C++

Lesson 2 : Object Oriented C++

Lesson Coverage

- Classes & Objects
- Inheritance
  - Types of Inheritance
- Overloading
  - Function
  - Operator
- Polymorphism
  - Overriding
  - Virtual function
- Abstraction
  - Abstract method
  - Abstract class
- Encapsulation
- Interfaces

Introduction to Object Orientation

- OOP is a paradigm of application development where programs are built around objects and their interactions with each other.
  - An Object Oriented program can be viewed as a collection of co-operating objects
- An OO program is made up of several objects that interact with each other to make up the application.
- **For example:** In a Banking System, there would be Customer objects pertaining to each customer. Each customer object would own its set of Account Objects, pertaining to the set of Savings and Current Accounts that the customer holds in the bank.

- Today, most programming languages are object oriented.
  For example: Java, C++, C#

Object Oriented Approach

- OO Approach is based on the concept of building applications and programs from a collection of “reusable entities” called “objects”.
  - Each object is capable of receiving and processing data, and further sending it to other objects.
  - Objects represent real-world business entities, either physical, conceptual, or software.
- **For example:** a person, place, thing, event, concept, screen, or report

Why Object Oriented Programming?

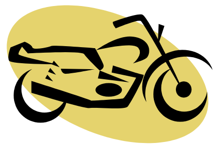
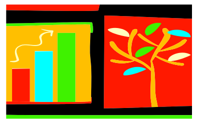
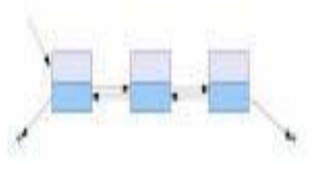
- There are problems associated with structured language, namely:
  - Emphasis is on doing things rather than on data
  - Most of the functions share global data which lead to their unauthorized access
  - More development time is required
  - Less reusability
  - Repetitive coding and debugging
  - Does not model real world well

Advantages with OOPS

- **Simplicity:** Software objects model the real world objects. Hence the complexity is reduced and the program structure is very clear.
- **Modularity:** Each object forms a “separate entity” whose internal workings are decoupled from other parts of the system.
- **Modifiability:** Changes inside a class do not affect any other part of a program, since the only “public interface” that the external world has to a class is through the use of “methods”.
- **Extensibility:** Adding new features or responding to changing operating environments can be solved by introducing a few new objects and modifying some existing ones.
- **Maintainability:** Objects can be separately maintained, thus making locating and fixing problems easier.
- **Re-usability:** Objects can be reused in different programs.

Objects

- An object is a tangible, intangible, or software entity.
- It is characterized by Identity, State, and Behavior.
  - **Identity:** It distinguishes one object from another.
  - **State:** It comprises set of properties of an object, along with its values.
  - **Behavior:** It is the manner in which an object acts and reacts to requests received from other objects

Object State

- **State** of an object is one of the possible conditions in which the object may exist.

Bank Account Modeled as a Software Object

Attributes of the object:

Account Number: A10056

Type: Savings

Balance: 40000

Name: Sarita Kale

The state of an object is not defined by a "state" attribute or a set of attributes. Instead the "state" of an object gets defined as a total of all the attributes and links of that object.

Object Behavior

- **Behavior** of an object determines how an object reacts to other objects.

Behaviors of the object

- Withdraw
- Deposit
- Check Balance
- Get Monthly statement

These are the operations that the object can perform, and represents its behavior.

Through these operations or methods, an object controls its state.

Object Identity

- Two objects can posses identical attributes (state) and yet have distinct identities.

Class

- A Class characterizes common structure and behavior of a set of objects.
- It constitutes of Attributes and Operations.
- It serves as a template from which objects are created in an application.

Class name	Customer
Class attributes	Name, Address, Email-ID, TelNumber
Class operations	displayCustomerDetails() changeContactDetails()

Class Attributes and Operations

- Each class has a name, attributes, and operations.
  - **Attribute** is a named property of a class that describes a range of values.
  - **Operation** is the implementation of a service that can be requested from any object of the class to affect behavior.

