Q.2.a. Briefly explain the features of OOP.

Ans. The important features of Object Oriented programming are:

1. Inheritance: Inheritance as the name suggests is the concept of inheriting or deriving properties of an exiting class to get new class or classes. In other words we may have common features or characteristics that may be needed by number of classes. So those features can be placed in a common tree class called base class and the other classes which have these characteristics can take the tree class and define only the new things that they have on their own in their classes. These classes are called derived class. The main advantage of using this concept of inheritance in Object oriented programming is it helps in reducing the code size since the common characteristic is placed separately called as base class and it is just referred in the derived class. This provide the users the important usage of terminology called as reusability

2. Polymorphism and overloading: Poly refers many. So Polymorphism as the name suggests is a certain item appearing in different forms or ways. That is making a function or operator to act in different forms depending on the place they are present is called Polymorphism. Overloading is a kind of polymorphism. In other words say for instance we know that +, - operate on integer data type and is used to perform arithmetic additions and subtractions. But operator overloading is one in which we define new operations to these operators and make them operate on different data types in other words overloading the existing functionality with new one. This is a very important feature of object oriented programming methodology which extended the handling of data type and operations.

3. Data Hiding: This concept is the main heart of an Object oriented programming. The data is hidden inside the class by declaring it as private inside the class. When data or functions are defined as private it can be accessed only by the class in which it is defined. When data or functions are defined as public then it can be accessed anywhere outside the class. Object Oriented programming gives importance to protecting data which in any system. This is done by declaring data as private and making it accessible only to the class in which it is defined. This concept is called data hiding. But one can keep member functions as public.

4. Encapsulation: The technical term for combining data and functions together as a bundle is encapsulation.

5. Reusability: Reusability is nothing but re- usage of structure without changing the existing one but adding new features or characteristics to it. It is very much needed for any programmers in different situations. Reusability gives the following advantages to users. It helps in reducing the code size since classes can be just derived from existing one and one need to add only the new features and it helps users to save their time.

b. What is a reference variable? Write a C++ program to find the sum of two numbers using reference variables.

Ans. C++ references allow you to create a second name for a variable that you can use to read or modify the original data stored in that variable.

Declaring a variable as a reference rather than a normal variable simply entails appending an ampersand to the type name, such as this "reference to an int"

```
int  a =5;
int & b =a;

#include<iostream.h>
void main()
{
        int a, b;
        int sum;
        int &c  =a;
        int &d = b;
        cout << “Enter data for a and b \n”;
        cin >> a >> b;
        sum = c + d;
        cout << “ The sum of two number “<< sum;
}
```

c. Describe the new and delete operators with examples.

Ans. C++ supports dynamic allocation and deallocation of objects using the new and delete operators. These operators allocate memory for objects from a pool called the free store. The new operator calls the special function operator new, and the delete operator calls the special function operator delete.

```
pointer = new type
pointer = new type [number_of_elements]

int * bobby;
bobby = new int [5];

delete pointer;
delete [] pointer;

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

```
int main ()
{
        int i,n;
        int * p;
        cout << "How many numbers would you like to type? ";
        cin >> i;
        p= new (nothrow) int[i];
        if (p == 0)
        cout << "Error: memory could not be allocated";
        else
        {
                for (n=0; n<i; n++)
                {
                        cout << "Enter number: ";
                        cin >> p[n];
                }
        cout << "You have entered: ";
                for (n=0; n<i; n++)
                cout << p[n] << ", ";
                delete[] p;
        }
        return 0;
}
```

Q3.a. What is an inline function? List its merits and demerits. Write a program to find the smaller of two numbers using inline function and the ternary operator.

Ans. Inline functions are functions where the call is made to inline functions. The actual code then gets placed in the calling program.
Normally, a function call transfers the control from the calling program to the function and after the execution of the program returns the control back to the calling program after the function call. These concepts of function save program space and memory space and are used because the function is stored only in one place and is only executed when it is called. This execution may be time consuming since the registers and other processes must be saved before the function gets called.

The extra time needed and the process of saving is valid for larger functions. If the function is short, the programmer may wish to place the code of the function in the calling program in order for it to be executed. This type of function is best handled by the inline function. In this situation, the programmer may be wondering "why not write the short code repeatedly inside the program wherever needed instead of going for inline function?". Although this could accomplish the task, the problem lies in the loss of clarity of the program. If the programmer repeats the same code many times, there will be a loss of clarity in the program.

The alternative approach is to allow inline functions to achieve the same purpose, with the concept of functions.

inline int min(int a, int b) { return (a < b) ? a: b; }

b. What is function overloading? Explain 3 steps of overload resolution with an example.

Ans. C++ permits the use of two functions with the same name. However such functions essentially have different argument list. The difference can be in terms of number or type of arguments or both.

This process of using two or more functions with the same name but differing in the signature is called function overloading.

But overloading of functions with different return types are not allowed. In overloaded functions, the function call determines which function definition will be executed.

The biggest advantage of overloading is that it helps us to perform same operations on different data types without having the need to use separate names for each version.

Example

