Shallow copy :
when we assign one object to another object at that time copying all the contents from source object to destination object as it is. Such type of copy is called as shallow copy.
Compiler by default create a shallow copy. Default copy constructor always create shallow copy.

Deep Copy : when class contains at least one data member of pointer type , when class contains user defined destructor and when we assign one object to another object at that time instead of copy base address allocate a new memory for each and every object and then copy contains form memory of source object into memory of destination object. Such type of copy is called as deep copy.

Condition for deep copy:
1. Class must contain at least one data member of pointer type.
2. class must contain user define destructor
3. one object must be assigned to another object.

Exception Handling:

runtime error which can be handled by programmer is called exception.

In c++ exceptions are handled using try, catch and throw.

```
#include<iostream.h>
int main()
{       int x=10,y=0;
        try
        {       if(y==0)
                {    throw 1;
                }
                else
                {               int res=x/y;
                        cout<<"res::"<<res<<endl;
                }
        }
        catch(int)
        {       cout<<"Enter Other Than 0"<<endl;
        }
        return  0;
}
```

Try block must have at least one catch handler.
One try block can have multiple catch blocks .

When exception is thrown but matching catch block is not available at that time compiler give call to library defined function terminate which internally give call to abort function.
Catch block which handles all kind of exceptions such type of catch is called generic catch block.

```
catch(…)  (ellipse)
{
        cout<<“inside genric block”<<endl;
}
```

catch handler for ellipse must last handler in exception handling

Generic Programming- Template

Many times we need to write the code that is common for many data types. The simplest example is Swap() function. We can observe that logic of swapping remain same irrespective of the data type used.

We can write templates of function, instead writing separate function for each data type. The syntax is as follows:

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

Now the same function can be used to swap, two integer variables of float variables or even any user defined type’s variables.

If function is called as:

```
        Swap(a, b);   // where a and b are int variable
```

Compiler converts above template function, to the function for swapping integer variables. To do this, compiler simply converts above function to a new function by replacing “T” with “int”. Thus compiler creates the functions corresponding to every data type for which function has been called.

The way we can use template functions to represent generic algorithms, we can write template class to represent generic classes. The best examples are data structure classes like stack, queue, linked list, etc. Template classes are also called as “meta classes”, because compiler generates real classes from template classes and then create the objects.

RTTI & casting operators

C++ provides feature of Run Time Type Information, which enables to find type of the object at runtime. For this feature, language contains “typeid” operator that returns reference to constant object of “type_info” class. This class is declared in header <typeinfo> and contains information about the data type i.e. name of type.

For example, assuming that class circle and rectangle are inherited from shape interface, following code explains use of RTTI:

```
#include<typeinfo>

Shape *ptshape=NULL;
int choice;
cin >> choice;
if(choice==1)
     ptshape = new Circle;
else
     ptshape = new Rectangle;

const type_info& info = typeid(*ptshape);
cout << info.name() << endl;
delete ptshape;
ptshape=NULL;
```

```
friend ostream& operator<<(ostream& out, Complex &c1 )
{
 out<<"c1->real = "<<c1.real<<endl;
 out<<"c1->imag ="<<c1.imag<<endl;
 return out;
}
// operator<<(cout,c1);
//  operator<<(operator<<(cout,c2), c3);

friend istream& operator>>(istream& in , Complex &c1)
{
 in>>c1.real;
 in>>c1.imag;
 return in;
}
// operator>>(cin, c1);
// operator>>(operator>>(cin, c2),c3);
```