SYLLABUS

OBJECT ORIENTED PROGRAMMING USING C++
(ETIT-209)

INSTRUCTIONS TO PAPER SETTERS: **MAXIMUM MARKS: 75**

1. Question No. 1 should be compulsory and cover the entire syllabus. This question should have objective or short answer type questions. It should be of 25 marks.
2. Apart from question no. 1, rest of the paper shall consist of four units as per the syllabus. Every unit should have two questions. However, student may be asked to attempt only 1 question from each unit. Each question should be of 12.5 marks.

UNIT I

Introduction: Introducing Object-Oriented Approach related to other paradigms (functional, data decomposition), Characteristics of Object-Oriented Languages.

Basic terms and ideas: Abstraction, Encapsulation, Information hiding, Inheritance, Polymorphism, Review of C, Difference between C and C++, cin, cout, new, delete operators.

[No. of Hrs: 11]

UNIT II

Classes and Objects: Abstract data types, Object & classes, attributes, methods, C++ class declaration, State identity and behavior of an object, Constructors and destructors, instantiation of objects, Default parameter value, Copy Constructor, Static Class Data, Constant and Classes, C++ garbage collection, dynamic memory allocation.

[No. of Hrs. 11]

UNIT III

Inheritance and Polymorphism: Inheritance, Types of Inheritance, Class hierarchy, derivation – public, private & protected, Aggregation, composition vs classification hierarchies, Polymorphism, Type of Polymorphism – Compile time and runtime, Method polymorphism, Polymorphism by parameter, Operator overloading, Parametric polymorphism, Generic function – template function, function name overloading, Overriding inheritance methods. **[No. of Hrs: 11]**

UNIT IV

Files and Exception Handling: Persistant objects, Streams and files, Namespaces, Exception handling, Generic Classes.

Standard Template Library: Standard Template Library, Overview of Standard Template Library, Containers, Algorithms, Iterators, Other STL Elements, The Container Classes, General Theory of Operation, Vectors. **[No. of Hrs: 11]**

MODEL TEST PAPER
FIRST TERM EXAMINATION
FOURTH SEMESTER (2014)
OBJECT ORIENTED PROGRAMMING
[ETCS-210]

Time : 1.5 hrs. **M.M. : 30**

Note: *Attempt in all three Questions. Q.No.1. is compulsory. Assume, missing data if any.*

Q.1.(*a*) What are the benefits of strict type checking. **(2.5)**

Ans. The default type checking scheme for new C++ applications is **STRICT**, as if you had set #define STRICT. Applications created prior to C++Builder 2007 still use the previous default, #define NO_STRICT. The VCL now mangles Windows handle parameters to match the C++ STRICT mangling scheme, and compiling in STRICT mode has the advantages of providing enhanced type-safety and matching the native mangling of the VCL.

The default type checking scheme for new C++ applications is STRICT, as if you had set #define STRICT (this applies to 64-bit Windows as well as 32-bit Windows applications). Applications created prior to C++Builder 2007 still use the previous default, #define NO_STRICT. The VCL now mangles Windows handle parameters to match the C++ STRICT mangling scheme, and compiling in STRICT mode has the advantages of providing enhanced type-safety and matching the native mangling of the VCL.

Q.1.(*b*) What are the difference between pointers to constants and constant pointers? Give example. **(2.5)**

Ans. char* const p :declare p as *const pointer* to char. Once *p* is initialized it will always point to same memory. You cannot make it point to some other address. However you can modify the value at the address pointed to by *p*

const char *p OR char const *p :

declare p as *pointer to const* char. Pointer p points to a const char and can be made to point to any memory location which belongs to const char or non-const char.

Q.1.(*c*) What are read only objects? What is the role of constructor in such objects? **(2.5)**

Ans. Objects declared const have a modified type whose value cannot be changed; they are read-only. For built-in types declared as const, this means that once initialized the compiler refuses to alter their values:

A class with a private static data member (a vector that contains all the characters a-z).A "static constructor" that will run before any instances of the class, and sets up the static data members of the class. It only gets run once (as the variables are read only and only need to be set once) and since it's a function of the class it can access its private members. Add code in the constructor that checks to see if the vector is initialized, and initialize it if it's not, but that introduces many necessary checks and doesn't seem like the optimal solution to the problem.

Q.1.(*d*) What is the difference betwen stack based and heap based object?

Ans. A static object is one that exists from the time it is constructed and created until the end of the program. Stack- and Heap- based objects are thus excluded. Static objects are destroyed when the program exits, *i.e.* their destructors are called when main finishes executing.

The stack is the part of the system memory where all the variables are stored before run-time. The heap is the part of the system memory where all the variables are

stored during run-time, e.g. dynamically allocated memory. This means that if I declare an integer variable i in my code and assign the value of say 123 to it, then that will be stored in my stack, because the compiler knows the value during the compile time (before run-time). But if I define a pointer variable and want to initialize it somewhere else, then that will be stored in my heap, since it is unknown to the compiler at the compile time.

Q.2.(a) Why classes are preferred over structures in C++ while having same features? Write a program which has elements such as roll no., name, date of birth (nested structure/class), and branch of a student. Explain the similarities and differences between structure and class with this example. (6)

Ans. A C struct only has public member variables whereas a C++ class combines member variables with member functions; the methods that operate upon the data. Moreover, each member of a class can be assigned a different level of access from public, protected or private access, thus limiting the member's exposure. This allows classes to hide data and implementation details from outside of the class, exposing only as much as is necessary in order to use the class. Thus the class becomes entirely responsible for the integrity of its data, while its methods act as the gatekeepers to that data.Thus we prefer class rather than struct.

Example:

```
#include <stdio.h>
struct student{
      char name[50];
      int roll;
      char branch;
};
int main(){
     struct student s[10];
     int i;
     printf("Enter information of students:\n");
     for(i=0;i<10;++i)
      {
          s[i].roll=i+1;
          printf("\nFor roll number %d\n",s[i].roll);
          printf("Enter name: ");
          scanf("%s",s[i].name);
          printf("Enter branch: ");
          scanf("%f",&s[i].branch);
          printf("\n");
      }
      printf("Displaying information of students:\n\n");
      for(i=0;i<10;++i)
      {
      printf("\nInformation for roll number %d:\n",i+1);
      printf("Name: ");
      puts(s[i].name);
      printf("date of birth: %.1f",s[i].date of birth);
      }
      return 0;
}
```


Example with class:

```
#include<iostream.h>
#include<fstream.h>
#include<conio.h>
class student
{
int rno;
char name[20];
int rollno;
public:
void get()
{
cout<<"\nEnter data: Roll no Name date of birth,branch\n";
cin>>rno>>name>>date of birth>>branch;
}
void disp()
{
cout<<"\nRoll no\tName\tdate of birth\branch";
cout<<"============================\n";
cout<<rno<<"\t"<<name<<"\t"<<date of birth<<branch<<endl;
}
};
void main()
{
fstream f;
f.open("school.dat",ios::in|ios::out);
student s;
s.get();
f.write((char*)&s,sizeof(s));
while(f.read((char*)&s,sizeof(s)));
s.disp();
getch();
}
```

Q.2.(b) Discuss the memory requirements for classes, objects, data members, member functions, static and non-static data ammbers. (4)

Ans. During runtime, however... in C++, classes define types but (unless you activate RTTI which allows limited introspection into classes) don't generally occupy any memory themselves1—they're just the frameworks for the construction and destruction of objects. Their *methods*, however—the constructors, destructors, instance methods, and class methods, occupy some portion of executable memory, but compilers can and do optimize away any such methods that are not used in the program.

Instances of types (that is, objects as well as primitives like int variables) occupy the bulk of memory in C++, but for their member functions they refer back to their classes. Exactly how much memory an instance of a particular class uses is entirely and utterly an implementation detail, and you should generally not need to care about it.

The C++ standard doesn't explicitly state when static memory is allocated, as long as it is available on first use. That said, it is most likely allocated during program initialization, thus guaranteeing its presence as soon as it is required, without needing special-case code to detect and perform allocation on access.

Q.3.(a) What is runtime memory management? What suport is provided by C++ for this and how does it differs from C's memory management? (5)

Ans. Garbage collection (GC) deals with the management of dynamic memory, with different levels of automation, where the construct called collector, attempts to reclaim garbage (memory that was used by application objects that will never be accessed or mutated again). This is often regarded as an important feature of recent languages, especially if they forbid manual memory management, that is very prone to errors and therefore requires a high level of experience from programmers. Errors due to memory management result mostly in instabilities and crashes that are only noticed at runtime, making them extremely hard to detect and correct.

Usually for C++ programs, they have a distinct runtime, but that runtime is implemented on top of the C run-time, so they delegate as much work as they can- and memory management is a fairly simple one. The only thing C++ has to do on top of C is deal with exceptions, destructors, and whatnot- the actual memory allocation itself can come right from malloc. However, the Standard does not guarantee this at all and it can't be relied upon.

Q.3.(b) Write a program to model Time class using constructors. (5)

Ans.

```
#include
using namespace std;
class Time
{
private:
    int hr;
    int min;
    int sec;
public:
    Time(); // constructor
    Time(int,int,int); // constructor
    void get ();
    void show ();
    Time operator + (Time);
    Time operator - (Time);
};
int main()
{
    Time t1;
    Time t2;
    cout<<"Enter first time:";
    t1.get();
    cout<<"Enter second time:";
    t2.get();
    Time t_sum = t1 + t2;
    Time t_diff = t1 - t2;
    cout<<"Sum";
    t_sum.show();
    cout<<"Diff ";
    t_diff.show();
    system("pause");
```



```
}
Time::Time()
{
    hr=0;
    min=0;
    sec=0;
}
Time::Time(int h,int m,int s)
{
    hr=h;
    min=m;
    sec=s;
}
void Time::get()
{
    cout<<"nEnter hrs:";
    cin>>hr;
    cout<<"Enter min:";
    cin>>min;
    cout<<"Enter sec:";
    cin>>sec;
}
void Time::show()
{
    cout<<"Time is "<<hr<<":"<<min<<":"<<sec<<endl;
}
Time Time::operator + ( Time t)
{
    int h,m,s;
    int sum;
    sum = (hr + t.hr)*3600 + (min + t.min)*60 + sec + t.sec;
    s = sum % 60;
    sum = sum / 60;
    m = sum % 60;
    h = sum / 60;
    return Time(h,m,s);
}
Time Time::operator - ( Time t)
{
    int h,m,s;
    int sum1,sum2,sum;
    sum1 = (hr)*3600 + (min )*60 + sec ;
    sum2 = (t.hr)*3600 + (t.min)*60 + t.sec;
    if (sum1>sum2)
    sum = sum1-sum2;
    else
    sum = sum2-sum1;
    s = sum %60;
```

```
        sum = sum/60;
        m = sum % 60;
        h = sum/60;
        return Time(h,m,s);
    }
    By Azm </sec<<endl;
    </min<<</hr<<
```

Q.4.(a) What is order of construction and destruction of objects? Write a program which these construction and destruction of global, local function and main function objects. **(5)**

Ans. When an object is constructed, its data members are constructed first.

• When the object is destructed, the data members are destructed afterthe destructor for the object has been executed.

• When object A owns other objects (via pointers), remember to explicitly destruct them in A's destructor.

• By default, the default constructor is used for the data members.

```
#include<iostream>
class country
{
  public:
  country()
  {
    std::cout<<"\n Constructor called \n";
  }
  void setNumOfCities(int num);
  int getNumOfCities(void);
  ~country()
  {
    std::cout<<"\n Destructor called \n";
  }
  private:
  int num_of_cities;
};
void country::setNumOfCities(int num)
{
  num_of_cities = num;
}
int country::getNumOfCities(void)
{
  return num_of_cities;
}
int main(void)
{
  country obj;
  int num = 5;
  obj.setNumOfCities(num);
  num = obj.getNumOfCities();
  std::cout<<"\n Number of cities is equal to "<<num;
```



```
  return 0;
}
```

Q.4.(b) What is function overloading? Write overloaded function for computing area of x triangle, circle, and a rectangle. Also discuss ambiguity in these functions. **(5)**

Ans. In some programming languages, **Function overloading** or **method overloading** is the ability to create multiple methods of the same name with different implementations. Calls to an overloaded function will run a specific implementation of that function appropriate to the context of the call, allowing one function call to perform different tasks depending on context.

```
#include <iostream>
// Area of a triangle
int area(int a,int b)
{
    return ½(a*b);
}
// Area of a cirle
double area(double r)
{
    return 3.14*r*r;
}
// Area of a rectangle
long area(long l, int b)
{
    return l*b;
}
int main()
{
    std::cout << volume(10,15);
    std::cout << volume(2.5);
    std::cout << volume(100, 75);
}
Ambiguity in function overloading with example:
void area(int,int);
void area(float,int);
void main()
{
area(10,10);
area(10.0,10);// this line not compile and generate error ambiguity between
area(int,int) and area(float,int)
}
}
```

MODEL TEST PAPER
SECOND TERM EXAMINATION
FOURTH SEMESTER (2014)
OBJECT ORIENTED PROGRAMMING
[ETCS-210]

M.M. : 30

Time : 1.5 hrs.

Note: *Attempt in all three Questions. Q.No.1. is compulsory. Assume, missing data if any.*

Q.1. (a) What are the differences between the access specifiers private and protected? **(3)**

Ans. Private Variables can only be used by that classes members and its friends.

Protected variables can be inherited by other classes, and can be used by the classes members and friends.

Q.1. (b) List the coerators the cannot be overloaded and justify why they cannot be overloaded?

Ans. The standard lists the following operators as non-overloadable:

.* :: ?:

as well as the preprocessing operat

#

and

##

.

size of

is also called an operator, and it can't be overloaded.

Q.1. (c) Can base class, access members of a derived class? Give reasons.

Ans. No the base class cannot access members of a derived class.

Q.1. (d) Justify the need for virtual functions in C++?

Ans. Virtual functions allow us to create a list of base class pointers and call methods of any of the derived classes without even knowing kind of derived class object. For example, consider a employee management software for an organization, let the code has a simple base class Employee , the class contains virtual functions like raiseSalary(), transfer(), promote(),.. etc. Different types of employees like Manager, Engineer, ..etc may have their own implementations of the virtual functions present in base class Employee. In our complete software, we just need to pass a list of employees everywhere and call appropriate functions without even knowing the type of employee. For example, we can easily raise salary of all employees by iterating through list of employees. Every type of employee may have its own logic in its class, we don't need to worry because if raiseSalary() is present for a specific employee type, only that function would be called.

Q.1. (e) Explain how the compiler processes calls to a function template. **(5 x 2)**

Ans. C++ does not compile the template function directly. Instead, at compile time, when the compiler encounters a call to a template function, it replicates the template function and replaces the template type parameters with actual types! The function with actual types is called a function template instance.

Template functions will work with both built-in types (eg. char, int, double, etc..) and classes, with one caveat. When the compiler compiles the template instance, it

compiles it just like a normal function. In a normal function, any operators or function calls that you use with your types must be defined, or you will get a compiler error. Similarly, any operators or function calls in your template function must be defined for any types the function template is instantiated for.

Q.2. (a) What are the limitations of overloading unary increment/decrement operator? How are they overcome?

Ans. First, note that we've distinguished the prefix from the postfix operators by providing an integer dummy parameter on the postfix version. Second, because the dummy parameter is not used in the function implementation, we have not even given it a name. This tells the compiler to treat this variable as a placeholder, which means it won't warn us that we declared a variable but never used it.

The postfix operators, on the other hand, need to return the state of the class before it is incremented or decremented. This leads to a bit of a conundrum - if we increment or decrement the class, we won't be able to return the state of the class before it was incremented or decremented. On the other hand, if we return the state of the class before we increment or decrement it, the increment or decrement will never be called.

The typical way this problem is solved is to use a temporary variable that holds the value of the class before it is incremented or decremented. Then the class itself can be incremented or decremented. And finally, the temporary variable is returned to the caller. In this way, the caller receives a copy of the class before it was incremented or decremented, but the class itself is incremented or decremented. Note that this means the return value of the overloaded operator much be a non-reference, because we can't return a reference to a local variable that will be destroyed when the function exits. Also note that this means the postfix operators are typically less efficient than the prefix operators because of the added overhead of instantiating a temporary variable and returning by value instead of reference.

Q.2. (b) Explain how base class member functions can be invoked in a derived class if the derived class also has a member function with the same name.

Ans. A virtual function is a member function that you expect to be redefined in derived classes. When you refer to a derived class object using a pointer or a reference to the base class, you can call a virtual function for that object and execute the derived class's version of the function.

Virtual functions ensure that the correct function is called for an object, regardless of the expression used to make the function call.

Suppose a base class contains a function declared as virtual and a derived class defines the same function. The function from the derived class is invoked for objects of the derived class, even if it is called using a pointer or reference to the base class. The following example shows a base class that provides an implementation of the PrintBalance function and two derived classes

```
// deriv_VirtualFunctions.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;
class Account {
public:
  Account( double d ) { _balance = d; }
  virtual double GetBalance() { return _balance; }
  virtual void PrintBalance() { cerr << "Error. Balance not available for base type."
<< endl; }
```

```
    private:
      double _balance;
    };
    class CheckingAccount : public Account {
    public:
      CheckingAccount(double d) : Account(d) {}
      void PrintBalance() { cout << "Checking account balance: " << GetBalance() <<
endl; }
    };
    class SavingsAccount : public Account {
    public:
      SavingsAccount(double d) : Account(d) {}
      void PrintBalance() { cout << "Savings account balance: " << GetBalance(); }
    };
    int main() {
      // Create objects of type CheckingAccount and SavingsAccount.
      CheckingAccount *pChecking = new CheckingAccount( 100.00 );
      SavingsAccount  *pSavings  = new SavingsAccount( 1000.00 );
      // Call PrintBalance using a pointer to Account.
      Account *pAccount = pChecking;
      pAccount->PrintBalance();
      // Call PrintBalance using a pointer to Account.
      pAccount = pSavings;
      pAccount->PrintBalance();
    }
```

Q.3. (*a*) 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 objects and process them.

Ans. An abstract class is one that is not used to create objects. An abstract class is designed only to act as a base class (to be inherited by other classes). It is a design concept in program development and provides a base upon which other classes may be built.

```
# include <isotream>
using mamespace std;
class student
{
   protected:
        int roll-number;
   public:
       void get-number (int a)
       roll-number = a;
}
void put-number (void)
{
cout <<"Roll. No.: " <<roll-number <<"\n";
```



```
}
};
    class engineering : virtual public student
    {
      proteced:
            float part 1, part 2;
      public:
            Voil get_marks (float x, float y)
            {
              part 1 = x;   part 2 = y;
            }
            void put_marks (void)
            {
              cout <<"Marks obtained:" <<"\n";
                   <<"part 1 = "<<part 1<<"\n"
                   <<"part 2 = "<< part 2 << "\n"
            }
          };
class medical : virtual public student
{
  protected:
            float part 1, part 2;
public:
      void get_mark (Heat x, float y)
{
  part 1 = x; part 2 = y;
}
void put-marks (void)
cout <<"marks obtained :" << "\n"
     <<"part 1 =" <<part 1 << "\n"
     <<"part 2 = <<part << "\n"
```

Q.3. (*b*) What are virtual destructors? How they differ from normal destructors? Can constructors be declared as virtual constructors? Give reasons.

Ans. C++ destructor is generally used to deallocate memory and do some other clean up for a class object and its class members whenever an object is destroyed.

Virtual destructors in C++ used to deallocate memory which is allocated due to virtual class. This uses virtual keyword to destroy the object and used for clean up process.

No, the constructors cannot be declared as virtual constructors. As constructors are used to initiate the objects, there is no need of virtual constructor. If we will use this virtual constructor the name cannot be initialized.

```
    class result; public engineering, public medical.
    {
    float total;
public:
```

```
Void display (void);
};
Void result :: display (void)
{
    total = part 1 + part 2;
    put_number ();
    put_marks ();
    cout<<"Total Score:" <<total <<"\n";
}
    int main ()
{
    result stu 1;
    stu 1. get number (346)
    stu 1. get marks (40, 50);
    stu 1. display ();
 return 0;
}
```

In the above program the student class is an abstract class since it was not used to create any objects.

Q.4.(a) What is a function template? Write a function template for finding the largest number in a given array. The array parameter must be of generic-data types.

Ans. Templates are the features of the C++ programing language that allows functions and classes to operate with generic types. This allows a function or class to works on many different date types without being rewritten for each one.

