



**SRM**  
INSTITUTE OF SCIENCE & TECHNOLOGY  
Dedicated to the University of Jaffna, Sri Lanka

# 18CSC202J - OBJECT ORIENTED DESIGN AND PROGRAMMING

## ACCESS SPECIFIERS

# Access control in classes

- **public :** A public member is accessible from anywhere outside the class but within a program.
- **private :** A private member variable or function cannot be accessed, or even viewed from outside the class. Only the class and friend functions can access private members.
- **protected :** A protected member variable or function is very similar to a private member but it provided one additional benefit that they can be accessed in child classes which are called derived classes

# Access Specifiers Example 1

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

class Line
{
public:
    double length;
    void setLength( double len );
    double getLength( void );
};

double Line::getLength(void)
{
    return length ;
}

void Line::setLength( double len )
{
    length = len;
}
```

```
// Main function for the program

int main( )
{
    Line line;

    // set line length
    line.setLength(6.0);
    cout << "Length of line : " << line.getLength() << endl;

    // set line length without member function

    line.length = 10.0; // OK: because length is public
    cout << "Length of line : " << line.length << endl;
    return 0;
}
```

```
Length of line : 6
Length of line : 10
```

# Access Specifiers Example 2

```
#include <iostream>

using namespace std;

class Box
{
public:
    double length;
    void setWidth( double wid );
    double getWidth( void );

private:
    double width;
};

// Member functions definitions
double Box::getWidth(void)
{
    return width ;
}

void Box::setWidth( double wid )
{
    width = wid;
}
```

```
// Main function for the program
int main( )
{
    Box box;

    // set box length without member function
    box.length = 10.0; // OK: because length is public
    cout << "Length of box : " << box.length << endl;

    // set box width without member function
    // box.width = 10.0; // Error: because width is private
    box.setWidth(10.0); // Use member function to set it.
    cout << "Width of box : " << box.getWidth() << endl;

    return 0;
}
```

Length of box : 10  
Width of box : 10

# Access Specifiers Example 3

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

class Box
{
protected:
    double width;
};

class SmallBox:Box
{
public:
    void setSmallWidth( double wid );
    double getSmallWidth( void );
};

// Member functions of child class
double SmallBox::getSmallWidth(void)
{
    return width ;
}
```

```
void SmallBox::setSmallWidth( double wid )
{
    width = wid;
}

// Main function for the program
int main( )
{
    SmallBox box;

    // set box width using member function
    box.setSmallWidth(5.0);
    cout << "Width of box : " << box.getSmallWidth();

    return 0;
}
```

Width of box : 5

# Friend function

- 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.

# Friend function Example

```
#include <iostream>
using namespace std;
class XYZ
{
private: int num=100;
char ch='Z';
public: friend void disp(XYZ obj);
};

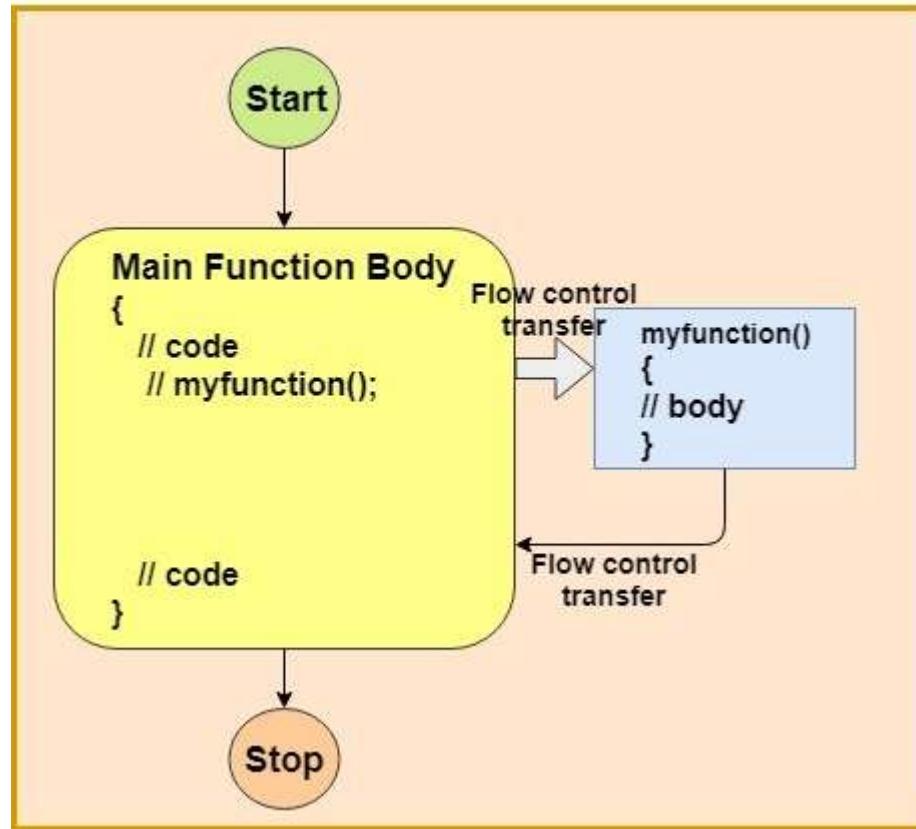
//Global Function
void disp(XYZ obj)
{
cout<<obj.num<<endl;
cout<<obj.ch<<endl;
}
int main()
{
XYZ obj;
disp(obj);
return 0;
}
```

# Inline Function

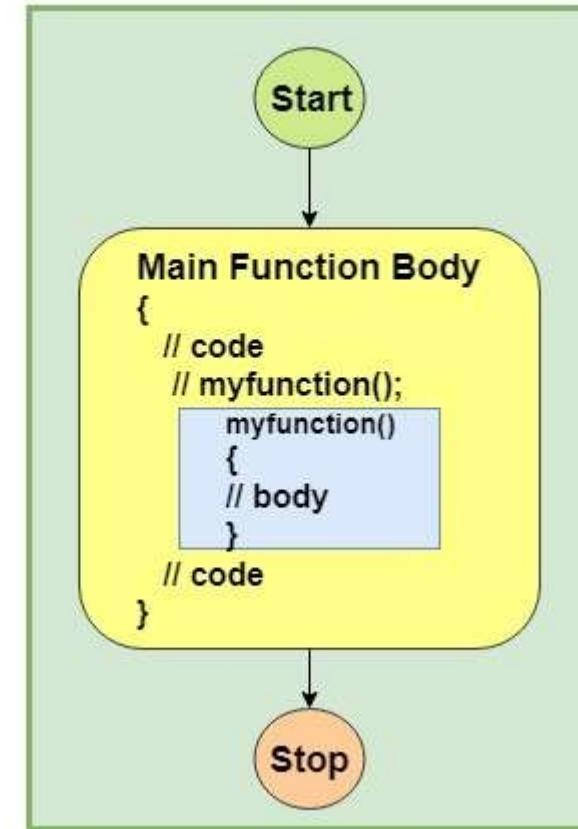
- C++ provides an inline functions to reduce the function call overhead.
- Inline function is a function that is expanded in line when it is called.
- When the inline function is called whole code of the inline function gets inserted or substituted at the point of inline function call. This substitution is performed by the C++ compiler at compile time.
- Inline function may increase efficiency if it is small.

# Inline Function - Example

Normal Function



Inline Functions



# Inline Function - Example

```
include <iostream>
using namespace std;
inline int Max(int x, int y) {
    return (x > y)? x : y;
}
// Main function for the program
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;
}
```

## Output:

Max (20,10): 20  
Max (0,200): 200  
Max (100,1010): 1010

*Thank  
you*



# Class and objects

## Class:

- Class is a user defined data type,
- It holds its own data members and member functions,
- It can be accessed and used by creating instance of that class.
- The variables inside class definition are called as data members and the functions are called member functions

## **example:**

Class of birds, all birds can fly and they all have wings and beaks. So here flying is a behavior and wings and beaks are part of their characteristics. And there are many different birds in this class with different names but they all posses this behavior and characteristics.

- class is just a blue print, which declares and defines characteristics and behavior, namely data members and member functions respectively.
- All objects of this class will share these characteristics and behavior.

- Class name must start with an uppercase letter(Although this is not mandatory).

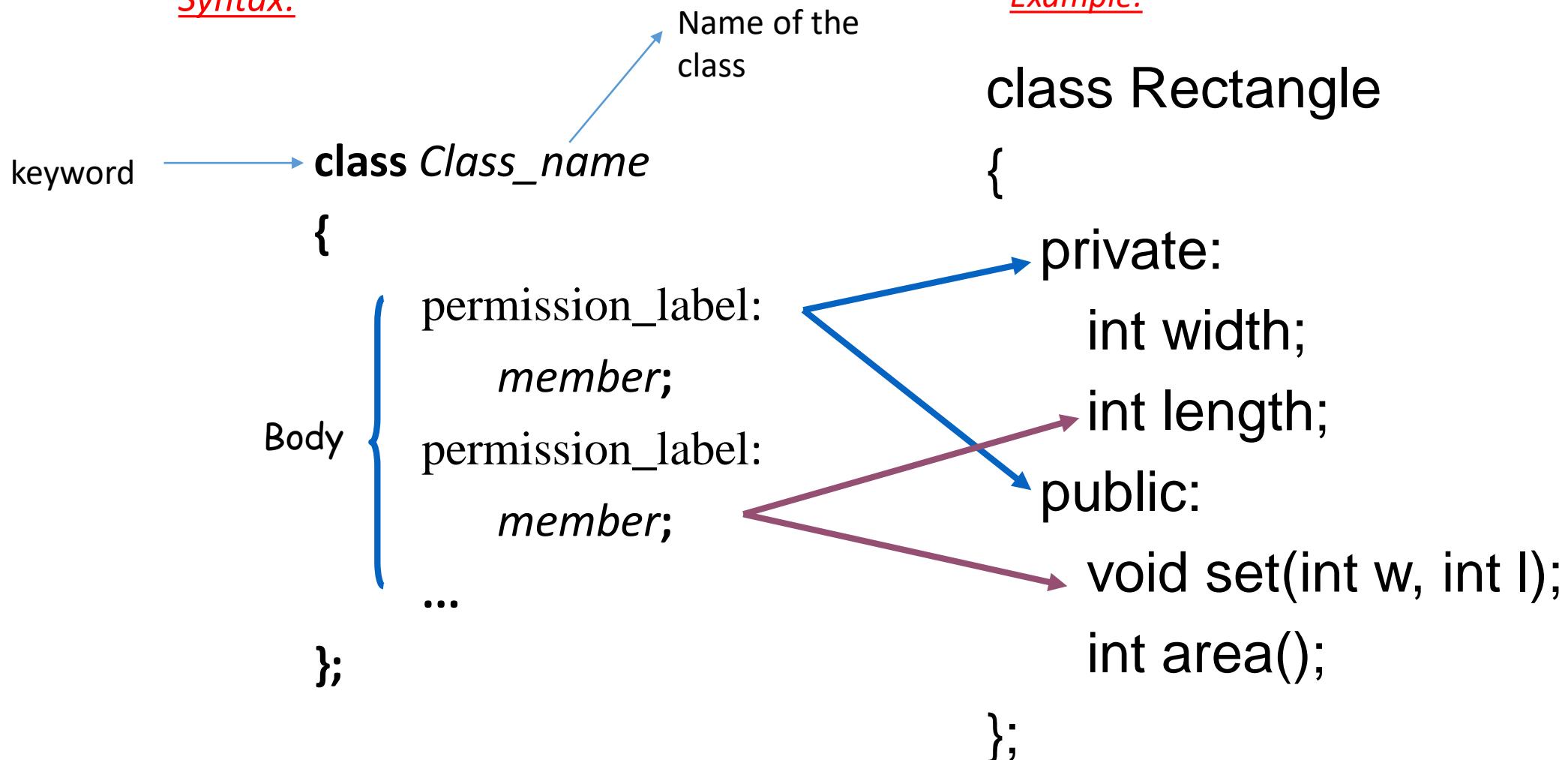
*Example,*

```
class Student
```

- Classes contain, data members and member functions, and the access of these data members and variable depends on the access specifiers.
- Class's member functions can be defined inside the class definition or outside the class definition.
- class defaults to private access control,
- Objects of class holds separate copies of data members. We can create as many objects of a class as we need.
- No storage is assigned when we define a class.

# Define a Class Type

Syntax:



- **Data members** Can be of any type, built-in or user-defined.

This may be,

- **non-static** data member

Each class object has its own copy

- **static** data member

Acts as a global variable

- **Static** data member is declared using the static keyword.
- There is only one copy of the static data member in the class. All the objects share the static data member.
- The static data member is always initialized to zero when the first class object is created.

*Syntax:*

*static data\_type datamember\_name;*

Here

**static** is the keyword.

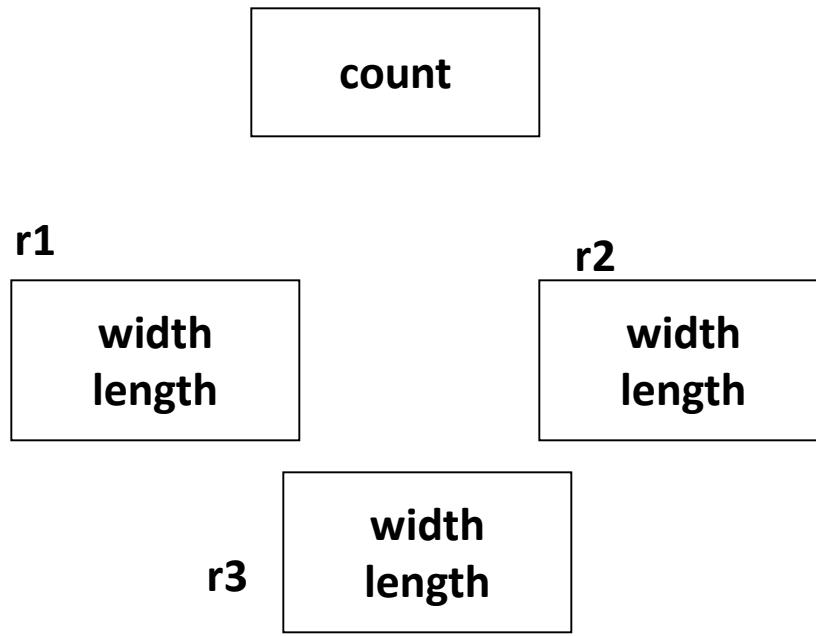
**data\_type** – int , float etc...

*datamember\_name – user defined*

# Static Data Member

```
class Rectangle
{
    private:
        int width;
        int length;
    ➔ static int count;
    public:
        void set(int w, int l);
        int area();
}
```

```
Rectangle r1;
Rectangle r2;
Rectangle r3;
```



# Member Functions

- Used to
  - access the values of the data members (accessor)
  - perform operations on the data members (implementor)
- Are declared inside the class body
- Their definition can be placed inside the class body, or outside the class body
- Can access both public and private members of the class
- Can be referred to using dot or arrow member access operator

# Member function

```
keyword          Class Name
class Rectangle
{
    private:
        int width, length;
    public:
        void set (int w, int l);
        int area()
        {
            return width*length;
        }
    }r1;
void Rectangle :: set (int w, int l)
{
    width = w;
    length = l;
}
```

**member function**

**inside the class**

**Object creation**

**class name**

**Member function**

**outside the class**

**scope operator**

```
int main()
{
    Rectangle r2;
    r1.set(5,8);
    int x=r1.area();
    cout<<"x value in r1 "<<x;
    r2.set(5,8);
    int x=r2.area();
    cout<<"x value in r2 "<<x;
}
```

**Object creation**

**Inside the function**

## **const** member function

- The const member functions are the functions which are declared as constant in the program.
- The object called by these functions cannot be modified.
- It is recommended to use const keyword so that accidental changes to object are avoided.
- A const member function can be called by any type of object.
- *Note: Non-const functions can be called by non-const objects only.*

*Syntax:*

datatype **function\_name const();**

User defined

keyword

# Const Member Function

```
class Time
{
private :
    int hrs, mins, secs ;
```

function declaration

```
public :
    void Write( ) const ;
```

function definition

```
} ;
```

```
void Time :: Write( ) const
{
    cout << hrs << ":" << mins << ":" << secs << endl;
}
```

# Access modifiers

- The access modifiers are used to set boundaries for availability of members of class be it data members or member functions
- Access modifiers in the program, are followed by a colon.
- we can use either one, two or all 3 modifiers in the same class to set different boundaries for different class members.

C++ has the following types of access modifiers,

1. **public**
2. **private**
3. **protected**

## Public:

- Public class members declared under **public** will be available to everyone.
- The data members and member functions can be accessed by other classes too.  
Hence there are chances that they might change them.  
(So the key members must not be declared public).

```
class Student
{
    // public access modifier
    public:
        int x;          // Data Member Declaration
        void display(); // Member Function decaration
}
```

## Private :

- No one can access the class members which are declared **private**, outside that class.
- When try to access the private members of a class, it will show a **compile time error**.
- The default class variables and member functions are private.

## Protected:

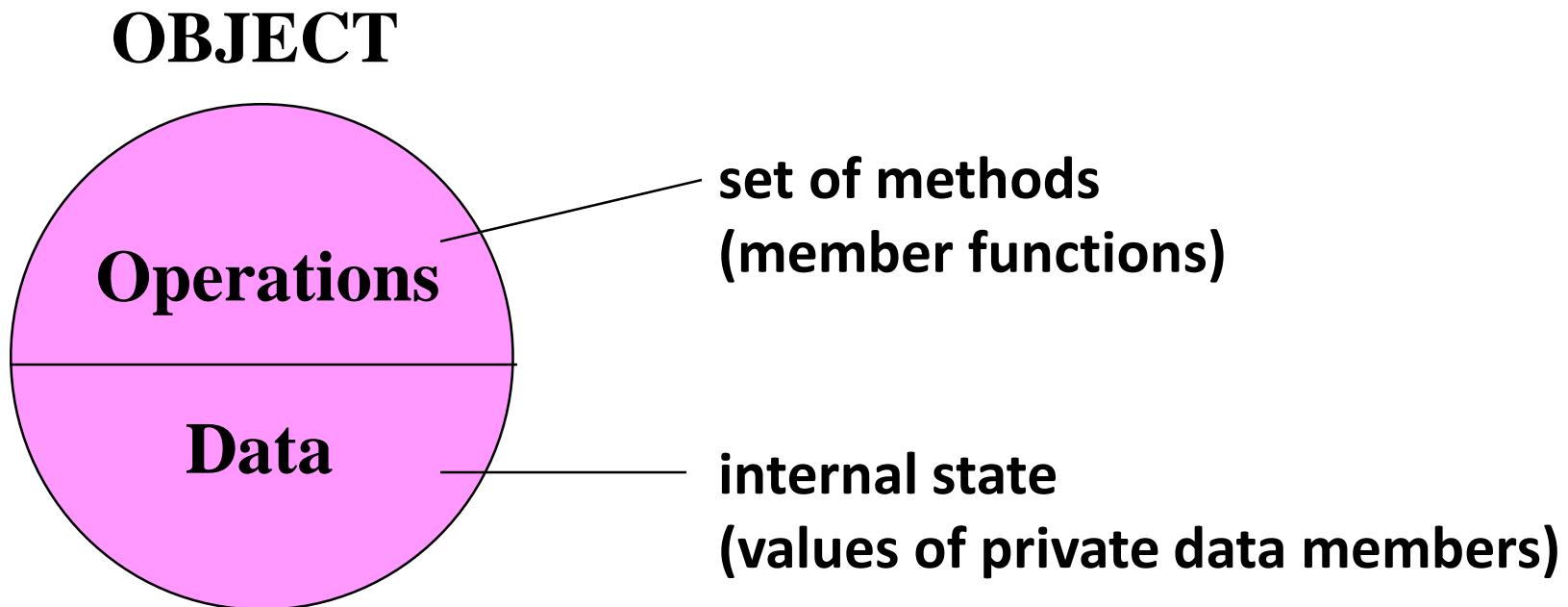
- it is similar to private, it makes class member inaccessible outside the class.  
But they can be accessed by any subclass of that class.

## Example:

class A is **inherited** by class B, then class B is subclass of class A.

so class B can access the class A data members and member functions

# What is an object?



## Object:

- Objects are instances of class, which holds the data variables declared in class and the member functions work on these class objects.
- Each object has different data variables.
- Objects are initialized using special class functions called **Constructors**.
- **Destructor** is special class member function, to release the memory reserved by the object.

## Syntax of creating object :

*Class\_name object\_name;*

# Declaration of an Object

```
class Rectangle
{
    private:
        int width;
        int length;
    public:
        void set(int w, int l);
        int area();
};
```

```
main()
{
    Rectangle r1;
    Rectangle r2;

    r1.set(5, 8);
    cout<<r1.area()<<endl;

    r2.set(8,10);
    cout<<r2.area()<<endl;
}
```

## Another Example

```
#include <iostream.h>

class circle
{
private:
    double radius;

public:
    void store(double);
    double area(void);
    void display(void);
};
```

```
// member function definitions

void circle::store(double r)
{
    radius = r;
}

double circle::area(void)
{
    return 3.14*radius*radius;
}

void circle::display(void)
{
    cout << "r = " << radius << endl;
}
```

```
int main(void) {
    circle c; // an object of circle class
    c.store(5.0);
    cout << "The area of circle c is " << c.area() << endl;
    c.display();
}
```

# Declaration of an Object

```
class Rectangle
{
    private:
        int width;
        int length;
    public:
        void set(int w, int l);
        int area();
};
```

**r1 is statically allocated**

```
main()
{
    Rectangle r1;
    ➔ r1.set(5, 8);
}
```

**r1**

**width = 5  
length = 8**

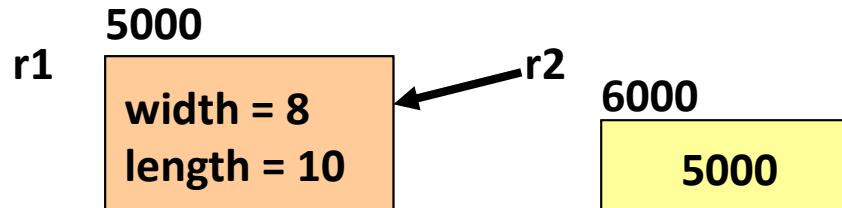
# Declaration of an Object

```
class Rectangle
{
    private:
        int width;
        int length;
    public:
        void set(int w, int l);
        int area();
};
```

r2 is a pointer to a Rectangle object

```
main()
{
    Rectangle r1;
    r1.set(5, 8);      //dot notation

    Rectangle *r2;
    r2 = &r1;
    r2->set(8,10);   //arrow notation
}
```



# Object Initialization

## 1. By Assignment

```
#include <iostream.h>

class circle
{
public:
    double radius;
};
```

```
int main()
{
    circle c1;          // Declare an instance of the class circle
    c1.radius = 5;      // Initialize by assignment

}
```

- Only work for public data members
- No control over the operations on data members

# Object Initialization

## 2. By Public Member Functions

```
#include <iostream.h>

class circle
{
    private:
        double radius;

    public:
        void set (double r)
            {radius = r;}
        double get_r ()
            {return radius;}
};
```

```
int main(void) {
    circle c;           // an object of circle class
    c.set(5.0);         // initialize an object with a public member function
    cout << "The radius of circle c is " << c.get_r() << endl;
                           // access a private data member with an accessor
}
```

# 18CSC202J - OBJECT ORIENTED DESIGN AND PROGRAMMING

## Session 12

**Topic : Constructor and Destructor**

# Object Initialization

## By Constructor

- Default constructor
- Copy constructor
- Constructor with parameters

## **Constructor**

- It is a special kind of class member function that is automatically called when an object of that class is instantiated.
- Constructors are typically used to initialize member variables of the class to appropriate default or user-provided values.

**constructors have specific rules for how they must be named:**

- Constructors must have the same name as the class (with the same capitalization) .Constructors have no return type (not even void)

Example for default constructor:

```
#include <iostream>
using namespace std;
class Employee
{
public:
    Employee()
    {
        cout<<"Default Constructor Invoked" << endl;
    }
};
int main(void)
{
    Employee e1; //creating an object of Employee
    Employee e2;
    return 0;
}
```

### **Constructor with parameter:**

- A constructor which has parameters is called parameterized constructor.
- It is used to provide different values to distinct objects.

## Example for constructor with parameter

```
#include <iostream>
using namespace std;
class Employee {
public:
    int id;//data member (also
instance variable)
    string name;//data
member(also instance variable)
    float salary;
    Employee(int i, string n, float
s)
    {
        id = i;
        name = n;
        salary = s;
    }
    void display()
    {
        cout<<id<<" "<<name<<""
"<<salary<<endl;
    }
};
```

```
int main(void) {
    Employee e1 =Employee(101, "Sonoo", 890000);
    e2=Employee(102, "Nakul", 59000);
    e1.display();
    e2.display();
    return 0;
}
```

Creating an  
object of  
Employee

## **Copy constructor:**

- Copy Constructor is a type of constructor which is used to create a copy of an already existing object of a class type.
- The compiler provides a default Copy Constructor to all the classes.

