Marathwada Institute of Technology
Faculty of Engineering
FY MCA Examination
May/June, 2013
Object Oriented Programming using C++

Time: Three Hours Max. Marks: 80

N.B.: (i) Question Nos. 1 and 8 are compulsory
(ii) Solve three questions from each section

Section A

1. Find the output and give reason: 4x2=8

a)
```
#include<iostream.h>
void stat()
{
        int m = 0;
         static int n = 0;
         m++;
         n++;
         cout<<m<<n;
 }
 void main()
 {
          stat();
          stat();
  }
```

Output:
1 1
1 2

In first call stat()
m++ becomes 1 (non-static)
n++ becomes 1 (static)
In second call stat()
m++ becomes 1 (non-static variables initialized to zero for each call ie does not persist previous value)
n++ becomes 2 (static variables initialized only once in next call it continues with previous value means persist values)

b)
```
#include<iostream.h>
int main()
{
        int x, y = 10, z = 10;
        x=(y==z)
        cout<<x;
        return 0;
}
```

Output:
1

First y == z is evaluated ie 10 == 10 returns true is 1
And then 1 is assigned to x therefore x = 1

2 a) Explain enumerated data types and derived data types with example. 8

Enumerated types (enum)

- enum(enumeration) is a user-defined type consisting of a set of enumerators(enumerator --- named integer constant)
- Objects of these enumerated types can take any of these enumerators as value.
- Syntax:
  enum type_name {value1,value2,value3 …… valuen} object_names;
- This creates the type `type_name`, which can take any of value1, value2, value3, ... as value.
- Objects (variables) of this type can directly be instantiated as object_names.
- For ex.:
  enum colors {red,blue,green};

```
#include <iostream.h>
int main()
{
        enum Fruits{orange, guava, apple};
        Fruits myFruit;
        int i;
        cout << "Please enter the fruit of your choice(0 to 2)::";
        cin >> i;
        switch(i)
        {
                case orange:  cout << "Your fruit is orange";
                              break;
                case guava:   cout << "Your fruit is guava";
                              break;
                case apple:   cout << "Your fruit is apple";
                              break;
        }
        return 0;
 }
```

Derived data types

- Derived types are four types
  - Array
  - Function
  - Pointer
  - reference

Array

- An array is a group of related data items that share a common name.
- Array is nothing more than easy way to keep track of list. The individual data items contained in an array are called the elements of an array.
- Every data element of an array must be of the same data type, only one name must be assign to an entire array and individual elements are referenced by specifying a subscript.
- A subscript is also called index. In C subscript start at zero (0) rather than one

(1), and cannot be negative.

- The single group name and subscript are associated by enclosing the subscript in square brackets to the right of the name.
- Types of Arrays
  - One Dimensional Array
  - Two Dimensional Array
  - Multidimensional Array
- For ex.
  - int a[5];
  - float per[10];
  - char name[10];

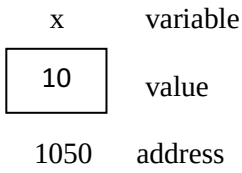
Function

- Functions are nothing but sub programs.
- There are two types of function
  - Library function
  - User Defined Function (UDF)
- Library functions are functions which are included in the C program library, and are not required to be written by the user in the main program whenever a job is to be repeatedly performed.
- The user defined functions can also be placed in the C program library and then can be called library functions.
- Examples of library functions are clrscr(), strlen() etc..
- Example of user defined function is main().

Pointer

- Computers use their memory for storing the instructions of a program, as well as the values of the variables that are associated with it.
- The computer memory is a sequential collection of storage cells.
- Each cell is known as a byte, has a number called address.
- The address are numbered starting from zero (0) to the last address depends on memory size (i.e. 0 to 65535).
- Consider the internal structure of a variable declaration.
- For ex.

```
int x = 10;
```



- Since memory addresses are simply numbers they can be assigned to some variables which can be stored in memory like any other variable, such variable that hold memory address are called pointers.
- A pointer is therefore nothing but a variable that contains an address which is a location of another variable in memory.
- For ex.:

```
int *p = &x;
```