Templates one of great utility to programmers in C++, especially when combined with multiple inheritance and operator overloading. A function template behaves like a function except that the template can have arguments of very different types; it represents a family of function.

```
template <typename Element Type>
Element Type largest (Element Type array [ ], int num Elements)
{
Element Type biggest = array [0];
For ((i + i = i); i < num Elements; i++).
if (array [i] > biggest)
biggest = array [i];
return biggest;
}
int main ()
{
    double x[10] = {1.1, 4.4, 3.3, 5.5, 2.2};
    cout <<"largest value in X is" << largest (X, 5);
    int num [20] = {2, 3, 4, 1};
    cout << "largest value in num is" <<largest (num, 4);
}
```


Q.4. (b) What is data conversion? Write a program to convert basic data types to an object of a class.

Ans. Data type conversion can be done in different ways implicitly and explicitly i.e. changing an entity of one data type into another. In OOPS type conversion allows programs to treat objects of one type as one of their ancestor types to simplify interacting with them

Program to Illustrate data conversion

```
# include <iostream.h>
#include <string.h>
#include <stdio.h>
#include <conio.h>
private:
    int day;
    int month;
    int year;
public:
doing ()
    {
        day = month = year = 0;
    }
doing (int, int, int);
    int get-day ()
    {
        return day;
    }
int get_month ()
    {
        return month;
    }
int get_year ()
    {
        return year;
    }
void show_date ()
    {
        cout <<"\it Date is:" <<day <<":"<<month<<":"<<year<<endl;
    };
    private :
        char str [9];
    public:
        date ()
    {
        strepy (str, s);
    }
date (dmy)
    void show-date ()
    {
```

```
        count <<"\t Date is "<<Str <<endl;
    }
};
dmy :: dmy (int d, int m, int y)
{
    int d = t. get_day ();
    int m = t. get_month ();
    int y = t. get_year ();
    char temp [3] = {'10'}
    itoa (d, str, 10);
    struct (str, "1"),
    itor (m, temp, 10);
    struct (str, "1");
    itoa (y, temp, 10);
    struct (Str, temp);
}
    main ()
{
    clrscr ()
    date d_1;
    dmy d – 2(6, 10, r1);
    d_1 = d_2;
    cout <<"\n value of d_1 is :"<<endl;
    d-1. show date ();
    count <<"\n value of d-2 is: "<<endl;
    d_2.Show_date ();
    getch ();
    return 0;
```

MODEL TEST PAPER
END TERM EXAMINATION
FOURTH SEMESTER (2014)
OBJECT ORIENTED PROGRAMMING
[ETCS-210]

Time : 3 hrs. **M.M. : 75**

Note: *Attempt Q.No. 1 which is compulsory and any two more questions from remaining. Each question carries 10 marks.*

Q.1. (a) Differentiate between Scan P () ion C and Cion in C++.

Ans: scanf is a function (available in C and C++)
cin is an istream object (C++ only)

Q.1. (b) Show pictrially an example of multi level inheritance.

Ans: Multilevel Inheritance is a method where a derived class is derived from another derived class.

```
#include <iostream.h> class mm
{
protected:
int rollno;
public:
void get_num(int a)
{ rollno = a; }
void put_num()
{ cout << "Roll Number Is:\n"<< rollno << "\n"; }
}; class marks : public mm
{
protected:
int sub1;
int sub2;
public:
void get_marks(int x,int y)
{
sub1 = x;
sub2 = y;
}
void put_marks(void)
{
cout << "Subject 1:" << sub1 << "\n";
cout << "Subject 2:" << sub2 << "\n";
}
}; class res : public marks
{
protected:
float tot;
public:
void disp(void)
{
tot = sub1+sub2;
put_num();
put_marks();
```

```
cout << "Total:"<< tot;
}
}; int main()
{
res std1;
std1.get_num(5);
std1.get_marks(10,20);
std1.disp();
return 0;
}
```

Q.1. (c) What are the different types of access specifiers in C++?

Ans. Different access specifiers in C++ are.

Private: It is default one and can be access from class member of the same class.

Protected: The protected members can be accessed from member functions of the same class or friend classes or from the members of their immediate derived class.

Public: The public members can be access from anywhere the class is visible.

Ans: What are the limitations of overloading = operator.

The following restrictions apply to operator overloading:

1. Invention of new operators is not allowed. For example:

void operator @ (int) ; //illegal, @ is not a built-in operator or a type name

2. Neither the precedence nor the number of arguments of an operator may be altered. An overloaded && for example, must have exactly two arguments—just like the built-in && operator.

3. The following operators cannot be overloaded:

Direct member access operator.....

De-reference pointer to class member operator.....

Scope resolution operator..... ::

Conditional operator.....?:

Sizeof operator..... sizeof

Ans: How does a compiler process call to a class template?

Ans. It's worth taking a brief look at how template functions are implemented in C++, because future lessons will build off of some of these concepts. It turns out that C++ does not compile the template function directly. Instead, at compile time, when the compiler encounters a call to a template function, it replicates the template function and replaces the template type parameters with actual types! The function with actual types is called a **function template instance.**

Q.1.(*f*) Define virtual base class?

Ans. An ambiguity can arise when several paths exist to a class from the same base class. This means that a child class could have duplicate sets of members inherited from a single base class.

C++ solves this issue by introducing a virtual base class. When a class is made virtual, necessary care is taken so that the duplication is avoided regardless of the number of paths that exist to the child class.

Q.1.(*g*) Write down the syntax of try,throw and catch?

```
try
{
buff = new char[1024];
if(buff == 0)
```



```
throw "Memory allocation failure!";
}
//catch what is thrown...
catch(char* strg)
{
cout<<"Exception raised: "<<strg<<endl;
}
```

- In grammar form:

The try-block:

try

{compound-statement handler-list

handler-list here

The throw-expression:

throw expression

}

{

The handler:

catch (exception-declaration) compound-statement

exception-declaration:

type-specifier-list here

}

Q.2 (*a*)List the different datatypes in C++?

Ans. C++ offer the programmer a rich assortment of built-in as well as user defined data types. Following table lists down seven basic C++ data types:

T ype	Keyword
Boolean	bool
Character	char
Integ er	int
Floating point	float
Double floating point	double
Valueless	void
Wide character	wchar_t

Several of the basic types can be modified using one or more of these type modifiers:

sig ned

unsigned

short

long

Q.2.(*b*) What are the difference between class and structure in C++?

Ans. In C++, the only difference between a struct and a class is that struct members are public by default, and class members are private by default.

However, as a matter of style, it's best to use the struct keyword for something that could reasonably be a struct in C (more or less POD types), and the class keyword if it uses C++-specific features such as inheritance and member functions.

The members and base classes of a struct are public by default, while in class, they default to private. Note: you should make your base classes *explicitly* public, private, or protected, rather than relying on the defaults.

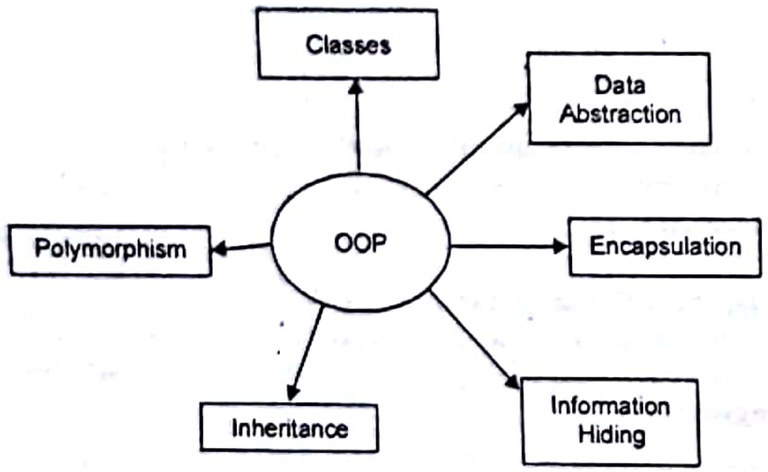
A struct simply *feels* like an open pile of bits with very little in the way of encapsulation or functionality. A class *feels* like a living and responsible member of society with intelligent services, a strong encapsulation barrier, and a well defined interface. Since that's the connotation most people already have, you should probably use the struct keyword if you have a class that has very few methods and has public data (such things *do* exist in well designed systems!), but otherwise you should probably use the class keyword.

Q.2(c) Write a program inC++ to illustrate the concept of call by reference?

Ans. ***Call By Reference:*** When a function is called by the reference then the values those are passed in the calling functions are affected when they are passed by Reference Means they change their value when they passed by the References. In the Call by Reference we pass the Address of the variables whose Arguments are also Send. So that when we use the Reference then, we pass the Address of the Variables.

Q.3 (a) Discuss the concept of object oriented programming features?

Features of Object Oriented Programming (OOP)



FEATURES OF OOP:

1. Object
2. Class
3. Data Hiding and Encapsulation
4. Dynamic Binding
5. Message Passing
6. Inheritance
7. Polymorphism

Brief Explanation of Points:

1. Object: Object is a collection of number of entities. Objects take up space in the memory. Objects are instances of classes. When a program is executed , the objects interact by sending messages to one another. Each object contain data and code to manipulate the data. Objects can interact without having know details of each others data or code.

2. Class: Class is a collection of objects of similar type. Objects are variables of the type class. Once a class has been defined, we can create any number of objects belonging to that class. Eg: grapes bannans and orange are the member of class fruit. Example:

Fruit orange;

In the above statement object mango is created which belong to the class fruit.

Note: Classes are user define data types.

3. Data Abstraction and Encapsulation:

Combining data and functions into a single unit called **class** and the process is known as **Encapsulation**.Data encapsulation is important feature of a class. Class contains both data and functions. Data is not accessible from the outside world and only those function which are present in the class can access the data. The insulation of the data from direct access by the program is called data hiding or information hiding. Hiding the complexity of proram is called **Abstraction** and only essential features are represented.In short we can say that internal working is hidden.

4. Dynamic Binding: Refers to linking of function call with function defination is called binding and when it is take place at run time called dynamic binding.

5. Message Passing: The process by which one object can interact with other object is called message passing.

6. Inheritance: it is the process by which object of one class aquire the properties or features of objects of another class. The concept of inheritance provide the idea of reusability means we can add additional features to an existing class without Modifying it. This is possible by driving a new class from the existing one. The new class will have the combined features of both the classes.

Example: Robine is a part of the class flying bird which is again a part of the class bird.

7. Polymorphism: A greek term means ability to take more than one form. An operation may exhibite different behaviours in different instances. The behaviour depends upon the types of data used in the operation.

Example:

- Operator Overloading
- Function Overloading

Q.3. (b) Write the syntax of new and delete operator?

Ans. The syntax for new is:

p_var = new type_name;

where p_var is a previously declared pointer of type type_name. type_name can be any basic data type or user-defined object (enum, class, and struct included). If type_name is of class type, the default constructor is called to construct the object.

To initialize a new variable created via new, use the following syntax:

p_var = new type(initializer);

where initializer is the initial value assigned to the new variable, or if type is of class type, initializer is the argument(s) to a constructor.

Example

```
int *p_var = nullptr; // new pointer declared
p_var = new int;      // memory dynamically allocated
/* .....
other code
.....*/
delete p_var;         // memory freed up
p_var = nullptr;      // pointer changed to nullptr (null pointer)
```

Arrays allocated with new [] can be similarly deallocated with delete []:

Q.4.(a) Explain Constructers.Explain how a base class constructor can be called from a derived class?

Ans. In class-based object-oriented programming, a **constructor** in a class is a special type of subroutine called to create an object. It prepares the new object for use, often accepting arguments that the constructor uses to set required member variables.

A constructor resembles an instance method, but it differs from a method in that it has no explicit return type, it is not implicitly inherited and it usually has different rules for scope modifiers. Constructors often have the same name as the declaring class. They have the task of initializing the object's data members and of establishing the invariant of the class, failing if the invariant is invalid. A properly written constructor leaves the resulting object in a *valid* state. Immutable objects must be initialized in a constructor.

Suppose a base class is declared as:

```
class Base{
--
}; and a derived class
class Derived : public Base{
--
```

} does this derived class always call the default constructor of the base class? i.e. the constructor that takes no parameters? For example If I define a constructor for the base class:

```
Base(int newValue);
```

but I do not define the default constructor(the parameterless constructor):

```
Base();
```

Q.4.(*b*) Write a program in C++ to find the largest of three numbers by using friend function?

Ans.

```
#include <iostream>
#include <algorithm>
using namespace std;
template <class T> class Num3;
template <typename V> V num3max(const Num 3<V>&);
template <class T>
class Num3{
public:
  Num3() : _a(0), _b(0), _c(0) {}
  Num3(T a, T b, T c): _a(a), _b(b), _c(c) {}
private:
  template <typename V> friend V num3max(const Num3<V>&);
  T _a, _b, _c;
};
int main (int argc, char *argv[]) {
  Num3<int> n0(4, 9, 2);
  Num3<float> n1(1.2, 0.5, 1.8);
  cout <<num3max(n0) <<endl;
  cout <<num3max(n1) <<endl;
  return 0;
}
template <typename V>
V num3max (const Num3<V>&n) {
  return max(n._a, max (n._b, n._c));
}
```


Q.5(*a*) Explain garbage collection in C++?

Ans. Garbage collection is often portrayed as the opposite of manual memory management, which requires the programmer to specify which objects to deallocate and return to the memory system. However, many systems use a combination of approaches, including other techniques such as stack allocation and region inference. Like other memory management techniques, garbage collection may take a significant proportion of total processing time in a program and can thus have significant influence on performance.

In computer science, **garbage collection** (GC) is a form of automatic memory management. The *garbage collector*, or just *collector*, attempts to reclaim *garbage*, or memory occupied by objects that are no longer in use by the program. Garbage collection was invented by John McCarthy around 1959 to solve problems in Lisp.

Resources other than memory, such as network sockets, database handles, user interaction windows, and file and device descriptors, are not typically handled by garbage collection. Methods used to manage such resources, particularly destructors, may suffice to manage memory as well, leaving no need for GC. Some GC systems allow such other resources to be associated with a region of memory that, when collected, causes the other resource to be reclaimed; this is called *finalization*. Finalization may introduce complications limiting its usability, such as intolerable latency between disuse and reclaim of especially limited resources, or a lack of control over which thread performs the work of reclaiming.

Q.5 (*b*)Write a program in C++ to illustrate how an object can be used as a function argument?

Ans. A typical use of a function object is in writing callback functions. A callback in procedural languages, such as C, may be performed by using function pointers. However it can be difficult or awkward to pass a state into or out of the callback function. This restriction also inhibits more dynamic behavior of the function. A function object solves those problems since the function is really a façade for a full object, carrying its own state.

Consider the example of a sorting routine that uses a callback function to define an ordering relation between a pair of items:

```
#include <stdlib.h>

/* Callback function, returns < 0 if a < b, > 0 if a > b, 0 if a == b */
int compareInts(const void* a, const void* b)
{
 return *(const int *)a - *(const int *)b;
}
...
// prototype of qsort is
// void qsort(void *base, size_t nel, size_t width, int (*compar)(const void *, const
void *));
...
int main(void)
{
   int items[] = { 4, 3, 1, 2 };
   qsort(items, sizeof(items) / sizeof(items[0]), sizeof(items[0]), compareInts);
  return 0;
}
```

In C++, a function object may be used instead of an ordinary function by defining a class that overloads the function call operator by defining an operator() member function. In C++, this is called a *class type functor*, and may appear as follows:

```
// comparator predicate: returns true if a < b, false otherwise
struct IntComparator
{
 bool operator()(const int &a, const int &b) const
 {
  return a < b;
 }
};
...
// An overload of std::sort is:
template <class RandomIt, class Compare>
void sort(RandomIt first, RandomIt last, Compare comp);
...
int main()
{
  std::vector<int> items { 4, 3, 1, 2 };
  std::sort(items.begin(), items.end(), IntComparator());
  return 0;
}
```

Q.6 (a) What is containership? How it is different from inheritance?

Ans. Containership: *Containership is the phenomenon of using one or more classes within the definition of other class. When a class contains the definition of some other classes, it is referred to as composition, containment or aggregation. The data member of a new class is an object of some other class. Thus the other class is said to be composed of other classes and hence referred to as containership. Composition is often referred to as a "has-a" relationship because the objects of the composite class have objects of the composed class as members.*

Inheritance: Inheritance is the phenomenon of deriving a new class from an old one. Inheritance supports code reusability. Additional features can be added to a class by deriving a class from it and then by adding new features to it. Class once written or tested need not be rewritten or redefined. Inheritance is also referred to as specialization or derivation, as one class is inherited or derived from the other. It is also termed as "is-a" relationship because every object of the class being defined is also an object of the inherited class.

Q.6 (b)Explain the following :

(i)Pure Virtual Function.

(ii) Operator overloading.

A pure virtual function is a function that has ***the notation "= 0"*** in the declaration of that function. Why we would want a pure virtual function and what a pure virtual function looks like is explored in more detail below.

Simple Example of a pure virtual function in C++

```
class SomeClass {
public:
  virtual void pure_virtual() = 0; // a pure virtual function
  // note that there is no function body
};
```


Operator Overloading: Overloaded operators are functions with special names the keyword operator followed by the symbol for the operator being defined. Like any other function, an overloaded operator has a return type and a parameter list.

Box operator+(const Box&);

declares the addition operator that can be used to **add** two Box objects and returns final Box object. Most overloaded operators may be defined as ordinary non-member functions or as class member functions. In case we define above function as non-member function of a class then we would have to pass two arguments for each operand as follows: Box operator+(const Box&, const Box&);

Following is the example to show the concept of operator over loading using a member function. Here an object is passed as an argument whose properties will be accessed using this object, the object which will call this operator can be accessed using **this** operator as explained below:

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

class Box
{
 public:

   double getVolume(void)
   {
    return length * breadth * height;
   }
   void setLength( double len )
   {
     length = len;
   }

   void setBreadth( double bre )
   {
     breadth = bre;
   }

   void setHeight( double hei )
   {
     height = hei;
   }
   // Overload + operator to add two Box objects.
   Box operator+(const Box& b)
   {
     Box box;
     box.length = this->length + b.length;
     box.breadth = this->breadth + b.breadth;
     box.height = this->height + b.height;
     return box;
   }
private:
```

```
    double length;    // Length of a box
    double breadth;   // Breadth of a box
    double height;    // Height of a box
};
// Main function for the program
int main( )
{
  Box Box1;          // Declare Box1 of type Box
  Box Box2;          // Declare Box2 of type Box
  Box Box3;          // Declare Box3 of type Box
  double volume = 0.0;   // Store the volume of a box here

  // box 1 specification
  Box1.setLength(6.0);
  Box1.setBreadth(7.0);
  Box1.setHeight(5.0);

  // box 2 specification
  Box2.setLength(12.0);
  Box2.setBreadth(13.0);
  Box2.setHeight(10.0);

 // volume of box 1
 volume = Box1.getVolume();
 cout << "Volume of Box1 : " << volume <<endl;

 // volume of box 2
 volume = Box2.getVolume();
 cout << "Volume of Box2 : " << volume <<endl;

 // Add two object as follows:
 Box3 = Box1 + Box2;

 // volume of box 3
 volume = Box3.getVolume();
 cout << "Volume of Box3 : " << volume <<endl;

 return 0;
}
```

Q.7(a) Explain the technique of late binding?

Ans. Late binding, or **dynamic binding**, is a computer programming mechanism in which the method being called upon an object is looked up by name at runtime.

In late binding the compiler does not have enough information to verify the method even exists. Instead the method is looked up of name at runtime.

The primary advantage of using late binding in Component Object Model (COM) programming is that it does not require the compiler to reference the libraries that contain the object at compile time. This makes the compilation process more resistant

to version conflicts, in which the class's v-table may be accidentally modified. (This is not a concern in *JIT*-compiled platforms such as .NET or Java, because the v-table is created at runtime by the virtual machine against the libraries as they are being loaded into the running application.

Q. Write a program in C++ to demonstrate the use of pure virtual functions?

AIM

Ans. A program to demonstrate how a pure virtual function is defined, declared and invoked from the object of derived class through the pointer of the base class.

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

            class  base
                    {
                     private: int x;
                        float y;
                     public : virtual void getdata( );
                        virtual void display( );
                    };
class dev : public base
                    {
                     private:  int roll;
                         char name[20];
                     public : void getdata( );
                         void  display( );
                    };
void base :: getdata( ) { }
void base :: display( ) { }

    void dev :: getdata( )
                    {
                     cout<<" Enter Roll of  the Student ";
                     cin>> roll;
                     cout<<" Enter name of  the student";
                     cin>>name;
}
    void dev :: display( )
                    {
                     cout<<"Name is :"<<name<<endl;
                     cout<<" Roll no is :"<<roll <<endl;
                    }

            void main( )
                    {
                     base * ptr;
                     dev obj;
                     clrscr( );
                     ptr = &obj;
```

```
        ptr -> getdata( );
        ptr -> display( );
        getch( );
       }
```

Q.8. (a) Explain the syntax of class and function templates?