## **Copy Constructor is called in the following scenarios:**

- When we initialize the object with another existing object of the same class type.

For example, `Student s1 = s2`, where `Student` is the class.

- When the object of the same class type is passed by value as an argument.
- When the function returns the object of the same class type by value.

## Example for copy constructor

```
#include <iostream>
using namespace std;
class A
{
public:
    int x;
    A(int a)      // parameterized
constructor.
    {
        x=a;
    }
    A(A &i)      // copy constructor
    {
        x = i.x;
    }
};
```

```
int main()
{
    A a1(20);          // Calling the
parameterized constructor.
    A a2(a1);         // Calling the
copy constructor.
    cout<<a2.x;
    return 0;
}
```

## Destructor:

- Destructors have the same name as their class and their name is preceded by a tilde(~).
- Destructors in C++ are members functions in a class that delete an object.
- They are called when the class object goes out of scope such as when the function ends, the program ends, a delete variable is called etc.
- Destructors are don't take any argument and don't return anything.

## **When does the destructor get called?**

A destructor is **automatically called** when:

- 1) The program finished execution.
- 2) When a scope (the { } parenthesis) containing **local variable** ends.
- 3) When you call the delete operator.

## Example program for destructor

```
#include <iostream>
using namespace std;
class HelloWorld{
public:
    HelloWorld(){
        cout<<"Constructor is
called"<<endl;
    }
    ~HelloWorld(){
        cout<<"Destructor is
called"<<endl;
    }
    //Member function
    void display(){
        cout<<"Hello World!"<<endl;
    }
};
```

Constructor

Destructor

```
int main()
{
    HelloWorld obj;
    //Object created
    //Member function
    obj.display();
    return 0;
}
```

### output

Constructor is called  
Hello World!  
Destructor is called

## **Destructor rules**

- 1) Name should begin with tilde sign(~) and must match class name.
- 2) There cannot be more than one destructor in a class.
- 3) Unlike constructors that can have parameters, destructors do not allow any parameter.
- 4) They do not have any return type, just like constructors.
- 5) When you do not specify any destructor in a class, compiler generates a default destructor and inserts it into your code.

# Constructor

- A constructor is a member function that is invoked automatically when an object is declared.
- It has the same name as the class and carries no return type (not even void).
- The Compiler calls the Constructor whenever an object is created

Syntax:

```
class A
{
    public:
        A() // constructor
    {
    };
}
```

# Constructor

- Constructor can be defined either inside the class or outside of the class using scope resolution operator ::

## Example

```
class A
{
    int a;
public:
    A(); // constructor declaration
};

A::A()// constructor definition
{
    a = 10;
}
```

# Methods

- A *method* is a function that is part of the class definition. The methods of a class specify how its objects will respond to any particular message.
- A method is a collection of statements that perform some specific task and return the result to the caller. A method can perform some specific task without returning anything.
- Methods allow us to reuse the code without retyping the code.
- A *message* is a request, sent to an object, that activates a method (i.e., a member function).

# Method Vs Constructor

- Used to define particular task for execution
- Method can have same name as class name or different name based on the requirement
- Explicit call statement required to call a method
- There is no default method provided by compiler
- Return data type must be declared
- Used to initialize the data members
- Constructor has same name as the class name
- Constructor is automatically called when an object is created
- There is always default constructor provided by compiler
- Return type is not required in constructor

# Types of Constructor

## Types of Constructors in C++

Constructors are of three types:

- **Default Constructor** - Default constructor is the constructor which doesn't take any argument. It has no parameter.
- **Parametrized Constructor** : These are the constructors with parameter. Using this Constructor you can provide different values to data members of different objects, by passing the appropriate values as argument.
- **Copy Constructor**: These are special type of Constructors which takes an object as argument, and is used to copy values of data members of one object into other object.

# Example Program

```
class Student
{
public:
    int rollno;
    string name;
    Student() //Default constructor
    {
        rollno = 0;
        name = "None";
    }
    Student(int x) // parameterized
    {
        rollno = x;
        name = "None";
    }
}

// copy constructor
Student (const Student &s)
{
    rollno = s.rollno;
    name = s.name;
}
};

int main()
{
    Student A1();
    Student A(10);
    Student B=A;
}
```

# Destructor

- A destructor is a special member function of a class that is executed whenever an object of its class goes out of scope or whenever the delete expression is applied to a pointer to the object of that class.

Example  
class Line

```
{  
public:  
    st1.....  
    st2.....  
    Line();  
    ~Line(); // This is the destructor: declaration  
};
```



# **18CSC202J - OBJECT ORIENTED DESIGN AND PROGRAMMING**

## **Session 8**

**Topic : Feature Abstraction and  
Encapsulation, Application of Abstraction and  
Encapsulation**

# Features Of OOPS

- **DATA ENCAPSULATION**
  - Combining data and functions into a single unit called **class** and the process is known as **Encapsulation**.
- Provides **security**, **flexibility** and **easy maintainability**
- Encapsulation helps us in **binding the data**(*instance variables*) and the **member functions**(that work on the instance variables) of a class.
- **Encapsulation** is also useful in **hiding the data**(*instance variables*) of a class from an *illegal direct access*.
- **Encapsulation** also helps us to make a **flexible code** which is easy to *change and maintain*.

# Features Of OOPS

- **ABSTRACTION OR DATA HIDING**
  - 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 program is called **Abstraction** and only the essential features are represented.
- Uses
  - 1) Makes the application secure by making data private and avoiding the user level error that may corrupt the data.
  - 2) This avoids code duplication and increases the code reusability.

# Application of Abstraction

- For example, when you send an email to someone you just click send and you get the success message, what actually happens when you click send, how data is transmitted over network to the recipient is hidden from you

# ENCAPSULATION

## ADVANTAGES

- **Encapsulated classes reduce complexity.**
- **Help protect our data.** A client cannot change an Account's balance if we encapsulate it.
- **Encapsulated classes are easier to change.**

## Example

- The common example of encapsulation is **Capsule**. In capsule all medicine are encapsulated in side capsule.
- **Automatic Cola Vending Machine** :Suppose you go to an automatic cola vending machine and request for a cola. The machine processes your request and gives the cola.

# Examples

## Data Abstraction and Encapsulation

Any C++ program where you implement a class with public and private members is an example of data encapsulation and data abstraction

- #include <iostream>
- using namespace std;
- class Adder {
- public:
- // constructor
- Adder(int i = 0) {
- total = i;
- }
- // interface to outside world
- void addNum(int number) {
- total += number;
- }
- // interface to outside world
- int getTotal() {
- return total;

*Thank  
you*





# PROGRAMMING

Feature : Objects and  
Classes

# Objects and Classes

- Class is a user defined data type, which holds its own data members and member functions, which can be accessed and used by creating instance of that class.
- **Attributes** – member of a class.
  - An attribute is the data defined in a class that maintains the current state of an object. The state of an object is determined by the current contents of all the attributes.
- **Methods** – member functions of a class

# Objects and classes

- A class itself does not exist; it is merely a description of an object.
- A class can be considered a template for the creation of object
- Blueprint of a building → class
- Building → object
- An object exists and is definable.
- An object exhibits behavior, maintains state, and has traits. An object can be manipulated.
- An object is an instance of a class.

# Methods

- A *method* is a function that is part of the class definition. The methods of a class specify how its objects will respond to any particular message.
- A method is a collection of statements that perform some specific task and return the result to the caller. A method can perform some specific task without returning anything.
- Methods allow us to reuse the code without retyping the code.
- A *message* is a request, sent to an object, that activates a method (i.e., a member function).

# Defining a Base Class - Example

- A base class is not defined on, nor does it inherit members from, any other class.

```
#include <iostream>
using std::cout;    //this example "uses" only the necessary
using std::endl;    // objects, not the entire std namespace
```

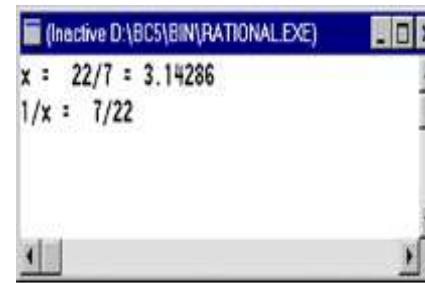
```
class Fraction {
public:
    void assign (int, int);           //member functions
    double convert();
    void invert();
    void print();
private:
    int num, den;                  //member data
};
```

**Continued...**

# Using objects of a Class - Example

## The main() function:

```
int main()
{
    Fraction x;
    x.assign(22, 7);
    cout << "x = "; x.print();
    cout << " = " << x.convert() << endl;
    x.invert();
    cout << "1/x = "; x.print(); cout << endl;
    return 0;
}
```



Continued...

# Defining a Class – Example

Class Implementation:

```
void Fraction::assign (int numerator, int denominator)
{
    num = numerator;
    den = denominator;
}

double Fraction::convert ()
{
    return double (num)/(den);
}

void Fraction::invert()
{
    int temp = num;
    num = den;
    den = temp;
}

void Fraction::print()
{
    cout << num << '/' <
}< den;
```

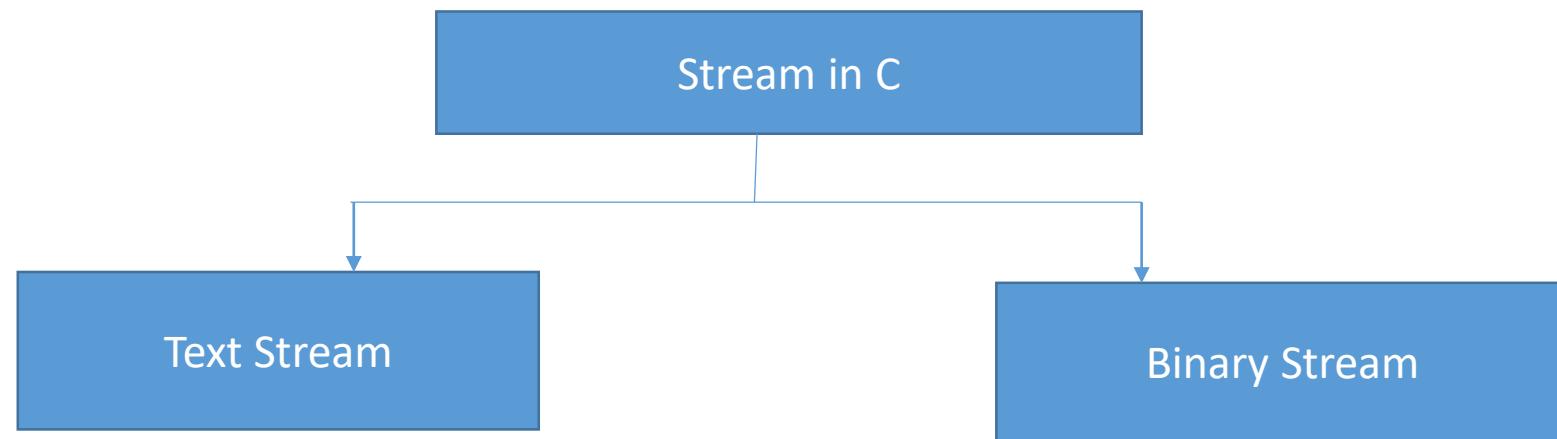
# Session-2

# Topics covered

- I/O Operation
- Data types
- Variable
- Static
- Constant
- Pointer
- Type conversion

# I/O Operation

- C++ I/O operation occurs in streams, which involve transfer of information into byte
- It's a sequences of bytes
- Stream involved in two ways
- It is the source as well as the destination of data
- C++ programs input data and output data from a stream.
- Streams are related with a physical device such as the monitor or with a file stored on the secondary memory.
- In a text stream, the sequence of characters is divided into lines, with each line being terminated.
- With a new-line character (\n) . On the other hand, a binary stream contains data values using their memory representation.



# Cascading of Input or Output Operators

- << operator –It can use multiple times in the same line.
- Its called Cascading
- Cout ,Cin can cascaded
- For example
- cout<<“\n Enter the Marks”;
- cin>> ComputerNetworks>>OODP;

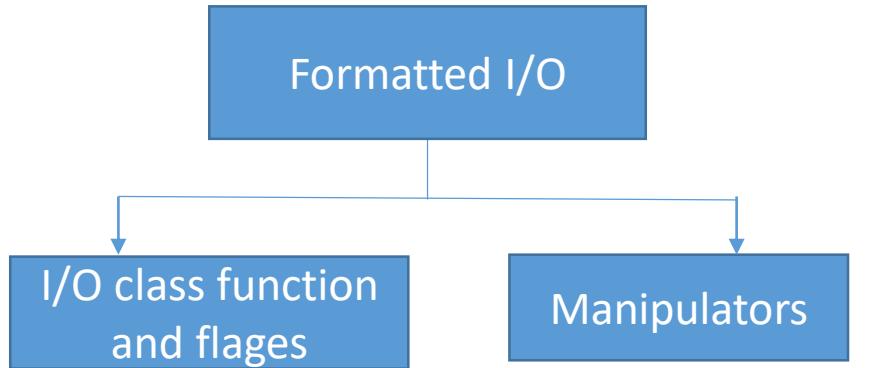
# Reading and Writing Characters and Strings

- char marks;
- cin.get(marks);//The value for marks is read
- OR
- marks=cin.get();//A character is read and assigned to marks
- string name;
- Cin>>name;
  
- string empname;
- cin.getline(empname,20);
- Cout<<“\n Welcome ,”<<empname;

# Formatted Input and Output Operations

Table 2.7 Functions for Input/Output

Function	Purpose	Syntax	Usage	Result	Comment
width()	Specifies field size (no. of columns) to display the value	cout.width(w)	cout.width(6);cout<<1239;	1 2 3 9	It can specify field width for only one value
precision()	Specifies no. of digits to be displayed after the decimal point of a float value	cout.precision(d)	cout.precision(3);cout<<1.234567;	1.234	It retains the setting in effect until it is reset
fill()	Specify a character to fill the unused portion of a field	cout.fill(ch)	Cout.fill('#');cout.width(6);cout<<1239;	# # 1 2 3 9	It retains the setting in effect until it is reset



```
#include<iostream.h>
#define PI 3.14159
main()
{
    cout.precision(3);
    cout.width(10);
    cout.fill('*');
    cout<<PI;
    Output
*****3.142
```

# Formatting with flags

- The setf() is a member function of the ios class that is used to set flags for formatting output.
- **syntax -cout.setf(flag, bit-field)**
- Here, flag defined in the ios class specifies how the output should be formatted and bit-field is a constant (defined in ios ) that identifies the group to which the formatting flag belongs to.
- There are two types of setf()—one that takes both flag and bit-fields and the other that takes only the flag .

Table 2.8 Types of setf()

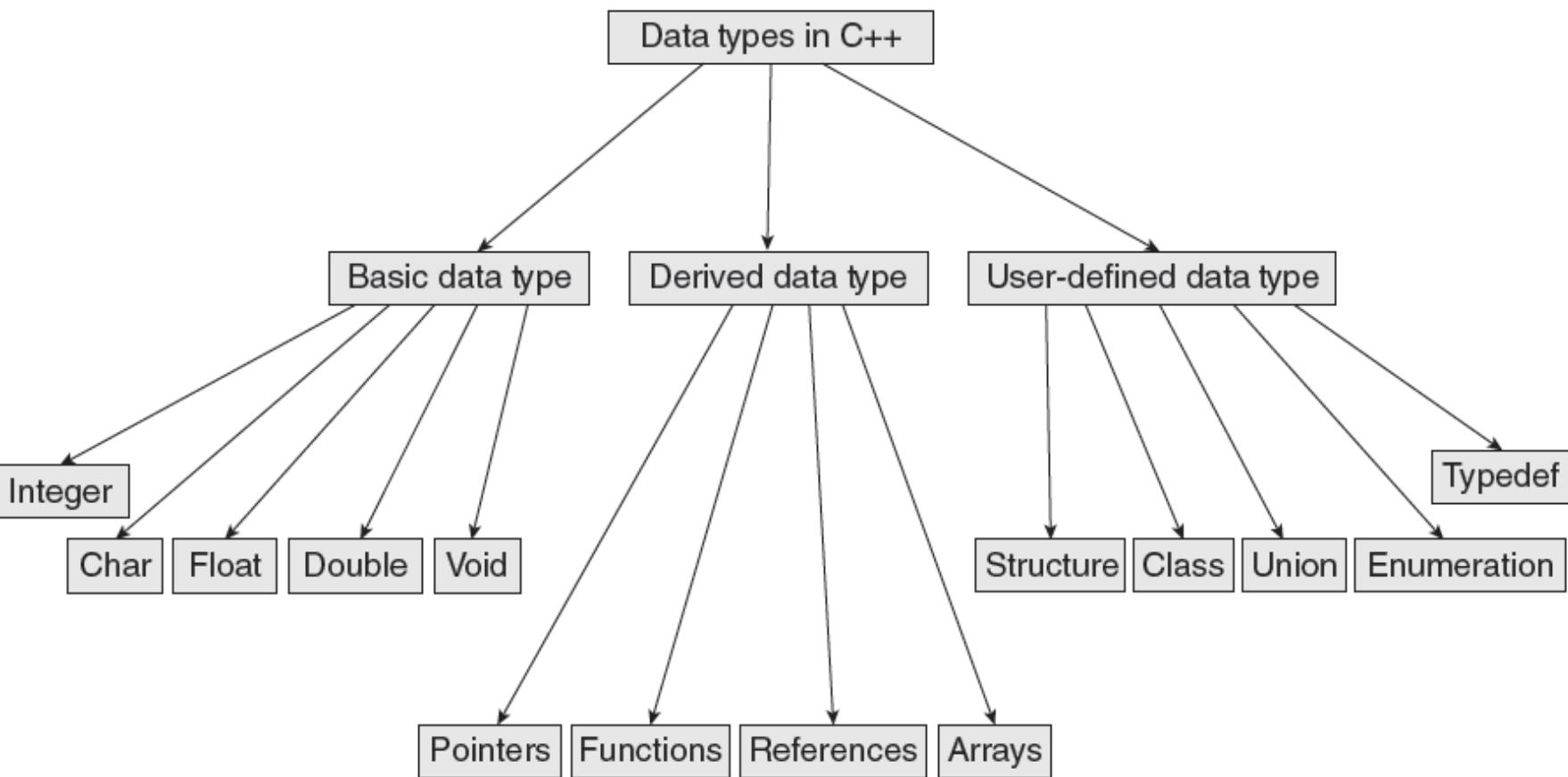
Flag	Bit-field	Usage	Purpose	Comment
ios :: left	ios :: adjustfield	<code>cout.setf(ios :: left, ios :: adjustfield)</code>	For left justified output	If no flag is set, then by default, the output will be right justified.
ios :: right	ios:: adjustfield	<code>cout.setf(ios :: right, ios :: adjustfield)</code>	For right justified output	
ios :: internal	ios:: adjustfield	<code>cout.setf(ios :: internal, ios :: adjustfield)</code>	Left-justify sign or base indicator, and right-justify rest of number	
ios :: scientific	ios:: floatfield	<code>cout.setf(ios :: scientific, ios :: floatfield)</code>	For scientific notation	If no flag is set then any of the two notations can be used for floating point numbers depending on the decimal point
ios :: fixed	ios :: floatfield	<code>cout.setf(ios :: fixed, ios :: floatfield)</code>	For fixed point notation	
ios :: dec	ios :: basefield	<code>cout.setf(ios :: dec, ios :: basefield)</code>	Displays integer in decimal	If no flag is set, then the output will be in decimal.
ios :: oct	ios :: basefield	<code>cout.setf(ios :: oct, ios :: basefield)</code>	Displays integer in octal	
ios :: hex	ios :: basefield	<code>cout.setf(ios :: hex, ios :: basefield)</code>	Displays integer in hexadecimal	
ios :: showbase	Not applicable	<code>cout.setf(ios :: showbase)</code>	Show base when displaying integers	
ios :: showpoint	Not applicable	<code>cout.setf(ios :: showpoint)</code>	Show decimal point when displaying floating point numbers	
ios :: showpos	Not applicable	<code>cout.setf(ios :: showpos)</code>	Show + sign when displaying positive integers	
ios :: uppercase	Not applicable	<code>cout.setf(ios :: uppercase)</code>	Use uppercase letter when displaying hexadecimal (OX) or exponential numbers (E)	

# Formatting Output Using Manipulators

- C++ has a header file `iomanip.h` that contains certain manipulators to format the output

Manipulator	Purpose	Usage	Result	Alternative						
<code>setw(w)</code>	Specifies field size (number of columns) to display the value	<code>cout&lt;&lt;setw(6) &lt;&lt;1239;</code>	<table border="1"><tr><td></td><td></td><td>1</td><td>2</td><td>3</td><td>9</td></tr></table>			1	2	3	9	<code>width()</code>
		1	2	3	9					
<code>setprecision(d)</code>	Specifies number of digits to be displayed after the decimal point of a float value	<code>cout&lt;&lt;setprecision(3) &lt;&lt;1.234567;</code>	1.235	<code>precision()</code>						
<code>setfill(c)</code>	Specify a character to fill the unused portion of a field	<code>cout&lt;&lt;setfill('#') &lt;&lt;setwidth(6) &lt;&lt;1239;</code>	<table border="1"><tr><td>#</td><td>#</td><td>1</td><td>2</td><td>3</td><td>9</td></tr></table>	#	#	1	2	3	9	<code>fill()</code>
#	#	1	2	3	9					

# Data Types in C++



Data type	Size in bytes	Range
Char	1	-128 to 127
unsigned char	1	0 to 255
signed char	1	-128 to 127
Int	2	-32768 to 32767
unsigned int	2	0 to 65535
signed short int	2	-32768 to 32767
signed int	2	-32768 to 32767
short int	2	-32768 to 32767
unsigned short int	2	0 to 65535
long int	4	-2147483648 to 2147483647
unsigned long int	4	0 to 4294967295
signed long int	4	-2147483648 to 2147483647
Float	4	3.4E-38 to 3.4E+38
Double	8	1.7E-308 to 1.7E+308
long double	10	3.4E-4932 to 1.1E+4932

# Variables

A variable is the content of a memory location that stores a certain value. A variable is identified or denoted by a variable name. The variable name is a sequence of one or more letters, digits or underscore, for example: character\_

## Rules for defining variable name:

- ❖ A variable name can have one or more letters or digits or underscore for example character\_.
- ❖ White space, punctuation symbols or other characters are not permitted to denote variable name.
- ❖ A variable name must begin with a letter.
- ❖ Variable names cannot be keywords or any reserved words of the C++ programming language.
- ❖ Data C++ is a case-sensitive language. Variable names written in capital letters differ from variable names with the same name but written in small letters.

01

Local Variables

02

Static Variables

03

Instance variables

04

Final Variables

# Variables

## Local Variables

**Local variable:** These are the variables which are declared within the method of a class.

Example:

```
public class Car {  
public:  
void display(int m){ //  
Method  
    int model=m;  
// Created a local variable  
model  
    cout<<model;  
}
```

## Instance Variables

**Instance variable:** These are the variables which are declared in a class but outside a method, constructor or any block.

Example:

```
public class Car {  
    private: String color;  
// Created an instance  
variable color  
Car(String c)  
{  
    color=c;  
}
```

## Static Variables

**Static variables:** Static variables are also called as class variables. These variables have only one copy that is shared by all the different objects in a class.

Example:

```
public class Car {  
    public static int tyres;  
// Created a class variable  
void init(){  
    tyres=4;  
} }
```

## Constant Variables

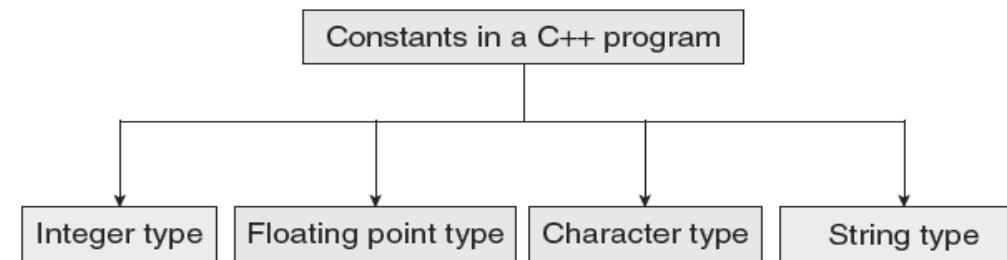
Constant is something that doesn't change. In C language and C++ we use the keyword const to make program elements constant.

Example:

```
const int i = 10;  
void f(const int i)  
class Test  
{  
    const int i;  
};
```

# CONSTANTS

- Constants are identifiers whose value does not change. While variables can change their value at any time, constants can never change their value.
- Constants are used to define fixed values such as Pi or the charge on an electron so that their value does not get changed in the program even by mistake.
- A constant is an explicit data value specified by the programmer.
- The value of the constant is known to the compiler at the compile time.