	Class Name	Account
Attributes	Class attributes	AccountNumber, Type, Balance
Operations	Class operations	withdraw(), checkBalance(), displayAccountDetails, deposit()

Constructors and Destructors

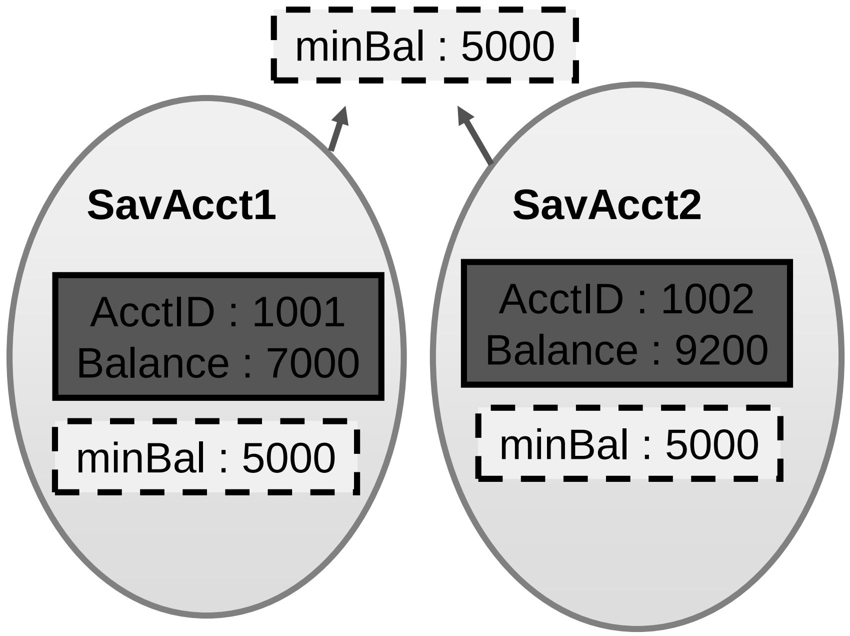
- Constructors:
  - They enable instantiation of objects against defined classes.
  - Memory is set aside for the object. Attribute values can be initialized along with object creation (if needed).

- Destructors:
  - They come into play at end of object lifetime. They release memory and other system locks.

Static Members

- Static data members are those that are shared amongst all object instances of the given type.

  - **Example:** Rate of Interest for Savings Account will be the same in the bank, so common copy is sufficient. Not so for Account Balances!

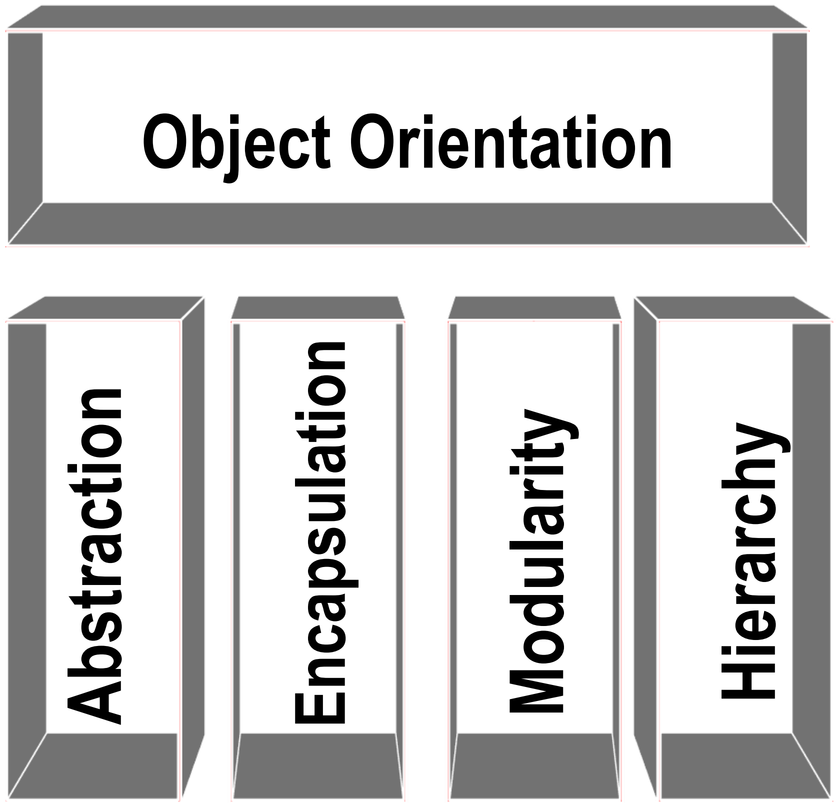


Static Members

- Static Member Functions can be invoked without an object instance.
  - **For example:** Counting the number of Customer Objects created in the Banking System – this is not specific to one object!