Ans. Consider this function that swaps its two integer arguments:

```
void swap(int& x, int& y)
{
 int tmp = x;
 x = y;
 y = tmp;
}
```

If we also had to swap floats, longs, Strings, Sets, and FileSystems, we'd get pretty tired of coding lines that look almost identical except for the type. Mindless repetition is an ideal job for a computer, hence a function template:

```
template<typename T>
void swap(T& x, T& y)
{
 T tmp = x;
 x = y;
 y = tmp;
}
```

Every time we used swap() with a given pair of types, the compiler will go to the above definition and will create yet another "template function" as an instantiation of the above. E.g.,

```
int main()
{
 int    i,j; /*..:*/ swap(i,j); // Instantiates a swap for int
 float   a,b; /*...*/ swap(a,b); // Instantiates a swap for float
 char    c,d; /*...*/ swap(c,d); // Instantiates a swap for char
 std::string s,t; /*...*/ swap(s,t); // Instantiates a swap for std::string
 std ... String
}
```

Q.8 (b)Explain the use of file pointers?

Ans. File Pointers The C++ input and output system manages two integer values associates with a file.These are:

- get pointer – specifies the location in a file where the next read operation will occur.
- put pointer – specifies the location in a file where the next write operation will occur.

In other words, these pointers indicate the current positions for read and write operations, respectively. Each time an input or an output operation takes place, the pointers are automatically advances sequentially. The term pointers should not be confused with normal C++ pointers used as address variables.

Often you may want to start reading an existing file from the beginning and continue sequentially until the end. When writing, you ma want to start from the beginning, deleting any existing contents, or appending new records (in which case you can open the file with the ios:app mode specifier). These are default actions, so no manipulation of the pointers is necessary.

Sometimes you may have to manipulate file pointers to read from and write to a particular location in a file. The seekg() and tellg() functions allow you to set and examine the get pointer, and the seekp() and tellp() functions perform these same actions on the put pointer. In other words, these four functions allow you to access the file in a non sequential or random mode.

Q.8. (c) Write a program in C++ in which an integer array can be entered into a file using write() function and access the numbers using read() function.

Ans.

```
#include <iostream>
#include <iomanip>
#include <string>
using namespace std;
const double REG_SERV_CHARGES = 10.00;
const int REG_FREE_MINUTES = 50;
const double REG_RATE_OVER_50 = 0.20;
const double PREM_SERV_CHARGES = 25.00;
const int PREM_FREE_DAY_MINUTES = 75;
const double PREM_DAY_RATE_OVER_75 = 0.10;
const int PREM_FREE_NIGHT_MINUTES = 100;
const double PREM_NIGHT_RATE_OVER_100 = 0.05;
int  getMinutes (string promptMessage);
void regularAmtDue (int minUsed, double &amtDue);
void premiumAmtDue (int dayMin, int nightMin, double &amtDue);
void writeOutput (int minutes1, int minutes2, double amtDue, char service);
int main()
{
  char serviceType;
char serviceType;
 int minutesUsed;
 int minutesUsedPN;
double amountDue;
cout << fixed << showpoint;
cout << setprecision(2);
cout << "Enter service type: (r or R) for regular,"
   << "(p or P) for premium service:"  << endl;
cin >> serviceType;
cout << endl;
switch (serviceType)
{
                    case 'r':
case 'R':
                        // call getMinutes for minutes of service here
                        // call regularAmtDue here
                            // call writeOutput here
    break;
case 'p':
case 'P':
                        // call getMinutes for day minutes here
                        // call getMinutes for night minutes here
```

```
                        // call premiumAmtDue here
                        // call writeOutput here
    break;
  default:
    cout << "Invalid Service Type" << endl;
                }//end switch
  system("PAUSE");
                    return 0;
} // end main
int getMinutes (string promptMessage)
{
//-------------------------------------------------------------------
// getMinutes is a value returning function that accepts a string and
// prompts the user for the number of minutes then returns this value
// to main.
//-------------------------------------------------------------------
} // end getMinutes
void regularAmtDue (int minUsed, double &amtDue)
{
   if (minUsed <= 50)
    {
     amtDue = REG_SERV_CHARGES;
    }
    else if (minUsed > 50)
    {
     amtDue = REG_SERV_CHARGES + minUsed * REG_RATE_OVER_50;
    }
} // end regularAmtDue
void premiumAmtDue (int dayMin, int nightMin, double &amtDue)
{
if (dayMin <= 75)
    {
     amtDue = PREM_SERV_CHARGES;
    }
else if (nightMin > 100)
    {
                amtDue = PREM_SERV_CHARGES + (nightMin *
PREM_NIGHT_RATE_OVER_100);
    }
    else if(nightMin > 100 && dayMin > 75)
    {
    amtDue = PREM_SERV_CHARGES + (dayMin * PREM_DAY_RATE_OVER_75)
            + (nightMin * PREM_NIGHT_RATE_OVER_100);
    }
    else
    {
```



```
    amtDue = PREM_SERV_CHARGES + (dayMin * PREM_DAY_RATE_OVER_75);
    }
} // end premiumAmtDue
void writeOutput (int minutes1, int minutes2, double amtDue, char service)
{
  //-------------------------------------------------------------------
// writeOutput is a void function
// All parameters passed by value.
// This function determines which service type is used and based on that decision
// writes the appropriate set of output statements.
// When calling this function any int parameter not used can be passed 0 in that
// position.
//-------------------------------------------------------------------
} // end writeOutput
```

Q.9 (*a*) Write a program to demonstrate the catch of all the exceptions?

Ans. There is a special catch block called 'catch all' catch(..) that can be used to catch all types of exceptions. For example, in the following program, an int is thrown as an exception, but there is no catch block for int, so catch(..) block will be executed.

```
#include <iostream>
using namespace std;
int main()
{
  try {
    throw 10;
  }
  catch (char *excp) {
     cout << "Caught" << excp;
  }
  catch (...) {
     cout << "Default Exception\n";
  }
  return 0;
}
```

Q.9. (*b*)Briefly Explain the followings:

(*i*) Standard Template library

(*ii*)Vectors

Ans. 9. (*b*) (*i*) The Standard Template Library (STL) is a software library for the C++ programming language that influenced many parts of the C++ Standard Library. It provides four components called *algorithms*, *containers*, *functional*, and *iterators*.[1]

The STL provides a ready-made set of common classes for C++, such as containers and associative arrays, that can be used with any built-in type and with any user-defined type that supports some elementary operations (such as copying and assignment). STL algorithms are independent of containers, which significantly reduces the complexity of the library.

The STL achieves its results through the use of templates. This approach provides compile-time polymorphism that is often more efficient than traditional run-time polymorphism. Modern C++ compilers are tuned to minimize any abstraction penalty arising from heavy use of the STL.

The STL was created as the first library of generic algorithms and data structures for C++, with four ideas in mind: generic programming, abstractness without loss of efficiency, the Von Neumann computation model and value semantics.

Ans. 9. (b) (ii) Vectors are sequence containers represent[illegible]rrays that can change insize. Just like arrays, vectors use contiguous storage loca[illegible] for their elements, which means that their elements can also be accessed using offsets on regular pointers to its elements, and just as efficiently as in arrays. But unlike arrays, their size can change dynamically, with their storage being handled automatically by the container. Internally, vectors use a dynamically allocated array to store their elements. This array may need to be reallocated in order to grow in size when new elements are inserted, which implies allocating a new array and moving all elements to it.

The vectors represents large amount of space for the input data as compared to the arrays.

This is a relatively expensive task in terms of processing time, and thus, vectors do not reallocate each time an element is added to the container. Instead, vector containers may allocate some extra storage to accommodate for possible growth, and thus the container may have an actual capacity greater than the storage strictly needed to contain its elements (i.e., its size). Libraries can implement different strategies for growth to balance between memory usage and reallocations, but in any case, reallocations should only happen at logarithmically growing intervals of size so that the insertion of individual elements at the end of the vector can be provided with *amortized constant time* complexity (see push_back). Therefore, compared to arrays, vectors consume more memory in exchange for the ability to manage storage and grow dynamically in an efficient way.

Compared to the other dynamic sequence containers (deques, lists and forward_lists), vectors are very efficient accessing its elements (just like arrays) and relatively efficient adding or removing elements from its end. For operations that involve inserting or removing elements at positions other than the end, they perform worse than the others, and have less consistent iterators and references than lists and forward_lists.

IMPORTANT QUESTION ACCORDING TO NEW SYLLABUS OF UNIT IV

Q.1. What is the importance of iterator in C++?

Ans. Iterator: a pointer-like object that can be incremented with ++, dereferenced with *, and compared against another iterator with !=.

Iterators are generated by STL container member functions, such as begin() and end(). Some containers return iterators that support only the above operations, while others return iterators that can move forward and backward, be compared with <, and so on.

The generic algorithms use iterators just as you use pointers in C to get elements from and store elements to various containers. Passing and returning iterators makes the algorithms

- more generic, because the algorithms will work for any containers, including ones you invent, as long as you define iterators for them
- more efficient (as discussed here)

Q.2. Why we use allocator? Define the term allocator?

Ans. Allocator: Encapsulates a memory allocation and deallocation strategy.

Every standard library component that may need to allocate or release storage, from std::string, std::vector, and every container except std::array, to std::shared_ptr and std::function, does so through an **Allocator**: an object of a class type that satisfies the following requirements.

Some requirements are optional: the template std::allocator_traits supplies the default implementations for all optional requirements, and all standard library containers and other allocator-aware classes access the allocator through std::allocator_traits, not directly.

a. destroy(xptr):Destructs an object of type X pointed to by xptr, but does not deallocate any storage.

A a1(std::move(a))Constructs a1 such that it equals the prior value of a. Does not throw exceptions.

Q.3. What is vector? Explain it with example.

Ans. Vectors: One of the basic classes implemented by the Standard Template Library is the vector class. A vector is, essentially, a resizable array; the vector class allows random access via the [] operator, but adding an element anywhere but to the end of a vector causes some overhead as all of the elements are shuffled around to fit them correctly into memory. Fortunately, the memory requirements are equivalent to those of a normal array. The header file for the STL vector library is vector. (Note that when using C++, header files drop the .h; for C header files - e.g. stdlib.h - you should still include the .h.) Moreover, the vector class is part of the std namespace, so you must either prefix all references to the vector template with std:: or include "using namespace std;" at the top of your program.

"Slicing" is where you assign an object of a derived class to an instance of a base class, thereby losing part of the information - some of it is "sliced" away.

```
class A {
 int foo;
};
```

```
class B : public A {
int bar;
};
```

So an object of type B has two data members, foo and bar

Then if you were to write this:

```
B b;
A a = b;
```

Then the information in b about member bar is lost in a.

Q.4. What does the header numeric signifies.

Ans. <numeric>

Generalized numeric operations

This header describes a set of algorithms to perform certain operations on sequences of numeric values.

Due to their flexibility, they can also be adapted for other kinds of sequences.

Q.5. What is the major difference between a sequence container and an associative container?

Ans. Maps are a kind of associative containers that stores elements formed by the combination of a key value and a mapped value.Lists are a kind of sequence containers. As such, their elements are ordered following a linear sequence.Sets are a kind of associative containers that stores unique elements, and in which the elements themselves are the keys. Arrays are also a kind of sequence containers, just much lower level than list. Their size is fixed, they don't manage memory dynamically and aren't generic. They are inherited from C.

FIRST TERM EXAMINATION [APRIL-2015]
FOURTH SEMESTER [B. TECH]
OBJECT ORIENTED PROGRAMMING [ETCS-210]

Time: 1 Hr. MM : 30

Note: *Question No.1 is compulsory. Attempt any two more questions from the rest.*

Q.1. (*a*) State the characteristics of object oriented programming. [2.5 × 4 = 0]

Ans. Class definitions: Basic building blocks OOP and a single entity which has data and operations on data together.

Objects: The instances of a class which are used in real functionality-its variables and operations.

Abstraction: Specifying what to do but not how to do, a flexible feature for having a overall view of an object's functionality.

Encapsulation: Binding data and operations of data together in a single unit.

Inheritance and class hierarchy: Reusability and extension of existing classes.

Polymorphism: Multiple definitions for a single name-functions with same name with different functionality.

Message passing: Objects communicates through invoking methods and sending data to them. This feature of sending and receiving information among objects through function parameters is known as Message Passing.

Q.1. (*b*) Difference Between Class and Structure.

Ans. Refer Q.2. (*b*) of End Term Examination 2014.

Q.1. (*c*) Difference between Pointer and Reference.

Ans: A pointer is a variable which stores the address of another variable.

A reference is a variable which refers to another variable.

For example:-

```
int i = 3;
int *ptr = &i;
int &ref = i;
```

The first line simply defines a variable. The second defines a pointer to that variable's memory address. The third defines a reference to the first variable.

Not only are the operators different, but you use the differently as well. With pointers must use the * operator to dereference it. With a reference no operator is required. It is understood that you are intending to work with the referred variable.

The following two lines will both change the value of i to 13.

```
*ptr = 13;
ref = 13;
```

Q.1. (*d*) Explain new and delete.

Ans. New and Delete operators are provided by C++ for runtime memory management. They are used for dynamic allocation and freeing of memory while a program is running.

The new operator allocates memory and returns a pointer to the start of it. The delete operator frees memory previously allocated using new.

Syntax of new is:

p_var = new type name: Where p_var is a previously declared pointer of type typename. typename can be any basic data type.

New can also create an array:

p_var = new type [size]: In this case, size specifies the length of one-dimensional array to create.

Example 1:

```
int *p; p=new int;
```

It allocates memory space for an integer variable. And allocated memory can be released using following statement,

```
delete p;
```

Example 2:

```
int *a; a = new int[100];
```

It creates a memory space for an array of 100 integers. And to release the memory following statement can be used,

```
delete []a;
```

Q.2. (a) What are Friend Function? Explain with example. **[5 × 2 = 10]**

Ans. Friend Functions:

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

A friend can be a function, function template, or member function, or a class or class template, in which case the entire class and all of its members are friends.

```
#include <iostream>
using namespace std;
class Rectangle {
                int width, height;
public:
Rectangle (int w = 1, int h = 1): width(w),height(h){}
    friend void display(Rectangle &);
};
void display(Rectangle &r) {
     cout << r.width * r.height << endl;
}
int main () {
            Rectangle rect(5,10);
            display(rect);
   return 0;
}
```

We make a function a friend to a class by declaring a prototype of this external function within the class, and preceding it with the keyword **friend.**

```
friend void display (Rectangle &);
```

The friend function **display(rect)** has an access to the private member of the Rectangle class object though it's not a member function. It gets the width and height using dot: **r.width** and **r.height**. If we do this inside main, we get an error because they

are private members and we can't access them outside of the class. But friend function to the class can access the private members.

Q.2. (b) What are Inline Functions? Explain with example.

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

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

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

Example : Use of inline function to return max of two numbers:

```
#include <iostream>
inline int Max(int x, int y)
{
  return (x > y)? x : y;
}
int main( )
{
  cout << "Max (20,10): " << Max(20,10) << endl;
  cout << "Max (0,200): " << Max(0,200) << endl;
  cout << "Max (100,1010): " << Max(100,1010) << endl;
  return 0;
}
```

Q.3. (a) What are copy constructors? Explain with example. **[5 × 2 = 10]**

Ans. The copy constructor is a special kind of constructor which creates a new object which is a copy of an existing one, and does it efficiently.The copy constructor receives an object of its own class as an argument, and allows to create a new object which is copy of another without building it. The copy constructor receives an object of its own class as an argument, and allows to create a new object which is copy of another without building it from scratch. A copy constructor has the following general function prototype:

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

Example of copy constructor.

```
#include<iostream>
class Point
{
private:
   int x, y;
public:
   Point(int x1, int y1) {x = x1; y = y1;}
// Copy constructor
Point(const Point and p2) {x = p2.x; y = p2.y;}
int getX()      { return x; }
int getY()      {return y; }
};
```

```
int main()
{
Point p1(10, 15); // Normal constructor is called here
Point p2 = p1; // Copy constructor is called here
cout << "p1.x = " << p1. getX() << ", p1.y = " << p1.getY();
cout << "\np2.x = " << p2. getX() << ", p2.y = " << p2.getY();
return 0;
}
```

Q. 3. (*b*) What are static data member? Explain with example.

Ans. A data member of a class can be qualified as static.The properties of a static member variable are similar to that of a *C* static variable. A static member variable has certain special characteristics.

1. It is initialized to zero when the first object of its class is created.

2. Only one copy of that member is created for the entire class and is shared by all the objects of that class, no matter how many objects are created.

3. It is visible by within the class, but its lifetime is the entire program.

Example:

```
#include<iostream.h>
using namespace std;
class item
{
static int count;
int num;
public:
void getdat ::nt a)
{
number=a; count++;
}
void getcount()
{
Cout<<count<<"\n";
}
};
int item : : count;
int main()
{
item a,b,c;
a.getcount();
b.getcount();
c.getcount();
a.getdata(100);
b.getdata(200);
c.getdata(300);
```



```
a.getcount();
b.getcount();
c.getcount();
return 0;
}
```

Output is : 0,0,0 and 3,3,3

Q. 4. (*a*) Difference between C and C++. **[5 × 2 = 10]**

Ans.

C	C++
1. C is Procedural Language.	1. C++ is non Procedural i.e Object oriented Language.
2. No virtual Functions are present in C	2. The concept of virtual Functions are used in C++.
3. In C, Polymorphism is not possible.	3. The concept of polymorphism is used in C++.Polymorphism is the most Important Feature of OOPS.
4. Operator overloading is not possible . in C	4. Operator overloading is one of the greatest Feature of C++.
5. Top down approach is used in Program Design.	5. Bottom up approach adopted in Program Design.
6. No namespace Feature is present in C Language.	6. Namespace Feature is present in C++ for avoiding Name collision.
7. Multiple Declaration of global variables are allowed.	7. Multiple Declaration of global variables are not allowed.
8. In C • scan f () Function used for Input. • printf () Function used for output.	8. In C++. • Cin>> Function used for Input. • Cout<< Function used for output.
9. Mapping between Data and Function is difficult and complicated.	9. Mapping between Data and Function can be used using "Objects"
10. In C, we can call main() Function through other functions	10. In C++, we cannot call main() Function through other functions.
11. C requires all the variables to be defined at the starting of a scope.	11. C++ allows the declaration of variable anywhere in the scope *i.e.* at time of its First use.
12. No inheritance is possible in C.	12. Inheritance is possible in C++
13. In C, malloc() and calloc() Functions are used for Memory Allocation and free() function for Memory Deallocating.	13. In C++, new and delete operators are used for Memory Allocating and Deallocating.
14. It supports built-in and primitive data types.	14. It support both built-in and user define data types.
15. In C, Exception handling is not present.	15. In C++, exception handling is done with Try and Catch block.

Q.4. (*b*) Write a program using class and objects to add two complex numbers to show the concepts of returning objects.

Ans.

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

```
class complex
{
float x;
float y;
public:
void input(float real, float imag)
{
X=real; y=imag; }
friend complex sum(complex, complex);
void show (complex);
};
complex sum(complex c1, complex c2)
{
complex c3;
c3.x=c1.x+c2.x;
c3.y=c1.y+c2.y;
return (c3);
}
void complex : : show(complex c)
{
cout<<c.x<<"+ j"<<c.y<<"\n";
}
int main()
{
complex a,b,c;
a.input(3.1, 5.65);
b.input(2.75 , 2);
c=sum(a, b);
cout<<a.show(a);
cout<<b.show(b);
cout<<c.show(c);
return 0;
}
```

SECOND TERM EXAMINATION [APRIL-2015]
FOURTH SEMESTER [B. TECH]
OBJECT ORIENTED PROGRAMMING [ETCS-210]

Time: 1 Hr **MM : 30**

Note: *Question No.1 is compulsory. Attempt any two more questions from the rest.*

Q.1. (a) Explain templates? **(2.5)**

Ans. Template is one of the feature added to C+ recently. A template can be used to create a family of classes or functions. For example, a class template for an array class would enable us to create arrays of various data types such as int array and float array etc. Similarly we can define a template for a function say mul(), that would help us create various versions of mul() for multiplying int, float and double type values.

