >> name;
        cout << "Enter age: "; cin >> age;

}
Void person : : display(void)
{
        cout << "\nNameame: " << name; cout << "\nAge:
        " << age;
}

int main()
{
        person p; p.getdata();
        p.display();

        return 0;

} //end of example
```

The output of program is:

```
Enter Name: Ravinder
Enter age:30
Name:Ravinder
Age: 30
```

- The program define **person** as a new data of type class. The class person includes two basic data type items and two function to operate on that data. These functions are called **member function**. The main program uses **person** to declare variables of its type. As pointed out earlier, class variables are known as objects. Here, p is an object of type **person**. Class object are used to invoke the function defined in that class.

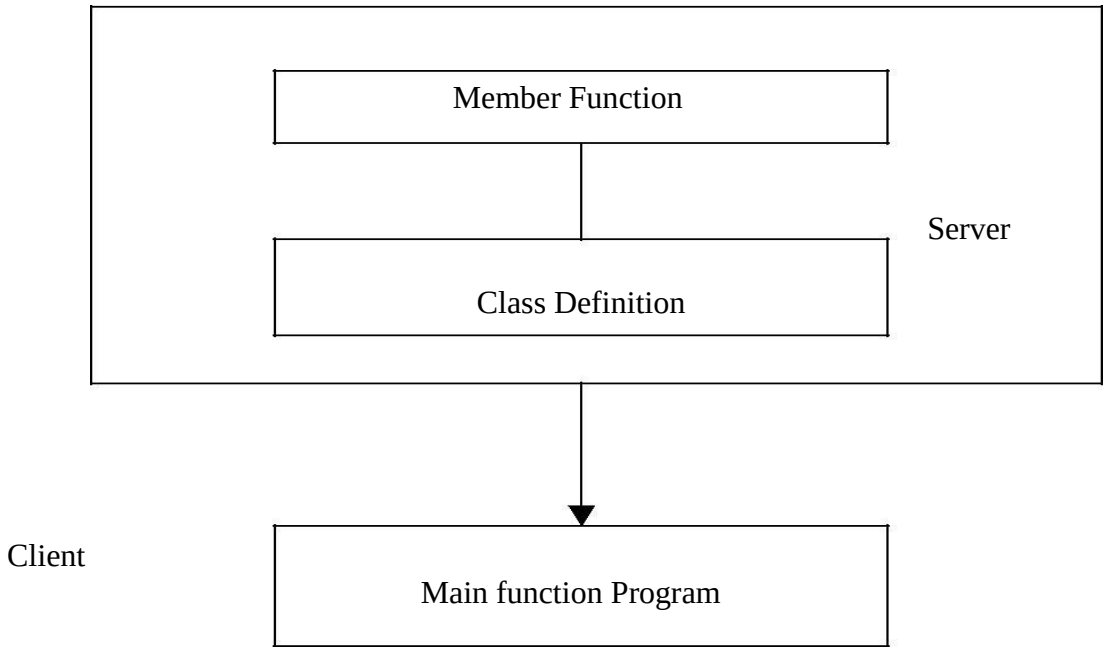
Structure of C++ Program

- This section may be placed in separate code files and then compiled independently or jointly. It is a common practice to organize a program into three separate files.

Include Files
Class declaration
Member functions definitions
Main function program

- This approach is based on the concept of client-server model as shown in fig. 1.10. The class definition including the member functions constitute the server that provides services to the main program known as client. The client uses the server through the public interface of the class.

The client-server model



Creating the Source File

- Like C programs can be created using any text editor. Foe example, on the UNIX, we can use vi or ed text editor for creating using any text editor for creating and editing the source code. On the DOS system, we can use endlin or any other editor available or a word processor system under non-document mode.

- Some systems such as Turboc C++ provide an integrated environment for developing and editing programs
- The file name should have a proper file extension to indicate that it is a C++ implementations use extensions such as .c,.C, .cc, .cpp and .cxx. Turboc C++ and Borland C++ use .c for C programs and .cpp(C plus plus) for C++ programs. Zortech C++ system use .cxx while UNIX AT&T version uses .C (capital C) and .cc. The operating system manuals should be consulted to determine the proper file name extension to be used.

Compiling and Linking

- The process of compiling and linking again depends upon the operating system. A few popular systems are discussed in this section.

Unix AT&T C++

- This process of implementation of a C++ program under UNIX is similar to that of a C program. We should use the "cc" (uppercase) command to compile the program. Remember,

we use lowercase “cc” for compiling C programs. The command

CC example.C

- At the UNIX prompt would compile the C++ program source code contained in the file **example.C**. The compiler would produce an object file **example.o** and then automatically link with the library functions to produce an executable file. The default executable filename is **a. out**.

- A program spread over multiple files can be compiled as follows:

CC file1.C file2.o

- The statement compiles only the file **file1.C** and links it with the previously compiled **file2.o** file. This is useful when only one of the files needs to be modified. The files that are not modified need not be compiled again.

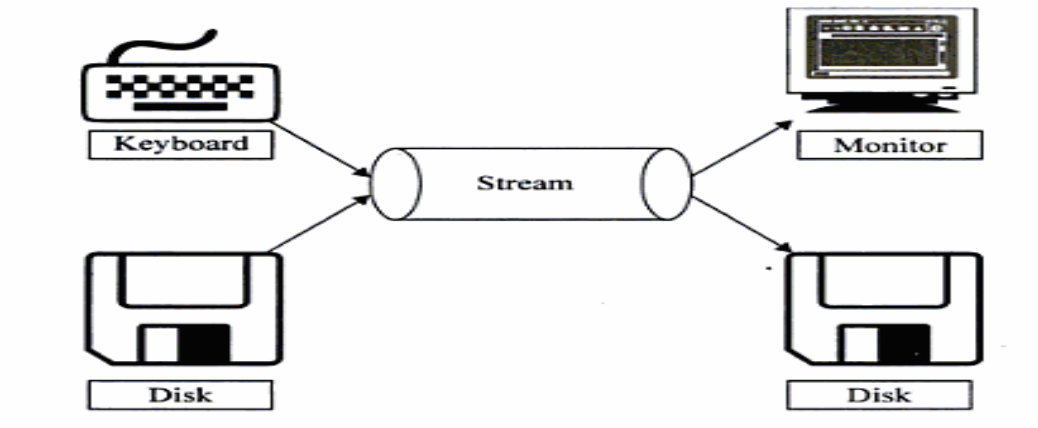
Turbo C++ and Borland C++

- Turbo C++ and Borland C++ provide an integrated program development environment under MS DOS. They provide a built-in editor and a menu bar includes options such as File, Edit, Compile and Run.

- We can create and save the source files under the **File option,** and edit them under the **Edit option.** We can then compile the program under the **compile** option and execute it under the **Run option.** The **Run option** can be used without compiling the source code.

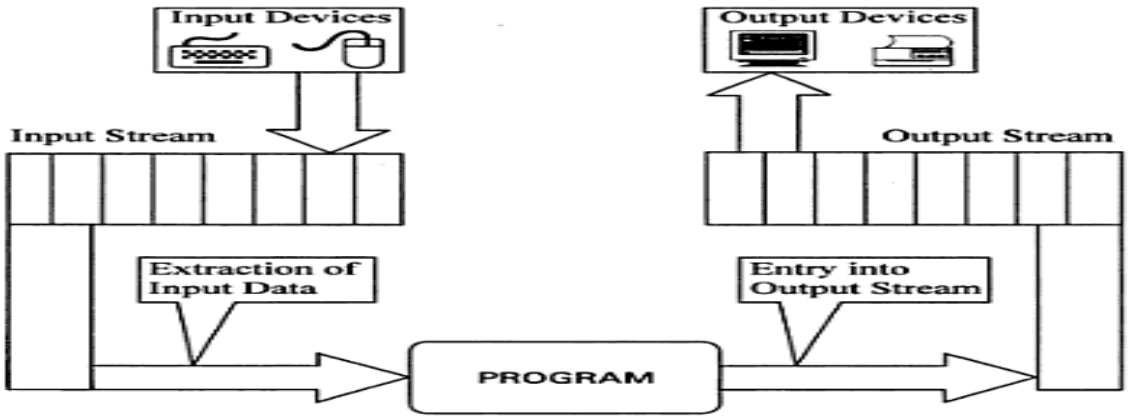
6. WRITE IN DETAIL ABOUT STREAM CLASSES IN C++

- C++ supports a number of Input/Output (I/O) operations to read and write operations.
- These C++ I/O functions make it possible for the user to work with different types of devices such as keyboard, disk, tape drivers etc.
- The stream is an intermediator between I/O devices and the user.
- The standard C++ library contains the I/O stream functions.
- Stream is flow of data in bytes in sequence.
- If data is received from input devices in sequence then it is called as *source stream*
- When the data is passed to output devices then it is called as *destination stream.*
- It is also referred as encapsulation through streams.
- The data is received from keyboard or disk and can be passed on to monitor or to the disk.



MECHANISM OF I/O STREAM

- Data in source stream can be used as input data by the program. So source stream is also called input stream
- The Destination stream that collects the output data from the program is known as output stream .



PRE-DEFINED STREAMS

- C++ has a number of pre-defined streams.
- These pre-defined streams are also called as standard I/O objects.
- These streams are automatically activated when the program execution starts

STREAMS	DESCRIPTION
CIN	Standard input, usually keyboards, corresponding to stdin in C. It handles input from input devices usually from keyboard

COUT	Standard output, usually screen, corresponding to stdout in C. It passes data to output devices such as monitor and printers. Thus, it controls output.
CLOG	A fully buffered version of cerr (no C equivalent). It controls error messages that are passed from buffer to the standard error device.
CERR	Standard error output, usually screen, corresponding to stderr in C. It controls the unbuffered output data. It catches the errors and passes to standard error device monitor

Program to display message using pre-defined objects

```
#include<iostream.h>
#include<conio.h>
void main()
{
        clrscr();
        cout <<"\nSTREAMS";
        cerr <<"\nSTREAMS";
        clog <<"\nSTREAMS";
}
```

OUTPUT:

STREAMS
STREAMS
STREAMS

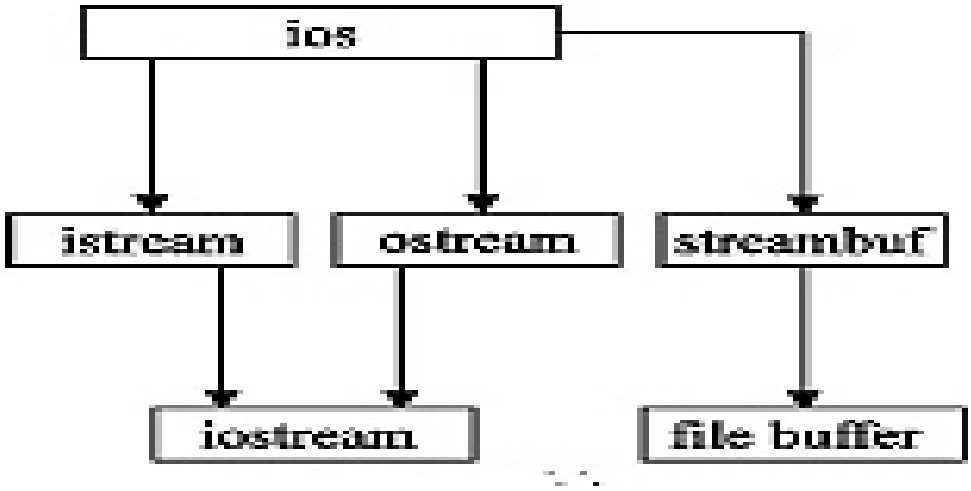
BUFFERING:

- It is a block of memory used to hold data temporarily.
- It is always located between a peripheral device and faster computer.
- Buffers are used to pass data between computer and devices.
- The buffer increases the performance of the computer by allowing read/write operation in larger chunks.
- For example, if the size of buffer is N, the buffer can hold *N* number of bytes of data

STREAM CLASSES IN C++

- C++ streams are based on class and object theory.
- C++ has a number of stream classes that are used to work with console and file operations.
- These classes are known as stream classes.

- All these classes are declared in the header file iostream.h.
- The file iostream.h must be included in the program if we are using functions of these classes
- The classes istream and ostream are derived classes of base class ios.
- The ios class contains member variable object streambuf.
- The streambuf places the buffer.
- The member function of streambuf class handles the buffer by providing the facilities to flush clear and pour the buffer.
- The class iostream is derived from the classes istream and ostream by using multiple inheritance



FORMATTED AND UNFORMATTED DATA

FORMATTED DATA

- Formatting means representation of data with different settings as per the requirement of the user.
- The various settings that can be done are number format, field width, decimal points etc.
- If the user needs to accept or display data in hexa-decimal format, manipulators with I/O functions are used.
- The data obtained or represented with these manipulators are known as formatted data.

UNFORMATTED DATA

- The data accepted or printed with default setting by the I/O function of the language is known as unformatted data.

- For example, when the cin statement is executed it asks for a number. The user enters a number in decimal. For entering decimal number or displaying the number in decimal using cout statement the user won't need to apply any external setting.
- By default the I/O function represents number in decimal format. Data handled in such a way is called as unformatted data.

UNFORMATTED CONSOLE I/O OPERATIONS

INPUT AND OUTPUT STREAMS

- The input stream uses *cin* object to read data.
- The output stream uses *cout* object to display data on the screen.
- The *cin* and *cout* are predefined streams for input and output of data.
- The data type is identified by these functions using operator overloading of the operators << (insertion operator) and >> *(extraction operator).*
- The operator << is overloaded in the ostream *class* and the operator >> is overloaded in istream *class.*

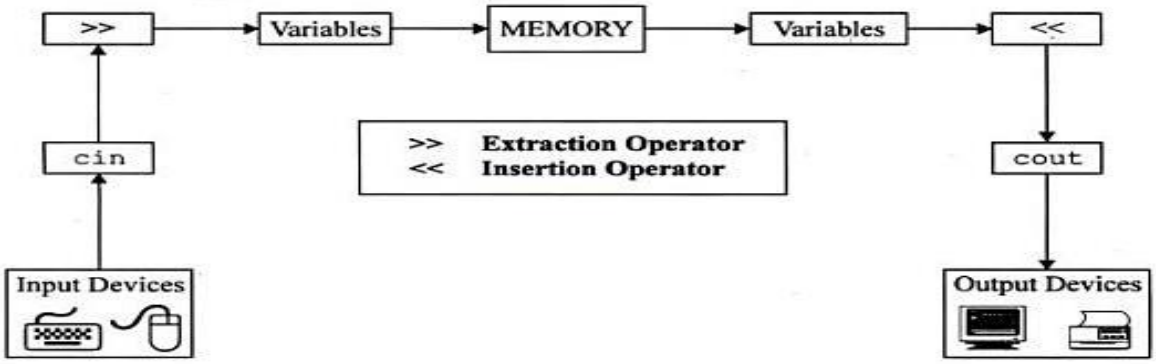


Fig. 2.7 Working of cin and cout statements

INPUT STREAM

- The input stream does read operation through keyboard.
- It uses cin as object.
- The cin statement uses >> (extraction operator) before variable name.
- The cin statement is used to read data through the input device

SYNTAX:

cin>>variable;

EXAMPLE

```
int weight;
cin>>weight            // Reads integer value

float height;
cin>>height;           // Reads float value

double volume;
cin>>volume;           // Reads double value
```

- The operator >> accepts the data and assigns it to the memory location of the variables.
- Each variable require >> operator
- Like scanf() statement, cin does not require control string like %d for integer, %f for float and so on.

CASCADING CIN:

cin>>v1>>v2>>v3;

OUTPUT STREAM

- The output stream manages output of the stream i.e., displays contents of variables on the screen.
- It uses << insertion operator before variable name.
- It uses the cout object to perform console write operation.

SYNTAX:

cout<<variable;

EXAMPLE

```
int weight;
cout<<weight          // Displays integer value

float height;
cout<<height;         // Displays float value

double volume;
cout<<volume;         // Displays double value


char result[10];
cout<<result;         // Reads char string
```

OUTPUT STREAM

- The syntax rules are same as cin.
- cout statement uses the insertion operator <<.
- cout does not use the format specification like %d, %f as used in c and so on.
- The cout statement allows us to use all 'C' escape sequences like '\n', '\t', and so on.

CASCADING COUT:

cout<<v1<<v2<<v3;

EXAMPLE PROGRAM:

2.5 Write a program to accept string through the keyboard and display it on the screen. Use cin and cout statements.

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

main()
{
   char name[15];
   clrscr();
   cout <<"Enter Your Name :";
   cin >>name;
   cout <<"Your name is "<<name;
   return 0;
}
```

OUTPUT

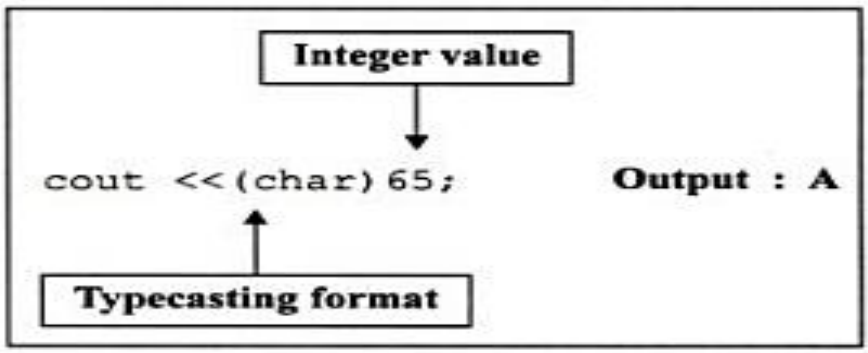
Enter Your Name :Amit

Your name is Amit

TYPE CASTING WITH COUT

Typecasting refers to the conversion of data from one basic type to another by applying external use of data type keywords

Syntax:



Example:

```
# include <iostream.h>
# include <conio.h>
void main( )
{
        int a=66;
        float f=2.5;
        double d=85.22;
        char c='K';
        clrscr( );
        cout <<"\n int in char format    : "<<(char)a;
        cout <<"\n float in int format   : "<<(int)f;
        cout <<"\n double in char format : "<<(char)d;
        cout <<"\n char in int format    : "<<(int)c;
}
```

OUTPUT

```
int in char format : B
float in int format : 2
double in char format : U
char in int format : 75
```

4. EXPLAIN IN DETAIL ABOUT FORMATTED

- Formatting means representation of data with different settings as per the requirement of the user. The various settings that can be done are number format, field width, decimal points etc. If the user needs to accept or display data in hexa-decimal format, manipulators with I/O functions are used. The data obtained or represented with these manipulators are known as formatted data.

Formatted Console I/O Operations:-

C++ supports a number of features that could be used for formatting the output. These features include:

- ios class function and flags.
- manipulators.

- User-defined output functions.

The ios class contains a large number of member functions that would help us to format the output in a number of ways.The most importany ones among them are listed in below table
ios format functions

Function	**Task**
Width()	To specify the required field size for displaying an output Value
Precision()	To specify the number of digits to be displayed after the decimal point of float value
Fill()	To specify a character that is used to fill the unused portion of a field
Setf()	To specify format flags that can control the form of output display(such as left-justification and right-justification)
Unsetf()	To clear the flags specified

Manipulators are special functions that can be included in the I/O statements to alter the format parameter of stream .Table 5.3 shows some important manipulator functions that are frequently used. To access these manipulators, the file iomanip should be included in the program.

Table Manipulators

Manipuators	**Equivalent ios function**
setw()	width()
setprecision()	precision()
setfill()	fill()
setiosflags()	setf()
resetiosflags()	unsetf()

In addition to these standard library manipulators we can create our own manipulator functions to provide any special output formats.

Defining Field Width: width()

- The width() function is used to define the width of a field necessary for the output of an item. As it is a member function object is required to invoke it like

cout.width(w);

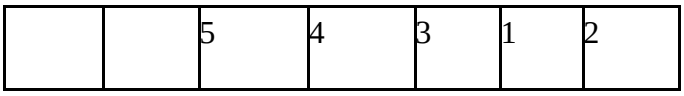
- here w is the field width. The output will be printed in a field of w character wide at the right end of field. The width() function can specify the field width for only one item(the item that follows immediately).After printing one item(as per the specification) it will revert back the default.

for example, the statements

cout.width(5);

cout<<543<<12<<"\n";

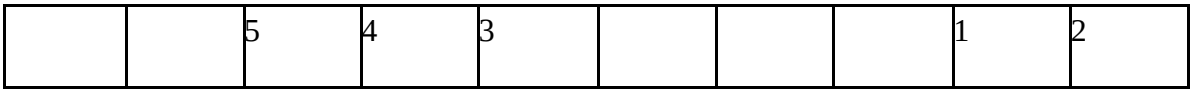
will produce the following output:



The value 543 is printed right justified in the first five columns.The specification width(5) does not retain the setting for printing the number 12.this can be improved as follows:

```
cout.width(5);
cout<<543;
cout.width(5);
cout<<12<<"\n";
```

This produces the following output:



The field width should be specified for each item.C++ never truncate the values and therefore,if the specified field width is smaller than the size of the value to be printed,C++ expands the field to fit the value. program 5.4 demonstrates how the function width() works.

Program

Specifying field size with width()

```
#include <iostream>
using namespace std;
int main()
{
 int item[4] ={ 10,8,12,15};
 int cost[4]={75,100,60,99};
cout.width(5);
cout<<"Items";
cout.width(8);
cout<<"Cost";
cout.width(15)
cout<<"Total Value"<<"\n"; int sum=0;
```

```
 for(int i=0;i<4 ;i++)
 {
  cout.width(5);
   cout<<items[i];
  cout.width(8);
  cout<<cost[i];
  int value = items[i] * cost[i]; cout.width(15); cout<<value<<"\n";
  sum= sum + value;
}
cout<<"\n Grand total = "; cout.width(2); cout<<sum<<"\n";
return 0;
}
```

ITEMS	COST	TOTAL VALUE
10	75	750
8	100	800
12	60	720
15	99	1485

Grand total =3755

The output of program 5.4 would be

Setting Precision: precision():-

By default ,the floating numbers are printed with six digits after the decimal points. However ,we can specify the number of digits to be displayed after the decimal point while printing the floating point numbers.

This can be done by using the precision () member function as follows:

cout.precision(d);

where d is the number of digits to the right of decimal point.for example the statements

```
cout.precision(3);
cout<<sqrt(2)<<"\n";
cout<<3.14159<<"\n";
cout<<2.50032<<"\n";
```

will produce the following output:

1.141 (truncated)

3.142 (rounded to nearest cent)

2.5 (no trailing zeros)

Unlike the function width(),precision() retains the setting in effect until it is reset.That is why we have declared only one statement for precision setting which is used by all the three outputs.We can set different valus to different precision as follows:

```
cout.precision(3);
cout<<sqrt(2)<<"\n";
```

cout.precision(5);

cout<<3.14159<<"\n";

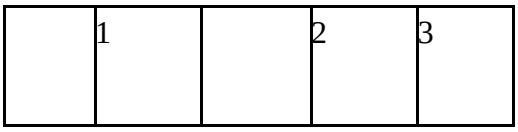
We can also combine the field specification with the precision setting.example:

cout.precision(2);

cout.width(5);

cout<<1.2345;

The first two statement instruct :"print two digits after the decimal point in a field of five character width".Thus the output will be:



Program 5.5 shows how the function width() and precision() are jointly used to control the output format.

Program 5.5

PRECISION SETTING WITH precision()

```
#include<iostream>
#include<cmath> using namespace std;
int main()
{
cout<<"precision set to 3 digits\n\n"; cout.precision(3);
cout.width(10);
cout<<"value";
cout.width(15);
cout<<"sqrt_of _value"<<"\n"; for (int n=1;n<=5;n++)

{
   cout.width(8);
  cout<<n;
  cout.width(13);
   cout<<sqrt(n)<<"\n";

}
```

VALUE	SQRT OF VALUE
1	1
2	1.41
3	1.73
4	2
5	2.24

Precision set to 5 digits

Sqrt(10)=3.1623

Sqrt(10)=3.162278 (Default setting)

```
cout<<"\n precision set to 5 digits\n\n"; cout.precision(5);
cout<<"sqrt(10) = " <<sqrt(10)<<"\n\n"; cout.precision(0);
cout<<"sqrt(10) = " <<sqrt(10)<<"(default setting)\n"; return 0;
}
```

The output is

Precision set to 3 digits

FILLING AND PADDING: fill()

The unused portion of field width are filled with white spaces, by default. The fill() function can be used to fill the unused positions by any desired character.It is used in the following form:

cout.fill(ch);

Where ch represents the character which is used for filling the unused positions.Example:

cout.fill('*');

cout.width(10);

cout<<5250<<"\n";

The output would be:

*	*	*	*	*	*	5	2	5	0

Financial institutions and banks use this kind of padding while printing cheques so that no one can change the amount easily.Like precision (),fill()

Stays in effect till we change it.As shown in following program

Program 5.6

```
#include<iostream>
using namespace std; int main()
{
cout.fill('<');
 cout.precision(3);
for(int n=1;n<=6;n++)
{
 cout.width(5);
cout<<n;
cout.width(10);
cout<<1.0/float(n)<<"\n";
if(n==3)
cout.fill('>');
}
cout<<"\nPadding changed \n\n"; cout.fill('#'); //fill()
reset cout.width(15); cout<<12.345678<<"\n";
return 0;
}
```

The output will be

```
  <<<1<<<<<<<<<1

  <<<<2<<<<<<<<0.5

  <<<<3<<<<<<0.333

  >>>>4>>>>>>0.25

  >>>>5>>>>>>>0.2

  PADDING CHANGED

  #########12.346
```

FORMATTING FLAGS,Bit Fields and setf()

The setf() a member function of the ios class, can provide answers left justified.The setf() function can be used as follows:

cout.setf(arg1.arg2)

The arg1 is one of the formatting flags defined in the class ios.The formatting flag specifies the format action required for the output.Another ios constant,arg2,known as bit field specifies the group to which the formatting flag belongs. for example:

cout.setf(ios::left,ios::adjustfield);

cout.setf(ios::scientific,ios::floatfield);

Note that the first argument should be one of the group member of second argument.

Consider the following segment of code:

```
cout.fill('*');
cout.setf(ios::left,ios::adjustfield);
cout.width(15);
cout<<"table1"<<"\n";
```

This will produce the following output:

T	A	B	L	E		1	*	*	*	*	*	*	*	*

The statements

```
cout.fill('*');
cout.precision(3);
cout.setf(ios::internal,ios::adjustfield);
cout.setf(ios::scientific,ios::floatfield);
cout.width(15);
cout<<-12.34567<<"\n";
```

Will produce the following output:

-	*	*	*	*	*	1	.	2	3	5	e	+	0	1

5. EXPLAIN IN DETAIL ABOUT MANIPULATORS?

MANIPULATORS

The header file iomanip provides a set of functions called manipulators which can be used to manipulate the output format. They provide the same features as that of the ios member functions and flags.For example,two or more manipulators can be used as a chain in one statement as follows

```
cout<<manip1<<manip2<<manip3<<item;
cout<<manip1<<item1<<manip2<<item2;
```

This kind of concatenation is useful when we want to display several columns of output. The most commonly used manipulators are shown below

In the table 5.4

Manipulator	**Meaning**	**Equivalent**
setw(int w)	Set the field width to w	width()

setprecision(int d)	Set the floating point precision to d	precision()
setfill(int c)	Set the fill character to c	fill()
setiosflags(long f)	Set the format flag f	setf()
resetiosflags(long f)	Clear the flag specified by f	unsetf()
Endl	Insert new line and flush Stream	"\n"

Examples of manipulators are given below:

cout<<setw(10)<<12345;

This statement prints the value 12345 right-justified in a field of 10 characters. The output can be made left-justified by modifying the statement

follows:

cout<<setw(10)<<setiosflags(ios::left)<<12345;

One statement can be used to format output for two or more values.For example, the statement

cout<<setw(5)<<setprecision(2)<<1.2345

<<setw(10)<<setprecision(4)<<sqrt(2)<<setw(15)<<setiosflags(ios::scientific)<<sqrt(3);

<<endl;

will print all the three values in one line with the field sizes of 5,10,15 respectively.

The following program illustrates the formatting of the output values using both manipulators and ios functions.

Program 5.7

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

{

  cout.setf(ios::showpoint);
 cout<<setw(5)<<"n"<<setw(15)<<"inverse of n"<<setw(15)<<"sum of terms";
double term,sum=0;
 for (int n=1;n<=10;n++)
 {
```

```
     term=1.0/float(n);
    sum=sum + term;
    cout<<setw(5)<<n<<setw(14)<<setprecision(4)<<setiosflags(ios::scientific)<<term
<<setw(13)<<resetioflags(ios::scientific) <<sum<<endl;
}
 return 0;
}
```

Designing our own manipulators:-

The general form for creating a manipulator without any argument is ostream & manipulator

```
  (ostream & output)
{
……………
…….(code)
……..
return output;
}
```

The following program illustrate the creation and use of user defined manipulators.

Program

```
#include<iostream>
#include <iomanip>
using namespace std;
ostream &currency (ostream & output)
{
     output<< "Rs"; return output;
}

ostream & form (ostream &output)
{
  output.set(ios::showpos);
  output.setf(ios::showpoint);
output.fill('*');
output.precision(2);
output<<setiosflags(ios::fixed)<<setw(10);
return output;
}
int main()
{
  cout<<currency<<form<<7864.5;
return 0;
}
```

the output of program is

Rs**+7864.50

In the program form represents a complex set of format functions and manipulators

MANIPULATORS WITH MULTIPLE PARAMETERS

In this type of manipulator, multiple arguments are passed on to the manipulator. The following program explains it.

Example program to create manipulator with two arguments.

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

struct w_p
 {
   int w;
   int p;
 };

IOMANIPdeclare (w_p);

static ostream& ff(ostream& os, w_p w_p)
 {
   os.width (w_p.w);
   os.precision(w_p.p);
   os.setf(ios::fixed);
   return os;
 }

OMANIP (w_p) column (int w, int p)
 {
   w_p w_p;
   w_p.w=w;
   w_p.p=p;
   return OMANIP (w_p) (ff,w_p);
}
int main ( )
{
   clrscr( );
   double n,sq,sqr;
   cout <<"number\t" <<"square\t" <<"\tsquare root\n";
   cout <<"==================================\n";
   n=1.0;
for (int j=1;j<16;j++)
{
   sq=n*n;
   sqr=sqrt(n);
   cout.fill('0');
   cout <<column(3,0)<<n <<"\t";
   cout <<column(7,1) <<sq <<"\t\t";
   cout <<column(7,6) <<sqr <<endl;
```

```
  n=n+1.0;
}
  return 0;
}
```

6. EXPLAIN IN DETAIL ABOUT THE CONTROL STRUCTURES

INTRODUCTION

- C++ is a superset of C. The control structures of C and C++ are same. A Program is nothing but a set of statements written in sequential order, one after the other. These statements are executed one after the other. Sometimes it may happen that the programmer requires to alter the flow of execution, or to perform the same operation for fixed iterations or whenever the condition does not satisfy.
- In such a situation the programmer uses the control structure. There are various control structures supported by C++. Programs which use such (one or three) control structures are said to be structured programs.

Decision making statements

Simple if statement

Syntax

```
if(expression)/*nosemi-colon*/
   Statement;
```

- The if statement contains an expression. The expression is evaluated. If the expression is true it returns 1 otherwise 0. The value 1 or any non-zero value is considered as true and 0 as false. In C++, the values (1) true and (0) false are known as bool type data. The bool type data occupies one byte in memory.

THE jump STATEMENT

- C/C++ has four statements that perform an unconditional control transfer. These are return( ), goto, break and continue. Of these, return( ) is used only in functions. The goto and return( ) may be used anywhere in the program but continue and break statements may be used only in conjunction with a loop statement. In 'switch case' 'break' is used most frequently.

THE switch case STATEMENT

- The switch statement is a multi-way branch statement and an alternative to if-else-if ladder in many situations. This statement requires only one argument, which is then checked with number of case options. The switch statement evaluates the expression and then looks for its value among the case constants.
- If the value is matched with a case constant then that case constant is executed until a break statement is found or end of switchblock is reached. If not then simply default (if present) is executed (if default isn't present then simply control flows out of the switch block). Thedefault is normally present at the bottom of switch case structure.
- But we can also define default statement anywhere in the switch structure. Thedefault block must not be empty. Every case statement terminates with : (colon). The break statement is

used to stop the execution of succeeding cases and pass the control to the switch end of block.

LOOPS IN C/C++

- C/C++ provides loop structures for performing some tasks which are repetitive in nature. The C/C++ language supports three types of loop control structures. The for loop comprises of three actions. The three actions are placed in the for statement itself. The three actions initialize counter, test condition, and re-evaluation parameters are included in one statement. The expressions are separated by semi-colons (;).
- This helps the programmer to visualize the parameters easily. The for statement is equivalent to the while and do-while statements. The only difference between for and while is that the latter checks the logical condition and then executes the body of the loop, whereas the for statement test is always performed at the beginning of the loop. The body of the loop may not be executed at all times if the condition fails at the beginning. The do-while loop executes the body of the loop at least once regardless of the logical condition.

THE for LOOP

- The for loop allows execution of a set of instructions until a condition is met. Condition may be predefined or open-ended. Although all programming languages provide for loops, still the power and flexibility provided by C/C++ is worth mentioning. The general syntax for the for loop is given below:

Syntax

for (initialization; condition; increment/decrement)
Statement block;

- Though many variations of for loop are allowed, the simplest form is shown above.
- The initialization is an assignment statement that is used to set the loop control variable(s). The condition is a relational expression that determines the number of the iterations desired or the condition to exit the loop.
- The increment or the re-evaluation parameter decides how to change the state of the variable(s) (quite often increase or decrease so as to approach the limit). These three sections must be separated by semi-colons. The body of the loop may consist of a block of statements (which have to be enclosed in braces) or a single statement (enclosure within braces is not compulsory but advised).

NESTED for LOOPS

- We can also nest for loops to gain more advantage in some situations. The nesting level is not restricted at all. In the body of a for loop, any number of sub for loop(s) may exist.

Example program to demonstrate nested for loops.

```
# include <iostream.h>
# include <constream.h>>
void main( )
{
int a,b,c;
clrscr( );
for (a=1;a<=2;a++) /* outer loop */
```

```
{
for (b=1;b<=2;b++) /* middle loop */
{
  for (c=1;c<=2;c++) /* inner loop */
  cout <<"\n a="<<a <<" + b="<<b <<"+ c="<<c <<" :"<<a+b+c;
  cout <<"\n Inner  Loop Over.";
}


cout<<"\n Middle Loop Over.";
}
cout<<"\n Outer  Loop Over.";
}
```

Explanation: The above program is executed in the sequence shown below. The total number of iterations are equal to 2*2*2=8. The final output provides 8 results.

THE while LOOP

Another kind of loop structure in C/C++ is the while loop. It's format is given below.

Syntax:

```
while (test condition)
{
  body of the loop
}
```

- The test condition may be any expression. The loop statements will be executed till the condition is true i.e., the test condition is evaluated and if the condition is true, then the body of the loop is executed. When the condition becomes false the execution will be out of the loop.
- The block of the loop may contain single statement or a number of statements. The same block can be repeated. The braces are needed only if the body of the loop contains more than one statement. However, it is a good practice to use braces even if the body of the loop contains only one statement.

Example program using while loop to find the sum of the digits of a number.

```
#include<iostream.h>
#include<conio.h>
void main()
{
 clrscr();
 int n,rem,sum=0;
 cout<<“Enter the numbers “;
 cin>>n;
 while(n!=0)
  {
     rem=n%10;
     sum+=rem;
     n/=10;
```

```
  }
 cout<<"The sum of the digits of the number is "<<sum;
 getch();
}
```

THE do-while LOOP

The format of do-while loop in C/C++ is given below.

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

- while (condition); The difference between the while and do-while loop is the place where the condition is to be tested. In the while loop, the condition is tested following the while statement and then the body gets executed. Where as in do-while, the condition is checked at the end of the loop. The do-while loop will execute at least once even if the condition is false initially. The do-while loop executes until the condition becomes false.

Example program using do..while to print first n natural numbers and their sum

```
#include<iostream.h>
#include<conio.h>
void main()
{
 clrscr();
 int n,i=1,sum=0;
 cout<<"Enter the value of n: ";
 cin>>n;
 cout<<"\nFirst "<<n<<" natural numbers are:\n\n";
 do
 {
   cout<<i<<' ';
   sum+=i;
   i++;
 }while(i<=n);
 cout<<"\n\nSum = "<<sum;
 getch();
}
```

11. WRITE IN DETAIL ABOUT FUNCTIONS IN C++ AND ITS TYPES OF PASSING ARGUMENTS WITH EXAMPLE PROGRAMS

FUNCTIONS

- When the concept of function or sub-program was not introduced, the programs were large in size and code got repeated. It was very difficult to debug and update these large programs because many bugs were encountered
- When the functions and sub-programs are introduced, the programs are divided into a number of segments and code duplication is avoided. Bugs in small programs can be searched easily and this step leads to the development of complex programs with functions
- One of the features of C/C++ language is that a large sized program can be divided into smaller ones. The smaller programs can be written in the form of functions.
- The process of dividing a large program into tiny and handy sub-programs and manipulating them independently is known as ***modular programming.*** Dividing a large program into smaller functions provides advantages to the programmer

ADVANTAGES OF FUNCTIONS

- Support for modular programming
- Reduction in program size
- Code duplication is avoided
- Code reusability is provided
- Functions can be called repetitively
- A set of functions can be used to form Libraries

TYPES OF FUNCTIONS

C++ supports two types of function, They are

- Library Functions
- User Defined Functions

LIBRARY FUNCTIONS

- The library functions can be used in any program by including respective header files
- For example, a mathematical functions uses math.h header file

USER DEFINED FUNCTIONS

- The programmer can also define and use his/her own functions for performing some specific tasks. Such functions are called as user defined functions

PARTS OF FUNCTIONS

A function has the following parts:

- Function prototype declaration
- Definition of a function
- Function call
- Actual and formal arguments
- The return statement

FUNCTION PROTOTYPE

- A prototype statement helps the compiler to check the return and argument types of the function.
- A function prototype declaration consists of the function return type, name, and arguments list.
- It tells the compiler
    - the name of the function,
    - the type of value returned, and
    - the type and number of arguments.
- When the programmer defines the function, the definition of function must be like its prototype declaration.
- If the programmer makes a mistake, the computer flags an error message
- The function prototype declaration statement is always terminate with semi-colon.

Examples of function prototypes:

- void show();
- float sum(float, int);
- float sum(float x, int y);

DEFINITION OF A FUNCTION

- The first line is called function declarator and is followed by function body
- The block of statements followed by function declarator is called as function definition
- The declarator and function declaration should match each other

- The function body is enclosed with curly braces
- The function can be defined anywhere.
- If the function is defined before its caller, then its prototype declaration is optional

EXAMPLE:

```
int sum(int a, int b)        Function Declarator
{
        c=a+b;
        return c;            Function Body
}
```

FUNCTION CALL

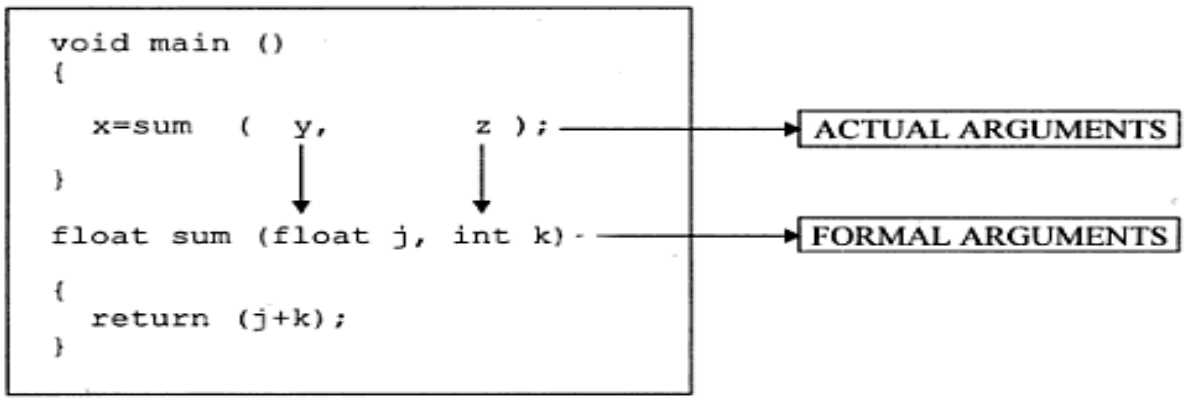
- A function is a latent body
- It gets activated only when a calling function is invoked
- A function must be called by its name, followed by argument list enclosed in parenthesis and terminated by semi-colon

EXAMPLE

- sum(a,b);
- int a = sum(a,b);

ACTUAL AND FORMAL ARGUMENTS

- The arguments declared in caller function and given in the function call are called as actual arguments
- The arguments declared in the function declarator are known as formal arguments



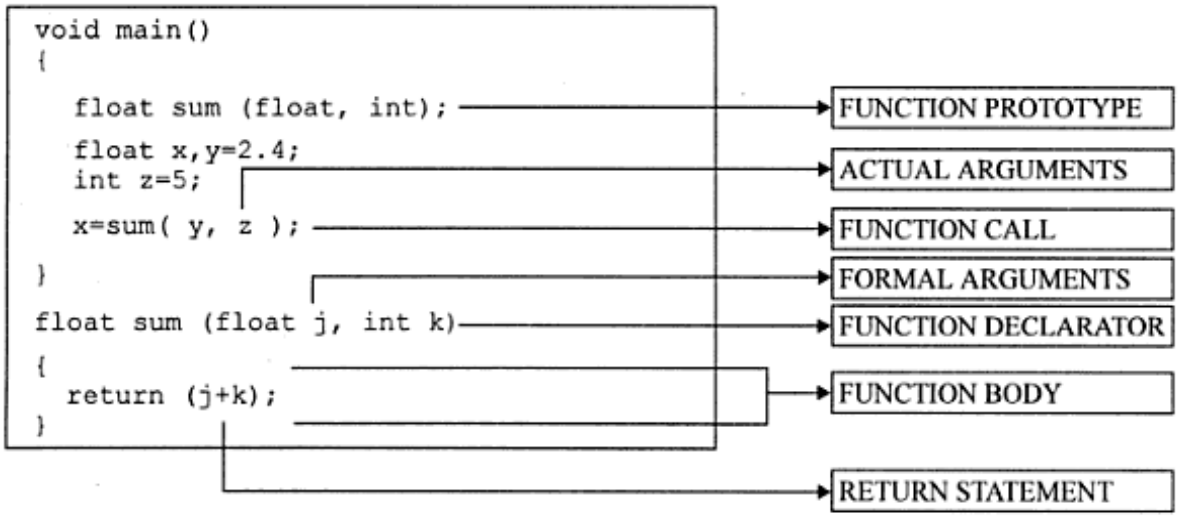
THE Return STATEMENT

- The return statement is used to return value to the caller function.
- The return statement returns only one value at a time.
- When a return statement is encountered, compiler transfers the control of the program to caller function
- The Parenthesis is optional

SYNTAX:

return (variable name); or return variable name;

PARTS OF A FUNCTION



TYPES OF FUNCTIONS

Four types of function

- No passing arguments, No return
- Passing arguments, No return
- No passing arguments, Return statement
- Passing arguments, Return statement

PASSING ARGUMENTS

- The main objective of passing argument to function is message passing.
- The message passing is also known as communication between two functions i.e., between caller and callee functions.
- There are three methods

- Call by value (pass by value)
- Call by address (pass by address)
- Call by reference(pass by reference)

CALL BY VALUE (Pass by value)

- In this type, value of actual argument is passed to the formal argument and operation is done on the formal arguments
- Any change in the formal arguments does not affect the actual argument because formal arguments are photocopy of actual arguments
- Changes made in the formal arguments are local to the block of called function
- Once control returns back to the calling function the changes made vanish

CALL BY ADDRESS (Pass by address)

- In this type, instead of passing values, addresses are passed
- Function operates on addresses rather than values
- Here the formal arguments are pointers to the actual arguments
- Hence changes made in the arguments are permanent

CALL BY REFERENCE (Pass by reference)

- C passes arguments by value and address.
- In C++ it is possible to pass arguments by value, address and reference.
- C++ reference types declared with & operator.
- Its nearly identical to the pointer type but not the same
- Reference type declare aliases for object variables and allow the programmer to pass arguments by reference to functions
- The reference decelerator (&) can be used to declare references outside functions

```
int k=0;
        int &kk = k;
        kk = 2;
```

- Creates kk as an alias for k. Any operation on kk will give the same result as operations on k.