Object Oriented Principles

- OO is based on four basic principles, namely:
  - **Principle 1:** Abstraction
  - **Principle 2:** Encapsulation
  - **Principle 3:** Modularity
  - **Principle 4:** Hierarchy

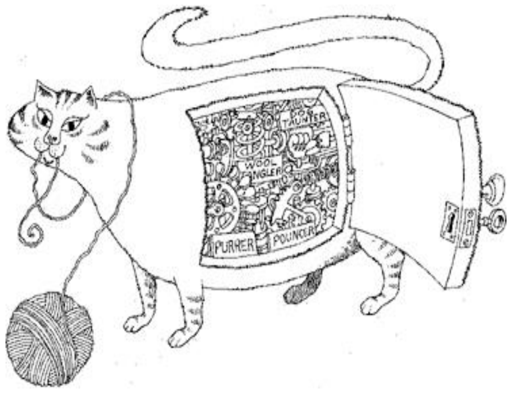


Concept of Abstraction

- Focus only on the essentials, and only on those aspects needed in the given context.
  - **For example:** Customer Height / Weight not needed for Banking System!
- Abstraction is determining the essential qualities. By emphasizing on the important characteristics and ignoring the non-important ones, one can reduce and factor out those details that are not essential, resulting in less complex view of the system.
- Abstraction means that we look at the external behavior without bothering about internal details. We do not need to become car mechanics to drive a car!

Concept of Encapsulation

- “To Hide” details of structure and implementation
  - **For example:** It does not matter what algorithm is implemented internally so that the customer gets to view Account status in Sorted Order of Account Number.
- Every object is encapsulated in such a way, that its data and implementations of behaviors are not visible to another object.
- Encapsulation allows restriction of access of internal data.



Encapsulation versus Abstraction

- Abstraction and Encapsulation are closely related.
  - Abstraction can be considered as User's perspective.
  - Encapsulation can be considered as Implementer's perspective.

- Why Abstraction and Encapsulation?
  - They result in "Less Complex" views of the System.
  - Effective separation of inside and outside views leads to more flexible and maintainable systems.

Concept of Modularity

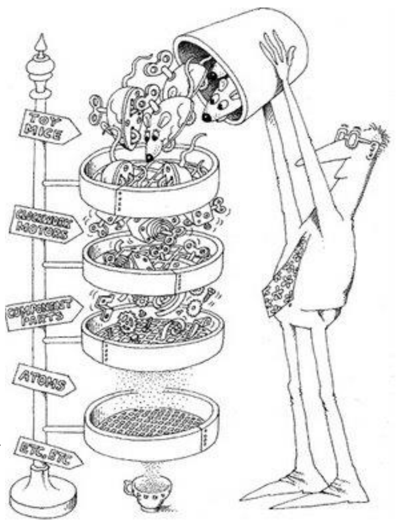
- Decomposing a system into smaller, more manageable parts
  - **Example:** Banking System can have different modules to take care of Customer Management, Account Transactions, and so on.

- Why Modularity?
  - Divide and Rule! Easier to understand and manage complex systems.
  - Allows independent design and development of modules.

- As modules are groups of related classes, it is possible to have parallel developments of modules. Changes in one may not affect the other modules. Modularity is an essential characteristic of all complex systems. Well designed modules can be reused in similar situations in other designs.

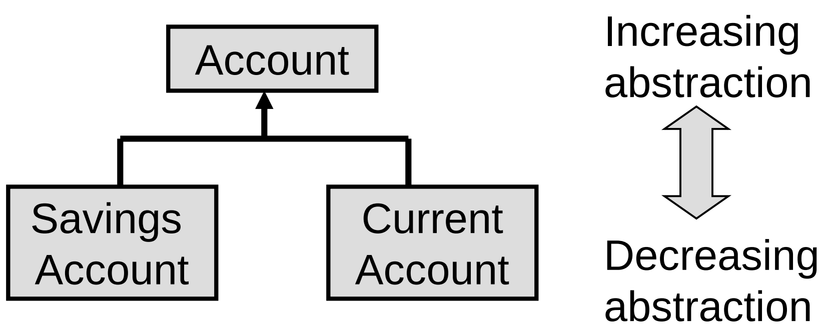
Concept of Hierarchy

- A ranking or ordering of abstractions on the basis of their complexity and responsibility
- It is of two types:
  - **Class Hierarchy:** Hierarchy of classes, **Is A** Relationship.
    - Example: Accounts Hierarchy
  - **Object Hierarchy:** Containment amongst Objects, **Has A** Relationship.
    - Example: Window has a Form seeking customer information, which has text boxes and various buttons.