Syntax for class template :

```
Template <class T>
Class  classname
{
};
```

Syntax for function template :

```
Template <class T>
Returntype functionname (arguments of type T)
{
}
```

Q.1. (b) Difference between Function Overloading and Function Overriding. **(2.5)**

Ans. Overloading is defining functions that have similar signatures, yet have different parameters.

Overriding is only pertinent to derived classes, where the parent class has defined a method and the derived class wishes to override that function.

Overriding	Overloading
Functions name and signatures must be same.	Having same Function name with different **Signatures.**
Overriding is the concept of **runtime polymorphism**	Overloading is the concept of **compile time polymorphism.**
When a function of base class is re-defined in the derived class called as Overriding	Two functions having same name and return type, but with different type and/or number of arguments is called as Overloading.
It needs inheritance.	It doesn't need inheritance.
Functions should have same data type.	Functions can have different data types
Function should be public.	Function can be different access specifies

Q1. (c) Difference between Early and Later binding? **(2.5)**

Ans. Early Binding: Events occurring at compile time are known as early binding. In the process of early binding all information which is required for a function call is known at compile time. Early binding is a fast and efficient process. Examples of early binding: function calls, overloaded function calls, and overloaded operators.

Late Binding: In this process all information which is required for a function call is not known at compile time. Hence, objects and functions are not linked at run time. Late Binding is a slow process. However, it provides flexibility to the code. Example of Late Binding: Virtual functions.

Q1. (d) Explain Exception handling? **(2.5)**

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

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

- **Throw:** A program throws an exception when a problem shows up. This is done using a **throw** keyword.
- **Catch:** A program catches an exception with an exception handler at the place in a program where you want to handle the problem. The **catch** keyword indicates the catching of an exception.
- **Try:** A **try** block identifies a block of code for which particular exceptions will be activated. It's followed by one or more catch blocks.

Q.2.(a) Explain the concept of Class to Basic Type Conversion with example? **(5)**

Ans. Class to Basic Type Conversions: The constructor handles the task of converting basic types to class types very well. But you cannot use constructors for converting class types to basic data types. Instead, you can define an overloaded *casting operator* that can be used to convert a class data type into a basic data type. The general form of an *overloaded casting operator* function is shown below. This function is also known as a *conversion function*.

```
Operator typename()
{
----------------------
Function body
----------------------
}
```

The above function converts the class type into the *typename* mentioned in the function. For example, **operator float( )** converts a class type to **float, operator int()** converts a class type to int, and so on.

Consider the following conversion function:

```
Vector :: operator double()
{
double sum=0;
for(int i=0;i<size;i++)
sum=sum + v[i] * u[i];
return sqrt(sum);
}
```


The above function converts a vector object into its corresponding scalar magnitude, in other words, it converts a vector type into a double.

Q.2. (b) Explain operator overloading? Write a program to overload binanry '+' operator. **(5)**

Ans. C++ allows you to specify more than one definition for a **function** name or an **operator** in the same scope, which is called **function overloading** and **operator overloading** respectively.

You can redefine or overload most of the built-in operators available in C++. Overloaded operators are functions with special names the keyword operator followed by the symbol for the operator being defined. Like any other function, an overloaded operator has a return type and a parameter list.

Program : Overload the binary + operator.

```
#include<iostream>
using namespace std;
class complex{
  float x1,y1;
public:
        void disp(){cout<<x1<<"+"<<y1<<"i"<<"\n";//to represent in complex form }
        complex(float a,float b)//constructor
        { x1=a; y1=b; }
        complex operator+(complex cc)
{ return complex(x1+cc.x1,y1+cc.y1);//operator overloading defined here }
};
int main(){
        complex c1(2.5,-0.3);//invokes constructor
        complex c2(4.5,7);
        complex c3=c1+c2;//invokes operator +
        c3.disp();// displays final answer
        return 0;
}
```

Q.3. (a) Explain Inheritance and type of Inheritance? **(5)**

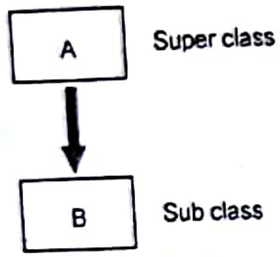
Ans. One of the most important concepts in object-oriented programming is that of inheritance. Inheritance allows us to define a class in terms of another class, which makes it easier to create and maintain an application. This also provides an opportunity to reuse the code functionality and fast implementation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

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

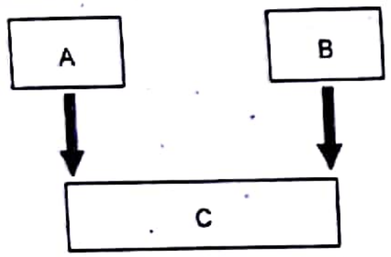
In C++, we have 5 different types of Inheritance. Namely,

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

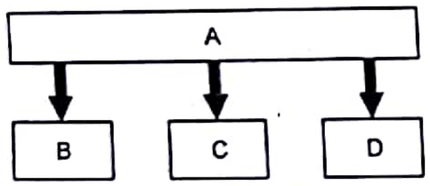
Single Inheritance: In this type of inheritance one derived class inherits from only one base class. It is the most simplest form of Inheritance.



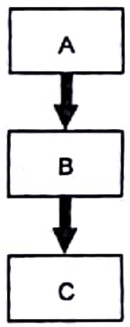
Multiple Inheritance: In this type of inheritance a single derived class may inherit from two or more than two base classes.



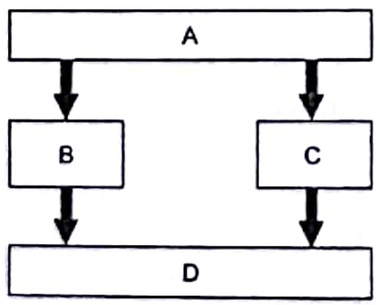
Hierarchical Inheritance: In this type of inheritance, multiple derived classes inherits from a single base class.



Multilevel Inheritance: In this type of inheritance the derived class inherits from a class, which in turn inherits from some other class. The Super class for one, is sub class for the other.



Hybrid Inheritance: Hybrid Inheritance is combination of Hierarchical and Mutilevel Inheritance.



Q.3. (b) Design 3 classes Student, Exam and Result. Such that Exam is inherited from Student class, Result is inherited from Exam class. Write a program to model this relationship. What type of inheritance does this model belong to. (5)

Ans.

```
#include<iostream.h>
#include<conio.h>
#include<string.h>
class student//base class
{
private:
 int 1;
 char nm[20];
public:
  void read();
  void display();
};
class exam: public student//dervivec from student
{
protected:
  int s1;
  int s2;
  int s3;
public:
  void getmarks();
  void putmarks();
};
class result : public exam //derived  from marks
{
private:
   int t;
   float p;
   public:
   void process();
   void printresult();
};
void student::read()
 {
cout<<"enter Roll no and Name "<<endl;
in>>1>>nm;
 }
 void student:: display()
 {
 cout <<"Roll NO:"<<1<<endl;
```

```
cout<<"name : "<<nm<<endl;
}
void exam ::getmarks()
{
cout<<"enter three subject marks  "<<endl;
cin>>s1>>s2>>s3;
}
void exam ::putmarks()
{
cout <<"subject 1:"<<s1<<endl;
cout<<" subject 2 : "<<s2<<endl;
cout <<"subject 3:"<<s3<<endl;
}
void result::process()
{
   t= s1+s2+s3;
   p = t/3.0;
}
void result::printresult()
{
cout<<"total = "<<t<<endl;
cout<<"per  = "<<p<<endl;
}
void main()
{
result x;
clrscr();
x.read();
x.getmarks();
x.process();
x.display();
x.putmarks();
x.printresult();
   getch();
}
```

Q.4. (a) When do we make a virtual function "pure"? What are the implications of making a function a pure virtual function? Explain with example. **(5)**

Ans. When should pure virtual functions be used in:

In C++, a regular, "non-pure" virtual function provides a definition, which means that the class in which that virtual function is defined does not need to be declared abstract. You would want to create a pure virtual function when it doesn't make sense to provide a definition for a virtual function in the base class itself, within the context of inheritance.

An example of when pure virtual functions are necessary: For example, let's say that you have a base class called Figure. The Figure class has a function called **draw**. And, other classes like Circle and Square derive from the Figure class. In the Figure class, it doesn't make sense to actually provide a definition for the draw function, because of the simple and obvious fact that a "Figure" has no specific shape. It is simply meant to act as a base class. Of course, in the Circle and Square classes it would be obvious what should happen in the draw function they should just draw out either a Circle or Square (respectively) on the page. But, in the Figure class it makes no sense to provide a definition for the draw function. And this is exactly when a pure virtual function should be used – the draw function in the Figure class should be a pure virtual function.

Q. 4. (b) Write a function template for finding the minimum value contained in an array. **(5)**

Ans.

```
#include<iostream.h>
template<class t>
void minarr(t a[])
{
t i,min;
min=a[0];
for(i=0;i<5;i++)
{
if(a[i]<min)
min=a[i];
}
cout<<min;
}
void main()
{
int a[5]={10,20,30,40,50};
char b[5]={'a','b','c','d','e'};
minarr(a);
minarr(b);
}
```

END TERM EXAMINATION [JUNE-2015]
FOURTH SEMESTER [B. TECH]
OBJECT ORIENTED PROGRAMMING [ETCS-210]

Time: 3 Hrs. **MM : 75**

Note: *Attempt any five questions including Q.No. 1 which is compulsory. Select one question from each unit.*

Q.1. Attempt the following questions: **(10 × 2.5 = 25)**

(a) Illustrate the use of delete operator in C++.

Ans. The delete Operator. Once the memory is allocated using new operator, it should released to the operating system. If the program uses large amount of memory using new, system may crash because there will be no memory available for operating system. The following expression returns memory to the operating system.

```
delete [] ptr;
```

The brackets [] indicates that, array is deleted. If you need to delete a single object then, you don't need to use brackets.

```
delete ptr;
```

In most cases, memory allocated dynamically is only needed during specific periods of time within a program; once it is no longer needed, it can be freed so that the memory becomes available again for other requests of dynamic memory. This is the purpose of operator delete.

Q.1. (b) Can you say that destructor in C++ may be overloaded? Justify your answer.

Ans. A destructor can never be overloaded. An overloaded destructor would mean that the destructor has taken arguments. Since a destructor does not take arguments, it can never be overloaded. An object that is going to be destroyed needs only one way of cleaning itself up.

Destructor has no arguments and no return type to enforce that once it is called, the object is no longer useable.

Q.1. (c) Abstraction and Encapsulation may be used interchangeable in the context of object orientation/justify your answer.

Ans.

Abstraction	Encapsulation
1. Abstraction solves the problem in the design level.	1. Encapsulation solves the problem in the implementation level.
2. Abstraction is used for hiding the unwanted data and giving relevant data.	2. Encapsulation means hiding the code and data into a single unit to protect the data from outside world.
3. Abstraction lets you focus on what the object does instead of how it does it	3. Encapsulation means hiding the details internal or mechanics of how an object does something.
4. **Abstraction**: Outer layout, used in terms of design. **For Example:** Outer Look of a Mobile, Phonelike it has a display screen and keypad buttons to dial a number.	4. **Encapsulation**: Inner layout, used in terms of implementation. **For Example:** Inner Implementation detail of a Mobile Phone, how keypad button and display screen are connect with each other using circuits.

Q. 1. (d) Differentiate between 'array pointer' and 'pointer array' by writing single line C++ statement in each case.

Array of pointers : int *array of pointers[4]; We know that pointer holds a address of int data type variable. So the above mentioned array_of_pointers can store four different or same int data type address *i.e.* array_of_pointers[0], array_of_pointers [1], array_of_pointers[2] array_of_pointers[3] each can hold a int data type address.

Pointer to array :

In simple language the pointer_to_array is a single variable that holds the address of the array.

Example:

```
int simple_array[4];
Int *p;
Now we can assign
 p = simple_array;
```

Q.1. (e) What is the significance of enum data type in C++? Exlain with illustration.

Ans. An enumeration is a user-defined data type consists of integral constants and each integral constant is give a name. Keyword enum is used to defined enumerated data type.

```
enum type_name{ value1, value2,...,valueN };
```

Here, *type_name* is the name of enumerated data type or tag. And *value1*, *value2*,....,*valueN* are values of type *type_name*.

By default, *value*1 will be equal to 0, *value*2 will be 1 and so on but, the programmer can change the default value.

Changing the default value of enum elements

```
enum suit{
   club=0;
   diamonds=10;
   hearts=20;
   spades=3;
};
```

Q.1. (f) How static data members are different from non static data members?

Ans : Static data mamber: A data member of a class can be qualified as static. The properties of a static member variable are similar to that of a static variable. A static member variable has certain special characeristics.

1. It is initialized to 0 (zero) when the first object of this class is created. No other initialization is permitted.

2. Only one copy of that member is created for the entire class and is shared by all the objects of that class, No matter how many objects are created.

3. It is visible only within the class but its lifetime is in the entire program.

The type and scope of each static member variable must be defined outside the class definition.This is necessary because the ststic data member are stored separately rather than as a part of an object. Since they are associated with the class itself rather than with any object. They are also known as class variable.

Data member: Data memeber is the simple variable of a class which is initialized by member function of same class. By default data member is private and external functions can not access it.

Q.1. (g) What do you mean be compile time polymmorhism.?

Ans. Polymorphism is the ability of an object or reference to take many different forms at different instances. These are of two types one is the "compile time polymorphism" and other one is the "run-time polymorphism".

Compile time polymorphism: In this method object is bound to the function call at the compile time itself or we can say that events occurring at compile time are known as compile time polymorphism or early binding or static binding. In the process of early binding all info which is required for a function call is known at compile time. Early binding is a fast and efficient process. Examples of early binding: function calls, overloaded function calls, and overloaded operators.

Q.1. (h) Interpret the C++ statement friend return type op++ (x, y);

Ans. Friend return-type op++(x, y)

This statement belongs to the declaration of the friend function. In this statement the unary operator++ is overloaded an arguments x and y with the help of friend function. To overload the unary operators with the help of friend function, minimum one argument is required and to overload the binary operators with the help of friend function, minimum two arguments are required.

The keyword friend procedes function prototype declaration. It must be written inside the class. This function can be defined inside or outside the class. The arguments used in friend function are generally objects of the friend class. Friend function can access private member of the class through the objects.

Q.1. (i) Define the term 'type casting'.

"Type Casting" is method using which a variable of one datatype is converted to another datatype, to do it simple the datatype is specified using parenthesis in front of the value.

Example:

```
#include <iostream.h> using namespace std; int main()
{
short x=40;
int y;
y = (int)x;
cout << "Value of y is:: "<< y <<"\nSize is::"<< sizeof(y);
return 0;
}
```

Result:

Value of y is:: 40

Size is::4

In the above example the short data type value of x is type cast to an integer data type, which occupies "4" bytes.

Type casting can also done using some typecast operators available in C++. Following are the typecast operators used in C++.

- static_cast

- const_cast
- dynamic_cast
- reinterpret_cast
- typeid

Q.1. (j) What is the purpose of virtual function?

Ans. Virtual function is a function or method whose behavior can be overridden within an inheriting class by a function with the same signature. In other words, the purpose of virtual functions is to allow customization of derived class implementations.

If there are member functions with same name in base class and derived class, virtual functions gives programmer capability to call member function of different class by a same function call depending upon different context. This feature in C++ programming is known as polymorphism which is one of the important feature of OOP.

If a base class and derived class has same function and if you write code to access that function using pointer of base class then, the function in the base class is executed even if, the object of derived class is referenced with that pointer variable.

UNIT-I

Q.2. Write a program to read records of students such as student name, rollno, total marks through a reading function that expect an array o strucute as input from the main function and also display the name of those students who got more than 60% marks through display function. (12.5)

Ans.

```
#include<iostream.h>
#include<conio.h>
struct stud
{
int rno;
char name[20];
int total;
}s[5];
void getdata(struct stud st[])
{
int i;
cout<<"Enter the Roll no, Name and Total marks of 5 students :-"<<"\n";
for(i=0;i<5;i++)
cin>>st[i].rno>>st[i].name>>st[i].total;
}
void main()
{
int i;
clrscr();
getdata(s);
for(i=0,i<5;i++)
{
```

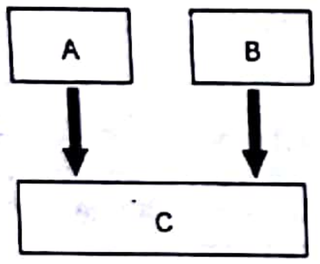
```
if(s[i].total>60)
cout<<s[i].rno<<"\t"<<s[i].name<<"\t"<<s[i].total<<"\n";
}
getch();
}
```

Q.3. Illustrate the difference between multiple inheritance, multilevel inheritance and hybrid inheritance through small working programs. **(12.5)**

Ans. One of the most important concepts in object-oriented programming is that of inheritance. Inheritance allows us to define a class in terms of another class, which makes it easier to create and maintain an application.

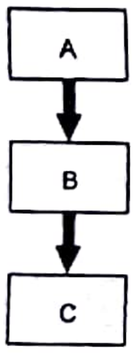
Multiple Inheritance

In this type of inheritance a single derived class may inherit from two or more than two base classes.



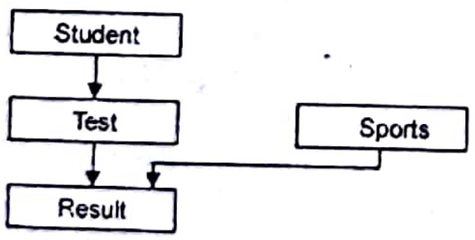
Multilevel Inheritance

In this type of inheritance the derived class inherits from a class, which in turn inherits from some other class. The Super class for one, is sub class for the other.



Hybrid Inheritance

Hybrid Inheritance is combination of two or more Inheritance such as multilevel and multiple.



Program to implement the hybrid inheritance :

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



```
class stu
{
protected:
        int rno;
public:
  void get_no(int a) { rno=a; }
void put_no(void) { cout<<"Roll no"<<rno<<"\n"; }
};
class test:public stu
{
      protected:
      float part1,part2;
public:
    void get_mark(float x,float y)
      { part1=x; part2=y;  }
void put_marks()
{
 cout<<"Marks obtained:"<<"part1="<<part1<<"\n"<<"part2="<<part2<<"\n";
}
};
class sports
{
    protected:
    float score;
public:
 void getscore(float s)  { score=s; }
void putscore(void)
{
   cout<<"sports:"<<score<<"\n";
}
};
class result: public test, public sports
{
   float total;
public:
   void display(void);
};
void result::display(void)
{
    total=part1+part2+score;
    put_no();
    put_marks();
```

```
  putscore(),
  cout<<"Total Score="<<total<<"\n";
}
int main()
{
  clrscr();
  result stu;
  stu.get_no(123);
  stu.get_mark(27.5,33.0);
  stu.getscore(6.0);
  stu.display();
  return 0;
}
```

UNIT-II

Q.4. What is the use of various kinds of constructors and destructors in C++? Explain with illustrations in each case **(12.5)**

Ans. Constructor: It is a member function having same name as it's class and which is used to initialize the objects of that class type with a legel initial value. Constructor is automatically called when object is created.

Types of Constructor

Default Constructor: A constructor that accepts no parameters is known as default constructor. If no constructor is defined then the compiler supplies a default constructor.

```
Circle :: Circle()
{
   radius = 0;
}
```

Parameterized Constructor: A constructor that receives arguments/parameters, is called parameterized constructor.

```
Circle :: Circle(double r)
{
   radius = r;
}
```

Copy Constructor : A constructor that initializes an object using values of another object passed to it as parameter, is called copy constructor. It creates the copy of the passed object.

```
Circle :: Circle(Circle &t)
{
   radius = t.radius;
}
```

There can be multiple constructors of the same class, provided they have different signatures.