CALL BY REFERENCE: EXAMPLE

```
kk=5;     // assigns 5 to k and

&kk               // returns the address of k
```

```
        void funcA(int i);
        void funcB(int &kk);                    //kk is type “reference to int”
        int s=4;
        funcA(s);                       // s passed by value
        funcB(s);                       // s passed by reference
```

EXAMPLE PROGRAM:

```
#include<iostream.h>
#include<conio.h>
void main()
{
        clrscr();
        void funA(int s);
        void funB(int &);
        void funC(int *);

        int s=4;            //initial value

        funA(s);            //s passed by value
        cout<<"Value of s:"<<s<<"Address of s:"<<unsigned(&s);

        funB(s);            //s passed by reference
        cout<<"\nValue of s:"<<s<<"Address of s:"<<unsigned(&s);

        funC(&s);                       //s passed by address
        cout<<"\nValue of s:"<<s<<"Address of s:"<<unsigned(&s);

        getch();
}
void funA(int i)
{
        i++;
}
void funB(int &k)
{
        k++;
}
void funC(int *j)
{
        ++*j;
}
```

12. EXPLAIN IN DETAIL ABOUT CLASSES AND OBJECTS WITH EXAMPLE PROGRAM

CLASSES IN C++

DEFINITION:

The class is a blue print/ template/ model of an object. Class is grouping of objects having identical properties, common behaviour and shared relationship.

- The class is used to pack data and function together.
- The class has a mechanism to prevent direct access to its members, which is the central idea of object-oriented programming.
- The class declaration is also known as formation of new abstract data type. The abstract data type can be used as basic data type such as int, float etc.
- The class consists of member data variables that hold data and member functions that operate on the member data variables of the same class

SYNTAX AND EXAMPLE OF CLASS DECLARATION:

Syntax of class declaration	Example of class

class <name of class>
{
private:
 	
declaration of variables;
prototype declaration of function;
 	
public:
 	
declaration of variables;
prototype declaration of function;
};
	class item // class declaration
	{
	private:
	int codeno;
	float prize;
	int qty;
	void values();
	public:
	void show();
	};

OBJECT:

Objects are primary run-time entities in an object-oriented programming. They may stand for a thing that has specific application for example, a spot, a person, any data item related to program, including user-defined data types. An object is an instance of a class. It has both state and behaviour.

DECLARING OBJECTS

- A class declaration only builds the structure of object. The member variables and functions are combined in the class. The declaration of objects is same as declaration of variables of basic data types.

- Defining objects of class data type is known as ***class instantiation.*** When objects are created only during that moment, memory is allocated to them.

EXAMPLES:

- int x,y,z; // Declaration of integer variables
- char a,b,c; // Declaration of character variables
- item a,b, *c; // Declaration of object or class type variables

ACCESSING CLASS MEMBERS

- The object can access the public member variables and functions of a class by using operator dot (.) and arrow (->).

SYNTAX:

[object name] [operator] [member name];

EXAMPLES:

a . show();
c->show();

ACCESS SPECIFIERS

C++ supports three types of access specifiers, They are

- Private
- Public
- Protected

PRIVATE:

- The member variables and functions declared followed by the keyword private can be accessed only inside the class

The declaration can be done as below:

```
class item      // class declaration
{
        private:        // private section begins
                int codeno;
                float prize;
                int qty;
};
```

PUBLIC:

- The member variables and functions declared followed by the keyword public can be accessed outside the class

PROTECTED:

- The member variables and functions declared followed by the keyword protected can be accessed inside its own class and its derived class

DEFINING MEMBER FUNCTIONS

- The member function must be declared inside the class.

They can be defined in

a) ***private or public section***
b) ***inside or outside the class.***

- The member functions defined inside the class are treated as inline function.

- If the member function is small then it should be defined inside the class, otherwise it should be defined outside the class

MEMBER FUNCTION INSIDE THE CLASS

- Member function inside the class can be declared in public or private section

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

class item                              //class declaration
{
        private:                        //private section begins

                int codeno;
                float prize;
                int qty;

        public:                         //public section begins

                void show()             //Member Function defined inside
                {
                        codeno=123;  //Access to private members
                        prize=123.45;
                        qty=150;

                        cout<<"\n CodeNo="<<codeno;
                        cout<<"\n Prize="<<prize;
                        cout<<"\n Quantity="<<qty;
                }
};              //END OF THE CLASS

void main()
{
        clrscr();
```

```
        item one;       //object declaration

        one.show();   //Call to member function

        getch();

    }
```

PRIVATE MEMBER FUNCTION

- To execute private member function, it must be invoked by public member function of the same class
- A member function of a class can invoke any other member function of its own class

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

class item                          //class declaration
{
    private:                        //private section begins

        int codeno;
        float prize;
        int qty;

        void values()  //private member function
        {
            codeno=123;
            prize=123.45;
            qty=150;
        }


    public:

        void show()             //public member function
        {
            values();       //call to private member function

            cout<<"\n CodeNo="<<codeno;
            cout<<"\n Prize="<<prize;
            cout<<"\n Quantity="<<qty;
        }
};
```

```
void main()
{
        clrscr();

        item one;                       //object declaration

        // one.values();        //not accessible

        one.show();                     //call to public member function

        getch();

}
```

MEMBER FUNCTION OUTSIDE THE CLASS

- In previous examples, the member functions are defined inside the class and the function prototype is also not declared
- To define a function outside the class, the following care must be taken
    - The prototype of function must be declared inside the class
    - The function name must be preceded by class name and its return type separated by scope resolution operator

SYNTAX:

return type class name : : function name (arguments)

EXAMPLE:

void item::show()

PROGRAM:

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

class item                      //class declaration
{
    private:                    //private section begins

            int codeno;
            float prize;
            int qty;

    public:

            void show();  //prototype declaration

};
```

```
void item::show()                    //Definition outside the class
{
    codeno=123;
    prize=123.45;
    qty=150;

    cout<<"\n CodeNo="<<codeno;
    cout<<"\n Prize="<<prize;
    cout<<"\n Quantity="<<qty;
}

void main()
{
    clrscr();

    item one;                //object declaration

    one.show();              //call to public member function

    getch();
}
```

CHARACTERISTICS OF MEMBER FUNCTIONS

- The difference between member and normal function is that the former function can be invoked freely where as the latter function only by using an object of the same class.
- The same function can be used in any number of classes. This is possible because the scope of the function is limited to their classes and cannot overlap one another.
- The private data or private function can be accessed by public member function. Other functions have no access permission.
- The member function can invoke one another without using any object or dot operator.

EXAMPLE PROGRAM:

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

class student
{
        private:
                int regno;
                char* name;
                int mark;
```

```
        public:

                void get();
                void calculate();

};

                void student::get()
                {
                        cout<<"Enter the Register no.";
                        cin>>regno;
                        cout<<"Enter the Name:";
                        cin>>name;
                        cout<<"Enter the mark";
                        cin>>mark;
                }

                void student::calculate()
                {

                        if(mark<=40)
                        {
                                cout<<"FAIL";
                        }
                        else
                        {
                                cout<<"PASS";
                        }
                }

void main()
{
        clrscr();

        student s1,s2;

        s1.get();
        s1.calculate();

        s2.get();
        s2.calculate();

        getch();
}
```

Note: you may use your own program or programs in ur class note for member functions definition.

12. WRITE BRIEFLY ABOUT INLINE FUNCTIONS

- The inline mechanism reduces overhead relating to accessing the member function. It provides better efficiency and allows quick execution of functions. An inline member function is similar to macros. Call to inline function in the program, puts the function code in the caller program. This is known as inline expansion

Rules of inline functions

- These are the functions designed to speed up program execution. An inline function is expanded (i.e. the function code is replaced when a call to the inline function is made) in the line where it is invoked.
- You are familiar with the fact that in case of normal functions, the compiler have to jump to another location for the execution of the function and then the control is returned back to the instruction immediately after the function call statement. So execution time taken is more in case of normal functions. There is a memory penalty in the case of an inline function.

The system of inline function is as follows :

```
inline function_name()
{
                    statement 1;
                    statement 2;
}
```

Example

```
inline float square(float k)
{
        return (k*k);
 }
```

- An inline function definition must be defined before being invoked as shown in the above example. Here min ( ) being inline will not be called during execution, but its code would be inserted into main ( ) as shown and then it would be compiled.
- If the size of the inline function is large then heavy memory pentaly makes it not so useful and in that case normal function use is more useful

.

The inlining does not work for the following situations :

1. For functions returning values and having a loop or a switch or a goto statement.

2. For functions that do not return value and having a return statement.

3. For functions having static variable(s).

4. If the inline functions are recursive (i.e. a function defined in terms of itself).

The benefits of inline functions are as follows :

1. Better than a macro.
2. Function call overheads are eliminated.
3. Program becomes more readable.
4. Program executes more efficiently.

EXAMPLE PROGRAM:

5.13 Write a program to define function `cube()` as inline for calculating cube.

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

void main()
{
   clrscr();
   int cube(int);
   int j,k,v=5;
   j = cube(3) ;
   k = cube(v);

   cout <<"\n Cube of j="<<j;
   cout <<"\n Cube of k="<<k;
}

inline int cube(int h)
{
   return (h*h*h);
}
```

OUTPUT

```
Cube of j=27
Cube of k=125
```

13. WRITE BRIEFLY ABOUT STATIC OBJECT ARRAY OF OBJECTS WITH SAMPLE PROGRAM

STATIC OBJECT

In C, it is common to declare variable static that gets initialized to zero. The object is a composition of one or more member variables. There is a mechanism called constructor to initialize member variables of the object to desired values. The constructors are explained in next chapter. The keyword staticcan be used to initialize all class data member variables to zero. Declaring object itself as static can do this. Thus, all its associated members get initialized to zero. The following program illustrates the working of static object.

Example program to declare static object. Display its contents.

include <iostream.h>

```
        # include <constream.h>
        class bita
        { private:
        int c;
        int k;
        public :
         void  plus( )
        {
         c+=2;
        k+=2;
        }

  void show( )
  {
    cout <<" c= "<<c<<"\n";

    cout <<" k= "<<k;
  }

};
void main( )
{
    clrscr( );
    static bita A;
    A.plus( );
    A.show( );
}
```

ARRAY OF OBJECTS

- Arrays are collection of similar data types. Arrays can be of any data type including user-defined data type, created by using struct, class and typedef declarations. We can also create an array of objects.Consider the following example:

```
class player
{
private:
   char name [20];
   int age;


public:


   void input (void);
   void display (void);
};
```

- In the example given above player is a user-defined data type and can be used to declare an array of object of type player. Each object of an array has its own set of data variables.

```
player  cricket[5];
player  football[5];
player  hockey[5];
```

- As shown above, arrays of objects of type player are created. The array cricket[5] contains name, age and information for five objects. The next two declarations can maintain the same information for other players in arrays hockey[5] and football[5]. These arrays can be initialized or accessed like an ordinary array.

OBJECTS AS FUNCTION ARGUMENTS

- Similar to variables, objects can be passed on to functions. There are three methods to pass an argument to a function as given below:
-

(a) **Pass-by-value**—In this type a copy of an object (actual object) is sent to function and assigned to object of callee function (Formal object). Both actual and formal copies of objects are stored at different memory locations. Hence, changes made in formal objects are not reflected to actual objects.
(b) **Pass-by-reference** — Address of object is implicitly sent to function.
(c) **Pass-by-address** — Address of the object is explicitly sent to function.

- In pass by reference and address methods, an address of actual object is passed to the function. The formal argument is reference pointer to the actual object. Hence, changes made in the object are reflected to actual object. These two methods are useful because an address is passed to the function and duplicating of object is prevented.

7. WHAT IS MEANT BY MEMBER FUNCTION? STATE THE TYPES OF DEFINING MEMBER FUNCTIONS IN CLASS WITH EXAMPLE

MEMBER FUNCTION

The functions that operate on the member data variables of the same class are called as member function

DEFINING MEMBER FUNCTIONS

- The member function must be declared inside the class.

They can be defined in

c) ***private or public section***
d) ***inside or outside the class.***

- The member functions defined inside the class are treated as inline function.

- If the member function is small then it should be defined inside the class, otherwise it should be defined outside the class

MEMBER FUNCTION INSIDE THE CLASS

- Member function inside the class can be declared in public or private section

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

class item                              //class declaration
{
        private:                        //private section begins

                int codeno;
                float prize;
                int qty;

        public:                         //public section begins

                void show()             //Member Function defined inside
                {
                        codeno=123;  //Access to private members
                        prize=123.45;
                        qty=150;

                        cout<<"\n CodeNo="<<codeno;
                        cout<<"\n Prize="<<prize;
                        cout<<"\n Quantity="<<qty;
                }
};              //END OF THE CLASS

void main()
{
        clrscr();

        item one;       //object declaration

        one.show();   //Call to member function

        getch();

}
```

PRIVATE MEMBER FUNCTION

- To execute private member function, it must be invoked by public member function of the same class
- A member function of a class can invoke any other member function of its own class

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

class item                              //class declaration
{
        private:                        //private section begins

                int codeno;
                float prize;
                int qty;

                void values()  //private member function
                {
                        codeno=123;
                        prize=123.45;
                        qty=150;
                }


        public:

                void show()             //public member function
                {
                        values();       //call to private member function

                        cout<<"\n CodeNo="<<codeno;
                        cout<<"\n Prize="<<prize;
                        cout<<"\n Quantity="<<qty;
                }
};
void main()
{
        clrscr();

        item one;                       //object declaration

        // one.values();        //not accessible

        one.show();                     //call to public member function

        getch();

}
```

MEMBER FUNCTION OUTSIDE THE CLASS

- In previous examples, the member functions are defined inside the class and the function prototype is also not declared
- To define a function outside the class, the following care must be taken
    - The prototype of function must be declared inside the class
    - The function name must be preceded by class name and its return type separated by scope resolution operator

SYNTAX:

return type class name : : function name (arguments)

EXAMPLE:

void item::show()

PROGRAM:

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


class item                      //class declaration
{
    private:                    //private section begins

            int codeno;
            float prize;
            int qty;

    public:

            void show();  //prototype declaration

};

void item::show()               //Definition outside the class
{
    codeno=123;
    prize=123.45;
    qty=150;

    cout<<"\n CodeNo="<<codeno;
    cout<<"\n Prize="<<prize;
    cout<<"\n Quantity="<<qty;
}
```

```
void main()
{
    clrscr();

    item one;              //object declaration

    one.show();            //call to public member function

    getch();
}
```

CHARACTERISTICS OF MEMBER FUNCTIONS

- The difference between member and normal function is that the former function can be invoked freely where as the latter function only by using an object of the same class.
- The same function can be used in any number of classes. This is possible because the scope of the function is limited to their classes and cannot overlap one another.
- The private data or private function can be accessed by public member function. Other functions have no access permission.
- The member function can invoke one another without using any object or dot operator.

EXAMPLE PROGRAM:

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

class student
{
        private:
                int regno;
                char* name;
                int mark;


        public:

                void get();
                void calculate();

};

                void student::get()
                {
                        cout<<"Enter the Register no.";
                        cin>>regno;
                        cout<<"Enter the Name:";
```

```
                cin>>name;
                cout<<"Enter the mark";
                cin>>mark;
        }

        void student::calculate()
        {

                if(mark<=40)
                {
                        cout<<"FAIL";
                }
                else
                {
                        cout<<"PASS";
                }
        }

void main()
{
        clrscr();

        student s1,s2;

        s1.get();
        s1.calculate();

        s2.get();
        s2.calculate();

        getch();
}
```

8. EXPLAIN IN DETAIL ABOUT FUNCTION OVERLOADING WITH EXAMPLE

FUNCTION OVERLOADING

It is possible in C++, to use the same function name for a number of times for different intentions.

Defining multiple functions with same name is known as Function Overloading

To see why function overloading is important, first consider three functions defined by the C subset: **abs(), labs(), and fabs().**

abs() returns the absolute value of an integer,

labs() returns the absolute value of a long, and

fabs() returns the absolute value of a double.

Although these functions perform almost identical actions, in C three slightly different names must be used to represent these essentially similar tasks.

Example: **Function Overloading**

Function Overloading – DEFINITION
It is the process of using the same name for two or more functions.

The secret to overloading is that each redefinition of the function must use either-

- different types of parameters
- different number of parameters.

The key to function overloading is a function's argument list.

A function's argument list is known as its signature.

Example :

If two function have same number and types of arguments in the same order, they are said to have the same signature.

There is no importance to their variable names.

- void print(int x, float y);
- void print(int p, float q); are said to have same signature.

Calling Overloaded Function

Overloaded functions are called just like ordinary functions.

The signature determines which function among the overloaded functions should be executed.

print(3);
print('B');

STEPS FOR FINDING BEST MATCH

The actual arguments are compared with formal arguments of the overloaded functions and the best match is found through a number of steps.

In order to find the best match, the compiler follows the steps given below:

STEPS - 1 : Search for an exact match

If the type of the actual and formal argument is exactly the same, the compiler invokes that function.

```
void print(int x);     // function #1
void print(float p); // function #2
void print(char c); // function #3
------------
-------------
print(5.3);  //invokes the function #2
```

STEPS - 2 : A match through promotion

If no exact match is found, an attempt is made to achieve a match through promotion.

```
void print(int x);     // function #1
void print(float p); // function #2
void print(double f); // function #3
------------
-------------
print('c');  //matches the function #1 through promotion
```

STEPS - 3 : A match through standard
C++ conversion rules

If the first and second steps fail, then an attempt is made to find a best match through conversion rule.

```
void print(char x);     // function #1
void print(float p); // function #2
------------
-------------
print(342);  //matches the function #2      int 342 is converted to float
```

STEPS - 4 : A match through user defined conversion

If all the above three steps fail to find a match, then an attempt is made to find a match by applying user defined conversion rules.

STEPS - 4 : A match through user defined conversion

If all the above three steps fail to find a match, then an attempt is made to find a match by applying user defined conversion rules.

STEPS - 4 : A match through user defined conversion

If all the above three steps fail to find a match, then an attempt is made to find a match by applying user defined conversion rules.

PRECAUTIONS WITH FUNCTION OVERLOADING

Function overloading is a powerful feature of C++.But, this facility should not be overused.

Otherwise it becomes an additional overhead in terms of readability and maintenance.

Following precautions must be taken:

(1) Only those functions that basically do the same task, on different sets of data, should be overloaded. The overloading of function with identical name but for different purposes should be avoided.

(2) In function overloading, more than one function has to be actually defined and each of these occupy memory.

(3) Instead of function overloading, using default arguments may make more sense and fewer overheads.

(4) Declare function prototypes before main( ) and pass variables instead of passing constant directly. This will avoid ambiguity that frequently occurs while overloading functions

(2) In function overloading, more than one function has to be actually defined and each of these occupy memory.

(3) Instead of function overloading, using default arguments may make more sense and fewer overheads.

(4) Declare function prototypes before main( ) and pass variables instead of passing constant directly. This will avoid ambiguity that frequently occurs while overloading functions

(2) In function overloading, more than one function has to be actually defined and each of these occupy memory.

(3) Instead of function overloading, using default arguments may make more sense and fewer overheads.

(4) Declare function prototypes before main( ) and pass variables instead of passing constant directly. This will avoid ambiguity that frequently occurs while overloading functions

EXAMPLE PROGRAM:

5.15 Write a program to find the area of rectangle, triangle and sphere. Use function overloading.

```
#include<iostream.h>
#include<constream.h>
#define pi 3.142857142857142857
int calcarea(int length,int breadth);
float calcarea(double base,double height);
float calcarea(double radius);

void main()
   {
      int area1;
      float area2;
      double area3;
      area1=calcarea(10,20);
      area2=calcarea(4.5,2.1);
      area3=calcarea(3.12145);
      clrscr();
      cout<<"Area of rectangle is  : "<<area1<<endl;
      cout<<"Area of traingle is   : "<<area2<<endl;
      cout<<"Area of sphere is     : "<<area3<<endl;
      getch();
   }
```

```
   int calcarea(int length,int breadth)
{
   return (length*breadth);
}
   float calcarea(double base,double height)
{

   return ((0.5)*base*height);
}
   float calcarea(double radius)
{
   return ((4/3)*pi*radius*radius*radius);
}
```

```
OUTPUT:
Area of rectangle is  : 200
Area of traingle is   : 4.725
Area of sphere is     : 95.58589
```

9. EXPLAIN BREIFLY ABOUT FRIEND FUNCTIONS IN C++
FRIEND FUNCTION

- Class operations are typically implemented as member functions
- Some operations are better implemented as ordinary (nonmember) functions

- **Friend functions** are not members of a class, but can access private member variables of the class
- A friend function is declared using the keyword **friend** in the class definition
    - A friend function is not a member function
    - A friend function is an ordinary function
    - A friend function has extraordinary access to data members of the class

USING FRIEND FUNCTION

- A friend function is declared as a friend in the class definition
- A friend function is defined as a nonmember function without using the "**::**" operator
- A friend function is called without using the '.' operator

FRIEND DECLARATION SYNTAX

The syntax for declaring friend function is

```
class class_name
{
  public:
        friend Declaration_for_Friend_Function_1
             friend Declaration_for_Friend_Function_2
           ...
        Member_Function_Declarations
  private:
        Private_Member_Declarations
 };
```

EXAMPLE PROGRAM

```
#include<iostream.h>

#include<conio.h>

class  base

{

   int val1,val2;

  public:

   void get()

   {
```

```
      cout<<"Enter two values:";
      cin>>val1>>val2;
    }
    friend float mean(base ob);
};
float mean(base ob)
{
  return float(ob.val1+ob.val2)/2;
}
void main()
{
    clrscr();
    base obj;
    obj.get();
    cout<<"\n Mean value is : "<<mean(obj);
    getch();
}
```

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

SUBJECT NAME: OBJECT ORIENTED PROGRAMMING AND DESIGN SUBJECT CODE: CST33

UNIT - II

Constructors and Destructors: Purpose of Constructors and Destructors – overloading constructors – constructors with default arguments – copy constructors – calling constructors and destructors – dynamic initialization using constructors – recursive constructor.
Overloading Functions: Overloading unary operators – constraint on increment and decrement operators – overloading binary operators – overloading with friend functions – type conversion – one argument constructor and operator function – overloading stream operators.
Inheritance: Introduction – Types of Inheritance – Virtual base classes – constructors and destructors and inheritance – abstract classes – qualifier classes and inheritance – common constructor – pointers and inheritance – overloading member function

2 MARKS

1. **What is constructor?**
   - A **constructor** (having the same name as that of the class) is a special member function which is automatically used to initialize the objects of the class type with legal initial values

2. **What is the purpose of destructor?**
   - Destructors are the functions that are complimentary to constructors. These are used to de-initialize objects when they are destroyed. A destructor is called when an object of the class goes out of scope, or when the memory space used by it is de allocated with the help of **delete** operator.

3. **What is constructor overloading?**
   - Like functions, it is also possible to overload constructors. A class can contain more than one constructor. This is known as constructor overloading. All constructors are defined with the same name as the class. All the constructors contain different number of arguments. Depending upon number of arguments, the compiler executes appropriate constructor.

4. **Define constructors with default arguments**
   - Like functions, it is also possible to declare constructors with default arguments. Consider the following example.

     power (int 9, int 3);

   - In the above example, the default value for the first argument is nine and three for second.

power p1 (3);

- In this statement, object p1 is created and nine raise to 3 expression n is calculated. Here, one argument is absent hence default value 9 is taken, and its third power is calculated

5. Define copy constructor

- It is of the form classname (classname &) and used for the initialization of an object form another object of same type

6. State the special characteristics of constructors

- These are called automatically when the objects are created.
- All objects of the class having a constructor are initialized before some use.
- These should be declared in the public section for availability to all the functions.
- Return type (not even **void**) cannot be specified for constructors.
- These cannot be inherited, but a **derived** class can call the base class constructor.
- These cannot be static.
- Default and copy constructors are generated by the compiler wherever required. Generated constructors are public.
- These can have default arguments as other C++ functions.
- A constructor can call member functions of its class.
- An object of a class with a constructor cannot be used as a member of a **union.**
- A constructor can call member functions of its class.
- We can use a constructor to create new objects of its class type by using the syntax.
- Name_of_the_class (expresson_list)
    - For example,
        - Employee obj3 = obj2; // see program 10.5
    - Or even

▪ Employee obj3 = employee (1002, 35000); //explicit call

- The make **implicit** calls to the memory allocation and deallocation operators **new** and **delete.**

7. State the special characteristics of destructors

Some of the characteristics associated with destructors are :

(i) These are called automatically when the objects are destroyed.

(ii) Destructor functions follow the usual access rules as other member functions.

(iii) These **de-initialize** each object before the object goes out of scope.

(iv) No argument and return type (even void) permitted with destructors.

(v) These cannot be inherited.

(vi) **Static** destructors are not allowed.

(vii) Address of a destructor cannot be taken.

(viii) A destructor can call member functions of its class.

(ix) An object of a class having a destructor cannot be a member of a union.

8. What is dynamic initialization of constructor?

- After declaration of the class data member variables, they can be initialized at the time of program execution using pointers.
- Such initialization of data is called as dynamic initialization. The benefit of dynamic initialization is that it allows different initialization modes using overloaded constructors. Pointer variables are used as argument for constructors.

9. What is recursive constructor?

- Like normal and member functions, constructors also support recursion. Constructor which acts like recursive function is called recursive constructor.

10. What is operator overloading?

- C++ has the ability to provide the operators with a special meaning for a data type. This mechanism of giving such special meanings to an operator is known as Operator overloading. It provides a flexible option for the creation of new definitions for C++ operators.

11. List out the operators that cannot be overloaded.

- Class member access operator (. , .*)
- Scope resolution operator (::)

- Size operator (sizeof)
- Conditional operator (?:)

12. Write the steps involved in operator overloading?

- First , create a class that defines the data type that is to be used in the overloading operation
- Declare the operator function in the public part of a class. It may be either a member function or a friend function.
- Define the operator function to implement the required operations.

13. Write at least four rules for Operator overloading.

- Only the existing operators can be overloaded.
- The overloaded operator must have at least one operand that is of user defined data type.
- The basic meaning of the operator should not be changed.
- Overloaded operators follow the syntax rules of the original operators.
- They cannot be overridden.

14. How will you overload Unary & Binary operator using member functions?

- When unary operators are overloaded using member functions it takes no explicit arguments and return no explicit values. When binary operators are overloaded using member functions, it takes one explicit argument. Also the left hand side operand must be an object of the relevant class.

15. How will you overload Unary and Binary operator using Friend functions?

- When unary operators are overloaded using friend function, it takes one reference argument (object of the relevant class) When binary operators are overloaded using friend function, it takes two explicit arguments.

16. How an overloaded operator can be invoked using member functions?

In case of Unary operators, overloaded operator can be invoked as

op object_name or object_name op

In case of binary operators, it would be invoked as

Object. Operator op(y)

Where op is the overloaded operator and y is the argument.

17. How an overloaded operator can be invoked using Friend functions?

In case of unary operators, overloaded operator can be invoked as

Operator op (x);

In case of binary operators, overloaded operator can be invoked as

Operator op (x , y)

18. List out the operators that cannot be overloaded using Friend function.

- Assignment operator =
- Function call operator ( )
- Subscripting operator [ ]

- Class member access operator

19. What is data conversion?

When we use the assignment operator we assign a value on the right hand side to a variable on the left side & if it is of a different data type then we have to perform data conversion or type conversion.

20. What are the methods of data conversion?

There are two methods for data conversion:

(i) Implicit data conversion

(ii) Explicit data conversion

21. What is meant by explicit data conversion?

Sometimes we want to force the compiler to convert one data type to another. To do this we use the cast operator. For example, to convert float into int we should say

```
int var= int (float var);
```

22. What is type casting?

Casting provides explicit conversion between type for example if we want to assign a float variable to an integer we have to use:

```
int var=int (floatvar);
```

23. Explain basic to class type conversion with an example.

Conversion from basic data type to class type can be done in destination class. Using constructors does it. Constructor takes a single argument whose type is to be converted.

Eg: Converting int type to class type

```
class time
{
        int hrs,mins;
        public:
        ……
        Time ( int t) //constructor
        {
                hours= t/60 ; //t in minutes
                mins =t % 60;
        }
};
```

24. State the three types of data conversions.

- Conversion from Built-in data type to an object
- Conversion from object to a Built-in data type
- Conversion from one object type to another object type

25. Define inheritance

Inheritance is the process of creating new classes from the existing classes. The new classes are called derived classes. The existing classes are called base classes. The derived classes inherit all the properties of the base class plus its own properties. The properties of inheritance do not affect the base classes.

26. What are the types of inheritance?

- Single inheritance
- Multiple inheritance
- Hierarchical inheritance
- Multilevel inheritance.
- Hybrid inheritance
- Multipath inheritance

27. What are the advantages of using a derived class?

- New classes can be derived by the user from the existing classes without any modification.
- It saves time and money.
- It reduces program coding time.
- It increases the reliability of the program.

28. Define single inheritance.

If a derived class is derived from a base class then it is called single inheritance.

29. Define multiple inheritance.

If a derived class is derived from more than one base class, then it is called multiple inheritance.

30. Define hierarchical inheritance.

If two or more derived classes are derived from the same base class then it is known as hierarchical inheritance.

31. Define multilevel inheritance.

If a derived class is derived from another derived class then it is known as multilevel inheritance.

32. Define hybrid inheritance.

The combination of one or more types of inheritance is called hybrid inheritance.

33. Define multipath inheritance.

When a class is derived from two or more classes which are derived from the same base class then such type of inheritance is called multipath inheritance.

34. What is virtual base class?

To overcome the ambiguity occurred due to multipath inheritance, C++ provides the keyword virtual. The keyword virtual declares the specified classes virtual. When classes are declared as virtual, the compiler takes necessary precaution to avoid duplication of member variables. Thus, we make a class virtual if it is a base class that has been used by more than one derived class as their base class.

35. How do you define constructor with inheritance concept?

The derived class does not require a constructor if the base class contains zero-argument constructor. In case the base class has parameterized constructor then it is essential for the derived class to have a constructor. The derived class constructor passes arguments to the base class constructor.

In inheritance, normally derived classes are used to declare objects. Hence, it is necessary to define constructor in the derived class. When an object of a derived class is declared, the constructors of base and derived classes are executed

36. Define abstract class.

A class is said to be an abstract class if it satisfies the following conditions

- It should act as a base class
- It should not be used to create any objects

37. Define common constructor

When a class is derived from another then it is possible to define a constructor in derived class and data members of both base and derived classes can be initialized. It is not essential to declare constructor in a base class. Thus, the constructor of the derived class works for its base class and such constructors are called as **common constructors.**

38. Define common constructor

When a class is derived from another then it is possible to define a constructor in derived class and data members of both base and derived classes can be initialized. It is not essential to declare constructor in a base class. Thus, the constructor of the derived class works for its base class and such constructors are called as **common constructors.**

39. State the advantage of inheritance

inheritance has two main advantages:

1.extendability

we can extend the already made classes by adding some new features.

2.maintainability

it is easy to debug a program when divided in parts.

inheritance provides an opportunity to capture the problem

40. What are the disadvantages of inheritance

Since inheritance inherits everything from the super class and interface, it may make the subclass too clustering and sometimes error-prone when dynamic overriding or dynamic

overloading in some situation. In addition, the inheritance may make peers hardly understand your code if they don't know how your super-class acts and add learning curve to the process of development.

42. what is default constructor?

Default Constructor is also called as Empty Constructor which has no arguments and It is Automatically called when we creates the object of class but Remember name of Constructor is same as name of class and Constructor never declared with the help of Return Type. Means we cant Declare a Constructor with the help of void Return Type. , if we never Pass or Declare any Arguments then this called as the Copy Constructors.

43. Explain constructor with parameter

This is Another type Constructor which has some Arguments and same name as class name but it uses some Arguments So For this We have to create object of Class by passing some Arguments at the time of creating object with the name of class. When we pass some Arguments to the Constructor then this will automatically pass the Arguments to the Constructor and the values will retrieve by the Respective Data Members of the Class.

44. Explain multiple constructor?

Multi constructor is also called as Empty Constructor which has no arguments and It is Automatically called when we creates the object of class but Remember name of Constructor is same as name of class and Constructor never declared with the help of Return Type. Means we cant Declare a Constructor with the help of void Return Type. , if we never Pass or Declare any Arguments then this called as the Copy Constructors.

45.What is copy constructor?

This is also Another type of Constructor. In this Constructor we pass the object of class into the Another Object of Same Class. As name Suggests you Copy, means Copy the values of one Object into the another Object of Class .This is used for Copying the values of class object into an another object of class So we call them as Copy Constructor and For Copying the values We have to pass the name of object whose values we wants to Copying and When we are using or passing an Object to a Constructor then we must have to use the & Ampersand or Address Operator.

46. List out various types of type conversion?

- conversion from basic type to class type
- conversion from class type to basic type
- conversion from one class type to another class type

47. What is an explicit constructor?

A conversion constructor declared with the explicit keyword.The compiler does not use an explicit constructor to implement an implied conversion of types. it’s purpose is reserved explicitly for construction

48. What are the types in inheritance?

- single inheritance

- multiple inheritance
- multilevel inheritance
- hierarchial inheritance
- hybrid inheritance

49. What is hybrid inheritance?

there could be situation where we need to apply two or more type of inheritance to design a program. this is called hybrid inheritance

50.what are the types in run time polymorphism?

- Function overloading
- operator overloading

51. What is RTTI?

Runtime type identification(RTTI) lets you find the dynamic type of an object when you have only a pointer or a reference to the base type. RTTI is the official way in standard c++ to discover the type of an object and to convert the type of a pointer or reference

52. List out RTTI types?

- Basic types
- Command types
- Container types

53.List out the member function accessibility

- class member function
- derived member function
- friend member function
- friend class member function

54.What is abstract class

An abstract class is one that is not used to create objects. An abstract class is designed only to act as a base class. it is a design concept is program development and provides a base upon which other classes may be built

55.What is the advantages of using dynamic initialization?

- various initialization formats an be provided using overloaded constructor

56.What are copy constructor?

- The constructor that creates a new class object from an existing object of the same class

57.Define type conversion?

- A conversion of value from one data type to another

58. When and how the conversion function exists?

- To convert the data from a basic type to user defined type. The conversion should be defined in user defined object's class in the form of a constructor. The constructor function takes a single argument of basic data type

59.What is an explicit constructor?

- A conversion constructor declared with the explicit keyword. The compiler does not use an explicit constructor to implement an implied conversion of types. it's purpose is reserved explicitly for construction

60.How can we over load a function?

With the help of a special operator called operator function. The general form of an operator function is

```
Return type class name :: operator op(arg list)
{
Fuction body
}
```

11 MARKS

1. EXPLAIN IN DETAIL ABOUT CONSTRUCTORS AND DESTRUCTORS

CONSTRUCTORS

- A constructor is a special member function whose task is to initialize the objects of its class.
- It is special because its name is same as the class name.
- The constructor is invoked whenever an object of its associated class is created.
- It is called constructor because it constructs the values of data members of the class.

CONSTRUCTOR - EXAMPLE

```
class add
{
    int m, n ;
  public :
    add () ;
    ------
};
add :: add ()
{
  m = 0; n = 0;
}
```

When a class contains a constructor, it is guaranteed that an object created by the class will be initialized automatically.

```
add a ;
```

Not only creates the object a of type add but also initializes its data members m and n to zero.

- There is no need to write any statement to invoke the constructor function.
- If a 'normal' member function is defined for zero initialization, we would need to invoke this function for each of the objects separately.

CHARACTERISTICS OF CONSTRUCTORS

- They should be declared in the public section.
- They are invoked automatically when the objects are created.
- They do not have return types, not even void and they cannot return values.
- They cannot be inherited, though a derived class can call the base class constructor.
- Like other C++ functions, Constructors can have default arguments.
- Constructors cannot be virtual.
- We cannot refer to their addresses.
- An object with a constructor (or destructor) cannot be used as a member of a union.
- They make 'implicit calls' to the operators ***new*** and ***delete*** when memory allocation is required.

TYPES OF CONSTRUCTORS

There are three types of constructors. They are

- Default Constructor
- Parameterized Constructor
- Copy Constructor

DEFAULT CONSTRUCTOR

- A constructor that accepts no parameters is called the default constructor.
- The default constructor for class A is A : : A ( ) in above program

PARAMETERIZED CONSTRUCTORS

It may be necessary to initialize the various data elements of different objects with different values when they are created. This is achieved by passing arguments to the constructor function when the objects are created.

The constructors that can take arguments are called parameterized constructors.

EXAMPLE:

```
class add
{
    int m, n ;
  public :
    add (int, int) ;
    ------
};
add : : add (int x, int y)
{
  m = x; n = y;
}
```

When a constructor is parameterized, we must pass the initial values as arguments to the constructor function when an object is declared.

TWO WAYS OF CALLING:

There are two ways of calling parameterized constructors. They are

- Explicit

  add sum = add(2,3);

- Implicit

  add sum(2,3)

  This method is also known as Shorthand method

COPY CONSTRUCTOR

A copy constructor is used to declare and initialize an object from another object.