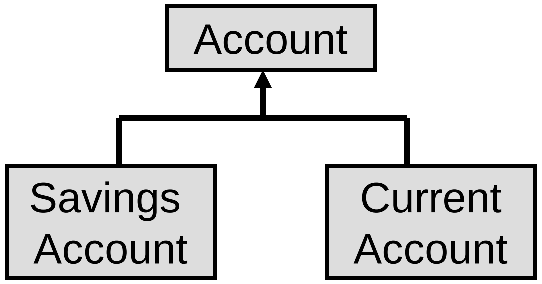
Inheritance

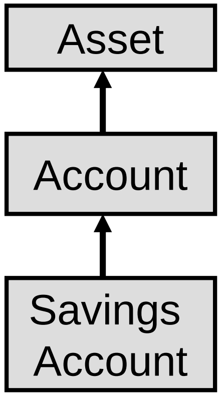
- Inheritance
  - It is a powerful technique that enables code reuse resulting in increased productivity, and reduced development time.
  - It allows for designing extensible software.

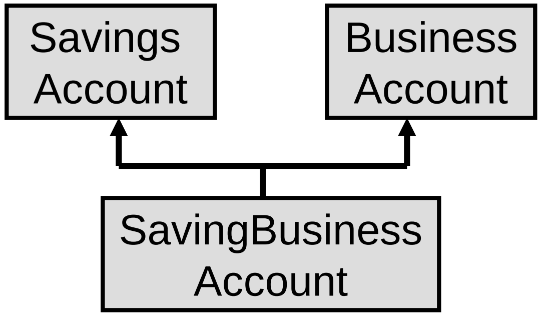


- Inheritance is the process of creating new classes, called *derived classes*, from existing or base classes. The derived class inherits all the capabilities of the base class, but can add some specificity of its own. The base class is unchanged by this process.

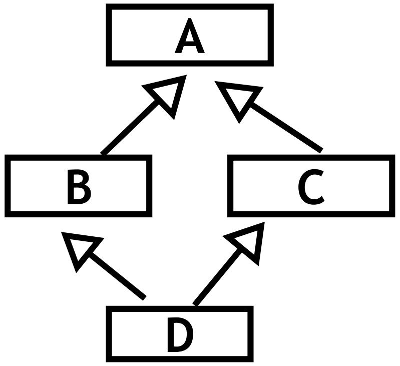
Types of Inheritance





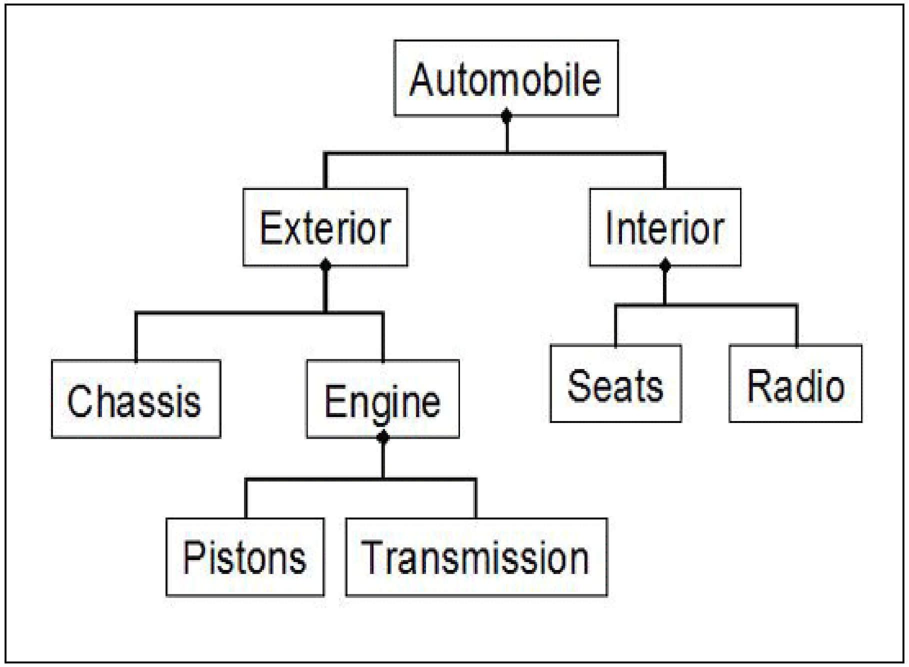


Multiple inheritance challenges: A name conflict introduced by a shared super-class (A) of super-classes (B and C) used with “multiple inheritance”.