```
int absInt( int );
long absInt( long);
float absInt( float);
```

c. Explain the use of scope resolution operator with an example.

Ans. The scope resolution operator helps to identify and specify the context to which an identifier refers.
The scope resolution operator (::) in C++ is used to define the already declared member functions (in the header file with the .hpp or the .h extension) of a particular class. In the .cpp file one can define the usual global functions or the member functions of the class. To differentiate between the normal functions and the member functions of the class, one needs to use the scope resolution operator (::) in between the class name and the member function name i.e. ship::foo() where ship is a class and foo() is a member function of the class ship. The other uses of the resolution operator is to resolve the scope of a variable when the same identifier is used to represent a global variable, a local variable, and members of one or more class (es). If the resolution operator is placed between the class name and the data member belonging to the class then the data name belonging to the particular class is referenced. If the resolution operator is placed in front of the variable name then the global variable is referenced. When no resolution operator is placed then the local variable is referenced.

```
int n = 12; // A global variable

int main()
{
        int n = 13; // A local variable
        cout << ::n << endl; // Print the global variable: 12
        cout << n   << endl; // Print the local variable: 13
}
```

Q.4.a. Define "class" and "object". With an example, explain the concept of data encapsulation and accessing of member elements.

Ans. A *class* is an expanded concept of a data structure: instead of holding only data, it can hold both data and functions.

An *object* is an instantiation of a class. In terms of variables, a class would be the type, and an object would be the variable.

Classes are generally declared using the keyword class, with the following format:

```
class class_name {
access_specifier_1:
 member1;
access_specifier_2:
 member2;
...
 } object_names;
```

Example:

```
#include <iostream>

class CRectangle
{
        int x, y;
                public:
                void set_values (int,int);
                int area () {return (x*y);}
};