Destructor: A destructor is a member function having same name as that of its class preceded by ~(tilde) sign and which is used to destroy the objects that have been created by a constructor. It gets invoked when an object's scope is over.

```
~Circle() {}
```


Example: In the following program constructors, destructor and other member functions are defined inside class definitions. Since we are using multiple constructor in class so this example also illustrates the concept of constructor overloading

```
#include<iostream>
using namespace std;
class Circle //specify a class
{
private :
double radius; //class data members
public:
  Circle() //default constructor
  {
      radius = 0;
  }
  Circle(double r) //parameterized constructor
  {
      radius = r;
  }
  Circle(Circle &t) //copy constructor
  {
    radius = t.radius;
  }
}
void setRadius(double r) //function to set data
{
   radius = r;
}
double getArea()
{
     return 3.14 * radius * radius;
}
~Circle() //destructor
   {}
};
int main()
{
  Circle c1; //defalut constructor invoked
  Circle c2(2.5); //parmeterized constructor invoked
  Circle c3(c2); //copy constructor invoked
  cout << c1.getArea()<<endl;
   cout << c2.getArea()<<endl;
   cout << c3.getArea()<<endl;
   return 0;
}
```

Q.5. Design a class to represent a bank account having data members as customer name, account number, account type, balance amount, etc. and methods as to assign initial values, to deposit an amount, to withdraw an amount and to display the name of the of those customers who have balance more than Rs. 50000. **(12.5)**

Ans.

```
#include<iostream.h>
#include<conio.h>
class bank
{
char name[20];
char type[10];
long int amount;
public:
long int acn;
void init();
void deposit();
void withdraw();
void display();
};
void bank :: init()
{
cout<<"Enter the Account No., Name of Customer, A/c Type and the Balance Amount
:"<<"\n";
cin>>acn>>name>>type>>amount;
}
void bank :: deposit()
{
int amt;
cout<<"Enter amount which you want to deposit:"<<"\n";
cin>>amt;
amount=amount+amt;
}
void bank :: withdraw()
{
int amt;
cout<<"Enter amount which you want to withdraw:"<<"\n";
cin>>amt;
amount=amount-amt;
}
void bank :: display()
{
if(amount>50000)
cout<<acn<<"\t"<<name<<"\t"<<type<<"\t"<<amount<<"\n";
```



```
}
void main()
{
clrscr();
bank obj[5];
int i,ch;
char choice;
long int acno;
cout<<"Enter details for 5 customers:"<<"\n";
for(i=0;i<5;i++)
obj[i].init();
do
{
cout<<"(1)Deposit an Amount"<<"\n";
cout<<"(2)Withdraw an Amount"<<"\n";
cout<<"(3)Display the Record"<<"\n";
cout<<"What do you want to do:-"<<"\n";
cin>>ch;
switch(ch)
{
case 1: cout<<"Enter an account no. in which you want to deposit:"<<"\n";
        cin>>acno;
        for(i=0;i<5;i++)
        {
        if(acno==obj[i].acn)
        obj[i].deposit();
        }
        break;
case 2: cout<<"Enter an account no. in which you want to withdraw:"<<"\n";
        cin>>acno;
        for(i=0;i<5;i++)
        {
        if(acno==obj[i].acn)
        obj[i].withdraw();
        }
        break;
case 3: for(i=0;i<5;i++)
        obj[i].display();
        break;
default:cout<<"Wrong choice";
}
cout<<"Do you want to continue (y/n)"<<"\n";
```

```
cin>>choice;
}
while(choice=='y' || choice=='Y');
getch();
}
```

UNIT-III

Q.6. When do you need operator overloading? Illustrate bianry operator overloading through a working a working program which shows multiplication of two complex numbers. **(12.5)**

Ans. C++ allows you to specify more than one definition for a function name or an operator in the same scope, which is called function overloading and operator overloading respectively.

You can redefine or overload most of the built-in operators available in C++. Overloaded operators are functions with special names the keyword operator followed by the symbol for the operator being defined. Like any other function, an overloaded operator has a return type and a parameter list.

Program: Overload the binary * operator.

```
#include<iostream.h>
using namespace std;
class complex{
  float x1,y1;
     public:
          void disp(){cout<<x1<<"+"<<y1<<"i"<<"\n";}
          complex(float a,float b)
          { x1=a; y1=b; }
          complex operator*(complex cc)
{ return complex(x1*cc.x1, y1*cc.y1);}
};
int main(){
     complex c1(2.5,-0.3);
     complex c2(4.5,7);
     complex c3=c1*c2;
     c3.disp();
     return 0;
}
```

Q.7. Write a program that counts the number of lines in a file considering a text file consisting of data that is passed to a function for counting the lines. **(12.5)**

Ans.

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

int number_of_lines = 0;
```



```
void numberoflines();
int main(){
    string line;
    ifstream myfile("textexample.txt");
  if(myfile.is_open()){
    while(!myfile.eof()){
      getline(myfile,line);
      cout<< line << endl;
      number_of_lines++;
    }
     myfile.close();
  }
   numberoflines();
}
void numberoflines(){
   number_of_lines—;
   cout<<"number of lines in text file: " << number_of_lines << endl;
}
```

UNIT-IV

Q.8.(a) What do you understandd by the concept of recursion? What is its advantages and illustrate the use by making your own power function through recursion. **(6.5)**

Ans. Recursion is the process of repeating items in a self-similar way. In programming languages, if a program allows you to call a function inside the same function, then it is called a recursive call of the function. But while using recursion, programmers need to be careful to define an exit condition from the function, otherwise it will go into an infinite loop. Recursive functions are very useful to solve many mathematical problems, such as calculating the factorial of a number, generating Fibonacci series, etc.

Advantages of Recursion: 1. Using recursion we can avoid unnecessary calling of functions.

2. Through Recursion one can Solve problems in easy way while its iterative solution is very big and complex. Ex : Tower of Hanoi. You reduce size of the code when you use recursive call.

3. Recursion is used to divide the problem into same problem of subtypes and hence replaces complex nesting code.

4. A recursive definition defines an object in simpler cases of itself reducing nested looping complexity.

5. Recursive functions can be effectively used to solve problems where the solution is expressed in terms of applying the same solution.

Power function using recursion :

```
#Include<iostream.h>
Int main()
{
Int pow,num;
long Int res;
long Int power(int,Int);
```

```
cout<<"Enter a number and a power:";
cin>>num>>pow;
res=power(num,pow);
cout<<"Result is"<<res;
return 0;
}
int i=1;
long int sum=1;
long int power(int num,int pow){
    if(i<=pow){
      sum=sum*num;
      power(num,pow-1);
    }
    else
    return sum;
}
```

Q.8. (b) What do you mean by exception handling? Explain various constracts supported by it through illustrations. **(6)**

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

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

- **Throw:** A program throws an exception when a problem shows up. This is done using a **throw** keyword.
- **Catch:** A program catches an exception with an exception handler at the place in a program where you want to handle the problem. The **catch** keyword indicates the catching of an exception.
- **Try:** A **try** block identifies a block of code for which particular exceptions will be activated. It's followed by one or more catch blocks.

The following is an example, which throws a division by zero exception and we catch it in catch block.

```
#include <iostream>
using namespace std;
double division(int a, int b)
{
  if( b == 0 )
  {
     throw "Division by zero condition!";
  }
  return (a/b);
}
int main ()
{
  int x = 50;
  int y = 0;
```



```
double z = 0;
try {
  z = division(x, y);
  cout << z << endl;
}catch (const char* msg) {
  cerr << msg << endl;
  }
  return 0;
}
```

Q.9 (a) Differentiate between function overloading and function overriding with the help of small working programs. **(6)**

Ans: Overloading is defining functions that have similar signatures, yet have different parameters.

Overriding is only pertinent to derived classes, where the parent class has defined a method and the derived class wishes to override that function.

Function Overloading

```
/*Calling overloaded function test() with different arguments.*/
#include <iostream>
using namespace std;
void test(int);
void test(float);
void test(int, float);
int main() {
   int a = 5;
   float b = 5.5;
   test(a);
   test(b);
   test(a, b);
   return 0;
}
void test(int var) {
     cout<<"Integer number: "<<var<<endl;
}
void test(float var){
    cout<<"Float number: "<<var<<endl;
}
void test(int var1, float var2) {
   cout<<"Integer number: "<<var1;
   cout<<" And float number:"<<var2;
}
```

Function Overriding

```
class A
{
 int a;
 public:
  A()   {       a = 10;   }
```

```
    void show()
    {          cout << a;        }
};
class B: public A
{
    int b;
    public:
    B()      {      b = 20;   }
    void show()
    {          cout << b;        }
};
int main()
{
    A ob1;
    B ob2;
    A::show();
    B::show();
    return 0;
}
```

Q.9. (*b*) What is standard template library? How is it different from the C++ standard library? Why is it gaining importance among the programmers? (6)

Ans. The C++ STL (Standard Template Library) is a powerful set of C++ template classes to provides general-purpose templatized classes and functions that implement many popular and commonly used algorithms and data structures like vectors, lists, queues, and stacks.

At the core of the C++ Standard Template Library are following three well-structured components:

Component	Description
Containers	Containers are used to manage collections of objects of a certain kind. There are several different types of containers like deque, list, vector, map etc.
Algorithms	Algorithms act on containers. They provide the means by which you will perform initialization, sorting, searching, and transforming of the contents of containers.
Iterators	Iterators are used to step through the elements of collections of objects. These collections may be containers or subsets of containers.

Difference between STL and C++ Standard Library: The standard C++ library is a collection of functions, constants, classes, objects and templates that extends the C++ language providing basic functionality to perform several tasks, like classes to interact with the operating system, data containers, manipulators to operate with them and algorithms commonly needed.

The Standard Template Library (STL), part of the C++ Standard Library, offers collections of algorithms, containers, iterators, and other fundamental components, implemented as templates, classes, and functions essential to extend functionality and standardization to C++. STL main focus is to provide improvements implementation standardization with emphasis in performance and correctness.

FIRST TERM EXAMINATION [MARCH–2016]
FOURTH SEMESTER [B.TECH]
OBJECT ORIENTED PROGRAMMING
[ETCS–210]

M.M. : 30

Time : 1½ hrs.

Note: *Question No.1 is compulsory. Attempt any two more Questions from the rest.*

Q.1. (*a*) Determine the output for the following code. (2)

```
int main()
{ int x = 10, y = 20;
int * ptr = &x;
int &ref = y;
* ptr ++;
ref ++;
cout <<x <<"" < < y;
return 0;
}
```

Ans. Output :10 21 **(2)**

Q.1. (*b*) Determine the output for the following code.

```
class A
{
int x = 10;
Public:
Void display()
{
cout<<" The value of X = "<<x<<endl;"
}
};
void main()
{
A Obj;
Obj. display();
}
```

Ans. Output: Error in Line no. 3. "Can Not Initialize a Class Member Here". **(2)**

Q.1. (*c*) Determine the output for the following code.

```
class Test
{
static int i;
int j;
};
int Test::i;
int main()
{
```

```
cout<< size of (Test):
return 0:
}
```

Ans. Output:2

Q.1. (*d*) Determine the output for the following code. **(2)**

```
class A
{
int i, j;
Public:
Void setdata()
{
i = 10;
j = 20;
}
void getdata (int i, int j)
i = i;
j=j;
cout<< "Value of I is  = <<i<<endl;
cout<<"Value of J is = " <<j <<endl;
}};
void main()
{
A Obj;
Obj.setdata();
Obj.getdata(2,3);
}
```

Ans. Output : Value of I is =2
Value of J is =3

Q.1. (*e*) Describe the use of scope resolution operator and referece operator. **(1)**

Ans. Scope resolution operator (::): Scope resolution operator(::) is used to define a function outside a class or when we want to use a global variable but also has a local variable with same name.

Reference operator (&): A reference variable is an alias, that is, another name for an already existing variable. Once a reference is initialized with a variable, either the variable name or the reference name may be used to refer to the variable.

Q.1. (*f*) What is the role of 'new' operator in C++? **(1)**

Ans. New Operator: You can allocate memory at run time within the heap for the variable of a given type using a special operator in C++ which returns the address of the space allocated. This operator is called new operator.

For example we can define a pointer to type double and then request that the memory be allocated at execution time. We can do this using the new operator with the following statements:

```
double* pvalue = NULL; // Pointer initialized with null
pvalue = new double; // Request memory for the variable
```


Q.2.(*a*) Explain the various features of object oriented programming. **(5)**

Ans. Refer Q.3(*b*) of End Term Examination 2016.

Q.2. (*b*) What is class? Write a program to create a class called employee which consist name, desg, ecode and salary as a data member and read, write as a function member, using this class to read and print 10 employee information. **(5)**

Class: A class in C++ is a user defined type or data structure declared with keyword class that has data and functions (also called methods) as its members whose access is governed by the three access specifiers private, protected or public (by default access to members of a class is private).

Program:

```
#include<iostream.h>
#include<string.h>
class employee
{
public:
char name[20];
char desg[20];
int ecode;
long int sal;
void read()
{
cout<<"Enter the name and designation of employee";
gets(name);
gets(desg);
cout<<"Enter the code and salary of employee";
cin>>ecode>>sal;
}
void print()
{
cout<<"Name is "<<name<<"\t"<<"Designation is "<<desg<<"\t"<<"Code is
"<<code<<"\t"<<"Salary is "<<sal<<"\n";
}
};
void main()
{
employee e[10];
int i;
cout<<"Enter the record of 10 employees :"
for(i=0;i<10;i++)
e[i].read();
for(i=0;i<10;i++)
e[i].print();
}
```

Q.3. (*a*) What is constructor? Mention its type. Explain copy constructor with example. **(5)**

Ans. Constructor: It is a member function having same name as it's class and which is used to initialize the objects of that class type with a legel initial value. Constructor is automatically called when object is created.

Types of Constructor

Default Constructor: A constructor that accepts no parameters is known as default constructor. If no constructor is defined then the compiler supplies a default constructor.

```
Circle :: Circle()
{
  radius = 0;
}
```

Parameterized Constructor : A constructor that receives arguments/parameters is called parameterized constructor.

```
Circle :: Circle(double r)
{
   radius = r;
}
```

Copy Constructor-: A constructor that initializes an object using values of another object passed to it as parameter, is called copy constructor. It creates the copy of the passed object.

```
Circle :: Circle(Circle &t)
{
   radius = t.radius;
}
```

There can be multiple constructors of the same class, provided they have different signatures.

Example :

```
class Circle //specify a class
{
   private :
      double radius; //class data members
   public:
Circle() //default constructor
{
  radius = 0;
}
Circle(double r) //parameterized constructor
{
  radius = r;
}
Circle(Circle &t) //copy constructor
{
  radius = t.radius;
}
void setRadius(double r) //function to set data
{
 radius = r;
 }
    double getArea()
```



```
 {
   return 3.14 * radius * radius;
 }
};

int main()
{
  Circle c1; //defalut constructor invoked
  Circle c2(2.5); //parmeterized constructor invoked
  Circle c3(c2); //copy constructor invoked
  cout << c1.getArea()<<endl;
  cout << c2.getArea()<<endl;
  cout << c3.getArea()<<endl;
  return 0;
}
```

Q.3. (b) Write a C++ program to keep track of the number of objects created a particular class without using extern variable. **(5)**

Ans.

```
#include<iostream.h>
class A
{
static int count;
public:
A() //constructor
{
cout<<(++count)<<"Objects have been created";
}
};
int A::count=0; //initialize
int main()
{
A ob1,ob2,ob3;
return 0;
}
```

Q.4.(a) Why friend function is required? Write a program to add two complex number using friend function. **(5)**

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

A friend can be a function, function template, or member function, or a class or class template, in which case the entire class and all of its members are friends.

To declare a function as a friend of a class, precede the function prototype in the class definition with keyword friend as follows:

```
class Box
{
   double width;
public:
```

```
    double length;
    friend void printWidth( Box box );
    void setWidth( double wid );
};
```

C++ Program for add two complex no using friend function.

```
#include<iostream.h>
#include<conio.h>
class Cmplx1
{
int real,imagin;
public :
void get()
{
cout<<"\n\n\tENTER THE REAL PART : ";
cin>>real;
cout<<"\n\n\tENTER THE IMAGINARY PART : ";
cin>>imagin;
}
friend void sum(Cmplx1,Cmplx1);
};
void sum(Cmplx1 c1,Cmplx1 c2)
{
cout<<"\n\t\tRESULT : ";
cout<<"\n\n\t["<<c1.real<<" + i "<<c1.imagin;
cout<<" ] + [ "<<c2.real<<" + i "<<c2.imagin;
cout<<" ] = "<<c1.real+c2.real<<" + i "<<c1.imagin+c2.imagin;
}
void main()
{
Cmplx1 op1,op2;
clrscr();
cout<<"\n\n\tADDITION OF TWO COMPLEX NUMBERS USING FRIEND
FUNCTIONS\n\n";
cout<<"\n\tINPUT\n\n\t\tOPERAND 1";
op1.get();
cout<<"\n\n\t\tOPERAND 2";
op2.get();
sum(op1,op2);
getch();
}
```

Q.4.(b) Define function overloading. Demonstrate with C++ program. **(5)**

Ans. You can have multiple definitions for the same function name in the same scope. The definition of the function must differ from each other by the types and/or the number of arguments in the argument list. You can not overload function declarations that differ only by return type.

Function Overloading

```
/*Calling overloaded function test() with different arguments.*/
#include <iostream>
using namespace std;
void test(int);
void test(float);
void test(int, float);
int main() {
        int a = 5;
        float b = 5.5;
      test(a);
      test(b);
      test(a, b);
      return 0;
 }
 void test(int var) {
      cout<<"Integer number: "<<var<<endl;
 }
 void test(float var){
      cout<<"Float number: "<<var<<endl;
 }
}
void test(int var1, float var2) {
      cout<<"Integer number: "<<var1;
      cout<<" And float number:"<<var2;
}
```

SECOND TERM EXAMINATION [APRIL-2016]
FOURTH SEMESTER [B.TECH]
OBJECT ORIENTED PROGRAMMING
[ETCS–210]

Time : 1½ hrs. **M.M. : 30**

Note: *Question No.1 is compulsory. Attempt any two more Questions from the rest.*

Q.1.(a) Differentiate between early and Late binding. (2)

Ans. Early Binding: Events occurring at compile time are known as early binding. In the process of early binding all info which is required for a function call is known at compile time. Early binding is a fast and efficient process. Examples of early binding: function calls, overloaded function calls, and overloaded operators.

Late Binding: In this process all info which is required for a function call is not known at compile time. Hence, objects and functions are not linked at run time. Late Binding is a slow process. However, it provides flexibility to the code. Example of Late Binding: Virtual functions.

Q.1.(b) How to properties of the following two derived classes differ? (2)
(i) class D1 : private B { } (ii) class D1 : public B { }

Ans. (*i*) Class D1: private B{}: In this statement the Class D1 is privately derived from Base Class B.

(*ii*) Class D1 : Public B { }: In this statement the Class D1 is publically derived from Base Class B.

Q.1.(c) Explain hybrid inheritance with example. (2)

Ans. Hybrid Inheritance:

Hybrid Inheritance is a method where one or more types of inheritance are combined together and used.

Program:

```
#include<iostream.h>
#include<conio.h>
class arithmetic
{
protected:
int num1, num2;
public:
void getdata()
{
cout<<"For Addition:";
cout<<"\nEnter the first number: ";
cin>>num1;
cout<<"\nEnter the second number: ";
cin>>num2;
}
};
class plus:public arithmetic
{
protected:
int sum;
public:
void add()
{
sum=num1+num2;
}
};
class minus
{
protected:
int n1,n2,diff;
public:
void sub()
{
cout<<"\nFor Subtraction:";
cout<<"\nEnter the first number: ";
cin>>n1;
cout<<"\nEnter the second number: ";
cin>>n2;
diff=n1-n2;
}
};
class result:public plus, public minus
{
public:
void display()
{
cout<<"\nSum of "<<num1<<" and "<<num2<<"= "<<sum;
cout<<"\nDifference of "<<n1<<" and "<<n2<<"= "<<diff;
}
};
void main()
{
clrscr();
result z;
z.getdata();
z.add();
z.sub();
z.display();
getch();
}
```

Q.1. (d) What are containers? Describe various types of containers. (2)

Ans. Containers are used to manage collections of objects of a certain kind. There are several different types of containers like deque, list, vector, map etc. The containers are class templates; when you declare a container variable, you specify the type of the elements that the container will hold. Containers can be constructed with initializer lists. They have member functions for adding and removing elements and performing other operations.

Two basic types of containers:

- Sequences
  - User controls the order of elements.

- vector, list, deque
- Associative containers
  - The container controls the position of elements within it.
  - Elements can be accessed using a *key*.
  - set, multiset, map, multimap

Q.1. (c) What is difference between a class and an abstract class in C++? (2

Ans. Abstract Class: Abstract Class is a class which contains atleast one Pure Virtual function in it. Abstract classes are used to provide an Interface for its sub classes. Classes inheriting an Abstract Class must provide definition to the pure virtual function, otherwise they will also become abstract class.

Class: A class in C++ is a user defined type or data structure declared with keyword class that has data and functions (also called methods) as its members whose access is governed by the three access specifiers private, protected or public (by default access to members of a class is private).

Q.2. (i) Create three classes namely Student, Exam and Result. Student class represents student information. Exam class, inherited from student class represents marks scored in five subjects. Result class is derived from exam class, calculating total marks. Write an interactive program to represents this inheritance. Also the type of inheritance depicted by this program. (8)

Ans.