```
class add
{
    int m, n ;
```

```
  public :
    add (add &a) ;
    ------
};
add:: add (add &a)
{
      m = a.m ;
      n = a.n ;
}
```

Add a2 (a1) ; or Add a2 = a1 ;

- The process of initializing through a copy constructor is known as ***copy initialization***.
- A reference variable has been used as an argument to the copy constructor.
- We cannot pass the argument by value to a copy constructor.

CONSTRUCTORS WITH DEFAULT ARGUMENTS

It is possible to define constructors with default arguments.

Consider complex (float real, float imag = 0);

The default value of the argument imag is zero.

complex C1 (5.0) assigns the value 5.0 to the real variable and 0.0 to imag.
complex C2(2.0,3.0) assigns the value 2.0 to real and 3.0 to imag.

DYNAMIC INITIALIZATION OF OBJECTS

Providing initial value to objects at run time.

Advantage :

- We can provide various initialization formats, using overloaded constructors.
- This provides the flexibility of using different format of data at run time depending upon the situation.

DYNAMIC CONSTRUCTORS

- The constructors can also be used to allocate memory while creating objects.
- This will enable the system to allocate the right amount of memory for each object when the objects are not of the same size.
- Allocation of memory to objects at the time of their construction is known as dynamic construction of objects.
- The memory is created with the help of the new operator

DESTRUCTORS

A destructor is used to destroy the objects that have been created by a constructor. Like constructor, the destructor is a member function whose name is the same as the class name but is preceded by a tilde.

eg: ~ Add ( ) { }

- A destructor never takes any argument nor does it return any value.
- It will be invoked implicitly by the compiler upon exit from the program – or block or function as the case may be – to clean up storage that is no longer accessible.
- It is a good practice to declare destructors in a program since it releases memory space for further use.
- Whenever ***new*** is used to allocate memory in the constructor, we should use ***delete*** to free that memory.

2. EXPLAIN IN DETAIL ABOUT CONSTRUCTOR OVERLOADING

CONSTRUCTORS

- A constructor is a special member function whose task is to initialize the objects of its class.
- It is special because its name is same as the class name.
- The constructor is invoked whenever an object of its associated class is created.
- It is called constructor because it constructs the values of data members of the class.

CONSTRUCTOR - EXAMPLE

```
class add
{
    int m, n ;
  public :
    add () ;
    ------
};
add :: add ()
{
  m = 0; n = 0;
}
```

When a class contains a constructor, it is guaranteed that an object created by the class will be initialized automatically.

add a ;

Not only creates the object a of type add but also initializes its data members m and n to zero.

- There is no need to write any statement to invoke the constructor function.

CHARACTERISTICS OF CONSTRUCTORS

- They should be declared in the public section.
- They are invoked automatically when the objects are created.
- They do not have return types, not even void and they cannot return values.
- They cannot be inherited, though a derived class can call the base class constructor.

TYPES OF CONSTRUCTORS

There are three types of constructors. They are

- Default Constructor
- Parameterized Constructor
- Copy Constructor

DEFAULT CONSTRUCTOR

- A constructor that accepts no parameters is called the default constructor.
- The default constructor for class A is A : : A ( ) in above program

PARAMETERIZED CONSTRUCTORS

- It may be necessary to initialize the various data elements of different objects with different values when they are created.
- This is achieved by passing arguments to the constructor function when the objects are created.
- The constructors that can take arguments are called parameterized constructors.

COPY CONSTRUCTOR

- A copy constructor is used to declare and initialize an object from another object.
- The process of initializing through a copy constructor is known as ***copy initialization***.
- A reference variable has been used as an argument to the copy constructor.
- We cannot pass the argument by value to a copy constructor.

CONSTRUCTOR OVERLOADING

Like functions, it is also possible to overload constructors. A class can contain more than one constructor. This is known as constructor overloading.

- All constructors are defined with the same name as the class.
- All the constructors contain different number of arguments.
- Depending upon number of arguments, the compiler executes appropriate constructor.

EXAMPLE:

```
class add
{
        int m, n ;

    public :
        add()           // Default constructor
        {
            m = 0 ;
            n = 0 ;
        }

        add(int a)      //Parameterized Constructor with one argument
        {
            m = a ;
        }

        add(int a,int b) //Parameterized Constructor with two arguments
        {
            m = a ;
```

```
                n = b ;
        }

        add (add &i)   //Copy Constructor
        {
                m = i.m ;
                n = i.n ;
        }
};

void main()
{
        add a1;
        add a2(10);
        add a3(10,20);
        add a4(a3);
}
```

EXPLANATION

- The first constructor receives no arguments.
- The second constructor receives one integer argument.
- The third constructor receives two integer arguments.
- The fourth constructor receives one add object as an argument.

add a1;
Would automatically invoke the first constructor and set both m and n of a1 to zero.

add a2(10);
Would call the second constructor which will initialize the data member m of a2 to 10.

add a3(10,20);
Would call the third constructor which will initialize the data members m and n of a3 to 10 and 20 respectively.

add a4(a3);
Would invoke the fourth constructor which copies the values of a3 into a4.

3. EXPLAIN IN DETAIL ABOUT CONSTRUCTORS

Dynamic Initialization of Objects

- In C++, the class objects can be initialized at run time (dynamically). We have the flexibility of providing initial values at execution time. The following program illustrates this concept:

```
//Illustration of dynamic initialization of objects

#include <iostream.h>
```

```
#include <conio.h>

Class employee

{

Int empl_no;

Float salary;
Public:

Employee() //default constructor {}

Employee(int empno,float s)//constructor with arguments

{

Empl_no=empno;

Salary=s;

}

Employee (employee &emp)//copy constructor

{

Cout<<"\ncopy constructor working\n"; Empl_no=emp.empl_no;
Salary=emp.salary;

}

Void display (void)

{

Cout<<"\nEmp.No:"<<empl_no<<"salary:"<<salary<<end1;

}

};

Void main()

{
```

```
int eno; float sal; clrscr();

cout<<"Enter the employee number and salary\n"; cin>>eno>>sal;

employee obj1(eno,sal);//dynamic initialization of object cout<<"\nEnter the employee number and
salary\n"; cin>eno>>sal;

        employee obj2(eno,sal); //dynamic initialization of object

        obj1.display(); //function called

        employee obj3=obj2; //copy constructor called

        obj3.display();

        getch();

    }
```

Constructors and Primitive Types

In C++, like derived type, i.e. class, primitive types (fundamental types) also have their constructors. Default constructor is used when no values are given but when we given initial values, the initialization take place for newly created instance. For example,

```
float x,y; //default constructor used

int a(10), b(20); //a,b initialized with values 10 and 20

float i(2.5), j(7.8); //I,j, initialized with valurs 2.5 and 7.8
```

Constructor with Default Arguments

Like functions, it is also possible to declare constructors with default arguments. Consider the following example.

```
power (int 9, int 3);
```

In the above example, the default value for the first argument is nine and three for second.

```
power p1 (3);
```

In this statement, object p1 is created and nine raise to 3 expression n is calculated. Here, one argument is absent hence default value 9 is taken, and its third power is calculated. Consider the example on the above discussion given below.

Example program to declare default arguments in constructor. Obtain the power of the number.

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

class power

{
    private:
    int num;
    int power;
    int ans;

    public :

power (int n=9,int p=3); // declaration of constructor with default arguments

    void show( )
    {
      cout <<"\n"<<num <<" raise to "<<power <<" is " <<ans;
    }
};

 power :: power (int n,int p )
 {
    num=n;
    power=p;
    ans=pow(n,p);
 }

main( )
{
 clrscr( );
 class power p1,p2(5);
 p1.show( );
 p2.show( );
 return  0;
}
```

SPECIAL CHARACTERISTICS OF CONSTRUCTORS

These have some special characteristics. These are given below:

- These are called automatically when the objects are created.
- All objects of the class having a constructor are initialized before some use.

- These should be declared in the public section for availability to all the functions.
- Return type (not even **void**) cannot be specified for constructors.
- These cannot be inherited, but a **derived** class can call the base class constructor.
- These cannot be static.
- Default and copy constructors are generated by the compiler wherever required. Generated constructors are public.
- These can have default arguments as other C++ functions.
- A constructor can call member functions of its class.
- An object of a class with a constructor cannot be used as a member of a **union.**
- A constructor can call member functions of its class.
- We can use a constructor to create new objects of its class type by using the syntax.

Name_of_the_class (expresson_list)

For example,

Employee obj3 = obj2; // see program 10.5

Or even

Employee obj3 = employee (1002, 35000); //explicit call

- The make **implicit** calls to the memory allocation and deallocation operators **new** and **delete.**
- These cannot be **virtual.**

4. EXPLAIN IN DETAIL ABOUT DESTRUCTORS WITH SAMPLE PROGRAM

Declaration and Definition of a Destructor

The syntax for declaring a destructor is :

-name_of_the_class()

{

}

So the name of the class and destructor is same but it is prefixed with a ~

(tilde). It does not take any parameter nor does it return any value. Overloading a destructor is not possible and can be explicitly invoked. In other words, a class can have only one destructor. A destructor can be defined outside the class. The following program illustrates this concept :

```
//Illustration of the working of Destructor function #include<iostream.h>
#include<conio.h>



class add

{

        private :

          int num1,num2,num3; public :

        add(int=0, int=0); //default argument constructor //to reduce the number of
                                constructors

        void sum(); void display();

        ~ add(void); //Destructor

};

//Destructor definition ~add()
Add:: ~add(void) //destructor called automatically at end of program

Obj1.sum(); //function call
Obj2.sum();
Obj3.sum();

cout<<"\nUsing obj1 \n";

obj1.display(); //function call

cout<<"\nUsing obj2 \n";
```

```
obj2.display();

cout<<"\nUsing obj3 \n";

obj3.display();

}
```

Special Characteristics of Destructors

Some of the characteristics associated with destructors are :

- These are called automatically when the objects are destroyed.
- Destructor functions follow the usual access rules as other member functions.
- These **de-initialize** each object before the object goes out of scope.
- No argument and return type (even void) permitted with destructors.
- These cannot be inherited.
- **Static** destructors are not allowed.
- Address of a destructor cannot be taken.
- A destructor can call member functions of its class.
- An object of a class having a destructor cannot be a member of a union.

5. WRITE IN DETAIL ABOUT DYNAMIC INITIALIZATION AND RECURSIVE CONSTRUCTOR

CALLING CONSTRUCTORS AND DESTRUCTORS

- The compiler automatically calls the constructor and destructor. We can also call the constructor and destructor in the same fashion as we call the normal user-defined function. The calling methods are different for constructors and destructors. In practice, it may not be useful but for the sake of understanding, few examples are illustrated on this concept

Example program to invoke constructor and destructor.

```
# include <iostream.h>
# include <conio.h>
class byte
{
   int bit;
   int bytes;

   public :

   byte( )
   {
     cout <<"\n Constructor invoked";
     bit=64;
     bytes=bit/8;
   }

   ~byte( )
   {
     cout <<"\n Destructor invoked ";
     cout <<"\n Bit = "<<bit;
     cout <<"\n Byte = "<<bytes;
   }
};

int main ( )
{
   clrscr( );
   byte x;
   byte( );          // calling constructor

   // x.byte( ) ;      // invalid statement
   // x.byte::byte( )  // valid statement
   // ~ byte( );       // invalid statement
   // byte.~byte( );   // invalid statement
   // x.~byte( );      // Member identifier expected

   x.byte::~byte( );

   return 0;
}
```

DYNAMIC INITIALIZATION USING CONSTRUCTORS

Example program to initialize member variables using pointers and constructors.

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

class city
{
  char city[20];
  char state[20];
  char country[20];

public:

city( )
{
city[0]=state[0]=country[0]=NULL;
}

void display( char *line);

city(char *cityn)
{

strcpy(city, cityn);
state[0]=NULL;
}

city(char *cityn,char *staten)
{
  strcpy(city,cityn);
  strcpy(state,staten);
  country[0]=NULL;
}

city(char *cityn,char *staten, char *countryn)
{
  _fstrcpy(city,cityn);
  _fstrcpy(state,staten);
  _fstrcpy(country,countryn);
}
};

void city:: display (char *line)
```

```
{
  cout <<line<<endl;
  if (_fstrlen(city))     cout<<"City    : "<<city<<endl;
  if (strlen(state))    cout <<"State   : "<<state <<endl;
  if (strlen(country))  cout <<"Country : "<<country <<endl;
}
void main( )
{
  clrscr( );
  city c1("Mumbai"),
       c2("Nagpur","Maharashtra"),
       c3("Nanded","Maharashtra","India"),
       c4('\0','\0','\0');

  c1.display("=========*============");
  c2.display("=========*============");
....
```

RECURSIVE CONSTRUCTOR

Like normal and member functions, constructors also support recursion. The program given next explains this.

Example program to invoke constructor recursively and calculate the triangular number of the entered number.

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

class tri_num
{
  int f;
  public :

tri_num( )
{
f=0;
 }

void sum( int j)
{
f=f+j;
```

```
}

tri_num(int m)
{
      if (m==0)
      {
       cout <<"Triangular number : "<<f;
       exit(1);
      }

    sum(m);
    tri_num::tri_num(--m);
}

 };

void main( )
{
clrscr( );

tri_num a;
a.tri_num::tri_num(5);
}
```

OUTPUT

Triangular number : 15

Explanation*:* In the above program, class tri_num is declared with private integer f and member function sum( ). The class also has zero-argument constructor and one–argument constructor.

6. EXPLAIN ABOUT OPERATOR OVERLOADING

Operator Overloading

- Operator overloading is a very important feature of Object Oriented Programming. It is because by using this facility programmer would be able to create new definitions to existing operators. In other words a single operator can perform several functions as desired by programmers.

- Operators can be broadly classified into:
    - Unary Operators
    - Binary Operators

Unary Operators:

- As the name implies takes operate on only one operand. Some unary operators are namely

++ - Increment operator
-- - Decrement Operator
! - Not operator
- - unary minus

Binary Operators:

The arithmetic operators, comparison operators, and arithmetic assignment operators come under this category.

Both the above classification of operators can be overloaded. So let us see in detail each of this.

Operator Overloading – Unary operators

As said before operator overloading helps the programmer to define a new functionality for the existing operator. This is done by using the keyword **operator**.

The general syntax for defining an operator overloading is as follows:

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

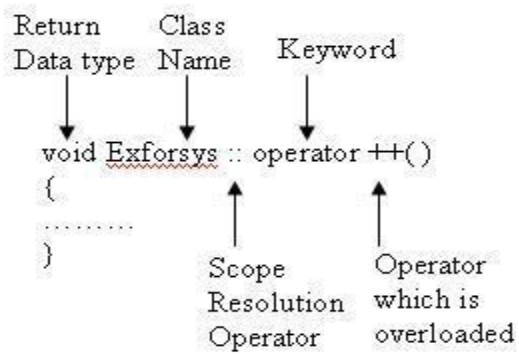
Thus the above clearly specifies that operator overloading is defined as a member function by making use of the keyword operator.

In the above:

- return_type – is the data type returned by the function
- class name - is the name of the class
- operator – is the keyword
- operator symbol – is the symbol of the operator which is being overloaded or defined for new functionality
- **::** - is the scope resolution operator which is used to use the function definition outside the class.

For example

Suppose we have a class say Exforsys and if the programmer wants to define a operator overloading for unary operator say ++, the function is defined as



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

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

The important steps involved in defining an operator overloading in case of unary

operators are namely:

- Inside the class the operator overloaded member function is defined with the return data type as member function or a friend function.
- If the function is a member function then the number of arguments taken by the operator member function is none.
- If the function defined for the operator overloading is a friend function then it takes one argument.

```
#include <iostream.h>
class Exforsys
{
   private:
   int x;
   public:
   Exforsys( ) { x=0; }       //Constructor
   void display();
   void Exforsys ++( );          //overload unary ++
};
void Exforsys :: display()
{
```

```
    cout<<‖\nValue of x is: — << x;
}
void Exforsys :: operator ++( ) //Operator Overloading for operator ++ defined
{
    ++x;
}
void main( )
{
    Exforsys e1,e2; //Object e1 and e2 created cout<<‖Before
    Increment‖
    cout <<‖\nObject e1: ‖<<e1.display(); cout <<‖\nObject
    e2: ‖<<e2.display();
    ++e1; //Operator overloading applied
    ++e2;
    cout<<‖\n After Increment‖
    cout <<‖\nObject e1: ‖<<e1.display(); cout <<‖\nObject
    e2: ‖<<e2.display();
}
```

The output of the above program is:

```
Before Increment
Object e1:
Value of x is: 0
Object e1:
Value of x is: 0
Before Increment
Object e1:
Value of x is: 1
Object e1:
Value of x is: 1
```

- In the above example we have created 2 objects e1 and e2 f class Exforsys. The operator ++ is overloaded and the function is defined outside the class Exforsys.

- When the program starts the constructor Exforsys of the class Exforsys initialize the values as zero and so when the values are displayed for the objects e1 and e2 it is displayed as zero. When the object ++e1 and ++e2 is called the operator overloading function gets applied and thus value of x gets incremented for each object separately.

- So now when the values are displayed for objects e1 and e2 it is incremented once each and gets printed as one for each object e1 and e2.

Operator Overloading – Binary Operators

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

- The difference is in the number of arguments used by the function. In the case of binary operator overloading, when the function is a member function then the number of arguments used by the operator member function is one (see below example).

- When the function defined for the binary operator overloading is a friend function, then it uses two arguments.

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

Binary operator overloading example:

```
#include <iostream.h> class Exforsys
{
        private:
                int x; int y;
        public:
                Exforsys()      //Constructor
                {
                        x=0;
                        y=0;
                }
        void getvalue( ) //Member Function for Inputting Values
        {
                cout << —\n Enter value for x: —; cin >> x;
                cout << —\n Enter value for y: —; cin>> y;
        }

        void displayvalue( )    //Member Function for Outputting Values
        {

                cout <<‖value of x is: — << x <<‖; value of y is: —<<y
        }

        Exforsys operator +(Exforsys);
};

Exforsys Exforsys :: operator + (Exforsys e2) //Binary operator overloading for + operator
```

```
{
        int x1 = x+ e2.x; int y1 = y+ e2.y;
        return Exforsys(x1,y1);
}

void main( )
{
        Exforsys e1,e2,e3; //Objects e1, e2, e3 created
        cout<<\n‖Enter value for Object e1:‖; e1.getvalue( );
        cout<<\n‖Enter value for Object e2:‖;
        e2.getvalue( );
        e3= e1+ e2; //Binary Overloaded operator used
        cout<< —\nValue of e1 is:‖<<e1.displayvalue();
        cout<< —\nValue of e2 is:‖<<e2.displayvalue();
        cout<< —\nValue of e3 is:‖<<e3.displayvalue();
}
```

Output

```
Enter value for Object e1:
Enter value for x: 10
Enter value for y: 20
Enter value for Object e2:
Enter value for x: 30
Enter value for y: 40
Value of e1 is: value of x is: 10; value of y is: 20
Value of e2 is: value of x is: 30; value of y is: 40
Value of e3 is: value of x is: 40; value of y is: 60
```

- In the above example, the class Exforsys has created three objects e1, e2, e3. The values are entered for objects e1 and e2. The binary operator overloading for the operator _+‘ is declared as a member function inside the class Exforsys. The definition is performed outside the class Exforsys by using the scope resolution operator and the keyword operator.

- The important aspect is the statement: e3= e1 + e2;

- The binary overloaded operator _+‘ is used. In this statement, the argument on the left side of the operator _+‘, e1, is the object of the class Exforsys in which the binary overloaded operator _+‘ is a member function. The right side of the operator _+‘ is e2. This is passed as an argument to the operator _+‘ .

- Since the object e2 is passed as argument to the operator‘+‘ inside the function defined for binary operator overloading, the values are accessed as e2.x and e2.y. This is added with e1.x and e1.y, which are accessed directly as x and y. The return value is of type class Exforsys as defined by the above example.

- There are important things to consider in operator overloading with C++ programming language. Operator overloading adds new functionality to its existing operators. The programmer must add proper comments concerning the new functionality of the overloaded operator. The program will be efficient and readable only if operator overloading is used only when necessary.

Some operators cannot be overloaded:

- Scope resolution operator denoted by ::
- Member access operator or the dot operator denoted by .
- Conditional operator denoted by ?:
- Pointer to member operator denoted by .*

7. EXPLAIN IN DETAIL ABOUT OPERATOR OVERLOADING USING FRIEND FUNCTIONS

Operator Overloading using friend functions

```
// Using friend functions to overload addition and subtraction operators
#include <iostream.h>
class myclass
 {
        int a; int b;
        public:
                myclass(){}
                myclass(int x,int y)
                {
                        a=x;
                        b=y;
                }

                void show()
                {
                        cout<<a<<endl<<b<<endl;
                }
// these are friend operator functions
// NOTE: Both the operans will be be passed explicitely.
// operand to the left of the operator will be passed as the first argument and operand to the
// right as the second argument
                friend myclass operator+(myclass,myclass);
                friend myclass operator-(myclass,myclass);
 };

myclass operator+(myclass ob1,myclass ob2)
{
```

```
        myclass temp;
        temp.a = ob1.a + ob2.a; temp.b = ob1.b + ob2.b;
        return temp;
}
myclass operator-(myclass ob1,myclass ob2)
{
        myclass temp;
        temp.a = ob1.a - ob2.a; temp.b = ob1.b - ob2.b;
        return temp;
}
void main()
{
        myclass a(10,20);
        myclass b(100,200);
        a=a+b;
        a.show();
}
```

Overloading the Assignment Operator (=)

- Objects of a class to be operated by common operators then we need to overload them. But there is one operator whose operation is automatically crested by C++ for every class we define, it is the assignment operator '='.

- Actually we have been using similar statements like the one below previously

  ob1=ob2; where ob1 and ob2 are objects of a class.

- This is because even if we don't overload the '=' operator, the above statement is valid because C++ automatically creates a default assignment operator. The default operator created, does a member-by-member copy, but if we want to do something specific we may overload it.

- The simple program below illustrates how it can be done. Here we are defining two similar classes, one with the default assignment operator (created automatically) and the other with the overloaded one. Notice how we could control the way assignments are done in that case.

```
// Program to illustrate the overloading of assignment operator '='

#include <iostream.h>
// class not overloading the assignment operator

class myclass
{
        int a; int b;
```

```
        public:
                myclass(int, int);
                void show();
  };

myclass::myclass(int x,int y)
{
        a=x;
        b=y;
}

void myclass::show()
{
        cout<<a<<endl<<b<<endl;
}
// class having overloaded assignment operator
class myclass2
{
        int a; int b;
        public:

                myclass2(int, int);
                void show();
                myclass2 operator=(myclass2);
};
myclass2 myclass2::operator=(myclass2 ob)
{
     // -- do something specific— this is just to illustrate that when overloading _=‘
     // we can define our own way of assignment
     b=ob.b;
     return *this;
};


myclass2::myclass2(int x,int y)
{
        a=x;
        b=y;
}

void myclass2::show()
{
        cout<<a<<endl<<b<<endl;
}
```

```
// main void main()
{
        myclass ob(10,11);
        myclass ob2(20,21);
        myclass2 ob3(100,110);
        myclass2 ob4(200,210);
// does a member-by-member copy _=‘ operator is not overloaded ob=ob2;
        ob.show();
// does specific assignment as defined in the overloaded operator definition
        ob3=ob4;
        ob3.show();
}
```

8. EXPLAIN IN DETAIL ABOUT TYPE CONVERSIONS IN C++

- We have overloaded several kinds of operators but we haven't considered the assignment operator (=). It is a very special operator having complex properties. We know that = operator assigns values form one variable to another or assigns the value of user defined object to another of the same type. For example,

```
int     x, y ;

        x = 100;

        y = x;
```

Here, first 100 is assigned to x and then x to y.

Consider another statement, 13 = t1 + t2;

This statement used in program 11.2 earlier, assigns the result of addition, which is of type time to object t3 also of type time.

So the assignments between basic types or user defined types are taken care by the compiler provided the data type on both sides of = are of same type.

But what to do in case the variables are of different types on both sides of the = operator? In this case we need to tell to the compiler for the solution.

Three types of situations might arise for data conversion between different types :

(i) Conversion form basic type to class type.

(ii) Conversion from class type to basic type.

(iii) Conversion from one class type to another class type. Now let us

discuss the above three cases :

(i) Basic Type to Class Type

This type of conversion is very easy. For example, the following code segment converts an int type to a class type.

```
class distance
    {
        int feet;
        int inches;
        public:
        .....
        .....
    distance (int dist) //constructor
       {
        feet = dist/12;
        inches = dist%12;
    }
};
```

The following conversion statements can be coded in a function :

```
distance dist1; //object dist1 created

int length = 20;
dist1=length; //int to class type
```

After the execution of above statements, the **feet** member of **dist1** will have a value of 1 and **inches** member a value of 8, meaning 1 feet and 8 inches.

A class object has been used as the left hand operand of = operator, so the type conversion can also be done by using an overloaded = operator in C++.

(ii) Class Type to Basic Type

For conversion from a basic type to class type, the constructors can be used. But for conversion from a class type to basic type constructors do not help at all. In C++, we have to define an overloaded **casting operator** that helps in converting a class type to a basic type.

The syntax of the **conversion function** is given below:

```
Operator typename()
    {
        .....
        ..... //statements
    }
```

Here, the function converts a class type data to typename. For example, the **operator float ( )** converts a class type to type **float, the operator int ( )** converts a class type object to type int. For example,

```
matrix :: operator float ()
    {
        float sum = 0.0; for(int
    i=0;i<m;i++)
    {
        for (int j=0; j<n; j++)
           sum=sum+a[i][j]*a[i][j];
        }
    Return sqrt(sum); //norm of the matrix
}
```

Here, the function finds the norm of the matrix (Norm is the square root of the sum of the squares of the matrix elements). We can use the operator float ( ) as given below :

```
float norm = float (arr);
                or
```

float norm = arr;

where **arr** is an object of type matrix. When a class type to a basic type conversion is required, the compiler will call the casting operator function for performing this task.

The following conditions should be satisfied by the casting operator function :

(a) It must not have any argument

(b) It must be a class member

(c) It must not specify a return type.

(i) One Class Type to Another Class Type

- There may be some situations when we want to convert one class type data to another class type data. For example,

  Obj2 = obj1; //different type of objects

- Suppose **obj1** is an object of class **studdata** and **obj2** is that of class **result.** We are converting the class **studdata** data type to class **result** type data and the value is assigned to obj2. Here **studdata** is known as **source class** and **result** is known as the **destination class.**

- The above conversion can be performed in two ways :

  - Using a constructor.

  - Using a conversion function.

- When we need to convert a class, a casting operator function can be used i.e. source class. The source class performs the conversion and result is given to the object of destination class.

- If we take a single-argument constructor function for converting the argument's type to the class type (whose member it is). So the argument is of the source class and being passed to the destination class for the purpose of conversion. Therefore it is compulsory that the conversion constructor be kept in the destination class.

9. EXPLAIN THE DIFFERENT FORMS OF INHERITANCE IN DETAIL.

- The mechanism of deriving a new class from old class is called Inheritance
- The old class is referred as BASE CLASS or PARENT CLASS or SUPER CLASS
- The new class is referred as DERIVED CLASS or CHILD CLASS or SUB CLASS

There were three ways to derive

- Public inheritance means, "public parts of super class remain public and protected parts of super class remain protected."

- Private Inheritance means "Public and Protected Parts of Super Class remain private in Sub-Class".

- Protected Inheritance means "Public and Protected Parts of Super class remain protected in Subclass.

DEFINING DERIVED CLASS

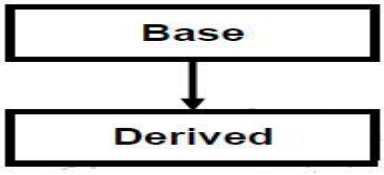
```
class derived-class name : visibility-mode base-class name
{
        //members of derived class
}
```

TYPES OF INHERITANCES

- Single Inheritance
- Multiple Inheritance
- Hierarchical Inheritance
- Multilevel Inheritance
- Hybrid Inheritance
- Multi-path Inheritance

SINGLE INHERITANCE

- The way of deriving a class from single class. So, there will be only one base class for the derived class.



Syntax:

class A

{ /*.....*/);
class B: visibility-mode A
{ /*
.
*/};

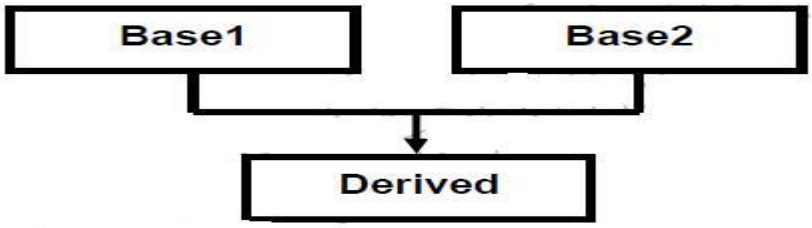
EXAMPLE:

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

class student                                          //Class Declaration
{
        protected:
                int regno,m1;           //Data Member Declaration
                char name[20],dept[5];

        public:
                void getdata()                  //Function to get student details
                {
                        cout<<"\nEnter the register number:";
                        cin>>regno;
                        cout<<"\nEnter the name:";
                        cin>>name;
                        cout<<"\nEnter the department:";
                        cin>>dept;
                        cout<<"\nEnter the mark:";
                        cin>>m1;
                }
};

class result: public student                    //Deriving new class from base class
{
        char result[6];

    public:
        void calculation()              //Function to calculate result of the student
        {
                if(m1>49)
                {
                        strcpy(result,"Pass");
                }
                else
                {
```

```
                strcpy(result,"Fail");
            }
        }
        void display()          //Function to display student details
        {
            cout<<"\n\t\t\tSTUDENT DETAILS";
            cout<<"\nRegister Number:"<<regno;
            cout<<"\nName:"<<name;
            cout<<"\nDepartment:"<<dept;
            cout<<"\nMark :"<<m1;
            cout<<"\nResult:"<<result;
        }
};

void main()
{
        clrscr();
        result r;                       //Object declaration for derived class

        cout<<"\n\t\t\t STUDENT INFORMATION";
        r.getdata();            //Calling Base class function using derived object
        r.calculation();
        r.display();

        getch();

}
```

MULTIPLE INHERITANCE

when a derived class is created from more than one base class then that inheritance is called as multiple inheritance



Syntax

```
class Derived-class-name: visibility-mode Base-class-name1,visibility-mode Base-class-name2,…
{
        Body of derived class
}
```

Example

```
class A
{…….};
class B
{…….};
class C : public A, public B
{…….};
```

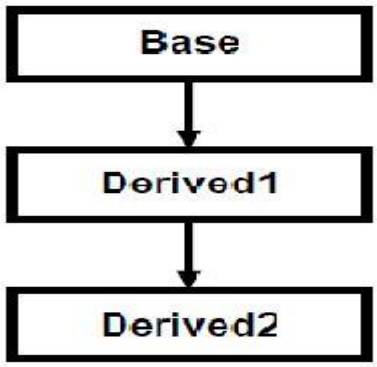
Program

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

class student                                   //Class Declaration
{
  protected:
                int regno;                      //Data Member Declaration
                char name[20],dept[5];
  public:
        void getdata()                         //Function to get student details
        {
                cout<<"\nEnter the register number:";
                cin>>regno;
                cout<<"\nEnter the name:";
                cin>>name;
                cout<<"\nEnter the department:";
                cin>>dept;
        }
};

class marks                                     //Class Declaration
{
  protected:
        int m1;                 //Data Member Declaration

  public:
        void getmarks()                 //Function to get student marks
                {
                        cout<<"\nEnter the mark 1:";
                        cin>>m1;
                }
};
class result : public student, public marks //Deriving new class from more two base class
{
```

```
    private:
                char result[6];
    public:
                void calculation()          //Function to calculate result of the student          {
                        if(m1>49)
                        {
                                strcpy(result,"Pass");
                        }
                        else
                        {
                                strcpy(result,"Fail");
                        }
                }

                void display()              //Function to display student details
                {
                        cout<<"\n\t\t\tSTUDENT DETAILS";
                        cout<<"\nRegister Number:"<<regno;
                        cout<<"\nName:"<<name;
                        cout<<"\nDepartment:"<<dept;
                        cout<<"\nMark:"<<m1;
                        cout<<"\nResult:"<<result;
                }
};
void main()
{
        clrscr();
        result r;                   //Object declaration for derived class

        cout<<"\n\t\t\t STUDENT INFORMATION";
        r.getdata();                //Calling Base class function using derived object
        r.getmarks();               //Calling Base class function using derived object
        r.calculation();
        r.display();

        getch();
}
```

MULTI LEVEL INHERITANCE

When a derived class is created from another derived class, then that inheritance is called as multi level inheritance.



SYNTAX:

```
Class A                         //Base Class
{
};

Class B: public A           //B derived from A
{.....};

Class C: public B               //C derived from B
{.....};
```

EXAMPLE PROGRAM:

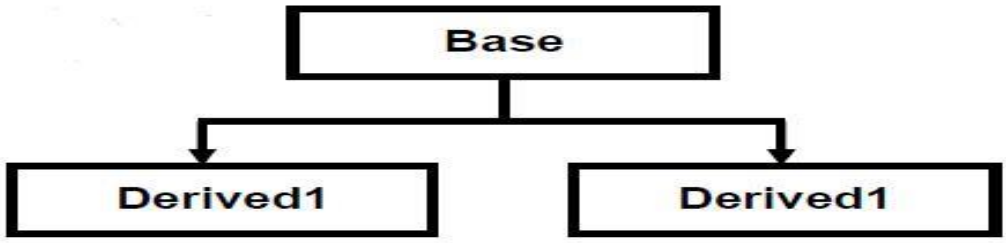
```
#include<iostream.h>
#include<conio.h>
#include<string.h>

class student                                           //Class Declaration
{
  protected:
            int regno;                                  //Data Member Declaration
            char name[20],dept[5];

  public:
            void getdata()                      //Function to get student details
            {
                  cout<<"\nEnter the register number:";
                  cin>>regno;
                  cout<<"\nEnter the name:";
                  cin>>name;
                  cout<<"\nEnter the department:";
                  cin>>dept;
            }
};

class marks: public student                     //Deriving new class from base class
```

```
{
  protected:
            int m1;                         //Data Member Declaration

  public:
            void getmarks()                 //Function to get student marks
            {
                  cout<<"\nEnter the mark 1:";
                  cin>>m1;
            }
};

class result: public marks          //Deriving new class from another derived class
{
  private:
            char result[6];

  public:

            void calculation()          //Function to calculate result of the student
            {
                  if(m1>49)
                  {
                        strcpy(result,"Pass");
                  }
                  else
                  {
                        strcpy(result,"Fail");
                  }
            }

            void display()                  //Function to display student details
            {
                  cout<<"\n\t\t\tSTUDENT DETAILS";
                  cout<<"\nRegister Number:"<<regno;
                  cout<<"\nName:"<<name;
                  cout<<"\nDepartment:"<<dept;
                  cout<<"\nMark:"<<m1;
                  cout<<"\nResult:"<<result;
            }
};

void main()
{
      clrscr();
      result r;                         //Object declaration for derived class

      cout<<"\n\t\t\t STUDENT INFORMATION";
      r.getdata();            //Calling Base class function using derived object
      r.getmarks();           //Calling Base class function using derived object
```

```
        r.calculation();
        r.display();

        getch();
  }
```

HIERARCHICAL INHERITANCE

when more than one derived class are created from a single base class, then that inheritance is called as hierarchical inheritance.



Example Program:

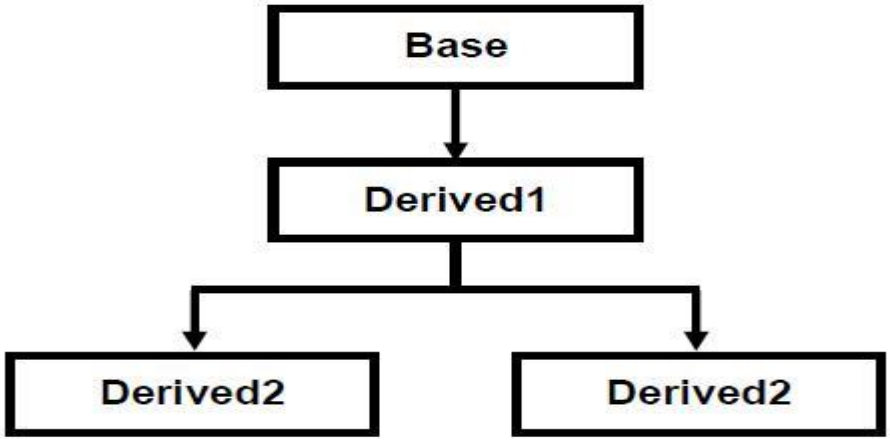
```
#include<iostream.h>
#include<conio.h>
#include<string.h>

class student                                   //Class Declaration
{
  protected:
        int regno;                              //Data Member Declaration
        char name[20],dept[5];

  public:
        void getdata()                          //Function to get student details
        {
            cout<<"\nEnter the register number:";
            cin>>regno;
            cout<<"\nEnter the name:";
            cin>>name;
            cout<<"\nEnter the department:";
            cin>>dept;
        }
};

class marks : public student                    //Class Declaration
{
  protected:
        int m1;                 //Data Member Declaration
```

```
        char result[6];

    public:
        void getmarks()                 //Function to get student marks
        {
            cout<<"\nEnter the mark:";
            cin>>m1;
        }

        void calculation()          //Function to calculate result of the student
        {
            if(m1>49)
            {
                strcpy(result,"Pass");
            }
            else
            {
                strcpy(result,"Fail");
            }
        }

        void display()              //Function to display student details
        {
            cout<<"\n\t\t\tSTUDENT DETAILS";
            cout<<"\nRegister Number:"<<regno;
            cout<<"\nName:"<<name;
            cout<<"\nDepartment:"<<dept;
            cout<<"\nMark :"<<m1;
            cout<<"\nResult:"<<result;
        }
};

class attendence: public student                //Deriving new class from base class
{
    private:
        int tod, top;
        float attper;

    public:

        void getattend()
        {
            cout<<"Enter the total number of days";
            cin>>tod;
```

```
                cout<<"\nEnter the total number of days present";
                cin>>top;
        }

        void calattend()
        {
                attper=(top/tod)*100;
        }

        void disattend()        //Function to display student attendence details
        {
                cout<<"\n\t\t\tSTUDENT DETAILS";
                cout<<"\nRegister Number:"<<regno;
                cout<<"\nName:"<<name;
                cout<<"\nDepartment:"<<dept;
                cout<<"\nTotal Number of Days:"<<tod;
                cout<<"\nNumber of Days present:"<<top;
                cout<<"\nAttendence Percentage"<<attper;
        }
};
void main()
{
        clrscr();
        marks m;        //Object declaration for derived class
        attendance d;

        cout<<"\n\t\t\t MARK INFORMATION";
        m.getdata();                    //Calling Base class function using derived object
        m.getmarks();                   //Calling Base class function using derived object
        m.calculation();
        m.display();

        cout<<"\n\t\t\t ATTENDANCE INFORMATION";
        d.getdate();
        d.getattend();
        d.calattend();
        d.disattend();

        getch();

}
```

HYBRID INHERITANCE

Any combination of single, hierarchical and multi level inheritances is called as hybrid inheritance.



.

Example program:

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

class student                                   //Class Declaration
{
  protected:
            int regno;                          //Data Member Declaration
            char name[20],dept[5];

  public:
            void getdata()                      //Function to get student details
            {
                  cout<<"\nEnter the register number:";
                  cin>>regno;
                  cout<<"\nEnter the name:";
                  cin>>name;
                  cout<<"\nEnter the department:";
                  cin>>dept;
            }
};

class marks: public student                             //Class Declaration
{
  protected:
            int m1;                 //Data Member Declaration
            char result[6];
```

```
  public:
                void getmarks()                 //Function to get student marks
                {
                        cout<<"\nEnter the mark 1:";
                        cin>>m1;
                }

                void calculation()          //Function to calculate result of the student
                {
                        if(m1>49)
                        {
                                strcpy(result,"Pass");
                        }
                        else
                        {
                                strcpy(result,"Fail");
                        }
                }

};

class attendence                                //Deriving new class from base class
{
  private:
                int tod, top;
                float attper;

  public:

                void getattend()
                {
                        cout<<"Enter the total number of days";
                        cin>>tod;
                        cout<<"\nEnter the total number of days present";
                        cin>>top;
                        attper=(top/tod)*100;
                }
};

class result: public marks, public attendance
{
        void display()                          //Function to display student details
                {
                        cout<<"\n\t\t\tSTUDENT DETAILS";
                        cout<<"\nRegister Number:"<<regno;
                        cout<<"\nName:"<<name;
                        cout<<"\nDepartment:"<<dept;
                        cout<<"\nMark :"<<m1;
                        cout<<"\nResult:"<<result;
                        cout<<"\nTotal Number of Days:"<<tod;
```

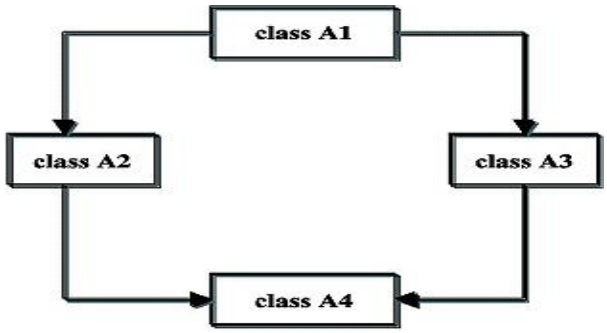
```
                cout<<"\nNumber of Days present:"<<top;
                cout<<"\nAttendence Percentage"<<attper;
        }
};

void main()
{
        clrscr();
        result r;

        cout<<"\n\t\t\t RESULT INFORMATION";
        r.getdata();                    //Calling Base class function using derived object
        r.getmarks();                   //Calling Base class function using derived object
        r.calculation();
        r.getattend();
        r.display();

        getch();

}
```

MULTIPATH INHERITANCE

- When a class is derived from two or more classes, which are derived from the same base class such type of inheritance is known as **multipath inheritance.**

Consider the following example:

```
class A1
{
 protected:
 int a1;
};

class A2 : public A1
{……..};

class A3: public  A1
{……..};

class A4: public A2,A3
{……..};
```



Ambiguity in classes

In the given example, class A2 and A3 are derived from the same base class i.e., class A1 (hierarchical inheritance). The classes A2 and A3 both can access variable a1 of class A1. The class A4 is derived from class A2 and class A3 by multiple inheritances. If we try to access the variable a1 of class A, the compiler shows error messages.

VIRTUAL BASE CLASSES

- To overcome the ambiguity occurred due to multipath inheritance, C++ provides the keyword virtual. The keyword virtual declares the specified classes virtual. The example given below illustrates the virtual classes.