void CRectangle::set_values (int a, int b)
{
        x = a;
        y = b;
}
```

```
int main ()
{
        CRectangle rect;
        rect.set_values (3,4);
        cout << "area: " << rect.area();
        return 0;
}
```

b. Write a note on parameterized constructor and destructor with default arguments.

Ans. The main use of constructors is to initialize objects. The function of initialization is automatically carried out by the use of a special member function called a constructor.

A constructor is a special member function that takes the same name as the class name. The syntax generally is as given below:

{ arguments};

Some important points about constructors:

- A constructor takes the same name as the class name.
- The programmer cannot declare a constructor as virtual or static, nor can the programmer declare a constructor as const, volatile, or const volatile.
- No return type is specified for a constructor.
- The constructor must be defined in the public. The constructor must be a public member.
- Overloading of constructors is possible.

Destructors are also special member functions used in C++ programming language. Destructors have the opposite function of a constructor. The main use of destructors is to release dynamic allocated memory. Destructors are used to free memory, release resources and to perform other clean up. Destructors are automatically named when an object is destroyed. Like constructors, destructors also take the same name as that of the class name.

~ classname();

Some important points about destructors:

- Destructors take the same name as the class name.

- Like the constructor, the destructor must also be defined in the public. The destructor must be a public member.
- The Destructor does not take any argument, which means that destructors cannot be overloaded.
- No return type is specified for destructors.

c. Create a class Date which has dd, mm and yy as its member variables. A constructor with 3 arguments to initialize data members and member functions to:,
i) Display date in dd:mm:yy format
ii) Find the difference between two dates and display the total number of days.
Also provide the main function to initialize 2 different date objects and display the number of days between them.

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

int days[13]={0,31,28,31,30,31,30,31,31,30,31,30,31};
int leap_days[13]={0,31,29,31,30,31,30,31,31,30,31,30,31};

int is_leap(int yy)
{
  if(yy%400==0 || (yy%4==0 && yy%100!=0))
    return 1;
  else
    return 0;
}

class date
{
        private:int dd, mm, yy;

         public:  date()
                  {
                        dd=1, mm=1, yy=2001;
                   }
                 date(int a, int b, int c)
                  {
                        dd=a, mm=b, yy=c;
                   }
```

```
    void read();   //checks valid date or not
    long julian();   //calc no of days from 1-yy 1-mm 1-dd
    friend ostream& operator <<(ostream& os, date& d);
    int operator -(date a);   //calls julian-a.julian
    date operator +(int nd);
};

long date :: julian()
{
  long i, t=0;
  for(i=1;i<yy;i++)
  {
    if(is_leap(i))
      t+=366;
    else
      t+=365;
  }
  for(i=1;i<mm;i++)
  {
    if(is_leap(yy))
      t+=leap_days[i];
    else
      t+=days[i];
  }
  t+=dd;
  return t;
}

void date :: read()
{
  char ch;
  cout<<endl<<"Enter date in (dd/mm/yy) format:";
  cin>>dd>>ch>>mm>>ch>>yy;

  if(is_leap(yy))
  {
    if(mm>12 || dd>leap_days[mm])
    {
      cout<<endl<<"Wrong date";
      read();
    }
  }
  else
  {
    if(mm>12 || dd>days[mm])
    {
```

```
        cout<<endl<<"Wrong date";
        read();
      }
    }
}

ostream& operator <<(ostream& os, date& d)
{
  os<<d.dd<<"/"<<d.mm<<"/"<<d.yy;
}

int date :: operator -(date a)
{
  int diff;
  diff=julian()-a.julian();
  return abs(diff);
}

date date :: operator +(int nd)
{
  date next=*this;

  while(nd)
  {
    next.dd++;
    nd--;
    if(is_leap(next.yy))   //checks whether leap year or not
    {
      if(next.dd>leap_days[next.mm])   //if leap year compares with leap_days
array

      {
        next.dd=1;
        next.mm++;
      }
    }
    else
    {
      if(next.dd>days[next.mm])
      {
        next.dd=1;
        next.mm++;
      }
    }
    if(next.mm>12)
    {
```

```
        next.mm=1;
        next.yy++;
      }
    }
    return next;
}

int main()
{
    date d1, d2;
    int nd, choice, done=0;

    while(!done)
    {
      cout<<endl<<"1.Difference in dates"
          <<endl<<"2.Future date"
              <<endl<<"0.Exit"
          <<endl<<"Enter your choice:";
          cin>>choice;

      switch(choice)
      {
        case 1: d1.read();
          d2.read();
          nd=d1-d2;
          cout<<endl<<"Difference="<<nd<<" days";
          break;
        case 2: d1.read();
          cout<<endl<<"Enter number of days to add:";
          cin>>nd;
          d2=d1+nd;
          cout<<endl<<"Future date="<<d2;
          break;
        case 0:
        default:done=1;
          exit(0);
      }
    }
    return 0;
}
```

Q.5.a. Write a C++ program to create a class called MATRIX using 2-dimensional array of integers. Implement the following by overloading the operator == which checks the compatibility of two matrices to be added and subtracted. Perform the following by overloading + (plus) and – (Minus) operators. Display the result by overloading the operator <<.

```
if ( m1== m2) {
m3=m1+m2;
m4=m1-m2;
}
else
        Display error.
```

Where m1, m2, m3 and m4 are MATRIX class objects.

Ans.

