CS 2203 **OBJECT-ORIENTED PROGRAMMING** **3 0 0 3**

(Common to CSE & IT)

Aim: To understand the concepts of object-oriented programming and master OOP using C++.

UNIT I **9**

Object oriented programming concepts – objects – classes – methods and messages – abstraction and encapsulation – inheritance – abstract classes – polymorphism.

Introduction to C++ – classes – access specifiers – function and data members – default arguments – function overloading – friend functions – const and volatile functions - static members – Objects – pointers and objects – constant objects – nested classes – local classes

UNIT II **9**

Constructors – default constructor – Parameterized constructors – Constructor with dynamic allocation – copy constructor – destructors – operator overloading – overloading through friend functions – overloading the assignment operator – type conversion – explicit constructor

UNIT III **9**

Function and class templates - Exception handling – try-catch-throw paradigm – exception specification – terminate and unexpected functions – Uncaught exception.

UNIT IV **9**

Inheritance – public, private, and protected derivations – multiple inheritance - virtual base class – abstract class – composite objects Runtime polymorphism – virtual functions – pure virtual functions – RTTI – typeid – dynamic casting – RTTI and templates – cross casting – down casting .

UNIT V **9**

Streams and formatted I/O – I/O manipulators - file handling – random access – object serialization – namespaces - std namespace – ANSI String Objects – standard template library.

Total: 45

TEXT BOOKS:

1. B. Trivedi, “Programming with ANSI C++”, Oxford University Press, 2007.

REFERENCES:

Ira Pohl, “Object Oriented Programming using C++”, Pearson Education, Second Edition Reprint 2004..
S. B. Lippman, Josee Lajoie, Barbara E. Moo, “C++ Primer”, Fourth Edition, Pearson Education, 2005.
B. Stroustrup, “The C++ Programming language”, Third edition, Pearson Education, 2004.

UNIT I

9

Object oriented programming concepts – objects – classes – methods and messages – abstraction and encapsulation – inheritance – abstract classes – polymorphism.

Introduction to C++ – classes – access specifiers – function and data members – default arguments – function overloading – friend functions – const and volatile functions - static members – Objects – pointers and objects – constant objects – nested classes – local classes

PART A — (10 x 2 20 marks)

1. **What is the difference between a local variable and a data member?(Nov/dec 2011)**
   Classes use the concept of abstraction and are defined as a list of abstract attributes such as size, weight, and cost and uses functions to operate on these attributes.
   The attributes are sometimes called as data members because they hold information.
2. **Define function overloading? (Nov/dec 2011)**
   Function overloading means we can use the same function name to create functions that perform a variety of different tasks.
   Eg: An overloaded add ( ) function handles different data types as shown below.
   // Declarations
   i. int add(int a, int b); //add function with 2 arguments of same type
   ii. int add(int a, int b, int c); //add function with 3 arguments of same type
   iii. double add(int p, double q); //add function with 2 arguments of different type
   //Function calls
   add (3 , 4); //uses prototype (i.)
   add (3, 4, 5); //uses prototype (ii.)
   add (3 , 10.0); //uses prototype (iii.)

3. **What are the applications of oops? (Nov/dec 2011)**
   - Real-time systems.
   - Simulation and modeling.
   - Object-oriented databases.
   - AI and expert systems.
4. **Define abstraction and encapsulation.(APR/MAY 2011)**
   Wrapping up of data and function within the structure is called as encapsulation.
   The insulation of data from direct access by the program is called as data hiding or information binding.
   The data is not accessible to the outside world and only those functions, which are wrapped in the class, can access it.
5. **Justify the need for static members. .(APR/MAY 2011)**
   Static variable are normally used to maintain values common to the entire class.
   Feature:
   • It is initialized to zero when the first object is created. No other initialization is permitted
   • only one copy of that member is created for the entire class and is shared by all the objects
   • It is only visible within the class, but its life time is the entire class type and scope of each static member variable must be defined outside the class
   • It is stored separately rather than objects
   Eg: static int count//count is initialized to zero when an object is created.
   int classname::count;//definition of static data member

6. **What is data hiding?(NOV/DEC 2010)**
   The insulation of data from direct access by the program is called as data hiding or information binding.

7. **What are the advantages of default arguments? (NOV/DEC 2010)**

Default arguments assign a default value to the parameter, which does not have matching argument in the function call. Default values are specified when the function is declared.
Eg : float amount(float principle,int period,float rate=0. 15)
Function call is
Value=amount(5000,7);
Here it takes principle=5000& period=7
And default value for rate=0.15
Value=amount(5000,7,0.34)
Passes an explicit value 0f 0.34 to rate
We must add default value from right to left

8. **Write a C++ program to check whether an integer is a prime or a composite number. (APR/MAY 2010)**

```
#include<iostream.h>
#include<conio.h>
void main()
{
clrscr();
int n,i=2,divisor=0;
cout<<"Enter a Number to Find Whether it is Prime or Composite ";
cin>>n;
while(i<n)
{
while(n%i==0)
{
i=i+1;
divisor=divisor+1;
}
i=i+1;
}
if(divisor==0)
{
cout<<"Number is Prime";
}
else
{
cout<<"Number is Composite"<<endl;
cout<<"Total Divisors except 1 and itself: "<<divisor;
}
getch();
}
```

9. **What is the difference between a class and a structure? (APR/MAY 2010)**

The only difference between a structure and a class is that structure members have public access by default and class members have private access by default, you can use the keywords `class` or `struct` to define equivalent classes.

For example, in the following code fragment, the class X is equivalent to the structure Y:

```
class X {
  // private by default
  int a;
public:
  // public member function
```

```
  int f() { return a = 5; };
};

struct Y {
  // public by default
  int f() { return a = 5; };
private:
  // private data member
  int a;
};
```

EXTRA QUESTIONS

1.Give the evolution diagram of OOPS concept.

Machine language
Procedure language
Assembly language
OOPS

2.What is Procedure oriented language?

Conventional programming, using high-level language such as COBOL, FORTRAN and C are commonly known as Procedure oriented language (POP). In POP number of functions are written to accomplish the tasks such as reading, calculating and printing.

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.

Function-1 Function-2 Function-3
Function-4 Function-5
Function-6 Function-7 Function-8
Main program

4) **Write any four features of OOPS.**

• Emphasis is on data rather than on procedure.
• Programs are divided into objects.
• Data is hidden and cannot be accessed by external functions.
• Follows bottom -up approach in program design.

5**) What are the basic concepts of OOPS?**

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

6) What are objects?

Objects are 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. Each object has the data and code to manipulate the data and theses objects interact with each other.

7)What is a class?

• The entire set of data and code of an object can be made a user-defined data type with the help of a class.
• Once a class has been defined, we can create any number of objects belonging to the classes.
• Classes are user-defined data types and behave like built-in types of the programming language.

8) what is encapsulation?
Wrapping up of data and function within the structure is called as encapsulation.
9)What is data abstraction?
The insulation of data from direct access by the program is called as data hiding or information binding.
The data is not accessible to the outside world and only those functions, which are wrapped in the class, can access it.
10)What are data members and member functions?
Classes use the concept of abstraction and are defined as a list of abstract attributes such as size, weight, and cost and uses functions to operate on these attributes.
The attributes are sometimes called as data members because they hold information. The functions that operate on these data are called as methods or member functions.
Eg: int a,b; // a,b are data members
Void getdata ( ) ; // member function
11)What is dynamic binding or late binding?
Binding refers to the linking of a procedure to the code to be executed in response to the call. Dynamic binding means that the code associated with a given procedure call is not known until the time of the call at the run-time.
12)Write the process of programming in an object-oriented language?
• Create classes that define objects and their behavior.
• Creating objects from class definition.
• Establishing communication among objects.
13)Give any four advantages of OOPS.
• The principle of data hiding helps the programmer to build secure programs that cannot be invaded by code in other parts of the program.
• It is possible to have multiple instances of an object to co-exist without any interference.
• Object oriented programming can be easily upgraded from small to large systems.
• Software complexity can be easily managed.
14)What are the features required for object-based programming Language?
• Data encapsulation.
• Data hiding and access mechanisms.
• Automatic initialization and clear up of objects.
• Operator overloading.
15)What are the features required for object oriented language?
• Data encapsulation.
• Data hiding and access mechanisms.
• Automatic initialization and clear up of objects.
• Operator overloading.
• Inheritance.
• Dynamic binding.
16)Give any four applications of OOPS
• Real-time systems.
• Simulation and modeling.
• Object-oriented databases.
• AI and expert systems.
17) Give any four applications of c++?
• Since c++ allows us to create hierarchy-related objects, we can build special object-oriented libraries, which can be used later by many programmers.
• C++ are easily maintainable and expandable.
• C part of C++ gives the language the ability to get close to the machine-level details.
• It is expected that C++ will replace C as a general-purpose language in the near future.
18) What are tokens?
The smallest individual units in a program are known as tokens. C++ has the following tokens,
• Keyword

• Identifiers
• Constants
• Strings
• Operator

19)What are keywords?

The keywords implement specific C++ language features. They are explicitly reserved identifiers and cannot be used as names fro the program variables or other user defined program elements.

Eg: go to, If, struct , else ,union etc.

20) Rules for naming the identifiers in C++.

• Only alphabetic characters, digits and underscore are permitted.
• The name cannot start with a digit.
• The upper case and lower case letters are distinct.
• A declared keyword cannot be used as a variable name.

21)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
delete Memory release operator
endl Line feed operator
new Memory allocation operator
setw Field width operator

22)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.

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()

23) What are free store operators (or) Memory management operators?

New and Delete operators are called as free store operators since they allocate the memory dynamically.

New operator can be used to create objects of any data type.

Pointer-variable = new data type;

Initialization of the memory using new operator can be done. This can be done as,

Pointer-variable = new data-type(value)

Delete operator is used to release the memory space for reuse. The general form of its use is

Delete pointer-variable;

24) What are manipulators?
Setw, endl are known as manipulators.
Manipulators are operators that are used to format the display. The endl manipulator when used in an output statement causes a linefeed to be inserted and its effect is similar to that of the newline character"\n".
Eg:Cout<<setw(5)<<sum<<endl;
25) What do you mean by enumerated datatype?
An enumerated datatype is another user-defined datatype, which provides a way for attaching names to numbers, thereby increasing comprehensibility of the code.The syntax of an enum statement is similar to that of the struct statesmen.
Eg:
enum shape{ circle, square, triangle}
enum color{ red, blue, green, yellow}
26) What are symbolic constants?
There are two ways for creating symbolic constants in C++:
• Using the qualifier constant.
• Defining a set of integer constants using enum keyword.
The program in any way cannot modify the value declared as constant in c++.
Eg:
Const int size =10;
Char name [size];
27)What do you mean by dynamic initialization of variables?
C++ permits initialization of the variables at run-time. This is referred to as dynamic initialization of variables.
In C++ ,a variable can be initialized at run-time using expressions at the place of declaration as,
……..
……
int n =strlen(string);
……..
float area=3.14*rad*rad;
Thus declaration and initialization is done simultaneously at the place where the variable is used for the first time.
28) What are reference variable?
A reference variable provides an alias(alternative name) for a previously defined variable.
sum total For example , if make the variable a reference to the variable , then sum and total can be used interchancheably to represent that variable.
Syntax :
Data-type &reference-name = variable-name
Eg:
float total = 100;
float sum = total;
29)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.
30)what is function prototype ?
The function prototype describes function interface to the compiler by giving details such as number ,type of arguments and type of return values
Function prototype is a declaration statement in the calling program and is of the
following
Type function_name(argument list); Eg float volume(int x,float y);
31)what is an inline function ?
An inline function is a function that is expanded in line when it is invoked. That is compiler replaces the function

call with the corresponding function code.
The inline functions are defined as Inline function-header
{
function body
}

32) Write some situations where inline expansion may not work

• for functions returning values, if loop, a switch, or a goto exists
• for functions not returning values ,if a return statement exists
• if function contain static variables
• if inline functions are recursive

33)what is a default argument ?

Default arguments assign a default value to the parameter, which does not have matching argument in the function call. Default values are specified when the f unction is declared.
Eg : float amount(float principle,int period,float rate=0. 15)
Function call is
Value=amount(5000,7);
Here it takes principle=5000& period=7
And default value for rate=0.15
Value=amount(5000,7,0.34)
Passes an explicit value 0f 0.34 to rate
We must add default value from right to left

34) What are constant arguments ?

keyword is const. The qualifier const tells the compiler that the function should not modify the argument. The compiler will generate an error when this condition is violated. This type of declaration is significant only when we pass arguments by reference or pointers
eg: int strlen(const char *p);

35) How the class is specified ?

Generally class specification has two parts
• class declaration
It describes the type and scope of its member
• class function definition
It describes how the class functions are implemented
The general form is
Class class_name
{
private:
variable declarations;
function declaration;
public:
variable declaration;
function declaration;
};

36) How to create an object ?

Once the class has been declared, we can create variables of that type by using the classname
Eg:classname x; //memory for x is created

37) How to access a class member ?

object-name. function-name(actual arguments)
eg:x.getdata(100,75.5);

38) How the member functions are defined ?

Member functions can be defined in two ways
• 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)
{
}
• 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;
};
39) What is static data member?
Static variable are normally used to maintain values common to the entire class.
Feature:
• It is initialized to zero when the first object is created. No other initialization is permitted
• only one copy of that member is created for the entire class and is shared by all the objects
• It is only visible within the class, but its life time is the entire class type and scope of each static member variable must be defined outside the class
• It is stored separately rather than objects
Eg: static int count//count is initialized to zero when an object is created.
int classname::count;//definition of static data member
40) What is static member function?
A member function that is declared as static has the following properties
• A static function can have access to only other static member declared in the same class
• A static member function can be called using the classname as follows
classname ::function_name;
41) How the objects are used as function argument?
This can be done in two ways
• A copy of the entire object is passed to the argument
• Only address of the objects is transferred to the f unction
42) What is called pass by reference?
In this method address of an object is passed, the called function works directly on the actual arguments.
43) Define const member
If a member function does not alter any data in the class, then we may declare it as const member function as
Void mul(int ,int)const;
44) Define pointers to member
It is possible to take the address of a member of a class and assign it to a pointer. The address of a member can be obtained by applying the operator &to a "fully qualified" class member name. A class member pointer can be declared using the operator::*with the class name.
Eg: class A
{
int m;
public:
void show( );
};
pointer to member m is defined as
int A::*ip=&A::m;

A::*->pointer to member of A class
&A::m->address of the m member of A class

45) When the deferencing operator ->* is used?

It is used to access a member when we use pointer to both the object and the member.

46) When the deferencing operator .* is used?

It is used to access a member when the object itself is used as pointers.

47) Define local classes.

Classes can be defined and used inside a function or a block. such classes are called local classes. It can use global variables and static variables declared inside the function but cannot use automatic local variables.

Eg;

```
void test(int a)
{
…….
}
class student
{
……
};
student s1(a);
}
```

48) What are Friend functions? Write the syntax

A function that has access to the private member of the class but is not itself a member of the class is called friend functions.

The general form is

friend data_type function_name( );

Friend function is preceded by the keyword 'friend'.

49)Write some properties of friend functions.

• Friend function is not in the scope of the class to which it has been declared as friend. Hence it cannot be called using the object of that class.
• Usually it has object as arguments.
• It can be declared either in the public or private part of a class.
• It cannot access member names directly. It has to use an object name and dot membership operator with each member name. eg: (A . x)

50) What is function overloading? Give an example.

Function overloading means we can use the same function name to create functions that perform a variety of different tasks.

Eg: An overloaded add ( ) function handles different data types as shown below.

```
// Declarations
i. int add( int a, int b); //add function with 2 arguments of same type
ii. int add( int a, int b, int c); //add function with 3 arguments of same type
iii. double add( int p, double q); //add function with 2 arguments of
different type
//Function calls
add (3 , 4); //uses prototype ( i. )
add (3, 4, 5); //uses prototype ( ii. )
add (3 , 10.0); //uses prototype ( iii. )
```

51) Define local classes.

Classes can be defined and used inside a function or a block. such classes are called local classes. It can use global variables and static variables declared inside the function but cannot use automatic local variables.

Eg;

```
void test(int a)
```

```
{
.......
}
class student
{
.........
};
student s1(a);}
```

PART B (5x 16=80 marks)

1. **Write brief notes on Friend function and show how Modifying a Class's private Data With a Friend Function. (Nov/dec 2011)**

A **friend function** for a class is used in object-oriented programming to allow access to public, private, or protected data in the class from the outside. Normally, a function that is not a member of a class cannot access such information; neither can an external class. Occasionally, such access will be advantageous for the programmer. Under these circumstances, the function or external class can be declared as a friend of the class using the `friend` keyword.

A friend function is declared by the class that is granting access. Friend declaration can be placed anywhere in the class declaration. It is not affected by the access control keywords.