Object Hierarchy

- Composition
- “Has-a” hierarchy is a relationship where one object “belongs” to (is a part or member of) another object, and behaves according to the rules of ownership.



Polymorphism

- It implies One Name, Many Forms.
- It is the ability to hide multiple implementations behind a single interface.
- Polymorphism allows different objects to respond to the same message in different ways!
- There are two types of Polymorphism, namely:
  - Static Polymorphism
  - Dynamic Polymorphism

Polymorphism

- Static Polymorphism:
  - The “Form” can be resolved at compile time, achieved through **overloading.**
    - For example: An operation “Sort” can be used for sorting integers, floats, doubles, or strings.

- Dynamic Polymorphism:
  - The “Form” can be resolved at run time, achieved through **overriding.**
    - For example: An operation “calculateInterest” for Accounts, where an Account can be of different types like Current or Savings.

Class and Object Implementation

```
#include <iostream>

using namespace std;

class Box
{
   public:
      double length;   // Length of a box
      double breadth;  // Breadth of a box
      double height;   // Height of a box
};

int main( )
{
   Box Box1;        // Declare Box1 of type Box
   Box Box2;        // Declare Box2 of type Box
   double volume = 0.0;     // Store the volume of a box here


   ……
```

Class and Object Implementation

```
// box 1 specification
   Box1.height = 5.0;
   Box1.length = 6.0;
   Box1.breadth = 7.0;

   // box 2 specification
   Box2.height = 10.0;
   Box2.length = 12.0;
   Box2.breadth = 13.0;
   // volume of box 1
   volume = Box1.height * Box1.length * Box1.breadth;
   cout << "Volume of Box1 : " << volume <<endl;

   // volume of box 2
   volume = Box2.height * Box2.length * Box2.breadth;
   cout << "Volume of Box2 : " << volume <<endl;
   return 0;
}
```

Inheritance Implementation

```
#include <iostream>

using namespace std;

// Base class
class Shape
{
   public:
      void setWidth(int w)
      {
         width = w;
      }
      void setHeight(int h)
      {
         height = h;
      }
   protected:
      int width;
      int height;
};
```

Inheritance Implementation

```
// Derived class
class Rectangle: public Shape
{
   public:
      int getArea()
      {
         return (width * height);
      }
};

int main(void)
{
   Rectangle Rect;

   Rect.setWidth(5);
   Rect.setHeight(7);

   // Print the area of the object.
   cout << "Total area: " << Rect.getArea() << endl;

   return 0;
}
```

Function Overloading

- Two or more methods within the same class share the *same* name.
- They should differ either in number of arguments or types of arguments.
- Java implements compile-time polymorphism by:
  - Overloading Constructors
  - Overloading Normal Methods

Function Overloading

```
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;
      }
};
```

Function Overloading

```
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;
}
```

Function Overriding

- In a class hierarchy, when a method in a subclass has the same *name* and *type signature* as a method in its super class, then the subclass method overrides the super class method.
- Overridden methods allow to support run-time polymorphism.
- Base class and derived class method has same signature.
- const methods can’t be overriden

Function Overriding

```
class Shape {
  protected:
    int width, height;
  public:
    Shape( int a=0, int b=0)
    {
      width = a;
      height = b;
    }
    virtual int area()
    {
      cout << "Parent class area :" <<endl;
      return 0;
    }
};
```

Function Overriding

```
class Rectangle: public Shape{
   public:
      Rectangle( int a=0, int b=0):Shape(a, b) { }
      int area ()      {
         cout << "Rectangle class area :" <<endl;
         return (width * height);
      } };
class Triangle: public Shape{
   public:
      Triangle( int a=0, int b=0):Shape(a, b) { }
      int area ()      {
         cout << "Triangle class area :" <<endl;
         return (width * height / 2);
      }
};
```