# Declaring Constants

- Rule 1 Constant names are usually written in capital letters to visually distinguish them from other variable names which are normally written in lower case characters.
- Rule 2 No blank spaces are permitted in between the # symbol and define keyword.
- Rule 3 Blank space must be used between #define and constant name and between constant name and constant value.
- Rule 4 #define is a preprocessor compiler directive and not a statement. Therefore, it does not end with a semi-colon.

```
const data_type const_name = value;
```

The const keyword specifies that the value of pi cannot change.

```
const float pi = 3.14;
```

```
#define PI 3.14159  
#define service_tax 0.12
```

# Data Operators

---

- 
- 1 Arithmetic Operators
  - 2 Unary operators
  - 3 Relational operators
  - 4 Logical Operators

# Arithmetic Operators

## 1 Arithmetic Operators

## 2 Unary operators

## 3 Relational operators:

## 4 Logical Operators

+

Add two operands or unary plus

```
>> 2 + 3  
5  
>> +2
```

-

Subtract two operands or unary subtract

```
>> 3 - 1  
2  
>> -2
```

\*

Multiply two operands

```
>> 2 * 3  
6
```

/

Divide left operand with the right and result is in float

```
>> 6 / 3  
2.0
```

%

Remainder of the division of left operand by the right

```
>> 5 % 3  
2
```

# Unary operators

1 Arithmetic Operators

2 Unary operators

3 Relational operators:

4 Logical Operators

++

X++ - Post Increment  
++X - Pre Increment

>> X = 5

>> ++x  
>> print(x)  
6

--

X-- - Post Decrement  
--X - Pre Increment

>> x--  
>> print(x)  
4

# Relational operators

1 Arithmetic Operators

2 Unary operators

3 Relational operators:

4 Logical Operators

>

True if left operand is greater than the right

>> 2 > 3  
False

<

True if left operand is less than the right

>> 2 < 3  
True

==

True if left operand is equal to right

>> 2 == 2  
True

!=

True if left operand is not equal to the right

>> x >>= 2  
>> print(x)  
1

# Logical operators

1 Arithmetic Operators

2 Unary operators

3 Relational operators:

4 Logical Operators

and

Returns True if both x and y are True,  
False otherwise

>> True  
&& False  
False

or

Returns True if at least x or y are True,  
False otherwise

>> True ||  
False  
True

not

Returns True if x is False, True otherwise

>> ! 1  
False

# Special Operators

## Scope resolution operator

1

In C, the global version of a variable cannot be accessed from within the inner block. C++ resolves this problem by using scope resolution operator (::), because this operator allows access to the global version of a variable.

## New Operator

2

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.

## Delete Operator

3

Since it is programmer's responsibility to deallocate dynamically allocated memory, programmers are provided delete operator by C++ language.

## Member Operator

4

C++ permits us to define a class containing various types of data & functions as members. To access a member using a pointer in the object & a pointer to the member.

# Operator Precedence

Rank	Operators	Associativity
1	<code>++, --, +, -, !(logical complement), ~(bitwise complement), (cast)</code>	Right
2	<code>*(mul), /(div), %(mod)</code>	Left
3	<code>+ , - (Binary)</code>	Left
4	<code>&gt;&gt;, &lt;&lt;, &gt;&gt;&gt; (Shift operator)</code>	Left
5	<code>&gt;, &lt;, &gt;=,&lt;=</code>	Left
6	<code>==, !=</code>	Left
7	<code>&amp; (Bitwise and)</code>	Left
8	<code>^ ( XOR)</code>	Left
9	<code>  (Bitwise OR)</code>	Left
10	<code>&amp;&amp; (Logical and)</code>	Left
11	<code>   (Logical OR)</code>	Left
12	<code>Conditional</code>	Right
13	<code>Shorthand Assignment</code>	Right

# Pointers

- Pointer is variable in C++
- It holds the address of another variable
- Syntax `data_type *pointer_variable;`
- Example `int *p,sum;`

## Assignment

- integer type pointer can hold the address of another int variable
- To assign the address of variable to pointer-**ampersand symbol (&)**
- `p=&sum;`

# How to use it

- P=&sum;//assign address of another variable
- cout<<&sum; //to print the address of variable
- cout<<p;//print the value of variable
- Example of pointer

```
#include<iostream.h>
using namespace std;
int main()
{ int *p,sum=10;
p=&sum;
cout<<"Address of sum:"<<&sum<<endl;
cout<<"Address of sum:"<<p<<endl;
cout<<"Address of p:"<<&p<<endl;
cout<<"Value of sum"<<*p;
}
```

Output:  
Address of sum : 0X77712  
Address of sum: 0x77712  
Address of p: 0x77717  
Value of sum: 10

# Pointers and Arrays

- assigning the address of array to pointer don't use ampersand sign(&)

```
#include <iostream>
using namespace std;
int main(){
    //Pointer declaration
    int *p;
    //Array declaration
    int arr[]={1, 2, 3, 4, 5, 6};
    //Assignment
    p = arr;
    for(int i=0; i<6;i++){
        cout<<*p<<endl;
        //++ moves the pointer to next int position
        p++;
    }
    return 0;
}
```

OUTPUT:  
0  
1  
2  
3  
4  
5  
6

# This Pointers

- this pointer hold the address of current object
- int num;
- This->num=num;

# Function using pointers

```
#include<iostream>
using namespace std;
void swap(int *a ,int *b );
//Call By Reference
int main()
{
    int p,q;
    cout<<"\nEnter Two Number You Want To Swap \n";
    cin>>p>>q;
    swap(&p,&q);
    cout<<"\nAfter Swapping Numbers Are Given below\n\n";
    cout<<p<<"  "<<q<<" \n";
    return 0;
}
```

```
void swap(int *a,int *b)
{
    int c;
    c=*a;
    *a=*b;
    *b=c;
}
```

Output:  
Enter Two Number You Want to Swap  
10 20  
After Swapping Numbers Are Given below  
20 10

# Type conversion and type casting

- Type conversion or typecasting of variables refers to changing a variable of one data type into another. While type conversion is done implicitly, casting has to be done explicitly by the programmer.

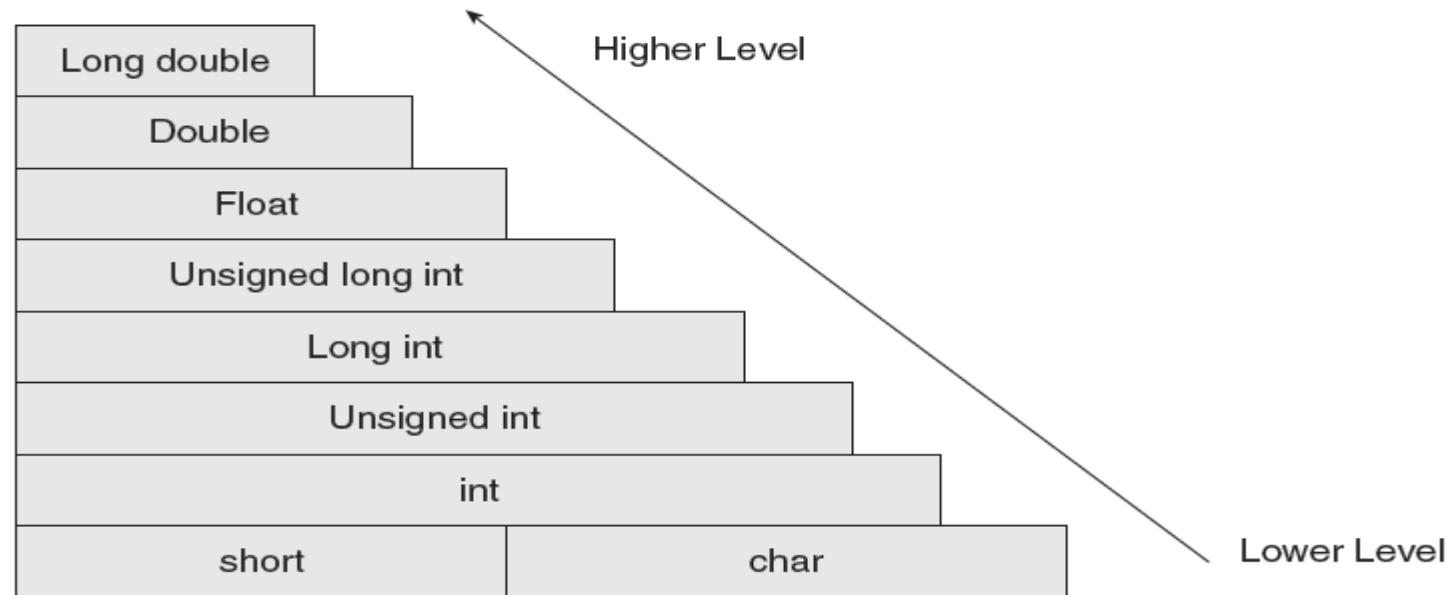


Figure 2.13 Conversion hierarchy of data types

# Type Casting

- Type casting an arithmetic expression tells the compiler to represent the value of the expression in a certain format.
- It is done when the value of a higher data type has to be converted into the value of a lower data type.
- However, this cast is under the programmer's control and not under the compiler's control. The general syntax for type casting is **destination\_variable\_name=destination\_data\_ty--pe(source\_variable\_name);**

```
float sal=10000.00;  
int income;  
Income=int(sal);
```

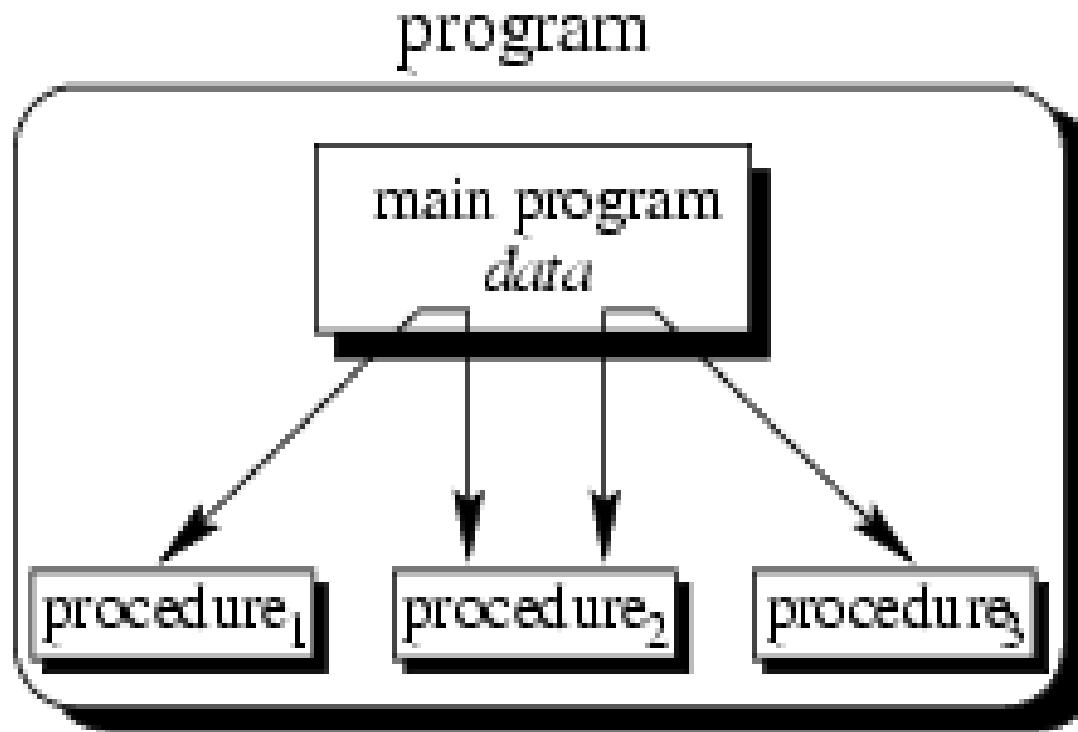


**SRM**  
INSTITUTE OF SCIENCE & TECHNOLOGY  
Dedicated to the University of Jaffna, Sri Lanka

# 18CSC202J - OBJECT ORIENTED DESIGN AND PROGRAMMING

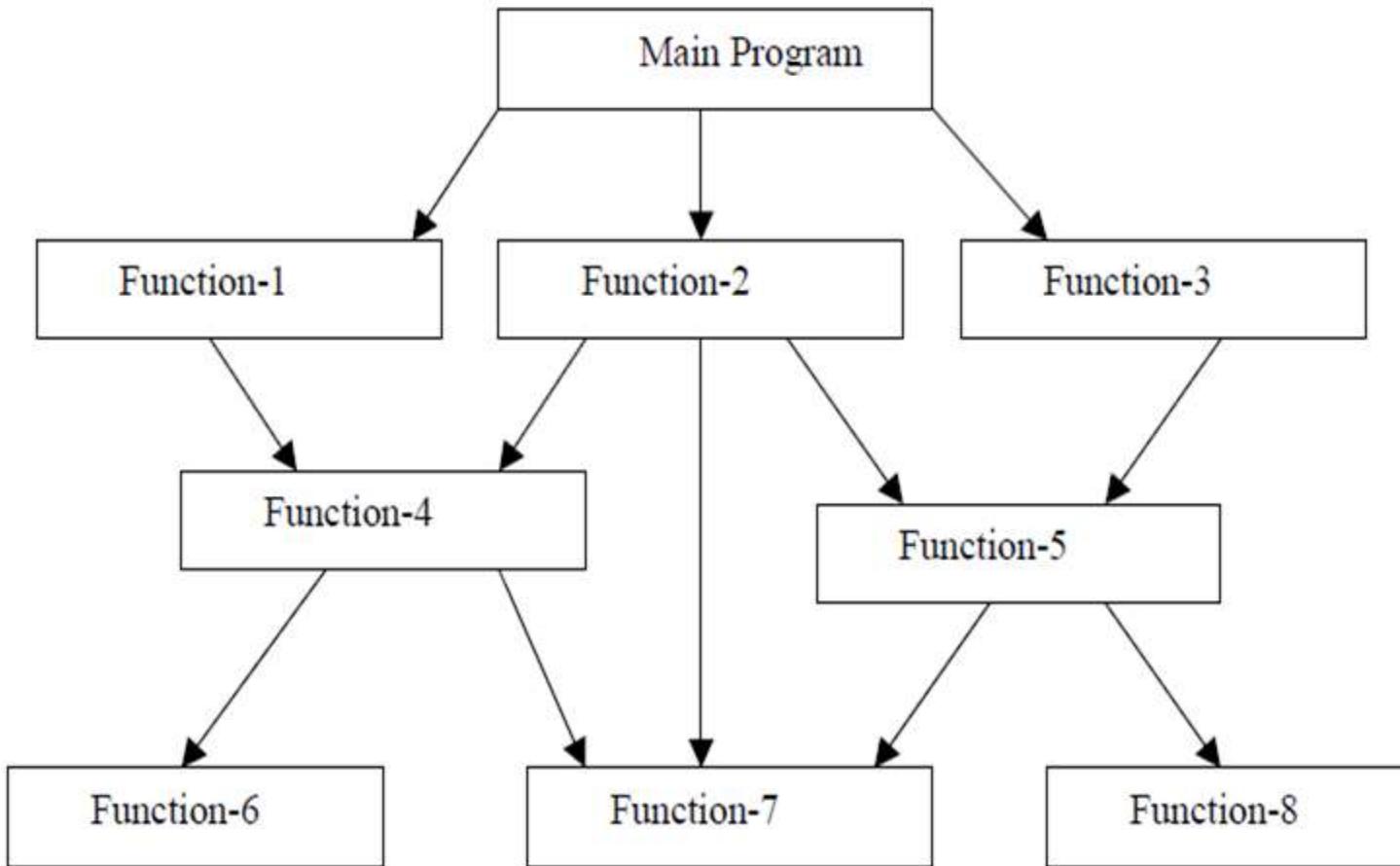
## Session 1

**Topic : PROCEDURAL AND OBJECT  
ORIENTED PROGRAMMING**



- The main program coordinates calls to procedures and hands over appropriate data as parameters.

# Procedure Oriented Programming Language



C function aspects	Syntax
Function declaration	Return-type function-name(argument list);  Eg : int add(int a, int b);
Function definition	Return-type function-name(argument list) { body of function;)  Eg : int add(int a, int b) {int c; c=a+b; Return c; }
Function call	Function-name(argument list);  Eg : add(5,10);

## Features of Procedure Oriented Programming Language

- **Smaller programs** - A program in a procedural language is a list of instructions.
- **Larger programs** are divided into smaller programs known as functions.
- Each **function** has a **clearly defined purpose and a clearly defined interface to the other functions** in the program.
- **Data is Global** and shared by almost all the functions.
- Employs **Top Down approach** in Program Design.

## Examples of Procedure Oriented Programming Language

- COBOL
- FORTRAN
- C

# Sample COBOL Program(Procedure Oriented)

IDENTIFICATION DIVISION.

PROGRAM-ID.

ENVIRONMENT DIVISION. \* procedure will be followed by using few Division

DATA DIVISION.

WORKING-STORAGE SECTION.

77 A PIC 9999.

77 B PIC 9999.

77 ANS PIC 999V99.

PROCEDURE DIVISION.

MAIN-PARA.

```
DISPLAY '-----'.
DISPLAY " ENTER A"
ACCEPT A.           * command line argument
DISPLAY "ENTER B".
ACCEPT B.
DISPLAY '-----'.
```

ADD-PARA.

```
ADD A B GIVING ANS.
DISPLAY '-----'.
```

DISP-PARA.

```
DISPLAY "A IS " A.
DISPLAY "B IS " B.
DISPLAY "ADDITION -" ANS.
STOP RUN.          * to stop the program
```

# Disadvantages of Procedural Programming Language

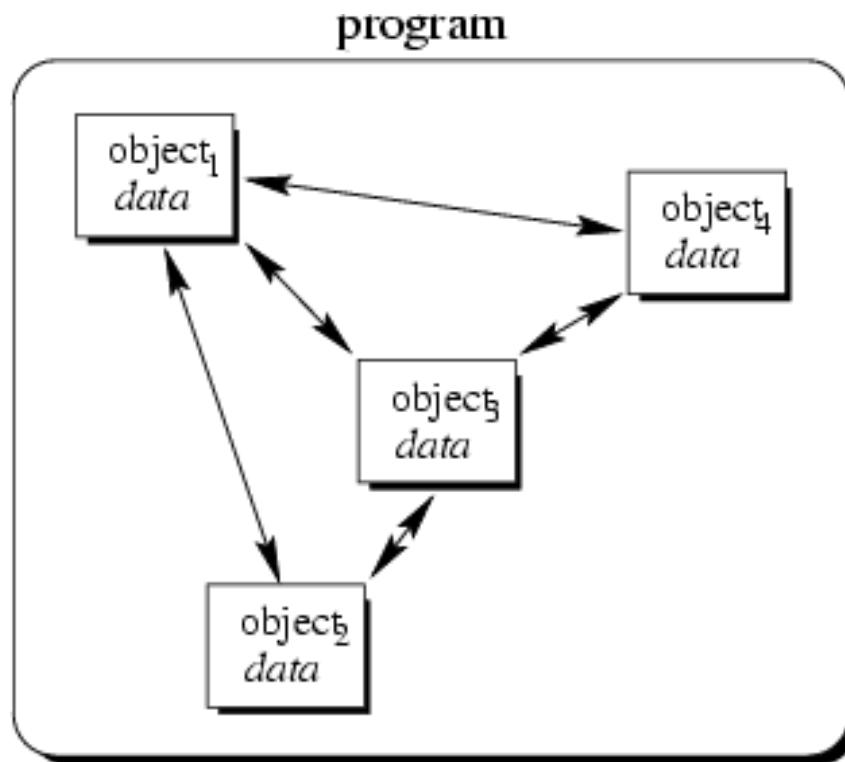
## Unrestricted access

- functions have unrestricted access to global data.

## Real-world modeling

- unrelated (separated) functions and data, the basis of the procedural paradigm, provide a poor model of the real world.
- Complex real-world objects have both *attributes (data)* and *behavior (function)*.

# Object-Oriented Concept



- Objects of the program interact by sending messages to each other, hence it increases the data security (data can be accessed only through its instances)

# Object Oriented Programming

- Object-oriented programming is a programming paradigm that uses **abstraction** in the form of **classes and objects** to create models based on the **real world environment**.
- An object-oriented application uses a collection of objects, which communicate by **passing messages** to request services.
- The aim of object-oriented programming is to try to increase the **flexibility and maintainability** of programs. Because programs created using an OO language are **modular**, they can be **easier** to develop, and **simpler** to understand after development

# Object-Oriented Concepts

- Everything is an object and each object has its **own memory**
- Computation is performed by objects communicating with each other
- Every object is an instance of a class. A class simply represents a grouping of similar objects, such as Integers or lists.
- The class is the repository for behavior associated with an object. That is, that all objects that are instances of the same class can perform the same actions.
- Classes are organized into a singly rooted tree structure, called the **inheritance hierarchy**. Memory and behavior associated with instances of a class are automatically available to any class associated with a descendant in this tree structure.

# Object-Oriented Programming vs. Procedural Programming

- Programs are made up of modules, which are parts of a program that can be coded and tested separately, and then assembled to form a complete program.
- In procedural languages (i.e. C) these modules are procedures, where a procedure is a sequence of statements.
- The design method used in procedural programming is called Top Down Design. This is where you start with a problem (procedure) and then systematically break the problem down into sub problems (sub procedures).

## Object-Oriented Programming vs. Procedural Programming

- The difficulties with ***Procedural Programming***, is that software maintenance can be **difficult and time consuming**.
- When changes are made to the main procedure (top), those changes can cascade to the sub procedures of main, and the sub-sub procedures and so on, where the change may impact all procedures in the pyramid.
- Object oriented programming is meant to address the difficulties with procedural programming.

# The main difference

Procedural	Object-oriented
Procedure	Method
Record	Object
Module	Class
Procedure call	Message

# Comparison

## Procedural Oriented

Program is divided into small parts called 'Functions'

Global and Local data

Doesn't have any proper way for hiding data

Eg: C, VB, FORTAN

## Object Oriented

Program is divided into small parts called 'Objects'

Has access specifiers : Public, Private, Protected

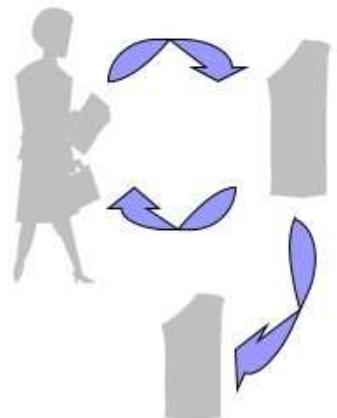
Provides data hiding and security

Eg: C++, JAVA, VB.NET

# Comparison

## Procedural vs. Object-Oriented

- Procedural



Withdraw, deposit, transfer

- Object Oriented



Customer, money, account

# **18CSC202J - OBJECT ORIENTED DESIGN AND PROGRAMMING**

## **Session 7**

**Topic : UML Class Diagram, its components,  
Class Diagram relations and Multiplicity**

# Overview of UML Diagrams

## Structural

: element of spec. irrespective of time

- Class
- Component
- Deployment
- Object
- *Composite structure*
- *Package*

## Behavioral

: behavioral features of a system / business process

- Activity
- State machine
- Use case
- *Interaction*

## Interaction

: emphasize object interaction

- Communication(collaboration)
- Sequence
- *Interaction overview*
- *Timing*

## UML Class diagram :

- The UML Class diagram is a graphical notation used to construct and visualize object oriented systems.
- A class diagram describes the structure of a system such as **Classes** and their **attributes**, **operations (or methods)** and the **relationships among objects**.
- A

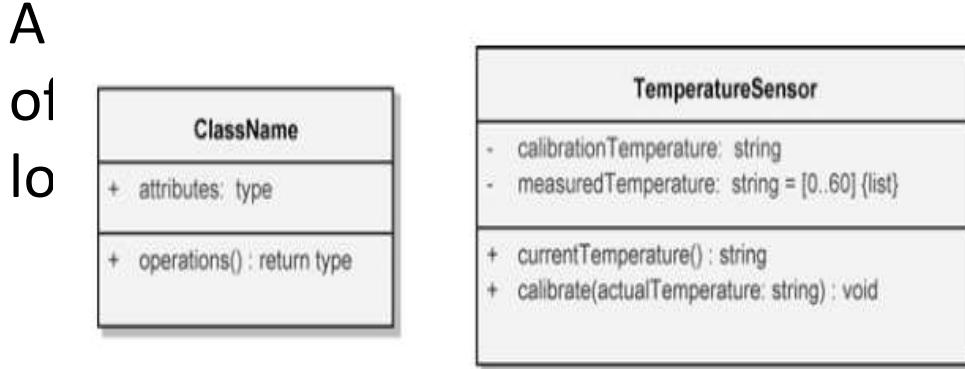


Figure 5–33 A General Class Icon and an Example for the Gardening System

# Basic components of a class diagram

The standard class diagram is composed of three sections:

**Upper section:** Contains the name of the class. This section is always required, to know whether it represents the classifier or an object.

**Middle section:** Contains the attributes of the class. Use this section to describe the qualities of the class. This is only required when describing a specific instance of a class.

**Bottom section:** Includes class operations (methods). Displayed in list format, each operation takes up its own line. The operations describe how a class interacts with data.