```
#include<iostream.h>
#include<conio.h>
#include<string.h>
class student    // base class
{
private:
 int rl;
 char nm[20];
public:
  void read();
  void display();
};
class exam : public student // dervivec from student
{
protected:
 int s1;
 int s2;
 int s3;
public:
  void getmarks();
  void putmarks();
};
class result : public exam // derived  from marks
{
private:
 int t;
float p;
public:
void process();
void printresult();
};
void student::read()
{
 cout<<"enter Roll no and Name "<<endl;
 cin>>rl>>nm;
}
void student:: display()
{
cout <<"Roll NO:"<<rl<<endl;
cout<<"name : "<<nm<<endl;
}
void exam ::getmarks()
{
cout<<"enter three subject marks  "<<endl;
 cin>>s1>>s2>>s3;
}
void exam ::putmarks()
{
 cout <<"subject 1:"<<s1<<endl;
 cout<<" subject 2 : "<<s2<<endl;
 cout <<"subject 3:"<<s3<<endl;
}

void result::process()
{
  t= s1+s2+s3;
  p = t/3.0;
  }
void result::printresult()
{
 cout<<"total = "<<t<<endl;
 cout<<"per  = "<<p<<endl;
 }
void main()
{
 result x;
 clrscr();
 x.read();
 x.getmarks();
 x.process();
 x.display();
 x.putmarks();
 x.printresult();
 getch();
 }
```

This the the program of Multilevel Inheritance.

Q.2.(ii) Distinguish between virtual function and pure virtual functions with example? **(4)**

Ans. Virtual Function and Pure Virtual Function defers in declaration. Virtual Function is declared with keyword 'virtual' at the start of declaration.

Example . virtual return_type function_name(function arguments);

While Pure Virtual Function is declared as

Example . virtual return_type function_name(function arguments) = 0;

Sometime it has been written that pure virtual function has no function body. But this is not always true. Having a default definition of pure virtual function is not common but its possible.

Example : virtual return_type function_name(function arguments) = 0; //Declaration

```
return_type class_name::function_name(function argument) //Definition
{
.
.
.
}
```

Virtual Function and Pure Virtual Function defers in use also. Virtual Function makes its class a polymorphic base class. Derived classes may override the virtual function and redefine the behavior. Virtual Function called from base class pointer/ reference will be decided on run time. Which is called run time binding.

Pure Virtual Function provides the similar functionality as Virtual Function. Apart from that Pure Virtual Function makes class a Abstract Class. It means, if a class have Pure Virtual Function, class can't be instantiated. Only reference or pointers can be created of such class.

Pure Virtual Function in base classes enforce derived classes to define the pure virtual function. Else derived class will also be Abstract Class.

Q.3. (i) Wirte a program to overload the pre-increment and post-increment++ operator. **(5)**

Ans.

```
#include <iostream.h>
using namespace std;
class Integer {
  private:
    int value;
  public:
    Integer(int v) : value(v) { }
    Integer operator++();
    Integer operator++(int);
    int get Value() {
      return value;
    }
};
// Pre-increment Operator
Integer Integer::operator++()
{
  value++;
  return *this;
}
```



```
// Post-increment Operator
Integer Integer::operator++(int)
{
  const Integer old(*this);
  ++(*this);
  return old;
}
int main()
{
  Integer i(10);
  cout << "Post Increment Operator" << endl;
  cout << "Integer++ : " << (i++).getValue() << endl;
  cout << "Pre Increment Operator" << endl;
  cout << "++Integer : " << (++i).getValue() << endl;
}
```

Q.3. (ii) Differentiate between Function Overloading and Function Overriding with example? **(5)**

Ans.

Overriding	Overloading
Functions name and signatures must be same.	Having same Function name with different Signatures.
Overriding is the concept of runtime polymorphism.	Overloading is the concept of compile time polymorphism.
When a function of base class is re-defined in the derived class called as Overriding.	Two functions having same name and return type, but with different type and/or number of arguments is called as Over-loading.
It needs inheritance.	It doesn't need inheritance.
Functions should have same data type.	Functions can have different data types.
Function should be public.	Function can be different access specifies.

Function Overloading

```
/*Calling overloaded function test() with different arguments.*/
#include <iostream.h>
using namespace std;
void test(int);
void test(float);
void test(int, float);
int main() {
   int a = 5;
   float b = 5.5;
   test(a);
   test(b);
   test(a, b);
   return 0;
}
void test(int var) {
```

```
  cout<<"Integer number: "<<var<<endl;
}
void test(float var){
  cout<<"Float number: "<<var<<endl;
}
void test(int var1, float var2) {
  cout<<"Integer number: "<<var1;
  cout<<" And float number:"<<var2;
}
```

Function Overriding

```
class A
{
      int a;
      public:
       A()  {          a = 10;       }
       void show()
       {          cout << a;       }
};
class B: public A
{
      int b;
      public:
      B()       {          b = 20;       }
      void show()
      {          cout << b;       }
};
int main()
{
     A ob1;
      B ob2;
      A::show();
      B::show();
      return 0;
}
```

Q.4.(*i*) Write a program to demonstrate the use of multiple catch block. (5)

Ans.

```
#include<iostream.h>
#include<conio.h>
void test(int x)
{
 try
 {
       if(x>0)
         throw x;
   else
       throw 'x';
}

catch(int x)
{
```



```
      cout<<"Catch a integer and that integer is:"<<x;
 }

 catch(char x)
 {
      cout<<"Catch a character and that character is:"<<x;
 }
}

void main()
{
  clrscr();
  cout<<"Testing multiple catches\n:";
  test(10);
  test(0);
  getch();
}
```

Q.4. (*ii*) What is generic programming? How is it implemented in C++? (5)

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

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

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

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

Function Template:

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

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

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

Class Template:

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

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

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

END TERM EXAMINATION [MAY-JUNE 2016]
FOURTH SEMESTER [B.TECH]
OBJECT ORIENTED PROGRAMMING
[ETCS–210]

Time : 3 hrs. **M.M. : 7**

Note: *Attempt any five questions including Q.No. 1 which is compulsory. Select one question from each unit.*

Q.1. (a) How does new operator in C++ is different from dynamic allocation functions in C? **(2.5×10=25)**

Ans. 1.Operator new constructs an object (calls constructor of object), malloc does not. Hence new invokes the constructor (and delete invokes the destructor) This is the most important difference.

2. Operator new is an operator, malloc is a function.

3. Operator new can be overloaded, malloc cannot be overloaded.

4. Operator new throws an exception if there is not enough memory, malloc returns a NULL.

5.Operator new[] requires you to specify the number of objects to allocate, malloc requires you to specify the total number of bytes to allocate.

6. Operator new/new[] must be matched with operator delete/delete[] to deallocate memory, malloc() must be matched with free() to deallocate memory.

Q.1. (b) How static member is different from a non static member?

Ans. STATIC DATA MEMBER: A data member of a class can be qualified as static. The properties of a static member variable are similar to that of a static variable. A static member variable has certain special characeristics.

(1) It is initialized to 0(zero) when the first object of this class is created. No other initialization is permitted.

(2) Only one copy of that member is created for the entire class and is shared by all the objects of that class, No matter how many objects are created.

(3) It is visible only within the class but its lifetime is in the entire program.

The type and scope of each static member variable must be defined outside the class definition.This is necessary because the static data member are stored separately rather than as a part of an object. Since they are associated with the class itself rather than with any object. They are also known as class variable.

NON STATIC DATA MEMBER: Data member is the simple variable of a class which is initialized by member function of same class. By default data member is private and external functions can not access it.

Q.1. (c) What do you mean by an abstract class?

Ans. Abstract Class: Abstract Class is a class which contains atleast one Pure Virtual function in it. Abstract classes are used to provide an Interface for its sub classes. Classes inheriting an Abstract Class must provide definition to the pure virtual function, otherwise they will also become abstract class.

Characteristics of Abstract Class:

1. Abstract class cannot be instantiated, but pointers and refrences of Abstract class type can be created.

2. Abstract class can have normal functions and variables along with a pure virtual function.

3. Abstract classes are mainly used for Upcasting, so that its derived classes can use its interface.

4. Classes inheriting an Abstract Class must implement all pure virtual functions, or else they will become Abstract too.

Q.1. (d) What is Inline function? How is it different from a macro?

Ans. Inline function: C++ **inline** function is powerful concept that is commonly used with classes. If a function is inline, the compiler places a copy of the code of that function at each point where the function is called at compile time. To inline a function, place the keyword inline before the function name and define the function before any calls are made to the function.

```
// The use of the 'inline' keyword

inline int sum(int a, int b)
{
   return (a+b);
}
```

The major difference between inline functions and macros is the way they are handled. Inline functions are parsed by the compiler, whereas macros are expanded by the C++ preprocessor. Inline follows strict parameter type checking, macros do not. Debugging of macros is also difficult. This is because the preprocessor does the textual replacement for macros, but that textual replacement is not visible in the source code itself. Because of all this, it's generally considered a good idea to use inline functions over macros.

Q.1.(e) How does C++ support data abstraction?

Ans. Data Abstraction : Data abstraction refers to, providing only essential information to the outside world and hiding their background details, i.e., to represent the needed information in program without presenting the details. Data abstraction is a programming (and design) technique that relies on the separation of interface and implementation.

While defining a class, both member data and member functions are described. However while using an object (that is an instance of a class) the built in data types and the members in the class are ignored. This is known as **data abstraction**. In C++, we use **classes** to define our own abstract data types (ADT). You can use the **cout** object of class iostream to stream data to standard output like this:

```
#include <iostream.h>
int main( )
{
  cout << "Hello C++" <<endl;
  return 0;
}
```

Here, you don't need to understand how cout displays the text on the user's screen. You need to only know the public interface and the underlying implementation of cout is free to change.

Q.1. (f) What do you mean by a generic data type?

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

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

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

You can use templates to define functions as well as classes.

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

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

The general form of a generic class declaration is shown here:

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

Q.1. (g) Explain the role of reference variable with suitable example?

Ans. Reference variable : A reference variable is an alias, that is, another name for an already existing variable. Once a reference is initialized with a variable, either the variable name or the reference name may be used to refer to the variable.

For example, suppose we have the following example:

```
int   i = 17;
```

We can declare reference variables for i as follows.

```
int&   r = i;
```

References are usually used for function argument lists and function return values.

Q.1. (h) What do you mean by the name space concept?

Ans. For example, you might be writing some code that has a function called xyz() and there is another library available which is also having same function xyz(). Now the compiler has no way of knowing which version of xyz() function you are referring to within your code.

A **namespace** is designed to overcome this difficulty and is used as additional information to differentiate similar functions, classes, variables etc. with the same name available in different libraries. Using namespace, you can define the context in which names are defined. In essence, a namespace defines a scope.

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

```
namespace namespace_name {
// code declarations
}
```

To call the namespace-enabled version of either function or variable, prepend the namespace name as follows:

```
name::code; // code could be variable or function.
```

Q.1. (i) What do you mean by base address of an array? Can you access the content of an array by incrementing the base address value?

Ans. An array is a sequence of objects, each of the same data type. The starting address of this array of objects, i.e. the address of the first object in the array is called the **base address** of the array. The address of each successive element of the array is offset from the base by the size of the array type, e.g. for each successive element of an integer array, the address is offset by the size of an integer type object.

Yes we can access the content of an array by incrementing the base address value.

Q.1.(f) What is type casting and when is it used?

Ans. "Type Casting" is method using which a variable of one datatype is converted to another datatype, to do it simple the datatype is specified using parenthesis in front of the value.

Example:

```
#include <iostream.h> using namespace std; int main()
{
short x=40;
int y;
y = (int)x;
cout << "Value of y is:: "<< y <<"\nSize is::"<< sizeof(y);
return 0;
}
```

Result:

```
Value of y is:: 40
Size is::4
```

In the above example the short data type value of x is type cast to an integer data type, which occupies "4" bytes.

Type casting can also done using some typecast operators available in C++. following are the typecast operators used in C++.

- static_cast
- const_cast
- dynamic_cast
- reinterpret_cast
- typeid

UNIT-I

Q.2. (a) Wirte a C++ code to store 6 integer numbers using dynamic memory allocation approach. **(6)**

Ans.

```
#include <iostream.h>
using namespace std;
int main()
{
int n, *pointer, c;
pointer = new int[6];
cout <<"Input 6 integers\n";
for (c = 0; c <6; c++)
cin >> pointer[c];
cout <<"Elements entered by you are\n";
for (c = 0; c <6; c++)
cout << pointer[c] << endl;
delete[] pointer;
return 0;
}
```

Q.2. (b) Write a program to print the following output. (6.5)

```
1
2 2
3 3 3
4 4 4 4
```

Ans.

```
#include<iostream.h>
#include<conio.h>
void main()
{
int i, j, n;
cout<<"Enter the number of rows";
cin>>n;
for(i=1;i<=n; i++)
{
for(j=1; j <=i; j++)
{
cout<<i;
}
cout<<"\n";
}
getch();
}
```

Q.3. (a) Write a function for multiplication of two matrices. (6)

Ans.

```
void multiplication(int a[][10],int b[][10],int mult[][10],int r1,int c1,int r2,int c2)
{
  int i,j,k;
/* Initializing elements of matrix mult to 0.*/
  for(i=0; i<r1; ++i)
  for(j=0; j<c2; ++j)
  {
    mult[i][j]=0;
  }
/* Multiplying matrix a and b and storing in array mult. */
  for(i=0; i<r1; ++i)
  for(j=0; j<c2; ++j)
  for(k=0; k<c1; ++k)
  {
    mult[i][j]+=a[i][k]*b[k][j];
  }
}
void display(int mult[][10], int r1, int c2)
{
```



```
  int i, j;
  cout << endl << "Output Matrix: " << endl;
  for(i=0; i<r1; ++i)
  for(j=0; j<c2; ++j)
  {
    cout << " " << mult[i][j];
    if(j==c2-1)
      cout << endl;
  }
}
```

Q.3. (b)Explain various object oriented features. (6.5)

Ans. Features of OOPs :-

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

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

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

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

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

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

This is a very important concept of object-oriented programming since this feature helps to reduce the code size.

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

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

UNIT-II

Q.4. (a) What is a friend function? Write its merits and demerits. (6)

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

A friend can be a function, function template, or member function, or a class or class template, in which case the entire class and all of its members are friends.

To declare a function as a friend of a class, precede the function prototype in the class definition with keyword friend as follows:

```
class class_name
{
  ..... .... .......
  friend return_type function_name(argument/s);
  ..... .... .......
}
```

Merits:

1. We can able to access the other class members in our class if, we use friend keyword.

2. We can access the members without inheriting the class.

Demerits:

1.Maximum size of the memory will occupied by objects according to the size of friend members.

2.We can't do any run time ploymorphism concepts in those members.

Q.4.(b) Write a program that will add two time values measured in terms of hours and minutes. **(6.5)**

Ans.

```
#include <iostream.h>
using namespace std;
class time {
private:
  int hr;
  int minute;
public:
  void get();
  void disp();
  class time add(class time t1, class time t2);
};
void time:: get(){
  cout << "Enter Hours: " ;
  cin >> hr;
  cout << endl;
  cout << "Enter Minutes:" ;
  cin >> minute;
  cout << endl;
}
void time:: disp(){
  cout << "Hours ------> " << hr << "hrs" << endl;

  cout << "Minutes ----> " << minute <<"mins"<< endl;
}
class time time:: add(class time rt1, class time rt2)
{
```



```
 class time rt3;
 rt3.minute = rt1.minute + rt2.minute;
 rt3.hr = rt1.hr + rt2.hr;
   if(rt3.minute >= 60){
   rt3.hr = rt3.hr + ((rt3.minute)/60);
   rt3.minute = rt3.minute % 60;
 }
 return rt3;
}
int main(){
  class time t1,t2,t3,t;
  t1.get();
  t2.get();
  t1.disp();
  cout<<"\n";
  t2.disp();
  cout<<"\n";
  cout << "Sum of times:"<< endl << endl;
  t3 = t.add(t1,t2);
  t3.disp();
}
```

Q.5.(a) What is a copy constructor? Explain with suitable example. **(6)**

Ans. The copy constructor is a constructor which creates an object by initializing it with an object of the same class, which has been created previously. The copy constructor is used to:

- Initialize one object from another of the same type.
- Copy an object to pass it as an argument to a function.
- Copy an object to return it from a function.

If a copy constructor is not defined in a class, the compiler itself defines one. If the class has pointer variables and has some dynamic memory allocations, then it is a must to have a copy constructor. The most common form of copy constructor is shown here:

```
classname (const classname &obj) {
  // body of constructor
}
```

Here, obj is a reference to an object that is being used to initialize another object.

Following is a simple example of copy constructor.

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

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

To declare a function as a friend of a class, precede the function prototype in the class definition with keyword friend as follows:

```
class class_name
{
  ..... ... .......
  friend return_type function_name(argument/s);
  ..... ... .......
}
```

Merits:

1. We can able to access the other class members in our class if, we use friend keyword.
2. We can access the members without inheriting the class.

Demerits:

1.Maximum size of the memory will occupied by objects according to the size of friend members.

2.We can't do any run time ploymorphism concepts in those members.

Q.4.(b) Write a program that will add two time values measured in terms of hours and minutes. **(6.5)**

Ans.

```
#include <iostream.h>
using namespace std;
class time {
private:
  int hr;
  int minute;
public:
  void get();
  void disp();
  class time add(class time t1, class time t2);
};
void time:: get(){
  cout << "Enter Hours: " ;
  cin >> hr;
  cout << endl;
  cout << "Enter Minutes:" ;
  cin >> minute;
  cout << endl;
}
void time:: disp(){
  cout << "Hours ------> " << hr << "hrs" << endl;

  cout << "Minutes ---> " << minute <<"mins"<< endl;
}
class time time:: add(class time rt1, class time rt2)
{
```



```
  class time rt3;
  rt3.minute = rt1.minute + rt2.minute;
  rt3.hr = rt1.hr + rt2.hr;
    if(rt3.minute >= 60){
    rt3.hr = rt3.hr + ((rt3.minute)/60);
    rt3.minute = rt3.minute % 60;
  }
  return rt3;
}
int main(){
  class time t1,t2,t3,t;
  t1.get();
  t2.get();
  t1.disp();
  cout<<"\n";
  t2.disp();
  cout<<"\n";
  cout << "Sum of times:"<< endl << endl;
  t3 = t.add(t1,t2);
  t3.disp();
}
```

Q.5.(a) What is a copy constructor? Explain with suitable example. **(6)**

Ans. The copy constructor is a constructor which creates an object by initializing it with an object of the same class, which has been created previously. The copy constructor is used to:

- Initialize one object from another of the same type.
- Copy an object to pass it as an argument to a function.
- Copy an object to return it from a function.

If a copy constructor is not defined in a class, the compiler itself defines one. If the class has pointer variables and has some dynamic memory allocations, then it is a must to have a copy constructor. The most common form of copy constructor is shown here:

```
classname (const classname &obj) {
  // body of constructor
}
```

Here, obj is a reference to an object that is being used to initialize another object.

Following is a simple example of copy constructor.

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

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

```
    Point(int x1, int y1) { x = x1; y = y1; }
    // Copy constructor
    Point(const Point &p2) { x = p2.x; y = p2.y; }

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

int main()
{
  Point p1(10, 15); // Normal constructor is called here
  Point p2 = p1; // Copy constructor is called here

  // Let us access values assigned by constructors
  cout << "p1.x = " << p1.getX() << ", p1.y = " << p1.getY();
  cout << "\np2.x = " << p2.getX() << ", p2.y = " << p2.getY();

  return 0;
}
```

Q.5. (b) Write a program that overloads and unary minus operator. **(6.5)**

Ans.