Reference

- It is an alternative name for an object.
- A reference variable provides an alias for a previously defined variable.
- It's declaration consists of a base type, an &(ampersand), a reference variable name equated to a variable name.
- the general form of declaring is:

  type &ref-var = var-name;

b) Create a class time hat has separate data members for hrs, mins and secs. One member function should initialize it to fixed values, another member function should display it as HH : MM : SS format. The final member function should add two objects of type time passed as argument. 8

```
//Program to add two objects of type Time passed as argument
# include<iostream.h>
#include<conio.h>
class Time
{
        int hh;
        int mm;
        int ss;

        public:
                void gettime();
                void puttime();
                void addtime(Time,Time) ;
};
void Time::gettime()
{
        cout<<"\n\tEnter the hours, minutes and seconds:\n";
        cin>>hh>>mm>>ss;
}
void Time::puttime()
{
        cout<<"\n\t\t"<<hh<<" : "<< mm<< " : "<<ss<<endl;
}
void Time::addtime(Time t1,Time t2)
{
        ss=t1.ss+t2.ss;
        mm=ss/60;
        ss=ss%60;
        mm=t1.mm+t2.mm+mm;
        hh=mm/60;
        mm=mm%60;
        hh=t1.hh+t2.hh+hh;
}
void main()
{
        Time tm1,tm2,tm3;
        clrscr();
        cout<<"\n\t\tFIRST  TIME\n";
        tm1.gettime();
        cout<<"\n\t\tSECOND  TIME\n";
        tm2.gettime();
```

```
        tm3.addtime(tm1,tm2);
        tm1.puttime();
        tm2.puttime();
        tm3.puttime();
}
```

Output:

```
        FIRST  TIME
Enter the hours, minutes and seconds:
1 30 45

        SECOND  TIME
Enter the hours, minutes and seconds:
2 20 30

        1 : 30 : 45
        2 : 20 : 30
        3 : 51 : 15
```

3 a) What is friend function? What are merits and demerits of using friend function. 8

Friend function

- Any data which is declared **private** inside a class is not accessible from outside the class.
- A function which is not a member or an external
- class can never access such private data. But there may be some cases, where a programmer will need access to the private data from non-member functions and external classes.
- C++ offers some exceptions in such cases.
- A class can allow non-member functions and other classes to access its own private data, by making them as **friends**.
- Once a non-member function is declared as a friend, it can access the private data of the class
- similarly when a class is declared as a friend, the friend class can have access to the private data of the class which made this a friend
- A non member function of the class
- Authorized by the class to have access even to the private and protected data members
- Declare the function as a friend as shown below:

```
class Sample
{
        -----
        public:
                friend void display(Sample);  //declaration
};
```

```
void display(Sample s)                    //definition
{
                ------
}
```

Merits:

- Not in the scope of the class to which it has been declared as friend
- Cannot be called using the object of that class
- Can be invoked like a normal function without using object
- Can able to access the other class members in our class if, friend keyword is used.
- Can access the members without inheriting the class.
- Unlike member functions, it cannot access the member names directly and has to use an object name and dot membership operator with each member name (Ex. s.n)
- Can be declared either in the public or private section of a class without affecting its meaning
- Usually, it has the object as arguments

Demerits:

- Maximum size of the memory will be occupied by objects according to the size of friend Members.
- Cannot do any run time polymorphism concepts in those members.

b) Write C++ code to add following series using constructor: 8
1 + 3 + 5 + …….. + n.

```
//Program to calculate sum of series 1 + 3 + 5 + …… + n using constructor
#include<iostream.h>
#include<conio.h>

class Sample
{
        int s;
        int n;
        public:
                Sample()
                {
                        s = 0;
                        n = 10;
                }
                void cal()
                {
                        for(int i=1;i<=n;i+=2)
                                s = s + i;
                }
                void print()
                {
```

```
                cout<<"Sum is : "<<s;
        }
        ~Sample(){}
};
void main()
{
    clrscr();
    Sample ob;
    ob.cal();
    ob.print();
}
```