## RULES TO BE FOLLOWED:

- Class name must be unique to its enclosing namespace.
- The class name begins in uppercase and the space between multiple words is omitted.
- The first letter of the attribute and operation names is lowercase with subsequent words starting in uppercase and spaces are omitted.
- Since the class is the namespace for its attributes and operations an attribute name must be unambiguous in the context of the class.

- Attribute specification format:

**visibility attributeName : Type [multiplicity] =  
DefaultValue {property string}**

- Operation specification format:

**visibility operationName (parameterName :  
Type) : ReturnType {property string}**

### VISIBILITY

**Public (+)** Visible to any element that can see the class.

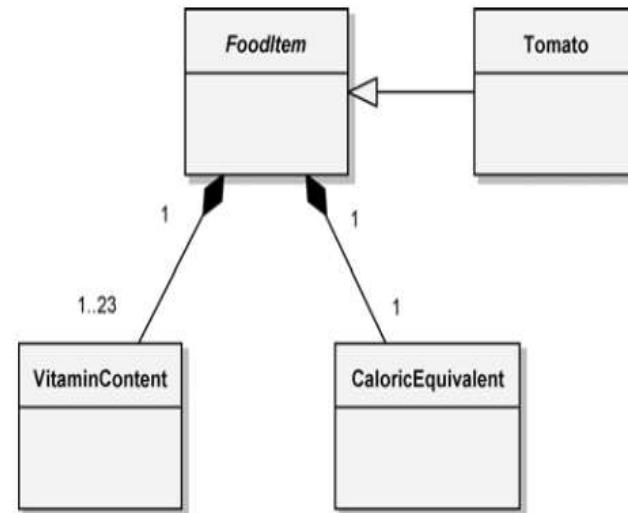
**Protected (#)** Visible to other elements within the class and to subclasses .

**Private (-)** Visible to other elements within the class .

**Package (~)** Visible to elements within the same package.

## Abstract class

- An abstract class is one for which no instances may be created.
- Because such classes are so important to engineering good class inheritance trees, there is a special way to designate an abstract class.



- To denote that an operation is abstract, we simply italicize the operation name; this means that this operation may be implemented differently by all instances of its subclasses.
  - In the **Hydroponics Gardening System**, we have food items that have a specific vitamin content and caloric equivalent, but there is not a type of food called “food item.”
  - Hence, the **FoodItem class is abstract.**
  - Above fig also shows the subclass Tomato, which represents a concrete (instantiable) food item grown in the greenhouse.

## Class Relationships

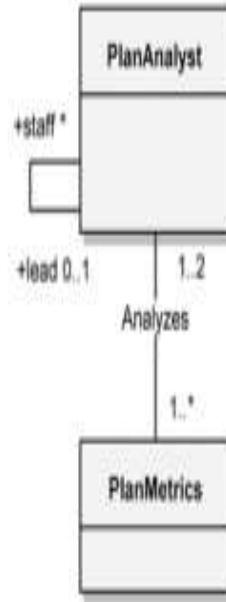
The essential connections among classes include

- Association
- Generalization
- Aggregation
- Composition

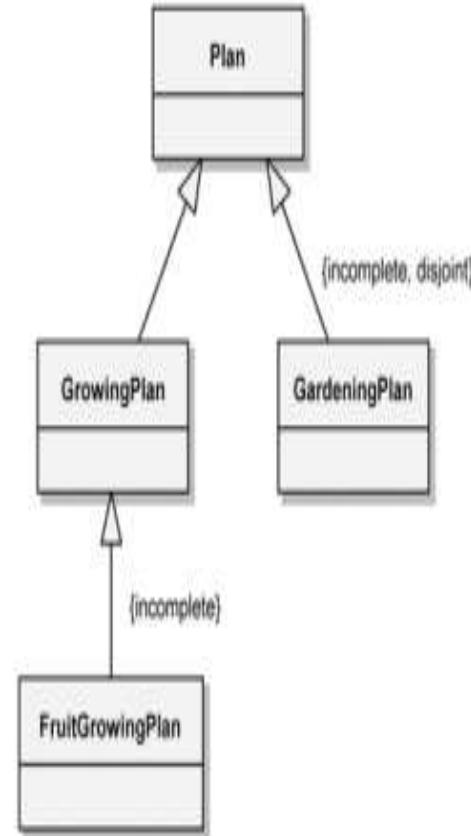
### Association :

- The association icon connects two classes and denotes a **semantic connection**.
- Associations are often labeled with **noun phrases**, such as **Analyzes**, denoting the **nature of the relationship**.
- A class may have an association to itself (**called a reflexive association**), such as the collaboration among instances of the **PlanAnalyst** class.

## ASSOCIATION



## GENERALIZATION



- A class may have an association to itself (**called a reflexive association**), such as the collaboration among instances of the **PlanAnalyst** class.
- The use of both the association end names and the association name is to provide clarity.
- It is also possible to have more than one association between the same pair of classes.
- Associations may be further adorned with their multiplicity, using the following syntax:

**1** Exactly one

\* Unlimited number (zero or more)

**0..\*** Zero or more

**1..\*** One or more

**0..1** Zero or one

**3..7** Specified range (from three through seven, inclusive)

.

## **GENERALIZATION**

- The generalization icon denotes a generalization/specialization relationship and appears as an association with a closed arrowhead.

The **GrowingPlan** class in Figure is the **superclass** and its **subclass** is the **FruitGrowingPlan**

## **AGGREGATION**

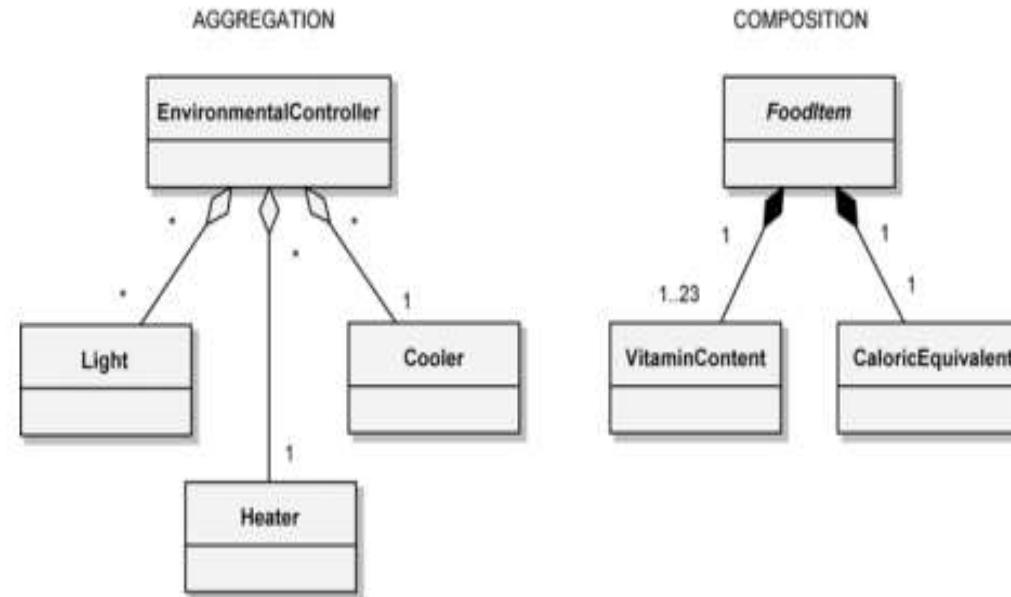
- Aggregation, as manifested in the “part of ” relationship, is a constrained form of the more general association relationship.

- It appears as an association with an unfilled diamond at the end denoting the aggregate (the whole).
- The class at the other end denotes the class whose instances are part of the aggregate object.
- Reflexive and cyclic aggregation is possible.  
In figure, an individual **EnvironmentalController** class has the **Light**, **Heater**, and **Cooler** classes as its parts.
- The multiplicity of \* (zero or more) at the aggregate end of the relationship further highlights this lack of physical containment.

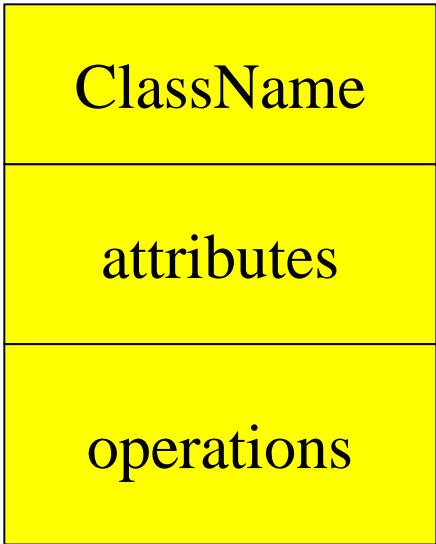
## **COMPOSITION**

- Composition implies that the construction and destruction of these parts occurs as a consequence of the construction and destruction of the aggregate.

- The icons described thus far constitute the essential elements of all class diagrams.
- Collectively, they provide the developer with a notation sufficient to describe the fundamentals of a system's class structure.



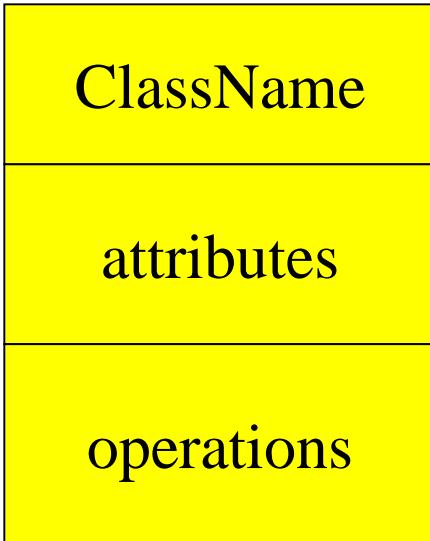
# Classes



A *class* is a description of a set of objects that share the same attributes, operations, relationships, and semantics.

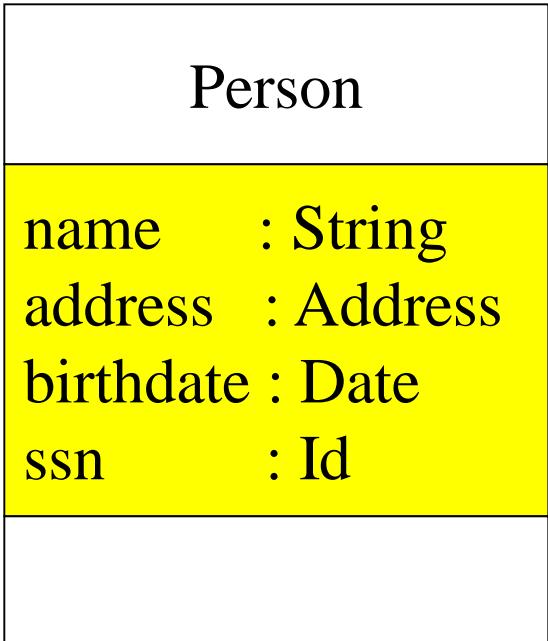
Graphically, a class is rendered as a rectangle, usually including its name, attributes, and operations in separate, designated compartments.

# Class Names



The name of the class is the only required tag in the graphical representation of a class. It always appears in the top-most compartment.

# Class Attributes



An *attribute* is a named property of a class that describes the object being modeled. In the class diagram, attributes appear in the second compartment just below the name-compartment.

# Class Attributes (Cont'd)

Person	
name	: String
address	: Address
birthdate	: Date
/ age	: Date
ssn	: Id

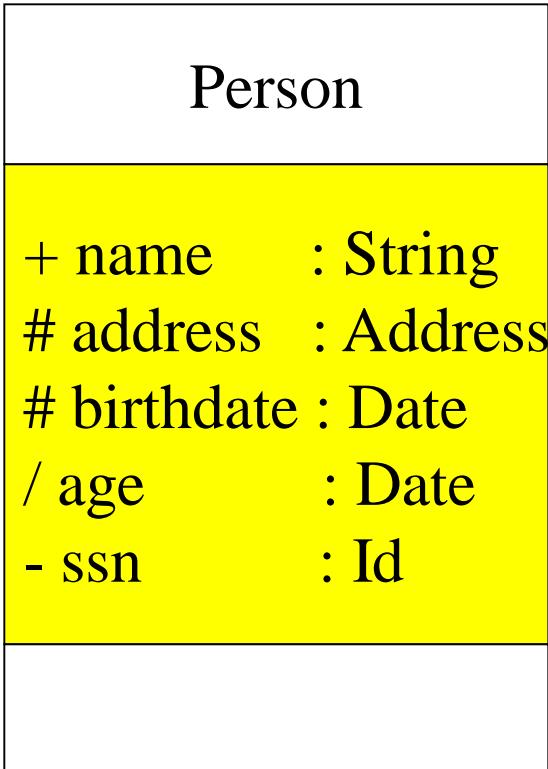
Attributes are usually listed in the form:

attributeName : Type

A *derived* attribute is one that can be computed from other attributes, but doesn't actually exist. For example, a Person's age can be computed from his birth date. A derived attribute is designated by a preceding '/' as in:

/ age : Date

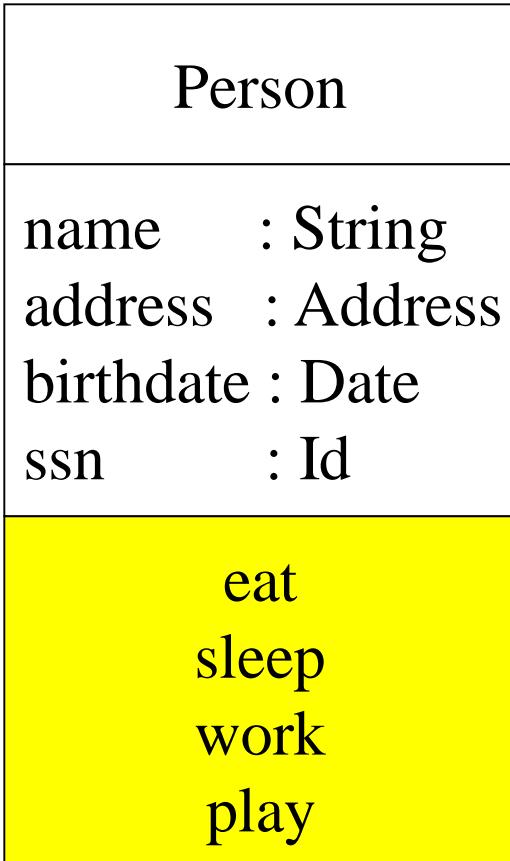
# Class Attributes (Cont'd)



Attributes can be:

- + public
- # protected
- private
- / derived

# Class Operations



*Operations* describe the class behavior and appear in the third compartment.

# Class Operations (Cont'd)

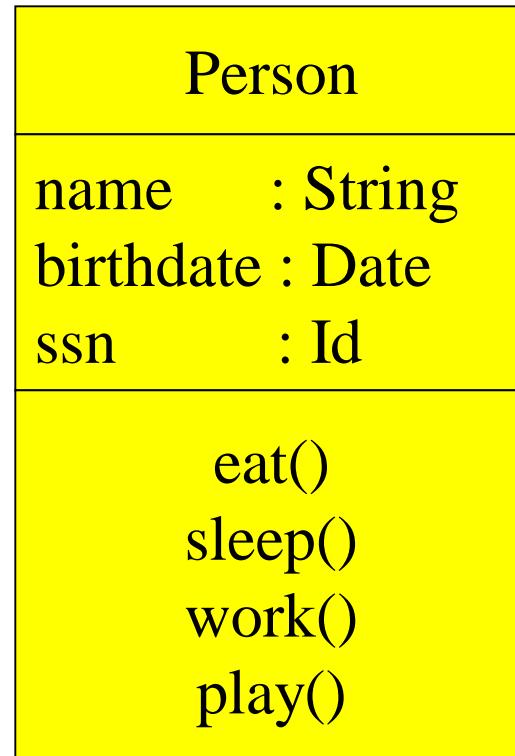
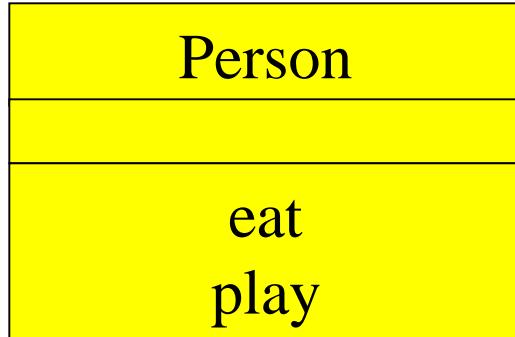
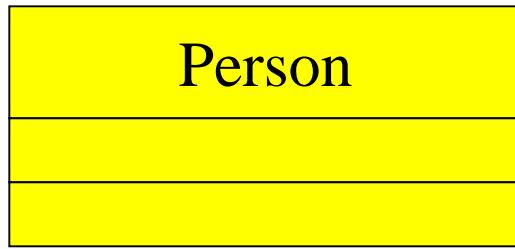
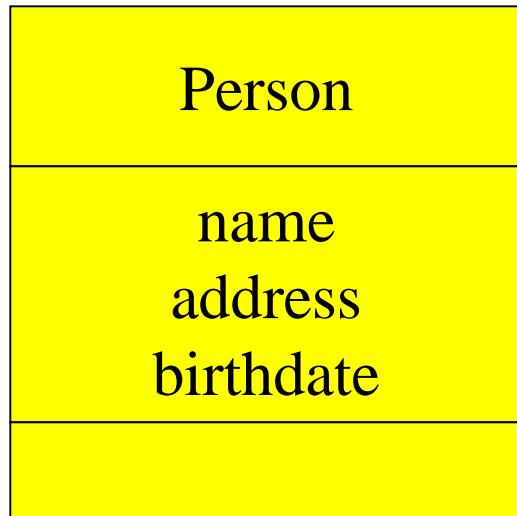
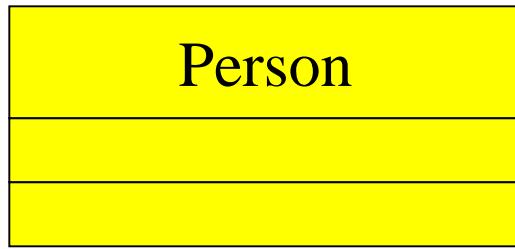
PhoneBook

newEntry (n : Name, a : Address, p : PhoneNumber, d : Description)  
getPhone ( n : Name, a : Address) : PhoneNumber

You can specify an operation by stating its signature: listing the name, type, and default value of all parameters, and, in the case of functions, a return type.

# Depicting Classes

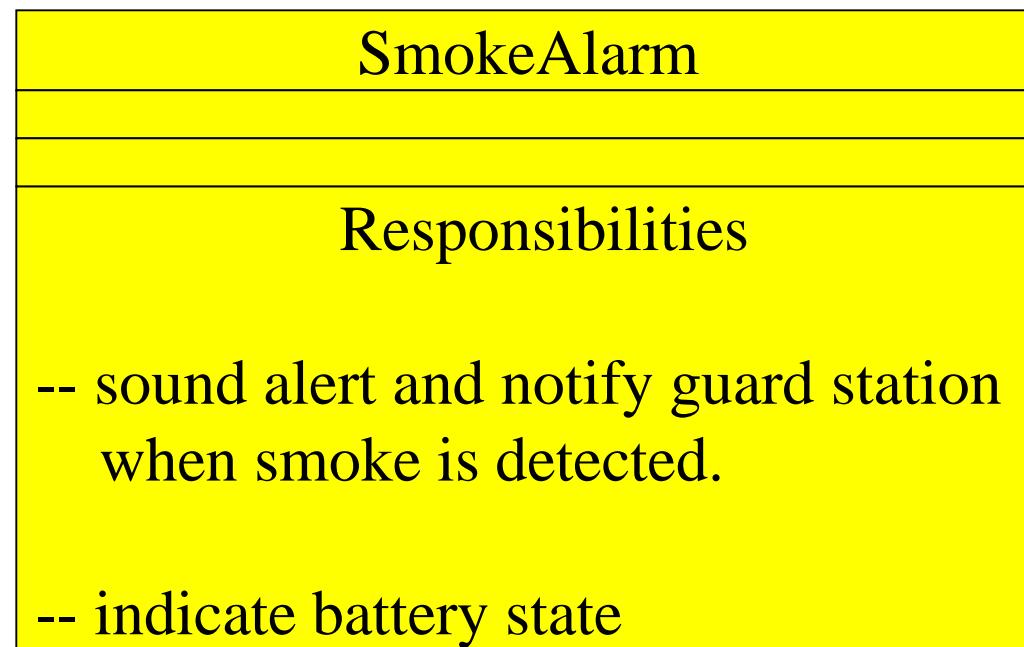
When drawing a class, you needn't show attributes and operation in every diagram.



# Class Responsibilities

A class may also include its responsibilities in a class diagram.

A responsibility is a contract or obligation of a class to perform a particular service.



# Relationships

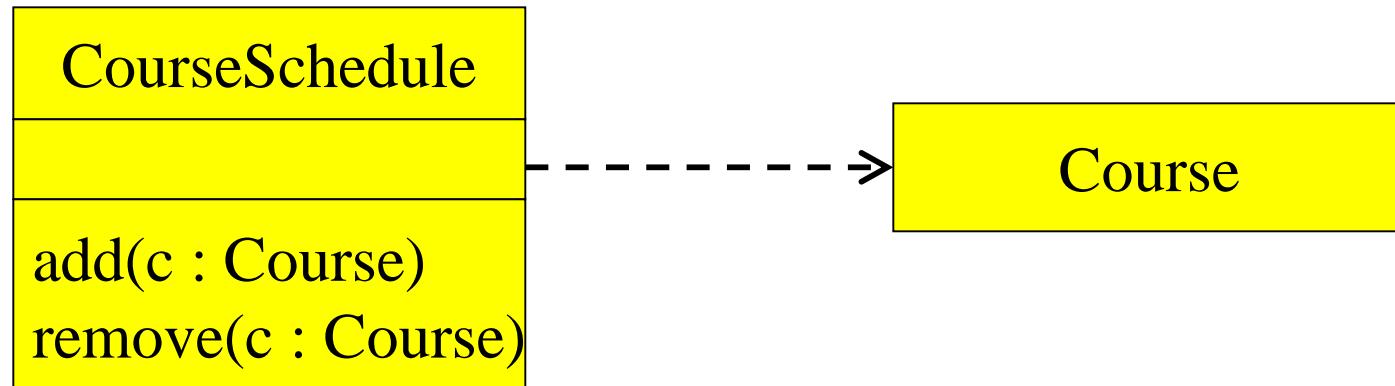
In UML, object interconnections (logical or physical), are modeled as relationships.

There are three kinds of relationships in UML:

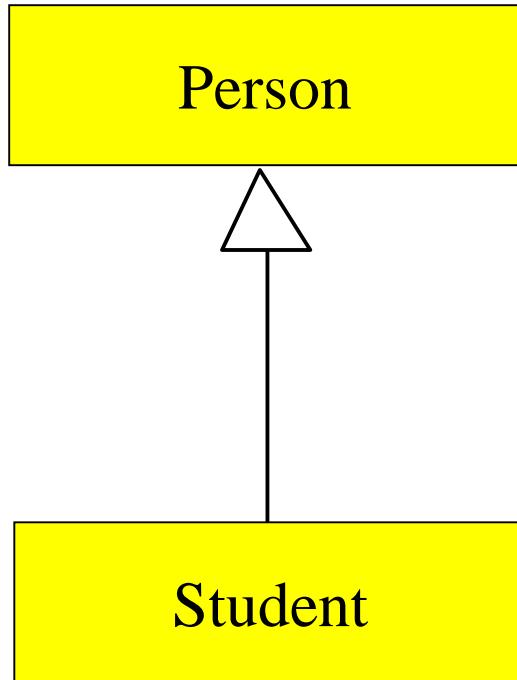
- dependencies
- generalizations
- associations

# Dependency Relationships

A *dependency* indicates a semantic relationship between two or more elements. The dependency from *CourseSchedule* to *Course* exists because *Course* is used in both the **add** and **remove** operations of *CourseSchedule*.