A similar concept is that of friend class.

Friends should be used with caution. Too many functions or external classes declared as friends of a class with protected or private data may lessen the value of encapsulation of separate classes in object-oriented programming and may indicate a problem in the overall architecture design.

This approach may be used when a function needs to access private data in objects from two different classes. This may be accomplished in two similar ways

- a function of global or namespace scope may be declared as friend of both classes
- a member function of one class may be declared as friend of another one.

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

class B; // Forward declaration of class B in order for example to compile
class A
{
private:
   int a;
public:
   A() { a = 0; }
   void show(A& x, B& y);
   friend void ::show(A& x, B& y); // declaration of global friend
};

class B
{
private:
   int b;
public:
   B() { b = 6; }
   friend void  ::show(A& x, B& y); // declaration of global friend
   friend void A::show(A& x, B& y); // declaration of friend from other class
};

// Definition of a member function of A; this member is a friend of B
void A::show(A& x, B& y)
```

```
{
  cout << "Show via function member of A" << endl;
  cout << "A::a = " << x.a << endl;
  cout << "B::b = " << y.b << endl;
}

// Friend for A and B, definition of global function
void show(A& x, B& y)
{
  cout << "Show via global function" << endl;
  cout << "A::a = " << x.a << endl;
  cout << "B::b = " << y.b << endl;
}
int main()
{
   A a;
   B b;

   show(a,b);
   a.show(a,b);
}
```

2. Write about polymorphism with example? (Nov/dec 2011)

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 << "Rectangle 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;
}
```

3. Explain pointers to objects with example program? (Nov/dec 2011)

```
class M
{
        int x;
        int y;
```

```
public:
        void set_xy(int a, int b)
        {
                x=a;
                y=b;
        }
        friend int sum(M,m);
};
int sum(M,m)
{
        int M :: * px = &M :: x;
        int M :: * py = &M :: y;
        M *pm = &m;
        int S = m. *px + pm-> *py;
        return S;
}
int main()
{
        M n;
        void (M :: *pf) (int,int) = &M :: set_xy;
        (n. *pf) (10,20);
        cout<< "SUM= " << sum(n) << "\n";
        M *op = &n;
        (op-> *pf) (30,40);
        cout<< "SUM= " << sum(n) << "\n";
        return 0;
}
```

4.Explain function overloading with an example .(APR/MAY 2011)

A single function name can be used to perform different types of tasks. The same function name can be used to handle different number and different types of arguments. This is known as function overloading or function polymorphism.

```
#include<iostream.h>
#include<conio.h>
void swap(int &x,int &y)
{
int t;
t=x;
x=y;
y=t;
}
void swap(float &p,float &q)
{
float t;
t=p;
p=q;
q=t;
}
void swap(char &c1,char &c2)
{
char t;
```

```
t=c1;
c1=c2;
c2=t;
}
void main()
{
int i,j;
float a,b;
char s1,s2;
clrscr();
cout<<"\n Enter two integers : \n";
cout<<" i = ";
cin>>i;
cout<<"\n j = ";
cin>>j;
swap(i,j);
cout<<"\n Enter two float numbers : \n";
cout<<" a = ";
cin>>a;
cout<<"\n b = ";
cin>>b;
swap(a,b);
cout<<"\n Enter two Characters : \n";
cout<<" s1 = ";
cin>>s1;
cout<<"\n s2 = ";
cin>>s2;
swap(s1,s2);
cout<<"\n After Swapping \n";
cout<<" \n Integers i = "<<i<<"\t j =  "<<j;
cout<<" \n Float Numbers a= "<<a<<"\t b =  "<<b;
cout<<" \n Characters s1 = "<<s1<<"\t s2 =  "<<s2;
getch();
}
```

5. **Explain the special features of object oriented programming. (16) (NOV/DEC 2010)**

- Objects : Objects are the basic run time entities in an object oriented system. They are instance of a class. They may represent a person, a place etc that a program has to handle. They may also represent user-defined data. They contain both data and code.
- Classes : Class is a collection of objects of similar data types. Class is a user-defined data type. The entire set of data and code of an object can be made a user defined type through a class.
- Data Abstraction : Abstraction refers to the act of representing the essential features without including the background details or explanations.
- Encapsulation The wrapping up of data and functions into a single unit is known as data encapsulation. Here the data is not accessible to the outside world. The insulation of data from direct access by the program is called data hiding or information hiding.
- Inheritance : Inheritance is the process by which objects of one class acquire the properties of objects of another class. It supports the concept of hierarchical classification and provides the idea of reusability. The class which is inherited is known as the base or super class and class which is newly derived is known as the derived or sub class.

- Polymorphism : Polymorphism is an important concept of OOPs. Polymorphism means one name, multiple forms. It is the ability of a function or operator to take more than one form at different instances.
- Dynamic Binding : Binding refers to the linking of procedure call to the code to be executed in response to the call. Dynamic Binding or Late Binding means that the code associated with a given procedure call is known only at the run-time.

- Message Passing : Objects communicate between each other by sending and receiving information known as messages. A message to an object is a request for execution of a procedure. Message passing involves specifying the name of the object, the name of the function and the information to be sent.

6. List the special characteristics of friend function.

- A friend function is used for accessing the non-public members of a class. A class can allow non-member functions and other classes to access its own private data, by making them friends. Thus, a friend function is an ordinary function or a member of another class.

The friend function is written as any other normal function, except the function declaration of these functions is preceded with the keyword friend. The friend function must have the class to which it is declared as friend passed to it in argument.

Some important points

- The keyword friend is placed only in the function declaration of the friend function and not in the function definition.

  It is possible to declare a function as friend in any number of classes.
- When a class is declared as a friend, the friend class has access to the private data of the class that made this a friend.
- It is possible to declare the friend function as either private or public.
- The function can be invoked without the use of an object.

7. (i) Write a C++ program to multiply two matrices and print the result.

```
#include<iostream.h>
Class matrix
{
Private: int a[3][3];
Public: void getmatrix(int r, int c);
void printmatrix(int r, int c);
void mul(matrix,matrix);
};
Void matrix::getmatrix(int r, int c)
{
int I,j;
cout<<"Enter matrix elements";
For( i=0;i<r;i++)
For(j=0;j<c;j++)
Cin>>a[i][j];
}
Void matrix::printmatrix(int r, int c)
{
Int I,j;
For(i=0;i<r;i++)
```

```
{
For(j=0;j<c;j++)
{
Cout<<”  “<<a[i][j];
}
Cout<<”\n”;
}
}
Void matrix::mul(matrix A, matrix B)
{
Matrix C;
Int I,j,k;
r1=A.r; c1=A.c;
r2=B.r; c2=B.c;
For(i=0;i<r1;i++)
{
For(j=0;j<c1;j++)
{
C.a[i][j]=0;
For(k=0;k<c2;k++)
{
c.a[i][j]=C.a[i][j]+A.a[i][k]*B.a[k][j];
}
}
}

Void main()
{
Matrix A[3][3],B[3][3],C[3][3];
Int r1,r2,c1,c2;
Cout<<”enter order of 2 matrices”;
Cin>>r1>>c1>>r2>>c2;
Try
{
If(r2==c1)
{
A.readmatrix(r1,c1);
B.readmatrix(r2,c2);
C=mul(A,B);
A.printmatrix(r1,c1);
B.printmatrix(r2,c2);
C.printmatrix(r1,c2);
Exit(0);
}
else
 { throw (c2);
}
}
Catch(int x)
{
Cout<<”Invalid matrix order”;
}
```

```
    }
     }
```

(ii) **Explain inline functions with an Example program(APR/MAY 2010)** An inline function is a function that is expanded in line when it is invoked. The compiler Replaces the function call with corresponding function code. The inline funcitions are defined As follows:

```
inline function-header
{
  Function body;
 }
```

Example:

```
inline double cube(double a)
    {
      Return(a*a*a);
     }
```

Some situations where inline expansion may not work are:

- For functions returning values, if a loop , a switch, or a goto exists.
- For functions not returning values, if a return statement exists.
- If functions contain static variables.
- If inline functions are recursive.

-Example program to illustrate inline functions :

```
#include <iostream.h>
inline float mul(float x, float y)
   {     return(x*y);          }
inline double div(double p, double q)
    {   return(p/q) ;       }
int main( )
{
        float a=1.2 ;
        float b=2.3;
        cout<< mul(a,b)<<"\n";
        cout<< div(a,b)<<"\n";
         return 0;
}
```

8.(i) Explain the concept of pointers with an example program written inC++.

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

#define maxitems 1000
typedef int vector[maxitems];

void addvectors(vector & temp,vector & v1,vector & v2) {
  for (int index =0;index < maxitems;index++)
    temp[index]= v1[index] + v2[index];
  }

int main(int argc, char* argv[])
{
  vector a,b,c;
  int index;
```

```
  for( index=0; index < maxitems; index++){
   a[index]=10+index;
   b[index]= index*1000;
  }
 addvectors(c,a,b) ;
 cout << "c[0]= " << c[0] << "\nc[999]= " << c[999] << endl;
 return 0;
}
```

The output from this is

```
c[0]= 10
c[999]= 1000009
```

This passes references to three arrays, each holding 1,000 ints and defined by the typedef **vector**. It loops through all elements, adding each of the elements and storing the result in the third.

(ii) How is function overloading different from operator overloading. (APR/MAY 2010)

Function Overloading:

A single function name can be used to perform different types of tasks. The same function name can be used to handle different number and different types of arguments. This is known as function overloading or function polymorphism.

```
#include<iostream.h>
#include<conio.h>
void swap(int &x,int &y)
{
int t;
t=x;
x=y;
y=t;
}
void swap(float &p,float &q)
{
float t;
t=p;
p=q;
q=t;
}
void swap(char &c1,char &c2)
{
char t;
t=c1;
c1=c2;
c2=t;
}
void main()
{
int i,j;
float a,b;
char s1,s2;
clrscr();
cout<<"\n Enter two integers : \n";
cout<<" i = ";
cin>>i;
```

```
cout<<"\n j = ";
cin>>j;
swap(i,j);
cout<<"\n Enter two float numbers : \n";
cout<<" a = ";
cin>>a;
cout<<"\n b = ";
cin>>b;
swap(a,b);
cout<<"\n Enter two Characters : \n";
cout<<" s1 = ";
cin>>s1;
cout<<"\n s2 = ";
cin>>s2;
swap(s1,s2);
cout<<"\n After Swapping \n";
cout<<" \n Integers i = "<<i<<"\t j =  "<<j;
cout<<" \n Float Numbers a= "<<a<<"\t b =  "<<b;
cout<<" \n Characters s1 = "<<s1<<"\t s2 =  "<<s2;
getch();
}
```

operator overloading:

The process of making an operator to exhibit different behaviors at different instances.

- Only existing operators can be overloaded.
- We cannot change the basic meaning of an operator.
- The overloaded operator must have at least one operand.
- Overloaded operators follow the syntax rules of the original operators.

-General form of operator function is:

Return type classname :: operator (op-arglist)
{
Function body
}

Overloaded operator functions can be invoked by expression
x op y for binary operators

In the following program overloaded function is invoked when the expression c1+c2 is encountered. This expression is the same as operator **op(x,y) (ie) operator +(c1,c2)**

using friend function

```
#include<iostream.h>
#include<conio.h>
class complex
{
private:
float real;
float img;
public:
complex()
{
real=0.0;
img=0.0;
}
```

```
complex(float a,float b)
{
real=a;
img=b;
}
friend complex operator +(complex,complex);
void display()
{
cout<<"\n"<<real<<"+-i"<<img<<"\n";
}
};
complex operator +(complex c1,complex c2)
{
complex t;
t.real=c1.real+c2.real;
t.img=c1.img+c2.img;
return(t);
}
void main()
{
clrscr();
complex c1(5.5,2.5);
complex c2(1.5,5.5);
complex c3;
c3=c1+c2;
c1.display();
c2.display();
c3.display();
getch();
}
```

UNIT II

9

Constructors – default constructor – Parameterized constructors – Constructor with dynamic allocation – copy constructor – destructors – operator overloading – overloading through friend functions – overloading the assignment operator – type conversion – explicit constructor

PART A — (10 x 2 20 marks)

1. **What is the difference between a parameter and an argument? (Nov/dec 2011)**

Parameters

A *parameter* represents a value that the procedure expects you to pass when you call it. The procedure's declaration defines its parameters.

When you define a **Function** or **Sub** procedure, you specify a *parameter list* in parentheses immediately following the procedure name. For each parameter, you specify a name, a data type, and a passing mechanism (ByVal or ByRef). You can also indicate that a parameter is optional, meaning the calling code does not have to pass a value for it.

The name of each parameter serves as a *local variable* within the procedure. You use the parameter name the same way you use any other variable.

Arguments

An *argument* represents the value you pass to a procedure parameter when you call the procedure. The calling code supplies the arguments when it calls the procedure.

When you call a **Function** or **Sub** procedure, you include an *argument list* in parentheses immediately following the procedure name. Each argument corresponds to the parameter in the same position in the list.
In contrast to parameter definition, arguments do not have names. Each argument is an expression, which can contain zero or more variables, constants, and literals. The data type of the evaluated expression should normally match the data type defined for the corresponding parameter, and in any case it must be convertible to the parameter type

2. **Explain the purpose of a function parameter. (Nov/dec 2011)**

The Function prototype serves the following purposes -

1) It tells the return type of the data that the function will return.
2) It tells the number of arguments passed to the function.
3) It tells the data types of the each of the passed arguments.
4) Also it tells the order in which the arguments are passed to the function.
Therefore essentially, function prototype specifies the input/output interlace to the function i.e. what to give to the function and what to expect from the function.

3. Explain the multiple meanings of the operators « and » in C++ and their precedence. (Nov/dec 2011)

Precedence	Operator	Description	Associativity
7	<<	Bitwise left shift	Left-to-right
	>>	Bitwise right shift	

4. **What is a copy constructor? (Nov/dec 2011)**
A copy constructor is used to declare and initialize an object from another object. It takes a reference to an object of the same class as an argument
Eg: integer i2(i1);
would define the object i2 at the same time initialize it to the values of i1.
Another form of this statement is
Eg: integer i2=i1;
The process of initializing through a copy constructor is known as copy initialization .

5. **Define copy constructor? (Nov/dec 2011)**
A copy constructor is used to declare and initialize an object from another object. It takes a reference to an object of the same class as an argument
Eg: integer i2(i1);
would define the object i2 at the same time initialize it to the values of i1.
Another form of this statement is
Eg: integer i2=i1;
The process of initializing through a copy constructor is known as copy initialization .

6. **What is type conversion? (Nov/dec 2011)**
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;
}
};
```

Constructor will be called automatically while creating objects so that this conversion is done automatically.

7. Explain the functions of default constructor. .(APR/MAY 2011)

The constructor with no arguments is called default constructor

Eg:

```
Class integer
{
int m,n;
Public:
Integer( );
…….
};
integer::integer( )//default constructor
{
m=0;n=0;
}
```

the statement

integer a;

invokes the default constructor

8. What is the need for overloading the assignment operator? .(APR/MAY 2011)

The **assignment operator** is used to copy the values from one object to another *already existing object*. The key words here are "already existing". Consider the following example:

```
1   Cents cMark(5); // calls Cents constructor
2   Cents cNancy; // calls Cents default constructor
3   cNancy = cMark; // calls Cents assignment
    operator
```

In this case, cNancy has already been created by the time the assignment is executed. Consequently, the Cents assignment operator is called. The assignment operator must be overloaded as a member function.

9. Write any four special properties of constructor. (NOV/DEC 2010)

T hey 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 therefore, and they cannot return values
- They cannot be inherited, though a derived class can call the base class
- They can have default arguments
- Constructors cannot be virtual f unction

10. **List any four operators that cannot be overloaded**. **(NOV/DEC 2010)**

- Class member access operator (. , .*)
- Scope resolution operator (::)
- Size operator (sizeof)
- Conditional operator (?:)

11. Write the difference between realloc() and free().(APR/MAY 2010)

An existing block of memory which was allocated by malloc() subroutine, will be freed by **free()** subroutine. In case , an invalid pointer parameter is passed, unexpected results will occur. If the parameter is a null pointer, then no action will occur.

Where as the **realloc()** subroutine allows the developer to change the block size of the memory which was pointed to by the pointer parameter, to a specified bytes size through size parameter and a new pointer to the block is returned.

The pointer parameter specified must have been created by using malloc(),calloc() or realloc() sub routines and should not deallocated with realloc() or free() subroutines. If the pointer parameter is a null pointer, then no action will occur.

12. **Give the purpose of gets and puts function**. **(APR/MAY 2010)**

gets()

char * gets (char * str);

Get string from stdin

Reads characters from stdin and stores them as a string into *str* until a newline character ('\n') or the End-of-File is reached.

puts()

int puts (const char * str);

Write string to stdout

Writes the C string pointed by *str* to stdout and appends a newline character ('\n').