Output:
Sum is: 25

4 a) Explain default arguments with example 8

- A function call without specifying all its arguments
- Default values are specified when the function is declared
- To establish a default value the function prototype or the function definition must be used
- Only the trailing arguments can have default values i.e. add defaults from right to left
- For ex.:
  void sum(int a, int b=10); //valid
  void sum(int a=10, int b); //invalid
  void sum(int a=10, int b, int c=5); //invalid
- If a value for that parameter is not passed when the function is called, the default value is used
- If a value is specified this default value is ignored and the passed value is used instead
- For ex.: Program1

```
//Demo of Default Function Arguments
#include<iostream.h>
#include<conio.h>

void repchar(char='*',int=10);

void main()
{
    clrscr();
    repchar();
    repchar('=');
    repchar('+',30);
    getch();
}
void repchar(char ch,int n)
{
    for(int i=1;i<=n;i++)
```

```
                cout<<ch;
        cout<<endl;
}
```

Output:

```
**********
==========
++++++++++++++++++++++++++++++
```

- For ex.: Program2

```
// default values in functions
#include <iostream.h>

int divide (int a, int b=2)
{
        int result;
        result=a/b;
        return (result);
}

void main ()
{
        cout << divide (12);
        cout << endl;
        cout << divide (20,4);
}
```

Output:

```
6
5
```

b) Write a C++ code to read and display matrix using function overloading 8

```
//To read and display matrix using function overloading
#include <iostream.h>
#include <iomanip.h>
#include <conio.h>
void main()
{
        const int row = 10;
        const int col = 10;
        void matrix(int rows,int cols=3);
        clrscr();

        cout<<"\n\tEnter Number of rows: ";
        int rows;
        cin>>rows;

        matrix(rows);

        cout<<"\n\tEnter number of rows and cols: ";
```

```
        int cols;
        cin>>rows>>cols;

        matrix(rows,cols);

        getch();
}


void matrix(int r, int c)
{
                clrscr();
                int mat[row][col];
                cout<<"\n\tEnter Your Data for Matrix "<<r<<"x"<<c<<"\n";

                //Reading Matrix
                for(int i=0;i<r;i++)
                {
                   for(int j=0;j<c;j++)
                  {
                     cin>>mat[i][j];
                  }
                }

                cout<<endl<<endl;

                //Displaying matrix
                cout<<"\n";
                 for(int i=0;i<r;i++)
                {
                        for(int j=0;j<c;j++)
                          {
                             cout<<setw(5)<<mat[i][j];
                          }
                        cout<<endl;
                }
}
```

Output:

Enter Number of rows: 3
Enter Your Data for Matrix 3x3
1 2 3
4 5 6
7 8 9

1 2 3
4 5 6
7 8 9

Enter number of rows and cols: 2 2
Enter Your Data for Matrix 2x2
1 2
3 4

1 2
3 4

Section B

5 a) Write C++ code to demonstrate multiple exceptions 8

```
//Program to demonstrate Multiple Exception
#include <iostream>
using namespace std;

int main()
{
        double op1, op2, result;
        char operator;

        cout << "First Number:  ";
        cin >> op1;
        cout << "Operator:      ";
        cin >> operator;
        cout << "Second Number: ";
        cin >> op2;

        try {
                // Make sure the user typed a valid operator
                if(operator != '+' && operator != '-' &&
                  operator != '*' && operator != '/')
                        throw operator;

                // Find out if the denominator is 0
                if(operator == '/')
                        if(op2 == 0)
                                throw 0;

                // Perform an operation based on the user's choice
                switch(operator)
                {
                case '+':
                        result = op1 + op;
                        break;

                case '-':
                        result = op1 - op2;
                        break;

                case '*':
                        result = op1 * op2;
                        break;

                case '/':
                        result = op1 / op2;
                        break;
                }

                // Display the result of the operation
                cout << "\n" << op1 << " " << operator << " "
                    << op2 << " = " << result << "\n\n";
        }
        catch(const char n)
```

```
        {
                cout << "\nOperation Error: " << n << " is not a valid operator\n\n";
        }
        catch(const int p)
        {
                cout << "\nBad Operation: Division by " << p << " not allowed\n\n";
        }
        cout<<"END";
        getch();
        return 0;
}
```