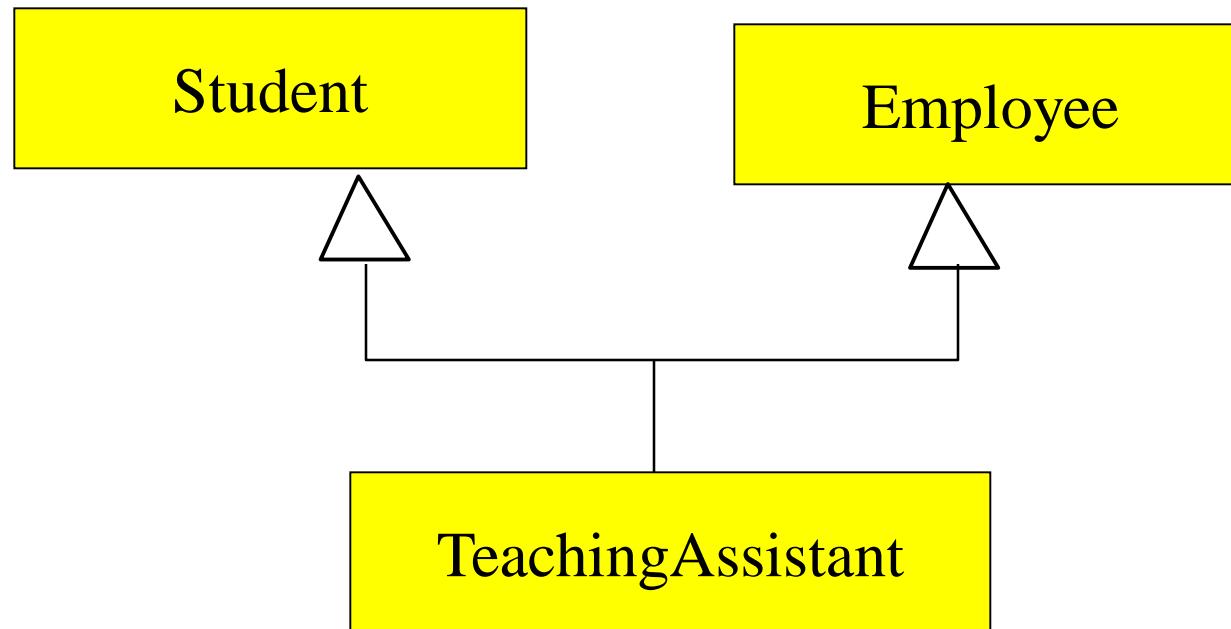
# Generalization Relationships



A *generalization* connects a subclass to its superclass. It denotes an inheritance of attributes and behavior from the superclass to the subclass and indicates a specialization in the subclass of the more general superclass.

# Generalization Relationships (Cont'd)

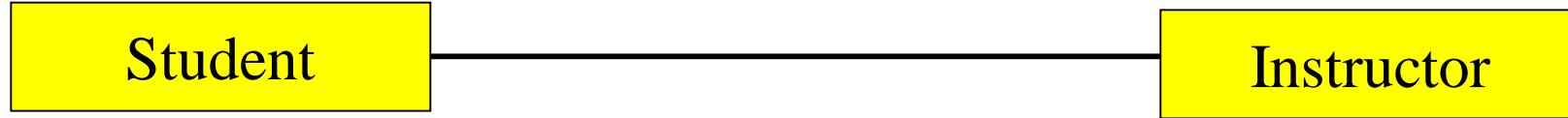
UML permits a class to inherit from multiple superclasses, although some programming languages (*e.g.*, Java) do not permit multiple inheritance.



# Association Relationships

If two classes in a model need to communicate with each other, there must be link between them.

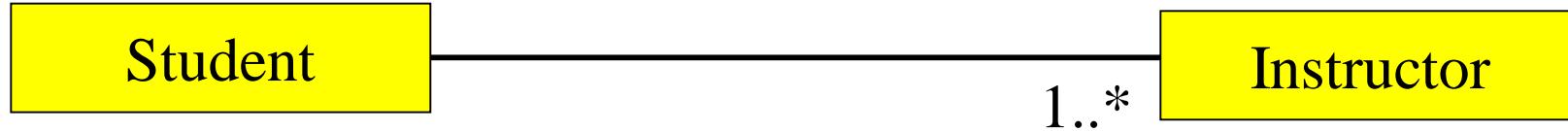
An *association* denotes that link.



# Association Relationships (Cont'd)

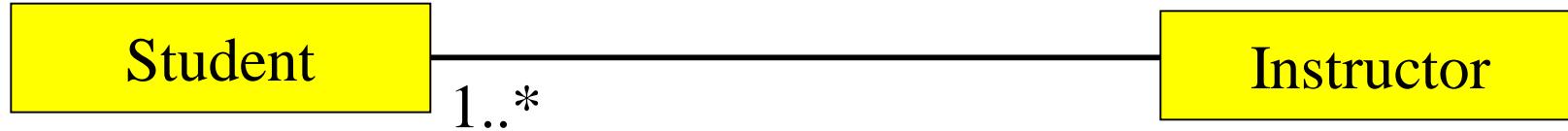
We can indicate the *multiplicity* of an association by adding *multiplicity adornments* to the line denoting the association.

The example indicates that a *Student* has one or more *Instructors*:



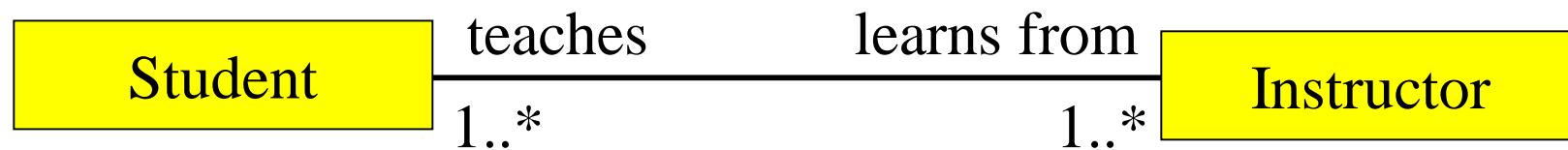
# Association Relationships (Cont'd)

The example indicates that every *Instructor* has one or more *Students*:



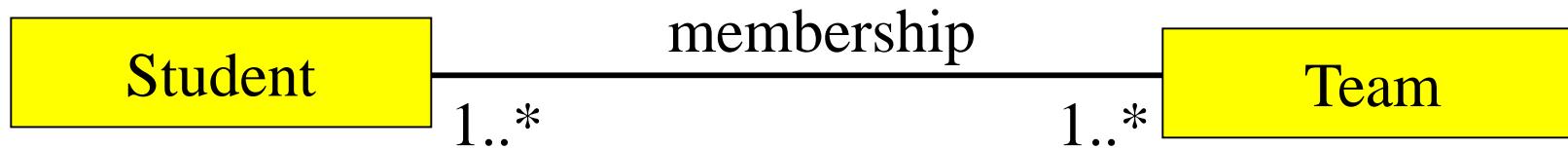
# Association Relationships (Cont'd)

We can also indicate the behavior of an object in an association (*i.e.*, the *role* of an object) using *rolenames*.



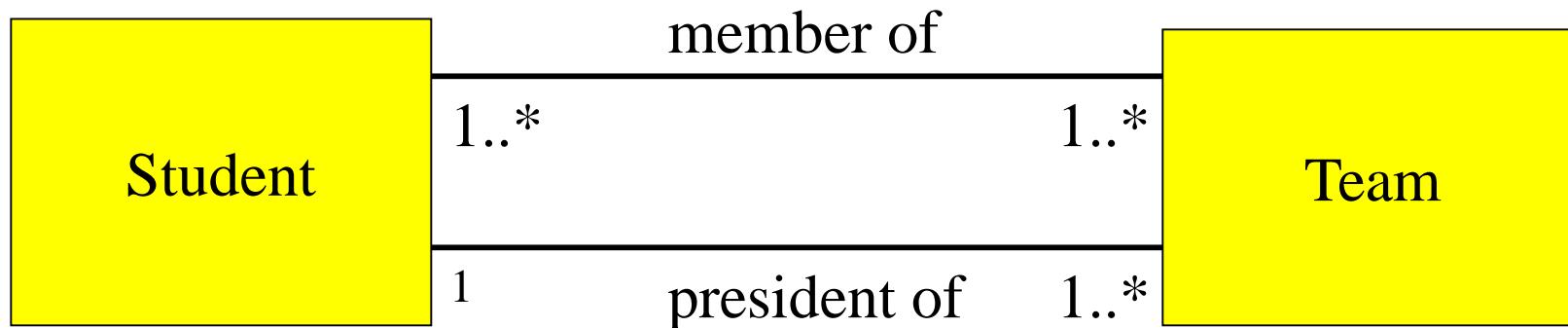
# Association Relationships (Cont'd)

We can also name the association.



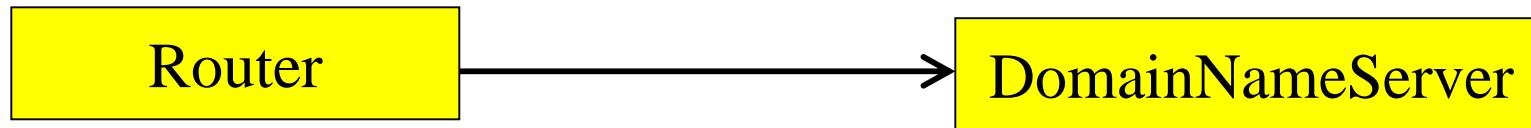
# Association Relationships (Cont'd)

We can specify dual associations.



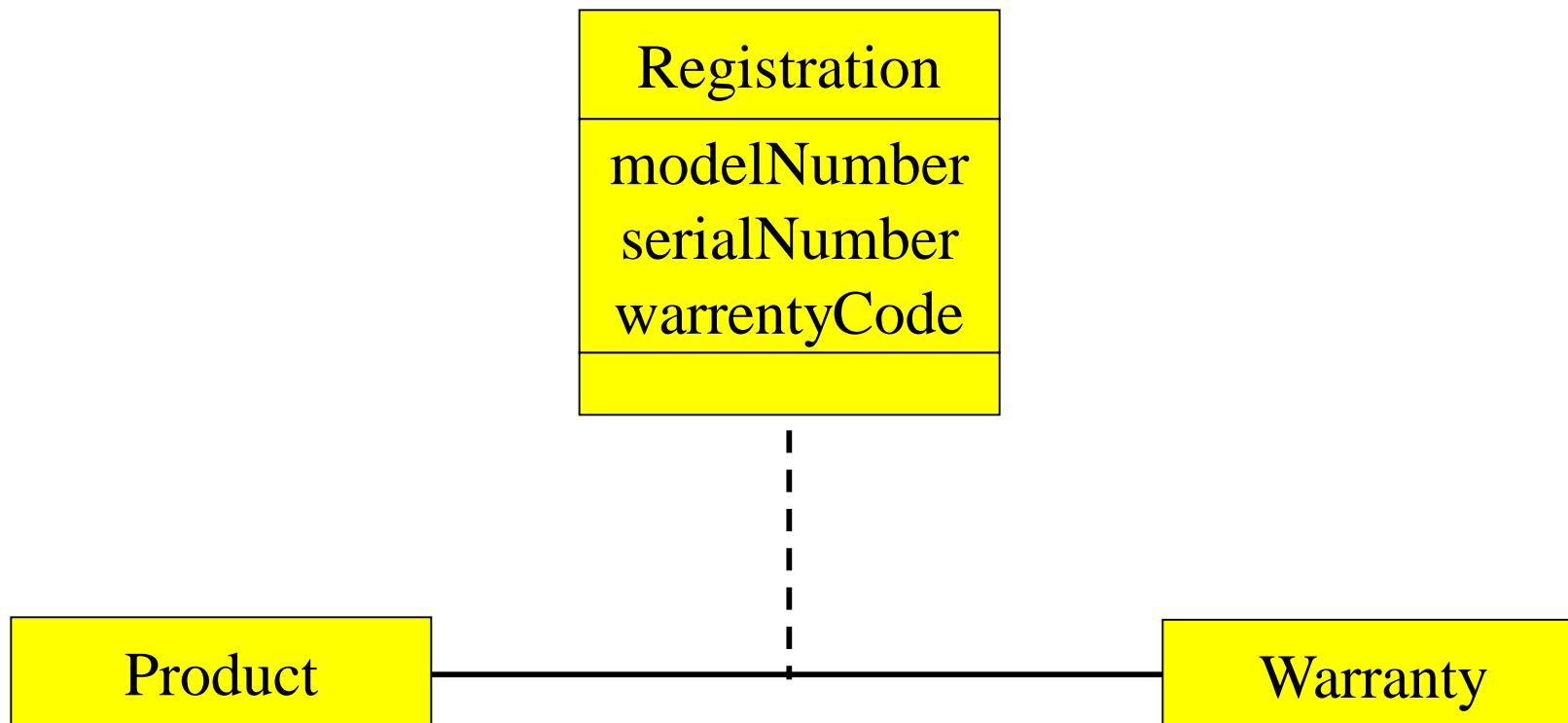
# Association Relationships (Cont'd)

We can constrain the association relationship by defining the *navigability* of the association. Here, a *Router* object requests services from a *DNS* object by sending messages to (invoking the operations of) the server. The direction of the association indicates that the server has no knowledge of the *Router*.



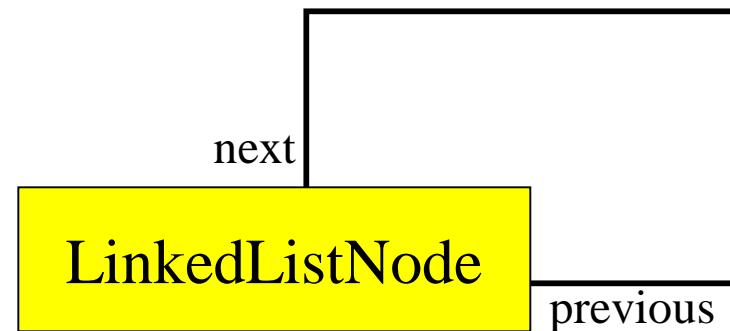
# Association Relationships (Cont'd)

Associations can also be objects themselves, called *link classes* or an *association classes*.



# Association Relationships (Cont'd)

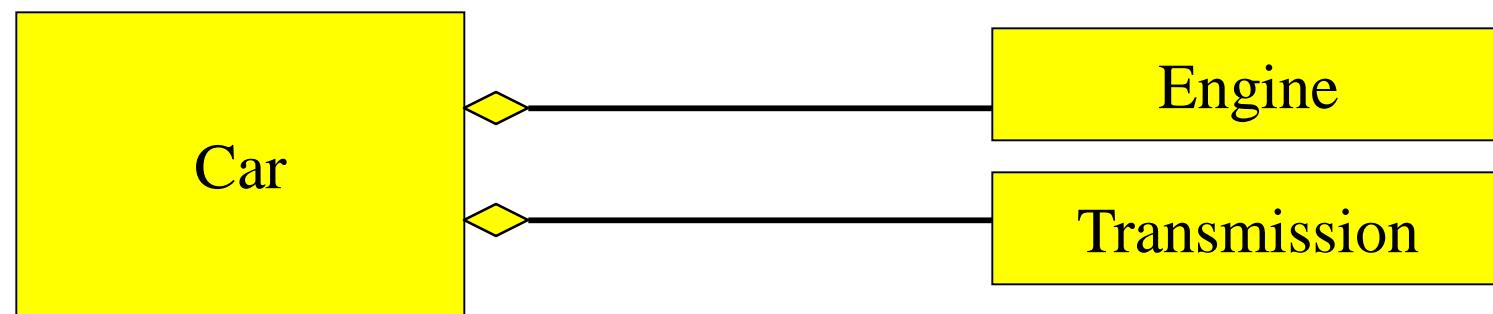
A class can have a *self association*.



# Association Relationships (Cont'd)

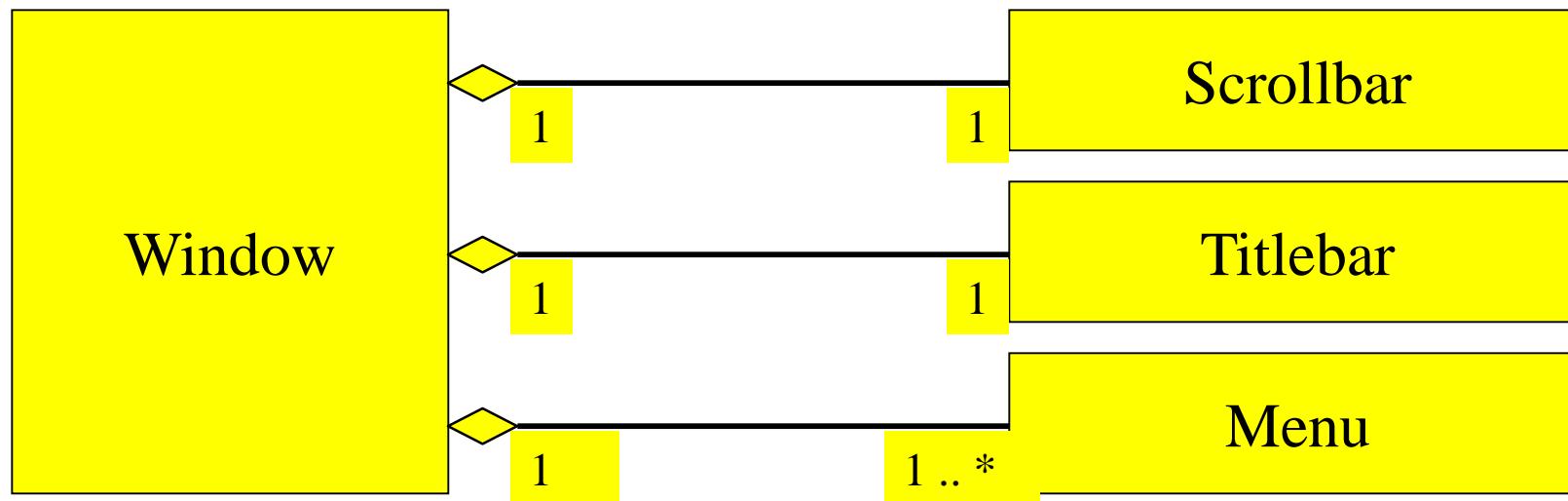
We can model objects that contain other objects by way of special associations called *aggregations* and *compositions*.

An *aggregation* specifies a whole-part relationship between an aggregate (a whole) and a constituent part, where the part can exist independently from the aggregate. Aggregations are denoted by a hollow-diamond adornment on the association.



# Association Relationships (Cont'd)

A *composition* indicates a strong ownership and coincident lifetime of parts by the whole (*i.e.*, they live and die as a whole). Compositions are denoted by a filled-diamond adornment on the association.



# Interfaces

<<interface>>  
ControlPanel

An *interface* is a named set of operations that specifies the behavior of objects without showing their inner structure. It can be rendered in the model by a one- or two-compartment rectangle, with the *stereotype* <<interface>> above the interface name.

## **Multiplicity**

1. One-to-one
2. One-to-many
3. Many-to-many

### **One-to-one relationship**

For example, in retail telemarketing operations,  
we would find

a one-to-one relationship between the class  
**Sale** and the class  
**CreditCardTransaction**.

### **Many-to-many relationship**

For example, each instance of the class  
**Customer** might initiate  
a transaction with several instances of the class  
**SalesPerson**,

# 18CSC202J - OBJECT ORIENTED DESIGN AND PROGRAMMING

## Session 12

**Topic : UML use case Diagram, use case, Scenario, Use case Diagram objects and relations**

## **USECASE DIAGRAM**

- Use case diagrams give us that capability.
- Use case diagrams are used to depict the context of the system to be built and the functionality provided by that system.
- They depict who (or what) interacts with the system. They show what the outside world wants the system to do.

## NOTATIONS

- Actors are entities that interface with the system.
- They can be people or other systems.
- Actors, which are external to the system they are using, are depicted as stylized stick figures.



Figure 5–20 Actors

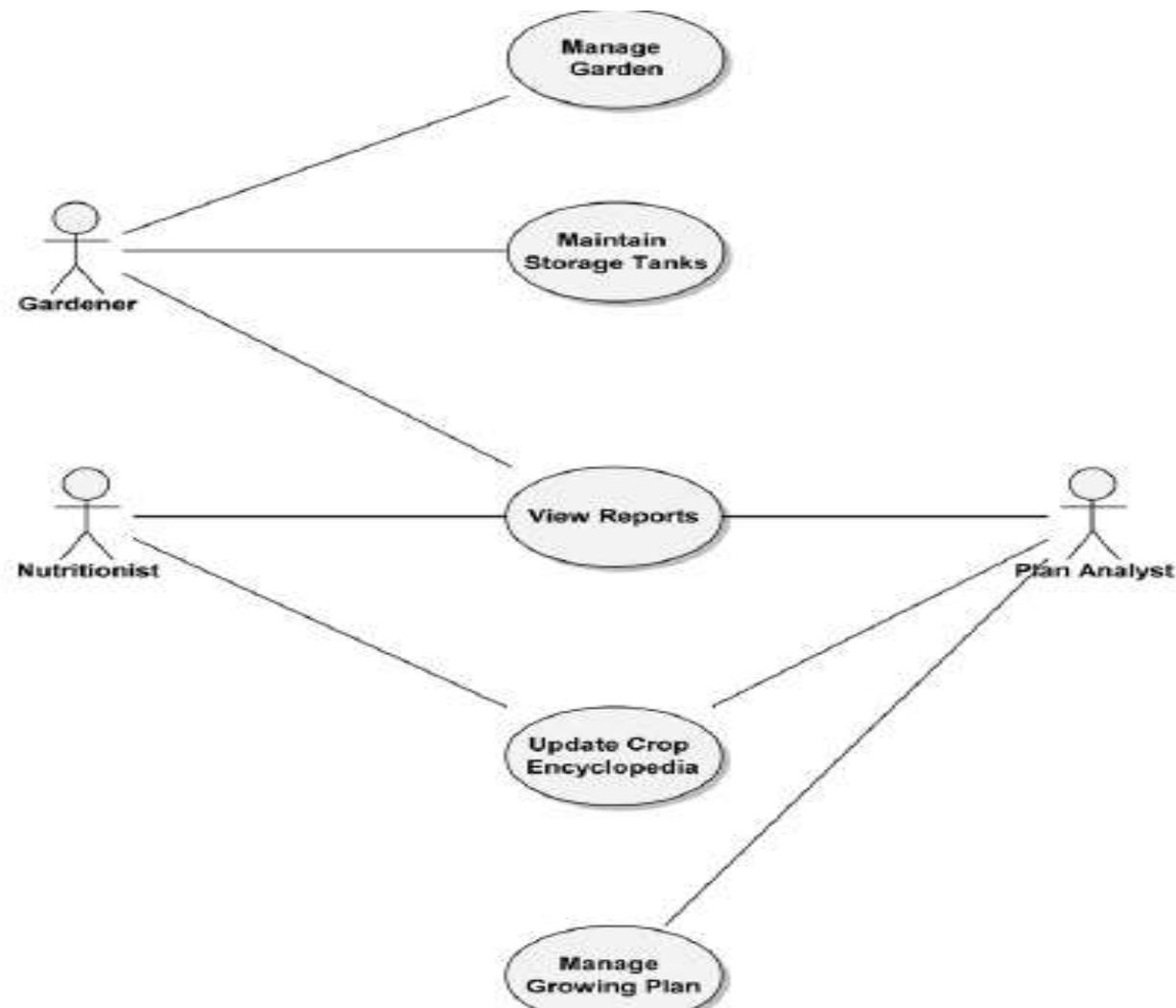


Figure 5–22 A Use Case Diagram

## **An Example Use Case Specification**

Let us look at an example for the use case Maintain Storage Tanks.

### **Use Case Specification**

**Use case name:** Maintain Storage Tanks

**Use case purpose:** This use case provides the ability to maintain the fill levels of the contents of the storage tanks. This use case allows the actor to maintain specific sets of water and nutrient tanks.

**Optimistic flow:**

- A. Actor examines the levels of the storage tanks' contents.
- B. Actor determines that tanks need to be refilled.
- C. Normal hydroponics system operation of storage tanks is suspended by the actor.
- D. Actor selects tanks and sets fill levels.

For each selected tank, steps E through G are performed.

- E. If tank is heated, the system disables heaters.
  1. Heaters reach safe temperature.
- F. The system fills tank.
- G. When tank is filled, if tank is heated, the system enables heaters.
  1. Tank contents reach operating temperature.
- H. Actor resumes normal hydroponics system operation.