EXTRA QUESTIONS:

1. **Define constructor?**
   A function with the same name as the class itself responsible for construction and returning objects of the class is called constructor.
2. **What do you mean by copy constructor?**
   Copy constructor is used to copy the value of one object into another. When one object is copied to another using initialization , they are copied by executing the copy constructor.
   The copy constructor contains the object as one of the passing argument.
3. **Define default constructor and give example?**
   A constructor without any argument is called default constructor.
   Ex Class complex
   { Public: void complex(); };
4. **What is parameterized constructor?**
   A constructor with one or more than one argument is called parameterized constructor.
   Ex
   Class complex
   { Public: Complex complex(int real, int imag); };
5. **Define destructor?**
   The function which bears the same name as class itself preceded by ~ is destructor. The destructors automatically called when the object goes out of scope.
6. **What do you mean by explicit constructor?**
   The one argument constructor which is defined with keyword explicit before the definition. Explicit word prohibits the generation of conversion operator for a single argument constructor.
7. **Explain multiple constructor?**
   A class with more than one constructor comes under multiple constructor. Multiple constructor is constructed with different argument list. That is either with empty default constructor or with parameterized constructor.
8. **What is operator overloading?**
   The process of giving an existing operator a new , additional meaning is called operator overloading.
9. **What is the lifetime of an object?**
   The life time of a global object is throughout the execution of the program. Otherwise , the object comes into existence when constructor is over and is alive till it gives out of scope, i.e just before the destructor is applied.
10. **List out the operator which can not be overloaded ?**
    1. The dot operator for member access.(.)
    2. The dereference member to class operator .(*)
    3. Scope resolution operator.
    4. Size of operator (sizeof).

5. Conditional ternary operator(:?)
6. Casting operators static_cast<>, dynamic_cast<>, reinterpret_cast<>, const_cast<>
7. # and ## tokens for macro preprocessor.

11. Define new operator?

The operator new is used for allocation of memory, it gets memory from heap. It is similar to malloc() in C. Ex to get memory for an integer and assign the address of allocated memory to pointer p,

Int* p = new int.

12. Define delete operator?

Delete in C++ does a similar job as free() function in C, i.e it releases the memory occupied by the new operator.

13. Define wrapper classes?

A class which makes a C like struct or built in type data represented as a class. For example, an Integer wrapper class represents a data type int as a class.

14. List out the operators that can not be overloaded as a friend?

1. Assignment operator =. 2.Function call operator () 3.Array subscript operator []
4. Access to class member using pointer to object operator ->.

15. Define type conversion?

When we need to convert between different types , we guide the compiler how to convert from one type to another by writing operator functions or constructors.

16. Difference between unary and binary operator?

All operators having a single argument are unary operators. When we overload these operators as member function, we do not need to pass any argument explicitly. Operators with two argument are known as binary operators. They will have a single argument when defined as member function. The first argument to that operator is always the invoking object.

17. Explain insertion and extraction operator?

Insertion operator : The << operator which is used to write to the console is known as insertion operator. It is also known as the output operator.

Extraction operator : The >> operator which is used to read from the keyboard is known as extraction operator. It is also known as the input operator.

18. Explain function overloading?

Function overloading in C++ allows more than one function with same name but with different set of arguments. This process is known as function overloading and each participating function is known as overloaded function.

19. List out the user defined conversion?

1.conversion from built in data type to an object. 2. conversion from object to a built in data type.

20. What is function objects?

Objects of the classes where () operator is overloaded. In this case objects can be written with a () and can be treated like functions. Such objects which can be called like a function are known as function objects.

PARTB (5x 16=80 marks)

1. Write a program to overload the stream insertion and stream extraction operators to handle data of a user-defined telephone number class called Phone Number. (Nov/dec 2011)

Overloaded stream insertion and stream extraction operators for class PhoneNumber

```
//  PhoneNumber.cpp
2  // Overloaded stream insertion and stream extraction operators
3  // for class PhoneNumber.
4  #include <iomanip>
5  using std::setw;
6
7  #include "PhoneNumber.h"
8
```

```
// overloaded stream insertion operator; cannot be
// a member function if we would like to invoke it with
// cout << somePhoneNumber;
ostream &operator<<( ostream &output, const PhoneNumber &number )
{
   output << "(" << number.areaCode << ") "
      << number.exchange << "-" << number.line;
   return output; // enables cout << a << b << c;
} // end function operator<<

// overloaded stream extraction operator; cannot be
// a member function if we would like to invoke it with
// cin >> somePhoneNumber;
istream &operator>>( istream &input, PhoneNumber &number )
{
   input.ignore(); // skip (
   input >> setw( 3 ) >> number.areaCode; // input area code
   input.ignore( 2 ); // skip ) and space
   input >> setw( 3 ) >> number.exchange; // input exchange
   input.ignore(); // skip dash (-)
   input >> setw( 4 ) >> number.line; // input line
   return input; // enables cin >> a >> b >> c;
} // end function operator>>
```

Overloaded stream insertion and stream extraction operators.

```
// Fig. 11.5: fig11_05.cpp
// Demonstrating class PhoneNumber's overloaded stream insertion
// and stream extraction operators.
#include <iostream>
using std::cout;
using std::cin;
using std::endl;

#include "PhoneNumber.h"

int main()
{
   PhoneNumber phone; // create object phone

   cout << "Enter phone number in the form (123) 456-7890:" << endl;

   // cin >> phone invokes operator>> by implicitly issuing
   // the global function call operator>>( cin, phone )
   cin >> phone;

   cout << "The phone number entered was: ";

   // cout << phone invokes operator<< by implicitly issuing
   // the global function call operator<<( cout, phone )
   cout << phone << endl;
```

```
26     return 0;
27  } // end main
```

```
Enter phone number in the form (123) 456-7890:
(800) 555-1212
The phone number entered was: (800) 555-1212
```

2. (i) Explain '+' operator overloading with an example. (8)

The process of making an operator to exhibit different behaviors at different instances.

• Only existing operators can be overloaded.
• We cannot change the basic meaning of an operator.
• The overloaded operator must have at least one operand.
• Overloaded operators follow the syntax rules of the original operators.

-General form of operator function is:

Return type classname :: operator (op-arglist)
{
Function body
}

Overloaded operator functions can be invoked by expression
x op y for binary operators

In the following program overloaded function is invoked when the expression c1+c2 is encountered. This expression is the same as operator **op(x,y) (ie) operator +(c1,c2)**
using friend function

```
#include<iostream.h>
#include<conio.h>
class complex
{
private:
float real;
float img;
public:
complex()
{
real=0.0;
img=0.0;
}
complex(float a,float b)
{
real=a;
img=b;
}
friend complex operator +(complex,complex);
void display()
{
cout<<"\n"<<real<<"+-i"<<img<<"\n";
}
};
complex operator +(complex c1,complex c2)
```

```
{
complex t;
t.real=c1.real+c2.real;
t.img=c1.img+c2.img;
return(t);
}
void main()
{
clrscr();
complex c1(5.5,2.5);
complex c2(1.5,5.5);
complex c3;
c3=c1+c2;
c1.display();
c2.display();
c3.display();
getch();
}
```

(ii) Explain type conversion with suitable example (8) (Nov/dec 2011)

There are three types of conversions. They are

- Conversion from basic type to class type – done using constructor
- Conversion from class type to basic type – done using a casting operator
- Conversion from one class type to another – done using constructor or casting operator

```
//TYPE CONVERSION FROM ONE OBJECT TO ANOTHER OBJECT
#include<iostream.h>
#include<conio.h>
class sample
{
private:
int val;
public:
sample(int a,int b,int c)
{
if((a>b)&&(a>c))
val=a;
else if(b>c)
val=b;
else
val=c;
}
int send()
{
return val;
}
};
class sample1
{
private:
int y;
public:
```

```
sample1()
{
}
sample1(sample s1)
{
y=s1.send();
}
void print()
{
cout<<"\n Greatest number is : "<<y;
}
};
void main()
{
int a,b,c;
clrscr();
cout<<"\n Enter three numbers \n";
cin>>a>>b>>c;
sample s1(a,b,c);
sample1 s2;
s2=s1;
s2.print();
getch();
}
```

3. Explain Parameterized constructor with example? (Nov/dec 2011)

Constructor with arguments is called parameterized constructor

Eg;

```
Class integer
{ int m,n;
public:
integer(int x,int y)
{ m=x;n=y;
}
```

To invoke parameterized constructor we must pass the initial values as arguments to the constructor function when an object is declared. This is done in two ways

1.By calling the constructor explicitly

eg: integer int1=integer(10,10);

2.By calling the constructor implicitly

eg: Integer int1(10,10);4) Define default argument constructor

The constructor with default arguments are called default argument constructor

Eg:

Complex(float real,float imag=0);

The default value of the argument imag is 0

The statement complex a(6.0)

assign real=6.0 and imag=0

the statement

complex a(2.3,9.0)

assign real=2.3 and imag=9.0

4. Write

1. Explain copy constructor with an example.(8)

The *copy constructor* lets you create a new object from an existing one by initialization. A copy constructor of a class A is a non-template constructor in which the first parameter is of type A&, const A&, volatile A&, or const volatile A&, and the rest of its parameters (if there are any) have default values.
If you do not declare a copy constructor for a class A, the compiler will implicitly declare one for you, which will be an inline public member.
The following example demonstrates implicitly defined and user-defined copy constructors:

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

struct A {
  int i;
  A() : i(10) { }
};

struct B {
  int j;
  B() : j(20) {
    cout << "Constructor B(), j = " << j << endl;
  }

  B(B& arg) : j(arg.j) {
    cout << "Copy constructor B(B&), j = " << j << endl;
  }

  B(const B&, int val = 30) : j(val) {
    cout << "Copy constructor B(const B&, int), j = " << j << endl;
  }
};

struct C {
  C() { }
  C(C&) { }
};

int main() {
  A a;
  A a1(a);
  B b;
  const B b_const;
  B b1(b);
  B b2(b_const);
  const C c_const;
//  C c1(c_const);
}
```

The following is the output of the above example:

```
Constructor B(), j = 20
Constructor B(), j = 20
Copy constructor B(B&), j = 20
Copy constructor B(const B&, int), j = 30
```

5.Explain the use of destructor with an example.(APR/MAY 2011)

Destructors are usually used to deallocate memory and do other cleanup for a class object and its class members when the object is destroyed. A destructor is called for a class object when that object passes out of scope or is explicitly deleted.
A destructor is a member function with the same name as its class prefixed by a ~ (tilde). For example:

```
class X {
public:
  // Constructor for class X
  X();
  // Destructor for class X
  ~X();
};
```

A destructor takes no arguments and has no return type. Its address cannot be taken. Destructors cannot be declared const, volatile, const volatile or static. A destructor can be declared virtual or pure virtual.
If no user-defined destructor exists for a class and one is needed, the compiler implicitly declares a destructor. This implicitly declared destructor is an inline public member of its class.
The compiler will implicitly define an implicitly declared destructor when the compiler uses the destructor to destroy an object of the destructor's class type. Suppose a class A has an implicitly declared destructor. The following is equivalent to the function the compiler would implicitly define for A:

```
A::~A() { }
```

The compiler first implicitly defines the implicitly declared destructors of the base classes and nonstatic data members of a class A before defining the implicitly declared destructor of A
A destructor of a class A is *trivial* if all the following are true:

- It is implicitly defined
- All the direct base classes of A have trivial destructors
- The classes of all the nonstatic data members of A have trivial destructors

If any of the above are false, then the destructor is *nontrivial*.
A union member cannot be of a class type that has a nontrivial destructor.
Class members that are class types can have their own destructors. Both base and derived classes can have destructors, although destructors are not inherited. If a base class A or a member of A has a destructor, and a class derived from A does not declare a destructor, a default destructor is generated.
The default destructor calls the destructors of the base class and members of the derived class.
The destructors of base classes and members are called in the reverse order of the completion of their constructor:

1. The destructor for a class object is called before destructors for members and bases are called.
2. Destructors for nonstatic members are called before destructors for base classes are called.
3. Destructors for nonvirtual base classes are called before destructors for virtual base classes are called.

When an exception is thrown for a class object with a destructor, the destructor for the temporary object thrown is not called until control passes out of the catch block.
Destructors are implicitly called when an automatic object (a local object that has been declared auto or register, or not declared as static or extern) or temporary object passes out of scope. They are implicitly called at program termination for constructed external and static objects. Destructors are invoked when you use the delete operator for objects created with the new operator.
For example:

```
#include <string>

class Y {
private:
  char * string;
  int number;
public:
  // Constructor
  Y(const char*, int);
  // Destructor
```

```
  ~Y() { delete[] string; }
};

// Define class Y constructor
Y::Y(const char* n, int a) {
  string = strcpy(new char[strlen(n) + 1 ], n);
  number = a;
}

int main () {
  // Create and initialize
  // object of class Y
  Y yobj = Y("somestring", 10);

  // ...

  // Destructor ~Y is called before
  // control returns from main()
}
```

6. Explain the different types of constructors with suitable examples. (16) (NOV/DEC 2010)

```
class code
{
        int id;
public:
        code(){}                        // default constructor
        code(int a) { id = a;}                  // parameterized constructor
        code(code & x)                          // copy constructor
        {
                id = x.id;
        }
        void display(void)
        {
                cout << id;
        }
};
int main()
{
        code A(100);
        code B(A);
        code C =A;
        code D;
        D = A;
        cout << "\n id of A: " ; A.display();
        cout << "\n id of B: " ; B.display();
        cout << "\n id of C: " ; C.display();
        cout << "\n id of D: " ; D.display();
        return 0;
}
```

7. **(i)Write the rules for overloading the operators. (6)**
   - Only existing operators can be overloaded. New operators cannot be created.
   - The overloaded operator must have atleast one operand that is of user defined type.
   - The basic meaning of an operator cannot be changed. That is the plus operator cannot be used to subtract one value from the other.
   - Overloaded operator follow the syntax rules of the original operators. They cannot be overridden.
   - There are some operators that cannot be overloaded. They are Size of, . ,: :,?:.
   - Friend function cannot be used to overload certain operators (= ,( ) ,[ ] ,->).However member functions can be used to overload them.
   - Unary operators, overload by means of a member function, take no explicit arguments and return no explicit values,but,those overloaded by means of a friend function, take one reference argument.
   - Binary operators overloaded through a member function take one explicit argument and those which are overloaded through a friend function take two explicit arguments.
   - When using binary operator overloaded through a member function, the left hand operand must be an object of the relevant class.
   - Binary arithmetic operators such as +,-,*,and / must explicitly return a value. They must not attempt to change their own arguments.

(ii) **Write a C++ program to add two complex numbers using operator overloading. (10) (NOV/DEC 2010)**

```
#include<iostream.h>
#include<conio.h>
class complex
{
private:
float real;
float img;
public:
complex()
{
real=0.0;
img=0.0;
}
complex(float a,float b)
{
real=a;
img=b;
}
friend complex operator +(complex,complex);
void display()
{
cout<<"\n"<<real<<"+-i"<<img<<"\n";
}
};
complex operator +(complex c1,complex c2)
{
complex t;
t.real=c1.real+c2.real;
t.img=c1.img+c2.img;
```

```
return(t);
}
void main()
{
clrscr();
complex c1(5.5,2.5);
complex c2(1.5,5.5);
complex c3;
c3=c1+c2;
c1.display();
c2.display();
c3.display();
getch();
}
```

8 .Explain friend functions with an example. (APR/MAY 2010)

A **friend** function is a function that is not a member of a class but has access to the class's private and protected members. Friend functions are not considered class members; they are normal external functions that are given special access privileges. Friends are not in the class's scope, and they are not called using the member-selection operators (**.** and –>) unless they are members of another class. A **friend** function is declared by the class that is granting access. The **friend** declaration can be placed anywhere in the class declaration. It is not affected by the access control keywords.

The following example shows a `Point` class and a friend function, `ChangePrivate`. The **friend** function has access to the private data member of the `Point` object it receives as a parameter.

```
// friend_functions.cpp
#include <iostream>
using namespace std;
class Point
{
   friend void ChangePrivate( Point & );
public:
   Point( void ) : m_i(0) {}
   void PrintPrivate( void ){cout << m_i << endl; }

private:
   int m_i;
};

void ChangePrivate ( Point &i ) { i.m_i++; }

int main()
{
  Point sPoint;
  sPoint.PrintPrivate();
  ChangePrivate(sPoint);
  sPoint.PrintPrivate();
}
```

Output

```
0
1
```

UNIT III **9**

Function and class templates - Exception handling – try-catch-throw paradigm – exception specification – terminate and unexpected functions – Uncaught exception.

PART A — (10 x 2 20 marks)

1. What is a Function templates? Explain. (Nov/dec 2011) (APR/MAY 2011)

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.