Output:
First Number: 10
Operator: >
Second Number: 5

Operation Error: > is not a valid operator

First Number: 10
Operator: /
Second Number: 0

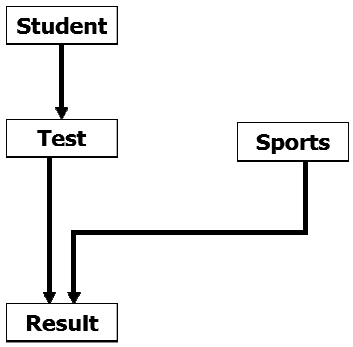
Bad Operation: Division by 0 not allowed

b) Explain how ambiguity occurs in hybrid inheritance and how to resolve it? 8

Hybrid Inheritance

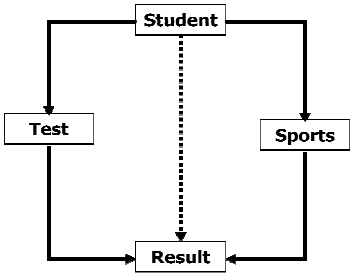
- Applying Two or more types of inheritance together

```
class Student
{ …… };
class Test : public Student
 { …… };
class Sports
{ …… };
class Result : public Test, public  Sports
{ …… };
```



Ambiguity occurs in hybrid inheritance

- Required in scenarios of multiple inheritance where the derivation is like diamond



- The **Result** class has two direct base classes **Test** and **Sports** which themselves have a common base class **Student**
- The **Result** inherits the traits of **Student** via two separate paths
- It can also inherit directly as shown by the broken line

To resolve this ambiguity

- Virtual base class is used also called as virtual inheritance

```
class Student
{
        ……
};
class Test : virtual public Student
{
        ……
};
class Sports : public virtual Student
{
        ……
};
class Result : public Test, public Sports
{
        ……
};
```

- For Ex.:

```
//Virtual Base Class
#include<iostream.h>
#include<conio.h>

class A
{
    public:
            void show_a()
            {
                    cout<<"\nClass A";
            }
```

```
};

class B : public virtual A
{
    Public:
        void show_b()
        {
            cout<<"\nClass B";
        }

};

class C : virtual public A
{
    public:
        void show_c()
        {
            cout<<"\nClass C";
        }

};

class D : public B, public C
{
    public:
        void show_d()
        {
            cout<<"\nClass D";
        }

};

void main()
{
    D ob;
    ob.show_a();
    ob.show_b();
    ob.show_c();
    ob.show_d();
}
```

6 a) Write C++ code to overload comparison operator using friend function 8

```
//Program to overload comparison operator using friend function

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

class Comp
```

```
{
   char *s;

   public:

            void setstring(char *str)
            {
            strcpy(s,str);
            }

            friend void operator==(Comp, Comp);
};

void operator ==(Comp ob, Comp ob1)
{
            if(strcmp(ob.s,ob1.s)==0)
            cout<<"\nStrings are Equal";
            else
            cout<<"\nStrings are not Equal";
}

void main()
{
            Comp ob, ob1;

            clrscr();


            ob.setstring("hello");

            ob1.setstring("hello");

            //Call Equality Operator
            ob==ob1;
            getch();
}
```

Output:
Strings are Equal

b) What is template? Explain class template and function template with example 8

Template

- Templates are the foundation of generic programming, which involves writing code in a way that is independent of any particular type.
- Mechanism to use one function or class to handle many different data types
- Templates supports generic programming
- 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>**.
- To develop reusable s/w components such as functions, classes etc. supporting

different data types in a single framework

- Using templates, single class/function can be designed that operates on data of many types, instead of having to create a separate class/function for each type
- When used with functions they are known as function templates
- Whereas when used with classes they are called class templates

Function Templates (Generic Function)