**Pragmatic flows:**

**Conditions triggering alternate flow:**

**Condition 1:** There is insufficient material to fill tanks to the levels specified by the actor.

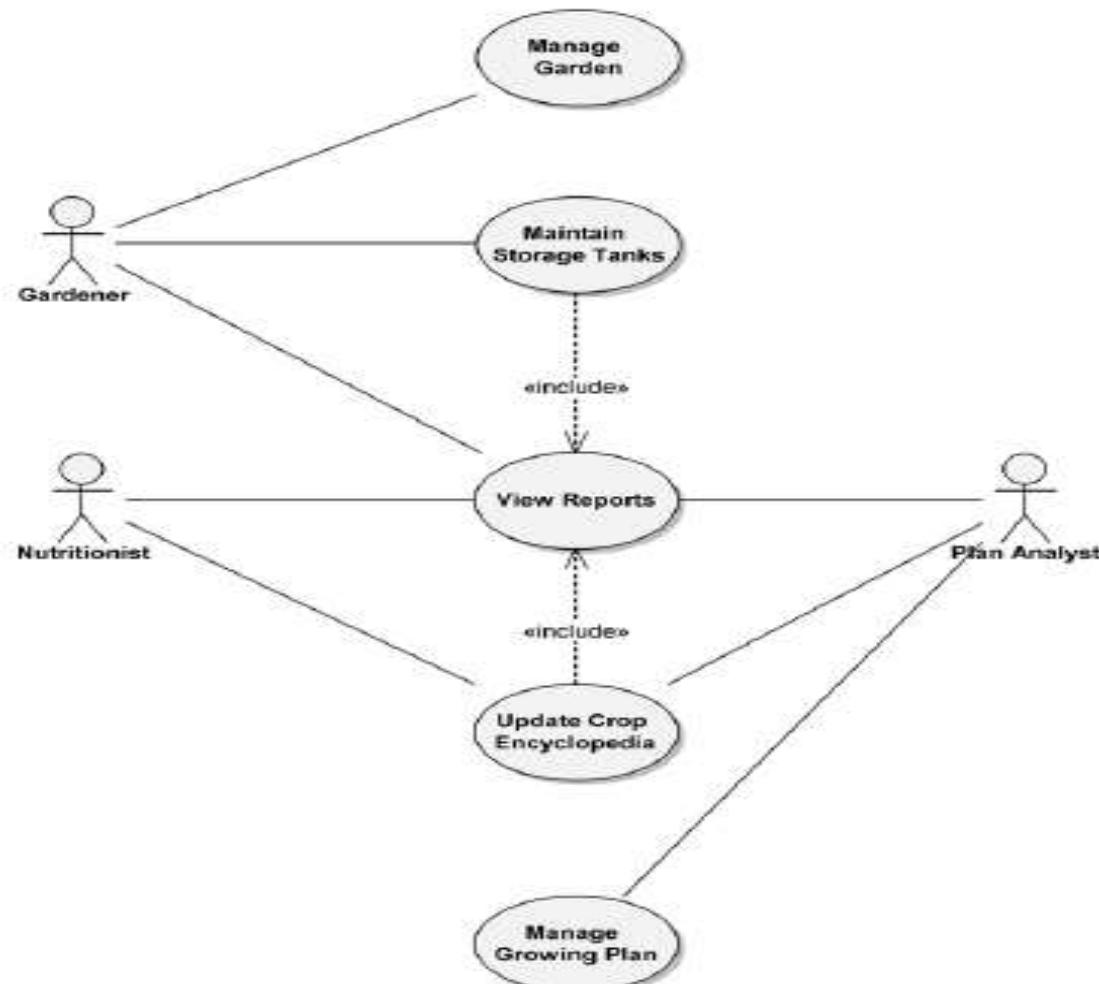
- D1. Alert actor regarding insufficient material available to meet tank setting. Show amount of material available.
- D2. Prompt actor to choose to end maintenance or reset fill levels.
- D3. If reset chosen, perform step D.
- D4. If end maintenance chosen, perform step H.
- D5. Else, perform step D2.

**Condition 2:** . . .

## Relationship

Two relationships used primarily for organizing use case models are both powerful

- «include» relationship
- «extend» relationship



**Figure 5–23** A Use Case Diagram Showing «include» Relationships

## «include» relationship

- In our hydroponics example, we have an *Update Crop Encyclopedia* use case.
- During analysis, we determine that the *Nutritionist* actor using that use case will have to see what is in the crop encyclopedia prior to updating it.
- This is why the *Nutritionist* can invoke the View Reports use case.
- The same is true for the *Gardener* actor whenever invoking *Maintain Storage Tanks*.

- Neither actor should be executing the use cases blindly. Therefore, the *View Report* use case is a common functionality that both other use cases need.
- This can be depicted on the use case model via an «*include*» *relationship*, as shown in Figure.
- This diagram states, for example, that the *Update Crop Encyclopedia* usecase includes the View Reports use case.
- This means that *View Reports* must be executed when *Update Crop Encyclopedia* is executed.
- *UpdateCrop Encyclopedia* would not be considered complete without View Reports.

## «extend» Relationships

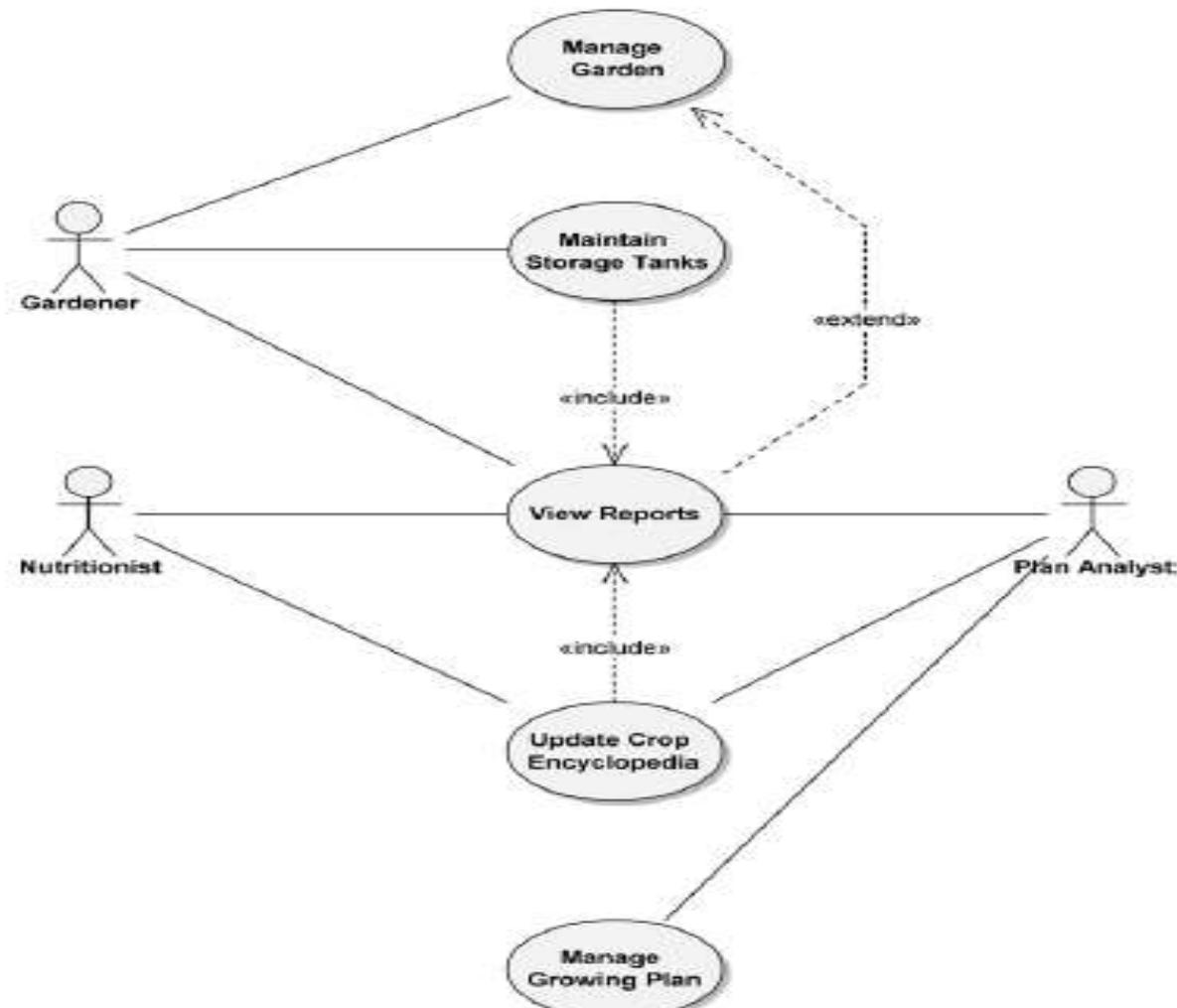


Figure 5–24 A Use Case Diagram Showing an «extend» Relationship

- While developing your use cases, you may find that certain activities might be performed as part of the use case but are not mandatory for that use case to run successfully.
- In our example, as the *Gardener* actor executes the *Manage Garden* use case, he or she may want to look at some reports.
  - This could be done by using the *View Reports* use case.
  - However, *View Reports* is not required when *Manage Garden* is run. *Manage Garden* is complete in and of itself. So, we modify the use case diagram to indicate that the *View Reports* use case extends the *Manage Garden* use case.

- Where an extending use case is executed, it is indicated in the use case specification as an extension point.
- The extension point specifies where, in the flow of the including use case, the extending use case is to be executed.

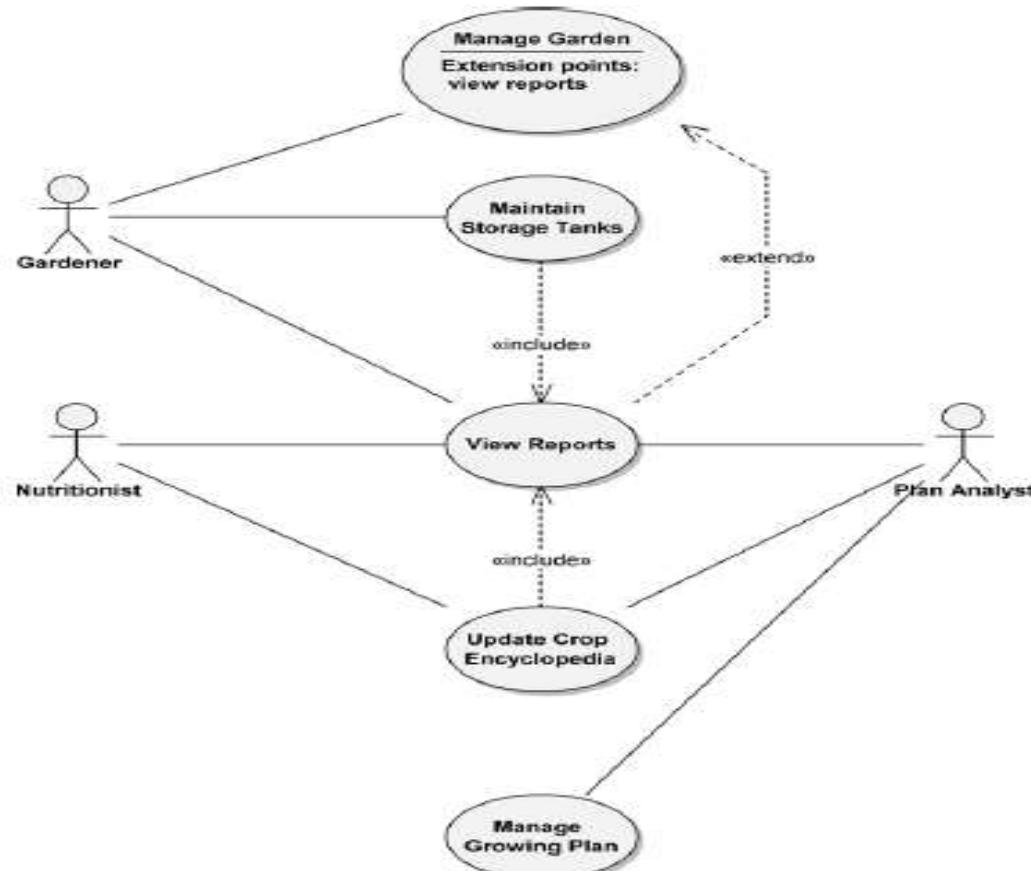


Figure 5–25 A Use Case Diagram Showing an Extension Point

**Table 5–1 Key Differences between «include» and «extend» Relationships<sup>a</sup>**

	Included Use Case	Extending Use Case
Is this use case optional?	No	Yes
Is the base use case complete without this use case?	No	Yes
Is the execution of this use case conditional?	No	Yes
Does this use case change the behavior of the base use case?	No	Yes

# **CONSTRUCTORS**

TYPES OF CONSTRUCTORS

# CONSTRUCTORS

---

- A constructor is a special member function whose task is to initialize the data members of an objects of its class.
- It is special because it has same name as its class name.
- It invokes automatically whenever a new object of its associated class is created.
- It is called constructor because it constructs the initial values of data members and build your programmatic object.



# CONSTRUCTORS

---

- It is very common for some part of an object to require initialization before it can be used.
  
- Suppose you are working on 100's of objects and the default value of a particular data member is needed to be zero.
  
- Initialising all objects manually will be very tedious job.
  
- Instead, you can define a constructor function which initialises that data member to zero. Then all you have to do is declare object and constructor will initialise object automatically



# CONSTRUCTORS

- While defining a constructor you must remember that the name of constructor will be same as the name of the class, and constructors will never have a return type.

```
class A
{
    public:
        int x;
        // constructor
        A()
        {
            // object initialization
        }
};
```



# CONSTRUCTORS

- Constructors can be defined either inside the class definition or outside class definition using class name and scope resolution :: operator.

```
class A
{
    public:
        int i;
        A(); // constructor declared
};

// constructor definition
A::A()
{
    i = 1;
}
```



# CONSTRUCTOR CHARACTERS

---

- They must be declared in the public scope.
- They are invoked automatically when the objects are created.
- They do not have return types, not even void and they cannot return values.
- They cannot be inherited, though a derived class can call the base class constructor.
- Like other C++ functions, Constructors can have default arguments.



# CONSTRUCTOR CHARACTERS

---

- Constructors cannot be virtual.
- We can not refer to their addresses.
- An object with a constructor (or destructor) can not be used as a member of a union.
- They make ‘implicit calls’ to the operators new and delete when memory allocation is required.



# CONSTRUCTOR TYPES

---

- Constructors are of three types:
  - Default Constructor
  - Parametrized Constructor
  - Copy Constructor



# DEFAULT CONSTRUCTOR

- Default constructor is the constructor which doesn't take any argument. It has no parameter.
- Syntax :

```
class_name(parameter1, parameter2, ...)  
{  
    // constructor Definition  
}
```



# DEFAULT CONSTRUCTOR

## Example

```
class Cube
{
public:
    int side;
    Cube()
    {
        side = 10;
    }
};

int main()
{
    Cube c;
    cout << c.side;
}
```

## Output : 10

# DEFAULT CONSTRUCTOR

---

- As soon as the object is created the constructor is called which initializes its data members.
  
- A default constructor is so important for initialization of object members, that even if we do not define a constructor explicitly, the compiler will provide a default constructor implicitly.



# DEFAULT CONSTRUCTOR

```
class Cube
{
public:
    int side;
};

int main()
{
    Cube c;
    cout << c.side;
}
```

Output: 0 or any random value

- In this case, default constructor provided by the compiler will be called which will initialize the object data members to default value, that will be 0 or any random integer value in this case.



# PARAMETERIZED CONSTRUCTOR

---

- These are the constructors with parameter.
- Using this Constructor you can provide different values to data members of different objects, by passing the appropriate values as argument.



# PARAMETERIZED CONSTRUCTOR

```
class Cube
{
public:
    int side;
    Cube(int x)
    {
        side=x;
    }
};

int main()
{
    Cube c1(10);
    Cube c2(20);
    Cube c3(30);
    cout << c1.side;
    cout << c2.side;
    cout << c3.side;
}
```

## OUTPUT

10

20

30

# PARAMETERIZED CONSTRUCTOR

```
class Cube
{
public:
int side;
Cube(int x)
{
    side=x;
}
};

int main()
{
    Cube c1(10);
    Cube c2(20);
    Cube c3(30);
    cout << c1.side;
    cout << c2.side;
    cout << c3.side;
}
```

- By using parameterized constructor in above case, we have initialized 3 objects with user defined values. We can have any number of parameters in a constructor.

# COPY CONSTRUCTOR

---

- These are special type of Constructors which takes an object as argument, and is used to copy values of data members of one object into other object.



# COPY CONSTRUCTOR

---

- These are special type of Constructors which takes an object as argument, and is used to copy values of data members of one object into other object.
  
- It is usually of the form **X (X&)**, where X is the class name. The compiler provides a default Copy Constructor to all the classes.

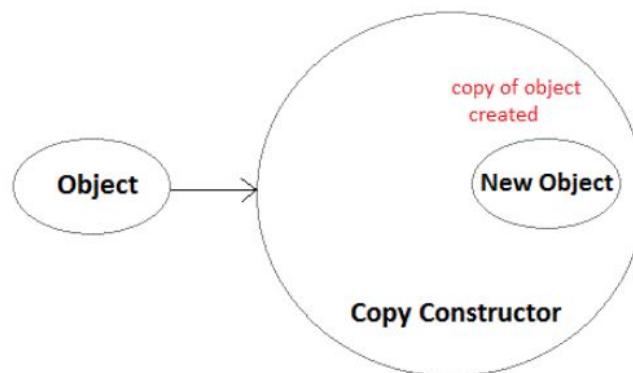


# COPY CONSTRUCTOR

---

```
classname(const classname & objectname)
{
    . . .
}
```

- As it is used to create an object, hence it is called a constructor. And, it creates a new object, which is exact copy of the existing copy, hence it is called **copy constructor**.



# COPY CONSTRUCTOR

```
#include<iostream>
using namespace std;
class Samplecopyconstructor
{
    private:
        int x, y; //data members

    public:
        Samplecopyconstructor(int x1, int y1)
        {
            x = x1;
            y = y1;
        }

        /* Copy constructor */
        Samplecopyconstructor (const Samplecopyconstructor &sam)
        {
            x = sam.x;
            y = sam.y;
        }

        void display()
        {
            cout<<x<<" "<<y<<endl;
        }
};
```



# COPY CONSTRUCTOR

```
/* main function */
int main()
{
    Samplecopyconstructor obj1(10, 15);      // Normal constructor
    Samplecopyconstructor obj2 = obj1;        // Copy constructor
    cout<<"Normal constructor : ";
    obj1.display();
    cout<<"Copy constructor : ";
    obj2.display();
    return 0;
}
```

## □ Output :

- Normal constructor : 10 15
- Copy constructor : 10 15

# STATIC CONSTRUCTOR

---

- C++ doesn't have static constructors but you can emulate them using a static instance of a nested class.

```
class has_static_constructor {  
    friend class constructor;  
    struct constructor {  
        constructor() { /* do some constructing here ... */ }  
    };  
    static constructor cons;  
};
```

```
// C++ needs to define static members externally.  
has_static_constructor::constructor has_static_constructor::cons;
```



# **OPERATOR OVERLOADING**

# Overloading in C++

## ✓ What is overloading?

- Overloading means assigning multiple meanings to a function name or operator symbol
- It allows multiple definitions of a function with the same name, but different signatures.

## ✓ C++ supports

- Function overloading
- Operator overloading

# Why is Overloading Useful?

- Function overloading allows functions that conceptually perform the same task on objects of different types to be given the same name.
- Operator overloading provides a convenient notation for manipulating user-defined objects with conventional operators.

# Operator Overloading

- Operator overloading
  - Enabling C++'s operators to work with class objects
  - Using traditional operators with user-defined objects
  - Requires great care; when overloading is misused, program difficult to understand

# Operator Overloading

- Examples of already overloaded operators
  - Operator `<<` is both the stream-insertion operator and the bitwise left-shift operator
  - `+` and `-`, perform arithmetic on multiple types
- Compiler generates the appropriate code based on the manner in which the operator is used

# General Rules for Operator Overloading

- Overloading an operator - Write function definition as normal

Syntax:

```
returnType classname::operator op(arguments)
```

```
{
```

Function body;

```
}
```

Function name is keyword **operator** followed by the symbol for the operator being overloaded

# General Rules for Operator Overloading

- Using operators
  - To use an operator on a class object it must be overloaded unless the assignment operator (=) or the address operator (&)
    - Assignment operator by default performs memberwise assignment
    - Address operator (&) by default returns the address of an object

# Steps

1. Create a class that is to be used
2. Declare the operator function in the public part of the class. It may be either member function or friend function.
3. Define operator function to implement the operation required
4. Overloaded can be invoked using the syntax such as:     **op x;**

# Restrictions on Operator Overloading

Following C++ Operator can't be overloaded

- Class member access operators(. & .\*)
- Scope Resolution Operator(::)
- Sizeof Operator(sizeof())
- Conditional Operator(? :)

# Restrictions on Operator Overloading

- Overloading restrictions
  - Precedence ,associativity, arity (number of operands) of an operator cannot be changed
  - No new operators can be created
    - Use only existing operators
  - No overloading operators for built-in types
    - Cannot change how two integers are added

# Operator Functions as Class Members vs. as friend Functions

In general, operator functions can be member or non-member functions

- Operator functions as member functions
  - Leftmost operand must be an object (or reference to an object) of the class
- Operator functions as non-member friend functions
  - Must be **friends** if needs to access private or protected members

## Overloadable Operators

<b>+</b>	<b>-</b>	<b>*</b>	<b>/</b>	<b>%</b>	<b>^</b>
<b>&amp;</b>	<b> </b>	<b>~</b>	<b>!</b>	<b>,</b>	<b>=</b>
<b>&lt;</b>	<b>&gt;</b>	<b>&lt;=</b>	<b>&gt;=</b>	<b>++</b>	<b>--</b>
<b>&amp;&lt;</b>	<b>&gt;&gt;</b>	<b>==</b>	<b>!=</b>	<b>&amp;&amp;</b>	<b>  </b>
<b>+=</b>	<b>=</b>	<b>/=</b>	<b>%=</b>	<b>^=</b>	<b>&amp;=</b>
<b>/=</b>	<b>*=</b>	<b>&lt;&lt;=</b>	<b>&gt;&gt;=</b>	<b>[]</b>	<b>()</b>
<b>-&gt;</b>	<b>-&gt;*</b>	<b>new</b>	<b>new []</b>	<b>delete</b>	<b>delete []</b>

## Non-overloadable Operators

<b>::</b>	<b>.*</b>	<b>.</b>	<b>?:</b>
-----------	-----------	----------	-----------

- Operator Overloading can be done by using **three approaches**, they are
  1. Overloading unary operator.
  2. Overloading binary operator.
  3. Overloading binary operator using a friend function.

# Function Overloading

- Is the process of using the same name for two or more functions
- Requires each redefinition of a function to use a different function signature that is:
  - different types of parameters,
  - or sequence of parameters,
  - or number of parameters
- Is used so that a programmer does not have to remember multiple function names

# Function Overloading

- Two or more functions can have the same name but different parameters
- Example:

```
int max(int a, int b)
{
    if (a>= b)
        return a;
    else
        return b;
}
```

```
float max(float a, float b)
{
    if (a>= b)
        return a;
    else
        return b;
}
```

# Overloading Function Call Resolution

- Overloaded function call resolution is done by compiler during compilation
  - The function signature determines which definition is used
- a Function signature consists of:
  - Parameter types and number of parameters supplied to a function
- a Function return type is not part of function signature and is not used in function call resolution

## Function overloading example:

```
void sum(int,int);
void sum(double,double);
void sum(char,char);
void main()
{
int a=10,b=20 ;
double c=7.52,d=8.14;
char e='a' , f='b' ;
sum(a,b);
//calls sum(int x,int y)
sum(c,d);
// calls sum (double x,double y)
sum(e,f);
// calls sum(char x,char y)
}
void sum(int x,int y)
{
vout<<“\n sum of integers are”<<x+y;
}
```

## **Function overloading example:**

```
#include<iostream>
```

```
Using namespace std;
```

```
void area(int x)
{ cout<<“area is”<<x*x;
}
void area(int x,int y)
{cout<<“area of rectang;e”=<<x*y;
}
void area(int x,int y,int z)
{cout<<“volume is”<<x*y*z;
}
int main()
{
int side=10,le=5,br=6,a=4,b=5,c=6;
area(side);
area(le,br);
area(a,b,c);
return 0;
}
```

# Function Selection Involves following Steps.

- Compiler first tries to find the Exact match in which the type of argument are the same, and uses that func.
- If an exact match is not found, the compiler user the integral promotions to the actual argument such as, char to int, float to double.
- When either of them fails , build in conversions are used (implicit conversion) to the actual arguments and then uses the function whose match is unique. But if there are multiple matches, then compiler will generate an error message.

- For ex: long square(long n)  
long square(double x)
  - Now a func. call such as **square(10)** will cause an error because int argument can be converted into long also and double also.so it will show ambiguity.
  - User defined conversion are followed if all the conversion are failed.

# Overload Member function

```
#include<iostream.h>
#include<stdlib.h>
#include<math.h>
#include<conio.h>
class absv
{
public :
    int num (int) ;
    double num (double) ;
};
int absv:: num (int x)
{
    int ans;
    ans=abs (x) ;
    return (ans) ;
}
```

```
double absv :: num (double d)
{
    double ans;
    ans=fabs (d) ;
    return (ans) ;
}
int main()
{
    clrscr () ;
    absv n;
    cout<<"\n Absolute value of
-25 is "<<n.num (-25) ;
    cout<<"\n Absolute value of
-25.1474 is "<<n.num (-25.1474)
    return 0;
}
```

Takes sample property as member function.

Note : Need to be explained with all function overloading concept.

# Overloading the assignment operator

- The assignment operator (`operator=`) is used to copy values from one object to another *already existing object*.