```
template<class T,…>
ReturnType FuncName(arguments)
{
        …..//body of the function template
        …..
}
```

2. List five common examples of exceptions. (Nov/dec 2011)

Common examples of exceptions are divide by zero, access to an array outside of its bounds, running out of memory, running out of disk space, abnormal program termination.

3. What is class template? (Nov/dec 2011)

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

```
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);
};
```

4. What are the basically 3 keywords of exception handling mechanism? (Nov/dec 2011)

The exception-handling mechanism uses three blocks

1)try block

2)throw block
3)catch block

The **try-block** must be followed immediately by a handler, which is a **catch-block.**
If an exception is thrown in the **try-block**

5. **What is an exception? .(APR/MAY 2011)**
   Exceptions which occur during the program execution, due to some fault in the input data. Exceptions are run time anomalies or unusual conditions that a program may encounter while executing . it includes divide by zero, access to an array out of its bound, running out of memory.
6. **What is a template? (NOV/DEC 2010)**
   Templates are generic programming concept. It doesn't depend on any data type. It is generic in nature, it is of two type 1. Function template 2. Class template
   1.function template : function templates are generic functions, which work for any data type that is passed to them. The data type is not specified while writing the function. While using the function , we pass the data type and get the required functionality.
   2. class template : class template are also generic , whose data members are generic not for specific. While creating objects for that class we can pass data member of our own.
7. **How is an exception handled in C++?(NOV/DEC 2010)**
   Exception handling mechanism has three building blocks. Try block, throw block ,and catch block .
   Try block – indicating program area where exception can be thrown,
   Trow – for throwing an exception
   Catch block – actually taking an action for the specific exception.

8. **What happens when a raised exception is not caught by catch block? (APR/MAY 2010)**
   When a raised exception not caught by catch block , if the match is not found, the catch block calls a built in function terminate(), which terminate the program execution by calling a built in function abort().
9. **Give the syntax of a pointer to a function which returns an integer and takes arguments one of integer type and 2 of float type. (APR/MAY 2010)**

```
#include<iostream.h>
Void main()
{
int  pointaccess( int d1, float f1);
int a =10;
float x = 2.5;
cout << pointaccess(d1, f1);
int (*pointfun)(int,float);
pointfun = pointaccess;
cout << (*pointfun) (a,x);
}
int pointaccess(int  aa, float ff)
{
Cout  << aa << ff ;
Return( aa + ff );
}
```

EXTRA QUESTIONS:

1. **What is function template?**
   Function templates are generic functions, which work for any data type that is passed to them. The data type is not specified while writing the function. while using that function, we pass the data type and get the required functionality.

2. **List out the draw back of using macros?**

Draw back of using macros are

1. Macros are not visible to the compiler. They are substituted by their body by a pre compiler. If there is some error in the body of the macro, the error is represented in a non user friendly form.
2. The type related information is lost in the macros. Moreover it is not possible to have any related validations in the macros.
3. Macros are evaluated twice. Once when they are copied and the next time when they are executed.
4.

3. What is class template?

A generic class outlined by specification of a generic type using template keyword. The actual class is defined later using this template.

4. What is the use of export keyword?

Export keyword is required for implementing the separate compilation method. When the template definition is proceeded by export, the compiler can manage to compile other files without needing the body of the template functions or member function of the template class.

5. Explain the use of Type name ?

The type name or class keywords are used to define generic name for the type in function or class template.

6. Explain the advantages of template?

Templates are used to increase software reusability, efficiency and flexibility. It increases the reusability without inheritance. Function templates in C++ provides generic functions independent of data type. The advantage of class template is that we can define a generic class that works for any data type that is passed to it as a parameter.

7. What is instantiation?

Generation of either function or class for a specific type using the class of function template is known as instantiation of the class or function.

8. Explain the two models for template compilation?

Compilation models for template are

1. Inline vs non-inline function calls in multiple files.
2. Template instantiation in multiple files.

9. Give an example of multi argument template?

It is possible to have template with more than one argument. Other argument can be generic or normal.
Example
Template <typename type1, typename type2>

10. Explain exception handling mechanism?

The error handling mechanism of C++ is generally referred to as exception handling. Exceptions are classified into two types

1. Synchronous
2. Asynchronous.

   The proposed exception handling mechanism in C++ is designed to handle only synchronous exceptions caused within a program. When a program encounters an abnormal situation for which it is not designed , the user may transfer control to some other part of the program that is designed to deal with the problem.

11. Write the need for exception handling?

Need for exception handling

1. Dividing the error handling
2. Unconditional termination and programmer preferred termination
3. Separating error reporting and error handling
4. The object destroy problem.

12. List out the components of exception handling mechanism?

Components of exception handling mechanism are
Try - try for indicating program area where exception can be thrown.
Throw – for throwing an exception.
Catch – catch for actually taking an action for the specific exception.

13. What is the use of catch all ?

Catch all the expression catch(...) is known as catch all. It is possible to catch all types of exceptions in a single catch section.

14. Define Rethrowing an exception?

Rethrowing the exception is once handled by a handler, can be rethrown to a higher block. This process is known as rethrowing.

15. Define uncaught exception?

Uncaught_exception() is the function to call when we want to check before throwing any exception if some other exception is already on.

16. What is the use of terminate()?

Terminate() is a function to call when the exception handling mechanism does not get any proper handler for a throw exception.

17. What is the use of unexpected()?

Unexpected() is a function to call when the exception handling mechanism encounters an exception not allowed from exception specification.

18. What are the disadvantages of the exception handling mechanism?

Exception handling has some disadvantage that is it adds runtime and demands different style of programming than conventional programming. It is not the tool to replace normal error handling.

19. What is uncaught()?

When the exception is thrown , the destructors of all the objects defined in the try block are called one by one. Then the exception is handled . Now it is said to be caught. If we check for uncaught exception now it is found to be caught.

PARTB (5x 16=80 marks)

1. **Define a DivideBy Zero definition and use it to throw exceptions on attempts to divide by zero**. (**Nov/dec 2011**)

```
#include<iostream.h>
Void mian()
{
int a,b;
Cout<<"\n enter values of a and b ";
Cin>>a;
Cin>>b;
Try
```

```
{
If(b!= 0)
{
Cout<<"\n Result = "<<a/b;
}
Else
{
Throw(b);
}
}
Catch(int i)
{
Cout<<"caught divide by zero exception ";
}
Cout<<"end"
}
```

2. Write a C++ program to demonstrate function template to print array of different types. (Nov/dec 2011)

```
#include<iostream.h>
#include<conio.h>
#define size 5
template<class  T>
void print(T *a, int n)
{
for(int i=0 ; i<n; i++)
cout<<" "<<a[i];
cout<<"\n";
}

void main()
{
int  a[10];
float b[10];
char c[10];
cout<<"\n Enter 5 integers:";
for(int i=0;i<size;i++)
cin>>a[i];
print(a, size);
cout<<"\n Enter 5 float values:";
for(i=0;i<size;i++)
cin>>b[i];
print(b, size);
cout<<"\n enter 5 characters:";
for(i=0;i<size;i++)
cin>>c[i];
print(c,size);
}
```

3. Describe class template with suitable example? (Nov/dec 2011)

```
#include<iostream.h>
#include<conio.h>
#define size 5
template<class  T>
```

```
        void print(T *a, int n)
        {
        for(int i=0 ; i<n; i++)
        cout<<" "<<a[i];
        cout<<"\n";
        }

void main()
{
int  a[10];
float b[10];
char c[10];
cout<<"\n Enter 5 integers:";
for(int i=0;i<size;i++)
cin>>a[i];
print(a, size);
cout<<"\n Enter 5 float values:";
for(i=0;i<size;i++)
cin>>b[i];
print(b, size);
cout<<"\n enter 5 characters:";
for(i=0;i<size;i++)
cin>>c[i];
print(c,size);
}
```

4. Explain exception handling in C++ with suitable example? (Nov/dec 2011)

Program:

```
#include<iostream.h>
#include<process.h>
#define size 3
class queue
{
private:
        int rear;
        int front;
        int s[size];
public:
        queue()
        {
        front=-1;
        rear=0;
        }
void insert(int);
void del();
int isempty();
int isfull();
void display();
};
int queue::isempty()
{
return((rear>front||(front==-1&&rear==0)?1:0);
```

```
}
int queue::isfull()
{
return(rear==front&&front==size?1:0);
}
void queue::insert(int item)
{
try
{
if(isfull())
{
throw"full";
}
else
{
front++
s[front]=item;
}
catch(char *msg)
{
cout<<msg;
}
}
void queue::del()
{
int item;
try
{
if isempty())
throw"empty";
else
{
if(rear!=0)
item=s[rear];
rear=rear+1;
cout<<"\n deleted element is \n"<<item;
}
}
catch(char *msg)
{
cout<<msg;
}
}
void queue::display()
{
if(!isempty())
{
cout<<"contents of queue";
for(int i=front;i>=rear;i--)
{
cout<<s[i]<<"\t";
}
```

```
}
}
int main()
{
int ch,item;
queue q;
do
{
cout<<"\n1.insert2.delete3.exit";
cout<<"enter the choice";
cin>>ch;
switch(ch)
{
case 1:
cout<<"enter the element";
cin>>item;
q.insert(item);
q.display();
break;
case 2:
q.del();
q.display();
break;
case 3:
return 0;
default:
cout<<"try again";
return(0);
}
}
while(ch!=3);
return 0;
}
```

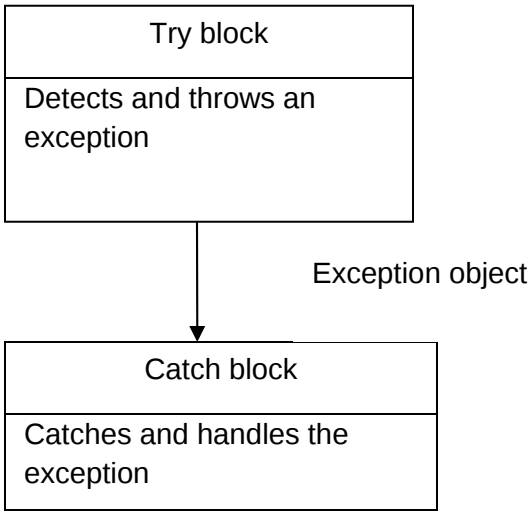
5. (i) **Explain with an example. How exception handling is carried out in C++. (8)**

Exceptions are run time anomalies. They include conditions like division by zero or access to an array outside to its bound etc.

Types: Synchronous exception
Asynchronous exception.

- Find the problem (Hit the exception)
- Inform that an error has occurred. (Throw exception)
- Receive error information (Catch exception)
- Take corrective action (Handle exception)

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 statements is known as try block. When an exception is detected it is thrown using a 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 appropitely.



- try block throwing an exception
- invoking function that generates exception
- throwing mechanism
- catching mechanism
- multiple catch statements
- catch all exceptions
- Rethrowing an exception

General form

```
try
  {
    …..
   throw exception;

    …….
  }
  Catch ( type arg)
 {
   ……
 }
```

Exceptions that has to be caught when functions are used- The form is as follows:

```
Type function (arg list)
{
 ……
  Throw (object)
  …..
}
try
      {
        …..
       Invoke function here;
```

```
    …….
   }
   Catch ( type arg)
  {
   Handles exception here
  }
```

Multiple catch statements:

```
   try
   {
     …..
    throw exception;

     …….
   }
   Catch ( type arg)
  {
   ……// catch block1
  }

    Catch ( type arg)
  {
   ……//catch block2
  }

 Catch ( type arg)
      {
   ……//catch block n
      }
```

Generic exception handling is done using ellipse as follows:

```
  Catch ( . . .)
      {
       ……
      }
```

(ii)Write a class template to insert an element into a linked list.(8) .(APR/MAY 2011)

Program:

```
#include<iostream.h>
#include<conio.h>
#include<process.h>
template<class T>
class list
{
private:
int data;
list *next;
public:
list()
```

```
{
data=0.0;
next=NULL;
}
list(int dat)
{
data=dat;
next=NULL;
}
void insert(list *node);
void display(list *);
};
template<class T>
void list<T>::insert(list<T> *node)
{
list *last=this;
while(last->next)
last=last->next;
last->next=node;
}
template<class T>
void list<T>::display(list<T> *first)
{
list *temp;
cout<<"\nThe Elements in the List:";
for(temp=first;temp;temp=temp->next)
cout<<temp->data<<"\t";
cout<<endl;
}

void main()
{
clrscr();
int choice,data;
list<int> *first=NULL;
list<int> *node;
while(1)
{
cout<<"\n 1.insert\n";
cout<<"\n 2.display\n";
cout<<"\n 3.quit\n";
cout<<"\n choice[1-3]:\n";
cin>>choice;
switch(choice)
{
case 1:
cout<<"\n Enter data:";
cin>>data;
node=new list<int>(data);
if(first==NULL)
first=node;
else
```

```
first->insert(node);
break;
case 2:
first->display(first);
break;
case 3:
exit(1);
}
}
}
```

6. Write a class template to implement a stack

```
  #include<iostream.h>
#include<stdlib.h>
#define size 5
class stack
{
private:
int a[size];
int top;
public:
stack();
void push(int);
int top;
int isempty();
int isfull();
void display();
};

stack::stack()
{
top=0;
for(int i=0;i<size;i++)
a[i]=0;
}
int stack::isempty()
{
return(top==0?1:0);
}
int stack::isfull()
{
return(top==size?1:0);
}
void stack::push(int i)
{
try
{
if(isfull())
{
throw"full";
}
else
```

```
{
a[top]=i;
top++;
}
}
catch(char *msg)
{
cout<<msg;
}
}
int stack::pop()
{
try
{
if(isempty())
{
throw"empty";
}
else
{
return(a[--top]);
}
}
catch(char *msg)
{
cout<<msg;
}
return 0;
}
void stack::display()
{
if(!isempty())
{
for(int i=0;i<top;i++)
cout<<a[i]<<endl;
}
}
int main()
{
stack s;
int num,ch=1;
while(ch!=0)
{
cout<<"\n 1.push2.pop3.display4.exit":
cout<<"enter the choice";
cin>>ch;
switch(ch)
{
case 1:
cout<<"enter the number";
cin>>num;
s.push(num);
```

```
break;
case 2:
cout<<"number to pop is"<<s.pop();
break
case 3:
cout<<"the number are";
s.display()
break;
default:
cout<<"try again";
exit(0);
return 0;
}
}
return 0;
}
```

7. (i) Explain the overloading of template function with suitable example.(8)

Template function to swap any two values

```
#include<iostream.h>
template<class X>
void swap(X &a, X &b)
{
X temp;
temp = a;
a = b;
b = temp;
}
Void main()
{
int a, b;
float x, y;
char c1,c2;
cin >> a >> b;
cout<<"numbers before swap"
cout << a<< b;
Swap(a,b);
Cout<<"numbers after swap"
Cout << a<<b;
Cin >> x>>y;
cout<<"numbers before swap"
Cout << x <<y;
Swap(x,y);
cout<<"numbers after swap"
Cout<<x <<y;
Cin>>c1>>c2;
cout<<"numbers before swap"
Cout<<c1<<c2;
Swap(c1,c2);
cout<<"numbers after swap"
cout<<c1<<c2;
}
```

(ii) Write a function template for finding the minimum value contained in an array. (8) (NOV/DEC 2010)

```
#include<iostream.h>
#include<conio.h>
template<class T>
void read(T *a,int n)
{
for(int i=0;i<n;i++)
cin<<" "<<a[i];
}
Template< class T>
T minimum( T *a, int n))
{
T min = 0;
for( int i=0; i<n; i++)
{
if(a[i] < min)
   min = a[i];
}
 return(min);
}

void main()
{
int a[10];
float b[10];
cout<<"\n Enter 5 integers:";
read(a, 5);
cout << minimum(a,5);
cout<<"\n Enter 5 float values:";
read(b,5);
cout << minimum(b,5);
}
```

8. (i) **List the advantages of exception handling mechanisms. (4)**

The following are some of the advantages of exception handling mechanisms

1. Diving by zero error handling : the library designer and library user
2. Unconditional termination and program preferred termination.
3. Separating error reporting and error handling.
4. The object destroy problem.

(ii) Write a C++ program for the following :

(1) A function to read two double type numbers from keyboard.

(2) A function to calculate the division of these two numbers.

(3) A try block to throw an exception when a wrong type of data is keyed in.

(4) A try block to detect and throw an exception if the condition''divide-by-zero'' occurs.

(5) Appropriate catch block to handle the exceptions thrown. (12) (NOV/DEC 2010)