```
class A1
{
 protected:
 int a1;
};

class A2 : virtual public A1    // virtual class declaration
{……};

class A3: virtual public  A1    // virtual class declaration
{……};

class A4: public A2,A3
{…..};
```

- When classes are declared as virtual, the compiler takes necessary precaution to avoid duplication of member variables. Thus, we make a class virtual if it is a base class that has been used by more than one derived class as their base class.

ADVANTAGES

- Increase Reusability
- Save time and money
- Reduces frustration
- Increase the reliability of the code
- To add some enhancements to the base class

DISADVANTAGES

- Though object-oriented programming is frequently propagandized as an answer for complicated projects, inappropriate use of inheritance makes a program more complicated.
- Invoking member functions using objects create more compiler overheads.
- In class hierarchy various data elements remain unused, the memory allocated to them is not utilized.

10. EXPLAIN IN DETAIL ABOUT CONSTRUCTORS, DESTRUCTORS AND INHERITANCE

- The constructors are used to initialize member variables of the object and the destructor is used to destroy the object. The compiler automatically invokes constructors and destructors. The derived class does not require a constructor if the base class contains zero-argument constructor.

- In case the base class has parameterized constructor then it is essential for the derived class to have a constructor. The derived class constructor passes arguments to the base class constructor. In inheritance, normally derived classes are used to declare objects. Hence, it is necessary to define constructor in the derived class. When an object of a derived class is declared, the constructors of base and derived classes are executed.

ABSTRACT CLASSES

- When a class is not used for creating objects it is called as abstract class. The abstract class can act as a base class only. It is a lay out abstraction in a program and it allows a base on which several levels of inheritance can be created. The base classes act as foundation of class hierarchy.
- An abstract class is developed only to act as a base class and to inherit and no objects of these classes are declared. An abstract class gives a skeleton or structure, using which other classes are shaped.
- The abstract class is central and generally present at the starting of the hierarchy. The hierarchy of classes means chain or groups of classes being involved with one another. In the last program, class *A* is an abstract class because no instance (object) of class A is declared.
-

QUALIFIER CLASSES AND INHERITANCE

- The following program explains the behaviour of qualifier classes and the classes declared within them with inheritance.

Example program to create derived class from the qualifier class.

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

```
# include <conio.h>


class A
{
    public:
    int x;
    A ( ) {}


        class B
        {

          public:
          int y;
             B( ) {}
    };


};   class C  : public A,A::B
    {
          public:
          int z;


    void show( )
    {
        cout <<endl<<"x = "<<x <<" y ="<<y<<" z = "<<z;
    }


        C (int j,int k, int l)
          {
             x=j;
             y=k;
             z=l;
          }
    };


void main( )
{
  clrscr( );
  C c(4,7,1);
  c.show( );
```

}

COMMON CONSTRUCTOR

- When a class is declared, constructor is also declared inside the class in order to initialize data members. It is not possible to use a single constructor for more classes. Every class has its own constructor and destructor with the same name as class.
- When a class is derived from another then it is possible to define a constructor in derived class and data members of both base and derived classes can be initialized. It is not essential to declare constructor in a base class. Thus, the constructor of the derived class works for its base class and such constructors are called as **common constructors.**

10. EXPLAIN IN DETAIL ABOUT POINTERS AND INHERITANCE

- The private and public member variables of a class are stored in successive memory locations. A pointer to public member variable gives us access to private member variables.

- The same is true for derived class. The member variables of base class and derived class are also stored in successive memory locations. The following program explains the mechanism of accessing private data members of the base class using the address of public member variable of derived class using pointer. Here, no member functions are used.

Example program to access private member variables of base class using pointers.

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


class A
{
 private:
 int x;
 int y;


 public:


 A( ) {
   x=1;
   y=2;
     }
};
class B : private A
{
public:
int z;
```

```
B( ) { z=3; }
};


void main( )
{
 clrscr( );
B b;   // object declaration
int *p;// pointer declaration
p=&b.z; // address of public member variabe is stored in pointer

cout<<endl<<" Address of z : "<<(unsigned)p <<" "<<"Value of z :"<<*p;
p--;  // points to previous location
cout<<endl<<" Address of y : "<<(unsigned)p <<" "<<"Value of y :"<<*p;
p--;
cout<<endl<<" Address of x : "<<(unsigned)p <<" "<<"Value of x :"<<*p;
}
```

OVERLOADING MEMBER FUNCTION

- The derived class can have the same function name as base class member function. An object of the derived class invokes member functions of the derived class even if the same function is present in the base class.

Example program to overload member function in base and derived class.

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


class B
{
  public:


  void show( )
  {
     cout <<"\n In base class function ";
  }
};


class D : public B
{

  public:
```

```
 void show( )
 {
   cout <<"\n In derived class function";
 }


};

int main( )
{
 clrscr( );
 B b;          // b is object of base class
 D d;          // d is object of derived class


 b.show( );     // Invokes Base class function
 d.show( );     // Invokes Derived class function
 d.B::show( );  // Invokes Base class function
 return 0;
}
```

11. WRITE A C++ PROGRAM HANDLING THE FOLLOWING DETAILS FOR STUDENTS AND STAFF USING INHERITANCE.

Student details: Name, address and percentage of marks
Staff details : Name, address and salary
Create appropriate base and derived classes. Input the details and output them

```
#include<iostream>
#include<conio>
using namespace std;
class Student                                   //Base class
{
   private:
        char *Name;
        char *Address;
        int marks;
  public:
        void ReadData( )
        {
           cout<<"Name:";
           cin>>Name;
           cout<<"Address:";
           cin>>Address;
           cout<<"marks:";
           cin>>marks;
```

```
        }
        void DisplayData( )
        {
          cout<<"Name:"<<Name<<endl;
           cout<<"Address:"<<Address<<endl;
           cout<<"marks:"<<marks<<endl;
        }
};
class staff : public Student                    //Derived class and public derivation
{
    private:
        int Salary;
    public:
        void ReadData( )
        {
   Student: :ReadData( );                       //Calling base class member function
                cout<<"Salary:";
                cin>>Salary;
        }
        void DisplayData( )
        {
          Person::DisplayData( );               //Calling base class member function
                cout<<"Salary:"<<salary<<endl;
        }
};
void main( )
{
    Staff S1;
    S1.ReadData( );                             //Calling base class member function
    S1.DisplayData( );                          //Calling base class member function
}
```

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

SUBJECT NAME: OBJECT ORIENTED PROGRAMMING AND DESIGN SUBJECT CODE: CST33

UNIT III

Pointers and arrays: Pointer to class and object – pointer to derived classes and base classes – accessing private members with pointers – address of object and void pointers –characteristics of arrays – array of classes.

Memory: Memory models – The new and delete operators – Heap consumption – Overloading new and delete operators – Execution sequence of constructors and destructors – specifying address of an object – dynamic objects.

Binding, Polymorphism and Virtual Functions: Binding in C++ – Pointer to derived class objects – virtual functions – Array of pointers – Abstract classes – Virtual functions in derived classes – constructors and virtual functions – virtual destructors – destructors and virtual functions. Strings - Declaring and initializing string objects – relational operators – Handling string objects – String attributes – Accessing elements of strings – comparing and exchanging and Miscellaneous functions

2 MARKS

1. What is pointer?

- Pointer variable stores the memory address of any type of variable.
- The pointer variable and normal variable should be of the same type.
- The pointer is denoted by (*) asterisk symbol.

  Pointer variables can be declared as below:

Example

```
int *x;
float *f;
char *y;
```

- In the first statement 'x' is an integer pointer and it informs to the compiler that it holds the address of any integer variable. In the same way, 'f' is a float pointer that

stores the address of any float variable and 'y' is a character pointer that stores the address of any character variable.

2. Define deference operator?

- The **indirection operator** (*) is also called the **deference operator.** When a pointer is dereferenced, the value at that address stored by the pointer is retrieved.
- Normal variable provides direct access to their own values whereas a pointer indirectly accesses the value of a variable to which it points.
- The **indirection operator** (*) is used in two distinct ways with pointers, declaration and deference.

3. How to assign pointers to class and object?

- Pointer is a variable that holds the address of another data variable. The variable may be of any data type i.e., int, float or double.
- In the same way, we can also define pointer to class.
- Here, starting address of the member variables can be accessed. Such pointers are called class pointers.

Example:

```
class book
{
        char name [25];
        char author [25];
        int pages;
};
```

a) class book *ptr;

or

a) struct book *ptr;

- In the above example, *ptr is pointer to class book. Both the statements (a) and (b) are valid. The syntax for using pointer with member is given below.
  1) ptr->name 2) ptr->author 3) ptr->pages.
- By executing these three statements, starting address of each member can be estimated.

4. What is void pointer?

- Pointers can also be declared as void type.
- Void pointers cannot be dereferenced without explicit type conversion. This is because being void the compiler cannot determine the size of the object that the pointer points to.
- Though void pointer declaration is possible, void variable declaration is not allowed.
- Thus, the declaration void p will display an error message "Size of 'p' is unknown or zero" after compilation.
- It is not possible to declare void variables like pointers.

5. What is wild pointer?

- When pointer points to an unallocated memory location or to data value whose memory is deallocated, such a pointer is called as wild pointer.
- The wild pointer generates garbage memory location and pendent reference.
- When a pointer pointing to a memory location gets vanished, the memory turns into garbage memory. It indicates that memory location exists but pointer is destroyed.
- This happens when memory is not de-allocated explicitly.

6. Why pointers become wild pointers?

The pointer becomes wild pointer due to the following reasons:

- Pointer declared but not initialized
- Pointer alteration
- Accessing destroyed data

7. What is array?

Array is a collection of elements of similar data types in which each element is unique and located in separate memory locations.

8. How to declare and Initialize an array?

Declaration of an array is shown below:

```
int a [5];
```

- It tells to the compiler that 'a' is an integer type of array and it must store five integers.

- The compiler reserves two bytes of memory for each integer array element.
- In the same way array of different data types are declared as below:

Array Declaration

```
char ch [10];
float real [10];
long num [5];
```

9. How to initialize an array?

The array initialization is done as given below:

```
int a [5]={1,2,3,4,5};
```

Here, five elements are stored in an array 'a'.

- The array elements are stored sequentially in separate locations.
- Then question arises how to call individually to each element from this bunch of integer elements.
- Reading of array elements begins from zero.

10. What is array of classes?

- Array is a collection of similar data types.
- In the same way, we can also define array of classes.
- In such type of array, every element is of class type.

Array of class objects can be declared as shown below:

```
class stud
{
 public:
 char name [12];      // class declaration
 int rollno;
 char grade[2];
};

class stud st[3];      // declaration of array of class objects
```

- In the above example, st[3] is an array of three elements containing three objects of class stud.
- Each element of st[3] has its own set class member variables i.e., char name[12], int rollno and char grade[2].

11. List the characteristic of an array?

- The declaration int a[5] is nothing but creation of five variables of integer type in memory. Instead of declaring five variables for five values, the programmer can define them in an array.
- All the elements of an array share the same name, and they are distinguished from one another with the help of element number.
- The element number in an array plays major role in calling each element.
- Any particular element of an array can be modified separately without disturbing other elements.

```
int a [5]={1,2,3,4,8};
```

- If the programmer needs to replace 8 with 10, he/she is not required to change all other numbers except 8. To carry out this task the statement a[4]=10 can be used. Here the other three elements are not disturbed.

12. Define the memory models?

- The memory model sets the supportable size of code and data areas.
- Before compiling and linking the source code, we need to specify the appropriate memory model.
- Using memory models, we can set the size limits of the data and code. C/C++ programs always use different segments for code and data.
- The memory model you opt decides the default method of memory addressing.
- The default memory model is small.

13. What is the use of new operator?

- The new operator not only creates an object but also allocates memory.
- The new operator allocates correct amount of memory from the heap that is also called as a free store.

14. What is the use of delete operator?

- The object created and memory allocated by using new operator should be deleted by the delete operator otherwise such mismatch operations may corrupt the heap or may crash the system.
- According to ANSI standard, it is a valid outcome for this invalid operation and the compiler should have routines to handle such errors.

15. What is heap consumption?

- The heap is used to allocate memory during program execution i.e., run-time.
- In assembly language, there is no such thing as a heap.
- In various ways, C/C++ has a better programming environment than assembly language. However, a cost has to be paid to use either C or C++.
- The cost is separation from machine. We cannot use memory anywhere; we need to ask for it. The memory from which we receive is called the heap.

16. Write about overloading new and delete operators?

- In C++ any time when we are concerned with memory allocation and deallocation, the new and delete operators are used.
- These operators are invoked from compiler's library functions. These operators are part of C++ language and are very effective.
- Like other operators the new and delete operators are overloaded.

17. Define address of an object?

- The compiler assigns address to objects created. The programmer has no power over address of the object.
- However, it is possible to create object at memory location given by the program explicitly. The address specified must be big enough to hold the object.
- The object created in this way must be destroyed by invoking destructor explicitly.

18. What is dynamic object?

- C++ allocates memory and initializes the member variables.
- An object can be created at run-time. Such object is called as dynamic object.

- The construction and destruction of dynamic object is explicitly done by the programmer.
- The dynamic objects can be created and destroyed by the programmer.
- The operator new and delete are used to allocate and deallocate memory to such objects.

19. What is binding in C++?

- The information pertaining to various overloaded member functions is to be given to the compiler while compiling. This is called as early binding or static binding.
- In C++ function can be bound either at compile time or run time.
- Deciding a function call at compiler time is called compile time or early or static binding.
- Deciding function call at runtime is called as runtime or late or dynamic binding.
- Dynamic binding permits suspension of the decision of choosing suitable member functions until run-time

20. Write about pointers to derived class objects

- Inheritance provides hierarchical organization of classes. It also provides hierarchical relationship between two objects and indicates the shared properties between them.
- All derived classes inherit properties from the common base class.
- Pointers can be declared to point base or derived classes. Pointers to object of base class are type compatible with pointers to object of derived class.
- A base class pointer can also point to objects of base and derived classes.
- In other words, a pointer to base class object can point to objects of derived classes whereas a pointer to derived class object cannot point to objects of base class object.

21. What is virtual function?

Virtual function or **virtual method** is a function or method whose behavior can be overridden within an inheriting class by a function with the same signature.

22. What are the rules of virtual function?

- The virtual functions should not be static and must be member of a class.

- A virtual function may be declared as friend for another class. Object pointer can access virtual functions.
- Constructors cannot be declared as virtual, but destructors can be declared as virtual.
- The virtual function must be defined in public section of the class. It is also possible to define the virtual function outside the class.
- In such a case, the declaration is done inside the class and definition is done outside the class. The virtual keyword is used in the declaration and not in function declarator.
- It is also possible to return a value from virtual function like other functions.
- The prototype of virtual functions in base and derived classes should be exactly the same. In case of mismatch, the compiler neglects the virtual function mechanism and treats them as overloaded functions

23. What is array of pointers?

- Polymorphism refers to late or dynamic binding i.e., selection of an entity is decided at run-time.
- In class hierarchy, same method names can be defined that perform different tasks, and then the selection of appropriate method is done using dynamic binding.
- Dynamic binding is associated with object pointers.
- Thus, address of different objects can be stored in an array to invoke function dynamically.

24. What is abstract class?

- Abstract classes are like skeleton upon which new classes are designed to assemble well-designed class hierarchy.
- The set of well-tested abstract classes can be used and the programmer only extends them.
- Abstract classes containing virtual function can be used as help in program debugging.
- When various programmers work on the same project, it is necessary to create a common abstract base class for them.
- The programmers are restricted to create new base classes.

25. Define virtual function in derived class?

- When functions are declared virtual in base classes, it is mandatory to redefine virtual functions in the derived class.
- The compiler creates VTABLES for the derived classes and stores address of function in it.
- In case the virtual function is not redefined in the derived class, the VTABLE of derived class contains address of base class virtual function.
- Thus, the VTABLE contains address of all functions.

26. Define pure virtual function?

- The member functions of base classes are rarely used for doing any operation; such functions are called as **do-nothing functions, dummy functions,** or **pure virtual functions.**
- The do-nothing function or pure functions are always virtual functions. Normally pure virtual functions are defined with null-body. This is so because derived classes should be able to override them.
- Any normal function cannot be declared as pure function. After declaration of pure function in a class, the class becomes abstract class.
- It cannot be used to declare any object. Any attempt to declare an object will result in an error "cannot create instance of abstract class".

27. What is virtual destructor?

- Destructors can be declared as virtual while constructor cannot be virtual, since it requires information about the accurate type of the object in order to construct it properly.
- The virtual destructors are implemented in the way like virtual functions. In constructors and destructors pecking order (hierarchy) of base and derived classes is constructed.
- Destructors of derived and base classes are called when a derived class object addressed by the base class pointer is deleted.
- For example, a derived class object is constructed using new operator. The base class pointer object holds the address of the derived object.

- When the base class pointer is destroyed using delete operator, the destructor of base and derived classes is executed.

28. What is string?

- A string is nothing but a sequence of characters.
- String can contain small and capital letters, numbers and symbols.
- String is an array of character type.
- Each element of string occupies a byte in the memory.
- Every string is terminated by a null character. The last character of such a string is null ('\0') character and the compiler identifies the null character during execution of the program.
- In a few cases, the null character must be specified explicitly. The null character is represented by '\0', which is the last character of a string. It's ASCII and Hex values are zero. The string is stored in the memory as follows:

```
char country[6] ="INDIA" ; ; // Declaration of an array 'country'.
```

- Here, text "INDIA" is assigned to array country[6]. The text is enclosed within double quotation mark.

29. How to declare and initialize string objects?

- In C, we declare strings as given below: char text[10]; whereas in C++ string is declared as an object.
- The string object declaration and initialization can be done at once using constructor in the string class.

The constructors of the string class are described in table

Constructors	**Meaning**
String ( );	Produces an empty string
String (const char *text);	Produces a string object from a null ended string

String (const string & text); Produces a string object from other string objects

We can declare and initialize string objects as follows:

string text; Declaration of string objects

string text("C++"); // Using constructor without argument

30. What are relational operators?

- These operators can be used with string objects for assignment, comparison etc. The following program illustrates the use of relational operators with string objects.

```
# include <iostream>
# include <string>
using namespace std;
int main( )
{
        string s1("OOP");
        string s2("OOP");
        if (s1==s2)
        cout <<"\n Both the strings are identical";
        else
        cout <<"\n Both the strings are different";
        return 0;
}
```

OUTPUT

Both the strings are identical

31. Name some of the String attributes?

- The various attributes of the string such as size, length, capacity etc. can be obtained using member functions.
- The size of the string indicates the total number of elements currently stored in the string.
- The capacity means the total number of elements that can be stored in string.
- The maximum size means the largest valid size of the string supported by the system.

32. How to access elements of string?

- It is possible to access particular word or single character of string with the help of member functions of string class.
- **at( ):**This function is used to access individual character. It requires one argument that indicates the element number.
- It is used in the given format s1.at(5); Here s1 is a string object and 5 indicates 5th element that is to be accessed.

33. What is formatted data?

- Formatting means representation of data with different settings as per the requirement of the user.
- The various settings that can be done are number format, field width, decimal points etc.
- If the user needs to accept or display data in hexa-decimal format, manipulators with I/O functions are used.
- The data obtained or represented with these manipulators are known as formatted data.

34. How to compare two strings?

- The string class contains functions, which allows the programmer to compare the strings.

  The related functions are illustrated below with suitable examples:

  compare( ): The member function compare( ) is used to compare two string or sub-strings. It returns 0 when the two strings are same, otherwise returns a non-zero value. It is used in the following formats:

  s1.compare(s2);//Here s1 and s2 are two string objects.
  s1.compare (0,4,s2,0,4);//Here s1 and s2 are two string objects. The integer number indicates the sub-string of both the strings that are to be compared.

35. Define assign function?

- This function is used to assign a string wholly/ partly to other string object. It is used in the following format:

  s2.assign(s1); //Here s2 and s1 are two string objects. The contents of string s1 are assigned to s2.

  s2.assign(s1,0,5);//In the above format, element from 0 to 5 are assigned to object s2

36. Define String Handling Objects

- The member functions insert( ), replace( ), erase( ) and append( ) are used to modify the string contents. The following program illustrates the use of these functions:

- **insert( )**

  - This member function is used to insert specified strings into another string at a given location. It is used in the following form:

    s1.insert(3,s2);

  Here, s1 and s2 are string objects. The first argument is used as location number for calling string where the second string is to be inserted.

37. Explain pointer to members

- It is possible to obtain address of member variables and store it to a pointer. The following programs explain how to obtain address and accessing member variables with pointers

38. How pointers can be declared

Pointer variables can be declared as below:

Example

```
int *x;
float *f;
char *y
```

9. Explain wild pointers

- Pointers are used to store memory addresses. An improper use of pointers creates many errors in the program. Hence, pointers should be handled cautiously.

  When pointer points to an unallocated memory location or to data value whose memory is deallocated, such a pointer is called as wild pointer

40. Explain Array Declaration and Initialization

- Declaration of an array is shown below:

  int a [5];

- It tells to the compiler that 'a' is an integer type of array and it must store five integers. The compiler reserves two bytes of memory for each integer array element. In the same way array of different data types are declared as below:

  Array Declaration

  char ch [10];

  float real [10];

  long num [5];

- The array initialization is done as given below:

  Array Initialization

  int a [5]={1,2,3,4,5};

41. What are the input and output operators used in C++?

- The identifier cin is used for input operation. The input operator used is >>, which is known as the extraction or get from operator.

  syntax : cin >> n1;

- The identifier cout is used for output operation. The input operator used is <<, which is known as the insertion or put to operator.

  syntax: cout << ―C++ is better than C‖;

42. list out the characteristics of arrays

- The declaration int a[5] is nothing but creation of five variables of integer type in memory. Instead of declaring five variables for five values, the programmer can define them in an array.
- All the elements of an array share the same name, and they are distinguished from one another with the help of element number.
- The element number in an array plays major role in calling each element.
- Any particular element of an array can be modified separately without disturbing other elements. int a[5]={1,2,3,45}
- If the programmer needs to replace 8 with 10, he/she is not required to change all other numbers except 8. To carry out this task the statement a[4]=10 can be used. Here the other three elements are not disturbed.

43. Explain initialization of arrays using functions

- The programmers always initialize the array using statement like int d[ ]={1,2,3,4,5}. Instead of this the function can also be directly called to initialize the array. The program given below illustrates this point.

44. what are the memory models in c++ language?

- Memory is one of the critical resources of a computer system. One must be conscious about the memory modules while an application is being developed.
- The memory model sets the supportable size of code and data areas. Before compiling and linking the source code, we need to specify the appropriate memory model.
- Using memory models, we can set the size limits of the data and code. C/C++ programs always use different segments for code and data.
- The memory model you opt decides the default method of memory addressing. The default memory model is small.

45.Explain in detail about address of an object and dynamic objects?

- The compiler assigns address to objects created. The programmer has no power over address of the object. However, it is possible to create object at memory location given by the program explicitly.
- The address specified must be big enough to hold the object. The object created in this way must be destroyed by invoking destructor explicitly.

46. Explain dynamic objects

- C++ supports dynamic memory allocation. C++ allocates memory and initializes the member variables.
- An object can be created at run-time. Such object is called as dynamic object.
- The construction and destruction of dynamic object is explicitly done by the programmer.

47. What is early binding?

- The reason for the incorrect output is that the call of the function area() is being set once by the compiler as the version defined in the base class. This is called **static resolution** of the function call, or **static linkage** - the function call is fixed before the program is executed.
- This is also sometimes called **early binding** because the area() function is set during the compilation of the program.
- But now, let's make a slight modification in our program and precede the declaration of area() in the Shape class with the keyword **virtual** so that it looks like this:

48. What is Virtual Function?

- A **virtual** function is a function in a base class that is declared using the keyword **virtual**.
- Defining in a base class a virtual function, with another version in a derived class, signals to the compiler that we don't want static linkage for this function.
- What we do want is the selection of the function to be called at any given point in the program to be based on the kind of object for which it is called. This sort of operation is referred to as **dynamic linkage**, or **late binding**.
-

49. List out the key concepts of object – oriented programming.

There are several fundamental or key concepts in object-oriented programming. Some of them are

- Objects.
- Classes.
- Data abstraction and Encapsulation.
- Inheritance.
- Polymorphism.

- Dynamic binding.
- Message passing.

50. What is Pure Virtual Functions?

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

51. What is abstract classes?

- Abstract classes are like skeleton upon which new classes are designed to assemble well-designed class hierarchy.
- The set of well-tested abstract classes can be used and the programmer only extends them.
- Abstract classes containing virtual function can be used as help in program debugging

52. Describe the virtual functions role with constructors and destructors?

- It is possible to invoke a virtual function using a constructor. Constructor makes the virtual mechanism illegal.
- When a virtual function is invoked through a constructor, the base class virtual function will not be called and instead of this the member function of the same class is invoked.

53. What is virtual destructors?

- We know how virtual functions are declared. Likewise, destructors can be declared as virtual. The constructor cannot be virtual, since it requires information about the accurate type of the object in order to construct it properly.
- The virtual destructors are implemented in the way like virtual functions. In constructors and destructors pecking order (hierarchy) of base and derived classes is constructed.

54. What are string attributes

- The various attributes of the string such as size, length, capacity etc. can be obtained using member functions.

- The size of the string indicates the total number of elements currently stored in the string.
- The capacity means the total number of elements that can be stored in string.
- The maximum size means the largest valid size of the string supported by the system.

55. What are the features of Object Oriented Programming?

- Programs are divided into objects
- Emphasis is on data rather than procedure
- Data Structures are designed such that they characterize the objects
- Functions that operate on the data of an object are tied together
- Data is hidden and cannot be accessed by external functions
- Objects may communicate with each other through functions
- New data and functions can easily be added whenever necessary
- Follows bottom-up approach

56.What is the use of new operator?

- The new operator not only creates an object but also allocates memory.
- The new operator allocates correct amount of memory from the heap that is also called as a free store.

57.What is the use of delete operator?

- The object created and memory allocated by using new operator should be deleted by the delete operator otherwise such mismatch operations may corrupt the heap or may crash the system.
- According to ANSI standard, it is a valid outcome for this invalid operation and the compiler should have routines to handle such errors.

58. What is heap consumption?

- The heap is used to allocate memory during program execution i.e., run-time.
- In assembly language, there is no such thing as a heap.

- In various ways, C/C++ has a better programming environment than assembly language. However, a cost has to be paid to use either C or C++.
- The cost is separation from machine. We cannot use memory anywhere; we need to ask for it. The memory from which we receive is called the heap.

59. Write about overloading new and delete operators?

- In C++ any time when we are concerned with memory allocation and deallocation, the new and delete operators are used.
- These operators are invoked from compiler's library functions. These operators are part of C++ language and are very effective.
- Like other operators the new and delete operators are overloaded.

60. Explain in detail about address of an object and dynamic objects?

- The compiler assigns address to objects created. The programmer has no power over address of the object. However, it is possible to create object at memory location given by the program explicitly.
- The address specified must be big enough to hold the object. The object created in this way must be destroyed by invoking destructor explicitly.

11 MARKS

1. WRITE IN DETAIL ABOUT POINTERS WITH CLASSES AND OBJECTS?

- Pointers can make programs quicker, straightforward and memory efficient. C/C++ gives more importance to pointers. Hence, it is important to know the operation and applications of pointers. Pointers are used as tool in C/C++.
- Like C, in C++ variables are used to hold data values during program execution. Every variable when declared occupies certain memory locations. It is possible to access and display the address of memory location of variables using '&' operator.
- Memory is arranged in series of bytes. These series of bytes are numbered from zero onwards. The number specified to a cell is known as memory address.
- Pointer variable stores the memory address of any type of variable.
- The pointer variable and normal variable should be of the same type. The pointer is denoted by (*) asterisk symbol.

POINTER DECLARATION

Pointer variables can be declared as below:

Example

```
int *x;
float *f;
char *y;
```

(1) In the first statement 'x' is an integer pointer and it informs to the compiler that it holds the address of any integer variable. In the same way, 'f' is a float pointer that stores the address of any float variable and 'y' is a character pointer that stores the address of any character variable.

(2) The **indirection operator** (*) is also called the **deference operator.** When a pointer is dereferenced, the value at that address stored by the pointer is retrieved.

(3) Normal variable provides direct access to their own values whereas a pointer indirectly accesses the value of a variable to which it points.

(4) The **indirection operator** (*) is used in two distinct ways with pointers, declaration and deference.

VOID POINTERS

- Pointers can also be declared as void type. Void pointers cannot be dereferenced without explicit type conversion. This is because being void the compiler cannot determine the size of the object that the pointer points to.
- Though void pointer declaration is possible, void variable declaration is not allowed. Thus, the declaration void p will display an error message "Size of 'p' is unknown or zero" after compilation.
- It is not possible to declare void variables like pointers. Pointers point to an existing entity. A void pointer can point to any type of variable with proper typecasting. The size of void pointer displayed will be two.
- When pointer is declared as void, two bytes are allocated to it. Later using typecasting, number of bytes can be allocated or de-allocated.
- Void variables cannot be declared because memory is not allocated to them and there is no place to store the address. Therefore, void variables cannot actually serve the job they are made for.

WILD POINTERS

- Pointers are used to store memory addresses. An improper use of pointers creates many errors in the program. Hence, pointers should be handled cautiously.
- When pointer points to an unallocated memory location or to data value whose memory is deallocated, such a pointer is called as wild pointer.
- The wild pointer generates garbage memory location and pendent reference. When a pointer pointing to a memory location gets vanished, the memory turns into garbage memory.
- It indicates that memory location exists but pointer is destroyed. This happens when memory is not de-allocated explicitly.

The pointer becomes wild pointer due to the following reasons:

(1) Pointer declared but not initialized

(2) Pointer alteration

(3) Accessing destroyed data

POINTER TO CLASS

- We know that pointer is a variable that holds the address of another data variable. The variable may be of any data type i.e., int, float or double.
- In the same way, we can also define pointer to class. Here, starting address of the member variables can be accessed. Such pointers are called classpointers.

Example:

```
class book
{
        char name [25];
        char author [25];
        int pages;
};
```

a) class book *ptr;

or

b) struct book *ptr;

In the above example, *ptr is pointer to class book. Both the statements (a) and (b) are valid.

The syntax for using pointer with member is given below.

1) ptr->name 2) ptr->author 3) ptr->pages.

By executing these three statements, starting address of each member can be estimated.

POINTER TO OBJECT

- Like variables, objects also have an address. A pointer can point to specified object. The following program illustrates this.

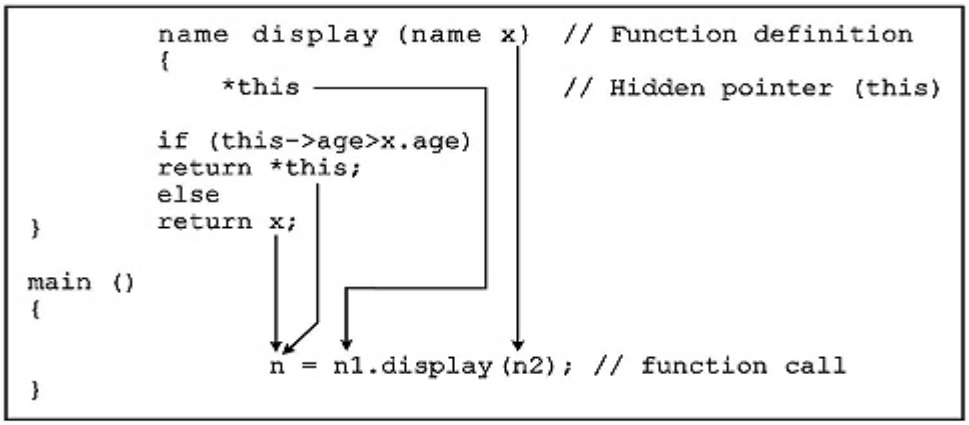
Write a program to declare an object and pointer to the class. Invoke member functions using pointer.

```
# include <iostream.h>
# include <conio.h>
class Bill
{
    int   qty;
    float price;
    float amount;

    public :

    void getdata (int a, float b, float c)
    {
     qty=a;
     price=b;
     amount=c;
    }
```

```
    void show( )
    {
     cout <<"Quantity   : " <<qty <<"\n";
     cout <<"Price      : " <<price <<"\n";
     cout <<"Amount     : " <<amount <<"\n";
    }
};
int main( )
{
    clrscr( );
    Bill s;
    Bill *ptr =&s;
    ptr->getdata(45,10.25,45*10.25);
    (*ptr).show( );
    return 0;
}
```

THE this **POINTER**

- Use of this pointer is now outdated. The objects are used to invoke the non-static member functions of the class.
- For example, if p is an object of classP and get( ) is a member function of P, the statement p.get( ) is used to call the function. The statement p.get( ) operates on p. In the same way ifptr is a pointer to P object, the function called ptr->get( ) operates on *ptr.
- However, the question is, how does the member function get( ) understand which p it is functioning on? C++ compiler provides get( ) with a pointer to p called this. The pointer this is transferred as an unseen parameter in all calls to non-static member functions.
- The keyword this is a local variable that is always present in the body of any non-static member function



2. EXPLAIN IN DETAIL ABOUT POINTER TO DERIVED CLASSES AND BASE CLASSES?

- It is possible to declare a pointer which points to the base class as well as the derived class. One pointer can point to different classes. For example, X is a base class and Y is a derived class. The pointer pointing to X can also point to Y.

Program to declare a pointer to the base class and access the member variable of base and derived class.

```
// Pointer to base object //

# include <iostream.h>
# include <conio.h>
    class A
    {
        public :
        int b;
        void display( )
        {
        cout <<"b = " <<b <<"\n";
    }
    };

class B : public A
{
    public :
    int d;
```

```
    void display( )
    {
    cout <<"b= " <<b <<"\n" <<" d="<<d <<"\n";
    }

};

main( )
 {
        clrscr( );
        A *cp;
        A base;
         cp=&base;
         cp->b=100;
        //  cp->d=200; Not Accessible
        cout <<"\n cp points to the base object \n";
        cp->display( );

B b;
cout <<"\n cp points to the derived class \n";
cp=&b;
 cp->b=150;
// cp->d=300;  Not accessible
 cp->display( );
 return 0; }
```

POINTER TO MEMBERS

- It is possible to obtain address of member variables and store it to a pointer. The following programs explain how to obtain address and accessing member variables with pointers.

Program to initialize and display the contents of the structure using dot (.) **and arrow** (->) **operator.**

```
# include <iostream.h>
# include <conio.h>
int main ( )
{
        clrscr( );
        struct c
        {
                char *name;
        };

        c b, *bp;
        bp->name=" CPP";
        b.name="C &";
        cout<<b.name;
        cout<<bp->name;
        return 0;
}
```

OUTPUT

C & CPP

Explanation: In the above program, structure c is declared. The variable b is object and bp is a pointer object to structure c. The elements are initialized and displayed using dot (.) and arrow (->) operators. These are the traditional operators which are commonly used in C. C++ allows two new operators .* and ->* to carry the same task. These C++ operators are recognized as pointer to member operators.

3. EXPLAIN IN DETAIL ABOUT ACCESSING PRIVATE MEMBERS WITH POINTERS WITH SAMPLE PROGRAM

- The public and private member variables are stored in successive memory locations.

The following program explains how private members can also be accessed using pointer.

Program to access private members like public members of the class using pointers.

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

   class A

   {
            private:
                  int j;
            public :
                  int x;
                  int y;
                  int z;
                  A ( )
                  {
                              j=20;
                  }
   };

 void main( )
 {
          clrscr( );
          A a;
          int *p;
          a.x=11;
          a.y=10;
          a.z=15;
          p=&a.x;
          p--;
          cout<<endl<<"j ="<<*p;
          p++;
          cout<<endl<<"x= "<<*p;
```

```
        p++;
        cout<<endl<<"y= "<<*p;
        p++;
        cout<<endl<<"z= "<<*p;
  }
```

OUTPUT

j =20

DIRECT ACCESS TO PRIVATE MEMBERS

- So far, we have initialized private members of the class using constructor or member function of the same class.
- In the last sub-topic we also learned how private members of the class can be accessed using pointers.
- In the same way, we can initialize private members and display them. Obtaining the address of any public member variable and using address of the object one can do this.
- The address of the object contains address of the first element. Programs concerning this are explained below:

Program to initialize the private members and display them without the use of member functions.

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

class A
{
  private :
  int x;
  int y;
};

 void main ( )
{
   clrscr( );
```

```
    A a;
    int *p=(int*)&a;
    *p=3;
     p++;
    *p=9;
     p--;
     cout <<endl <<"x= "<<*p;
     p++;
     cout <<endl<<"y= "<<*p;
}
```

ADDRESS OF OBJECT AND VOID POINTERS

- The size of object is equal to number of total member variables declared inside the class. The size of member function is not considered in object. The object itself contains address of first member variable.
- By obtaining the address we can access the member variables directly, no matter whether they are private or public. The address of object cannot be assigned to the pointer of the same type.
- This is because increase operation on pointers will set the address of object according to its size (size of object=size of total member variables). The address of object should be increased according to the basic data type.
- To overcome this situation a void pointer is useful.

3. EXPLAIN IN DETAIL ABOUT ARRAYS?

- Array is a collection of elements of similar data types in which each element is unique and located in separate memory locations.

Array Declaration and Initialization

Declaration of an array is shown below:

```
int a [5];
```

It tells to the compiler that 'a' is an integer type of array and it must store five integers. The compiler reserves two bytes of memory for each integer array element. In the same way array of different data types are declared as below:

Array Declaration

```
char ch [10];
float real [10];
long num [5];
```

The array initialization is done as given below:

Array Initialization

```
int a [5]={1,2,3,4,5};
```

Here, five elements are stored in an array 'a'. The array elements are stored sequentially in separate locations. Then question arises how to call individually to each element from this bunch of integer elements. Reading of array elements begins from zero.

CHARACTERISTICS OF ARRAYS

- The declaration int a[5] is nothing but creation of five variables of integer type in memory. Instead of declaring five variables for five values, the programmer can define them in an array.
- All the elements of an array share the same name, and they are distinguished from one another with the help of element number.
- The element number in an array plays major role in calling each element.
- Any particular element of an array can be modified separately without disturbing other elements. int a[5]={1,2,3,45}
- If the programmer needs to replace 8 with 10, he/she is not required to change all other numbers except 8. To carry out this task the statement a[4]=10 can be used. Here the other three elements are not disturbed.

INITIALIZATION OF ARRAYS USING FUNCTIONS

- The programmers always initialize the array using statement like int d[ ]={1,2,3,4,5}. Instead of this the function can also be directly called to initialize the array. The program given below illustrates this point.

Program to initialize an array using functions.

```
# include <stdio.h>
# include <conio.h>
main( )
{
int k,c( ),d[]={c( ),c( ),c( ),c( ),c( )};
clrscr( );
printf ("\n Array d[] elements are :");
for (k=0;k<5;k++)
printf ("%2d",d[k]);
return (NULL);
}

c( )
{
static int m,n;
m++;
printf ("\nEnter Number d[%d] : ",m);
scanf ("%d",&n);
return(n);
}
```

OUTPUT:

Enter Number d[1] : 4
Enter Number d[2] : 5
Enter Number d[3] : 6
Enter Number d[4] : 7

Enter Number d[5] : 8

5. WHAT ARE THE MEMORY MODELS IN C++ LANGUAGE?

- Memory is one of the critical resources of a computer system. One must be conscious about the memory modules while an application is being developed.
- The memory model sets the supportable size of code and data areas. Before compiling and linking the source code, we need to specify the appropriate memory model.
- Using memory models, we can set the size limits of the data and code. C/C++ programs always use different segments for code and data.
- The memory model you opt decides the default method of memory addressing. The default memory model is small.

Tiny

- Use the tiny model when memory is at an absolute premium. All four segment registers (CS, DS, ES, SS) are initialized with same address and all addressing is accomplished using 16 bits. Total memory capacity in this case is 64 K bytes.
- In short, memory capacity is abbreviated as KB. This means that the code, data, and stack must all fit within the same 64 KB segment.
- Programs are executed quickly if this model is selected. Near pointers are always used. Tiny model programs can be converted to .COM format.

THE new **AND** delete **OPERATORS**

Some points about new and delete operators

(1) The new operator not only creates an object but also allocates memory.

(2) The new operator allocates correct amount of memory from the heap that is also called as a free store.

(3) The object created and memory allocated by using new operator should be deleted by the delete operator otherwise such mismatch operations may corrupt the heap or may crash the system. According to ANSI standard, it is a valid outcome for this invalid operation and the compiler should have routines to handle such errors.

HEAP CONSUMPTION

- The heap is used to allocate memory during program execution i.e., run-time. In assembly language, there is no such thing as a heap. All memory is for programmer and he/she can use it directly.
- In various ways, C/C++ has a better programming environment than assembly language. However, a cost has to be paid to use either C or C++.
- The cost is separation from machine.
- We cannot use memory anywhere; we need to ask for it. The memory from which we receive is called the heap.
- The C/C++ compiler may place automatic variables on the stack. It may store static variables earlier than loading the program. The heap is the piece of memory that we allocate.
- Local variables are stored in the stack and code is in code space. The local variables are destroyed when a function returns.
- Global variables are stored in the data area. Global variables are accessible by all functions.
- The heap can be thought of as huge section of memory in which large number of memory locations is placed sequentially

6. EXPLAIN IN DETAIL ABOUT OVERLOADING NEW AND DELETE OPERATORS WITH SAMPLE PROGRAM?

- In C++ any time when we are concerned with memory allocation and deallocation, the new and delete operators are used.
- These operators are invoked from compiler's library functions. These operators are part of C++ language and are very effective.
- Like other operators the new and delete operators are overloaded. The program given next illustrates this.

Program to overload new **and** delete **operators.**

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

```
void main( )
{
 clrscr( );
 void warning( );
 void *operator new (size_t, int);
 void operator delete (void*);
 char *t= new (‘#’) char [10];
 cout <<endl<<"First allocation : p="<<(unsigned)long(t)<<endl;
 for (int k=0;k<10;k++)
 cout <<t[k];
 delete t;
 t=new (‘*’) char [64000u];
 delete t;
}

void warning( )
{
 cout <<"\n insufficient memory";
 exit(1);
}

void *operator new (size_t os, int setv)
{
 void *t;
 t=malloc(os);
 if (t==NULL)   warning( );
 memset(t,setv,os);
 return (t);
}

void operator delete(void *ss)
{
        free(ss);
```

}

7. EXPLAIN IN DETAIL ABOUT THE EXECUTION SEQUENCE OF CONSTRUCTORS AND DESTRUCTORS?

- Overloaded new and delete operators function within the class and are always static. We know that static function can be invoked without specifying object. Hence, this pointer is absent in the body of static functions.
- The compiler invokes the overloaded new operator and allocates memory before executing constructor.
- The compiler also invokes overloaded delete operator function and deallocates memory after execution of destructor.

The following program illustrates this concept.

Program to display the sequence of execution of constructors and destructors in classes when new **and** delete **operators are overloaded.**

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

class boy
{
 char name[10];
 public :
 void *operator new (size_t );
 void operator delete (void *q);
 boy( );
 ~boy( );
};

char limit[sizeof(boy)];
boy::boy( )
{
```

```
  cout <<endl<<"In Constructor";
}

boy::~boy( )
{
  cout <<endl<<"In Destructor";
}

void  *boy :: operator  new (size_t s )
{
  cout <<endl<<"In boy ::new operator";
  return limit;
}

void boy::operator delete (void *q)
{   cout <<endl<<"In boy ::delete operator";
}

void main( )
{
  clrscr( );
  boy *e1;
  e1 = new boy;
  delete e1;
}
```

8. EXPLAIN IN DETAIL ABOUT ADDRESS OF AN OBJECT AND DYNAMIC OBJECTS?

- The compiler assigns address to objects created. The programmer has no power over address of the object. However, it is possible to create object at memory location given by the program explicitly.
- The address specified must be big enough to hold the object. The object created in this way must be destroyed by invoking destructor explicitly.