## Assignment vs Copy constructor

- The purpose of the copy constructor and the assignment operator are almost equivalent -- both copy one object to another. However, the copy constructor initializes new objects, whereas the assignment operator replaces the contents of existing objects.
- The difference between the copy constructor and the assignment operator causes a lot of confusion for new programmers, but it's really not all that difficult. Summarizing:
  - If a new object has to be created before the copying can occur, the copy constructor is used (note: this includes passing or returning objects by value).
  - If a new object does not have to be created before the copying can occur, the assignment operator is used.

```
#include <iostream>
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; }
void operator = (const Distance &D )
{ feet = D.feet; inches = D.inches; }
// method to display distance
void displayDistance()
{ cout << "F: " << feet << " I:" << inches << endl;
} }
int main()
{ Distance D1(11, 10), D2(5, 11);
cout << "First Distance : "; D1.displayDistance();
cout << "Second Distance :"; D2.displayDistance();
// use assignment operator
D1 = D2; cout << "First Distance : ";
D1.displayDistance(); return 0; }
```

### Output:

First Distance : F: 11 I:10

Second Distance :F: 5 I:11

First Distance :F: 5 I:11

# **INHERITANCE AND ITS TYPES**

# Introduction to Inheritance

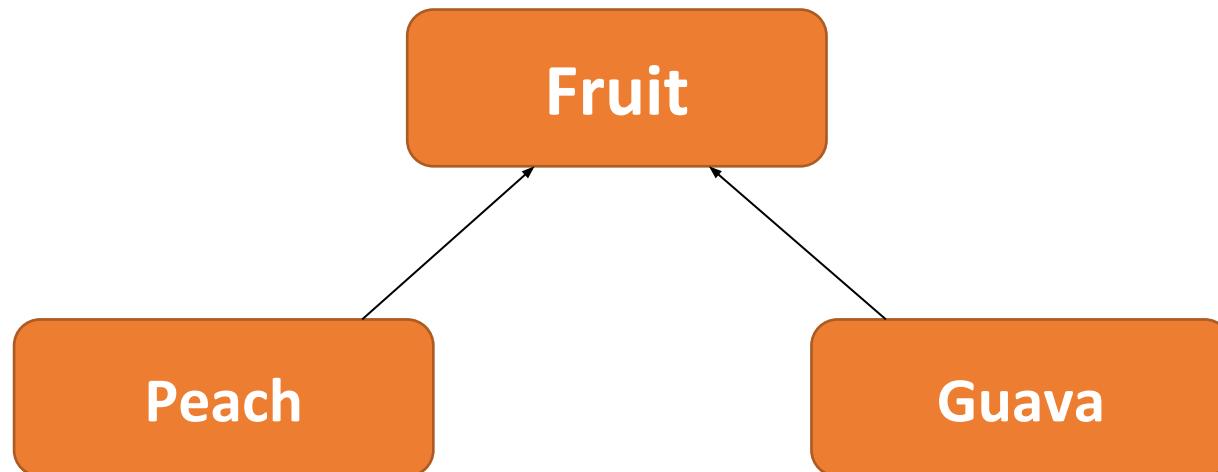
- Inheritance is the process by which objects of one class obtain the properties of objects of another class in the hierarchy.
- Subclasses provide specialized behavior from the basis of common elements provided by the super class.
- Through the use of inheritance, programmers can reuse the code in the super class many times.
- Once a parent class is written and debugged, it need not be touched again but at the same time can be revised to work in different situations.
- Reusing existing code saves time and money and increases a program's reliability.

## Cont.

- For example-1, the scooter is a type of the class two-wheelers, which is again a type of (or kind of) the class motor vehicles
- For example-2, all fruits have a name, a color, and a size. Therefore, peaches and Guavas also have a name, a color, and a size.
- We can say that peaches and Guavas inherit (acquire) these all of the properties of fruit because they are fruit.
- We also know that fruit undergoes a ripening process, by which it becomes edible.
- Because peaches and Guavas are fruit, we also know that peaches and Guavas will inherit the behavior of ripening.

# Cont.

- Put into a diagram, the relationship between peaches, Guavas, and fruit might look something like this:



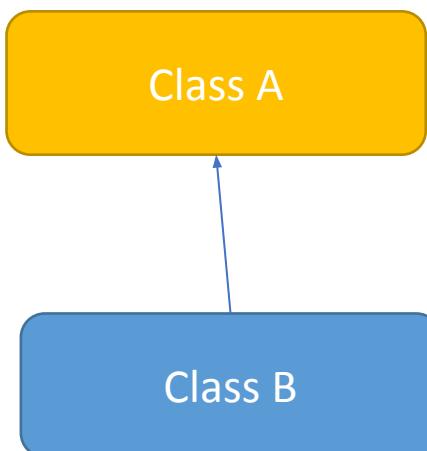
# Types of Inheritance

- Single Inheritance
- Multiple Inheritance
- Multilevel Inheritance
- Hierarchical Inheritance
- Hybrid Inheritance

# Single Inheritance

- In single inheritance, a class derives from one base class only. This means that there is only one subclass that is derived from one superclass.
- Syntax:

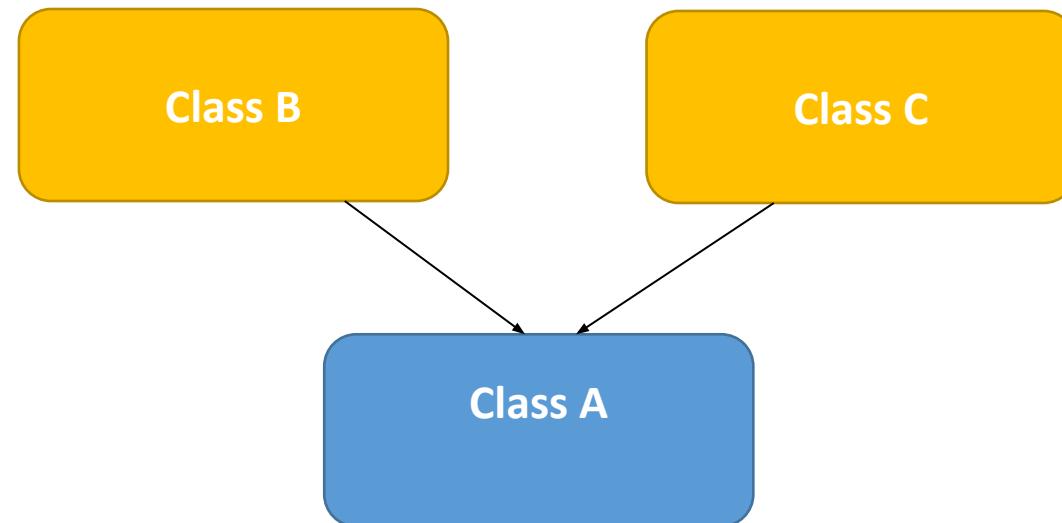
```
class subclass_name : accessSpecifier base_class  
{ //body of subclass };
```



# Multiple Inheritance

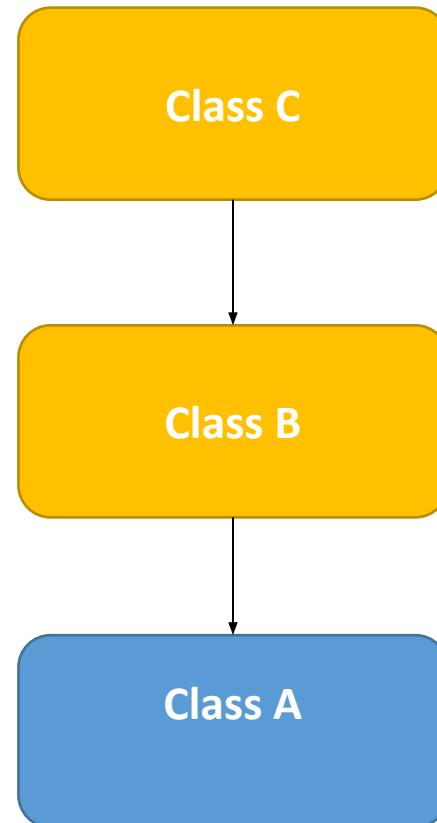
- A class can inherit properties from more than one class which is known as multiple inheritance.
- Syntax:

```
class subclass_name : accessSpecifier base_class1, access_mode  
base_class2, ....  
{ //body of subclass };
```



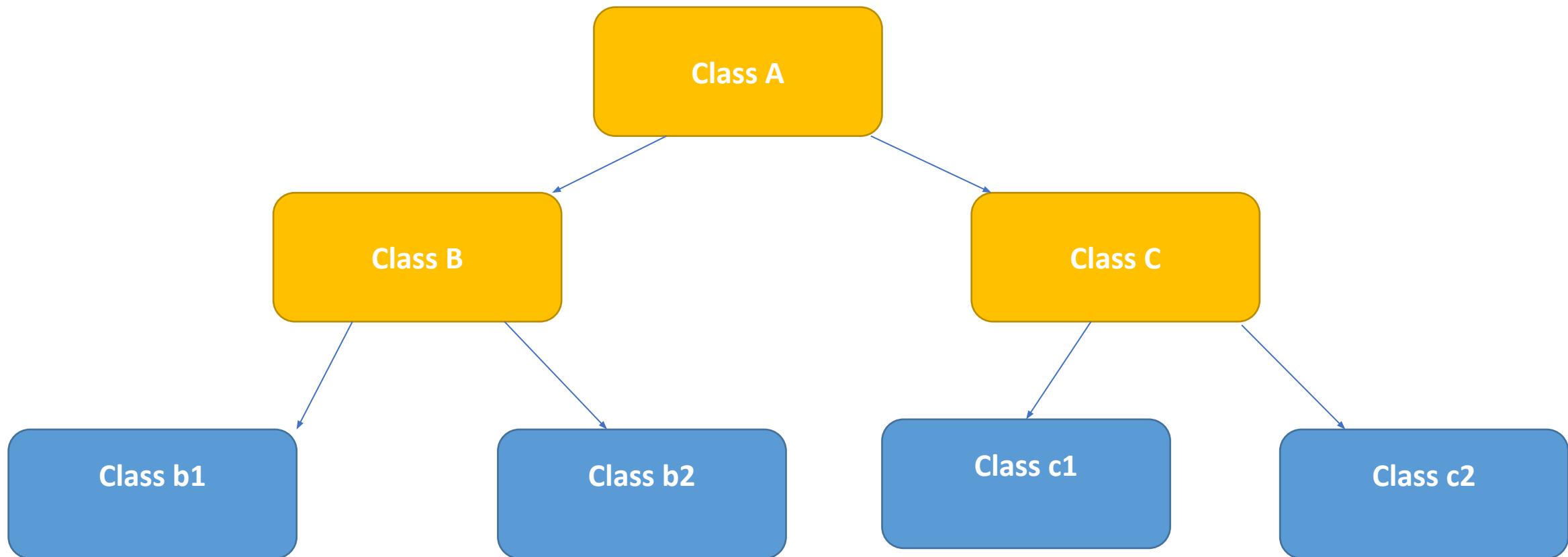
# Multilevel Inheritance

- A class can be derived from another derived class which is known as multilevel inheritance.



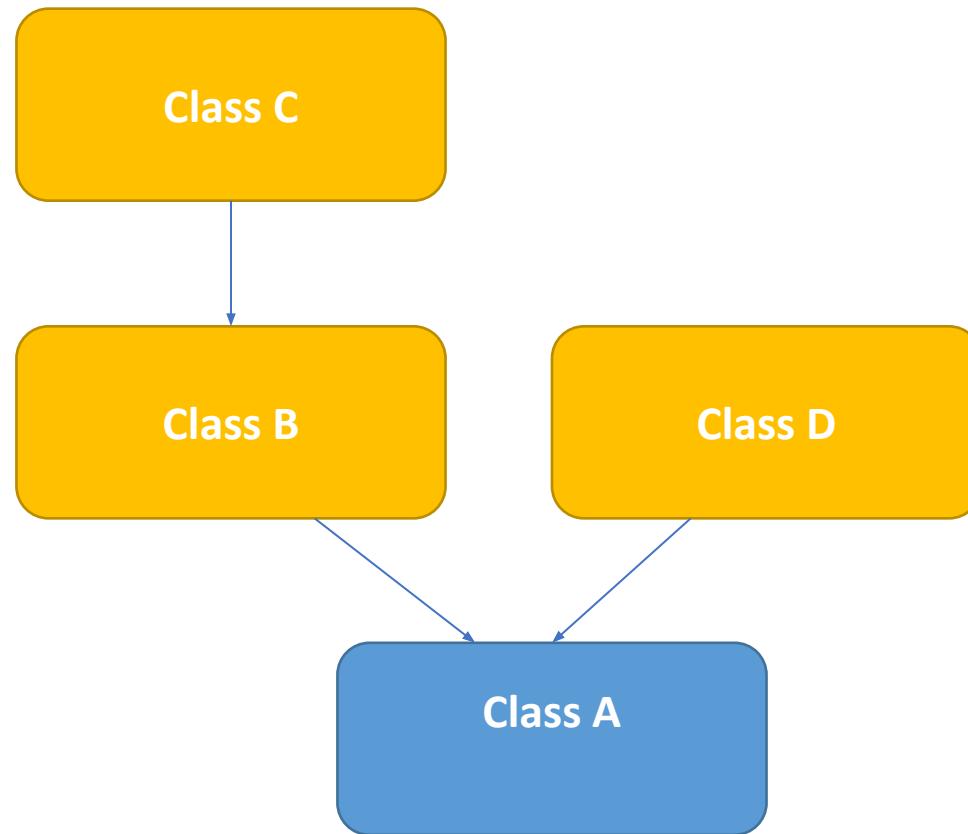
# Hierarchical Inheritance

- In this type of inheritance, one parent class has more than one child class



# Hybrid Inheritance

- There could be situations where we need to apply two or more types of inheritance to design one inheritance called hybrid inheritance.



# INTERACTION DIAGRAM

# Definition:

- Interaction Shows an Arc, consisting of a set of objects and their relationships, including the messages that may be dispatched among them.
- Interaction diagram address the dynamic view of the system.

# Purpose of Interaction Diagram:

The purpose of interaction diagram is :

- Interaction diagrams are used to observe the dynamic behavior of a system.
- Interaction diagram visualizes the communication and sequence of message passing in the system.
- Interaction diagram represents the structural aspects of various objects in the system.
- Interaction diagram represents the ordered sequence of interactions within a system.
- Interaction diagram provides the means of visualizing the real time data via UML.

- This interactive behavior is represented in UML by two diagrams known as
  - **Sequence diagram**
  - **Collaboration diagram.**
- Sequence diagram emphasizes on time sequence of messages from one object to another.
- Collaboration diagram emphasizes on the structural organization of the objects that send and receive messages.

# How to Draw an Interaction Diagram?

- The purpose of interaction diagrams is to capture the dynamic aspect of a system. So to capture the dynamic aspect, we need to understand what a dynamic aspect is and how it is visualized. Dynamic aspect can be defined as the snapshot of the running system at a particular moment.
- Following things are to be identified clearly before drawing the interaction diagram
  - Objects taking part in the interaction.
  - Message flows among the objects.
  - The sequence in which the messages are flowing.
  - Object organization.

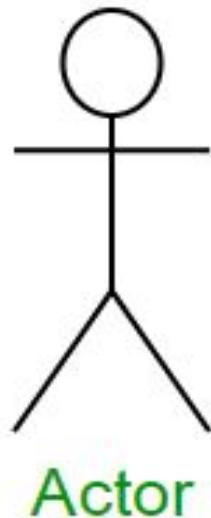
# Sequence Diagram:

- A sequence diagram simply depicts interaction between objects in a sequential order i.e. the order in which these interactions take place.

# Sequence Diagram Notations

## 1. Actors :

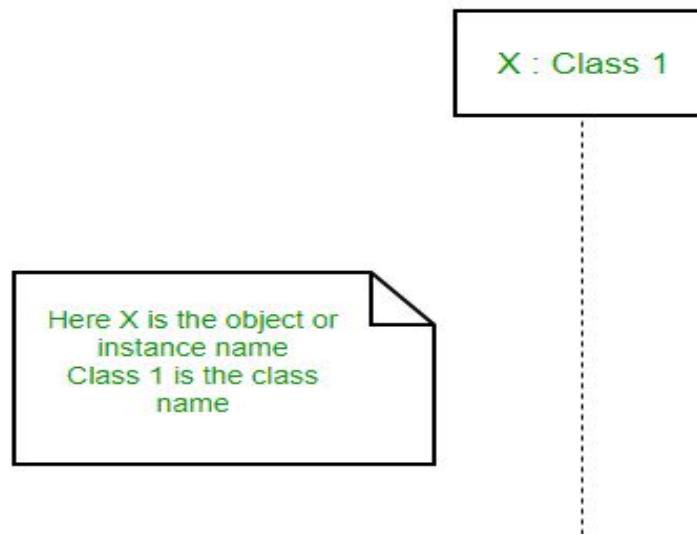
An actor in a UML diagram represents a type of role where it interacts with the system and its objects.



## 2.Lifelines :

- A lifeline is a named element which depicts an individual participant in a sequence diagram. So basically each instance in a sequence diagram is represented by a lifeline.
- Lifeline elements are located at the top in a sequence diagram.
- lifeline follows the following format :

Instance Name : Class Name

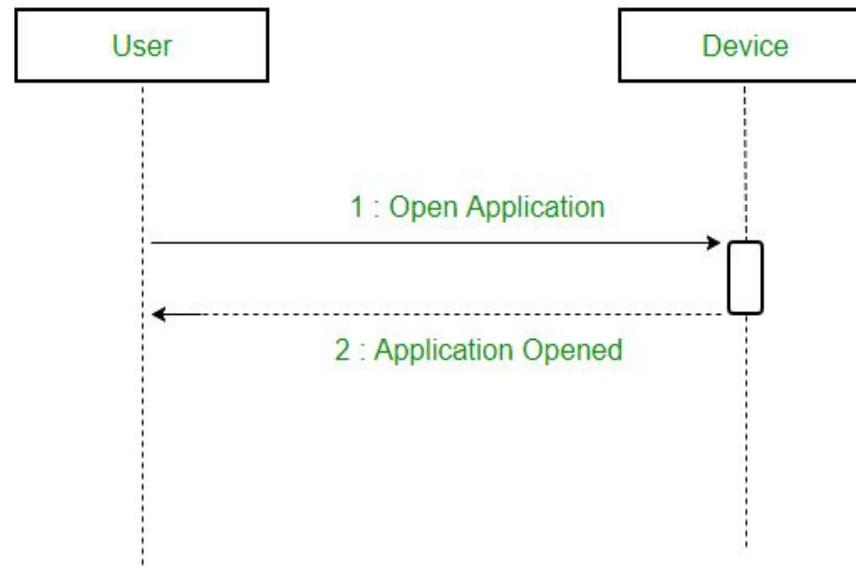


### **3.Messages :**

- Communication between objects is depicted using messages. The messages appear in a sequential order on the lifeline.
- We represent messages using arrows. Lifelines and messages form the core of a sequence diagram.

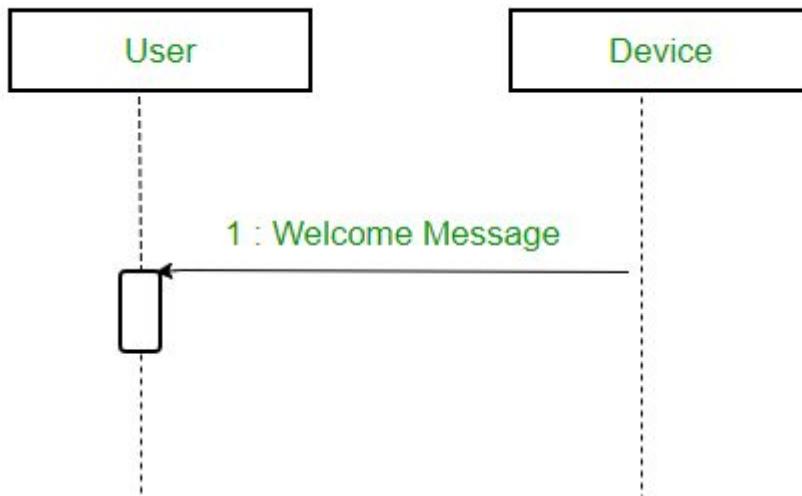
# (i)Synchronous messages

- A synchronous message waits for a reply before the interaction can move forward.
- The sender waits until the receiver has completed the processing of the message.
- The caller continues only when it knows that the receiver has processed the previous message i.e. it receives a reply message.
- A large number of calls in object oriented programming are synchronous. We use a solid arrow head to represent a synchronous message.



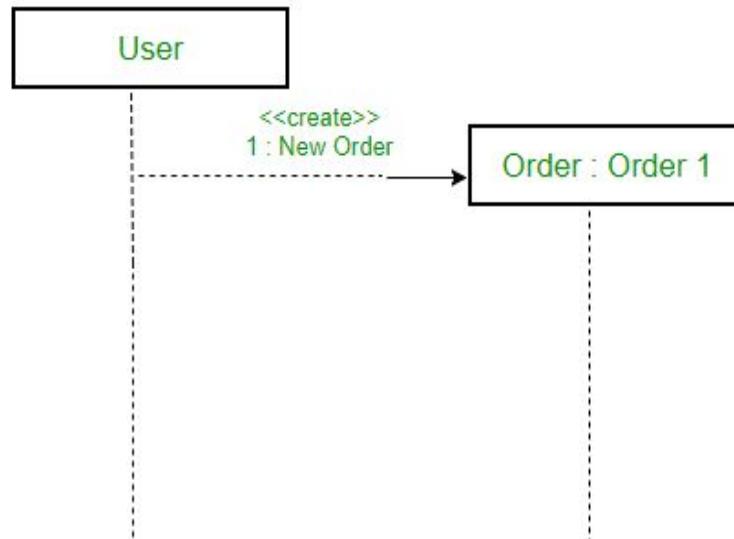
## (ii)Asynchronous Messages

- An asynchronous message does not wait for a reply from the receiver.
- The interaction moves forward irrespective of the receiver processing the previous message or not.
- We use a lined arrow head to represent an asynchronous message.



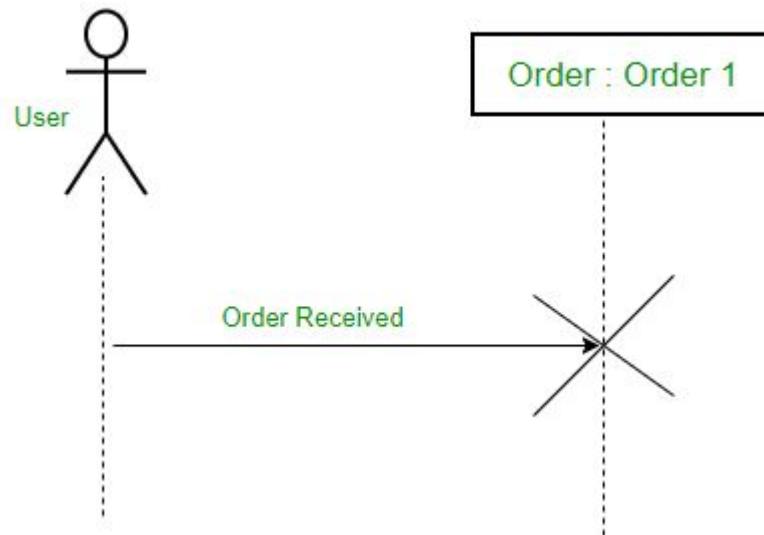
# (iii) Create message

- We use a Create message to instantiate a new object in the sequence diagram.
- It is represented with a dotted arrow and create word labeled on it to specify that it is the create Message symbol.  
For example :
  - The creation of a new order on a e-commerce website would require a new object of Order class to be created.



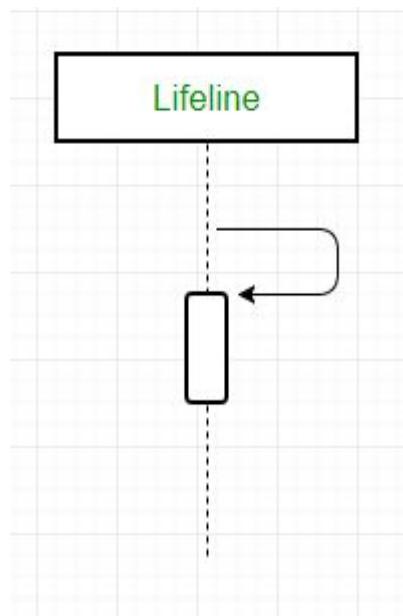
# (iv) Delete Message

- We use a Delete Message to delete an object.
  - It destroys the occurrence of the object in the system.
  - It is represented by an arrow terminating with a x.
- For example – In the scenario below when the order is received by the user, the object of order class can be destroyed



# (v) Self Message

- A message an object sends to itself, usually shown as a U shaped arrow pointing back to itself.



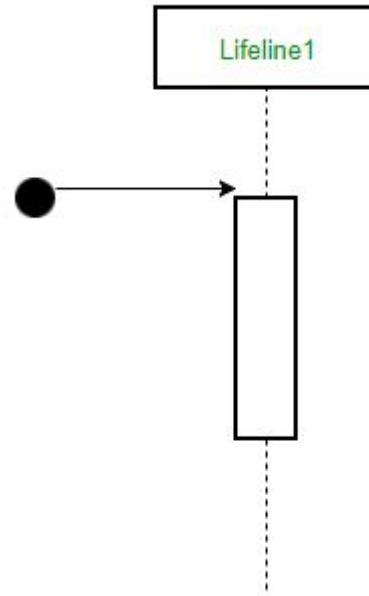
## (vi) Reply Message

- Reply messages are used to show the message being sent from the receiver to the sender.
- We represent a return/reply message using an open arrowhead with a dotted line.
- The interaction moves forward only when a reply message is sent by the receiver.