- A generic function defines a general set of operations that will be applied to various types of data
- A generic function is created using the keyword 'template'
- The general form is as follows:
  template <class Ttype>
  ret-type fun-name(para-list) { }
- Ttype is a placeholder name for a data type used by the function
- For ex.:

```
//function template to find max of 2 (int, float, char)

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

template<class T>
T max(T x, T y)
{
        return (x>y) ? x : y;
}

void main()
{
        int a;
        float b;

        clrscr();

        a = max(15,28);
        b = max(2.1,7.0);

        cout<<a<<endl;
        cout<<b<<endl;
}
```

Class Templates

- A class templates specifies how individual classes can be constructed similar to normal specification
- These classes model a generic class which support similar operations for different data types

- Syntax:

```
template <class T1, class T2 ……..>
class classname
{
};
```

- For ex.:

```
// class templates
#include<iostream.h>
#include<conio.h>

template <class T>
class Max
{
        T a,b;

        public:
                Max(T n1, T n2)
                {
                        a=n1;
                        b=n2;
                }
                T getmax();
};
template <class T>
T Max<T>::getmax ()
{
        return a>b? a : b;
}
void main ()
{
        clrscr();
        Max<int>ob1(100, 75);
        cout<<endl<<ob1.getmax();
        Max<float> ob2(10.4,3.5);
        cout<<endl<<ob2.getmax();
        getch();
}
```

7 a) Write C++ code to swap contents of 2 files 8

```
//Swap the contents of one file to another
#include<iostream.h>
#include<conio.h>
#include<process.h>
#include<fstream.h>
void main()
{
        char s1[15],s2[15],t[15],ch;
        ifstream fin1,fin2;
        ofstream fout;
        clrscr();
        cout<<"Enter name of file1 : ";
```

```
        cin>>s1;
        cout<<"Enter name of file2 : ";
        cin>>s2;

        fin1.open(s1);
        if(!fin1)
        {
                cout<<"File does not exist";
                exit(0);
        }

        fin2.open(s2);
        if(!fin2)
        {
                cout<<"File does not exist";
                exit(0);
        }

        fout.open(t);
        while(!fin1.eof())
        {
                fin1.get(ch);
                fout.put(ch);
        }

        fout.open(s1);
        while(fin2)
        {
                fin2.get(ch);
                fout.put(ch);
        }

        fout.open(s2);
        fin1.open(t);
        while(fin1)
        {
                fin1.get(ch);
                fout.put(ch);
        }
        cout<<"Contents of files swapped....";
        fin1.close();
        fin2.close();
        fout.close();
        getch();

}
```

Output:

Enter name of file1: address
Enter name of file2: email
Contents of files swapped….

b) What is containership? Explain with the help of example 8

Containership (Nesting of Classes)

- Inheritance is the mechanism of deriving certain properties of one class into another
- C++ supports a new way of inheriting classes:
  - An object can be collection of many other objects
  - A class can contain objects of other classes as its members

- For ex.:

```
class Alpha { …… };
class Beta { …… };
class Gamma
{
        Alpha a;
        Beta b;
        ……
};
```

- All objects of Gamma class will contain the objects a and b
- This is called containership or nesting

```
//Program to demonstrate Containership
#include<iostream.h>
#include<conio.h>
#include<stdio.h>

class Emp
{
    char ename[35];
    char des[15];
        public:
            void getdata()
            {
                cout<<"Enter emp. name and designation : \n";
                gets(ename);
                gets(des);
            }
            void putdata()
            {
                cout<<endl<<"Faculty name : "<<ename;
                cout<<endl<<"Designation  : "<<des;
            }
};
class Student
{
    char sname[35];
    char cname[15];
        public:
            void getdata()
            {
                cout<<"Enter student name and class name : \n";
                gets(sname);
                gets(cname);
            }
```

```
                    void putdata()
                    {
                        cout<<"\nStudent Name : "<<sname;
                        cout<<"\nClass Name   : "<<cname;
                    }
};
class Committee
{
    char comm[50];
    Emp e;
    Student s;
    public:
        void getdata()
        {
            cout<<"Enter Committee Name : ";
            gets(comm);
            e.getdata();
            s.getdata();
        }
        void putdata()
        {
            cout<<"Committee : "<<comm;
            e.putdata();
            s.putdata();
        }

};
void main()
{
    clrscr();
    Committee c;
    c.getdata();
    c.putdata();
    getch();
}
```