```
  #include<iostream.h>
 Void main()
 {
```

```
Void data_read() throw ( int , float, char);  // call a function to read two double value
divide();
}
void data_read()
{
double d1, d2;
Cin >> d1 >> d2;
try
{
  throw d1;
  throw d2;
}
Catch ( int i)
{
Cout << " wrong data"
}
Catch ( float f)
{
Cout << "wrong data";
}
Catch( char c)
{
Cout<< "wrong data";
}
}
Void divide()
{
Try
{
if(d2 != 0 )
{   cout >> "result = "<< d1/d2;
}
else
{
Throw d2;
}
Catch( double d)
{
Cout << " divide by zero error has occurred ";
}
}
```

9. Write a C++ program to create a base class called house. There are two classes called door and window available. The house class has members which provide information related to the area of construction, doors and windows details. It delegates responsibility of computing the cost of doors and window construction to door and window classes respectively. Write a C++ program to model the above relationship and find the cost of constructing the house. (APR/MAY 2010)

```
Class door
{
public:
```

```
int no_of_doors;
float d_cost;
void getdoor()
{
Cout<< " enter no of doors and cost of a single door"
Cin >> no_of_doors>>d_cost;
}
float Door_cost()
{
return( no_of_doors * d_cost);
}
}
Class window
{
Public:
Int no_of_window;
Float w_cost;
Void getwindow()
{
Cout << "enter no of windows and cost of a single window"
Cin>>no_of_window>>w_cost;
}
Float window_cost()
{
Return(no_of_window * w_cost);
}
}
Class house : public door, public window
{
Public:
   Int total_area;
   Float const_cost;
 Void getarea()
{
Cout<<"enter size of an area and the constructing cost per square feet"
Cin>>total_area>>const_cost;
float  Total_cost()
 {
Cout<< Const_cost * total_area + window_cost + door_cost ;
}
}
Void main()
 {
House H1;
Door d;
Window w;
  d.getdoor();
  d.door_cost();
  w.get_window();
  w.window_cost();
  h.getarea();
  h.total_cost();
```

```
}
```

10. Write a C++ program to compute the square root of a number. The input value must be tested for validity. If it is negative, the user defined function mysqrt() should raise an exception. (APR/MAY 2010)

```
#include<iostream.h>
Void Mysqrt()
{
cout << " inside the function mysqrt"
try
{
if(x < 0)
  throw x;
else
   Cout << " square root of x = << sqrt(x);
}
void main()
{
int x;
cin >> x;
Mysqrt();
catch( int z)
{
cout << " caught  an exception , the input value is negative"
}
}
```

UNIT IV **9**

Inheritance – public, private, and protected derivations – multiple inheritance - virtual base class – abstract class – composite objects Runtime polymorphism – virtual functions – pure virtual functions – RTTI – typeid – dynamic casting – RTTI and templates – cross casting – down casting .

PART A — (10 x 2 20 marks)

1. Give the use of protected access specifier. (**Nov/dec 2011**)

Protected :

member declared as protected is accessible by :

- A member function with in its class

- Any class immediately derived from it.

It cannot be accessed by the functions outside these two classes.

2. Give the difference between virtual function and pure virtual function. (**Nov/dec2011**)

A virtual function could possibly have an implementation in the class where it is defined, whereas a pure virtual function does not.

virtual:

```
1    class Foo
2    {
3        virtual void bar()
4        {
5            do stuff;
6        }
7    }
```

pure virtual:

```
1    class Foo
2    {
3        virtual void bar() = 0;
4    }
```

3. Define inheritance? (**Nov/dec 2011**)

- The mechanism of deriving a new class from an old one is called inheritance.
- The old class → Base class.
- The new class → Derived class or subclass.
- The derived class inherits some or all of the properties from the base class.

4. What is pure virtual function? (**Nov/dec 2011**)/**(APR/MAY 2011)**

Pure virtual function is declared as avirtual function with its declaration followed by =0.

Syntax:

```
Class Myclass
{
Public:
….
Virtual returntype Functionname(arguments)=0;
….
);
```

5. What is meant by dynamic casting? .(APR/MAY 2011)

Dynamic_cast can be used only with pointers and references to objects. Its purpose is to ensure that the result of the type conversion is a valid complete object of the requested class.

6. Distinguish between early binding and late binding. **(NOV/DEC 2010)**

Early binding: The type of the instance is determined in the compile time. It follows that the static (declared) type of the pointer or reference is used. This is the default for all methods in C++, C#, or Object Pascal.

Late binding: The type of the instance is determined in the run time. It follows that the actual type of the instance is used and the method of this type is called. This is always used for the methods in Java. In C++, the virtual keyword denotes the methods using the late binding.

Late binding gives the class polymorphic behavior. On the other hand, late binding is less effective than early binding, even though the difference may be negligible.

7. **What is visibility mode? What are the different inheritance visibility modes supported by C++? (APR/MAY 2010)**

Base class visibility	Derived class visibility		
	public	private	protected
Private	Not inherited	Not inherited	Not inherited
Protected	protected	private	Protected
public	public	private	protected

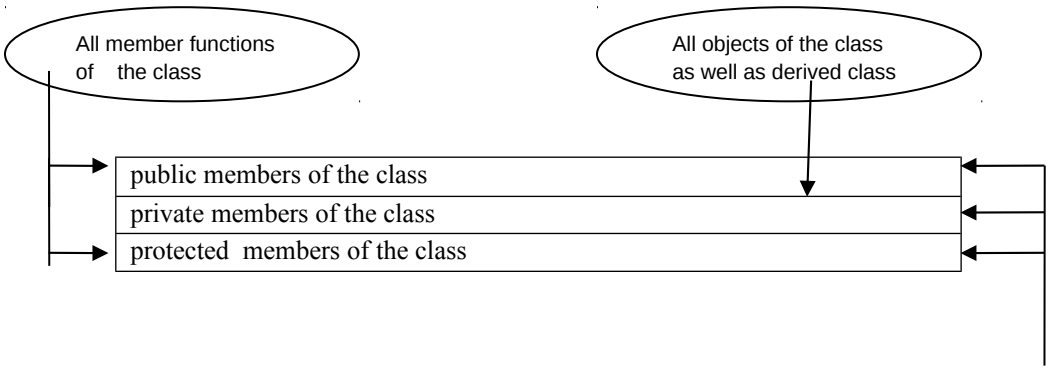
8. **Write the rules for virtual function. (APR/MAY 2010)**

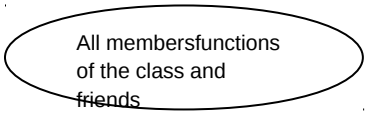
1.the virtual function should be a member of some class
2.they cannot be a static member
3.they are accessed by using object pointers

EXTRA QUESTIONS:

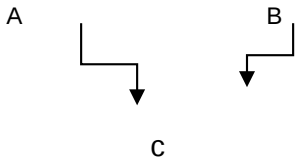
1. **List the different types of inheritance?**
   Different types of inheritance are
   1.single inheritance 2. multiple inheritance 3.hierarchical inheritance 4.multilevel inheritance 5.hybrid inheritance
2. **What is derived class?**
   The mechanism of deriving a new class from an old one is called inheritance. The old class is referred to as the base class and new one is called derived class.
3. **What are the advantages of using inheritance?**
   Advantages of using inheritance
   1. We are in need of extending the functionality of an existing class.
   2. We have multiple classes with some attributes common to them. We would like to avoid problems of inconsistencies between the common attributes.
   3. We would like to model a real world hierarchy in our program in a natural way.
4. **Explain access control?**
   Member functions of a class , member functions of the derived class, friend and object can access different parts of the class. Access for public , private and protected members is different for all entities. Access control describes who can access what and in which form.

The following fig shows the access control





5. **Define multiple inheritance?**
   Derivation of a single class from more than one class is called multiple inheritance.



6. **What do you mean by virtual base class?**
   A virtual base class which is defined as virtual at the time of inheriting it. The compiler takes a note of it and when it is inherited further using multiple inheritance, it ensures that only one copy of the sub object of the virtual inherited class exists in the derived class.
7. **Define abstract class?**
   A class without any object is known as an abstract class.
8. **Define composite objects?**
   Composite or container object is defined as an object which includes other objects as data members.
9. **Define runtime polymorphism?**
   Polymorphism achieved using virtual functions is knows as runtime polymorphism.
10. **Define compile time polymorphism?**
    Polymorphism achieved using operator overloading and function overloading is known as compile time Polymorphism.
11. **Define this pointer?**
    It is the pointer to invoking an object implicitly when a member function is called.
12. **Define virtual function?**
    A function defined with virtual keyword in the base class is known as virtual function. Compiler will decide the exact class pointed to by the base class pointer and call the respective function of that class if the function is defined as virtual.
13. **What do you mean by pure virtual function?**
    A virtual function with '=0' on place of a body in the class. They may not have a body. It is possible to have a body of pure virtual function defined outside the class.

14. **What is RTTI?**
    RTTI is defined as run time type information and it is a mechanism to decide the type of the the object at runtime.
15. **Define static binding?**
    Linking a function during linking phase is known as static binding. Normal functions are statically bound.
16. **Define dynamic binding?**
    Linking the function at run time is known as dynamic binding. Virtual functions are dynamically bound. Though it is possible for the compiler to statically bind the static invocations of virtual functions and also the case where it is possible to decide at compile time. In that case virtual function are also statically bound.
17. **What is cross casting?**
    When a multiply derived class object is pointed to by one of its base class pointers, casting from one base class pointer into another base class pointer is known as cross casting.
18. **What is down casting?**
    Casting from a base class pointer to a derived class pointer is known as down casting.

19. **What is reinterpret_cast?**
    Reinterpret_cast is defined as a casting operator for abnormal casting like int to pointer and pointer to int.
20. **What is the use of typeid operator?**
    This operator can be applied to any object or class or built in data type. It returns the typeinfo object associated with the object under consideration.

PARTB (5x 16=80 marks)

1. **Explain different types of inheritances(Nov/dec 2011)**

Inheritance

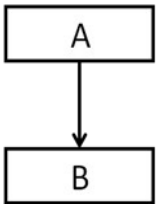
- The mechanism of deriving a new class from an old one is called inheritance.
- The old class → Base class.
- The new class → Derived class or subclass.
- The derived class inherits some or all of the properties from the base class.

Types of Inheritance

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

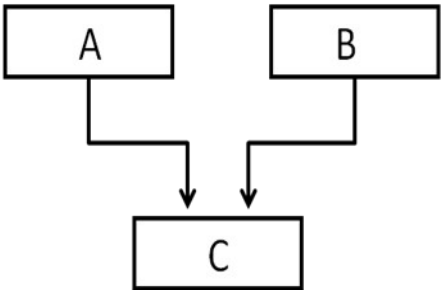
Single Inheritance

A derived class with only one base class



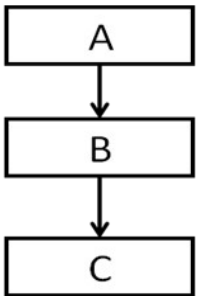
Multiple Inheritance

- A class derived from more than one Base class



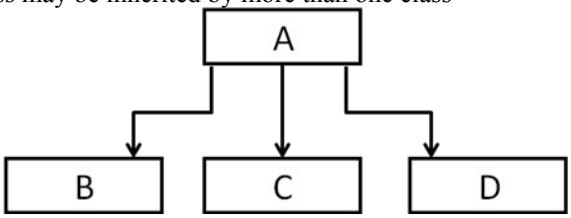
Multilevel Inheritance

- Deriving a class from another Derived class.



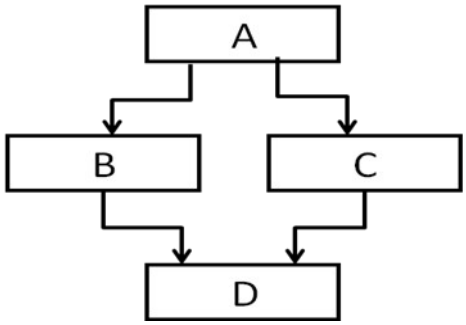
Hierarchical Inheritance

- One class may be inherited by more than one class



Hybrid Inheritance

- A class may be inherited from more than one Derived class



Defining Derived classes

The general form of Defining a Derived class is:

```
Class  derived-classname : visibility mode base class name
{
 ...........
 ...........              // members of derived- class
 ...........
};
```

Examples

```
Class ABC : private XYZ
  {
     members of ABC
  };
Class ABC : public XYZ
  {
     members of ABC
  };
Class ABC :  XYZ
  {
     members of ABC
  };
```

Private derivation

```
Class ABC : private XYZ
   {
       members of ABC
    };
```

- When a base class is privately inherited by a derived class, *Public members* of the base class become *private members* of derived class.
- the public members of base class can only be accessed by the *member functions* of the derived class.
- They are inaccessible to the *objects* of the derived class

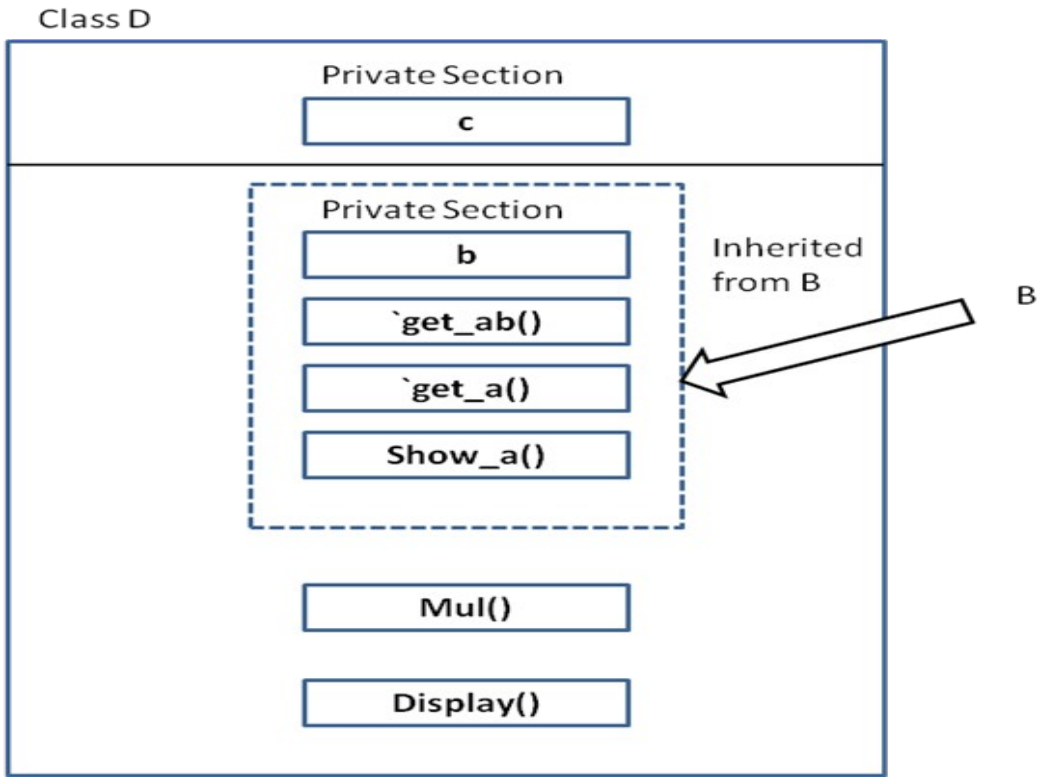
Public Derivation

```
Class ABC : public XYZ
   {
       members of ABC
    };
```

- When a base class is publicly inherited by a derived class, *Public members* of the base class become *public members* of derived class.
- They are accessible to the *objects* of the derived class.
- The private members of base class will never become the members of its derived class.

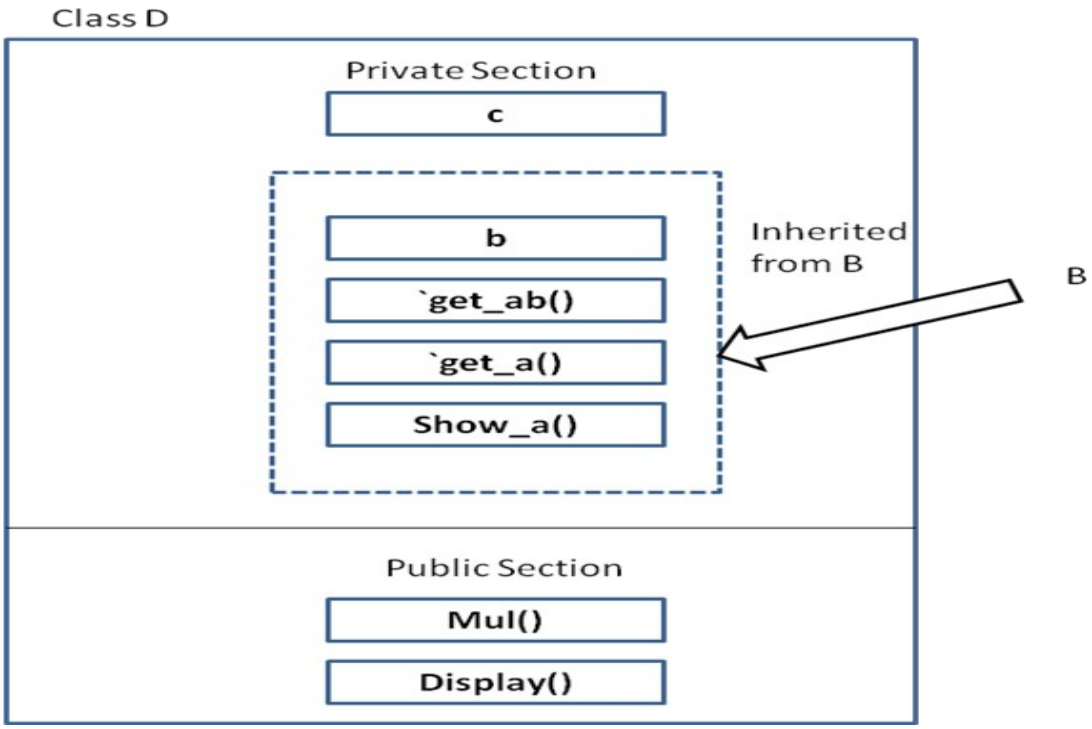
Example