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

class Matrix
{
    int a[10][10];
    int row,col;

public:

    void getmatrix()
    {
        cin>>row>>col;
    }

    void read();

friend ostream & operator << ( ostream & x,Matrix & m );

    Matrix operator + ( Matrix );
    Matrix operator - ( Matrix );

    int operator == ( Matrix m2 )
    {
        if( row==m2.row && col==m2.col )
        return(1);
        else
        return(0);
    }
};
    void Matrix :: read()
{
```

```
                int i,j;

                for(i=0; i<row; i++ )
                for(j=0; j<col; j++ )
                        cin>>a[i][j];
        }

ostream & operator << ( ostream & x,Matrix & m )
{

        int i,j;

        for(i=0; i<m.row; i++ )
        {
                for(j=0; j<m.col; j++ )
                   cout<<m.a[i][j]<<"   "
        cout<<"\n";
        }
        return(x);
}

Matrix Matrix :: operator + ( Matrix m2 )
{
        Matrix res;
        res.row=row;
        res.col=col;

        int i,j;

        for(i=0; i<row; i++ )
        for(j=0; j<col; j++ )
           res.a[i][j]=a[i][j]+m2.a[i][j];

        return(res);
}

Matrix Matrix :: operator - ( Matrix m2 )
{
        Matrix res;
        res.row=row;
        res.col=col;

        int i,j;

        for(i=0; i<row; i++ )
        for(j=0; j<col; j++ )
```

```
            res.a[i][j]=a[i][j]-m2.a[i][j];

        return(res);
}

int main()
{
        clrscr();
        Matrix m1,m2,m3,m4;

        cout<<"\nEnter the order of 1st matrix : ";
        m1.getmatrix();

        cout<<"\nEnter the order of 2nd matrix : ";
        m2.getmatrix();

        if( m1==m2 )
        {
                cout<<"\nEnter the elements of 1st matrix\n";
                m1.read();

                cout<<"\nEnter the elements of 2nd matrix\n";
                m2.read();

                m3=m1+m2;
                m4=m1-m2;

                cout<<"\nm1 is..\n";
                cout<<m1;

                cout<<"\nm2 is..\n";
                cout<<m2;

                cout<<"\nm3 is..\n";
                cout<<m3;

                cout<<"\nm4 is..\n";
                cout<<m4;
        }
        else
                cout<<"\nOperation is not possible !";

getch();
return 0;
}
```

b. Write a program to overload new and delete operators.

Ans. Pg. No. 253-256 of Text Book 1- C++ & OOPs Paradigm, Debasish Jana, Second Edition, PHI 2005

Q.6.a. What is a derived class? Explain with examples the three ways in which a class can be inherited.

Ans. Creating or deriving a new class using another class as a base is called inheritance in C++. The new class created is called a Derived class and the old class used as a base is called a Base class in C++ inheritance terminology.

The derived class will inherit all the features of the base class in C++ inheritance. The derived class can also add its own features, data etc.; it can also override some of the features (functions) of the base class, if the function is declared as virtual in base class.

Public inheritance

Public inheritance is by far the most commonly used type of inheritance. In fact, very rarely will you use the other types of inheritance, so your primary focus should be on understanding this section. Fortunately, public inheritance is also the easiest to understand. When you inherit a base class publicly, all members keep their original access specifications. Private members stay private, protected members stay protected, and public members stay public.

```
class Base
{
        public:
           int m_nPublic;
        private:
           int m_nPrivate;
        protected:
           int m_nProtected;
};

class Pub: public Base
{  };
```

Private inheritance

With private inheritance, all members from the base class are inherited as private. This means private members stay private, and protected and public members become private.

Note that this does not affect that way that the derived class accesses members inherited from its parent! It only affects the code trying to access those members through the derived class.

class Base

```
{
public:
   int m_nPublic;
private:
   int m_nPrivate;
protected:
   int m_nProtected;
};

class Pri: private Base
{  };
```

Protected inheritance

Protected inheritance is the last method of inheritance. It is almost never used, except in very particular cases. With protected inheritance, the public and protected members become protected, and private members stay private.

b. Explain how pointers are used in base and derived classes with an example.

Ans. Pointers to base class

One of the key features of derived classes is that a pointer to a derived class is type-compatible with a pointer to its base class. Polymorphism is the art of taking advantage of this simple but powerful and versatile feature, that brings Object Oriented Methodologies to its full potential.