```
#include <iostream.h>
using namespace std;
class Distance
{
 private:
   int feet;          // 0 to infinite
   int inches;        // 0 to 12
 public:
   // required constructors
   Distance(){
     feet = 0;
     inches = 0;
   }
   Distance(int f, int i){
     feet = f;
     inches = i;
   }
   // method to display distance
   void displayDistance()
   {
     cout << "F: " << feet << " I:" << inches <<endl;
   }
   // overloaded minus (-) operator
   Distance operator- ()
   {
```



```
     feet = -feet;
     inches = -inches;
     return Distance(feet, inches);
   }
};
int main()
{
  Distance D1(11, 10), D2(-5, 11);

  -D1;              // apply negation
  D1.displayDistance();   // display D1

  -D2;              // apply negation
  D2.displayDistance();   // display D2

  return 0;
}
```

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

```
F: -11 I:-10
F: 5 I:-11
```

UNIT-III

Q.6. What do you mean by polymorphism? Explain various types of polymorphism with suitable example. **(12.5)**

Ans. Polymorphism: Polymorphism is the ability to use an operator or function in different ways. Polymorphism gives different meanings or functions to the operators or functions. Poly, referring too many, signifies the many uses of these operators and functions. A single function usage or an operator functioning in many ways can be called polymorphism. Polymorphism refers to codes, operations or objects that behave differently in different contexts.

Types of Polymorphism:

C++ provides two different types of polymorphism.

- run-time
- compile-time

Run-time:

The appropriate member function could be selected while the programming is running. This is known as run-time polymorphism. The *run-time* polymorphism is implemented with inheritance and virtual functions.

- Virtual functions

Virtual functions

A function qualified by the virtual keyword. When a virtual function is called via a pointer, the class of the object pointed to determines which function definition will be used. Virtual functions implement polymorphism, whereby objects belonging to different classes can respond to the same message in different ways.

Example of Virtual Function :

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

```
class base
{
  public.
   virtual void show()
   {
        cout<<"\n Base class show:";
   }
   void display()
   {
        cout<<"\n Base class display:" ;
   }
};
class drive:public base
{
  public:
   void display()
   {
        cout<<"\n Drive class display:";
   }
   void show()
   {
        cout<<"\n Drive class show:";
   }
};
void main()
{
  clrscr();
  base obj1;
  base *p;
  cout<<"\n\t P points to base:\n" ;

  p=&obj1;
  p->display();
  p->show();
  cout<<"\n\n\t P points to drive:\n";
  drive obj2;
  p=&obj2;
  p->display();
  p->show();
  getch();
}
```

Output:

P points to Base

Base class display

Base class show

P points to Drive

Base class Display

Drive class Show

Compile-time:

The compiler is able to select the appropriate function for a particular call at compile-time itself. This is known as compile-time polymorphism. The *compile-time* polymorphism is implemented with templates.

- Function name overloading
- Operator overloading

1. Operator Overloading: The process of making an operator to exhibit different behaviors in different instances is known as operator overloading.

The general form of an operator function is:

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

The process of overloading involves the following steps:

1. Create a class that defines the data type that is to be used in the overloading operation.
2. Declare the operator function operator op () in the public part of the class. It may be either a member function or a friend function.
3. Define the operator function to implement the required operations.

2. Function Overloading: Using a single function name to perform different types of tasks is known as function overloading.Using the concept of function overloading, design a family of functions with one function name but with different argument lists. The function would perform different operations depending on the argument list in the function call. The correct function to be invoked is determined by checking the number and type of the arguments but not on the function type.

Example of Function Overloading :

```
#include <iostream.h>
using namespace std;
class printData
{
  public:
    void print(int i) {
     cout << "Printing int: " << i << endl;
    }
    void print(double  f) {
     cout << "Printing float: " << f << endl;
    }
    void print(char* c) {
     cout << "Printing character: " << c << endl;
    }
};
```

```
int main(void)
{
  printData pd;
  // Call print to print integer
  pd.print(5);
  // Call print to print float
  pd.print(500.263);
  // Call print to print character
  pd.print("Hello C++");
  return 0;
}
```

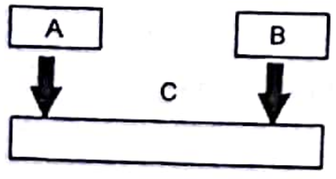
Q.7.(a) Write a template function that finds maximum value in an array which is passed to it as an argument. **(6)**

```
#include<iostream.h>
template<class t>
void maxarr(t a[])
{
t i,max;
max=a[0];
for(i=0;i<5;i++)
{
if(a[i]>max)
max=a[i];
}
cout<<max;
}
void main()
{
int a[5]={10,20,30,40,50};
char b[5]={'a','b','c','d','e'};
maxarr(a);
maxarr(b);
}
```

Q.7. (b) Explain multiple inheritance with suitable example. **(6.5)**

Ans. Multiple Inheritance

In this type of inheritance a single derived class may inherit from two or more than two base classes.




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



```
class student
{
  protected:
    int rno,m1,m2;
  public:
        void get()
        {
                cout<<"Enter the Roll no :";
                cin>>rno;
                cout<<"Enter the two marks  :";
                cin>>m1>>m2;
        }
};
class sports
{
   protected:
     int sm;              // sm = Sports mark
   public:
          void getsm()
         {
           cout<<"\nEnter the sports mark :";
           cin>>sm;

         }
};
class statement:public student,public sports
{
   int tot,avg;
   public:
   void display()
         {
           tot=(m1+m2+sm);
           avg=tot/3;
           cout<<"\n\n\tRoll No   : "<<rno<<"\n\tTotal    : "<<tot;
          cout<<"\n\tAverage   : "<<avg;
         }
};
void main()
{
  clrscr();
   statement obj;
   obj.get();
   obj.getsm();
   obj.display();
   getch();
}
```

In this example class statement is derived from the class student and class sports. So there is multiple inheritance.

UNIT–IV

Q.8. (a) How exception handling is performed in C++? **(6.5)**

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

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

- **throw:** A program throws an exception when a problem shows up. This is done using a throw keyword.
- **catch:** A program catches an exception with an exception handler at the place in a program where you want to handle the problem. The catch keyword indicates the catching of an exception.
- **try:** A try block identifies a block of code for which particular exceptions will be activated. It's followed by one or more catch blocks.

Assuming a block will raise an exception, a method catches an exception using a combination of the try and catch keywords. A try/catch block is placed around the code that might generate an exception. Code within a try/catch block is referred to as protected code, and the syntax for using try/catch looks like the following:

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

Q.8. (b) Write a C++ program that displays the content of a file. **(6)**

Ans.

```
#include<iostream.h>
#include<conio.h>
#include<string.h>
#include<fstream.h>
#include<stdlib.h>
void main()
{
    clrscr();
    ifstream ifile;
    char s[100], fname[20];
    cout<<"Enter file name to read and display its content (like file.txt) : ";
    cin>>fname;
```



```
ifile.open(fname);
if(!ifile)
{
    cout<<"Error in opening file..!!";
    getch();
    exit(0);
}
while(ifile.eof()==0)
{
 ifile>>s;
 cout<<s<<" ";
}
    cout<<"\n";
    ifile.close();
    getch();
}
```

Q.9. Write a program that reads Guru Gobind Singh Indraprastha University from the keboard into five separate string objects and then concatenates them into a new string using append() function. **(12.5)**

Ans.

```
#include<iostream.h>
#include<conio.h>
#include<stdio.h>
void main()
{
char str1[20],str2[20], str3[20], str4[20], str5[20];
char append(char[],char [],char [],char [],char []);
cout<<"Enter 5 strings";
gets(str1);
gets(str2);
gets(str3);
gets(str4);
gets(str5);
append(str1, str2, str3, str4, str5);
getch();
}
char append(char str1[],char str2[], char str3[], char str4[], char str5[])
{
char cnct[100];
int i,j;
for(i=0; str1[i]!='\0'; i++)
cnct[i]=str1[i];
cnct[i]=NULL;
for(i=0;cnct[i]!='\0'; i++);
for(j=0; str2[j]!='\0'; j++)
```

```
{
cnct[i]=str2[j];
i++;
}
cnct[i]=NULL;
for(i=0;cnct[i]!='\0';i++);
for(j=0;str3[j]!='\0';j++)
{
cnct[i]=str3[j];
i++;
}
cnct[i]=NULL;
for(i=0; cnct[i]!='\0'; i++);
for(j=0; str4[j]!='\0'; j++)
{
cnct[i]=str4[j];
i++;
}
cnct[i]=NULL;
for(i=0; cnct[i]!='\0'; i++);
for(j=0;str5[j]!='\0';j++)
{
cnct[i]=str5[j];
i++;
}
cnct[i]=NULL;
puts(cnct);
return(0);
}
```

FIRST TERM EXAMINATION [FEB. 2017]
FOURTH SEMESTER [B.TECH.]
OBJECT ORITENTED PROGRAMMING USING C++ [ETCS–210]

Time : 1½ hrs. **M.M. : 30**

Note: *Q.No. 1 is compulsory. Attempt any two more question from the rest.*

Q.1. (*a*) State the characteristics of Object Oritented Programming. **(2)**

Ans. Class definitions: Basic building blocks OOP and a single entity which has data and operations on data together.

Objects: The instances of a class which are used in real functionality-its variables and operations.

Abstraction: Specifying what to do but not how to do, a flexible feature for having a overall view of an object's functionality

Encapsulation: Binding data and operations of data together in a single unit.

Inheritance and class hierarchy: Reusability and extension of existing classes.

Polymorphism: Multiple definitions for a single name-functions with same name with different functionality.

Message passing: Objects communicates through invoking methods and sending data to them. This feature of sending and receiving information among objects through function parameters is known as Message Passing.

Q.1. (*b*) What is the difference between reference variables and normal variable? **(2)**

Ans. Normal variables carry the specified data where as the pointer variable carried the address of the specified data, *e.g.* if we give int x =10; ptr p = *x; Where x is the normal variable carries 10 and pointer variable is p which carried address of the integer variable x. Normal variable contains the value of variable either int or float whereas pointer variable contains the address of another variable. "Pointer variable holds memory address or physical address of any variable value means it indirectly points any variable, we can perform all operations +,*,–,and / through it but normal variable directly points variable value".

Q.1. (*c*) What are empty classes? Can instances of empty class be created? **(2)**

Ans. We can declare empty classes, but objects of such types still have nonzero size. The memory allocated for such objects is of nonzero size; therefore, the objects have different addresses. Having different addresses makes it possible to compare pointers to objects for identity. Also, in arrays, each member array must have a distinct address. An empty base class typically contributes zero bytes to the size of a derived class.

Q.1. (*d*) What are the differences between default and parameterized constructors? **(2)**

Ans. Default Constructor :

A constructor that has no parameter is called the default constructor. The default constructor for class A is A::A(), If no such constructor is defined, then the compiler supplies a default constructor.Therefore a statement such as

A a;

Invokes the default constructor of the compiler to create the object a.

Parameterized constructor :

The constructor that can take arguments are called the parameterized constructors. In the parameterized constructors we passed the arguments when the object of class are created The constructor integer( ) may be modified to take arguments as shown below.

```
class integer
{
int m,n;
public:
integer(int x,int y)                    //parameterized constructor
{
m=x;
n=y;
}
};
main()
{
integer int(100, 200);                  //invoke the parameterized constructor
}
```

Q.1. (e) Define const member function and const objects. **(2)**

Ans. const member function

If a member function does not alter any data in the class, then we may declare it as a **const** member function as follows :

```
void mul(int, int) const;
```

The qualifier const is appended to the function prototypes. The compiler will generate an error message if such function try to alter the data values.

const Objects

We may create and use constant objects using const keyword before object declaration. For example, we may create X as a constant object of the class matrix as follows :

```
Const matrix X(m, n);                   //object X is constant
```

Any attempt to modify the values of m and n will generate compile time error. Further a constant object can call only const member function.

Q.2. (a) Declare a class to represent Time. Time is expressed in hours, minutes and seconds. Time class includes member functions to input and display time. Use constructor and destructor is program. **(5)**

Ans.

```
#include<iostream.h>
#include<conio.h>
class time
{
int hour;
int min;
int sec;
public:
time(int h,int m,int s)          //parameterized constructor declaration
```



```
{
hour=h;
min=m;
sec=s;
}                                //destructor declaration
~time()
{
cout<<"Destructor invoked";
}
void display()                   //display function
{
cout<<"Time is "<<hour<<" Hour:"<<min<<" Minutes:"<<sec<<" Seconds"<<"\n";
}
};
void main()
{
time t1(5,35,20);                //constructor invoked
clrscr();
t1.display();
getch();
}                                //destructor invoked
```

Q.2. (b) With the help of an example, explain how an object can be passed and returned from a member function of a class. **(5)**

Ans.

```
#include<iostream.h>
#include<conio.h>
class complex
{
float x;
float y;
public:
void input(float real, float imag)
{
x=real;
y=imag;
}
friend complex sum(complex, complex); //declaration with objects as arguments
void show(complex);
};
complex sum(complex c1, complex c2)   //c1, c2 are the objects
{                                     //object c3 is created
complex c3;
```

```
c3.x=c1.x+c2.x;
c3.y=c1.y+c2.y;
return(c3);          // returns object c3
}
void complex :: show(complex c)
{
cout<<c.x<<"+j"<<c.y<<"\n";
}
void main()
{
complex a,b,c;
clrscr();
a.input(3.1, 5.65);
b.input(2.75, 1.2);
c=sum(a,b);                          //c+a+b
cout<<"a= "; a.show(a);
cout<<"b= "; b.show(b);
cout<<"c= "; c.show(c);
getch();
}
```

Q.3. (*a*) What is function overloading? Write overloaded functions for computing area of a circle a square and a rectangle. Also discuss ambiguity in these functions. **(5)**

Ans. Function Overloading in C++

You can have multiple definitions for the same function name in the same scope. The definition of the function must differ from each other by the types and/or the number of arguments in the argument list. You cannot overload function declarations that differ only by return type.

Program to calculate the area of circle, square and triangle :

```
#include<iostream.h>
#include<conio.h>
const float pi=3.14;
float area (float n, float b, float h)
{
float ar;
ar=n*b*h;
return ar;
}
float area (float r)
{
float ar;
ar=pi*r*r;
```



```
return ar;
}
float area (float s)
{
float ar;
ar=s*s;
return ar;
}
void main()
{
float b,h,r,l;
float result;
clrscr();.
cout<<"\nEnter the Base & Height of Triangle;\n";
cin>>b>>h;
result=area(0.5,b,h);
cout<<"\nArea of Triangle: "<<result<<endl;
cout<<"\nEnter the Radius of Circle: \n";
cin>>r;
result=area(r);
cout<<"\nArea of Circle: "<<result<<endl;
cout<<"\nEnter the Side of Square: \n";
cin>>s;
result=area(s);
cout<<"\nArea of Square: "<<result<<endl;
getch();
}
```

Q.3. (*b*) Discuss the memory requirement, data members, member functions, static and non-static data members. **(5)**

Ans. During runtime, however... in C++, classes define types but (unless you activate RTTI which allows limited introspection into classes) don't generally occupy any memory themselves 1—they're just the frameworks for the construction and destruction of objects. Their *methods*, however—the constructors, destructors, instance methods, and class methods, occupy some portion of executable memory, but compilers can and do optimize away any such methods that are not used in the program.

Instances of types (that is, objects as well as primitives like int variables) occupy the bulk of memory in C++, but for their member functions they refer back to their classes. Exactly how much memory an instance of a particular class uses is entirely and utterly an implementation detail, and you should generally not need to care about it.

The C++ standard doesn't explicitly state when static memory is allocated, as long as it is available on first use. That said, it is most likely allocated during program initialization, thus guaranteeing its presence as soon as it is required, without needing special-case code to detect and perform allocation on access.

Q.4. Write short notes on the following:

Q.4. (a) Copy Constructor (2)

Ans. The **copy constructor** is a constructor which creates an object by initializing it with an object of the same class, which has been created previously. The copy constructor is used to

- Initialize one object from another of the same type.
- Copy an object to pass it as an argument to a function.
- Copy an object to return it from a function.

If a copy constructor is not defined in a class, the compiler itself defines one. If the class has pointer variables and has some dynamic memory allocations, then it is a must to have a copy constructor. The most common form of copy constructor is shown here "

```
classname (const classname &obj) {
  // body of constructor
}
```

Here, obj is a reference to an object that is being used to initialize another object.

Q.4. (b) Static Members. (2)

Ans. We can define class members static using **static** keyword. When we declare a member of a class as static it means no matter how many objects of the class are created, there is only one copy of the static member.

A static member is shared by all objects of the class. All static data is initialized to zero when the first object is created, if no other initialization is present. We can't put it in the class definition but it can be initialized outside the class as done in the following example by redeclaring the static variable, using the scope resolution operator :: to identify which class it belongs to.

Q.4. (c) Inline Functions. (2)

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

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

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

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

Q.4. (d) Data Hiding. (2)

Ans. Encapsulation is an Object Oriented Programming concept that binds together the data and functions that manipulate the data, and that keeps both safe from outside interference and misuse. Data encapsulation led to the important OOP concept of **data hiding**.

Data encapsulation is a mechanism of bundling the data, and the functions that use them and **data abstraction** is a mechanism of exposing only the interfaces and hiding the implementation details from the user.

C++ supports the properties of encapsulation and data hiding through the creation of user-defined types, called **classes**. We already have studied that a class can contain **private, protected** and **public** members. By default, all items defined in a class are private.

Q.4. (e) Scope Resolution Operator. (2)

Ans. Scope resolution operator in C++

✓ In C++ language the scope resolution operator is written ":".

✓ C++ supports to the global variable from a function, Local variable is to define the same function name.

✓ Identify variables with use of scope resolution operator when we use the same name for local variable and global variable (or in class or namespace)

✓ Resolution operator is placed between the front of the variable name then the global variable is affected. If no resolution operator is placed between the local variable is affected.

END TERM EXAMINATION [MAY-JUNE 2017]
FOURTH SEMESTER [B.TECH.]
OBJECT ORITENTED PROGRAMMING USING C++ [ETCS–210]

Time : 3 hrs. **M.M. : 75**

Note: *Attempt any five questions including Q.No. 1 which is compulsory.*

Q.1. Attempt the following:

Q.1. (a) Differentiate between pointer and reference variables. **(2.5)**

Ans. Difference Between Pointer & Reference Variable :

A pointer is a variable which stores the address of another variable.

A reference is a variable which refers to another variable.

For example :-

```
int i = 3;
int *ptr = &i;
int &ref = i;
```

The first line simply defines a variable. The second defines a pointer to that variable's memory address. The third defines a reference to the first variable.

Not only are the operators different, but you use the differently as well. With pointers must use the * operator to dereference it. With a reference no operator is required. It is understood that you are intending to work with the referred variable.

The following two lines will both change the value of i to 13.