```
class B
{
        int a;
        public:
                int b;
                void get_ab();
                int get_a(void);
                void show_a(void);
};
class D : public B                      // public derivation
{
        int c;
        public:
                void mul(void)
                void display(void);
};
```



Private derivation example

```
class B
{
        int a;
        public:
                int b;
                void get_ab();
                int get_a(void);
                void show_a(void);
};
class D : private B                     // private derivation
{
        int c;
        public:
                void mul(void)
                void display(void);
};
```



Making a private member inheritable

Protected :

member declared as protected is accessible by :

- A member function with in its class
- Any class immediately derived from it.

Note:

It cannot be accessed by the functions outside these two classes.

Types of visibility mode

```
class  alpha
  {
    Private:
     …….
     ……..
    Protected:
    ……..
    ……
    Public:
    ……..
    ……..
 };
```

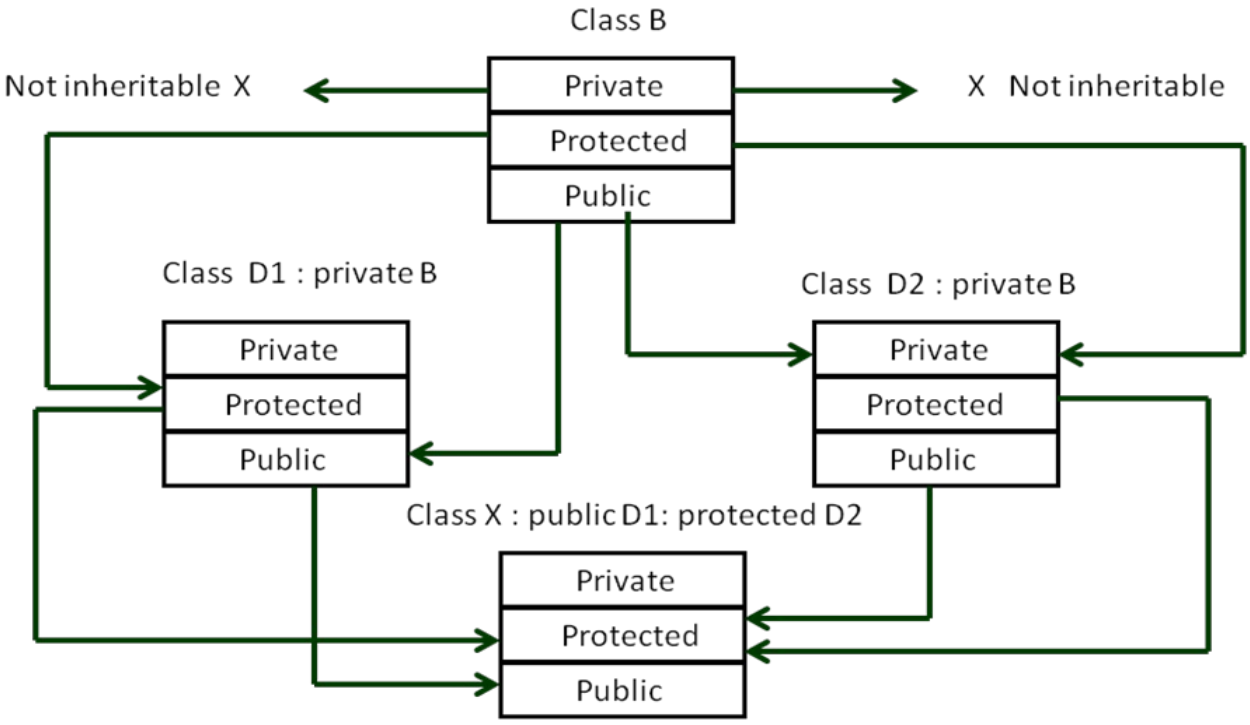


Fig: Effect of inheritance on the visibility of members

The various functions that can access to the private & protected members are:

1. A function that is a friend of the class.
2. A member function of a class that is a friend of the class.
3. A member function of a derived class

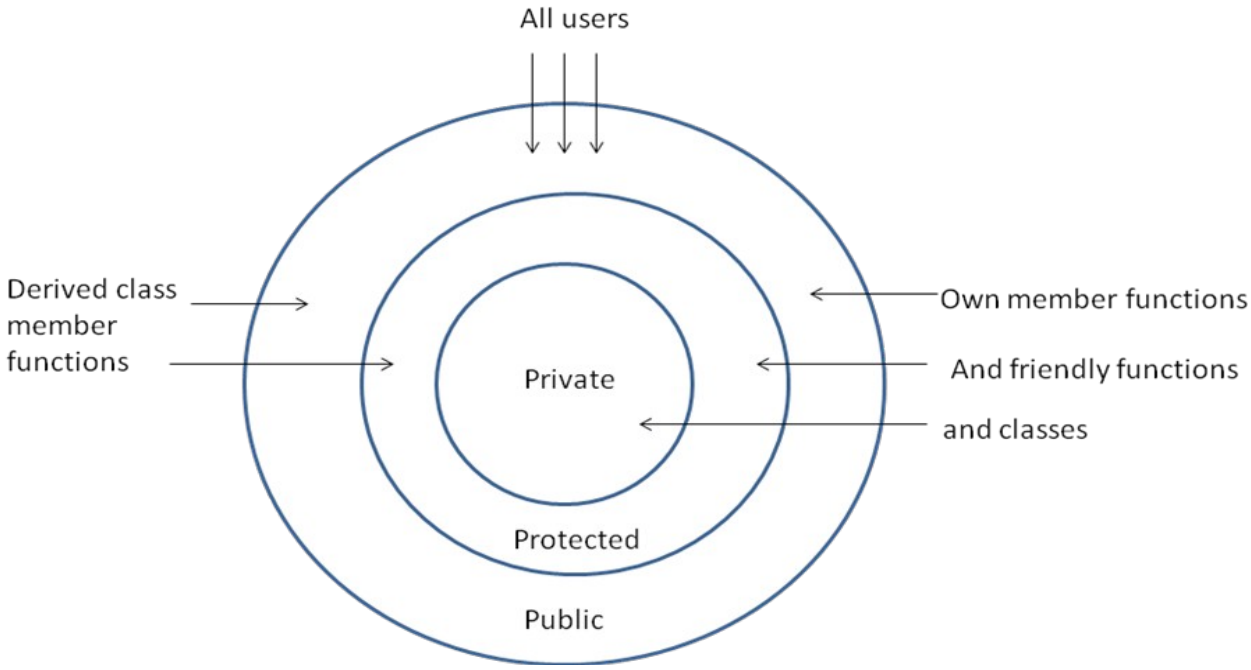
The friend function and member function of friend class can have direct access to both private and protected data.

The member function of derived class can directly access only the protected data

Visibility of inherited Members

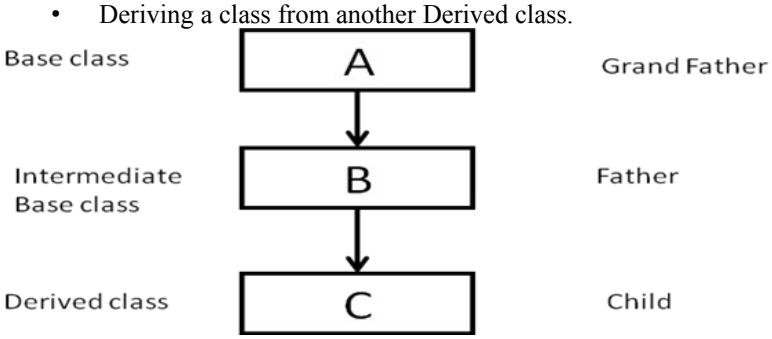
Base class visibility	Derived class visibility		
	public	private	protected

Private	Not inherited	Not inherited	Not inherited
Protected	protected	private	Protected
public	public	private	protected



Multilevel Inheritance

- Deriving a class from another Derived class.



Example for multilevel inheritance

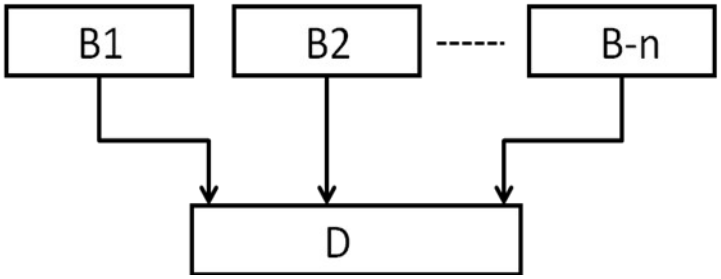
```
#include<iostream.h>
Using namespace std;
Class  student
{
Protected:
Int   roll_number;
Public:
Void get_number(int);
Void put_number(void);
};
Void  student  ::get_number(int a)
{
Roll_number=a;
}
Void  student ::  put_number()
{
Cout<<"Roll number:"<< roll_number;
}
Class test : public student
{
Protected:
Float sub1;
Float  sub2:
Public:
 Void get_marks(float float);
Void put_marks(void);
};
Void test::get_marks(float x,float y)
{
Sub1=x;
Sub2=y;
}
Void test:: put_marks()
{
Cout<<"marks in sub1="<<sub1;
Cout<< "marks in sub2="<<sub2;
}
Class result: public test
{
Float total;
Pulic:
Void display(void);
};
Void result:: display(void)
Total=sub1+sub2;
Put_number();
Put_marks();
Cout<<"Total="<<total;
}
Int  main()
{
```

```
Result student1;
Student1.get_number(111);
Student1.get_marks(85.0,70.0);
Student1.display();
Return 0;
}
```

Output:
Roll number=111
Marks in sub1=85
Marks in sub2=70
Total=155

Multiple Inheritance

- A class derived from more than one Base class



- Syntax of a derived class with multiple base classes

```
Class  D : visibility  B1,visibility  B2....
 {
   .........
   .........  (Body of D)
   .........
 };
```

- Example

```
Class P : public M, public N
   {
      public:
               void  display(void);
   }
```

Example for multiple inheritance

```
#include<iostream.h>
Using namespace std;
Class  M
{
Protected:
Int   m;
Public:
Void get_m(int);
};

Class  N
{
Protected:
```

```
Int   n;
Public:
Void get_n(int);
};
Class  P: public M, public N
{
Public:
Void display(void);
};
Void M:: get_m(int x)
{
M=x;
}

Void N:: get_n(int y)
{
N=y;
}
Void P:: display(void)
{
Cout<<"m="<<m<<"\n";
Cout<<"n="<<n<<"\n";
Cout<<"m*n="<<m*n<<"\n";
}
Int main()
{
P p;
p.get_m(10);
p.get_n(20);
p.display();
return 0;
}
```

Output:
M=10
N=20
M*n=200

Ambiguity Resolution in Multiple inheritance

```
Class M
{
  public:
          void display(void)
         {  cout<<"class M \n";   }
};
Class N
{
  public:
          void display(void)
         {   cout<<"class  N \n";  }
};
Class  P :  public M , public N
```

```
{
  public:
          void display(void)
          {
             M ::display() ;
          }
};
Int main()
{
  P p;
  p.display();
}
```

Ambiguity in single Inheritance

```
Class  A
{
  public:
          void display()
          {
             cout<<  "A";
          }
};
Class  B :public A
{
  public:
          void display()
          {
             cout<<  "B";
          }
};
Int main()
{
  B b;
  b.display();
  b.A::display();
  b.B::display();
   return 0;
}
```

Output:
B
A
B

2. Demonstrate runtime polymorphism with an example. (Nov/dec 2011)

The two types of polymorphism are,

• Compile time polymorphism – The compiler selects the appropriate function for a particular call at the compile time itself. It can be achieved by function overloading and operator overloading.

• Run time Polymorphism - The compiler selects the appropriate function for a particular call at the run time only. It can be achieved using ***virtual functions***

Program to implement runtime polymorphism:

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

```
template<class T>
T sqr(T & n)
{
return(n*n);
}
void main()
{
int a;
float b;
double c;
clrscr();
cout<<"\n\n Enter an Integer : ";
cin>>a;
cout<<"\n Square of a = "<<sqr(a)<<endl;
cout<<"\n\n Enter a Float Value : ";
cin>>b;
cout<<"\n Square of b = "<<sqr(b)<<endl;
cout<<"\n\n Enter a Double Value : ";
cin>>c;
cout<<"\n Square of c = "<<sqr(c);
getch();
}
```

3. **Illustrate virtual function and pure virtual function with suitable example? (Nov/dec 2011)**

Virtual Function:

```
#include<iostream.h>
#include<conio.h>
class abc
{
protected :
int a,b;
public:
void take()
{
cout<<"\nEnter two 'int' values::";
cin>>a>>b;
}
virtual void display()=0;
};
class xyz:public abc
{
public:
void display()
{
cout<<"\nSum for 1st derived class="<
}
};
class mn:public abc
{
public:
void display()
{
```

```
cout<<"\nSum for 2nd derived class:"<
}
};
void main()
{
clrscr();
xyz ob;
mn ob1;
abc *ptr;
ptr=&ob;
cout<<"\nFor object of 1st derived class."<
ptr->take();
ptr->display();
cout<<"\nFor object of 2nd derived class."<
ptr=&ob1;
ptr->take();
ptr->display();
getch();
}
```

Pure Virtual Function:

A virtual function body is known as Pure Virtual Function. In above example we can see that the function is base class never gets invoked. In such type of situations we can use pure virtual functions

Example : same example can re-written

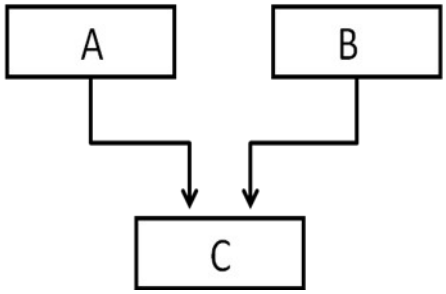
```
class base
{
public:
virtual void show()=0; //pure virtual function
};

class derived1 : public base
{
public:
void show()
{
cout<<"\n Derived 1";
}
};

class derived2 : public base
{
public:
void show()
{
cout<<"\n Derived 2";
}
};

void main()
{
base *b; derived1 d1; derived2 d2;
```

```
b = &d1;
b->show();
b = &d2;
b->show();
}
```

4. **Describe multiple inheritance with example? (Nov/dec 2011)**

Multiple Inheritance

- A class derived from more than one Base class



Example for multiple inheritance

```
#include<iostream.h>
Using namespace std;
Class  M
{
Protected:
Int   m;
Public:
Void get_m(int);
};

Class  N
{
Protected:
Int   n;
Public:
Void get_n(int);
};
Class  P: public M, public N
{
Public:
Void display(void);
};
Void M:: get_m(int x)
{
M=x;
}

Void N:: get_n(int y)
{
N=y;
}
Void P:: display(void)
{
```

```
Cout<<"m="<<m<<"\n";
Cout<<"n="<<n<<"\n";
Cout<<"m*n="<<m*n<<"\n";
}
Int main()
{
P p;
p.get_m(10);
p.get_n(20);
p.display();
return 0;
}
```

Output:
M=10
N=20
M*n=200

Ambiguity Resolution in Multiple inheritance