```
// pointers to base class
#include <iostream>
using namespace std;

class CPolygon {
 protected:
   int width, height;
 public:
   void set_values (int a, int b)
     { width=a; height=b; }
 };

class CRectangle: public CPolygon {
 public:
   int area ()
     { return (width * height); }
 };

class CTriangle: public CPolygon {
 public:
```

```
  int area ()
    { return (width * height / 2); }
};

int main () {
  CRectangle rect;
  CTriangle trgl;
  CPolygon * ppoly1 = &rect;
  CPolygon * ppoly2 = &trgl;
  ppoly1->set_values (4,5);
  ppoly2->set_values (4,5);
  cout << rect.area() << endl;
  cout << trgl.area() << endl;
  return 0;
}
```

In function main, we create two pointers that point to objects of class CPolygon (ppoly1 and ppoly2). Then we assign references to rect and trgl to these pointers, and because both are objects of classes derived from CPolygon, both are valid assignment operations.

The only limitation in using *ppoly1 and *ppoly2 instead of rect and trgl is that both *ppoly1 and *ppoly2 are of type CPolygon* and therefore we can only use these pointers to refer to the members that CRectangle and CTriangle inherit from CPolygon. For that reason when we call the area() members at the end of the program we have had to use directly the objects rect and trgl instead of the pointers *ppoly1 and *ppoly2.

In order to use area() with the pointers to class CPolygon, this member should also have been declared in the class CPolygon, and not only in its derived classes, but the problem is that CRectangle and CTriangle implement different versions of area, therefore we cannot implement it in the base class. This is when virtual members become handy:

```
// virtual members
#include <iostream>
using namespace std;

class CPolygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b; }
    virtual int area ()
      { return (0); }
  };
```

```
class CRectangle: public CPolygon {
  public:
    int area ()
      { return (width * height); }
  };

class CTriangle: public CPolygon {
  public:
    int area ()
      { return (width * height / 2); }
  };

int main () {
  CRectangle rect;
  CTriangle trgl;
  CPolygon poly;
  CPolygon * ppoly1 = &rect;
  CPolygon * ppoly2 = &trgl;
  CPolygon * ppoly3 = &poly;
  ppoly1->set_values (4,5);
  ppoly2->set_values (4,5);
  ppoly3->set_values (4,5);
  cout << ppoly1->area() << endl;
  cout << ppoly2->area() << endl;
  cout << ppoly3->area() << endl;
  return 0;
}
```

Now the three classes (CPolygon, CRectangle and CTriangle) have all the same members: width, height, set_values() and area().

The member function area() has been declared as virtual in the base class because it is later redefined in each derived class. You can verify if you want that if you remove this virtual keyword from the declaration of area() within CPolygon, and then you run the program the result will be 0 for the three polygons instead of 20, 10 and 0. That is because instead of calling the corresponding area() function for each object (CRectangle::area(), CTriangle::area() and CPolygon::area(), respectively), CPolygon::area() will be called in all cases since the calls are via a pointer whose type is CPolygon*.

Therefore, what the virtual keyword does is to allow a member of a derived class with the same name as one in the base class to be appropriately called from a pointer, and more precisely when the type of the pointer is a pointer to the base class but is pointing to an object of the derived class, as in the above example.

A class that declares or inherits a virtual function is called a *polymorphic class*.

Note that despite of its virtuality, we have also been able to declare an object of type CPolygon and to call its own area() function, which always returns 0.

c. What are virtual functions? Explain the usage of virtual functions with examples.

Ans. In object-oriented programming, a virtual function or virtual method is a function or method whose behavior can be overridden within an inheriting class by a function with the same signature. This concept is a very important part of the polymorphism portion of object-oriented programming (OOP).

C++ virtual function is,

- A member function of a class
- Declared with *virtual* keyword
- Usually has a different functionality in the derived class
- A function call is resolved at run-time