Function Overriding

```
int main( )
{
   Shape *shape;
   Rectangle rec(10,7);
   Triangle  tri(10,5);

   // store the address of Rectangle
   shape = &rec;
   // call rectangle area.
   shape->area();

   // store the address of Triangle
   shape = &tri;
   // call triangle area.
   shape->area();
```

Virtual Functions

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

class Rectangle: public Polygon {
  public:
    int area ()
      { return width * height; }
};
```

Virtual Functions

```
class Triangle: public Polygon {
  public:
    int area ()      { return (width * height / 2); }
};
int main () {
  Rectangle rect;   Triangle trgl;
  Polygon poly;
  Polygon * ppoly1 = &rect;
  Polygon * ppoly2 = &trgl;
  Polygon * ppoly3 = &poly;
  ppoly1->set_values (4,5);
  ppoly2->set_values (4,5);
  ppoly3->set_values (4,5);
  cout << ppoly1->area() << '\n';
  cout << ppoly2->area() << '\n';
  cout << ppoly3->area() << '\n';
  return 0;
}
```

Abstract Classes

- Abstract Method
  - Methods do not have implementation.
  - Methods declared in interfaces are always abstract.
    - Example:

```
virtual int area (void) =0;
```

- Abstract class
  - Provides common behavior across a set of subclasses.
  - Not designed to have instances that work.
  - One or more methods are declared but may not be defined
  - Advantages:
    - Code reusability.
    - Help at places where implementation is not available.

Abstract classes

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

Abstract classes

```
class Triangle: public Polygon {
  public:
    int area (void)    { return (width * height / 2); }
};
int main () {
  Rectangle rect;
  Triangle trgl;
  Polygon * ppoly1 = &rect;
  Polygon * ppoly2 = &trgl;
  ppoly1->set_values (4,5);
  ppoly2->set_values (4,5);
  cout << ppoly1->area() << '\n';
  cout << ppoly2->area() << '\n';
  return 0;
}
```

Operator Overloading

```
class Sample
{   int n;
    public:
    Sample(){}
    Sample(int i){n=i;}
    Sample operator+(int);
    void disp() {cout<<"n="<<n<<endl;}
};
Sample Sample::operator + (int x)
{
 Sample temp;
  temp.n = n + x;
  return temp;
}
```

Operator Overloading

```
int main()
{
  Sample s1(10),s2,s3;
  s1.disp();
  s2=s1.operator +(10);
  s2.disp();
  s3 = s1 +10; //s3=s1.operator+(10)
  s3.disp();




}
```

Operator Over/Copy constructor/this

```
class Sample
{
  int n;
Public:
  Sample(){}
  Sample(int i){n=i;}
  //  Sample(const Sample&);
  Sample& operator=(const Sample&);
  void disp() {std::cout << "n=" << n << std::endl;}
};
Sample& Sample::operator=(const Sample& s2){
  n = s2.n;
  std::cout<<"using =  "<< n << std::endl;
  return *this;
}
```

Operator Overloading

```
// Sample::Sample(const Sample& s2){
//   n = s2.n;
//   std::cout<<"using copy constructor "<< n << std::endl;
// }
int main()
{
  Sample s1(10),s2(20),s3;
  s1.disp();
  s1.operator=(s2);
  s1.disp();
  s1=s2;
  s1.disp();
 // s3=s2=s1; // (s3.operator =(s2.operator =(s1))
  //  std::cout<<"now let's look at s4 " << std::endl;
  //  Sample s4 = s1;
```

Exceptions

- Exception
- throw

```
try{

}catch(){

}
```

- Creating user-define exceptions

Exception

```
int main () {
  char myarray[10];
  try
  {
    for (int n=0; n<=10; n++)    {
      if (n>9) throw "Out of range";
      myarray[n]='z';
      cout<<myarray[n]<<" ";
    }
  }
  catch (char const * str)  {
    cout << "Exception: " << str << endl;
  }
  return 0;
}
```

Throwing Exceptions

```
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;
}
```

User-defined exception

```
#include <exception>
using namespace std;
struct MyException : public exception{
  const char * what () const throw ()  {
    return "C++ Exception";
  }
};
int main(){
  try  {
    throw MyException();
  }
  catch(MyException& e)  {
    std::cout << "MyException caught" << std::endl;
    std::cout << e.what() << std::endl;
  }
  catch(std::exception& e)
  {
    //Other errors
  }
}
```