```
Class M
{
  public:
          void display(void)
         {  cout<<"class M \n";   }
};
Class N
{
  public:
          void display(void)
         {   cout<<"class  N \n";  }
};
Class  P :  public M , public N
{
  public:
          void display(void)
          {
             M ::display() ;
          }
};
Int main()
{
  P p;
  p.display();
 }
```

a. Write short notes on: RTTI, Down casting(3 + 3) .(APR/MAY 2011)

- RTTI stands for Run-time Type Identification.
- RTTI is useful in applications in which the type of objects is known only at run-time.
- Use of RTTI should be minimized in programs and wherever possible static type system should be used.
- RTTI allows programs that manipulate objects or references to base classes to retrieve the actual derived types to which they point to at run-time.

- Two operators are provided in C++ for RTTI.
- **dynamic_cast operator.** The dynamic_cast operator can be used to convert a pointer that refers to an object of class type to a pointer to a class in the same hierarchy. On failure to cast the dynamic_cast operator returns 0.
- **typeid operator**. The typeid operator allows the program to check what type an expression is. When a program manipulates an object through a pointer or a reference to a base class, the program needs to find out the actual type of the object manipulated.
- The operand for both dynamic_cast and typeid should be a class with one or more virtual functions.

EXAMPLE: Demonstrate the RTTI mechanisms in C++

```
#include <iostream>
#include <typeinfo> // Header for typeid operator
using namespace std;

// Base class
class MyBase {
  public:
    virtual void Print() {
        cout << "Base class" << endl;
    };
};

// Derived class
class MyDerived : public MyBase {
  public:
    void Print() {
        cout << "Derived class" << endl;
    };
};

int main()
{
    // Using typeid on built-in types types for RTTI
    cout << typeid(100).name() << endl;
    cout << typeid(100.1).name() << endl;

    // Using typeid on custom types for RTTI
    MyBase* b1 = new MyBase();
    MyBase* d1 = new MyDerived();

    MyBase* ptr1;
    ptr1 = d1;

    cout << typeid(*b1).name() << endl;
    cout << typeid(*d1).name() << endl;
    cout << typeid(*ptr1).name() << endl;

    if ( typeid(*ptr1) == typeid(MyDerived) ) {
    cout << "Ptr has MyDerived object" << endl;
    }

    // Using dynamic_cast for RTTI
```

```
    MyDerived* ptr2 = dynamic_cast<MyDerived*> ( d1 );
    if ( ptr2 ) {
    cout << "Ptr has MyDerived object" << endl;
    }
}

OUTPUT:
i
d
6MyBase
9MyDerived
9MyDerived
Ptr has MyDerived object
Ptr has MyDerived object
```

5.

a. Give the rules for writing virtual functions. (6)

1.The virtual function must be member of class
2. They cannot be static members
3. They are accessed by using object pointers
4. Prototype of base class function & derived class must be same
5. Virtual function in base class must be defined even though it is not used
6. A virtual function can be friend function of another class
7. We could not have virtual constructor
8. If a virtual function is derived in base class, it need not be necessarily redefined in the derived class
9. Pointer object of base class can point to any object of derived class but reverse is not true
10. When a base pointer points to derived class, incrementing & decrementing it will not make it point to the next object of derived class

Write a C++ program to illustrate the use of virtual function. (10) (NOV/DEC 2010)

```
#include<iostream.h>
#include<conio.h>
class abc
{
protected :
int a,b;
public:
void take()
{
cout<<"\nEnter two 'int' values::";
cin>>a>>b;
}
virtual void display()=0;
};
class xyz:public abc
{
public:
void display()
{
cout<<"\nSum for 1st derived class="<
}
```

```
};
class mn:public abc
{
public:
void display()
{
cout<<"\nSum for 2nd derived class:"<
}
};
void main()
{
clrscr();
xyz ob;
mn ob1;
abc *ptr;
ptr=&ob;
cout<<"\nFor object of 1st derived class."<
ptr->take();
ptr->display();
cout<<"\nFor object of 2nd derived class."<
ptr=&ob1;
ptr->take();
ptr->display();
getch();
}
```

6. **What are abstract classes? Write a program having student as an abstract class and create many derived classes such as engineering, science, medical etc., from the student class. Create their object and process them. (APR/MAY 2010)**

A class with no object is called abstract class.

```
#include<iostream.h>
class student
{
Public:
Int regno;
char name[20];
};
class engineering:public student
{
char branch[20];
public:
void get()
{
cin>>regno>>name>>branch;
}
void put()
{
cout<<regno<<name<<branch;
};
class science:public student
{
char branch[20];
```

```
public:
void get()
{
cin>>regno>>name>>branch;
}
void put()
{
cout<<regno<<name<<branch;
};
class medical: public student
{
{
char branch[20];
public:
void get()
{
cin>>regno>>name>>branch;
}
void put()
{
cout<<regno<<name<<branch;
};
void main()
{
engineering e:
e.get();
e.put();
science s;
s.get();
s.put();
medical m;
m.get();
m.put();
}
```

UNIT V **9**

Streams and formatted I/O – I/O manipulators - file handling – random access – object serialization – namespaces - std namespace – ANSI String Objects – standard template library.

PART A — (10 x 2 20 marks)

1. **List the file-open modes. (Nov/dec 2011)**
   ios::in-open the file for reading.
   ios::out-open the file for writing.
2. **What are the three standard template library adapters? (Nov/dec 2011)**
   - Containers
   - Algorithms
   - Iterators
3. **Define namespace? (Nov/dec 2011)/ (APR/MAY 2011)**

Namespaces allow to group entities like classes, objects and functions under a name. This way the global scope can be divided in "sub-scopes", each one with its own name.

The format of namespaces is:

namespace identifier
{
entities
}

Where identifier is any valid identifier and entities is the set of classes, objects and functions that are included within the namespace. For example:

```
1   namespace myNamespace
2   {
3     int a, b;
```

4 }

4. **What is the member function used in manipulating string objects? (Nov/dec 2011)**
   The member function used in manipulating string objects are
   insert()-inserts character at specified location
   erase()- removes characters as specified
   replace()-replace specified characters with a given string
   append()- appends a part of string to another string
5. **Justify the need for object serialization. (APR/MAY 2011)**
   Serialization is simple, it is basically converting a class into binary form so that it can be read later on or sent over a network and then read out of the file or over the network as an object. It is a simple yet powerful concept, and allows an object to retain its form even across a network.
6. **Name the features included in C++ for formatting the output. (NOV/DEC 2010)**
   1. Field Width
   2. Justification in Field
   3. Controlling Precision
   4. Leading zeros
7. **What is file mode? List any four file modes. (NOV/DEC 2010)**
8. **Give the meaning of the flag ios::out. (APR/MAY 2010)**
   ios::out-open the file for writing.
9. **What is a C++ manipulator? (APR/MAY 2010)**
   Manipulators are the functions to manipulate the output formats.

EXTRA QUESTIONS:

1. **What are streams?**
   A stream is a conceptual pipe like structure, which can have one end attached to the program and other end attached by default to a keyboard, screen or a file. It is possible to change where one end is pointing to, while keeping the other end as it is.
2. **List all formatted I/O in C++?**
   The following list of ios functions are called formatted I/O in C++
   Width(), precision(), fill(), setf(), unsetf().
3. **Define manipulator?**
   Manipulator are funtions which are non member but provide similar formatting mechanism as ios functions
4. **Write the difference between manipulators and ios function?**
   The major difference in the way the manipulators are implemented as comp[ared to the ios member functions. The ios member functions return the previous format state which can be used latter if necessary. But the manipulator does not return the previous state.
5. **Explain setf()?**
   The function specifies format flags that controls output display like left or right justification, padding after sign symbol, scientific notation display, displaying base of the number like hexadecimal, decimal, octal etc.
   Ex cout.setf(ios::internal,ios::adjustfield); Cout.setf(ios::scitific,ios::floatfield);
6. **Define Namespace?**
   Namespace is a kind of enclosure for functions, classes and variables to separate them from other entities.
7. **Explain get() and put() function?**
   The classes istream and ostream define two member functions get() and put() respectively to handle a single character input and output operations. The function cin.get() had two different versions. The first version has a prototype void get(char) and the other has prototype char get(void). The fuction cout.put() used to display a character .
8. **Explain read() and write() function?**

The functions read() and write() are used for read and write operations. In addition to string parameter for reading and writing these two functions have additional parameter indicating the size of the string.
Cin.read(string variable, maximum size of the string variable that can be input)
Cin.write(string variable, maximum size of the string variable that can be output)

9. **Explain seekg() and seekp() function?**
   Seekg() – moves get pointer to a specified location
   Seekp() – moves put pointer to a specified location
   Seekg(offset, refposition);
   Seekp(offset,refposition);
10. **Explain tellg() and tellp() function?**
    Tellg() – gives the current location of the get pointer
    Tellp() – gives the current position of the put pointer.
11. **What is global namespace?**
    The global namespace is namespace where every function or class is copied in older C++ by default.
12. **Define Koening lookup?**
    Knoeing lookup is the ability of the compiler to find out exact namespace of the object in use even when not specified by the programmer.
13. **List different way of constructing a string object?**
    A string object can be created in 3 ways.
    1. Defining a string object in a normal way.
    2. Defining a string object using initialization
    3. Defining a string object using a constructor.
       Ex String firststring; // normal defintion
       String secondstring("Hello"); // definition using one agrument constructor
       String thirdstring(secondstring); // definition using initialization
       String fourthstring=firststring; // defining and initializing
14. **Define standard template library?**
    Standard template Library or STL , is a collection of generic software components(generic containers) and generic algorithms, glued by objects called iterators.
15. **Write some example manipulators in C++?**
    The following are some examples of manipulators
    Setw(), setprecision(), setfill(), setiosflags(), resetiosflags()
16. **List out the advantages of readymade software components?**
    There are few advantages of readymade components
    1. Small in size
    2. Generality
    3. Efficient, tested , debugged and standardized
    4. Portability and reusability.
17. **Compare C++ and C strings?**
    The C type string have some problem like assignment using = is not possible, comparison of two string using == is not possible, initializing a string with another is not possible. These problems are solved in C++ by using string objects. Generic algorithms like fine(), replace(), sort() etc are also available for operations on string objects in C++.
18. **Write functions that can help us in finding out different characteristics of the string object?**
    String characteristics can be found out using built in functions available.
    1. The empty() – which checks if the string object contains any data . ex st1.empty() to check if st1 is an empty string object. The function returns a Boolean value. If it returns 1 if the function is empty , else it returns 0.
    2. The size() – function returns the size of the string
    3. The max_size() gives the maximum permissible size of a string in a given system.
    4. Tge resize() which resizes the string by the number supplied as argument.
19. **List out different substring operations for string object?**

The following are some substring operations in C++

1. Find location of a substring or a character in a given string.
2. Find the character at a given location in a given string.
3. Insert a specific substring at specific place.
4. Replacing specific characters by other characters.

20. What is the advantages of using generic algorithm?

Some distinct advantages of using generic algorithms are

1. Programmers are freed from writing routines like sort(), merge(), binary_search(), find() etc. They are available as readymade from STL.
2. The algorithm use the best mechanisms to be as efficient as possible; desighing which may not be possible for most of the programmers.
3. Generic algorithms are standardized and hence , have more acceptability than proprietary algorithms.
4. Similar semantics are designed for all these algorithm.

PARTB (5x 16=80 marks)

1. Write a C++ program that maintains a bank's account information The program adds new accounts, deletes accounts in a text file. Assume that a file credit.dat has been created and the initial data has been inserted. (Nov/dec 2011)

```
// This program reads a random access file sequentially, updates
// data previously written to the file, creates data to be placed
// in the file, and deletes data previously in the file.
#include <iostream>
using std::cerr;
using std::cin;
using std::cout;
using std::endl;
using std::fixed;
using std::ios;
using std::left;
using std::right;
using std::showpoint;

#include <fstream>
using std::ofstream;
using std::ostream;
using std::fstream;

#include <iomanip>
using std::setw;
using std::setprecision;

#include <cstdlib>
using std::exit; // exit function prototype

#include "ClientData.h" // ClientData class definition

int enterChoice();
void createTextFile( fstream& );
void updateRecord( fstream& );
void newRecord( fstream& );
void deleteRecord( fstream& );
```

```
void outputLine( ostream&, const ClientData & );
int getAccount( const char * const );

enum Choices { PRINT = 1, UPDATE, NEW, DELETE, END };

int main()
{
  // open file for reading and writing
  fstream inOutCredit( "credit.dat", ios::in | ios::out );

  // exit program if fstream cannot open file
  if ( !inOutCredit )
  {
    cerr << "File could not be opened." << endl;
    exit ( 1 );
  } // end if

  int choice; // store user choice

  // enable user to specify action
  while ( ( choice = enterChoice() ) != END )
  {
    switch ( choice )
    {
      case PRINT: // create text file from record file
        createTextFile( inOutCredit );
        break;
      case UPDATE: // update record
        updateRecord( inOutCredit );
        break;
      case NEW: // create record
        newRecord( inOutCredit );
        break;
      case DELETE: // delete existing record
        deleteRecord( inOutCredit );
        break;
      default: // display error if user does not select valid choice
        cerr << "Incorrect choice" << endl;
        break;
    } // end switch

    inOutCredit.clear(); // reset end-of-file indicator
  } // end while

  return 0;
} // end main

// enable user to input menu choice
int enterChoice()
{
  // display available options
  cout << "\nEnter your choice" << endl
```

```
      << "1 - store a formatted text file of accounts" << endl
      << "    called \"print.txt\" for printing" << endl
      << "2 - update an account" << endl
      << "3 - add a new account" << endl
      << "4 - delete an account" << endl
      << "5 - end program\n? ";

   int menuChoice;
   cin >> menuChoice; // input menu selection from user
   return menuChoice;
} // end function enterChoice

// create formatted text file for printing
void createTextFile( fstream &readFromFile )
{
   // create text file
   ofstream outPrintFile( "print.txt", ios::out );

   // exit program if ofstream cannot create file
   if ( !outPrintFile )
   {
      cerr << "File could not be created." << endl;
      exit( 1 );
   } // end if

   outPrintFile << left << setw( 10 ) << "Account" << setw( 16 )
      << "Last Name" << setw( 11 ) << "First Name" << right
      << setw( 10 ) << "Balance" << endl;

   // set file-position pointer to beginning of readFromFile
   readFromFile.seekg( 0 );

   // read first record from record file
   ClientData client;
   readFromFile.read( reinterpret_cast< char * >( &client ),
      sizeof( ClientData ) );

   // copy all records from record file into text file
   while ( !readFromFile.eof() )
   {
      // write single record to text file
      if ( client.getAccountNumber() != 0 )  // skip empty records
         outputLine( outPrintFile, client );

      // read next record from record file
      readFromFile.read( reinterpret_cast< char * >( &client ),
         sizeof( ClientData ) );
   } // end while
} // end function createTextFile

// update balance in record
void updateRecord( fstream &updateFile )
```

```
{
   // obtain number of account to update
   int accountNumber = getAccount( "Enter account to update" );

   // move file-position pointer to correct record in file
   updateFile.seekg( ( accountNumber - 1 ) * sizeof( ClientData ) );

   // read first record from file
   ClientData client;
   updateFile.read( reinterpret_cast< char * >( &client ),
      sizeof( ClientData ) );

   // update record
   if ( client.getAccountNumber() != 0 )
   {
      outputLine( cout, client ); // display the record

      // request user to specify transaction
      cout << "\nEnter charge (+) or payment (-): ";
      double transaction; // charge or payment
      cin >> transaction;

      // update record balance
      double oldBalance = client.getBalance();
      client.setBalance( oldBalance + transaction );
      outputLine( cout, client ); // display the record

      // move file-position pointer to correct record in file
      updateFile.seekp( ( accountNumber - 1 ) * sizeof( ClientData ) );

      // write updated record over old record in file
      updateFile.write( reinterpret_cast< const char * >( &client ),
         sizeof( ClientData ) );
   } // end if
   else // display error if account does not exist
      cerr << "Account #" << accountNumber
         << " has no information." << endl;
} // end function updateRecord

// create and insert record
void newRecord( fstream &insertInFile )
{
   // obtain number of account to create
   int accountNumber = getAccount( "Enter new account number" );

   // move file-position pointer to correct record in file
   insertInFile.seekg( ( accountNumber - 1 ) * sizeof( ClientData ) );

   // read record from file
   ClientData client;
   insertInFile.read( reinterpret_cast< char * >( &client ),
      sizeof( ClientData ) );
```

```
   // create record, if record does not previously exist
   if ( client.getAccountNumber() == 0 )
   {
      char lastName[ 15 ];
      char firstName[ 10 ];
      double balance;

      // user enters last name, first name and balance
      cout << "Enter lastname, firstname, balance\n? ";
      cin >> setw( 15 ) >> lastName;
      cin >> setw( 10 ) >> firstName;
      cin >> balance;

      // use values to populate account values
      client.setLastName( lastName );
      client.setFirstName( firstName );
      client.setBalance( balance );
      client.setAccountNumber( accountNumber );