```
class Window // Base class for C++ virtual function example
{
 public:
   virtual void Create() // virtual function for C++ virtual function example
   {
      cout <<"Base class Window"<<ENDL;
   }
};

class CommandButton : public Window
{
 public:
   void Create()
   {
      cout<<"Derived class Command Button - Overridden C++ virtual
function"<<ENDL;
   }
};

void main()
{
   Window  *x, *y;

   x = new Window();
   x->Create();

   y = new CommandButton();
```

```
    y->Create();
}
```

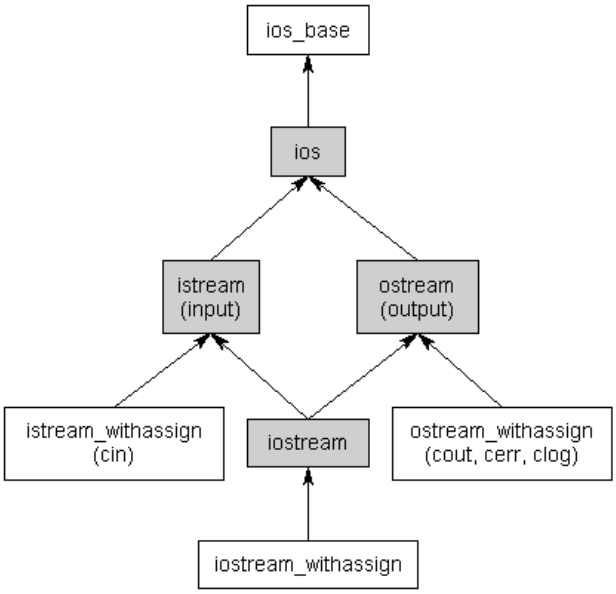
Q7 a. Mention any two functions for each of the following:-

i. I/P Stream
ii. O/P Stream

Ans. Pg. No. 354, 362, 363, 364-365 of Text Book 1- C++ & OOPs Paradigm, Debasish Jana, Second Edition, PHI 2005

b. Give the structure of Stream class hierarchy.

Ans.



Abstractly, a stream can be thought of as a sequence of bytes of infinite length that is used as a buffer to hold data that is waiting to be processed.

Typically we deal with two different types of streams. Input streams are used to hold input from a data producer, such as a keyboard, a file, or a network. For example, the user may press a key on the keyboard while the program is currently not expecting any input. Rather than ignore the users key press, the data is put into an input stream, where it will wait until the program is ready for it.

Conversely, output streams are used to hold output for a particular data consumer, such as a monitor, a file, or a printer. When writing data to an output device, the device may not be ready to accept that data yet — for example, the printer may still be warming up when the program writes data to its output stream. The data will sit in the output stream until the printer begins consuming it.

The istream class is the primary class used when dealing with input streams. With input streams, the extraction operator (>>) is used to remove values from the stream. This makes sense: when the user presses a key on the keyboard, the key code is placed in an input stream. Your program then extracts the value from the stream so it can be used.

The ostream class is the primary class used when dealing with output streams. With output streams, the insertion operator (<<) is used to put values in the stream. This also makes sense: you insert your values into the stream, and the data consumer (eg. monitor) uses them.

The iostream class can handle both input and output, allowing bidirectional I/O.

c. Write a C++ program demonstrate reading from and writing to a text file.

Ans:

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

using namespace std;

class File
{
        ifstream f1;
        ofstream f2;
        char s[200];

public:
        void ShowFile(char *nume);
         void ReadInFile(char *nume);
};

        void File::ReadInFile(char *nume)
        {
                f2.open(nume);
                for(int i=0; i<strlen(s); i++)
                {
                        f2<<s[i];
                }

                f2.close();
        }

void File::ShowFile(char *nume)
{
```

```
    f1.open(nume);
    if(f1.fail())
    {
              cout<<"Error opening file!";
              getch();
              exit(1);
    }
    while(!f1.eof())
    {
               for(int i=0; i<strlen(s); i++)
               {
                     f1>>s[i];
                     cout<<s[i];
               }
    }
    f1.close();
}

int main()
{
   File F;
   F.ShowFile("slav.txt");
   F.ReadInFile("slav2.txt");

    getch();
    return 0;
}
```

Q.8.a. What is a class template? With syntax, explain the purpose of the class template with multiple parameters.

Ans. Templates are a feature of the C++ programming language that allow functions and classes to operate with generic types. This allows a function or class to work on many different data types without being rewritten for each one.