Output:
Enter Committee Name : Sports
Enter emp. name and designation :
Prof. Sheetal Kadam
Ass. Prof.
Enter student name and class name :
Rajiv Sharma
SY MCA
Committee : Sports
Faculty name : Prof. Sheetal Kadam
Designation : Ass. Prof.
Student Name : Rajiv Sharma
Class Name   : SY MCA

8 Write short notes on (any two): 4x2=8

a) Virtual Destructor

- Virtual destructor ensures that the object destruction happens from the most derived class towards the base class.
- Significant in scenarios where a derived class object is assigned to a base class pointer.
- EXAMPLE: Demonstrate the object destruction sequence.

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

class Base
{
        public:
                Base()
                {
                        cout<<"\nBase constructor";
                }
                 ~Base()
                {
                        cout<<endl<<"Base Destructor";
                }

};
class Derived : public Base
{
        public:
                Derived()
                {
                        cout<<endl<<"Derived constructor";
                }
                ~Derived()
                {
                        cout<<endl<<"Derived destructor";
                }
};
void main()
{
        clrscr();
        Base *p = new Derived;
        delete p;
        getch();
}
```

Output:
Base constructor
Derived destructor
Base destructor

- EXAMPLE: Demonstrate the object destruction sequence using virtual destructor

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

```
#include<conio.h>

class Base
{
        public:
                Base()
                {
                        cout<<"\nBase constructor";
                }
                virtual  ~Base()
                {
                        cout<<endl<<"Base Destructor";
                }

};
class Derived : public Base
{
        public:
                Derived()
                {
                        cout<<endl<<"Derived constructor";
                }
                ~Derived()
                {
                        cout<<endl<<"Derived destructor";
                }
};
void main()
{
        clrscr();
        Base *p = new Derived;
        delete p;
        getch();
}
```

Output:

Base constructor
Derived destructor
Base destructor
Derived destructor

b) Command line arguments

- C++ supports a feature that facilitates the supply of argument to the main( ) function
- These arguments are supplied at the time of invoking the program
- They are typically used to pass the names of data files
- For Ex.:-
  fcopy source target
- The command-line arguments are typed by the user and are delimited by a space
- The main function can take two arguments
  main(int argc, char * argv [ ])
- The first argument argc (argument counter) represents the number of

arguments in the command line

- The second argument argv (argument vector) is an array of char type pointers that points to the command line arguments
- The size of the array will be equal to the value of argc
- d:\> fcopy source target
- The value of argc would be 3 and argv would be an array of three pointers to strings as:

```
argv[0] → fcopy
argv[1] → source
argv[2] → target
……
……
fin.open(argv[1]);
……
fout.open(argv[2]);
……
```

```
//Command Line Arguments
#include<iostream.h>
#include<conio.h>

void main(int argc,char *argv[])
{
        clrscr();
        cout<<"Total No. of arg. : "<<argc;
        for(int i=0;i<argc;i++)
                cout<<endl<<argv[i];
        getch();
}
```

Output:

```
d:\demo\cmdarg.exe hello world
Total No. of arg.: 3
cmdarg.exe
hello
world
```

c) STL

- The C++ STL (Standard Template Library) is a powerful set of C++ template classes to provide general-purpose templatized classes and functions that implement many popular and commonly used algorithms and data structures like vectors, lists, queues, and stacks.
- The STL is a set of abstract data-types, functions, and algorithms designed to handle user-specified data-types
- The Standard Template Library is the idea of generic programming
    - The implementation of algorithms or data structures without being dependent on the type of data being handled

- The STL is a generic library, meaning that its components are heavily parameterized:
    - almost every component in the STL is a template
- STL is divided into three parts namely containers, algorithms, and iterators
    - All these three parts can be used for different programming problems
- 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.