      // move file-position pointer to correct record in file
      insertInFile.seekp( ( accountNumber - 1 ) * sizeof( ClientData ) );

      // insert record in file
      insertInFile.write( reinterpret_cast< const char * >( &client ),
         sizeof( ClientData ) );
   } // end if
   else // display error if account already exists
      cerr << "Account #" << accountNumber
         << " already contains information." << endl;
} // end function newRecord

// delete an existing record
void deleteRecord( fstream &deleteFromFile )
{
   // obtain number of account to delete
   int accountNumber = getAccount( "Enter account to delete" );

   // move file-position pointer to correct record in file
   deleteFromFile.seekg( ( accountNumber - 1 ) * sizeof(ClientData ) );

   // read record from file
   ClientData client;
   deleteFromFile.read( reinterpret_cast< char * >( &client ),
      sizeof( ClientData ) );

   // delete record, if record exists in file
   if ( client.getAccountNumber() != 0 )
   {
      ClientData blankClient; // create blank record

      // move file-position pointer to correct record in file
```

```
      deleteFromFile.seekp( ( accountNumber - 1 ) *
        sizeof( ClientData ) );

      // replace existing record with blank record
      deleteFromFile.write(
        reinterpret_cast< const char * >( &blankClient ),
        sizeof( ClientData ) );

      cout << "Account #" << accountNumber << " deleted.\n";
    } // end if
    else // display error if record does not exist
      cerr << "Account #" << accountNumber << " is empty.\n";
  } // end deleteRecord

  // display single record
  void outputLine( ostream &output, const ClientData &record )
  {
    output << left << setw( 10 ) << record.getAccountNumber()
      << setw( 16 ) << record.getLastName()
      << setw( 11 ) << record.getFirstName()
      << setw( 10 ) << setprecision( 2 ) << right << fixed
      << showpoint << record.getBalance() << endl;
  } // end function outputLine

  // obtain account-number value from user
  int getAccount( const char * const prompt )
  {
    int accountNumber;

    // obtain account-number value
    do
    {
      cout << prompt << " (1 - 100): ";
      cin >> accountNumber;
    } while ( accountNumber < 1 || accountNumber > 100 );

    return accountNumber;
  } // end function getAccount
```

2. Write brief notes on Standard template Library (16) (Nov/dec 2011)

The collection of generic classes and functions is called the Standard Template Library (STL). STL components which are now part of the standard C++ library are defined in the **namespace std.** We must therefore use the **using namespace** directive

using namespace std;

- Components of STL
- Containers

types of container are;

sequence container

- Vector - it allows insertion and deletion at back – it permits direct access- header file is < vector >
- Deque - double ended queue – it allows insertion and deletion at both ends- permits direct access- header file is < deque>
- List - allows insertion and deletion anywhere – header file is < list >

Associative container

- Set - used for storing unique set – it allows rapid look up- bidirectional access - header file is < set >
- Multiset - Used for storing non unique set – header file is < set >
- Map - Used for storing unique key/value – header file is <map>
- Multimap

Derived container

- Stack - Last in first out – no iterator- header file is < stack >
- Queue - First in first out – no iterator- header file is < queue>
- priority queue - first element out is always with the highest priority- header file is <queue>- no iterator

  - Algorithms
  - Iterators
  - Application of Container Classes

3. Illustrate different file operations in C++ with suitable example? (Nov/dec 2011)

The Basic operation on text/binary files are : Reading/writing ,reading and manipulation of data stored on these files. Both types of files needs to be open and close.

How to open File

Using member function Open( )	**Using Constructor**
Syntax Filestream object; Object.open("filename",mode); Example ifstream fin; fin.open("abc.txt")	Syntax Filestream object("filename",mode); Example ifstream fin("abc.txt");

- Mode are optional and given at the end .
- Filename must follow the convention of 8.3 and it's extension can be anyone

How to close file

All types of files can be closed using **close( )** member function

Syntax

```
fileobject.close( );
```

Example

```
fin.close( );     // here fin is an object of istream class
```

Program

Program	ABC.txt file contents

#include<fstream>
using namespace std;
int main()
{
ofstream fout;
fout.open("abc.txt");
fout<<"This is my first program in file handling";
fout<<"\n Hello again";
fout.close();
return 0;
}
	This is my first program in file handling
	Hello again

```
include<fstream>
#include<iostream>
#include<conio.h>
using namespace std;
int main()
{
   ifstream fin;
   char str[80];
   fin.open("abc.txt");
   fin>>str;     // read only first //string from file
   cout<<"\n From File :"<<str;   // as         //spaces is treated
as termination point
   getch();
   return 0;
}
NOTE : To overcome this problem use
  fin.getline(str,79);
```

Detecting END OF FILE

Using **EOF( ) member function**	Using **filestream object**

Syntax
Filestream_object.eof( );
Example
#include<iostream>
#include<fstream>
#include<conio.h>
using namespace std;
int main()
{
char ch;
ifstream fin;
fin.open("abc.txt");
while(!fin.eof()) // using **eof()** function
{
	Example
	// detecting end of file
	#include<iostream>
	#include<fstream>
	#include<conio.h>
	using namespace std;
	int main()
	{
	char ch;
	ifstream fin;
	fin.open("abc.txt");
	while(fin) // file object
	{
	fin.get(ch);

```
      fin.get(ch);
      cout<<ch;
    }
  fin.close();
  getch();
  return 0;}
```

```
     cout<<ch;
   }
  fin.close();
  getch();
  return 0;
}
```

Example **: To read the contents of a text file and display them on the screen**.

Program (using **getline** member function)

```
#include<fstream>
#include<conio.h>
#include<iostream>
using namespace std;
int main()
{
   char str[100];
   ifstream fin;
   fin.open("c:\\abc.txt");
   while(!fin.eof())
    {
       fin.getline(str,99);
       cout<<str;
       }
   fin.close();
   getch();
   return 0;
}
```

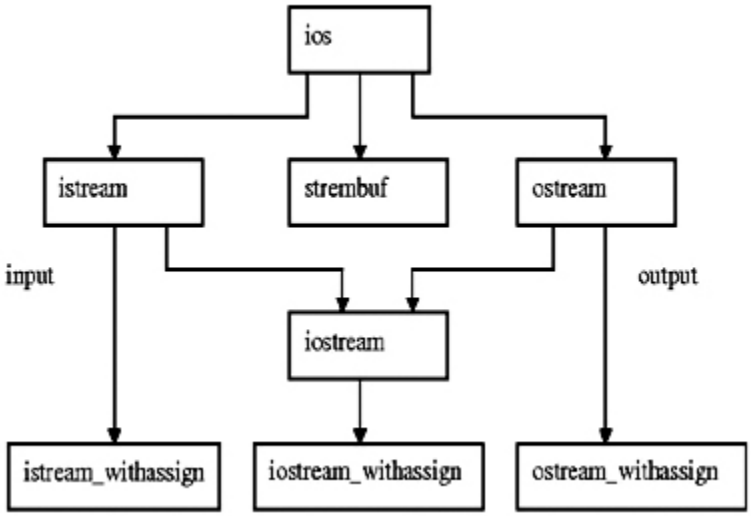
Program (using **get( )** member function)

```
#include<fstream>
#include<conio.h>
#include<iostream>
using namespace std;
int main()
{
   char ch;
   ifstream fin;
   fin.open("file6.cpp");
   while(!fin.eof())
    {
       fin.get(ch);
       cout<<ch;
       }
   fin.close();
   getch();
   return 0;
}
```

4.

a. List the different stream classes supported in C++

Stream classes for console I/O operations:



b. Write a C++ program to read the contents of a text file .(APR/MAY 2011)

```
 #include<fstream>
#include<conio.h>
#include<iostream>
using namespace std;
int main()
{
   char str[100];
   ifstream fin;
   fin.open("c:\\abc.txt");
   while(!fin.eof())
    {
       fin.getline(str,99);
       cout<<str;
       }
   fin.close();
   getch();
   return 0;
}
```

5.

a. Explain the use of any six manipulators with example. (6)

Manipulators are functions specifically designed to be used in conjunction with the insertion (<<) and extraction (>>) operators on stream objects, for example:

```
cout << boolalpha;
```

They are still regular functions and can also be called as any other function using a stream object as argument, for example:

```
boolalpha (cout);
```

Manipulators are used to change formatting parameters on streams and to insert or extract certain special characters.

nput manipulators

ws-Extract whitespaces (manipulator function)

Output manipulators

endl-Insert newline and flush (manipulator function)
ends-Insert null character (manipulator function)
flush-Flush stream buffer (manipulator function)

Parameterized manipulators

These functions take parameters when used as manipulators. They require the explicit inclusion of the header file `<iomanip>`.

setiosflags-Set format flags (manipulator function)
resetiosflags-Reset format flags (manipulator function)
setbase-Set basefield flag (manipulator function)
setfill-Set fill character (manipulator function)
setprecision-Set decimal precision (manipulator function)
setw-Set field width (manipulator function)

b. Discuss in detail the unformatted I/O operations. (10) (NOV/DEC 2010)

- Unformatted I/O performed with read() and write() member functions. They simply input or output as raw byte.

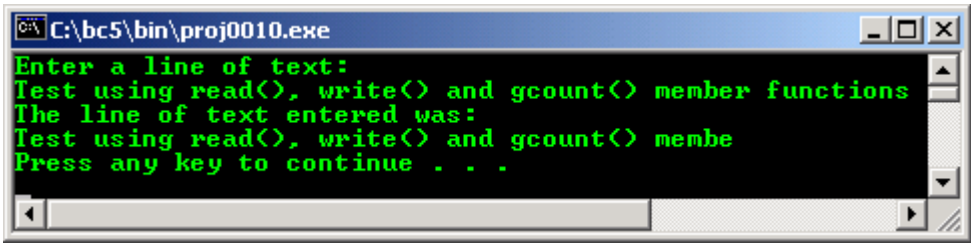
- The read() member function extracts a given number of characters into an array and the write() member function inserts n characters (nulls included). For example:

```
char texts[100];
...
cin.read(texts, 100);
// read 100 characters from input stream and don't append '\0'
```

- Program example:

```
// using read(), write() and gcount() member functions
#include <iostream>
using namespace std;

const int SIZE = 100;

void main(void)
{
        char buffer[SIZE];

        cout<<"Enter a line of text:"<<endl;
        cin.read(buffer,45);
        cout<<"The line of text entered was: "<<endl;
        cout.write(buffer, cin.gcount());
        // the gcount() member function returns
        // the number of unformatted characters last extracted
        cout<<endl;
}
```

Output:



6. **Write a program which copies the contents of one file to a new file by removing unnecessary spaces between words. (APR/MAY 2010)**

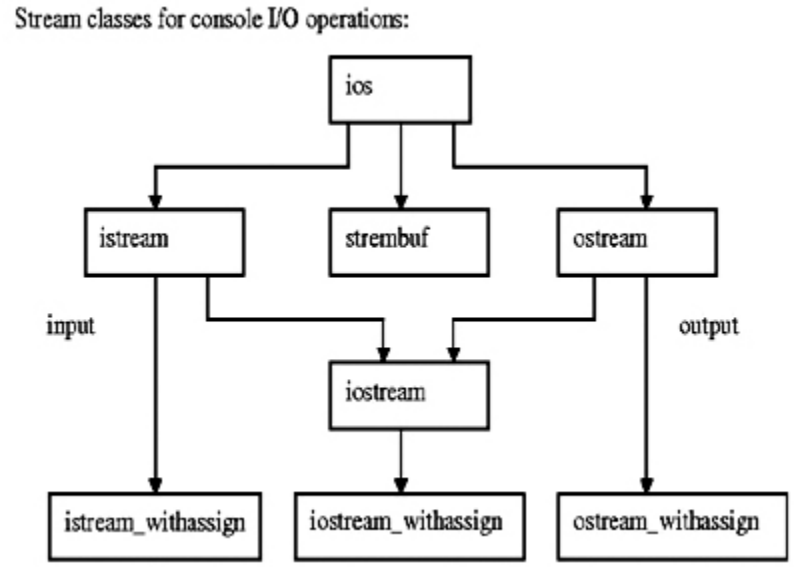
```
#include<iostream.h>
#include<fstream.h>
main()
{
ofstream outfile;
ifstream infile;
infile.open("sample");
outfile.open("sample1");
while(outfile.feof()!=0)
{
char c=outfile.get();
```

```
if(c!="")
{
infile.put();
else
continue;
}
}
outfile.close();
infile.close();
}
```

7. Explain various file stream classes needed for file manipulations. (APR/MAY 2010)

The C++ I/O system contains a hierarchy of classes that are used to define various streams to deal with both the console and disk files. This classes are called String Classes. These classes are declared in the header file iostream.This file should be included in all the programs that communicate with the console unit.

Stream classes for console I/O operations



ios is the base class for istream(input stream)and ostream(output stream) which are ,in turn, base classes for iostream(input/output stream).The class ios is declared as the virtual base class so that only one copy of its members are inherited by the iostream.

The class ios provides the basic support for formatted and unformatted I/O operations. The class istream provides the facilities for formatted and unformatted input while the class ostream provides the facilities for formatted and unformatted output. The class iostream provides the facility for handling both input and output streams. Three classes,namely,istream_withassign,ostream_withassign and iostream_withassign add assignment operators these classes.

ios (General input/output stream class)

1. Contains basic facilities that are used by all other input and output classes.
2. Also contains a pointer to a buffer object(streambuf object).
3. Declares constants and functions that are necessary for handling formatted input and output operation.

istream(input stream)

1. Inherits the properties of ios
2. Declares input functions such as get(),getline() and read().

3. Contains overloaded extraction operator.

ostream(output stream)

1. Inherits the properties of ios
2. Declares output functions such as put() and write().
3. Contains overloaded initiation operator.

iostream(input/output stream)

1. Inherits the properties of ios istream and ostream through multiple inheritance and thus contains all the input and output functions.

Streambuf

1. Provides an interface to physical devices through buffers.

Acts as a base for filebuf class used ios files.

10. . Explain Namespaces with examples.

One of C++'s less heralded additions is addition of *namespaces*, which can be used to structure a program into "logical units". A namespace functions in the same way that a company division might function -- inside a namespace you include all functions appropriate for fulfilling a certain goal. For instance, if you had a program that connected to the Internet, you might have a namespace to handle all connection functions:

```
namespace net_connect
{
  int make_connection();
  int test_connection();
  //so forth...
}
```

You can then refer to functions that are part of a namespace by prefixing the function with the namespace name followed by the scope operator -- *::*. For instance,

net_connect::make_connection()

By enabling this program structure, C++ makes it easier for you to divide up a program into groups that each perform their own separate functions, in the same way that classes or structs simplify object oriented design. But namespaces, unlike classes, do not require instantiation; you do not need an object to use a specific namespace. You only need to prefix the function you wish to call with *namespace_name*:: -- similar to how you would call a static member function of a class.

Another convenience of namespaces is that they allow you to use the same function name, when it makes sense to do so, to perform multiple different actions. For instance, if you were implementing a low-level IO routine and a higher level IO routine that uses that lower level IO, you might want to have the option of having two different functions named "input" -- one that handles low-level keyboard IO and one that handles converting that IO into the proper data type and setting its value to a variable of the proper type.

So far, when we've wanted to use a namespace, we've had to refer to the functions within the namespace by including the namespace identifier followed by the scope operator. You can, however, introduce an entire namespace into a section of code by using a using-directive with the syntax

using namespace *namespace_name*;

Doing so will allow the programmer to call functions from within the namespace without having to specify the namespace of the function while in the current scope. (Generally, until the next closing bracket, or the entire file, if you aren't inside a block of code.) This convenience can be abused by using a namespace globally, which defeats some of the purpose of using a namespace. A common example of this usage is

using namespace std;

which grants access to the std namespace that includes C++ I/O objects cout and cin.

Finally, you can introduce only specific members of a namespace using a using-declaration with the syntax

using *namespace_name*::*thing*;

One trick with namespaces is to use an unnamed namespace to avoid naming conflicts. To do so, simply declare a namespace with the normal syntax, but leave off the identifier; when this is done, you will have

```
namespace
{
  //functions
}
```

and within the namespace you are assured that no global names will conflict because each namespace's function names take precedence over outside function names.

Now, you might ask, how can you actually use anything in that namespace? When your program is compiled, the "anonymous" namespace you have created will be accessible within the file you created it in. In effect, it's as though an additional "using" clause was included implicitly. This effectively limits the scope of anything in the namespace to the file level (so you can't call the functions in that namespace from another other file). This is comparable to the effect of the static keyword.

Renaming namespaces

Finally, if you just don't feel like typing the entire name of namespace, but you're trying to keep to a good style and not use the *using* keyword, you can rename a namespace to reduce the typing:

```
namespace <new> = <old>
```