Class templates

A class template provides a specification for generating classes based on parameters. Class templates are commonly used to implement containers. A class template is instantiated by passing a given set of types to it as template arguments.[1] The C++ Standard Library contains many class templates, in particular the containers adapted from the Standard Template Library, such as vector.

```
template <class T>
class mypair
{
```

```
  T values [2];
public:
        mypair (T first, T second)
{
 values[0]=first; values[1]=second;
}
};
```

The class that we have just defined serves to store two elements of any valid type. For example, if we wanted to declare an object of this class to store two integer values of type int with the values 115 and 36 we would write:

```
mypair<int> myobject (115, 36);
```

this same class would also be used to create an object to store any other type:

```
mypair<double> myfloats (3.0, 2.18);
// class templates
#include <iostream>
using namespace std;

template <class T>
class mypair
{
   T a, b;
  public:
   mypair (T first, T second)
     {a=first; b=second;}
   T getmax ();
};

template <class T>
T mypair<T>::getmax ()
{
  T retval;
  retval = a>b? a : b;
  return retval;
}

int main () {
  mypair <int> myobject (100, 75);
  cout << myobject.getmax();
  return 0;
}
```

b. Write a C++ program to overload a template function called swap().

Ans.

```
template<class T>
void swap(T& a, T& b)
{
  T tmp = a;
 a = b;
 b = tmp;
}

void swap(int& a, int& b)
{
        int tmp = a;
        a = b;
        b = tmp;
}
```

c. Explain the exception handling mechanism with an example.

Ans. *Exception handling* is a mechanism that separates code that detects and handles exceptional circumstances from the rest of your program. Note that an exceptional circumstance is not necessarily an error.

When a function detects an exceptional situation, you represent this with an object. This object is called an *exception object*. In order to deal with the exceptional situation you *throw the exception*. This passes control, as well as the exception, to a designated block of code in a direct or indirect caller of the function that threw the exception. This block of code is called a *handler*. In a handler, you specify the types of exceptions that it may process. The C++ run time, together with the generated code, will pass control to the first appropriate handler that is able to process the exception thrown. When this happens, an exception is *caught*. A handler may *rethrow* an exception so it can be caught by another handler.

The exception handling mechanism is made up of the following elements:

- try blocks
- catch blocks
- throw expressions
- Exception specifications

One benefit of C++ over C is its exception handling system. An exception is a situation in which a program has an unexpected circumstance that the section of code containing the problem is not explicitly designed to handle. In C++, exception handling is useful

because it makes it easy to separate the error handling code from the code written to handle the chores of the program. Doing so makes reading and writing the code easier.

Furthermore, exception handling in C++ propagates the exceptions up the stack; therefore, if there are several functions called, but only one function that needs to reliably deal with errors, the method C++ uses to handle exceptions means that it can easily handle those exceptions without any code in the intermediate functions. One consequence is that functions don't need to return error codes, freeing their return values for program logic.

When errors occur, the function generating the error can 'throw' an exception. For example, take a sample function that does division:

```
const int DivideByZero = 10;
//....
double divide(double x, double y)
{
   if(y==0)
   {
      throw DivideByZero;
   }
   return x/y;
}
```

The function will throw DivideByZero as an exception that can then be caught by an exception-handling catch statement that catches exceptions of type int. The necessary construction for catching exceptions is a try catch system. If you wish to have your program check for exceptions, you must enclose the code that may have exceptions thrown in a try block. For example:

```
try
{
   divide(10, 0);
}
catch(int i)
{
   if(i==DivideByZero)
   {
      cerr<<"Divide by zero error";
   }
}
```

The catch statement catches exceptions that are of the proper type. You can, for example, throw objects of a class to differentiate between several different exceptions. As well,

once a catch statement is executed, the program continues to run from the end of the catch.

Q9 a. Explain various control statements used in C++.

Ans. Pg. No.-60-62 of Text Book 1- C++ & OOPs Paradigm, Debasish Jana, Second Edition, PHI 2005

b. Write a program to illustrate pointers to pointers.

Ans. Pg. No.-111 of Text Book 1- C++ & OOPs Paradigm, Debasish Jana, Second Edition, PHI 2005

c. Explain the role of pointer constants and pointer arithmetic. Give an example.

Ans. Pg. No.-114 of Text Book 1- C++ & OOPs Paradigm, Debasish Jana, Second Edition, PHI 2005