The following program clears this concept.

Program to create object at given memory address.

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

class data
{
 int j;
 public:
 data ( int k)  { j=k; }
 ~data ( )       {  }


 void *operator new ( size_t, void *u)
 {
   return (data *)u;
 }

 void show( )   { cout<<"j="<<j;  }
};

void main( )
{
 clrscr( );
 void *add;
 add=(void*)0x420;
 data *t= new (add) data(10);
 cout<<"\n Address of object : "<<t;
 cout<<endl;
 t->show( );
 t->data::~data( );
}
```

DYNAMIC OBJECTS

- C++ supports dynamic memory allocation. C++ allocates memory and initializes the member variables.
- An object can be created at run-time. Such object is called as dynamic object.
- The construction and destruction of dynamic object is explicitly done by the programmer.
- The dynamic objects can be created and destroyed by the programmer.
- The operator new and delete are used to allocate and deallocate memory to such objects.

A dynamic object can be created using new operator as follows:

ptr= new classname;

- The new operator returns the address of object created and it is stored in the pointer ptr. The variable ptr is a pointer object of the same class.
- The member variables of object can be accessed using pointer and -> (arrow) operator. A dynamic object can be destroyed using delete operator

9. WRITE ABOUT POLYMORPHISM AND BINDING IN C++.

- The word **polymorphism** means having many forms. Typically, polymorphism occurs when there is a hierarchy of classes and they are related by inheritance.
- C++ polymorphism means that a call to a member function will cause a different function to be executed depending on the type of object that invokes the function.

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

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

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

```
    Shape( int a=0, int b=0)
    {
      width = a;
      height = b;
    }
    int area()
    {
      cout << "Parent class area :" <<endl;
      return 0;
    }
};
class Rectangle: public Shape{
  public:
    Rectangle( int a=0, int b=0):Shape(a, b) { }
    int area ()
    {
      cout << "Rectangle class area :" <<endl;
      return (width * height);
    }
};
class Triangle: public Shape{
  public:
    Triangle( int a=0, int b=0):Shape(a, b) { }
    int area ()
    {
      cout << "Triangle class area :" <<endl;
      return (width * height / 2);
    }
};
// Main function for the program
int main( )
{
  Shape *shape;
  Rectangle rec(10,7);
```

```
  Triangle  tri(10,5);

  // store the address of Rectangle
  shape = &rec;
  // call rectangle area.
  shape->area();

  // store the address of Triangle
  shape = &tri;
  // call triangle area.
  shape->area();
return 0;
}
```

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

```
Parent class area
```

- The reason for the incorrect output is that the call of the function area() is being set once by the compiler as the version defined in the base class. This is called **static resolution** of the function call, or **static linkage** - the function call is fixed before the program is executed.
- This is also sometimes called **early binding** because the area() function is set during the compilation of the program.
- But now, let's make a slight modification in our program and precede the declaration of area() in the Shape class with the keyword **virtual** so that it looks like this:

```
class Shape
{
  protected:
    int width, height;
  public:
    Shape( int a=0, int b=0)
    {
```

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

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

  Rectangle class area

  Triangle class area

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

  As you can see, each of the child classes has a separate implementation for the function area().

  This is how **polymorphism** is generally used. You have different classes with a function of the same name, and even the same parameters, but with different implementations.

Virtual Function:

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

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

Pure Virtual Functions:

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

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

```
class Shape
{
   protected:
         int width, height;
   public:
         Shape( int a=0, int b=0)
         {
               width = a;
               height = b;
         }
   // pure virtual function
   virtual int area() = 0;
};
```

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

10. EXPLAIN IN DETAIL ABOUT VIRTUAL FUNCTION WITH SAMPLE PROGRAM

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

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

RULES FOR VIRTUAL FUNCTIONS

- The virtual functions should not be static and must be member of a class.
- A virtual function may be declared as friend for another class. Object pointer can access virtual functions.
- Constructors cannot be declared as virtual, but destructors can be declared as virtual.
- The virtual function must be defined in public section of the class. It is also possible to define the virtual function outside the class. In such a case, the declaration is done inside the class and definition is done outside the class.
- The virtual keyword is used in the declaration and not in function declarator.
- It is also possible to return a value from virtual function like other functions.
- The prototype of virtual functions in base and derived classes should be exactly the same. In case of mismatch, the compiler neglects the virtual function mechanism and treats them as overloaded functions.

ARRAY OF POINTERS

- Polymorphism refers to late or dynamic binding i.e., selection of an entity is decided at run-time. In class hierarchy, same method names can be defined that perform different tasks, and then the selection of appropriate method is done using dynamic binding.
- Dynamic binding is associated with object pointers. Thus, address of different objects can be stored in an array to invoke function dynamically. The following program explains this concept.

Program to create an array of pointers. Invoke functions using array objects.

```
# include <iostream.h>
# include <constream.h>
class A
```

```
{
  public:

  virtual void show( )
  {
    cout <<"A\n";
  }
};

class B : public A
{
  public :
  void show( )
{ cout <<"B\n"; }
};

  class C : public A
    {
    public :
    void show( )
    {
      cout <<"C\n";
    }
};

class D : public A
  {
    public:
    void show( )
  {
    cout <<"D\n" ;
  }
};
```

```
class E : public A {
 public :
 void show( )
 {
  cout <<"E";
 }

};
 void main( )
 {
  clrscr( );
  A a;
  B b;
  C c;
  D d;
  E e;

  A *pa[] = {&a,&b,&c,&d,&e };

  for ( int j=0;j<5;j++)
  pa[j]->show( );
 }
```

Pure Virtual Functions:

- In practical applications, the member functions of base classes is rarely used for doing any operation; such functions are called as **do-nothing functions, dummy functions,** or **pure virtual functions.**
- The do-nothing function or pure functions are always virtual functions. Normally pure virtual functions are defined with null-body. This is so because derived classes should be able to override them.
- Any normal function cannot be declared as pure function. After declaration of pure function in a class, the class becomes abstract class. It cannot be used to declare any object.

- Any attempt to declare an object will result in an error "cannot create instance of abstract class".

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

```
class Shape
{
  protected:
    int width, height;
  public:
    Shape( int a=0, int b=0)
    {
      width = a;
      height = b;
    }
    // pure virtual function
    virtual int area() = 0;
};
```

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

ABSTRACT CLASSES

- Abstract classes are like skeleton upon which new classes are designed to assemble well-designed class hierarchy.
- The set of well-tested abstract classes can be used and the programmer only extends them.
- Abstract classes containing virtual function can be used as help in program debugging.
- When various programmers work on the same project, it is necessary to create a common abstract base class for them.
- The programmers are restricted to create new base classes.
- The development of such software can be demonstrated by creating a header file.

- The file abstract.h is an example of file containing abstract base class used for debugging purpose.

Contents of file abstract.h is as follows:

```
# include <iostream.h>
struct debug
{
   virtual void show ( )
   {
     cout<<"\n No function show ( ) defined for this class";
   }
};
```

WORKING OF VIRTUAL FUNCTIONS

Before learning mechanisms of virtual functions let's revise few points related to virtual functions.

- Binding means link between a function call and the real function that is executed when function is called.
- When compiler knows which function to call before execution, it is known as early binding.
- Dynamic binding means that the actual function invoked at run time depends on the address stored in the pointer. In this binding, link between function call and actual function is made during program execution.
- The keyword virtual prevents compiler to perform early binding. Binding is postponed until program execution.

VIRTUAL FUNCTIONS IN DERIVED CLASSES

- When functions are declared virtual in base classes, it is mandatory to redefine virtual functions in the derived class.
- The compiler creates VTABLES for the derived classes and stores address of function in it.

- In case the virtual function is not redefined in the derived class, the VTABLE of derived class contains address of base class virtual function.
- Thus, the VTABLE contains address of all functions.
- Thus, to some extent it is not possible that a function existing has its address not present in the VTABLE.

11. DESCRIBE THE VIRTUAL FUNCTIONS ROLE WITH CONSTRUCTORS AND DESTRUCTORS?

- It is possible to invoke a virtual function using a constructor. Constructor makes the virtual mechanism illegal.
- When a virtual function is invoked through a constructor, the base class virtual function will not be called and instead of this the member function of the same class is invoked.

Program to call virtual function through constructor.

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

class B
{
        int k;
        public :

                B ( int l) { k=l; }
                virtual void show ( ) { cout <<endl<<" k="<<k; }
};
class D: public B
{
        int h;
        public :

                D (int m, int n) : B (m)
        {
                h=n;
```

```
                B *b;
                b=this;
                b->show( );
        }

         void show ( )
        {
                cout <<endl<<" h="<<h;
        }
};

void main( )
{
        clrscr( );
        B b(4);
        D d(5,2);
}
```

VIRTUAL DESTRUCTORS

- We know how virtual functions are declared. Likewise, destructors can be declared as virtual. The constructor cannot be virtual, since it requires information about the accurate type of the object in order to construct it properly.
- The virtual destructors are implemented in the way like virtual functions. In constructors and destructors pecking order (hierarchy) of base and derived classes is constructed.
- Destructors of derived and base classes are called when a derived class object addressed by the base class pointer is deleted.
- For example, a derived class object is constructed using new operator. The base class pointer object holds the address of the derived object.
- When the base class pointer is destroyed using delete operator, the destructor of base and derived classes is executed

DESTRUCTORS AND VIRTUAL FUNCTIONS

- When a virtual function is invoked through a non-virtual member function, late binding is performed. When call to virtual function is made through the destructor, the redefined function of the same class is invoked.

Program to call virtual function using destructors.

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

class B
{
 public :

~ B ( ) { cout <<endl<<" in virtual destructor";}
virtual void joy( ) { cout <<endl<<"In joy of class B";}
};

class D : public B
{
 public :
~ D ( )
{
 B *p;
 p=this;
 p->joy( );

}
void joy( ) { cout <<endl<<" in joy( ) of class D"; }

};

void main( ) {
 clrscr( );
```

```
 D X;
}
```

OUTPUT

in joy() of classD in virtual destructor

12. WRITE IN DETAIL ABOUT DECLARING AND INITIALIZING STRING OBJECTS WITH SAMPLE PROGRAM?

- To make the string manipulation easy the ANSI committee added a new class called string. It allows us to define objects of string type and they can be used as built in data type.
- The string class is considered as another container class and not a part of STL (Standard Template Library).
- The programmer should include the string header file.
- Instead of declaring character array an object of string, class is defined. Using member function of string class, string operations such as copying, comparison, concatenation etc. are carried out more easily as compared to C library functions.

In C, we declare strings as given below:

```
char text[10];
```

whereas in C++ string is declared as an object. The string object declaration and initialization can be done at once using constructor in the string class.

The constructors of the string class are described in below Table

Table : String constructors

Constructors	Meaning
String ( );	Produces an empty string
String (const char *text);	Produces a string object from a null ended string
String (const string & text);	Produces a string object from other string objects

We can declare and initialize string objects as follows:

```
string text;                Declaration of string objects

string text("C++");         // Using constructor without argument
```

13. WRITE ABOUT RELATIONAL OPERATORS?

- These operators can be used with string objects for assignment, comparison etc.

The following program illustrates the use of relational operators with string objects.

Program to compare two strings using string objects and relational operators.

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

int main( )
{
        string s1("OOP");
        string s2("OOP");
        if (s1==s2)
        cout <<"\n Both the strings are identical";
        else
        cout <<"\n Both the strings are different";
        return 0;
}
```

OUTPUT

Both the strings are identical

Explanation: In the above program, two string objects s1 and s2 are declared. Both the string objects are initialized with the string "OOP". The ifstatement checks whether the

two strings are identical or different. Appropriate message will be displayed on comparison. Thus, in this program the two string objects contain the same string and hence, it displays the message "Both the strings are identical".

14. WRITE ABOUT SOME OF THE STRING HANDLING FUNCTIONS WITH SAMPLE PROGRAM?

- The member functions insert( ), replace( ), erase( ) and append( ) are used to modify the string contents. The following program illustrates the use of these functions:

insert( ):

- This member function is used to insert specified strings into another string at a given location. It is used in the following form:

  s1.insert(3,s2);

  Here, s1 and s2 are string objects. The first argument is used as location number for calling string where the second string is to be inserted.

Program to insert one string into another string using insert( ) **function.**

```
# include <iostream>
# include <string>

using namespace std;
int main( )
{
string s1("abchijk");
string s2("defg");

cout <<"\n s1= "<<s1;
cout <<"\n s2= "<<s2;
cout <<"\n after insertion";

s1.insert(3,s2);
```

```
cout <<"\n s1= "<<s1;
cout <<"\n s2= "<<s2;

return 0;
}
```

STRING ATTRIBUTES

- The various attributes of the string such as size, length, capacity etc. can be obtained using member functions.
- The size of the string indicates the total number of elements currently stored in the string.
- The capacity means the total number of elements that can be stored in string.
- The maximum size means the largest valid size of the string supported by the system.

size( )

- The member function size( ) determines the size of the string object i.e., number of bytes occupied by the string object. It is used in the given format.

  s1.size( )

  Here s1 is a string object and size( ) is a member function.

Program to display the size of the string object before and after initialization.

```
# include <iostream>
# include <string>

using namespace std;
 int main( )
{
  string s1;
  cout <<"\nsize : "<<s1.size( );
```

```
 s1="hello";
 cout <<"\nNow size : "<<s1.size( );
 return 0;
}
```

ACCESSING ELEMENTS OF STRINGS

- It is possible to access particular word or single character of string with the help of member functions of string class.

The supporting functions are illustrated with suitable examples.

- at( )

This function is used to access individual character. It requires one argument that indicates the element number. It is used in the given format.

```
s1.at(5);
```

Here s1 is a string object and 5 indicates 5th element that is to be accessed.

Program to display the string elements one by one. Use the member function at( )**.**

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

int main( )
{
string s1("PROGRAMMING");

for (int j=0;j<s1.length( );j++)
cout<<s1.at(j);

return 0;
}
```

OUTPUT

PROGRAMMING

COMPARING AND EXCHANGING

- The string class contains functions, which allows the programmer to compare and exchange the strings.

The related functions are illustrated below with suitable examples:

- compare( )

The member function compare( ) is used to compare two string or sub-strings. It returns 0 when the two strings are same, otherwise returns a non-zero value.

It is used in the following formats:

s1.compare(s2);

Here s1 and s2 are two string objects.

s1.compare (0,4,s2,0,4);

Here s1 and s2 are two string objects.

The integer number indicates the sub-string of both the strings that are to be compared.

Program to compare two strings. Use compare( ) **function.**

```
# include <iostream>
# include <string>

using namespace std;

int main( )
```

```
{
string s1 ("Take");
string s2 ("Taken");

int d=s1.compare(s2);
cout <<"\n d= "<<d;

d=s1.compare (0,4,s2,0,4);
cout <<"\n d= "<<d;
return 0;
}
```

MISCELLANEOUS FUNCTIONS

- assign( )

  This function is used to assign a string wholly/ partly to other string object. It is used in the following format:

```
s2.assign(s1)
```

Here s2 and s1 are two string objects. The contents of string s1 are assigned to s2.

```
s2.assign(s1,0,5);
```

In the above format, element from 0 to 5 are assigned to object s2.

Program to assign a sub-string from main string to another string object.

```
# include <iostream>
# include <string> using namespace std;
int main( )
{
 string s1("c plus plus");
 string s2;
 int x=0;
 s2.assign(s1,0,6);
```

```
 cout <<s2;
 return 0;
}
```

OUTPUT

C plus

Explanation: In the above program, s1 and s2 are two string objects. The object s1 is initialized with the string "c plus plus". The object s2 invokes the assign( ) function with the integer argument 0,6, that indicates the sub-string. The sub-string is assigned to object s2 i.e. "C plus".

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

SUBJECT NAME: OBJECT ORIENTED PROGRAMMING AND DESIGN SUBJECT CODE: CST33

UNIT IV

Files: File Stream classes **–** Checking for errors – file opening modes – file pointers and manipulators – manipulators with arguments – read and write operations – Binary and ASCII files – Random access operation – Error handling functions – command line arguments – stdstreams.

Generic Programming with Templates: Generic Functions- Need of Template – Normal function template – class template with more parameters – Function template with more parameters, overloading of function templates, class template with overloaded operators – class templates and inheritance.

Exception Handling: Fundamentals of Exception Handling – Catching Class Types – Using Multiple catch statements – Catching All Exception – Rethrowing Exception – Specifying Exception – Exceptions in constructors and destructors – controlling uncaught Exceptions – Exception and operator overloading – Exception and inheritance – Class Template and Exception handling.

---

2 MARKS

1. **What is a file?**

   - A file is a collection of inter related data stored in a particular area on the disk.
   - Most of the applications will have their own features to save some data to the local disk and read data from the disk again.
   - C++ file I/O classes simplify such file to read/write operations for the programmer by providing easier way to use classes.

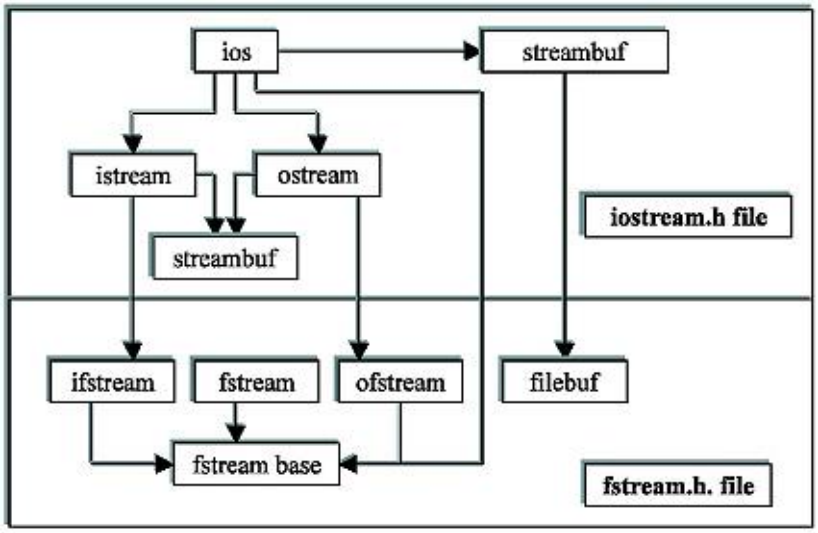
2. **What are streams?**

   - Stream is a flow of data between the source and destination. In the computer streams act as an interface between the data /programs and the files.
   - The stream that supplies data to the program is known as input stream and one that receives data from the program is known as output stream.

3. **What is file stream?**

   Input stream-extracts (or reads) data from the file. Output stream-extracts (or writes) data to the file is called file stream.

4. **Draw the file stream classes for hierarchy.**



5. **Define ios.**

It is the base class for all sub base classes and derived classes, which inherits and properties to its child or derived classes the member functions are get(), put(), read(), write() etc;

6. **What is istream?**

It is derived from the base class ios, which provides supports for input operation and contains the same member functions of iostream class. It inherits the properties to its fstream class.

7. **What is ostream?**

It is derived from the base class ios, which provides support for output operation and contains the same member functions of iostream class. It inherits the properties to its fstream class.

8. **What is streambuf?**

The streambuf class doesnot organize streams for input or output operations.the derived classes of streambuf perform these operations. It also arranges a space for keeping input data and for sending output.

9. **What is filebuf?**

Filebuf accomplishes input and output operations with files.

Its purpose is to set the file buffers to read and write. Contains **Openprot** constant used in the **open()** of file stream classes. Also contain **close()** and **open()** as members.

10. What is fstreambase?

The fstreambase Provides operations common to file streams. Serves as a base for **fstream,ifstream** and **ofstream** class. Contains **open()** and **close()** functions.

11. What is ifstream?

The ifstream provides input operations. Contains **open()** with default input mode. Inherits the functions **get(),getline(),read(),seekg(),tellg()** functions from istream.

12. What is ofstream?

The ofstream provides output operations. Contains **open()** with default output mode. Inherits **put(),seekp(),tellp()** and **write()** functions from **ostream**.

13. What is fstream?

The fstreamProvides support for simultaneous input and output operations. Contains open with default input mode. Inherits all the functions from **istream** and **ostream** classes through **iostream**. For example, fstream invokes the member function **istream::getline()** to read character from the file. It provides **open()** function with default input mode.

14. What are the steps of file operations?

- File name
- Opening file
- Reading or writing the file (File processing)
- Detecting errors
- Closing the file

15. List some of file extensions.

The following file names are valid in MS-DOS and WINDOWS-98 operating systems.

- data.dbf //extension is .dbf
- tc.exe //extension is .exe
- prg.cpp //extension is .cpp
- prg.obj //extension is .obj

16. What are the methods for opening a file?

Before opening the file the header file fstream.h must be include. In file systems all file must be opened in two ways

- Using Constructors functions
- Using open() functions

17. How to open a file using constructors function?

Constructors is special memer function for automatic initialization of an object. A constructor is executed automatically whenever an object is created. A constructor function is different from all other non static member function in a class because it is used to initialize the variables of whatever instance is being created.

Syntax:

Class object name(value);

Where, Class- ofstream/ifstream
Object- c++ variable
Value- group of characters within double quotes

18. How to open input file output file using constructors function?

- **Output file:**

Syntax

ofstream fileobject ("disk filename");

Where, ofstream - name of class, fileobject – output file object name , diskfile name- name of the input file to create on the disk

Example

ofstream employee ("std.dat"); - This creates an output file std.dat and initialize with the file object employee.

- **Input file:**

Syntax

ifstream fileobject("disk filename");

Where, ifstream - name of class, fileobject – input file object name ,diskfile name- name of the input file to create on the disk

Example

ifstream employee ("std.dat"); - This creates an input file std.dat and initialize with the file object employee.

19. How to open a file using open() function?

The function open() is used to open afile. The open() function uses the stream object.

- **Output file:**

Syntax

ofstream fileobject;
fileobject.open("disk filename");

Where, ofstream - name of class, fileobject – name of output file object ,open – name of member function ,diskfile name- name of the input file to create on the disk

Example ofstream employee;
Employee.open ("std.dat"); - This creates an output file std.dat and initialize with the file object employee.

- **Input file:**

Syntax

ifstream fileobject;
fileobject.open("disk filename");

Where, ifstream - name of class, fileobject – name of output file object ,open – name of member function ,diskfile name- name of the input file to create on the disk

Example ifstream employee;

Employee.open ("std.dat"); - This creates an input file std.dat and initialize with the file object employee.

20. What are the steps for processing a file?

There are two steps are involved in file concept for processing the data.

i. Reading
ii. Writing

Reading: Which means storing the values from already opened input file.

Syntax

Fileobject>>var1>>var2….>>varN;

Where, file object – name of the already opened file
>> - overloaded operator (or) put to operator
Var1,var2,varN – data to read from the file.

Writing: Which means retrieving stored the values from already opened output file.

Syntax

fileobject<<var1<<var2….<<varN;

Where, file object – name of the already opened file
<< - overloaded operator (or) put to operator
Var1,var2,varN – data to write into the file.

21. How to close a file?

All the opened files should be closed before the end of the program. When the below statement is executed, it closes the file created with the file object.

Syntax:

fileobject.close();

Where, file object – opened file name, close – member function

Example

Employee.close (); - The closes the file std.dat with the file object employee.

22. What is the necessity to check for errors in file operation?

If any fault occurs during file operation, fault cannot be detected. Various errors can be made by the user while performing file operation. Such errors must be reported in the program to avoid further program failure.

23. What sort of errors will occur in file operation?

- The user attempts to read file that does not exist
- Opens a read-only file for writing purpose

24. What is Logical negation operator?

The ! (Logical negation operator) overloaded operator is useful for detecting the errors. It is a unary operator and in short it is called as not operator.

25. What is the use of Logical negation operator?

The (!) not operator can be used with object of stream classes. This operator returns non-zero value if stream error is found during operation.

26. What is the requirement to find end of file?

While reading a data from a file, it is necessary to find where the file ends i.e., end of file. The programmer cannot predict the end of file. In a program while reading the file, if the program does not detect end of file, the program drops in an infinite loop. To avoid this, it is necessary to provide correct instruction to the program that detects the end of file. Thus, when end of file is detected, the process of reading data can be easily terminated.

27. What is the purpose of end of file (eof)?

It is an instruction given to the program by the operating system that end of file is reached. The eof( ) function returns non-zero value when end of file is detected, otherwise zero.

Syntax

Objectname.eof()=0

Where, objectname- file name is used in open command

Eof()- member function

28. What are file modes?

File can be accessed in different modes. For example the file mode should be specified while opening the file. The possible mode flags are defined in the base class ios.

Syntax for function open() with two arguments

Stream-object.open("filename",mode);

- Mode-specifies the purpose foe which the file is opened.
- ios:: in for ifstream functions meaning open for reading only
- ios:: out for ofstream functions meaning open for writing only

29. List some of the file modes.

- ios::in- Open file for reading only
- ios::out- Open file for writing only
- ios::ate- Go to end-of-file on opening
- ios::app- Append to end-of-file
- ios::trunk- Delete the contents of the file if it exists
- ios::nocreate- Open fails if file the file does not exist
- ios::noreplace- Open fails if the file already exists

- ios::binary- Binary file

30. Write the Syntax for file mode using constructor?

Syntax

fstream fileobjectname(Arg1, Arg2);

Where, fstream - name of class, fileobjectname – name of object , Arg1- name of file should be within double quotes, Arg2- flag.

Example

fstream employee ("std.dat", ios::in); - This opens the file in input mode.

Syntax

fstream fileobjectname;
fileobjectname(Arg1, Arg2);

Where, fstream - name of class, fileobjectname – name of object , Arg1- name of file should be within double quotes, Arg2- flag.

Example

fstream employee ;
employee,open("std.dat", ios::app); - This opens the file in input mode.

31. Syntax for file mode using member function?

Syntax

fileobjectname.open(Arg1, Arg2|Arg3|Arg4…|ArgN);

Where, fileobjectname – name of object , Arg1- name of file should be within double quotes, Arg2, Arg3, Arg4, ArgN- flag.

Example

fstream employee ;
employee.open("std.io, ios::app|ios::nocreate);

32. What are the two pointers in file pointer?

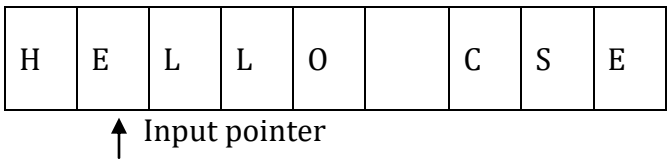
Each file has two associated pointers known as file pointer.

- **Input pointer(or get pointer):** This is used for reading the contents of a given file location and incremented automatically to the next line.
- **Output pointer(or put pointer):** This is used for writing to a given file location and incremented automatically to the next line.

33. What is read-only mode?

When a file is opened in read only mode;the input(get) pointer is initialized to point to the beginning of the file, so that the file can be read from the start.

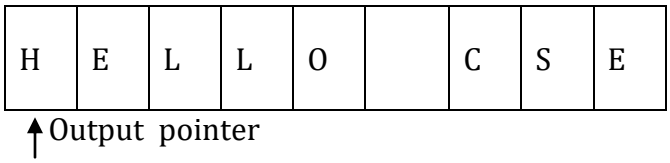
Read mode



34. What is write-only mode?

When a file is opened in write only mode;the existing contents of the file are deleted and the output pointer is set to print to the beginning of the file, so that data can be written from the start

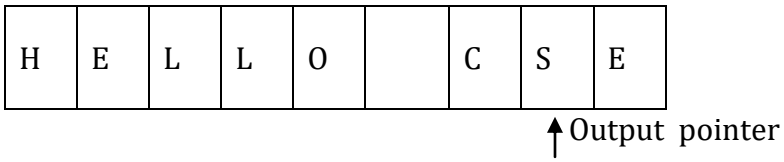
Write mode



35. What is append mode?

When a file is opened in append mode;the existing contents of the file remain unaffected and the output pointer is set to print to the end of the file, so that data can be written at the end of the existing contents.

Append mode



36. What are the functions for manipulators of file pointers?

- seekg()-Moves get pointer to a specified location

  Syntax: filename_object.seekg(pos,flag);
  Where, filename_object- name of the object, pos- byte number to move, flag- this is optional.position to move.

- seekp()-Moves put pointer to a specified location

  Syntax: filename_object.seekp(pos,flag);
  Where, filename_object- name of the object, pos- byte number to move, flag- this is optional.position to move.

- tellg()-Gives the current position of the get pointer

  Syntax: int var1 = filename_object.tellg();
- tellp()-Gives the current position of the put pointer
- **Syntax:** int var1 = filename_object.tellp();

36. What are functions used for manipulators with arguments?

- Seekg(offset, pre_position);
- Seekp(offset, pre_position);

Where, offset- specifies the number of bytes the file pointer is to be shifted from the argument.

Pre_postion of the pointer.

The offset must be positive or negative number.The negative no moves the pointer in forward direction whereas negative number moves the pointer in backward direction.

The pre-position argument may have one of the following values

- ios::beg- Beginning of the file.
- ios::cur -Current position of the file pointer
- ios::end -End of the file

37. List some of the file pointers with arguments.

- fout.seekg(o,ios::beg) -Go to start
- fout.seekg(o,ios::cur) -Stay at the current position
- fout.seekg(o,ios::end) -Go to the end of file
- fout.seekg(m,ios::beg) -Move to (m+1)th byte in the file
- fout.seekg(m,ios::cur) -Go forward by m byte from current position
- fout.seekg(-m,ios::cur) -Go backward by m bytes from current position.
- fout.seekg(-m,ios::end) -Go backward by m bytes from the end

38. What are the types of file accessing?

- **Sequential access**

  This type of file is to be accessed sequentially that is to access a particular data all the preceding data items have to be read and discarded.

- **Random access**

  This type of file allows access to the specific data directly with out accessing its preceding data items.

39. What are the sequential input and output file operations.

The file stream classes support a number of member functions to do the input and output operations on files.

put(): write character by character into the file at a time.

Syntax file_objectname.put(var);

get():read the file character by character and assigns it to the given character variable.

Syntax file_objectname.get(var);

Read(): read block of binary data pointed by the input file pointer and assigns it to the pointer variable.

Syntax file_objectname.read((char*)&var,sizeof(var));

Write(): writes block of binary data into the file.

Syntax file_objectname.write((char*)&var,sizeof(var));

40. what are the two formats for storing numeric data?

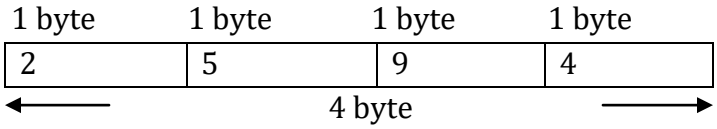
The binary number is more accurate for storing the numbers as they stored in the exact internal representation.There are no conversions while saving the data and therefore saving is much faster. There are two formats for storing numeric data.

i. Character format(ACSII)
ii. Binary format(0's and 1's)

41. Define character format.

In this format the given number by the user is stored in the form of ASCII and each occupies 1 memory byte. i.e. if the number contains 6 digits it will occupy 6 memory bytes. But it requires conversion before processing (ASCII to binary format)

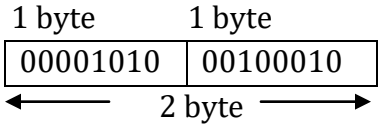
Character format



42.Define binary format.

In this format the given number by the user is stored as o's and 1's i.e. binary form. For examples an interger takes 2b memory bytes ,float takes 4 memory bytes, double takes 8 memory bytes, etc.

Binary format



43. What are the uses of binary I/O?

- It occupies less memory space
- It reduce execution time
- No need for data conversion before processing.

44. Why random access operation is essential?

Data files always contain large information and the information always changes. The changed information should be updated otherwise the data files are not useful. Thus to update data in the file we need to update the data files with latest information. To update a particular record of data file it may be stored anywhere in the file but it is necessary to obtain at which location (in terms of byte number) the data object is stored.

45. Give some examples for random access operation?

The sizeof( ) operator determines the size of object. Consider the following statements.

(a) int size=sizeof(o);

Where, o is an object and size is an integer variable. The sizeof( ) operator returns the size of object o in bytes and it is stored in the variable size. Here, one object is equal to one record.

The position of nth record or object can be obtained using the following statement.

(b) int p=(n-1 *size);

Here. P is the exact byte number of the object that is to be updated, n is the number of object, and size is the size of bytes of an individual object (record).

Suppose we want to update 5th record. The size of individual object is 26.

(c) p=(5-1*26 i.e. p=104)

Thus, the fifth object is stored in a series of bytes from 105 to 103. Using functions seekg() and seekp() we can set the file pointer at that position.

46. **List out some of the reasons result in error during read/write operation**

- An attempt to read a file which does not exist.
- The file name specified for opening a new file may already exist.
- An attempt to read contents of file when file pointer is at the end of file.
- Insufficient disk space.
- Invalid file name specified by the programmer.
- An effort to write data to the file that is opened in read only mode.
- A file opened may already be opened by another program.
- An attempt to open read only file for writing operation.
- Device error.

47. List out some of the error handling functions

- eof()-Returns true if end-of-file is encountered
- fail()-Returns true when an input or output operation has failed
- bad()-Returns true if invalid operation is attempted by or any uncoverable error has occurred
- good()-Returns true if no error has occurred

48. Write a note on command line arguments

C++ also support the feature of command line argument i.e passing the arguments at the time of invoking the program. They are typically used to pass the names of data files.

Example:

C>exam data results

Here exam is the name of file containing the program to be executed and data and results are the filenames passed to program as command line arguments. The command line arguments are typed by the user and are delimited by a space. The first argument is always the filename and contains the program to be executed. The **main()** functions which have been using upto now without any argument can take two arguments as shown below:

main(int argc,char * argv[])

49. What are the two arguments used in command line arguments?

Synatx:

main(int argc,char * argv[])

The first argument **argc** represents the number of arguments in commandline. The second argument **argv** is an array of character type pointers that points to the command line arguments. The size of this array will be equal to the value of argc.

Example:

C>exam data results

The value of **argc** would be 3 and the **argv** would be an array of three pointers to string as shown:

argv[0] exam

argv[1] data

argv[2] results

The argv[0] always represents the command name that invokes the program. The character pointer argv[1], and argv[2] can be used as file names in the file opening statements as shown:

inline.open(argv[1]); //open data file for reading

outfile.open(argv[2]); //open result file for writing

50. Define OSTRSTREAM

The strstream class is derived from istrstream and ostrstream classes. The strstream class works with memory. Using objects of ostrstream class different type of data values can be stored in an array.

51. What is generic programming?

Generic programming is an approach where gneris types are used as parameters in algorithms so that they work for variety of suitable data types and data structures.

52. Define generic function?

A function that works for all C++ data types is called as generic function. The template provides generic programming by defining generic classes. In templates, generic data types are used as arguments and they can handle a variety of data types.

53. What is template?

Template can be used to create a family of classes or functions.

Example: A class template for an array class would enable us to create arrays for various data types such as int array and float array. Similarly, we can define a template for a function, say mul(), that would help us create various versions of mul() for multiplying int, float and double type values.

54. What are the advantages of using template?

- It supports all data types.
- It reduces number of objects, classes and functions in programs.
- This means with the help of single function or class we can do so many operation with different data types.

55. What are the rules for declaring template?

- The keyword template should be placed in front of function name.
- The function template call is same as ordinary function call.
- The user can use any number for template class data type normally we can use T.
- The argument list should contain at least one argument from each template class data type.

56. What is function template?

A function template specifies how an individual function can be constructed. The limitation of such functions is that they operate only on a particular data type. It can be overcome by defining that function as a function template or generic function.

For example

we can create a template for function square( ). It helps us to calculate square of a number of any data type including int, float, long, and double.

Syntax:

```
template<class T,…>
ReturnType FuncName(arguments)
{
        …..//body of the function template
        …..
}
```

57. What is class template?

Classes can also be declared to operate on different data types. Such classes are called class templates. A class template specifies how individual classes can be constructed similar to normal class specification

Syntax:

Class name <data type>object1,object2,…

General form

```
template <class T1,class T2,…>
class classname
{
T1 data1;
….    //functions of template arguments T1,T2,….
void func1(T1 a,T2 &b);
…
```

```
T func2(T2 *x,T2 *y);
};
```

58. Write a template function to find the maximum number from a template array of size N.