```
*ptr = 13;
ref = 13;
```

Q.1. (b) What are empty classes? Can instances of empty class be created? **(2.5)**

Ans. Refer to Q.1. (c) First Term Examination 2017.

Q.1. (c) Differentiate between default and parameterized constructors. **(2.5)**

Ans. Default Constructor :

A constructor that has no parameter is called the default constructor. The default constructor for **class A is A::A(),** If no such constructor is defined, then the compiler supplies a default constructor.Therefore a statement such as

```
A a;
```

Invokes the default constructor of the compiler to create the object a.

Parameterized constructor :

The constructor that can take arguments are called the parameterized constructors. In the parameterized constructors we passed the arguments when the object of class are created.The constructor integer() may be modified to take arguments as shown below:

```
class integer
{
int m,n;
public:
integer(int x,int y)                    //parameterized constructor
{
m=x;
n=y;
}
};
main()
{
integer int(100, 200);                  //invoke the parameterized constructor
}
```


Q.1. (d) What is Garbage Collection in C++. **(2.5)**

Ans. Garbage collection is often portrayed as the opposite of manual memory management, which requires the programmer to specify which objects to deallocate and return to the memory system. However, many systems use a combination of approaches, including other techniques such as stack allocation and region inference. Like other memory management techniques, garbage collection may take a significant proportion of total processing time in a program and can thus have significant influence on performance.

In computer science, **garbage collection** (GC) is a form of automatic memory management. The *garbage collector*, or just *collector*, attempts to reclaim *garbage*, or memory occupied by objects that are no longer in use by the program. Garbage collection was invented by John McCarthy around 1959 to solve problems in Lisp.

Resources other than memory, such as network sockets, database handles, user interaction windows, and file and device descriptors, are not typically handled by garbage collection. Methods used to manage such resources, particularly destructors, may suffice to manage memory as well, leaving no need for GC. Some GC systems allow such other resources to be associated with a region of memory that, when collected, causes the other resource to be reclaimed; this is called *finalization*. Finalization may introduce complications limiting its usability, such as intolerable latency between disuse and reclaim of especially limited resources, or a lack of control over which thread performs the work of reclaiming.

Q.1. (e) Why are virtual functions used? **(2.5)**

Ans. Use of Virtual Function:

Virtual functions allow us to create a list of base class pointers and call methods of any of the derived classes without even knowing kind of derived class object. For example, consider a employee management software for an organization, let the code has a simple base class *Employee* , the class contains virtual functions like *raise Salary(), transfer(), promote()* etc. Different types of employees like *Manager, Engineer,* ..etc may have their own implementations of the virtual functions present in base class *Employee*. In our complete software, we just need to pass a list of employees everywhere and call appropriate functions without even knowing the type of employee. For example, we can easily raise salary of all employees by iterating through list of employees. Every type of employee may have its own logic in its class, we don't need to worry because if *raiseSalary()* is present for a specific employee type, only that function would be called.

Q.1. (f) What is containership? Explain with an example. **(2.5)**

Ans. When a class contains objects of another class or its members, this kind of relationship is called containership or nesting and the class which contains objects of another class as its members is called as container class.

Syntax for the declaration of another class is:

```
Class class_name1
{
____
____
};

Class class_name2
{
____
____
};

Class class_name3
{
Class_name1 obj1;          // object of class_name1
Class_name2 obj2;          // object of class_name2
_____
_____
};
```

Q.1. (g) Define static objects with example. **(2.5)**

Ans. Static Object :

Static keyword works in the same way for class objects too. Objects declared static are allocated storage in static storage area, and have scope till the end of program.

Static objects are also initialized using constructors like other normal objects. Assignment to zero, on using static keyword is only for primitive datatypes, not for user defined datatypes.

```
class Abc
{
 int i;
 public:
 Abc()
 {
  i=0;
  cout << "constructor";
 }
 ~Abc()
 {
  cout << "destructor";
 }
};

void f()
{
 static Abc obj;
}
```



```
int main()
{
 int x=0;
 if(x==0)
 {
  f();
 }
 cout << "END";
}
```

Output :

constructor END destructor

You must be thinking, why was destructor not called upon the end of the scope of if condition. This is because object was static, which has scope till the program lifetime, hence destructor for this object was called when main() exits.

Q.1. (h) How constructors and destructors are executed in multilevel inheritance. **(2.5)**

Ans. C++ constructor call order will be from top to down that is from base class to derived class and c++ destructor call order will be in reverse order.

Below is the example of constructor destructor call order for multi-level inheritance, in which Device class is the base class and Mobile class is derived from Device base class and then Android class is derived from Mobile class as a base class.

When we create the object of Android class, order of invocation of constructor destructor will be as below

Constructor : Device
Constructor : Mobile
Constructor : Android
Constructor : Android
Constructor : Mobile
Constructor : Device

Q.1. (i) Define Reusability, how C++ supports Reusability? **(2.5)**

Ans. Reusability :

C++ strongly supports the concept of reusability. The C++ classes can be reused in several ways. Once a class has been written and tested, it can be adapted by another programmer to suit their requirements. This is basically done by creating new classes, reusing the properties of the existing ones. 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 the new one is called the derived class or subclass. A derived class includes all features of the generic base class and then adds qualities specific to the derived class.

Q.1. (j) Differentiate function overloading and function overriding. **(2.5)**

Ans. Overloading is defining functions that have similar signatures, yet have different parameters.

Overriding is only pertinent to derived classes, where the parent class has defined a method and the derived class wishes to override that function.

Overriding	Overloading
Functions name and signatures must be same	Having same Function name with different **Signatures**.
Overriding is the concept of **runtime polymorphism**	Overloading is the concept of **compile time polymorphism**
When a function of base class is re-defined in the derived class called as Overriding	Two functions having same name and return type but with different type and/or number of arguments is called as Overloading
It needs inheritance.	It doesn't need inheritance.
Functions should have same data type.	Functions can have different data types
Function should be public.	Function can be different access specifies

Q.2. (a) Explain the characteristic of Object-oriented language, with appropriate examples. **(8)**

Ans. Characteristics of OOPs:

Class

Here we can take **Human Being** as a class. A class is a blueprint for any functional entity which defines its properties and its functions. Like Human Being, having body parts, and performing various actions.

Inheritance

Considering HumanBeing a class, which has properties like hands, legs, eyes etc, and functions like walk, talk, eat, see etc. Male and Female are also classes, but most of the properties and functions are included in HumanBeing, hence they can inherit everything from class HumanBeing using the concept of **Inheritance**.

Objects

My name is Abhishek, and I am an **instance/object** of class Male. When we say, Human Being, Male or Female, we just mean a kind, you, your friend, me we are the forms of these classes. We have a physical existence while a class is just a logical definition. We are the objects.

Abstraction

Abstraction means, showcasing only the required things to the outside world while hiding the details. Continuing our example, **Human Being's** can talk, walk, hear, eat, but the details are hidden from the outside world. We can take our skin as the Abstraction factor in our case, hiding the inside mechanism.

Encapsulation

This concept is a little tricky to explain with our example. Our Legs are binded to help us walk. Our hands, help us hold things. This binding of the properties to functions is called Encapsulation.

Polymorphism

Polymorphism is a concept, which allows us to redefine the way something works, by either changing how it is done or by changing the parts using which it is done. Both the ways have different terms for them.

If we walk using our hands, and not legs, here we will change the parts used to perform something. Hence this is called **Overloading**.

And if there is a defined way of walking, but I wish to walk differently, but using my legs, like everyone else. Then I can walk like I want, this will be called as **Overriding**.

Q.2. (b) Explain the use of copy constructor with example program. **(4.5)**

Ans. A The copy constructor is a special kind of constructor which creates a new object which is a copy of an existing one, and does it efficiently. The copy constructor receives an object of its own class as an argument, and allows to create a new object which is copy of another without building it. The copy constructor receives an object of its own class as an argument, and allows to create a new object which is copy of another without building it from scratch. A copy constructor has the following general function prototype:

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

Example of copy constructor.

```
#include<iostream>
class Point
{
private:
  int x, y;
public:
  Point(int x1, int y1) { x = x1; y = y1; }

  // Copy constructor
  Point(const Point &p2) {x = p2.x; y = p2.y; }
  int getX()      { return x; }
  int getY()      { return y; }
};
int main()
{
  Point p1(10, 15); // Normal constructor is called here
  Point p2 = p1; // Copy constructor is called here
  cout << "p1.x = " << p1.getX() << ", p1.y = " << p1.getY();
  cout << "\np2.x = " << p2.getX() << ", p2.y = " << p2.getY();
  return 0;
}
```

Q.3. (a) Write a program to show the use of friend function and friend class. **(7.5)**

Ans. Friend Class A friend class can access private and protected members of other class in which it is declared as friend. It is sometimes useful to allow a particular class to access private members of other class. For example a LinkedList class may be allowed to access private members of Node.

```
class Node
{
private:
int key;
Node *next;
/* Other members of Node Class*/
friend class LinkedList; //Now class LinkedList can
// access pri.ate members of Node
};
```

Friend Function Like friend class, a friend function can be given special grant to access private and protected members. A friend function can be:

```
public
void showB(B&);
};
class B
private.
int b;
public
B() {b = 0;}
friend void A:: shows B(B& x); // Friend Function
};
void A:: shows B(B &x)
{
//Since show() is friend of B, it can
// access private members of B
std:: cout <<"B::b ="<<x. b;
}
int main()
{
A a;
B x;
a. shows B(x);
return 0;
}
```

Run on IDE

Output:

B::b = 0

Q.3. (b) What are Destructors? Write a program to show the order in which local objects are destructed. **(5)**

Ans. The Class Destructor

A destructor is a special member function of a class that is executed whenever an object of it's class goes out of scope or whenever the delete expression is applied to a pointer to the object of that class.

A destructor will have exact same name as the class perfixed with a tilde (~) and it can neither return a value nor can it take any parameters. Destructor can be very useful for releasing before coming out of the program like closing files, releasing memories etc.

Example :

```
class A
{
public:
~A(); // This is the destructor
};
```

Order in which objects are Destructed:

C++ constructor call order will be from top to down that is from base class to derived class and c++ destructor call order will be in resverse order.

Below is the example of constructor destructor call order for multi-level inheritance, in which Device class is the base class and Mobile class is derived from Deivice base class and then Android class is derived from Mobile class as a base class.

```
#include
```



```
using namespace std;
//base class
class Device{
public:
Device(){cout<<"Constructor: Device\n";}
~Device (){cout<<" Destructor : Device\n";}
};
//derived class
class Mobile: public Device{
public
Mobile(){cout<<"Constructor: Mobile\n";}
~Mobile(){cout<<"Destructor : Mobile\n";}
};
//derived class
class Android: public Mobile {
public:
Android(){cout<<"Constructor : Android\n";}
~Android(){cout<<"Destructor : Android\n";}
};
//----------TEST----------
int main ()
{
Android _android;
// create the object that will call required constructors
return 0;
}
```

When we create the object of Android class, oder of invocation of constructor and destructor will be as below:

Constructor : Device
Constructor : Mobile
Constructor : Android
Destructor : Android
Destructor : Mobile
Destructor ; Device

Q.4. (a) Create a class, which keeps track of the number of its instances, use static data member, constructors and destructors to maintain updated information about active objects. **(7.5)**

Ans.

```
#include<iostream.h>
#include<conio.h>
class man
{
static int no;
char name;
int age;
public:
man()
{
no++;
cout<<"\ n Number of objects exists:"<<no;
}
```

```
- man ()
{
--no;
cout<<"\ n Number of objects exists:"<<no;
}
};
int man :: no = 0;
void main ()
{
clrscr ();
man A, B, C;
cout <<"\n Press any key to destroy object";
getch ();
}
```

OUTPUT

Number of Objects exists: 1
Number of Objects exists: 2
Number of Objects exists: 3
Press any key to destory object
Number of objects exists: 2
Number of objects exists: 1
Number of objects exists: 0

Explanation : In this program, the class man has one static data member no. The static data member is initialized to zero. Only one copy of static data member is created and all objects share the same copy of static data memory.

In function main(), objects A, B, and C are declared. When objects are declared, constructor is executed and static data member no is increased with one. The constructor also displays the value of no on the screen. The value of static member shows us the number of objects present. When the user presses a key, destructor is executed, which destroys the object. The value of static variable shows the number of existing objects.

Q.4. (b) How to achieve dynamic memory allocation in C++? Explain with a program. (5)

Ans. Dynamic memory allocation:

Dynamic memory allocation in C/C++ refers to performing memory allocation manually by programmer. Dynamically allocated memory is allocated on **Heap** and non-static and local variables get memory allocated on **Stack**.

new operator:

The new operator denotes a request for memory allocation on the Heap. If sufficient memory is available, new operator initializes the memory and returns the address of the newly allocated and initialized memory to the pointer variable.

Syntax to use new operator: To allocate memory of any data type, the syntax is:

- pointer-variable = **new** data-type;

Here, pointer-variable is the pointer of type data-type. Data-type could be any built-in data type including array or any user defined data types including structure and class.

Example

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

    void main()
    {
```



```
int size,i;
int *ptr;

cout<<"\n\tEnter size of Array : ";
cin>>size;

ptr = new int[size];
//Creating memory at run-time and return first byte of address to ptr.

for(i=0;i<5;i++)      //Input arrray from user.
{
      cout<<"\nEnter any number : ";
      cin>>ptr[i];
}

for(i=0;i<5;i++)      //Output arrray to console.
cout<<ptr[i]<<", ";

delete[] ptr;
//deallocating all the memory created by new operator
}
```

Output :

```
Enter size of Array : 5

Enter any number : 78
Enter any number : 45
Enter any number : 12
Enter any number : 89
Enter any number : 56

78, 45, 12, 89, 56,
```

Q.5. (a) How base class member functions can be involved in a derived class if the derived class also has member function with the same name? Explain with example. (8.5)

Ans. A virtual function is a member function that you expect to be redefined in derived classes. When you refer to a derived class object using a pointer or a reference to the base class, you can call a virtual function for that object and execute the derived class's version of the function.

Virtual functions ensure that the correct function is called for an object, regardless of the expression used to make the function call.

Suppose a base class contains a function declared as virtual and a derived class defines the same function. The function from the derived class is invoked for objects of the derived class, even if it is called using a pointer or reference to the base class. The following example shows a base class that provides an implementation of the PrintBalance function and two derived classes

```
// deriv_VirtualFunctions.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;
```

```
class Account {
public:
   Account( double d) {_balance = d;}
   virtual double GetBalance( ) {return _balance;}
   virtual void PrintBalance( ) {cerr << "Error. Balance not available for base type."
<<endl;}
private:
   }; double _balance;
class CheckingAccount : public Account
{ public:
   CheckingAccount(double d) : Account(d) {}
   void PrintBalance( ) { cout << "Checking account balance:" << GetBalance ( ) <<
endl;}
};
class SavingsAccount : public Account {
public:
   SavingsAccount( double d) : Account(d) {}
   void PrintBalance( ) { cout << "Savings account balance: " << GetBalance( ); }
};
int main() {
   // Create objects of type CheckingAccount and SavingsAccount.
   CheckingAccount *pChecking = new CheckingAccount( 100.00 );
   SavingsAccount *pSavings = new SavingsAccount( 1000.00 );
   // Call PrintBalance using a pointer to Account.
   Account *pAccount = pChecking;
   pAccount->PrintBalance();
   // Call PrintBalance using a pointer to Account.
   pAccount = pSavings;
   pAccount->PrintBalance();
}
```

Q.5. (b) Differentiate public, protected and private access specifiers. **(4)**

Ans. Access specifiers defines the access rights for the statements or functions that follows it until another access specifier or till the end of a class. The three types of access specifiers are "private", "public", "protected".

private:

The members declared as "private" can be accessed only within the same class and not from outside the class.

```
class PrivateAccess
{
 private:  // private access specifier
 int x;        // Data Member Declaration
 void display();  // Member Function decaration
}
```


public:

The members declared as "public" are accessible within the class as well as from outside the class.

```
class PublicAccess
{
 public:  // public access specifier
 int x;        // Data Member Declaration
 void display();  // Member Function decaration
}
```

protected:

The members declared as "protected" cannot be accessed from outside the class, but can be accessed from a derived class. This is used when inheritance is applied to the members of a class.

```
class ProtectedAccess
{
 protected:  // protected access specifier
 int x;        // Data Member Declaration
 void display();  // Member Function decaration
}
```

Q.6. (a) What is generic programming? Write its advantages? **(5)**

Ans. Generic Programming:

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

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

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

You can use templates to define functions as well as classes.

Advantages of generic programming :

- Templates are easier to write. You create only one generic version of your class or function instead of manually creating specializations.
- Templates can be easier to understand, since they can provide a straightforward way of abstracting type information.
- Templates are typesafe. Because the types that templates act upon are known at compile time, the compiler can perform type checking before errors occur.
- You can also use class templates to develop a set of typesafe classes.

Q.6. (b) What is the difference between C & C++. Show and explain the usage of new and delete keyword. **(7.5)**

Ans. Difference between C and C++.

C	C++
1. C is Procedural Language.	1. C++ is non Procedural i.e Object oriented Language.
2. No virtual Functions are present in C.	2. The concept of virtual Functions are used in C++.
3. In C, Polymorphism is not possible.	3. The concept of polymorphism is used Important Feature of OOPS.
4. Operator overloading is not possible in C.	4. Operator overloading is one of the greatest Feature of C++.
5. Top down approach is used in Program Design.	5. Bottom up approach adopted in Program Design.
6. No namespace Feature is present in C Language.	6. Namespace Feature is present in C++ for avoiding Name collision.
7. Multiple Declaration of global variables are allowed.	7. Multiple Declaration of global varioables are not allowed.
8. In C • scanf() Function used for Input. • printf() Function used for output.	8. In C++ • Cin>> Function used for Input. • Cout<< Function used for output.
9. Mapping between Data and Function is difficult and complicated.	9. Mapping between Data and Function can be used using "Objects"
10. In C, we can call main() Function through other functions	10. In C++, we cannot call main() Function through other functions.
11. C requires all the variables to be defined at the starting of a scope.	11. C++ allows the declaration of variable anywhere in the scope i.e at time of its First use.
12. No inheritance is possible in C.	12. Inheritance is possible in C++
13. In C, malloc() and calloc() Functions are used for Memory Allocation and free() function for memory Deallocating.	13. In C++, new and delete operators are used for Memory Allocating and Deallocating.
14. It supports built-in and primitive data types.	14. It support both built-in and user define data types.
15. In C, Exception Handling is not present	15. In C++, Exception Handling is done with Try and Catch block.

New & delete Keywords :

new and delete operators are provided by C++ for runtime memory management. They are used for dynamic allocation and freeing of memory while a program is running.

The new operator allocates memory and returns a pointer to the start of it. The delete operator frees memory previously allocated using new.

Syntax of new is:

p_var = new type name;

where p_var is a previously declared pointer of type typename. typename can be any basic data type.

new can also create an array:

p_var = new type [size];

In this case, size specifies the length of one-dimensional array to create.

Example 1:

int *p; p=new int;

It allocates memory space for an integer variable. And allocated memory can be released using following statement,

delete p;

Example 2:

int *a; a = new int[100];

It creates a memory space for an array of 100 integers. And to release the memory following statement can be used,

delete []a;

Q.7. (a) What are Abstract classes? **(3)**

Ans. Abstract Class

An abstract class is a class that is designed to be specifically used as a base class. An abstract class contains at least one pure virtual function. You declare a pure virtual function by using a pure specifier (= 0) in the declaration of a virtual member function in the class declaration.

The following is an example of an abstract class:

```
class AB {
public:
  virtual void f() = 0;
};
```

Q.7. (b) 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 objects and process them. **(9.5)**

Ans.

```
# include <isotream>
using namespace std;
class student
{
   protected:
      int roll-number;
   public:
      void get-number (int a)
      roll-number = a;
 }
 void put-number (void)
 {
 cout << "Roll. No.:" <<roll-number <<"\n";
 }
};
 class engineering: virtual public student
 {
    proteced:
       float part 1, part 2;
    public:
       void get_marks (float x, float y)
```

```
    {
      part 1 = x;  part 2 = y;
    }
    void put_marks (void)
    {
      cout << "Marks obtained." << "\n";
           <<"part 1 = "<<part 1<<"\n"
           <<"part 2 = "<<part 2<<"\n"
    }
};
class medical : virtual public student
{
  protected:
    float part 1, part 2;
public:
    void get_mark (Heat x, float y)
  {
    part 1 = x; part 2 = y;
  }
  void put-marks (void)
  cout <<"marks obtained :" << "\n"
    <<"part 1 =" << part 1 << "\n"
    <<"part 2 = << part << "\n"
```

Q.8. (a) What are exceptions? How reliability is affected by exception handling? **(4)**

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

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

- **throw:** A program throws an exception when a problem shows up. This is done using a **throw** keyword.
- **catch:** A program catches an exception with an exception handler at the place in a program where you want to handle the problem. The **catch** keyword indicates the catching of an exception.
- **try:** A **try** block identifies a block of code for which particular exceptions will be activated. It's followed by one or more catch blocks.

Q.8. (b) Write an interactive 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 my_sqrt() should raise an exception. **(8.5)**

Ans.

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

double SqrtNumber(double num)
{
    double lower_bound=0;
```



```
  double temp=0;          /* ek edited this line */

  int nCount = 50;

while(nCount != 0)
{
  temp=(lower_bound+upper_bound)/2;
  if(temp*temp==num)
  {
      return temp;
  }
  else if(temp*temp > num)

  {
      upper_bound = temp;
  }
  else
  {
      lower_bound = temp;
  }
  nCount --
}
  return temp;
}

int main()
{
double num;
cout<<"Enter the number\n";
cin>>num;

if(num < 0)
{
cout<<"Error: Negative number!";
return 0;
}

cout<<"Square roots are: +"<<sqrtnum(num) and <<" and -"<<sqrtnum(num);
return 0;
}
```