```
using namespace std;
template<class T>
void max(T a[],T &m, int n)
{
 for(int i=0;i<n;i++)
 if(a[i]>m)
 m=a[i];
}

int main()
{
 int x[5]={10,50,30,40,20};
 int m=x[0];
 max(x,m,5);
 cout<<"\n The maximum value is : "<<m;
 return 0;
}
```

59. Write the syntax for class template with more arguments?

The class template may contain one or more parameters of generic data type. The arguments are separated by comma with template declaration.

Declaration Syntax:

```
Template<class T1,class T2>
Class name_of_class
{
//class declarations and definitions
}
```

60. Write the syntax for function template with more arguments?

The function template define member function with generic arguments.

Declaration Syntax:

```
Template<class T>
Return_data_type function_-name(parameter of template type)
{
Statement1;
Statement2;

Statement3;

}
```

61. How the template function will be overloaded?

A template function also supports overloading mechanism. It can be overloaded by normal function or template function. While invoking these functions an error occurs if no accurate match is met. No implicit conversion is carried out in parameters of template functions. The complier follows the following rules for chossing appropritate function when program contains overloaded function.

- Searches fpr accurate match of functions, if found it is invoked
- Searches for template function through which a function that can be invoked with accurate match can be generated; if found it is invoked.
- Attempts normal overloading declaration for the function.
- In case no match is found,an error will be reported.

62. How the Class template is overloaded with operators?

The template class permits to declare overload operators and member function. The syntax is same as class opertorfunction is similar to class template members and functions

63. Write a note on class template and inheritance?

The template class can also act as base class. When inheritance is applied with template class it helps to compose hierarchical data structure known as container class.

- Derive a template class and add new member to it. The base class must be of template type.
- Derive a class from non-template class. Add new template type member to derived class.
- It is also possible to derive classes from template base class and omit the template features of the derived classes.

Syntax:

Template <class Tl,…>

Class XYZ

{//template type data members and member functions

}Template<class TL>

Class ABC: public XYZ <TL>

{template type data membbers and member function.

}

64. What are Exceptions?

Exceptions which occur during the program execution, due to some fault in the input data.

65. What is exception handling?

Exception handling is the programming technique Using this method, run time errors can be detected and reported to the user for taking necessary action. In another words, it is defined as the run time anomalies that a program may encounter while executing the coding.

66. What are the uses of exeception handling?

It avoids abnormal situation (syatem lock)during run time.

It help the user by reporting the reason for abnormal conditions so that the appropriate action will be taken.

This mechanism suggests a separate error handling code that performs the following tasks

- Find the problem (hit the exception).
- Inform the error that has occurred(throw the exception)
- Receive the error details(catch the exception).

67. What are the two types of errors?

Errors are produced while we writing a program.

- Syntax error
- Logical error
- Exception(or) runtime error.

68. What is syntax error.

This type of error occurs when the user violates the language's rule and regulations.This happens because of the poor understanding of the language by the user.

Example,

Cout>>"Welcome to CSE:" // invalid

Cout<<"Welcome to CSE:" // valid

69. What is logical error.

This type of error occurs when the programmer violates the logical concepts of the language.This happen because of the poor understanding of the problem solving by the user.These errors can be identified only by the wrong output for the right input.

70. What is Exceptions or run time error.

This type error occurs during the run time. There are two types of exception or runtime errors.

- Synchronous exception
- Asynchronous exception

71. What is a synchronous exception?

It is defined as the errors such as "out of range" and overflow belongs to the synchronous type execeptions.

Example,

- Division by zero

- Exceeding maximum limit of an array
- Running out of memory
- Disk space problem

72. What is an Asynchronous exception?

Errors that are caused by events beyond the control of the programs.

Example,

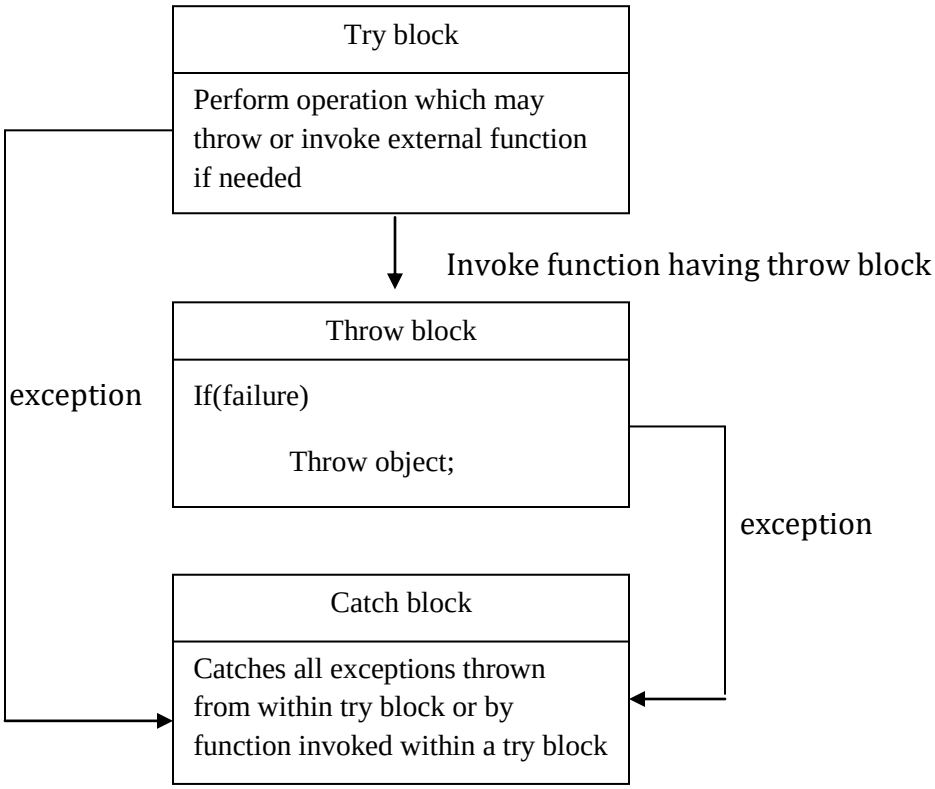
- Disk error
- Keyboard problems

73. What are the blocks used in the Exception Handling?

The exception-handling mechanism uses three blocks

1)try block
2)throw block
3)catch block

74. Draw the Exception handling model?



75. Define the exception handling constructs.

- **try**

This keyword defines a boundary within which an exception can occur. A block of code in which an exception may occur must be prefixed by this keyword.

- **throw**

Throw is used to raise an exception when an error is generated in the computation. It initializes a temporary object to be used in throw.

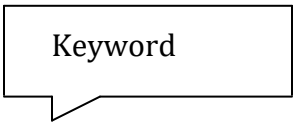
- **catch**

This keyword represents exception handler. It must be compulsorily used immediately after the statements marked by try keyword. It can also occur immediately after catch keyword. Each handler will only evaluate an exception that matches or can be converted to the type specified in the argument list

76. Write the syntax of try construct

The try keyword defines a boundary within which an exception can occur. A block of code in which an exception can occur must be prefixed by the keyword try. Following the try keyword is a

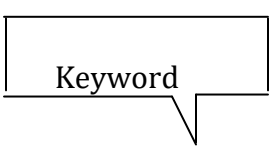
block of code enclosed by braces. This indicates that the program is prepared to test for the existence of exceptions.

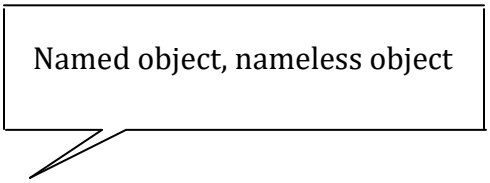



```
try
{
//code raising exception or referring to a function raising exception
}
catch(type_id1)
{
//actions for handling an exception
}
...
catch(type_idn)
{
//actions for handling an exception
 }
```

77. Write the syntax of throw construct

The keyword throw is used to raise an exception when an error is generated in the computation. The throw expression initializes a temporary object of the type T(to match the type of argument arg)used in throw(T arg)

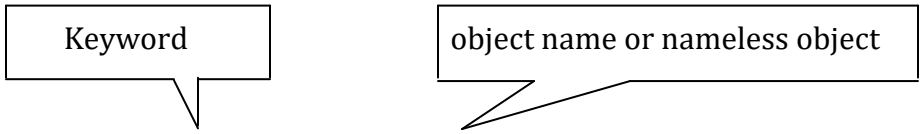





```
throw T;
```

78. Write the syntax of catch construct

The exception handler is indicated by the catch keyword. It must be used immediately after the statements marked by the try keyword. The catch handler can also occur immediately after another catch. Each handler will only evaluate an exception that matches, or can be covered to the type specified in its argument list.




```
Catch(T)
 {
    //      actions for handling an exception
    }
```

79. Define throwing mechanism

When an exepion that is desired to be handled is detected, it is thrown using the throw statement in one of the following form:

Syntax:

```
throw(exception);
throw exception;
throw;
```

the operand object exception may be of any type,including constant it is also possible to throw objectnot intended for error handling.

80. Define catch mechanism.

The catch statement catches an exception whose type matches with the type of catch argument.When it is caught,the code in the catch block is executed.

Syntax:catch (type arg)

```
{
……..
}
```

81. What are multiple catch statements?

It is possible that a program segment have more one condition to throw an exception. When an exception is thrown,the exception handlers are reached in order for an appropriate match.The first handler that yields a match is executed.

Syntax:

```
try
{
//try block
//statements;
}
Catch(type1 arg)
{
//catch block1;
```

```
//statements
}
Catch(type2 arg)
{
//catch block2;
//statements
}
……..
……..
Catch(typeN arg)
{
//catch block1;
//statements
}
```

82. When do you caught all exceptions? Give the syntax.

We can force catch statements to catch all exceptions instead of a certain type alone. This could be achieved by defining the catch statement using ellipses as follows.

Syntax: catch(..)

```
{
//statements for processing all exceptions;
}
```

83. When do you rethrow any (caught) exception? Give the syntax.

An exception handler can rethrow an exception that it has caught without even processing it. It simply invokes the throw without passing any arguments.

Syntax: throw;

This exception is then thrown to the next try/catch and is caught by a catch statement listed after that enclosing try block.

84. Write the syntax of specifying a list of exceptions

It defining the possible exceptions expected in program by the programmer.This can be implemented with the help of functions.


```
FunctionSpecification throw(type id1,type id2,……)
{
```

```
        //Function body raising exceptions if error occurs
    }
```

85. How do you use exceptions in constructors and destructors?

Copy constructors is called in exception handing when an exception is thrown from the try block using throw statement. Copy constructor mechanism is applied to initialize the temporay object. Destructors are also executed to destroy the object. If the exception is thrown from constructor,destructors are called only completely constructed objects.

86. List the functions for handling uncaught exceptions.

terminate() – it is invoked when an exception is raised and the handler is not found.

set_terminate() – allows the user to install a function that defines the program's actions to be taken to terminate the program when a handler for the exception cannot be found

unexpected() – this function is called when a function throws an exception not listed in its exception specification

set_unexpected() – it allows the user to install a function that defines the program's actions to be taken when a function throws an exception not listed in its exception specification.

11 MARKS

1. DISCUSS THE FILE STREAM CLASSES AND FILE OPENING MODES IN DETAIL

In object-oriented programming the streams are controlled using the classes. The operations with the files are mainly of two types. They are read and write. C++ provides various classes as shown in fig.1 to perform these operations.

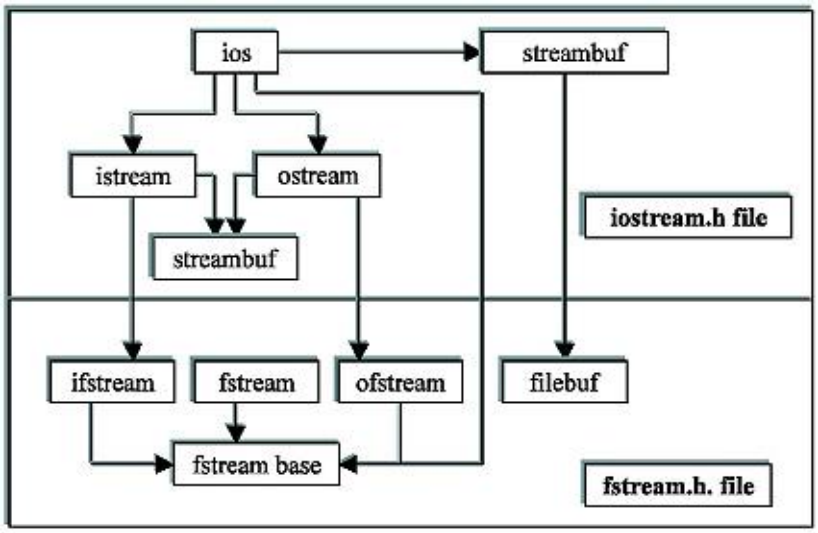


Fig .1 iostream class summary

STREAM CLASSES FOR CONSOLE OPERATIONS

- ios : ios class is the base class. All other classes are derived from the ios class. These classes contain several member functions that perform input and output operations.
- istream: istream class control input function.
- ostream :ostream class control output function.
- iostream : iostream class control both input and output functions.
- streambuf: streambuf class has low-level routines for controlling data buffer.

DETAIL OF FILE STREAM CLASSES

Class	**Contents**
filebuf	Its purpose is to set the file buffers to read and write. Contains **Openprot** constant used in the **open()** of file stream classes. Also contain **close()** and **open()** as members.
fstreambase	Provides operations common to file streams. Serves as a base for **fstream,ifstream** and **ofstream** class. Contains **open()** and **close()** functions.
ifstream	Provides input operations. Contains **open()** with default input mode. Inherits the functions **get(),getline(),read(),seekg(),tellg()** functions from istream.
ofstream	Provides output operations. Contains **open()** with default output mode. Inherits **put(),seekp(),tellp()** and **write()** functions from **ostream**.
fstream	Provides support for simultaneous input and output operations. Contains open with default input mode. Inherits all the functions from **istream** and **ostream** classes through **iostream**.

The member functions of these classes handle formatted and unformatted operations. The functions get( ), getline( ), read( ) and overloaded extraction operators (>>) are defined in the istream class. The functions put( ), write( ) and overloaded insertion operators (<<) are defined in the ostream class.

The I/O system of C++ contains a set of classes that defines the file handling methods. These include ifstream, ofstream and fstream. These classes are derived from fstreambase and form the corresponding iostream class. These classes ,designed to manage the disk files are declared in fstream and therefore this file is included in any program that uses files. The filebuf accomplishes input and output operations with files.

FILE OPENING MODES

FILE:

A file is a collection of inter related data stored in a particular area on the disk.

Most of the applications will have their own features to save some data to the local disk and read data from the disk again.

C++ file I/O classes simplify such file to read/write operations for the programmer by providing easier way to use classes.

STEPS OF FILE OPERATIONS:

Before performing file operations, it is necessary to create a file. Operation of a file involves the following basic activities:

- File name
- Opening file
- Reading or writing the file (File processing)
- Detecting errors
- Closing the file

The file name can be a sequence of characters, called as string. Strings are always declared with character array. Using file name a file is recognized.

The length of file name depends on the operating system,

Example

WINDOWS-98 supports long file names.
MS-DOS supports only eight characters.

A file name also contains extension of three characters. You might have observed the .pp extension to the C++ program file name separated by dot (.). The extension is optional.

The Following file names are valid in MS-DOS and WINDOWS-98 operating systems.

```
data.dbf     //extension is .dbf
tc.exe       //extension is .exe
prg.cpp      //extension is .cpp
prg.obj      //extension is .obj
marks        //without extension
```

For opening a file firstly a file stream is created and then it is linked to the filename. A file stream can be defined using the classes ifstream**, ofstream** and fstream that contained in the header file **fstream.h**. The class to be used depends upon the purpose whether the write data or read data operation is to be performed on the file.

FILE MODE OPERATION

Parameter	**Meaning**
ios::app	Append to end-of-file
ios::ate	Go to end-of-file on opening
ios::binary	Binary file

ios::in	Open file for reading only
ios::nocreate	Open fails if file the file does not exist
ios::noreplace	Open fails if the file already exists
ios::out	Open file for writing only
ios::trunc	Delete the contents of the file if it exists

- The mode **ios:: out** and **ios::trunc** are the same. When **ios::out** is used the contents of specified file (if present) will be deleted (truncated). The file is treated as a new file.
- When file is opened using **ios::app** and **ios::ate** modes, the character pointer is set to end of the file. The **ios::app** lets the user to add data at end of file where as **ios:ate** allows user to add or update data anywhere in the file. If the given file does not exist, new file is created. The mode **ios::app** is applicable to the output file only.
- The **ifstream** creates input stream and **ofstream** creates output stream. Hence, it is not compulsory to give mode parameters.
- While creating an object of **fstream** class, the programmer should provide the mode parameter. The **fstream** class does not have default mode.
- The file can be opened with one or more mode parameters. When more than one parameters are necessary, bitwise or operator separates them. The following statement opens a file for appending. It does not create a new file if the specified file is not present.

A FILE CAN BE OPENED IN TWO WAYS:

- Using the constructor function of class.
- Using the member function **open()** of the class.

The first method is useful only when one file is used in the stream.

The second method is used when multiple files are to be managed using one stream.

OPENING FILES USING CONSTRUCTOR:

While using constructor for opening files, filename is used to initialize the file stream object. This involves the following steps

(i) Create a file stream object to manage the stream using the appropriate class i.e) the class **ofstream** is used to create the output stream and the class **ifstream** to create the input stream.

(ii) Initialize the file object using desired file name.

For example, the following statement opens a file named “results” for output:

ofstream outfile(“results”); //output only

This create outfile as an **ofstream** object that manages the output stream. Similarly, the following statement declares infile as an **ifstream** object and attaches it to the file data for reading (input).

ifstream infile ("data"); //input only

The same file name can be used for both reading and writing data.

For example

Program1

……..

……..

ofstream outfile ("salary"); //creates outfile and connects salary to it

……

……..

Program 2

……

……

ifstream infile ("salary"); //creates infile and connects salary to it

……..

……..

The connection with a file is closed automatically when the stream object expires i.e when a program terminates.In the above statement ,when the program 1 is terminated,the salary file is disconnected from the outfile stream.The same thing happens when program 2 terminates.

Instead of using two programs,one for writing data and another for reading data ,a single program can be used to do both operations on a file.

……

……..

outfile.close(); //disconnect salary from outfile and connect to infile

ifstream infile ("salary");

……..

……

infile.close();

The following program uses a single file for both reading and writing the data .First it take data from the keyboard and writes it to file.After the writing is completed the file is closed.The

program again opens the same file read the information already written to it and displays the same on the screen.

Program working with single file

```
//Creating files with constructor function
#include <iostream.h>
#include <fstream.h>
int main()
{
ofstream outf("ITEM");
cout <<"enter item name: ";
char name[30];
cin >>name;
outf <<name <<"\n";
cout <<"enter item cost :";
float cost;
cin >>cost;
outf <<cost <<"\n";
outf.close();
ifstream inf("item");
inf >>name;
inf >>cost;
cout <<"\n";
cout <<"item name : " << name <<"\n";
cout <<"item cost: " << cost <<"\n";
inf.close();
return 0;
}
```

OPENING FILES USING OPEN()

In the above example we have learnt how to open files using constructor and open( ) function by using the objects of ifstream and ofstream classes. The opening of file also involves several modes depending upon operation to be carried out with the file. The open( ) function has two arguments as given below:

Syntax of open( ) function

```
object.open ( "file_ name", mode);
```

Here object is a stream object, followed by open( ) function. The bracket of open function contains two parameters. The first parameter is name of the file and second is mode in which file is to be opened. In the absence of mode parameter default parameter is considered.

The default values for ifstream is (ios::in),reading only and fstream is (ios::out),writing only

(i) Opening file for write operation

ofstream out; // creates stream object out

out.open("marks.dbf"); // opens file link with the object out

out.close() // closes he file pointed by the object out

out.open("results.dbf"); //opens another file

(ii) Opening file for read operation

ifstream in; // creates stream object in

in.open("marks.dbf"); // opens file link with the object in

in.close() // closes he file pointed by the object in

in.open("results.dbf"); //opens another in

The function **open()** can be used to open multiple files that uses the same stream object. For example to process a set of files sequentially, in such case a single stream object can be created and can be used to open each file in turn. This can be done as follows;

File-stream-class stream-object;

stream-object.**open** ("filename");

The following example shows how to work simultaneously with multiple files

Program working with multiple files

```
//Creating files with open() function
#include <iostream.h>
#include<fstream.h>
int main()
{
ofstream fout;
fout.open("country");
fout<<"United states of America \n";
fout<<"United Kingdom";
fout<<"South korea";
fout.close();
```

```
fout.open("capital");
fout<<"Washington\n";
fout<<"London\n";
fout<<"Seoul \n";
fout.close();
const int N =80;
char line[N];
ifstream fin;
fin.open("country");
cout<<"contents of country file \n";
while (fin)
{
fin.getline(line,N);
cout<<line;
}
fin.close();
fin.open("capital");
cout<<"contents of capital file";
while(fin)
{
fin.getline(line,N);
cout<<line;
}
fin.close();
return 0;
}
```

2. WRITE A NOTE ON CHECKING FOR ERRORS AND EOF WITH EXAMPLES?

CHECKING FOR ERRORS

When we carried out the file operations without thinking whether it is performed successfully or not. If any fault occurs during file operation, fault cannot be detected. Various errors can be made by the user while performing file operation. Such errors must be reported in the program to avoid further program failure. When the user attempts to read file that does not exist or opens a read-only file for writing purpose, in such situations operation fails. Such errors must be reported and proper actions have to be taken before further operations.

The ! (Logical negation operator) overloaded operator is useful for detecting the errors. It is a unary operator and in short it is called as not operator. The (!) not operator can be used with object of stream classes. This operator returns non-zero value if stream error is found during operation.

Program to detect error in file operation using ! operator

```
#include <fstream.h>
#include<constream.h>
#include<iomanip.h>
main()
{
clrscr();
char c, f_name[10];
cout <<"\n enter file name :";
cin>>f_name;
ifstream in (f_name);
if (!in)
{
 cerr <<"Error in opening file" <<F_name <<endl;
return 1;
}
in>> resetiosflags(ios::skipws);
while (in)
{
in>>c;
cout <<c;
}
return 0;
}
```

OUTPUT

```
Enter file name : TEXT
One two three four
1 2 3 4
**The End **
```

Explanation: In the above program ,using constructors of a class ifstream a file is opened. The file that is to be opened is entered by the user during program execution. If the file does not exist, the

ifstream object (in) contains 0. The if () statement checks the value of object in and if operation fails a message will be displayed a program is terminated. In case the file exists the using while () loop contents of file is read one character at a time and displayed on the screen. You can observe use of not operator with object in if() and while () statement. If the statement in>> resetiosflags(ios::skipws); is removed the contents of file would be displayed without space in one line.The operator >> ignores white space character.

Consider the following statement:

With not **operator**

```
if (!in)
{
statement1;
else
statement2;
}
```

The not operator is used in association with the object. It is also possible to perform an operation without the use of not operator. The statement would be as follows and it works opposite as compared to the above statement.

Without not **operator**

```
if (in){
statement1;
else
statement2;
}
```

EOF:

The eof ( ) stands for end of file. It is an instruction given to the program by the operating system that end of file is reached.While reading a data from a file, it is necessary to find where the file ends i.e., end of file. The programmer cannot predict the end of file. In a program while reading the file, if the program does not detect end of file, the program drops in an infinite loop.

To avoid this, it is necessary to provide correct instruction to the program that detects the end of file. Thus, when end of file is detected, the process of reading data can be easily terminated. The eof( ) member function is used for this purpose.

It checks the ios::eofbit in the ios::state. The eof( ) function returns non-zero value when end of file is detected, otherwise zero.

Program to read and display contents of file. Use eof() function

```
#include <fstream.h>
#include<constream.h>
#include<iomanip.h>
```

```
main()
{
clrscr();
char c, f_name[10];
cout <<"\n enter file name :";
cin>>f_name;
ifstream in (f_name);
if (!in)
{
 cerr <<"Error in opening file" <<F_name <<endl;
return 1;
}
while (in.eof()==0)
{
in.get(c);
cout <<c;
}
return 0;
}
```

OUTPUT

Enter file name : text

Programming with ANSI and Turbo C++

Explanation: The above program is same as the previous one. Here, the member function eof () is used in the if () statement. The program displays the contents of file "text". While specifying a file name for reading purpose, be sure it must exist.

3. DESCRIBE FILE MANIPULATORS WITH THEIR SYNTAXES?

FILE POINTERS AND MANIPULATORS:

Each file has two pointers known as file pointers, one is called the input pointer and the other is called output pointer. The input pointer is used for reading the contents of a given file location and the output pointer is used for writing to a given file location. Each time an input or output operation takes place, the appropriate pointer is automatically advanced.

DEFAULT ACTIONS:

When a file is opened in read-only mode, the input pointer is automatically set at the beginning so that file can be read from the start. Similarly when a file is opened in write-only mode the existing contents are deleted and the output pointer is set at the beginning. This enables us to write the file from start. In case an existing file is to be opened in order to add more data, the file is opened in 'append' mode. This moves the pointer to the end of file.

FUNCTIONS FOR MANIPULATIONS OF FILE POINTER:

All the actions on the file pointers takes place by default. For controlling the movement of file pointers file stream classes support the following functions

- **seekg()** Moves get pointer (input)to a specified location.
- **seekp()** Moves put pointer (output) to a specified location.
- **tellg()** Give the current position of the get pointer.
- **tellp()** Give the current position of the put pointer.

For example,

```
infile.seekg(10);
```

the statement moves the pointer to the byte number 10.The bytes in a file are numbered beginning from zero. Therefore ,the pointer to the 11^{th} byte in the file.

Consider the following statements:

```
ofstream fileout;

fileout.open("hello",ios::app);

int p=fileout.tellp();
```

On execution of these statements, the output pointer is moved to the end of file "hello"

And the value of p will represent the number of bytes in the file.

MANIPULATORS WITH ARGUMENTS

The seekp( ) and seekg( ) functions can be used with two arguments. Their formats with two arguments follows below:

- seekg (offset, pre_position);
- seekp (offset,pre_position);

The first argument offset specifies the number of bytes the file pointer is to be shifted from the argument pre_position of the pointer. The offset must be a positive or negative number. The positive number moves the pointer in forward direction whereas negative number moves the pointer in backward direction. The pre_position argument may have one of the following values.

- ios::beg **Beginning of the file**
- ios::cur **Current position of the file pointer**

- ios::end **End of the file**

FILE POINTER WITH ITS ARGUMENTS

Seek call	**Action**
fout.seekg(o,ios::beg)	Go to start
fout.seekg(o,ios::cur)	Stay at the current position
fout.seekg(o,ios::end)	Go to the end of file
fout.seekg(m,ios::beg)	Move to (m+1)th byte in the file
fout.seekg(m,ios::cur)	Go forward by m byte from current position
fout.seekg(-m,ios::cur)	Go backward by m bytes from current position.
fout.seekg(-m,ios::end)	Go backward by m bytes from the end

4. EXPLAIN SEQUENTIAL AND RANDOM FILE OPERATIONS?

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.

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 for 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 0;
}
```

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 for 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;
}
```

BINARY AND ASCII FILES

ASCII codes are used by the I/O devices to share or pass data to the computer system, but central processing unit (CPU) manipulates the data using binary numbers i.e., 0 and 1.

For this reason, it is essential to convert the data while accepting data from input devices and displaying the data on output devices. Consider the following statements:

```
cout<<k;   // Displays value of k on screen
cin>>k;    // Reads value for k from keyboard
```

Here, k is an integer variable. The operator << converts value of integer variable k into stream of ASCII characters. In the same fashion, the << operator converts the ASCII characters entered by the user to binary. The data is entered through the keyboard, a standard input device.

For example,

You enter 21. The stream operator >> gets ASCII codes of the individual digits of the entered number 21 i.e., 50 and 49. ASCII code of 2 and 1 are 50 and 49 respectively. The stream operator >> converts the ASCII value to equivalent binary format and assigns it to the variable k. The stream operator << converts the value of k (21) that is stored in binary format into equivalent ASCII codes i.e., 50 and 49.

RANDOM ACCESS OPERATION

Data files always contain large information and the information always changes. The changed information should be updated otherwise the data files are not useful. Thus to update data in the file we need to update the data files with latest information. To update a particular record of data file it may be stored anywhere in the file but it is necessary to obtain at which location (in terms of byte number) the data object is stored.

The sizeof( ) operator determines the size of object. Consider the following statements.

(a) int size=sizeof(o);

Where, o is an object and size is an integer variable. The sizeof( ) operator returns the size of object o in bytes and it is stored in the variable size. Here, one object is equal to one record.

The position of nth record or object can be obtained using the following statement.

(b) int p=(n-1 *size);

Here. P is the exact byte number of the object that is to be updated, n is the number of object, and size is the size of bytes of an individual object (record).

Suppose we want to update 5^{th} record. The size of individual object is 26.

(c) p=(5-1*26 i.e. p=104)

Thus, the fifth object is stored in a series of bytes from 105 to 103. Using functions seekg() and seekp() we can set the file pointer at that position.

5. EXPLAIN ABOUT ERROR HANDLING DURING FILE OPERATIONS IN DETAIL

There are many problems encountered while dealing with files like

- a file which we are attempting to open for reading does not exist.
- The file name used for a new file may already exist.
- We are attempting an invalid operation such as reading past the end of file.
- There may not be any space in the disk for storing more data.
- We may use invalid file name.

- We may attempt to perform an operation when the file is not opened for that purpose. The C++ file stream inherits a 'stream-state 'member from the class ios. This member records information on the status of a file that is being currently used. The stream state member uses bit fields to store the status of error conditions stated above. The class ios support several member functions that can be used to read the status recorded in a file stream.

ERROR HANDLING FUNCTIONS

Function	**Return value and meaning**
eof()	Returns true(non zero value) if end of file is encountered while reading otherwise returns false(zero).
fail()	Returns true when an input or output operation has failed .
bad()	Returns true if an invalid operation is attempted or any unrecoverable error has occurred. However, if it is false ,it may be possible to recover from any other error reported and continues operation.
good()	Returns true if no error has occurred. This means all the above functions are false. For instance,if file.good() is true.all is well with the stream file and we can proceed to perform I/O operations. When it returns false, no further operations is carried out.

These functions can be used at the appropriate places in a program to locate the status of a file stream and thereby to take the necessary corrective measures.

Example:

……

……..

ifstream infile; infile.open("ABC");

while(!infile.fail())

{

……

…….. (process the file)

……..

}

if (infile.eof())

{

…….(terminate the program normally)

}

else

if (infile.bad())

{

……..(report fatal error)

}

else

{

infile.clear(); //clear error state

…….

…….

}

……..

……..

The function **clear()** resets the error state so that further operations can be attempted.

The insertion and extraction operators, known as stream operator, handles formatted data. The programmer needs to format data in order to represent it in a suitable fashion.

COMMAND LINE ARGUMENTS

Like C,C++ also support the feature of command line argument i.e passing the arguments at the time of invoking the program. They are typically used to pass the names of data files.

Example:

C>exam data results

Here exam is the name of file containing the program to be executed and data and results are the filenames passed to program as command line arguments. The command line arguments are typed by the user and are delimited by a space. The first argument is always the filename and contains the program to be executed. The **main()** functions which have been using upto now without any argument can take two arguments as shown below:

main(int argc,char * argv[])

The first argument **argc** represents the number of arguments in commandline. The second argument **argv** is an array of character type pointers that points to the command line arguments. The size of this array will be equal to the value of argc.

For instance, command line

C>exam data results

The value of **argc** would be 3 and the **argv** would be an array of three pointers to string as shown:

argv[0] exam

argv[1] data

argv[2] results

The argv[0] always represents the command name that invokes the program. The character pointer argv[1], and argv[2] can be used as file names in the file opening statements as shown:

……

……

inline.open(argv[1]); //open data file for reading

……..

……..

outfile.open(argv[2]); //open result file for writing

……..

OSTRSTREAM

The strstream class is derived from istrstream and ostrstream classes. The strstream class works with memory. Using objects of ostrstream class different type of data values can be stored in an array.

P**rogram to demonstrate use of ostrstream objects.**

```
# include <strstream.h>

# include <iomanip.h>

# include <conio.h>

main ( )

{

 clrscr( );

 char h='C';

 int j=451;

 float PI=3.14152;

 char txt[]="applications";
```

```
 char buff[70];
 ostrstream o (buff,70);
 o<<endl <<setw(9)<<"h="<<h<<endl <<setw(9) <<"j="<<oct<<j<<endl
 <<setw(10)<<"PI="<<setiosflags(ios::fixed)<<PI<<endl  <<setw(11)
 <<"txt="<<txt <<ends;
 cout<<o.rdbuf( );
return 0;
}
```

OUTPUT

h=C
j=703
PI=3.14152
txt= applications

6. DISCUSS ABOUT CLASS TEMPLATE WITH MORE PARAMETERS?

TEMPLATE

- Template is a new concept which enables us to define generic and functions and thus provides support for generic programming.
- Generic programming as an approach where generic types are used as parameters in algorithms so that they work for a variety of suitable data types and data structures.

A template can be used to create a family of classes or functions. For example, a class template for an **array** class would enable us to create arrays of various data types such as int array and **float** array .similarly, we can define a template for a function, say mul(),hat would help us create versions of **mul()** for multiplying **int**, **float** and **double** type values.

A template can be considered as a kind of macro. When an object of a specific type is define for actual use, the template definition for that class is substitute with the required data type. Since a template is defined with a parameter that would be replaced by a specified data type at the time of actual use of the class or function, the templates are sometimes called parameterized class or functions.

Like functions, classes can be declared to handle different data types. Such classes are known as **class templates.** These classes are generic type and member functions of these classes can operate on different data types. The class template may contain one or more parameters of generic data type. The arguments are separated by comma with template declaration. The declaration is as follows:

CLASS TEMPLATE:

To declare a class of template type, following syntax is used.

TEMPLATE DECLARATION

```
template class <T>
  class name_of_class
  {
    // class data member and function
  }
```

The first statement template class <T> tells the compiler that the following class declaration can use the template data type. T is a variable of template type that can be used in the class to define variable of template type. Both template and class are keywords. The <> (angle bracket) is used to declare variables of template type that can be used inside the class to define variables of template type. One or more variables can be declared separated by comma. Templates cannot be declared inside classes or functions. They must be global and should not be local.

```
Class vector
{
int *v ;
int size;
public:
vector(int m ) // create a null vector
{
v=new int[size = m];
for(int i=0;i<size;i++)
v[i]=0;
}
vector(int *a) //create a vector from an array
{

for(int i=0;i<size;i++)
v[i]=a[i];
}
int operator*9vector &y) //scalar product
{
int sum=0;
for(int i=0;i<size;i++)
sum+=this->v[i]*y-v[i];

return sum;

;}
```

The vector class can store an array of **int** numbers and perform the scalar product of two **int** vector as shown below: int main()

```
{
int x[3]={1,2,3};
int y[3]={4,5,6};
vector v1(3); //create a null vector of 3 integers
vector v2(3);
v1=x; //create v1 from the array x
v2=y;
int R=v1*v2;
```

```
cout<<"R="r;
return 0;

}
```

Now suppose we want to define a vector that can store an array of **float** value. We can do this simply replacing the appropriate **int** declaration with **float** in the **vector** class. This means that we can have to redefine the entire class all over again.

Assume that we want to define a **vector** class with the data type as a parameter and then use this class to create a vector of any data type instead of defining a new class every time. The template mechanism enables us to achieve this goal.

As mentioned earlier, template allows us to define generic classes. It is simple process to create a generic class using a template with an anonymous type.

CLASS TEMPLATE WITH MORE ARGUMENTS:

We can use more than one generic data type in a class. They are declared as a comma-separated list within the **template** specification as shown below: **Template**<**class** T1, **class** t2, ...> **class** classname

```
{
……
…….
} ;
```

Example with two generic data types:

```
 #include <iostream>
using name space std;
template<class t1,class t2>
class Test
{
T1 a;
T2 b;
public:
test(T1 x, T2 y)
{
a=x;
b=y;
}
void show()
{
cout<<a<<"and"<<<<"\n";
}
};
int main()
{
Test<float,int> test1(1.23,123);
```

```
Test<int,char> test2(100,'W');
test1.show();

test2.show();

return 0;
};
```

OUTPUT

1.23 and 123

100 and W

7. EXPLAIN FUNCTION TEMPLATE IN DETAIL?

FUNCTION TEMPLATE:

A function template specifies how an individual function can be constructed. The limitation of such functions is that they operate only on a particular data type. It can be overcome by defining that function as a function template or generic function.

Templates help the programmer to declare group of functions. When used with functions they are called function templates. For example, we can create a template for function square( ). It helps us to calculate square of a number of any data type including int, float, long, and double.

SYNTAX OF FUNCTION TEMPLATE

```
template<class T,...>
ReturnType FuncName(arguments)
{
        .....//body of the function template
        .....
}
```

NORMAL FUNCTION TEMPLATE

A normal function (not a member function) can also use template arguments. The difference between normal and member function is that normal functions are defined outside the class.

They are not members of any class and hence can be invoked directly without using object of dot operator. The member functions are the class members. It can be invoked using object of the class to which they belong.

In C++ normal functions are commonly used as in C. However, the user who wants to use C++ as better C, can utilize this concept.

WORKING OF FUNCTION TEMPLATES

After compilation, the compiler cannot guess with which type of data the template function will work. When the template function is called at that moment, from the type of argument passed to the template function, the compiler identifies the data type.

Every argument of template type is then replaced with the identified data type and this process is called as **instantiating.**

Thus, according to different data types respective versions of template functions are created. Though the template function is compatible for all data types, it will not save any memory. When template functions are used, four versions of functions are used. They are data type int, char, float, and double. The programmer need not write separate functions for each data type.

FUNCTION TEMPLATES WITH MORE ARGUMENTS

we can also define member functions with generic arguments. The format is given below:

FUNCTION TEMPLATE DECLARATION

```
template <class T>
return_data_type function_name (parameter of template type)
 {
   statement1;
   statement2;
   statement3;
}
```

Program to display the elements of integer **and** float **array using template variables with member function.**

```
# include <iostream.h>
# include <conio.h>
template <class Tl, class T2>

class data
{
   public :

void show (T1 a, T2 b)
   {     cout  <<"\na = "<<a <<" b = "<<b;   }
};

int main( )
{
clrscr( );
int  i[]={3,5,2,6,8,9,3};
float f[]={3.1,5.8,2.5,6.8,1.2,9.2,4.7};
data <int,float> h;

for (int m=0;m<7;m++)
h.show(i[m],f[m]);
```

```
return 0;
}
```

OVERLOADING OF TEMPLATE FUNCTIONS

A template function also supports overloading mechanism. It can be overloaded by normal function or template function.

1. Call an ordinary function that has an exact match.

2. Call a template function could be created with an exact match.

3. Try normal overloading resolution to ordinary functions and call the one that matches.

An error is generated if no match is found. Note that no automatic conversions are applied to arguments on the template functions. Example for showing how a template function is overloaded with an explicit function.

```
#include <iostream>
#include <string>
using namespace std;
template <class T>
void display(T x)
{
cout<<"template display:" << x<< "\n";
}
void display ( int x)
{
cout<<"Explicit display: "<< x <<"\n";
}
int main()
{
display(100);
display(12.34);
display('c');
return 0;
}
```

The output would be:

```
Explict display:100
template display:12.34
```

template display:c

NOTE:

The call display (100) invokes the ordinary version of display() and not the template version.

MEMBER FUNCTION TEMPLATES

In the previous example, the template function defined were inline i.e., they were defined inside the class. It is also possible to define them outside the class. While defining them outside, the function template should define the function and the template classes are parameterized by the type argument.

Program to define definition of member function template outside the class and invoke the function.

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

template <class T>
class data
{
     public :
     data (T c);

  };
  template <class T>
  data<T>::data (T c)
     {
     cout <<"\n"<< " c = "<<c;
     }
int main( )
{
clrscr( );
data <char> h('A');
data <int> i(100);
data <float> j(3.12);
return 0;
}
```

RECURSION WITH TEMPLATE FUNCTION

Like normal function and member function, the template function also supports recursive execution of itself. The following program illustrates this:

Program to invoke template function recursively.

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

```
# include <assert.h>

template <class O>

void display (O d)
{
 cout<<(float) (rand( )%(d*d))<<"\n";

 // if (d==1) exit(1);
 assert(d!=1);
 display(--d);

}

void main ( )
{
 int x=10;
 clrscr( ),

 display(x);
}
```

OUTPUT

```
46
49
38
12
28
17
11
7
0
0
```

Assertion failed: d!=1, file REC_TEMP.CPP, line 14
Abnormal program termination

Explanation: The template function generates random number and displays it each time when the function display( ) is invoked. The function rand( ) defined in stdlib.h is used. The function calls itself recursively until the value of d becomes 1. The assert( ) statement checks the value of d and terminates the program when condition is satisfied. The assert( ) is defined in assert.h. We can also use if statement followed by exit( ) statement as given in comment statement.

8. DISCUSS ABOUT CLASS TEMPLATE WITH OVERLOADED OPERATORS AND INHERITANCE

CLASS TEMPLATE WITH OVERLOADED OPERATORS

The template class permits to declare overloaded operators and member functions. The syntax for creating overloaded operator function is similar to class template members and functions. The following program explains the overloaded operator with template class.

Program to overload + operator for performing addition of two template based class objects.

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

template <class T>

class num
{
 private :
 T number;

 public:

 num ( )  {  number=0;  }
 void input( )
 {
  cout <<"\n Enter a number : ";
  cin>>number;
 }
 num operator +(num);
 void show( ) {  cout <<number;  }
};

template <class T>

num <T> num <T> :: operator + (num <T> c)
{
 num <T> tmp;
 tmp.number=number+c.number;
 return (tmp);
}

void main( )
{
 clrscr( );
 num <int> n1,n2,n3;
 n1.input( );

 n2.input( );
 n3=n1+n2;
 cout <<"\n\t n3 = ";
 n3.show( );
}
```

CLASS TEMPLATE REVISITED

- The template mechanism can be used safely in every concept of C++ programming such as functions, classes etc.
- We know how to define class templates. Class templates are frequently used for storage of data and can be very powerfully implemented with data structures such as stacks, linked lists etc.
- Our conventional style is to create a stack or linked list that manipulates or stores only a single type of data, for example, int.
- To store float data type we need to create a new class to handle float data type and so on can be repeated for various data types.
- Template mechanism saves our work of re-defining codes by allowing a single template based class or function to work with different data types.

CLASS TEMPLATES AND INHERITANCE

The template class can also act as base class. When inheritance is applied with template class it helps to compose hierarchical data structure known as container class.

- Derive a template class and add new member to it. The base class must be of template type.
- Derive a class from non-template class. Add new template type member to derived class.
- It is also possible to derive classes from template base class and omit the template features of the derived classes.

The template characteristics of the base class can be omitted by declaring the type while deriving the class. All the template-based variables are substituted with basic data types.

9. EXPLAIN MECHANISM OF EXPECTION HANDLING?

EXCEPTION HANDLING MECHANISM

C++ exception handling mechanism is basically built upon three keywords namely try, throw and catch. The keyword try is used to preface a block of statements which may generate exceptions. This block of statement is called try block. When an exception is detected it is thrown using throw statement in the try block. A catch block defined by the keyword catch 'catches' the exception thrown by the throw statement in the try block and handles it appropriately. The catch block that catches an exception must immediately follow the try block that throws the exception. The general form for this is

……..

……...

try

{

……

…….. //block of statements which detects and throw an exceptions

throw exception;

……..

……..

}

catch(type arg) //catches exceptions

{

…… // Block of statements that handles the exceptions

……..

……..

}

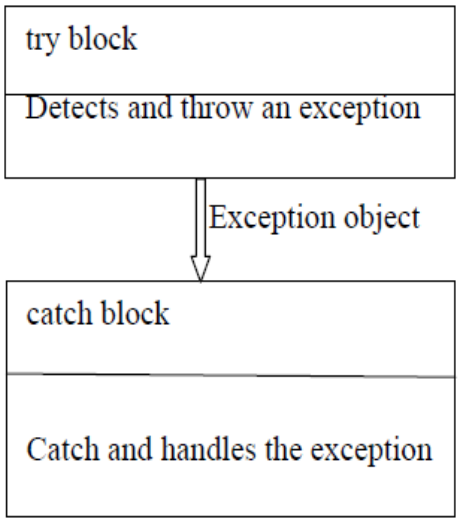
……..

……..

When the try block throws an exception, the program control leaves the try block and enters the catch statement of the catch block. If the type of object thrown matches the arg type in the catch statement, then the catch block is executed for handling the exception.

If they donot match, the program is aborted with the help of abort() function which is executed implicitly by the compiler. When no exception is detected and thrown, the control goes to

the statement immediately after the catch block i.e catch block is skipped. The below diagram will show the mechanism of exception handling



The block throwing exception

Usually there are mainly two type of bugs, logical errors and syntactic errors. The logical errors occur due to poor understanding of problem and syntactic errors arise due to poor understanding of language. There are some other problems called exceptions that are run time anomalies or unused conditions that a program may encounter while executing. These anomalies can be division by zero, access to an array outside of its bounds or running out of memory or disk space.

When a program encounters an exceptional condition it is important to identify it and dealt with it effectively. An exception is an object that is sent from the part of the program where an error occurs to that part of program which is going to control the error.

Exceptions are basically of two types namely, synchronous and asynchronous exceptions. Errors such as "out of range index" and "over flow" belongs to synchronous type exceptions .The errors that are caused by the events beyond the control of program(such as keyboard interrupts) are called asynchronous exceptions.

The purpose of exception handling mechanism is to detect and report an exceptional circumstances so that appropriate action can be taken. The mechanism for exception handling is

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

The error handling code mainly consist of two segments, one to detect error and throw exceptions and other to catch the exceptions and to take appropriate actions.

Program shows the use of try-catch blocks.

```
#include<iostream>

using namespace std;
int main()
{
int a,b;
cout<<"enter the values of a and b";
cin>>a;
cin>>b;
int x = a- b;
try
{
if(x!=0)
{
cout<<"result(a/x) = "<<a/x<<"\n";
}
else
{
throw(x);
}
```

```
}
catch(int i)
{
cout<<"exception caught : x = "<<x<<"\n";
}
cout<<"end";
return 0;
}
```

OUTPUT:

```
enter value of a and b
20 15
result(a/x)=4
end
Second run
Enter value of a and b
10 10
exception caught:x=0
end
```

The program detects and catches a division by zero problem.The output of first run shows a successful execution. When no exception is thrown, the catch statement is skipped and execution resumes with the first line after the catch. In the second run the denominator x become zero and therefore a division by zero situation occurs. This exception is thrown using the object x.Since the exception object is of integer type, the catch statement containing int type argument catches the exception and displays necessary message.

The exceptions are thrown by functions that are invoked from within the try block. The point at which the throw is executed is called throw point. Once an exception is thrown to catch block ,control cannot return to the throw point.

The general format of code for this kind of relationship is shown below

```
type function (arg list) //function with exception
{ ……..
……….
throw(object); //throw exception
……..
}
……
……..
try
```

{ …….

……. Invoke function here

……..

}

catch(type arg) //catches exception

{

……..

…….. Handle exception here

……..

}

……..

It is to be noted here that the try block is immediately followed by the catch block irrespective of the location of the throw point.

Program demonstrates how a try block invokes a function that generates an exception

```
//Throw point outside the try block
# include <iostream>
using namespace std;

void divide (int x,int y,int z)
{
cout<<"we are outside the function";
if ( ( x-y) != 0)
{ int r=z/(x-y);
cout<<"result = "<<r;
}
else
{
throw(x-y);
}
}
int main()
{
try
{
cout<<"we are inside the try block";
divide(10,20,30);
divide(10,10,20);
```

```
}
catch (int i)
{
cout<<"caught the exception";
}
return 0;
}
```

OUTPUT:

We are outside the try block
We are inside the function
Result = -3
We are inside the function
Caught the exception

THROWING MECHANISM

When an exception is encountered it is thrown using the throw statement in the following form:

```
throw (exception);

throw exception;

throw;
```

The operand object exception may be of any type including constants. It is also possible to throw objects not intended for error handling. When an exception is thrown, it will be caught by the catch statement associated with the try block. In other words the control exits the try block and transferred to catch block after the try block. Throw point can be in the deep nested scope within the try block or in a deeply nested function call.

CATCHING MECHANISM

Code for handling exceptions is included in catch blocks. The catch block is like a function definition and is of form

```
Catch(type arg)

{ statements for managing exceptions

}
```

The type indicates the type of exception that catch block handles. The parameter arg is an optional parameter name. The catch statement catches an exception whose type matches with the type of catch argument.

When it is caught, the code in the catch block is executed. After executing the handler, the control goes to the statement immediately following in catch block. Due to mismatch ,if an exception is not caught abnormal program termination will occur. In other words catch block is simply skipped if the catch statement does not catch an exception.

MULTIPLE CATCH STATEMENTS

The program segment has more than one condition to throw an exception. In such case more than one catch blocks can be associated with a try block as shown below

```
try
{//try block
}
catch(type1 arg)
{
//catch block1
}
catch(type 2 arg)
{
//catch block 2
}


……..
……..
catch (type N arg)
{
//catch block N
}
```

When an exception is thrown, the exception handlers are searched in order for an appropriate match. The first handler that yields a match is executed. After executing the handler, the control goes to the first statement after the last catch block for that try. When no match is found, the program is terminated. If in some case the arguments of several catch statements match the type of an exception, then the first handler that matches the exception type is executed.

Program shows the example of multiple catch statements.

MULTIPLE CATCH STATEMENTS

```
#include <iostream>
```

```
using namespace std;
void test (int x)
{
try
{
if (x==1) throw x; //int
else
if(x==0) throw 'x'; //char
else
if (x== -1 ) throw 1.0; //double
cout<<"end of try- block \n";
}
catch(char c) //Catch 1
{
cout<<"Caught a character \n";
}

catch (int m) //Catch 2
{ cout <<"caught an integer\n";
}
catch (double d) //catch 3
{ cout<<"caught a double \n";
}
cout<<"end of try –catch system \n\n";
}
int main()
{
cout<<"Testing multiple catches \n";
cout<<"x== 1 \n";
```

```
test(1);

cout<<"x== 0 \n";

test(0);

cout<<"x == -1 \n";

test (-1);

cout <<"x== 2 \n";

test (2);

return 0;

}
```

The program when executed first invokes the function test() with x=1 and throws x an int exception. This matches the type of parameter m in catch 2 and therefore catch2 handler is executed. Immediately after the execution , the function throws 'x', a character type exception and therefore the first handler is executed. Finally the handler catch3 is executed when a double type exception is thrown. Every time only the handler which catches the exception is executed and all other handlers are bypassed.

CATCH ALL EXCEPTIONS

In some cases when all possible type of exceptions can not be anticipated and may not be able to design independent catch handlers to catch them, in such situations a single catch statement is

forced to catch all exceptions instead of certain type alone. This can be achieved by defining the catch statement using ellipses as follows

```
catch(. . .)

{

//statement for processing all exceptions

}
```

Program illustrate the functioning of catch(…)
CATCHING ALL EXCEPTIONS

```
#include <iostream>
using namespace std;
void test(int x)
{
try
{
if (x== 0) throw x; //int
if ( x== -1) throw 'x'; //char
```

```
if ( x== 1) throw 1.0; //float
}
catch(. . .) //catch all
{
cout<<"caught an exception \n";
}
}
int main()
{
cout<<"testing generic catch\n";
test(-1);
test(0);
test(1);
return 0;}
```

We can use the catch(. . .) as a default statement along with other catch handlers so that it can catch all those exceptions that are not handled explicitly.

RETHROWING AN EXCEPTION

A handler may decide to rethrow an exception caught without processing them. In such situations we can simply invoke throw without any argument like

```
throw;
```

This cause the current exception to be thrown to the next enclosing try/catch sequence and is caught by a catch statement listed after that enclosing try block.

Program shows how an exception is rethrown and caught.

RETHROWING AN EXCEPTION

```
#include <iostream>

using namespace std;

void divide(double x, double y)

{

cout<<"Inside Function \n";

try

{ if (y== 0.0)

throw y; //throwing double

else

cout<<"division = "<< x/y<<"\n";

}
```

```
catch(double) //Catch a double

{

cout<<"Caught double inside a function \n";

throw; //rethrowing double

}

cout<<"end of function\n\n";
}
int main()
{
cout <<"inside main \n";
try
{ divide(10.5,2.0);
divide(20.0,0.0);
}
catch (double)
{ cout <<"caught double inside main \n";
}
cout <<"End of mai\n ";
return 0;
}
```

When an exception is rethrown, it will not be caught by the same catch statement or any other catch in that group. It will be caught by the an appropriate catch in the outer try/catch sequence for processing.

SPECIFYING EXCEPTIONS

In some cases it may be possible to restrict a function to throw only certain specified exceptions. This is achieved by adding a throw list clause to function definition. The general form of using an exception specification is:

```
Type function (arg-list) throw (type-list)

{

……..

……. function body

…….

}
```

The type list specifies the type of exceptions that may be thrown.Throwing any other type of exception will cause abnormal program termination. To prevent a function from throwing any exception, it can be done by making the type list empty like in the function header line.

```
        throw(); //empty list
```

Program will show this

TESTING THROW RESTRICTIONS

```
#include <iostream>
using namespace std;
void test (int x) throw (int,double)
{
if (x== 0) throw 'x'; //char
else
if (x== 1) throw x; //int
else
if (x== -1) throw 1.0; //double
cout <<"End of function block \n ";
}
int main()
{
try
{
cout<<"testing throw restrictions\n";
cout<<"x== 0\n ";
test (0);
cout<<"x==1 \n";
test(1);
cout<<"x== -1 \n";
test(-1);
cout <<"x== 2 \n";
test(2);
}
catch( char c)
{
```

```
cout <<"caught a character \n";
}
catch(int m)
{
cout<<"caught an integer \n";
}
catch (double d)
{
cout<<"caught a double \n";
}
cout<<" end of try catch system \n \n";
return 0;
```

10. EXPLAIN EXCEPTIONS IN CONSTRUCTORS AND DESTRUCTORS

Copy constructor is called in exception handling when an exception is thrown from the try block using throw statement. Copy constructor mechanism is applied to initialize the temporary object. In addition, destructors are also executed to destroy the object. If exception is thrown from constructor, destructors are called only completely constructed objects.

P**rogram to use exception handling with constructor and destructor.**

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

class number
{
    float  x;
 public :

    number (float);

    number( ) {};

    ~number ( )
    {
       cout<<"\n In destructor";
```

```
    }
    void  operator ++ (int)  //  postfix notation
    {  x++;  }

    void operator -- ( )  // prefix notation
    {
       --x; }
    void show ( )
    {
       cout <<"\n x="<<x;
    }
};


number :: number ( float k)
    {
       if (k==0)
          throw number( );
       else
       x=k;
 }


void main( )
{
    try

{

    number N(2.4);
    cout <<"\n Before incrimination: ";
    N.show ( );
    cout <<"\n After incrimination: ";
    N++;   // postfix increment
    N.show( );
    cout <<"\n After decrementation:";
    --N;   // prefix decrement
    N.show( );
    number N1(0);
}


catch (number)
{


    cout <<"\n invalid number";
    exit(1);
}
}
```

CONTROLLING UNCAUGHT EXCEPTIONS

C++ has the following functions to control uncaught exceptions.

(1) The terminate( ) Function

In case exception handler is not defined when exception is thrown terminate( ) function is invoked. Implicitly the abort( ) function is invoked.

Program to catch uncaught exceptions.

```
# include <iostream.h>
class one {};
class two {};


void main( )
{


   try
   {
    cout <<" An uncaught exception\n";
    throw two ( );

  }
   catch (one)
   {
    cout <<" Exception for one ";
   }
}
```

Explanation: In the above program, an exception is thrown for class two using the statement throw two( ). The catch( ) block is defined to handle exception for class one and it contains an argument of class one type. When an exception is thrown it does not find a match hence the program is terminated. For termination, abort( ) function is called implicitly by the compiler.

11. WRITE A PROGRAM TO THROW EXCEPTION FROM OVERLOADED OPERATOR FUNCTION.

EXCEPTION AND OPERATOR OVERLOADING

Exception handling can be used with operator-overloaded functions. The following program illustrates the same.

Program to throw exception from overloaded operator function.

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

```
class number
{
   int x;
   public :
   number ( ) {};
   void operator -- ( );

   void show ( ) { cout <<"\n x="<<x; }
   number ( int k) { x=k; }
};

void number :: operator -- ( ) // prefix notation
   {
     if (x==0) throw number( );
     else --x;
   }

void main ( )
{
   try
{
   number N(4);
   cout <<"\n Before decrementation: ";
   N.show ( );


   while (1)
   {
   cout <<"\n After decrementation";
   --N;
   N.show( );
   }
}
catch (number)
{
   cout <<"\n Reached to zero";
   exit(1);
}

}
```

EXCEPTION AND INHERITANCE

In the last few examples, we learned how exception mechanism works with operator function and with constructors and destructors. The following program explains how exception handling can be done in inheritance

P**rogram to throw an exception in derived class.**

```
# include <iostream.h>

class ABC
{

   protected:
   char name[15];
   int age;
};

class abc : public ABC   //   public derivation
{
   float height;
   float weight;
  public:

   void getdata( )
   {
     cout <<"\n Enter Name and Age : ";
     cin >>name>>age;

     if (age<=0)
     throw abc( );


   cout <<"\n Enter Height and Weight : ";
     cin >>height >>weight;
   }

void show( )
{
  cout <<"\n Name : "<<name <<"\n Age : "<<age<<" Years";
 cout <<"\n Height : "<<height <<" Feets"<<"\n Weight : " <<weight <<" Kg.";
}
};
void main( )
{
   try
   {

     abc x;
     x.getdata( );  // Reads data through keyboard
     x.show( );     // Displays data on the screen

   }
  catch (abc) { cout <<"\n Wrong age"; }
}
```

12. WRITE THE CLASS TEMPLATE WITH EXCEPTION HANDLING WITH AN EXAMPLE

The following program illustrates how excepting handling can be implemented with class templates.

Program to show exception handling with class template.

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

class sq{};

template <class T>
class square
{
   T s;

public:

   square ( T x)

   {
     sizeof(x)==1 ? throw sq ( ) : s=x*x;
   }

   void show( )
   {

     cout <<"\n Square : "<<s;
   }
};

void main( )
{

   try
   {

     square <int> i(2);
     i.show( );
     square <char> c('C');
     c.show( );

   }

   catch (sq)
   {
     cout <<"\n Square of character cannot be calculated";
   }
}
```

OUTPUT

Square: 4

Square of character cannot be calculated

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

SUBJECT NAME: OBJECT ORIENTED PROGRAMMING AND DESIGN SUBJECTCODE: CST33

UNIT V

Object Modelling and Object Oriented Software development: Overview of OO concepts – UML – Use case model – Class diagrams – Interaction diagrams – Activity diagrams – state chart diagrams - Patterns – Types – Object Oriented Analysis and Design methodology – Interaction Modelling – OOD Goodness criteria.

---

2 MARKS

1. What is object oriented analysis and design goals?

It is one of the method of tracking problems by capturing the design that is easy to communicate, review, implement and evolve.

2. Define analysis.

Analysis emphasizes an investigation of problem rather than how a solution is defined.

"Do the right thing".

3. Define design.

Design emphasizes conceptual solution of problem rather than implementation.

"Do the thing right".

4. Define Object Oriented Analysis.

Object Oriented Analysis (OOA) is a method of analysis that examines requirements from the perspective of the classes and objects found in the vocabulary of the problem domain.

5. Define OO analysis and design.

- OO Analysis find and describe the object.
- OO Design emphasizes defining the object's attribute and methods.

6. List the key steps in OOAD.

- Define usecases
- Define interaction diagram
- Define design class diagram

7. What is a model in software development?

- A model in the context of software development can be graphical, textual, mathematical, or program code-based.
- Models are very useful in documenting the design and analysis results.
- Models also facilitate the analysis and design procedures themselves.

- Graphical models are very popular because they are easy to understand and construct.
- UML is primarily a graphical modeling tool. However, it often requires text explanations to accompany the graphical models.

8. Why a model is needed?

Constructing a model is that it helps manage complexity. Once models of a system have been constructed, these can be used for a variety of purposes during software development, including the following:

- Analysis
- Specification
- Code generation
- Design
- Visualize and understand the problem and the working of a system
- Testing, etc.

9. Define UML.

Unified modeling language is a visual language used for specifying, constructing and documenting a system. It is a standard diagrammatic notation used for drawing, presenting pictures related to software engineering.

10. What are the three ways to apply UML?

- UML as a sketch.
- UML as a blue print.
- UML as a programming language.

11. Define Use case.

Use case is a textual description or a story describing the system. Use case are not an object oriented, they are simply written stories It is one of the popular tool in requirement analysis.

12. What are the uses of Use case?

- Use cases are very good way to keep everything simple.
- It gives the user's goal and perspectives.

13. Define actor and scenario.

Actor is something with behavior. It can be a system, person or a organization. Scenario is a sequence of actions and interaction between actor and system. It is also called as instance of use cases.

14. List the types of actors.

- Primary actor.
- Supporting actor.
- Off stage actor.

15. List the types of use case relationships.

- Include relationship.

- External relationship.
- Generalization.

16. What are the different types of view in UML diagrams?

The UML diagrams can capture the following five views of a system

- User's view
- Structural view
- Behavioral view
- Implementation view
- Environmental view

17. Define extension point.

Extending use case is triggered by some condition. Extension point is labelled in the base class and extending use case gives the full reference point of the extension.

18. Define five views in UML diagram?

1) **User's view:** User's view defines the functionalities (facilities) made available by the system to its users.
2) **Structural view:** The structural view defines the kinds of objects (classes) important to the understanding of the working of a system and to its implementation.
3) **Behavioral view:** The behavioral view captures how objects interact with each other to realize the system behavior. The system behavior captures the time-dependent (dynamic) behavior of the system.
4) **Implementation view:** The implementation view captures the important components of the system and their dependencies.
5) **Environmental view**: The models of different components are implemented on different pieces of hardware in environmental view

19. List out the UML diagrams?

1) Class diagram
2) Object diagram
3) Use case diagram
4) Sequence diagram
5) Collaboration diagram
6) Activity diagram
7) State chart diagram
8) Deployment diagram
9) Component diagram

20. What is use case diagram?

A use case diagram is a representation of a user's interaction with the system and depicting the specifications of a use case. A use case diagram can describe the different types of users of a system and the various ways that they interact with the system. This type of diagram is typically

used in conjunction with the textual use case and will often be accompanied by other types of diagrams as well.

21. What is class diagram?

Class diagram is a type of static structure diagram that describes the structure of a system by showing the system's classes, their attributes, operations (or methods), and the relationships among objects.

22. What is interaction diagram?

Interaction diagram can picture a control flow with nodes. It visualizes a sequence of activities.

23. What are the two types of interaction diagram?

- Sequence diagrams
- Collaboration diagrams

24. What is sequence diagram?

Sequence diagrams describe interactions among classes in terms of an exchange of messages over time.

25. What is collaboration diagram?

A collaboration diagram describes interactions among objects in terms of sequenced messages. Collaboration diagrams represent a combination of information taken from class, sequence, and use case diagrams describing both the static structure and dynamic behavior of a system.

26. Mention the goal of interaction diagram.

The main goal of interaction diagram is to show the responsibilities of objects and how they collaborate to fulfill the requirements.

27. What is activity diagram?

Activity diagrams are graphical representations of workflows of stepwise activities and actions with support for choice, iteration and concurrency.

28. What is start chart diagram?

A state chart diagram shows the behavior of classes in response to external stimuli. This diagram models the dynamic flow of control from state to state within a system.

29. Which diagrams in UML capture the behavioral view of the system?

The behavioral view is captured by the following UML diagrams:

- Sequence diagrams
- Collaboration diagrams
- State chart diagrams

- Activity diagrams

30. What is the necessity for developing use case diagram?

- The utility of the use cases are represented by ellipses and the text description serve as a type of requirements specification of the system and form the core model to which all other models must conform.
- users (actors) is in identifying and implementing a security mechanism through a login system, so that each actor can involve only those functionalities to which he is entitled to.
- It is used to prepare the documentation (e.g. users' manual) targeted at each category of user.
- Actors help in identifying the use cases and understanding the exact functioning of the system.

31. Define Association.

Association is basically a set of links that connects elements of an UML model. It also describes how many objects are taking part in that relationship.

32. Define Aggregation.

Aggregation is a special type of association where the involved classes represent a whole-part relationship. The aggregate takes the responsibility of forwarding messages to the appropriate parts. Thus, the aggregate takes the responsibility of delegation and leadership.

33. Define Generalization.

Generalization can be defined as a relationship which connects a specialized element with a generalized element. It basically describes inheritance relationship in the world of objects.

34. Define Realization.

Realization can be defined as a relationship in which two elements are connected. One element describes some responsibility which is not implemented and the other one implements them. This relationship exists in case of interfaces.

35. What is the difference between an operation and a method in the context of object-oriented design technique?

- An operation is something that is supported by a class and invoked by objects of other classes. There might be multiple methods implementing the same operation. This is called static polymorphism.
- The method names can be the same; however, it should be possible to distinguish the methods by examining their parameters. Thus, the terms operation and the method are distinguishable only when there is polymorphism.

36. What is the difference between object-oriented analysis (OOA) and object-oriented design(OOD).

- O**bject-Oriented Analysis (OOA)** refers to a method of developing an initial model of the software from the requirements specification.
- The analysis model is refined into a design model.
- The design model can be implemented using a programming language.

- The term object-oriented programming refers to the implementation of programs using object-oriented concepts.

- O**bject-Oriented Design (OOD)** paradigm suggests that the natural objects (i.e. the entities) occurring in a problem should be identified first and then implemented.
- It is not only identify objects but also identify the internal details of these identified objects.
- The relationships existing among different objects are identified and represented in such a way that the objects can be easily implemented using a programming language.

37. Why design patterns are important in creating good software design?

Design patterns are reusable solutions to problems that recur in many applications. A pattern serves as a guide for creating a "good" design. Patterns are based on sound common sense and the

application of fundamental design principles. These are created by people who spot repeating themes across designs. The pattern solutions are typically described in terms of class and interaction diagrams.

38. Define design patterns

Design patterns are very useful in creating good software design solutions. In addition to providing the model of a good solution, design patterns include a clear specification of the problem, and also explain the circumstances in which the solution would and would not work.

39. Name some of the design pattern

- Expert pattern
- Creator pattern
- Controller pattern

40. What are the four important parts in design pattern

- The problem.
- The context in which the problem occurs.
- The solution.
- The context within which the solution works.

41. What are the advantages of using design patterns?

- Design patterns are reusable solutions to problems that recur in many applications.
- A pattern serves as a guide for creating a "good" design.
- Patterns are based on sound common sense and the application of fundamental design principles. Patterns are created by people who spot repeating themes across designs.
- The pattern solutions are typically described in terms of class and interaction diagrams.

42. How design pattern solutions described.

The design pattern solutions are typically described in terms of both class and interaction diagrams

43. Define expert pattern.

The class should be responsible for doing certain things for which it has the necessary information.

44. Define creator pattern.

The class should be responsible for creating a new instance of some class.

45. Define controller pattern.

The class should be responsible for handling the actor requests.

46. What is facade pattern?

A facade pattern tells how should the services be requested from a service package and also model view separation model tells the way that non-GUI classes should communicate with the GUI classes.

47. List the objects identified during domain analysis.

The objects identified during domain analysis can be classified into

- Boundary objects
- Controller objects
- Entity objects

48. What are boundary objects?

The objects with which the actors interact are known as boundary objects.

49. What are entity objects?

The objects which are responsible for storing data, fetching data, and doing some fundamental kinds of operation that do not change often are known as entity objects.

50. What is the most critical part in domain modelling activity?

The most critical part of the domain modelling activity is to identify entity objects.

51. Define controller objects.

The objects which effectively decouple the boundary and entity objects from one another making the system tolerant to changes of the user interface and processing logic are controller objects

52. Is the use cases should not be tightly tied to the GUI, why?

Yes, the use cases should not be too tightly tied to the GUI.

For example, the use cases should not make any reference to the type of the GUI element appearing on the screen, e.g. radioButton, pushButton, etc. This is necessary because, the type of the user interface component used may change frequently. However, the functionalities do not change so often.

53. What are the responsibilities assigned to a controller object?

The responsibilities assigned to a controller object are closely related to the realization of a specific use case. Normally, each use case is realized using one controller object. More complex use cases may require more than one controller object to realize the use case. A complex use case can have several controller objects such as transaction manager, resource coordinator, and error handler.

54. Why the entity objects are not responsible for implementing the business logic?

Entity objects normally hold information such as data tables and files that need to outlive use case execution, e.g. Book, BookRegister, LibraryMember, etc. Many of the entity objects are “dumb

Servers". They are normally responsible for storing data, fetching data, and doing some fundamental kinds of operation that do not change often. The controller objects are responsible for implementing the business logic.

55. Write a short note on CRC.

- CRC (Class-Responsibility-Collaborator) cards are index cards that are prepared one per each class. On each of these cards, the responsibility of each class is written briefly.
- The objects with which this object needs to collaborate its responsibility are also written.
- Once we assign the responsibility to classes using CRC cards, then we can develop the interaction diagrams by flipping through the CRC cards.
- The CRC cards help determining the methods to be supported by different classes and the interaction among the classes.

56. What is meant by domain modelling?

Domain modelling is known as conceptual modelling. A domain model is a representation of the concepts or objects appearing in the problem domain. It also captures the obvious relationships among these objects.

Examples of such conceptual objects are the Book, BookRegister, MemeberRegister, LibraryMember, etc.

57. What are the three types of objects identified during domain analysis?

The objects identified during domain analysis can be classified into three types:

- Boundary objects
- Controller objects
- Entity objects

The boundary and controller objects can be systematically identified from the use case diagram whereas identification of entity objects requires practice. So, the core of the domain modelling activity is to identify the entity models.

58. What are the two goals of interaction modelling.

The primary goals of interaction modelling are the following:

- To allocate the responsibility of a use case realization among the boundary, entity, and controller objects. The responsibilities for each class are reflected as an operation to be supported by that class.
- To show the detailed interaction that occurs over time among the objects associated with each use case.

59. Define cohesion in OOD.

Cohesion is a measure of functional strength of a module.

60. What are the levels of cohesion in OOD.

In OOD, cohesion is about three levels:

- Cohesiveness of the individual methods.
- Cohesiveness of the data and methods within a class.
- Cohesiveness of an entire class hierarchy.

61.Define each level of cohesion in OOD

- **Cohesiveness of the individual metho**ds - Cohesiveness of each of the individual method is desirable, since it assumes that each method does only a well-defined function.
- **Cohesiveness of the data and methods within a class** -This is desirable since it assures that the methods of an object do actions for which the object is naturally responsible, i.e. it assures that no action has been improperly mapped to an object.
- **Cohesiveness of an entire class hierarchy** - Cohesiveness of methods within a class is desirable since it promotes encapsulation of the objects.

62. Which class is not a good OOD.

The height of the inheritance tree should not be very large. The deeper a class is in the class inheritance hierarchy, the greater is the number of methods it is likely to inherit, making the design more complex.

63. What is good OOD

Cohesiveness of the data and methods within a class is a sign of good OOD.

Cohesiveness of the data and the methods within a class is desirable since it assures that the methods of an object do actions for which the object is naturally responsible, i.e. it assures that no action has been improperly mapped to an object.

64.What is Complex message protocols

Complex message protocols are an indication of excessive coupling among objects. We also know that excessive coupling among objects are not desirable for a good OOD. If a message requires more than 3 parameters, then it is an indication of bad design.

11 MARKS

1. WRITE A SHORT NOTE ON OBJECT ORIENTED CONCEPTS

UML can be described as the successor of object oriented analysis and design.

An object contains both data and methods that control the data. The data represents the state of the object. A class describes an object and they also form hierarchy to model real world system. The hierarchy is represented as inheritance and the classes can also be associated in different manners as per the requirement.

The objects are the real world entities that exist around us and the basic concepts like abstraction, encapsulation, inheritance, polymorphism all can be represented using UML.

So UML is powerful enough to represent all the concepts exists in object oriented analysis and design. UML diagrams are representation of object oriented concepts only. So before learning UML, it becomes important to understand OO concepts in details.

Following are some fundamental concepts of object oriented world:

- **Objects:** Objects represent an entity and the basic building block.
- **Class:** Class is the blue print of an object.

- **Abstraction:** Abstraction represents the behavior of an real world entity.
- **Encapsulation:** Encapsulation is the mechanism of binding the data together and hiding them from outside world.
- **Inheritance:** Inheritance is the mechanism of making new classes from existing one.
- **Polymorphism:** It defines the mechanism to exists in different forms.

2. DEFINE UML? EXPLAIN ITS GOALS AND ROLE IN OO DESIGN.UML

UML is a standard language for specifying, visualizing, constructing, and documenting the artifacts of software systems. UML was created by Object Management Group (OMG) and UML 1.0 specification draft was proposed to the OMG in January 1997.

OMG is continuously putting effort to make a truly industry standard.

- UML stands for **U**nified **M**odeling **L**anguage.
- UML is different from the other common programming languages like C++, Java, COBOL etc.
- UML is a pictorial language used to make software blue prints.

UML can be described as a general purpose visual modeling language to visualize, specify, construct and document software system.UML is generally used to model software systems but it is not limited within this boundary. It is also used to model non software systems as well like process flow in a manufacturing unit etc.

UML is not a programming language but tools can be used to generate code in various languages using UML diagrams. UML has a direct relation with object oriented analysis and design. After some standardization UML is become an OMG (Object Management Group) standard.

GOALS OF UML

- A picture is worth a thousand words, this absolutely fits while discussing about UML.
- Object oriented concepts were introduced much earlier than UML.
- So at that time there were no standard methodologies to organize and consolidate the object oriented development. At that point of time UML came into picture.
- There are a number of goals for developing UML but the most important is to define some general purpose modeling language which all modelers can use and also it needs to be made simple to understand and use.
- UML diagrams are not only made for developers but also for business users, common people and anybody interested to understand the system. The system can be a software or non software.
- So it must be clear that UML is not a development method rather it accompanies with processes to make a successful system.
- At the conclusion the goal of UML can be defined as a simple modeling mechanism to model all possible practical systems in today.s complex environment.

ROLE OF UML IN OO DESIGN:

- UML is a modeling language used to model software and non software systems. Although UML is used for non software systems the emphasis is on modeling object oriented software applications.
- Most of the UML diagrams discussed so far are used to model different aspects like static, dynamic etc. Now what ever be the aspect the artifacts are nothing but objects.
- If we look into class diagram, object diagram, collaboration diagram, interaction diagrams all would basically be designed based on the objects.
- So the relation between OO design and UML is very important to understand. The OO design is transformed into UML diagrams according to the requirement.

- Before understanding the UML in details the OO concepts should be learned properly. Once the OO analysis and design is done the next step is very easy.
- The input from the OO analysis and design is the input to the UML diagrams.

3. EXPLAIN ABOUT UML BUILDING BLOCKS

UML describes the real time systems it is very important to make a conceptual model and then proceed gradually. Conceptual model of UML can be mastered by learning the following three major elements:

- UML building blocks
- Rules to connect the building blocks
- Common mechanisms of UML

The building blocks of UML can be defined as:

- Things
- Relationships
- Diagrams

(1)THINGS:

Things are the most important building blocks of UML. Things can be:

- Structural
- Behavioral
- Grouping
- Annotational

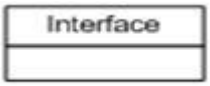
STRUCTURAL THINGS

The Structural things define the static part of the model. They represent physical and conceptual elements.

Following are the brief descriptions of the structural things.

a. Class: Class represents set of objects having similar responsibilities.

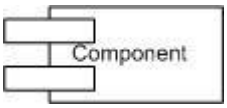
b. Interface: Interface defines a set of operations which specify the responsibility of a class.



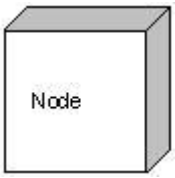
c. Collaboration: Collaboration defines interaction between elements.

d. Use case: Use case represents a set of actions performed by a system for a specific goal.

e. Component: Component describes physical part of a system.



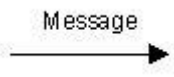
f. Node: A node can be defined as a physical element that exists at run time.



BEHAVIORAL THINGS

A behavioral thing consists of the dynamic parts of UML models. Following are the behavioral things:

a. Interaction: Interaction is defined as a behavior that consists of a group of messages exchanged among elements to accomplish a specific task.

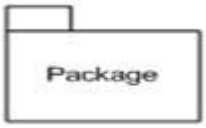


b. State machine: State machine is useful when the state of an object in its life cycle is important. It defines the sequence of states an object goes through in response to events. Events are external factors responsible for state change.

GROUPING THINGS

Grouping things can be defined as a mechanism to group elements of a UML model together. There is only one grouping thing available:

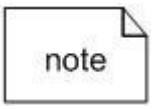
a. Package: Package is the only one grouping thing available for gathering structural and behavioral things.



ANNOTATIONAL THINGS

Annotational things can be defined as a mechanism to capture remarks, descriptions, and comments of UML model elements. **Note** is the only one Annotational thing available.

a. Note: A note is used to render comments, constraints etc of an UML element.

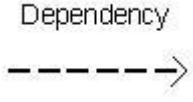


(2) RELATIONSHIP

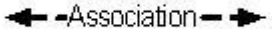
Relationship is another most important building block of UML. It shows how elements are associated with each other and this association describes the functionality of an application.

There are four kinds of relationships available.

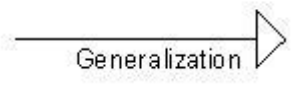
a. Dependency: Dependency is a relationship between two things in which change in one element also affects the other one.



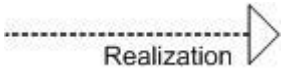
b. Association: Association is basically a set of links that connects elements of an UML model. It also describes how many objects are taking part in that relationship.



c. Generalization: Generalization can be defined as a relationship which connects a specialized element with a generalized element. It basically describes inheritance relationship in the world of objects.



d. Realization: Realization can be defined as a relationship in which two elements are connected. One element describes some responsibility which is not implemented and the other one implements them. This relationship exists in case of interfaces.



(3) UML DIAGRAMS

UML diagrams are the ultimate output of the entire discussion. All the elements, relationships are used to make a complete UML diagram and the diagram represents a system.The visual effect of the UML diagram is the most important part of the entire process. All the other elements are used to make it a complete one.

UML includes the following nine diagrams

1. Class diagram
2. Object diagram
3. Use case diagram
4. Sequence diagram
5. Collaboration diagram
6. Activity diagram
7. Statechart diagram
8. Deployment diagram
9. Component diagram

4. EXPLAIN THE DIFFERENT TYPES OF VIEWS OF A SYSTEM BY UML DIAGRAMS.

UML can be used to construct nine different types of diagrams to capture five different views of a system. Different UML diagrams provide different perspectives of the software system to be developed and facilitate a comprehensive understanding of the system. Such models can be refined to get the actual implementation of the system.

The UML diagrams can capture the following five views of a system

- User's view
- Structural view
- Behavioral view
- Implementation view
- Environmental view

USER'S VIEW

User's view defines the functionalities (facilities) made available by the system to its users.

- It captures the external users' view of the system in terms of the functionalities offered by the system.
- It is a black-box view of the system where the internal structure, the dynamic behavior of different system components, the implementation etc. are not visible.
- It is very different from all other views in the sense that it is a functional model compared to the object model of all other views.
- It can be considered as the central view and all other views are expected to conform to this view. This thinking is in fact the crux of any user centric development style.

STRUCTURAL VIEW

The structural view defines the kinds of objects (classes) important to the understanding of the working of a system and to its implementation.

- It also captures the relationships among the classes (objects).
- The structural model is also called the static model, since the structure of a system does not change with time.

BEHAVIORAL VIEW

The behavioral view captures how objects interact with each other to realize the system behavior. The system behavior captures the time-dependent (dynamic) behavior of the system.

IMPLEMENTATION VIEW

The implementation view captures the important components of the system and their dependencies.

ENVIRONMENTAL VIEW

The environmental models how the different components are implemented on different pieces of hardware.

5. EXPLAIN USE CASE MODEL IN DETAIL

USE CASE MODEL

- Model a system the most important aspect is to capture the dynamic behaviour.
- Dynamic behaviour means the behaviour of the system when it is running /operating.
- Static behaviour is not sufficient to model a system

In UML there are five diagrams available to model dynamic nature and use case diagram is one of them.The use case diagram is dynamic in nature there should be some internal or external factors for making the interaction.

These internal and external agents are known as actors. Use case diagrams are consists of actors, use cases and their relationships. The diagram is used to model the system/subsystem of an application. A single use case diagram captures a particular functionality of a system. So to model the entire system numbers of use case diagrams are used.

PURPOSE OF USE CASE MODEL

The purpose of use case diagram is to capture the dynamic aspect of a system. The other four diagrams (activity, sequence, collaboration and Statechart) are also having the same purpose. So we will look into some specific purpose which will distinguish it from other four diagrams.

Use case diagrams are used to gather the requirements of a system including internal and external influences. These requirements are mostly design requirements. So when a system is analyzed to gather its functionalities use cases are prepared and actors are identified.Now when the initial task is complete use case diagrams are modelled to present the outside view.

So in brief, the purposes of use case diagrams can be as follows:

- Used to gather requirements of a system.
- Used to get an outside view of a system.
- Identify external and internal factors influencing the system.
- Show the interacting among the requirements are actors.

HOW TO DRAW USE CASE DIAGRAM?

Use case diagrams are considered for high level requirement analysis of a system. So when the requirements of a system are analyzed the functionalities are captured in use cases.

Use cases are the system functionalities written in an organized manner. Use cases are relevant to the actors. Actors are interacts with the system.The actors can be human user, some internal applications or external applications. when we are planning to draw an use case diagram we should have the following items identified.

- Functionalities to be represented as an use case
- Actors
- Relationships among the use cases and actors.

Use case diagrams are drawn to capture the functional requirements of a system. So after identifying the above items we have to follow the following guidelines to draw an efficient use case diagram.

- The name of a use case is very important. The name is used to identify the functionalities performed in the system.
- Give a suitable name for actors.
- Show relationships and dependencies clearly in the diagram.

- Do not try to include all types of relationships. Because the main purpose of the diagram is to identify requirements.
- Use note when ever required to clarify some important points.

The following is a sample use case diagram representing the **order management system.** In this diagram, we will find three use cases (Order, SpecialOrder and NormalOrder) and one actor which is customer.

The *SpecialOrder* and *NormalOrder* use cases are extended from *Order* use case. So they have extends relationship. Another important issue is to identify the system boundary which is shown in the picture. The actor *Customer* lies outside the system as it is an external user of the system.

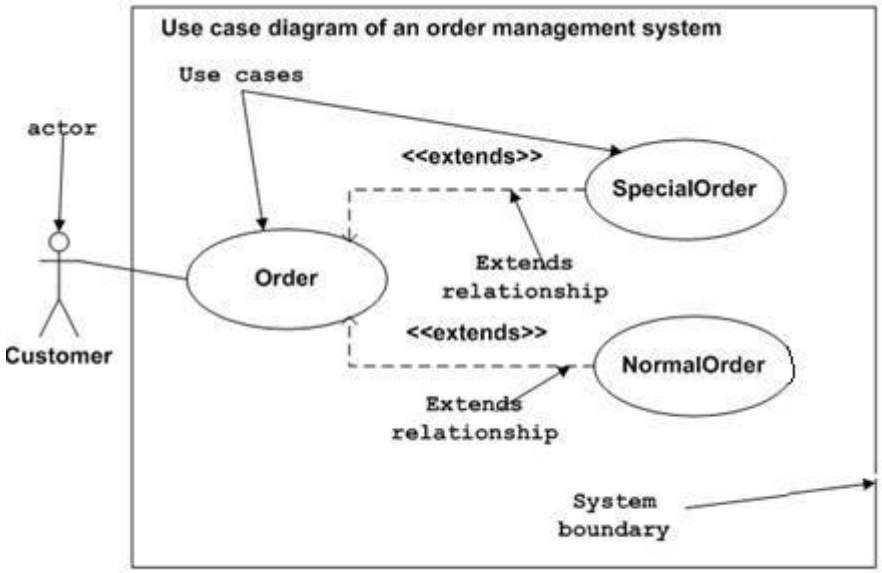


Figure: Sample Use Case diagram

WHERE TO USE CASE DIAGRAMS?

- The five UML diagrams are used to model dynamic view of a system. Now each and every model has some specific purpose to use.
- To understand the dynamics of a system we need to use different types of diagrams. Use case diagram is one of them and its specific purpose is to gather system requirements and actors.
- Use case diagrams specify the events of a system and their flows. But use case diagram never describes how they are implemented. Use case diagram can be imagined as a black box where only the input, output and the function of the black box is known.
- These diagrams are used at a very high level of design. Then this high level design is refined again and again to get a complete and practical picture of the system.
- A well structured use case also describes the pre condition, post condition, exceptions. And these extra elements are used to make test cases when performing the testing.
- Although the use cases are not a good candidate for forward and reverse engineering but still they are used in a slight different way to make forward and reverse engineering.

- And the same is true for reverse engineering. Still use case diagram is used differently to make it a candidate for reverse engineering.
- In forward engineering use case diagrams are used to make test cases and in reverse engineering use cases are used to prepare the requirement details from the existing application.

- So the following are the places where use case diagrams are used:

    - Requirement analysis and high level design.
    - Model the context of a system.
    - Reverse engineering.
    - Forward engineering.

6. DISCUSS ABOUT CLASS DIAGRAM

CLASS DIAGRAMS

A class diagram describes the static structure of a system. It shows how a system is structured rather than how it behaves. The static structure of a system comprises of a number of class diagrams and their dependencies. The main constituents of a class diagram are classes and their relationships:generalization, aggregation, association, and various kinds of dependencies.

CLASSES

The classes represent entities with common features, i.e. attributes and operations. Classes are represented as solid outline rectangles with compartments. Classes have a mandatory name compartment where the name is written centered in boldface. The class name is usually written using mixed case convention and begins with an uppercase. The class names are usually chosen to be singular nouns.

Classes have optional attributes and operations compartments. A class may appear on several diagrams. Its attributes and operations are suppressed on all but one diagram.

(i) ATTRIBUTES

An attribute is a named property of a class. It represents the kind of data that an object might contain. Attributes are listed with their names, and may optionally contain specification of their type, an initial value, and constraints. The type of the attribute is written by appending a colon and the type name after the attribute name. Typically, the first letter of a class name is a small letter. An example for an attribute is given,

bookName : String

(ii) OPERATION

Operation is the implementation of a service that can be requested from any object of the class to affect behaviour. An object's data or state can be changed by invoking an operation of the

object. A class may have any number of operations or no operation at all. Typically, the first letter of an operation name is a small letter. Abstract operations are written in italics. The parameters of an operation (if any), may have a kind specified, which may be 'in', 'out' or 'inout'. An operation may have a return type consisting of a single return type expression.
An example for an operation is given,

issueBook(in bookName):Boolean

PURPOSE OF CLASS DIAGRAM

The purpose of the class diagram is to model the static view of an application. The class diagrams are the only diagrams which can be directly mapped with object oriented languages and thus widely used at the time of construction.

The UML diagrams like activity diagram, sequence diagram can only give the sequence flow of the application but class diagram is a bit different. So it is the most popular UML diagram in the coder community.

So the purpose of the class diagram can be summarized as:

- Analysis and design of the static view of an application.
- Describe responsibilities of a system.
- Base for component and deployment diagrams.
- Forward and reverse engineering.

HOW TO DRAW CLASS DIAGRAM?

Class diagrams are the most popular UML diagrams used for construction of software applications. So it is essential to learn the drawing procedure of class diagram.Class diagrams have lot of properties to consider while drawing but here the diagram will be considered from a top level view.

Class diagram is basically a graphical representation of the static view of the system and represents different aspects of the application. So a collection of class diagrams represent the whole system.

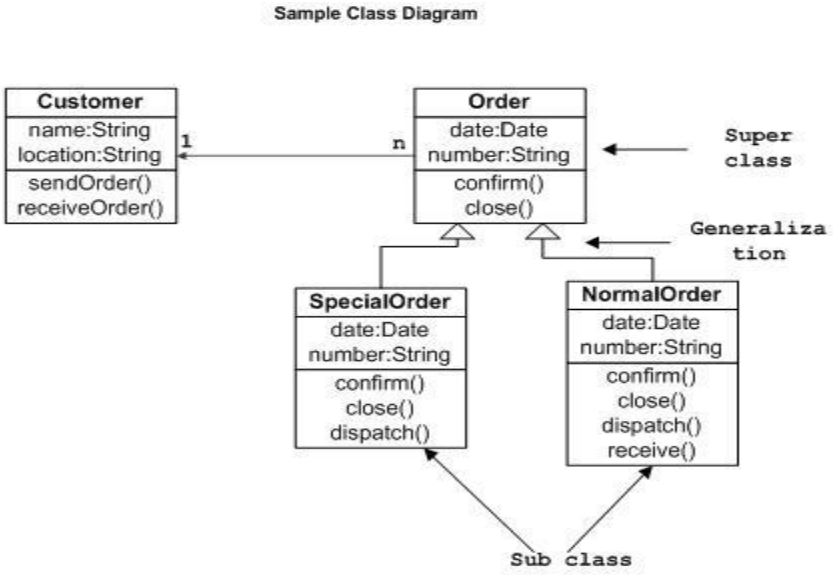
The following points should be remembered while drawing a class diagram:

- The name of the class diagram should be meaningful to describe the aspect of the system.
- Each element and their relationships should be identified in advance.
- Responsibility (attributes and methods) of each class should be clearly identified.
- For each class minimum number of properties should be specified. Because unnecessary properties will make the diagram complicated.
- Use notes when ever required to describe some aspect of the diagram. Because at the end of the drawing it should be understandable to the developer/coder.
- Finally, before making the final version, the diagram should be drawn on plain paper and rework as many times as possible to make it correct.

Now the following diagram is an example of an *Order System* of an application. So it describes a particular aspect of the entire application.

- First of all *Order* and *Customer* are identified as the two elements of the system and they have a *one to many* relationship because a customer can have multiple orders.
- We would keep *Order* class is an abstract class and it has two concrete classes (inheritance relationship) *SpecialOrder* and *NormalOrder*.
- The two inherited classes have all the properties as the *Order* class. In addition they have additional functions like *dispatch ()* and *receive ()*

So the following class diagram has been drawn considering all the points mentioned above:



WHERE TO USE CLASS DIAGRAMS?

Class diagram is a static diagram and it is used to model static view of a system. The static view describes the vocabulary of the system.

Class diagram is also considered as the foundation for component and deployment diagrams. Class diagrams are not only used to visualize the static view of the system but they are also used to construct the executable code for forward and reverse engineering of any system.

Generally UML diagrams are not directly mapped with any object oriented programming languages but the class diagram is an exception.

Class diagram clearly shows the mapping with object oriented languages like Java, C++ etc. So from practical experience class diagram is generally used for construction purpose.

So in a brief, class diagrams are used for:

- Describing the static view of the system.
- Showing the collaboration among the elements of the static view.
- Describing the functionalities performed by the system.
- Construction of software applications using object oriented languages.

7. EXPLAIN INTERACTION DIAGRAM

INTERACTION DIAGRAM

The interaction diagram is used to describe some type of interactions among the different elements in the model. So this interaction is a part of dynamic behaviour of the system.This interactive behaviour is represented in UML by two diagrams known as **Sequence diagram** and **Collaboration diagram**. The basic purposes of both the diagrams are similar.

Sequence diagram emphasizes on time sequence of messages and collaboration diagram emphasizes on the structural organization of the objects that send and receive messages.

PURPOSE OF INTERACTION DIAGRAM

The purposes of interaction diagrams are to visualize the interactive behaviour of the system. Now visualizing interaction is a difficult task. So the solution is to use different types of models to capture the different aspects of the interaction. Sequence and collaboration diagrams are used to capture dynamic nature but from a different angle.

So the purposes of interaction diagram can be describes as:

- To capture dynamic behaviour of a system.
- To describe the message flow in the system.
- To describe structural organization of the objects.
- To describe interaction among objects.

HOW TO DRAW INTERACTION DIAGRAM?

The interaction diagrams are to capture the dynamic aspect of a system. To capture the dynamic aspect we need to understand what a dynamic aspect is and how it is visualized. Dynamic aspect can be defined as the snap shot of the running system at a particular moment.

We have two types of interaction diagrams in UML. One is sequence diagram and the other is a collaboration diagram.

- The sequence diagram captures the time sequence of message flow from one object to another.
- The collaboration diagram describes the organization of objects in a system taking part in the message flow.

So the following things are to identify visibly before drawing the interaction diagram:

- Objects taking part in the interaction.
- Message flows among the objects.
- The sequence in which the messages are flowing.
- Object organization.

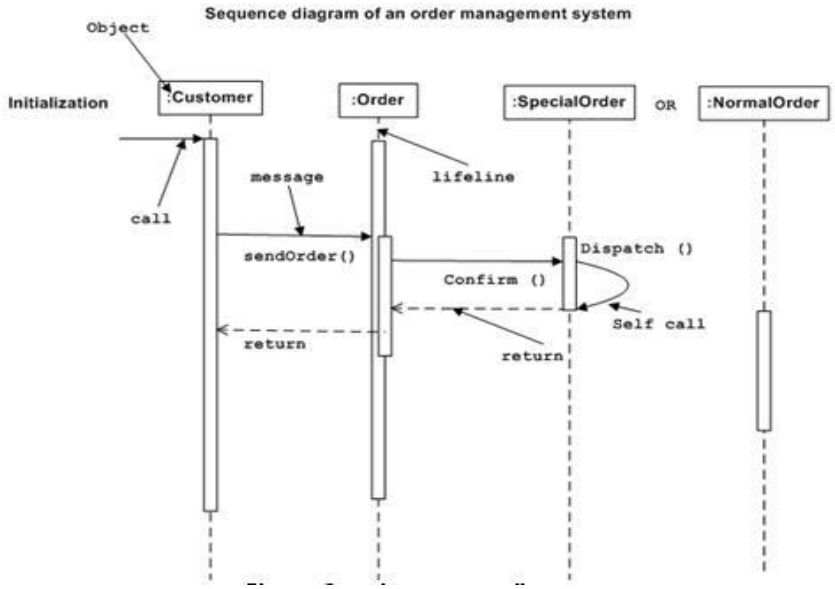
Following are two interaction diagrams modeling **order management system**.

THE SEQUENCE DIAGRAM

The sequence diagram is having four objects (Customer, Order, SpecialOrder and NormalOrder).

The following diagram has shown the message sequence for *SpecialOrder* object and the same can be used in case of *NormalOrder* object. Now it is important to understand the time sequence of message flows. The message flow is nothing but a method call of an object.

The first call is *sendOrder ()* which is a method of *Order* object. The next call is *confirm ()* which is a method of *SpecialOrder* object and the last call is *Dispatch ()* which is a method of*SpecialOrder* object. So here the diagram is mainly describing the method calls from one object to another and this is also the actual scenario when the system is running.

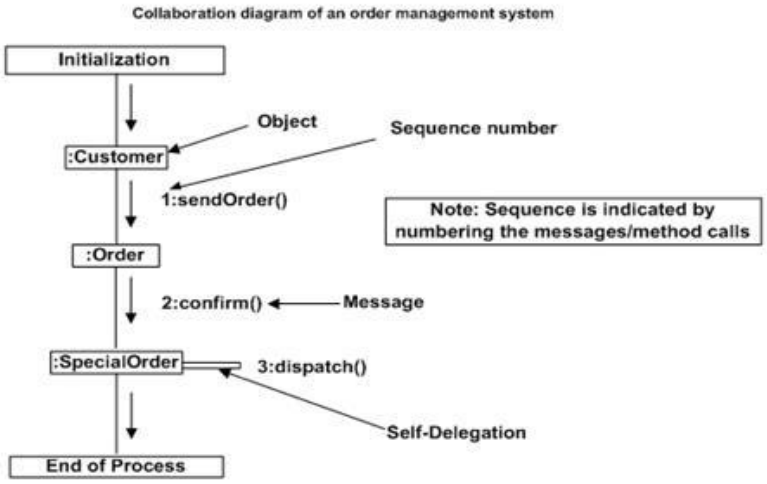


COLLABORATION DIAGRAM

In collaboration diagram the method call sequence is indicated by some numbering technique as shown below. The number indicates how the methods are called one after another. We have taken the same order management system to describe the collaboration diagram.

The method calls are similar to that of a sequence diagram. But the difference is that the sequence diagram does not describe the object organization whereas the collaboration diagram shows the object organization.

Now to choose between these two diagrams the main emphasis is given on the type of requirement. If the time sequence is important then sequence diagram is used and if organization is required then collaboration diagram is used.



WHERE TO USE INTERACTION DIAGRAMS?

The interaction diagrams are used to describe dynamic nature of a system. To understand the practical application of the interaction diagram we need to understand the basic nature of sequence and collaboration diagram.The main purposes of both the diagrams are similar as they are used to capture the dynamic behaviour of a system. But the specific purposes are more important to clarify and understood.

Sequence diagrams are used to capture the order of messages flowing from one object to another. and the collaboration diagrams are used to describe the structural organizations of the objects taking part in the interaction. A single diagram is not sufficient to describe the dynamic aspect of an entire system so a set of diagrams are used to capture is as a whole.

The interaction diagrams are used when we want to understand the message flow and the structural organization. Now message flow means the sequence of control flow from one object to another and structural organization means the visual organization of the elements in a system.

In a brief the following are the usages of interaction diagrams:

- To model flow of control by time sequence.
- To model flow of control by structural organizations.
- For forward engineering.
- For reverse engineering.

8. DISCUSS ABOUT ACTIVITY DIAGRAMS IN DETAIL

ACTIVITY DIAGRAM

Activity diagram is another important diagram in UML to describe dynamic aspects of the system.

Activity diagram is basically a flow chart to represent the flow form one activity to another activity. The activity can be described as an operation of the system.

The control flow is drawn from one operation to another. This flow can be sequential, branched or concurrent. Activity diagrams deals with all type of flow control by using different elements like fork, join etc.

PURPOSE OF ACTIVITY DIAGRAM

The basic purposes of activity diagrams are similar to other four diagrams. It captures the dynamic behaviour of the system. Other four diagrams are used to show the message flow from one object to another but activity diagram is used to show message flow from one activity to another.

Activity is a particular operation of the system. Activity diagrams are not only used for visualizing dynamic nature of a system but they are also used to construct the executable system by using forward and reverse engineering techniques. The only missing thing in activity diagram is the message part.

It does not show any message flow from one activity to another. Activity diagram is some time considered as the flow chart. Although the diagrams looks like a flow chart but it is not. It shows different flow like parallel, branched, concurrent and single.

So the purposes can be described as:

- Draw the activity flow of a system.
- Describe the sequence from one activity to another.
- Describe the parallel, branched and concurrent flow of the system.

HOW TO DRAW ACTIVITY DIAGRAM?

Activity diagrams are mainly used as a flow chart consists of activities performed by the system. But activity diagram are not exactly a flow chart as they have some additional capabilities. These additional capabilities include branching, parallel flow, swim lane etc.

Before drawing an activity diagram we must have a clear understanding about the elements used in activity diagram. The main element of an activity diagram is the activity itself. An activity is a function performed by the system. After identifying the activities we need to understand how they are associated with constraints and conditions.

So before drawing an activity diagram we should identify the following elements:

- Activities
- Association
- Conditions
- Constraints

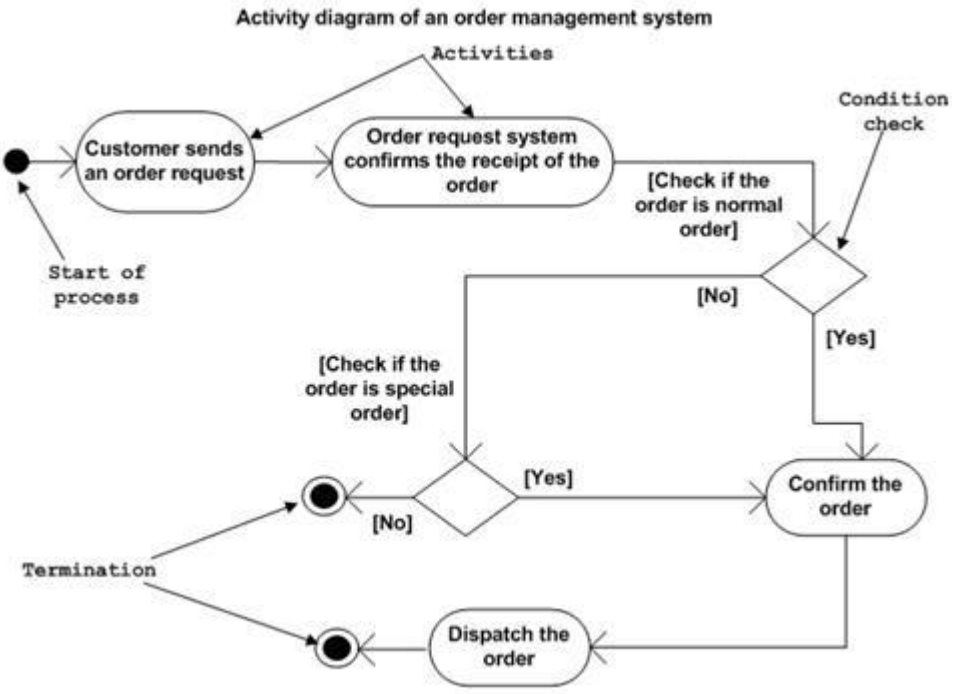
Once the above mentioned parameters are identified we need to make a mental layout of the entire flow. This mental layout is then transformed into an activity diagram.

The following is an example of an activity diagram for order management system. In the diagram four activities are identified which are associated with conditions. One important point should be clearly understood that an activity diagram cannot be exactly matched with the code. The activity diagram is made to understand the flow of activities and mainly used by the business users.

The following diagram is drawn with the four main activities:

- Send order by the customer
- Receipt of the order
- Confirm order
- Dispatch order

After receiving the order request condition checks are performed to check if it is normal or special order. After the type of order is identified dispatch activity is performed and that is marked as the termination of the process.



WHERE TO USE ACTIVITY DIAGRAMS?

The basic usage of activity diagram is similar to other four UML diagrams. The specific usage is to model the control flow from one activity to another. This control flow does not include messages.

The activity diagram is suitable for modeling the activity flow of the system. An application can have multiple systems. Activity diagram also captures these systems and describes flow from one system to another. This specific usage is not available in other diagrams. These systems can be database, external queues or any other system.

It is clear that an activity diagram is drawn from a very high level. So it gives high level view of a system. This high level view is mainly for business users or any other person who is not a technical person.This diagram is used to model the activities which are nothing but business requirements. So the diagram has more impact on business understanding rather implementation details.

Following are the main usages of activity diagram:

- Modeling work flow by using activities.
- Modeling business requirements.

- High level understanding of the system's functionalities.
- Investigate business requirements at a later stage.

9. EXPLAIN STATE CHART DIAGRAMS

STATE CHART DIAGRAM

State chart diagram describes different states of a component in a system. The states are specific to a component/object of a system.A State chart diagram describes a state machine. State machine is defined as a machine with different states of an object and these states are controlled by external or internal events.

PURPOSE OF STATE CHART DIAGRAMS

Statechart diagram is one of the five UML diagrams used to model dynamic nature of a system. They define different states of an object during its lifetime. These states are changed by events. Statechart diagrams are useful to model reactive systems. Reactive systems can be defined as a system that responds to external or internal events.

Statechart diagram describes the flow of control from one state to another state. States are defined as a condition in which an object exists and it changes when some event is triggered. The most important purpose of Statechart diagram is to model life time of an object from creation to termination.

Statechart diagrams are also used for forward and reverse engineering of a system. But the main purpose is to model reactive system.

Following are the main purposes of using Statechart diagrams:

- To model dynamic aspect of a system.
- To model life time of a reactive system.
- To describe different states of an object during its life time.
- Define a state machine to model states of an object.

HOW TO DRAW STATECHART DIAGRAM?

Statechart diagram is used to describe the states of different objects in its life cycle. The emphasis is given on the state changes upon some internal or external events. These states of objects are important to analyze and implement them accurately.

Statechart diagrams are very important for describing the states. States can be identified as the condition of objects when a particular event occurs.

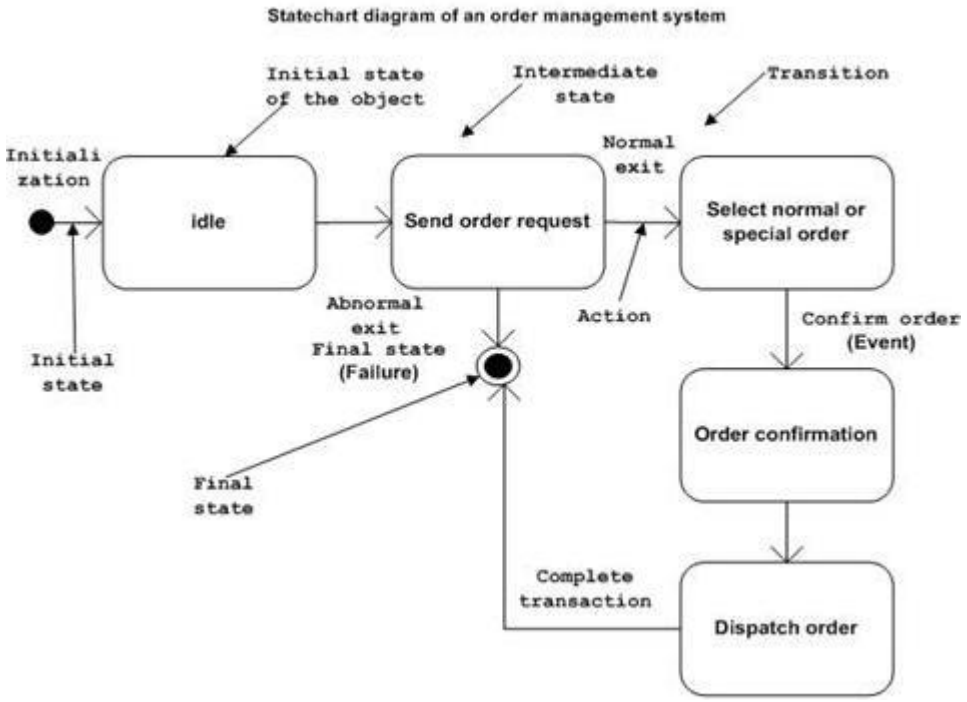
Before drawing a Statechart diagram we must have clarified the following points:

- Identify important objects to be analyzed.
- Identify the states.
- Identify the events.

The following is an example of a Statechart diagram where the state of *Order* object is analyzed.

The first state is an idle state from where the process starts. The next states are arrived for events like *send request*, *confirm request,* and *dispatch order*. These events are responsible for state changes of order object.

During the life cycle of an object (here order object) it goes through the following states and there may be some abnormal exists also. This abnormal exit may occur due to some problem in the system. When the entire life cycle is complete it is considered as the complete transaction as mentioned below. The initial and final state of an object is also shown below.



WHERE TO USE STATECHART DIAGRAMS?

Statechart diagrams are used to model dynamic aspect of a system like other four diagrams discussed in this tutorial. But it has some distinguishing characteristics for modeling dynamic nature.

Statechart diagram defines the states of a component and these state changes are dynamic in nature. So its specific purpose is to define state changes triggered by events. Events are internal or external factors influencing the system.

Statechart diagrams are used to model states and also events operating on the system. When implementing a system it is very important to clarify different states of an object during its life time and statechart diagrams are used for this purpose. When these states and events are identified they are used to model it and these models are used during implementation of the system.

The practical implementation of Statechart diagram then it is mainly used to analyze the object states influenced by events. This analysis is helpful to understand the system behaviour during its execution.

So the main usages can be described as:

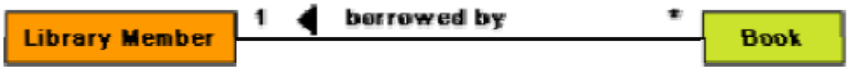
- To model object states of a system.
- To model reactive system. Reactive system consists of reactive objects.
- To identify events responsible for state changes.
- Forward and reverse engineering.

10. EXPLAIN THE THREE RELATIONSHIP AMONG CLASSES? GIVE AN EXAMPLE FOR EACH RELATIONSHIP.

ASSOCIATION

Associations are needed to enable objects to communicate with each other. An association describes a connection between classes. The association relation between two objects is called object connection or link. Links are instances of associations. A link is a physical or conceptual connection between object instances. An association describes a group of links with a common structure and common semantics.

For example, consider the statement that Library Member borrows Books. Here, borrows is the association between the class LibraryMember and the class Book. Usually, an association is a binary relation (between two classes). However, three or more different classes can be involved in an association. A class can have an association relationship with itself (called recursive association). In this case, it is usually assumed that two different objects of the class are linked by the association relationship.



Association between two classes

Association between two classes is represented by drawing a straight line between the concerned classes. In the above figure illustrates the graphical representation of the association relation.

The name of the association is written alongside the association line. An arrowhead may be placed on the association line to indicate the reading direction of the association. The arrowhead should not be misunderstood to be indicating the direction of a pointer implementing an association. On each side of the association relation, the multiplicity is noted as an individual number or as a value range.

The multiplicity indicates how many instances of one class are associated with each other. Value ranges of multiplicity are noted by specifying the minimum and maximum value, separated by two dots, e.g. 1.5. An asterisk is a wild card and means many (zero or more).

The association of the above figure should be read as "Many books may be borrowed by a Library Member". Observe that associations (and links) appear as verbs in the problem statement.

AGGREGATION

Aggregation is a special type of association where the involved classes represent a whole-part relationship. The aggregate takes the responsibility of forwarding messages to the appropriate parts. Thus, the aggregate takes the responsibility of delegation and leadership.

When an instance of one object contains instances of some other objects, then aggregation (or composition) relationship exists between the composite object and the component object. Aggregation is represented by the diamond symbol at the composite end of a relationship.

Representation of Aggregation

A document may consist of several paragraphs and each paragraph consists of many lines. Aggregation is represented by the diamond symbol (in the above figure) at the composite end of a relationship.

COMPOSITION

Composition is a stricter form of aggregation, in which the parts are existence-dependent on the whole. This means that the life of the parts closely ties to the life of the whole. When the whole is created, the parts are created and when the whole is destroyed, the parts are destroyed.

A typical example of composition is an invoice object with invoice items. As soon as the invoice object is created, all the invoice items in it are created and as soon as the invoice object is destroyed, all invoice items in it are also destroyed. The composition relationship is represented as a filled diamond drawn at the composite-end.

An example of the composition relationship is shown in figure

Representation of composition

11. WRITE DOWN SOME POPULAR DESIGN PATTERNS AND THEIR NECESSITIES

DESIGN PATTERNS

Design patterns are reusable solutions to problems that recur in many applications. A pattern serves as a guide for creating a "good" design. Patterns are based on sound common sense and the application of fundamental design principles. These are created by people who spot repeating themes across designs. The pattern solutions are typically described in terms of class and interaction diagrams.

In addition to providing the model of a good solution, design patterns include a clear specification of the problem, and also explain the circumstances in which the solution would and would not work. Thus, a design pattern has four important parts:

- The problem
- The context in which the problem occurs
- The solution
- The context within which the solution works

EXPERT PATTERN

The class should be responsible for doing certain things for which it has the necessary information.

EXAMPLE:

Problem: Which class should be responsible for doing certain things?

Solution: Assign responsibility to the information expert – the class that has the information necessary to ful fill the required responsibility. The expert pattern expresses the common intuition that objects do things related to the information they have. The class diagram and collaboration diagrams for this solution to the problem of which class should compute the total sales is shown in the fig. a

(a)Class Diagram

(b)Collaboration Diagram

CREATOR PATTERN

The class should be responsible for creating a new instance of some class.

Problem: Which class should be responsible for creating a new instance of some class?

Solution: Assign a class C1 the responsibility to create an instance of class C2, if one or more of the following are true**:**

- C1 is an aggregation of objects of type C2.
- C1 contains objects of type C2.
- C1 closely uses objects of type C2.
- C1 has the data that would be required to initialize the objects of type C2, when they are created.

CONTROLLER PATTERN:

The class should be responsible for handling the actor requests.

Problem: Who should be responsible for handling the actor requests?

Solution: For every use case, there should be a separate controller object which would be responsible for handling requests from the actor. Also, the same controller should be used for all the actor requests pertaining to one use case so that it becomes possible to maintain the necessary information about the state of the use case. The state information maintained by a controller can be used to identify the out-of-sequence actor requests, e.g. whether voucher request is received before arrange payment request.

FACADE PATTERN:

A facade pattern tells how should the services be requested from a service package and also model view separation model tells the way that non-GUI classes should communicate with the GUI classes.

Problem: How should the services be requested from a service package?

Context in which the problem occurs: A package as already discussed is a cohesive set of classes – the classes have strongly related responsibilities. For example, an RDBMS interface package may contain classes that allow one to perform various operations on the RDBMS.

Solution: A class (such as DBfacade) can be created which provides a common interface to the services of the package.

MODEL VIEW SEPARATION PATTERN:

Domain objects need to communicate with windows to cause a real-time ongoing display update as the state of information in the domain object changes.

Problem: How should the non-GUI classes communicate with the GUI classes?

Context in which the problem occurs: This is a very commonly occurring pattern which occurs in almost every problem. Here, model is a synonym for the domain layer objects, view is a synonym for the presentation layer objects such as the GUI objects.

Solution: The model view separation pattern states that model objects should not have direct knowledge (or be directly coupled) to the view objects. This means that there should not be any direct calls from other objects to the GUI objects. This results in a good solution, because the GUI classes are related to a particular application whereas the other classes may be reused.

There are actually two solutions to this problem which work in different circumstances as follows:

Solution 1: Polling or Pull from above

It is the responsibility of a GUI object to ask for the relevant information from the other objects, i.e. the GUI objects pull the necessary information from the other objects whenever required. This model is frequently used. However, it is inefficient for certain applications.

For example,

Simulation applications which require visualization, the GUI objects would not know when the necessary information becomes available. Other examples are, monitoring applications such as network monitoring, stock market quotes, and so on. In these situations, a "push-from-below" model of display update is required. Since "push-from-below" is not an acceptable solution, an indirect mode of communication from the other objects to the GUI objects is required.

Solution 2: Publish- subscribe pattern

An event notification system is implemented through which the publisher can indirectly notify the subscribers as soon as the necessary information becomes available. An event manager class can be defined which keeps track of the subscribers and the types of events they are interested in. An event is published by the publisher by sending a message to the event manager object.

The event manager notifies all registered subscribers usually via a parameterized message (called a callback). Some languages specifically support event manager classes. For example, Java provides the EventListener interface for such purposes.

INTERMEDIARY PATTERN OR PROXY PATTERN

The interaction between the intermediary function is the encapsulation of objects. If an object operation may cause changes in other related object, and this object does not want to deal with these relationships.

Problem: How should the client and server objects interact with each other?

Context in the problem occurs: The client and server terms as used here refer to software components existing across a network. The clients are consumers of services provided by the servers.

Solution: A proxy object at the client side can be defined which is a local sit-in for the remote server object. The proxy hides the details of the network transmission. The proxy is responsible for determining the server address, communicating the client request to the server, obtaining the server response and seamlessly passing that to the client.

The proxy can also augment (or filter) information that is exchanged between the client and the server. The proxy could have the same interface as the remote server object so that the client

feels as if it is interacting directly with the remote server object and the complexities of network transmissions are abstracted out.

12. EXPLAIN THE PURPOSE OF DIFFERENT TYPES OF OBJECTS IDENTIFIED DURING DOMAIN ANALYSIS. EXPLAIN HOW THESE OBJECTS INTERACT AMONG EACH OTHER.

DOMAIN MODELLING

Domain modelling is known as conceptual modelling. A domain model is a representation of the concepts or objects appearing in the problem domain. It also captures the obvious relationships among these objects.

Examples of such conceptual objects are the Book, BookRegister, MemeberRegister, LibraryMember, etc.

The different kinds of objects identified during domain analysis and their relationships are as follows:

BOUNDARY OBJECTS

- The boundary objects are those with which the actors interact. These include screens, menus, forms, dialogs, etc.
- The boundary objects are mainly responsible for user interaction.They normally do not include any processing logic.
- They may be responsible for validating inputs, formatting, outputs, etc. The boundary objects were earlier being called as the interface objects.
- The term interface class is being used for Java, COM/DCOM, and UML with different meaning. A recommendation for the initial identification of the boundary classes is to define one boundary class per actor/use case pair.

ENTITY OBJECTS

- These normally hold information such as data tables and files that need to outlive use case execution, e.g. Book, BookRegister, LibraryMember, etc. Many of the entity objects are “dumb servers”.

- They are responsible for storing data, fetching data, and doing some fundamental kinds of operation that do not change often.

CONTROLLER OBJECTS

- The controller objects coordinate the activities of a set of entity objects and interface with the boundary objects to provide the overall behavior of the system.
- The responsibilities assigned to a controller object are closely related to the realization of a specific use case.
- The controller objects effectively decouple the boundary and entity objects from one another making the system tolerant to changes of the user interface and processing logic.

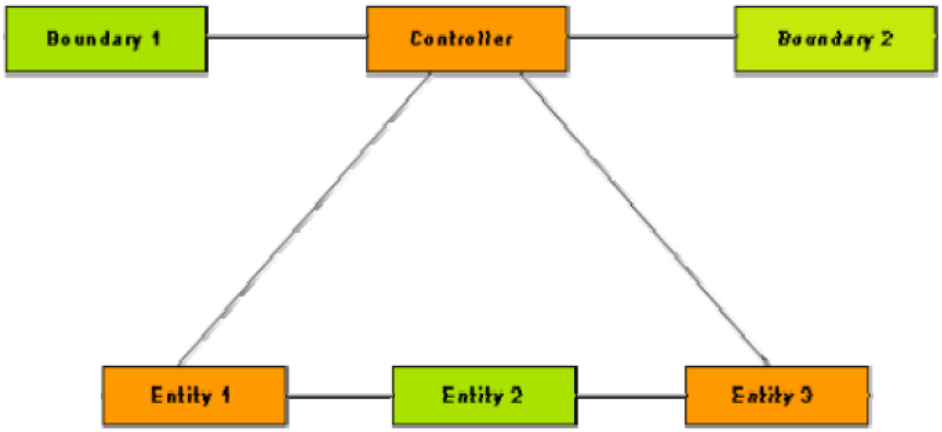
- The controller objects embody most of the logic involved with the use case realization (this logic may change time to time).
- A typical interaction of a controller object with boundary and entity objects is shown in figure below, each use case is realized using one controller object.
- Some use cases can be realized without using any controller object, i.e. through boundary and entity objects only. This is often true for use cases that achieve only some simple manipulation of the stored information.

For example,

Consider the "query book availability" use case of the **Library Information System (LIS).**

Realization of the use case involves only matching the given book name against the books available in the catalog. More complex use cases may require more than one controller object to realize the use case.

A complex use case can have several controller objects such as transaction manager, resource coordinator, and error handler. There is another situation where a use case can have more than one controller object. Sometimes the use cases require the controller object to transit through a number of states. In such cases, one controller object might have to be created for each execution of the use case.



A typical realization of a use case through the collaboration of boundary, controller, and entity objects.

13.EXPLAIN OBJECT-ORIENTED DESIGN METHODOLOGY WITH SUITABLE EXAMPLE?

OBJECT-ORIENTED DESIGN METHODOLOGY

In any object-oriented design methodology one of the most important steps is the identification of objects. In fact, the quality of the final design depends to a great extent on the appropriateness of the objects identified. However, to date no formal methodology exists for identification of objects.

Several semi-formal and informal approaches have been proposed for object identification.

These can be classified into the following broad classes:

- Grammatical analysis of the problem description.
- Derivation from data flow.
- Derivation from the entity relationship (E-R) diagram.

A widely accepted object identification approach is the grammatical analysis approach. Grady Booch originated the grammatical analysis approach [1991]. In Booch's approach, the nouns occurring in the extended problem description statement (processing narrative) are mapped to objects and the verbs are mapped to methods. The identification approaches based on derivation from the data flow diagram and the entity-relationship model are still evolving and therefore will not be discussed in this text.

BOOCH'S OBJECT IDENTIFICATION METHOD

- Booch's object identification approach requires a processing narrative of the given problem to be first developed. The processing narrative describes the problem and discusses how it can be solved.
- The objects are identified by noting down the nouns in the processing narrative. Synonym of a noun must be eliminated.
- If an object is required to implement a solution, then it is said to be part of the solution space. Otherwise, if an object is necessary only to describe the problem, then it is said to be a part of the problem space.
- Several of the nouns may not be objects. An imperative procedure name, i.e., noun form of a verb actually represents an action and should not be considered as an object.

A potential object found after lexical analysis is usually considered legitimate, only if it satisfies the following criteria:

(i) RETAINED INFORMATION

Some information about the object should be remembered for the system to function. If an object does not contain any private data, it cannot be expected to play any important role in the system.

(ii) MULTIPLE ATTRIBUTES

Objects have multiple attributes and support multiple methods. It is very rare to find useful objects which store only a single data element or support only a single method, because an object having only a single data element or method is usually implemented as a part of another object.

(iii) COMMON OPERATIONS

A set of operations can be defined for potential objects. If these operations apply to all occurrences of the object, then a class can be defined. An attribute or operation defined for a class must apply to each instance of the class. If some of the attributes or operations apply only to some specific instances of the class, then one or more subclasses can be needed for these special objects.

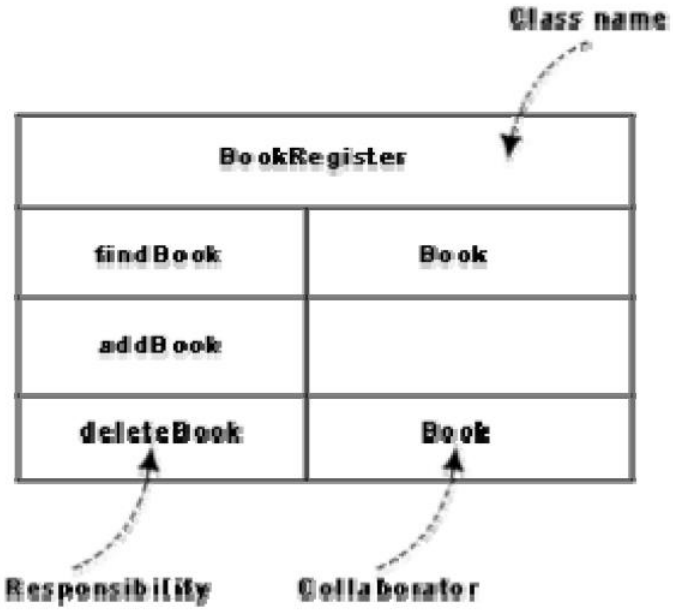
The actors themselves and the interactions among themselves should be excluded from the entity identification exercise. Sometimes there is a need to maintain information about an actor within the system. This is not the same as modelling the actor. These classes are sometimes called surrogates.

NECESSITY OF CRC (CLASS-RESPONSIBILITY-COLLABORATOR)

The interactions diagrams for only simple use cases that involve collaboration among a limited number of classes can be drawn from an inspection of the use case description. More complex use cases require the use of CRC cards where a number of team members participate to determine the responsibility of the classes involved in the use case realization.

CRC (Class-Responsibility-Collaborator) technology was pioneered by Ward Cunningham and Kent Becka at the research laboratory of Tektronix at Portland, Oregon, USA. CRC cards are index cards that are prepared one per each class. On each of these cards, the responsibility of each class is written briefly. The objects with which this object needs to collaborate its responsibility are also written.

CRC cards are usually developed in small group sessions where people role play being various classes. Each person holds the CRC card of the classes he is playing the role of. The cards are deliberately made small (4 inch ´ 6 inch) so that each class can have only limited number of responsibilities. A responsibility is the high level description of the part that a class needs to play in the realization of a use case.



CRC card for the BookRegister class

An example CRC card for the BookRegister class of the Library Automation System is shown in the above figure After assigning the responsibility to classes using CRC cards, it is easier to develop the interaction diagrams by flipping through the CRC cards.

For example, in the **Library Information System (LIS)** we would need to store information about each library member. This is independent of the fact that the library member also plays the role of an actor of the system. Although the grammatical approach is simple and intuitively appealing, yet through a naive use of the approach, it is very difficult to achieve high quality results. In particular, it is very difficult to come up with useful abstractions simply by doing grammatical analysis of the problem description. Useful abstractions usually result from clever factoring of the problem description into independent and intuitively correct elements.

EXAMPLE

TIC-TAC-TOE

Identify the entity objects of the following Tic-tac-toe software

1) Tic-tac-toe is a computer game in which a human player and the computer make alternative moves on a 3 X 3 square.
2) A move consists of marking a previously unmarked square.
3) A player who first places three consecutive marks along a straight line (i.e., along a row, column, or diagonal) on the square wins.
4) As soon as either the human player or the computer wins, a message congratulating the winner should be displayed. If both players manages to get three consecutive marks along a straight line, and all the squares on the board are filled up, then the game is drawn. The computer always tries to win a game.

By performing a grammatical analysis of this problem statement, it can be seen that nouns that have been underlined in the problem description and the actions as the italicized verbs. On closer examination synonyms can be eliminated from the identified nouns.

The lists of nouns after eliminating the synonyms are the following

- Tic-tac-toe, computer game, human player, move, square, mark, straight line, board, row, column, and diagonal. From this list of possible objects, nouns can be eliminated like human player as it does not belong to the problem domain.
- Also, the nouns square, game, computer, Tic-tac-toe, straight line, row, column, and diagonal can be eliminated, as any data and methods cannot be associated with them.
- The noun move can also be eliminated from the list of potential objects since it is an imperative verb and actually represents an action. Thus, there is only one object left – board.
- After experienced in object identification, it is not normally necessary to really identify all nouns in the problem description by underlining them or actually listing them down, and systematically eliminate the non-objects to arrive at the final set of objects.

14. WHAT ARE THE FIVE IMPORTANT CRITERIA FOR JUDGING THE GOODNESS OF ANOBJECT-ORIENTED DESIGN.

It is quite obvious that there are several subjective judgments involved in arriving at a good object-oriented design. Therefore, several alternative design solutions to the same problem are possible. In order to be able to determine which of any two designs is better, some criteria for judging the goodness of a design must be identified.

The following are some of the accepted criteria for judging the goodness of a design.

1) COUPLING GUIDELINES

The number of messages between two objects or among a group of objects should be minimum. Excessive coupling between objects is determined to modular design and prevents reuse.

2) COHESION GUIDELINE

In OOD, cohesion is about three levels:

(i) Cohesiveness of the individual methods: Cohesiveness of each of the individual method is desirable, since it assumes that each method does only a well-defined function.

(ii) Cohesiveness of the data and methods within a class: This is desirable since it assures that the methods of an object do actions for which the object is naturally responsible, i.e. it assures that no action has been improperly mapped to an object.

(iii) Cohesiveness of an entire class hierarchy: Cohesiveness of methods within a class is desirable since it promotes encapsulation of the objects.

3) Hierarchy and factoring guidelines: A base class should not have too many subclasses. If too many subclasses are derived from a single base class, then it becomes difficult to understand the design. In fact, there should approximately be no more than 7±2 classes derived from a base class at any level.

4) Keeping message protocols simple: Complex message protocols are an indication of excessive coupling among objects. If a message requires more than 3 parameters, then it is an indication of bad design.

5) Number of Methods: Objects with a large number of methods are likely to be more application-specific and also difficult to comprehend – limiting the possibility of their reuse. Therefore, objects should not have too many methods. This is a measure of the complexity of a class. It is likely that the classes having more than about seven methods would have problems.

6) Depth of the inheritance tree: The deeper a class is in the class inheritance hierarchy, the greater is the number of methods it is likely to inherit, making it more complex. Therefore, the height of the inheritance tree should not be very large.

7) Number of messages per use case: If methods of a large number of objects are invoked in a chain action in response to a single message, testing and debugging of the objects becomes complicated. Therefore, a single message should not result in excessive message generation and transmission in a system.

8) Response for a class: This is a measure of the maximum number of methods that an instance of this class would call. If the same method is called more than once, then it is counted only once. A class which calls more than about seven different methods is susceptible to errors.