# (vii) Found Message

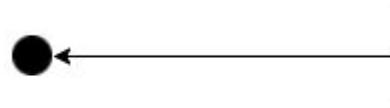
- A Found message is used to represent a scenario where an unknown source sends the message.
- It is represented using an arrow directed towards a lifeline from an end point.



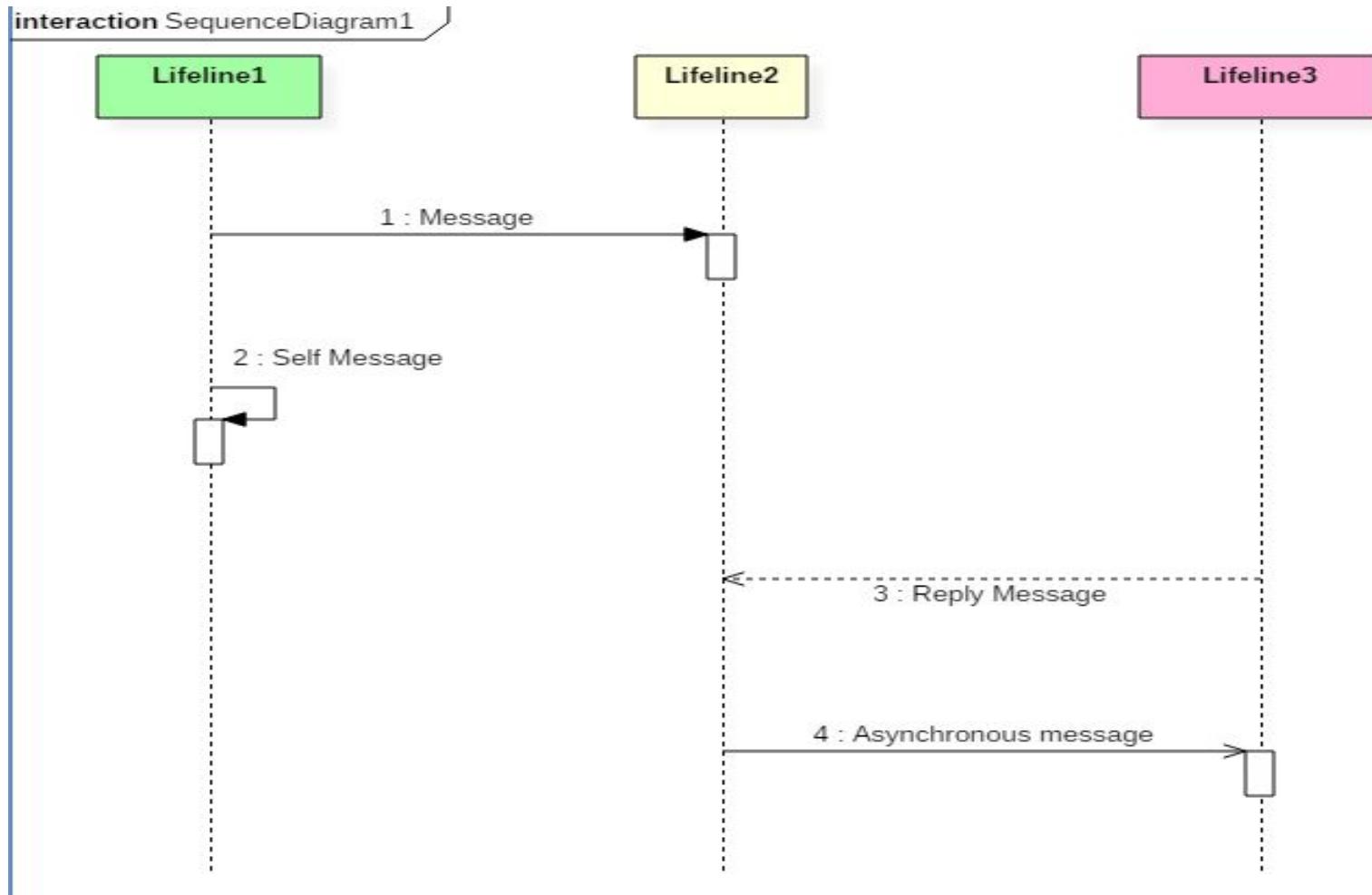
# (viii) Lost Message

- A Lost message is used to represent a scenario where the recipient is not known to the system.
- It is represented using an arrow directed towards an end point from a lifeline.

For example:



# Sample example:



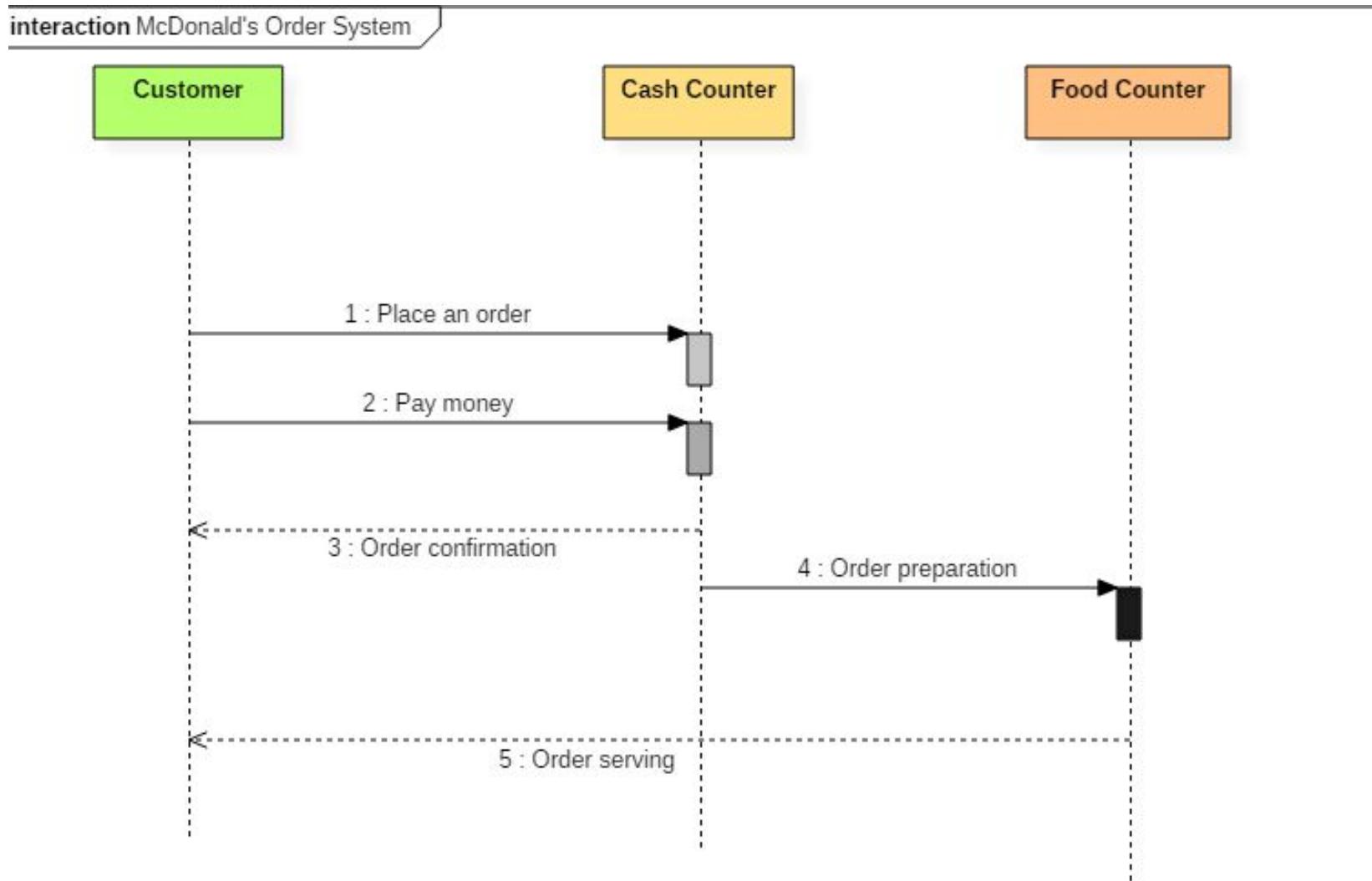
The above sequence diagram contains lifeline notations and notation of various messages used in a sequence diagram such as a create, reply, asynchronous message, etc.

# Example:

- The ordered sequence of events in a given sequence diagram is as follows:
  - Place an order.
  - Pay money to the cash counter.
  - Order Confirmation.
  - Order preparation.
  - Order serving.

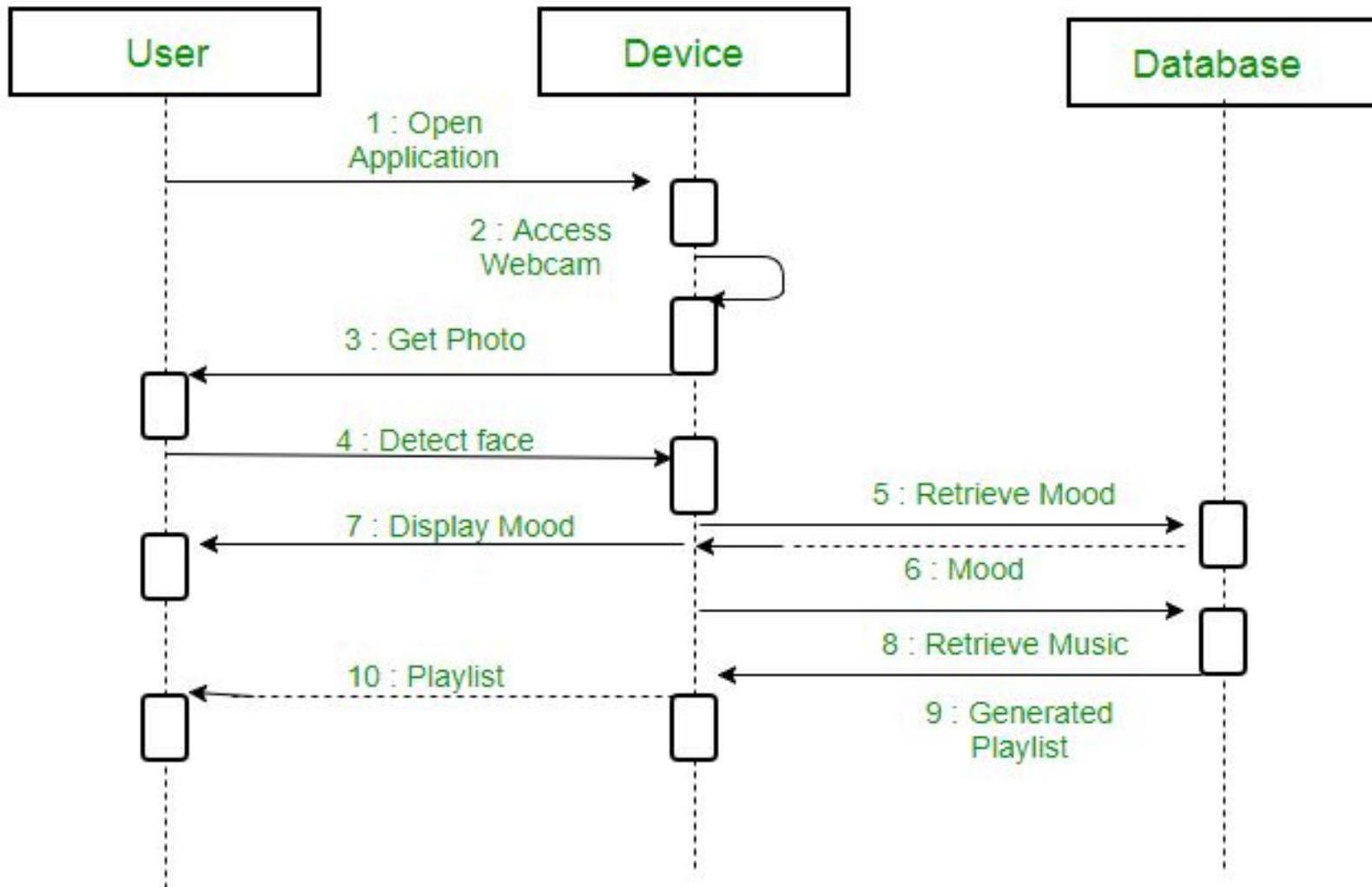
# Sequence diagram example(1)

The following sequence diagram example represents McDonald's ordering system:



# Example(2)

A sequence diagram for an emotion based music player



- Firstly the application is opened by the user.
- The device then gets access to the web cam.
- The webcam captures the image of the user.
- The device uses algorithms to detect the face and predict the mood.
- It then requests database for dictionary of possible moods.
- The mood is retrieved from the database.
- The mood is displayed to the user.
- The music is requested from the database.
- The playlist is generated and finally shown to the user.

# Benefits of a Sequence Diagram

- Sequence diagrams are used to explore any real application or a system.
- Sequence diagrams are used to represent message flow from one object to another object.
- Sequence diagrams are easier to maintain.
- Sequence diagrams are easier to generate.
- Sequence diagrams can be easily updated according to the changes within a system.
- Sequence diagram allows reverse as well as forward engineering.

# Drawbacks of a sequence diagram

- Sequence diagrams can become complex when too many lifelines are involved in the system.
- If the order of message sequence is changed, then incorrect results are produced.
- Each sequence needs to be represented using different message notation, which can be a little complex.
- The type of message decides the type of sequence inside the diagram.

# Collaboration diagram

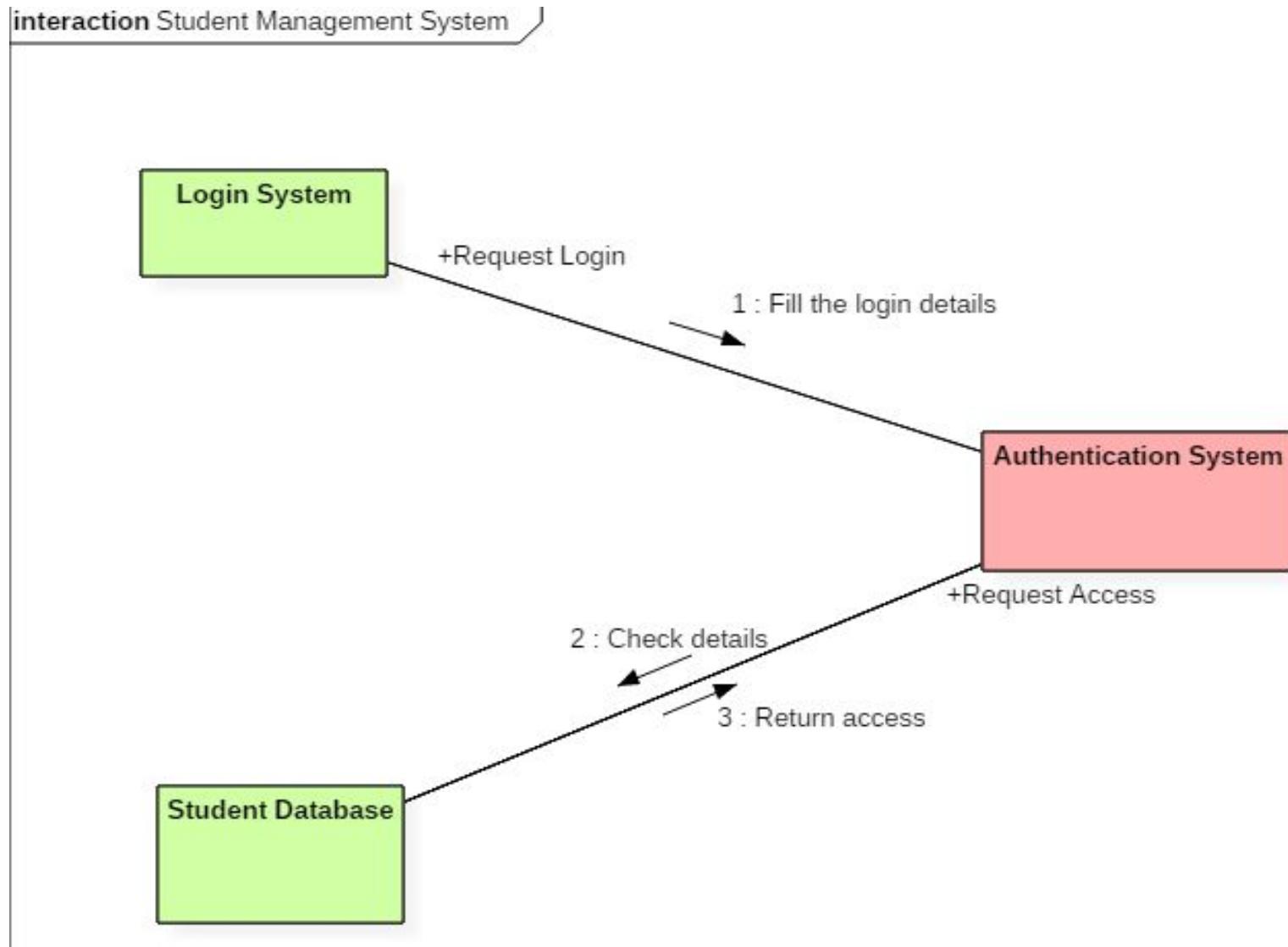
- **COLLABORATION DIAGRAM** depicts the relationships and interactions among software objects. They are used to understand the object architecture within a system rather than the flow of a message as in a sequence diagram. They are also known as “Communication Diagrams.”
- In the collaboration diagram, the method call sequence is indicated by some numbering technique. The number indicates how the methods are called one after another.

# Benefits of Collaboration Diagram

- It is also called as a communication diagram.
- It emphasizes the structural aspects of an interaction diagram - how lifeline connects.
- Its syntax is similar to that of sequence diagram except that lifeline don't have tails.
- Messages passed over sequencing is indicated by numbering each message hierarchically.
- Compared to the sequence diagram communication diagram is semantically weak.
- Object diagrams are special case of communication diagram.
- It allows you to focus on the elements rather than focusing on the message flow as described in the sequence diagram.
- Sequence diagrams can be easily converted into a collaboration diagram as collaboration diagrams are not very expressive.

# Example

Following diagram represents the sequencing over student management system:



- The above collaboration diagram represents a student information management system. The flow of communication in the above diagram is given by,
  - A student requests a login through the login system.
  - An authentication mechanism of software checks the request.
  - If a student entry exists in the database, then the access is allowed; otherwise, an error is returned.



# METHOD OVERLOADING

- Method overloading is a feature in C++ that allows creation of several methods with the same name but with different parameters.
- For example, print(), print(int), and print("Hello") are overloaded methods.
- While calling print() , no arguments are passed to the function
- When calling print(int) and print("Hello") , an integer and a string arguments are passed to the called function.
- Allows one function to perform different tasks

# METHOD OVERLOADING

- Basically, there are two types of polymorphism:
  - **Compile time (or static)** polymorphism
  - **Run-time (or dynamic)** polymorphism.
- Static polymorphism -> Method overloading - calls a function using the best match technique or overload resolution.

# MATCHING FUNCTION CALLS WITH OVERLOADED METHODS

When an overloaded function is called, one of the following cases occurs:

- Case 1: A **direct match is found**, and there is no confusion in calling the appropriate overloaded function.
- Case 2: If a **match is not found**, a linker error will be generated. However, if a direct match is not found, then, at first, the compiler will try to find a match through the type conversion or type casting.
- Case 3: If an **ambiguous match is found**, that is, when the arguments match more than one overloaded function, a compiler error will be generated. This usually happens because all standard conversions are treated equal.

```
void print(int); // Function declaration
print('R');    // Function call
```

**Example 4.19** To demonstrate the ambiguity in function call

```
#include<iostream.h>
void print(int n){ cout<<n; }
void print(char c){ cout<<c; }
void print(float f){ cout<<f; }
main()
{   print(5);    //Function call
    print (98.7);
    print('R');
}
```

**OUTPUT**

Error

**Example 4.20** To compute the volume of different shapes using function overloading concept

```
#include<iostream.h>
int volume(int side)
{
    return side*side*side; // cube
}
float volume(float radius, float height)
{
    return 3.14+radius*radius*height; //cylinder
}
long int volume(int length, int breadth, int height)
{
    return length*breadth*height; //cuboid
}
main()
{
    int s,l,b,height;
    float r, h;
    cout<<"\n Enter the side of the cube : ";
    cin>>s;
    cout<<"\n Volume of cube with side "<<s<<" = "<<volume(s);
    cout<<"\n \n Enter the radius and height of the cylinder : ";
    cin>>r>>h;
    cout<<"\n Volume of cylinder with radius "<<r<<" and height "<<h<<" = "<<volume(r,h);
    cout<<"\n Enter the length, breadth and height of the cuboid : ";
    cin>>l>>b>>height;
    cout<<"\n Volume of cuboid with length "<<l<<" breadth "<<b<<" and height
    "<<height<<" = "<<volume(l,b,height);
}
```

**OUTPUT**

```
Enter the side of the cube : 3
Volume of cube with side 3 = 27
Enter the radius and height of the cylinder : 3 4
Volume of cylinder with radius 3 and height 4 = 39.13
Enter the length, breadth and height of the cuboid : 2 3 4";
Volume of cuboid with length 2 breadth 3 and height 4 = 24
```

# Methods that Cannot be Overloaded

1. Methods that differ only in the return type
2. Parameter declarations that differ only in a pointer \* versus an array []
3. If parameters differ only in the presence or absence of const and/or volatile
4. Using typedef does not introduce a new type, hence cannot be overloaded
5. Methods that differ only in their default arguments
6. Methods that differ only in a reference parameter and a normal parameter

# Method overloading- Points to Remember

1. There are multiple definitions for the same function name in the same scope
2. The definitions of these functions vary according to the types and/or the number of arguments in the argument list
3. Data type of the return value is not considered while writing overloaded functions because the appropriate function is called at the compile time while the return value will be obtained only when the function is called and executed

# **UNARY OPERATOR OVERLOADING IN C++**

# INTRODUCTION

- The utility of operators such as +, =, \*, /, >, <, and so on is predefined in any programming language.
- Programmers can use them directly on built-in data types to write their programs.
- However, these operators do not work for user-defined types such as objects.
- Therefore, C++ allows programmers to redefine the meaning of operators when they operate on class objects. This feature is called *operator overloading*

# INTRODUCTION

## Operator Overloading:

- **Operator** – It is a symbol that indicates an operation. Arithmetic operators are + (add two numbers), - (subtract two numbers), \* ( Multiply two numbers), / ( Divide between two numbers).
- At now, we will take an Addition ‘+’ Sign, its use of ‘+’ sign is

$$5+5=10$$

$$2.5+2.5=5$$

# INTRODUCTION

- ❖ **Operator Overloading** means multiple functions or multiple jobs. In operator overloading the ‘+’ sign use of add the two objects.
- ❖ One of C++’s great features is its extensibility, Operator Overloading is major functionality related to extensibility.
- ❖ In C++, most of operators can be overloaded so that they can perform special operations relative to the classes you create.

# INTRODUCTION

- ❖ **For Example,** ‘+’ operator can be overloaded to perform an operation of string concatenation along with its pre-defined job of adding two numeric values.
- ❖ When an operator is overloaded, none of its original meaning will be lost.
- ❖ After overloading the appropriate operators, you can use C++’s built in data types.

# **TYPES OF OPERATOR**

## **□ Unary Operator**

- Operators attached to a single operand.

( $-a$ ,  $+a$ ,  $--a$ ,  $++a$ ,  $a--$ ,  $a++$ )

## **□ Binary Operator**

- Operators attached to two operand.

( $a-b$ ,  $a+b$ ,  $a*b$ ,  $a/b$ ,  $a\%b$ ,  $a>b$ ,  $a<b$  )

# Syntax

return-type class-name:: operator op(arg-list)

{

function body

}

## EXPLANATION

- ❖ return type – It is the type of value returned by the specified operation.
- ❖ op - It is the operator being overloaded. It may be unary or binary operator. It is preceded by the keyword operator.
- ❖ operator op - It is the function name, Where operator is a keyword.

# UNARY OPERATOR OVERLOADING

## Introduction

- One of the exciting features of C++
- Works only on the single variable
- It can be overloaded two ways
  1. Static member function
  2. Friend function
- `-,+,++,--` those are unary operator which we can overloaded.

# Using a member function to Overload Unary Operator

```
#include<iostream.h>
class Number
{
    private:
        int x;
    public:
        Number()
        {   x = 0;      }
        Number(int n) //parameterized constructor
        {   x = n;      }
        void operator -() // operator overloaded function
        {   x = -x;  }
        void show_data()
        {   cout<<"\n x = "<<x;      }
};
main()
{   Number N(7);    // create object
    N.show_data();
    -N;           // invoke operator overloaded function
    N.show_data();
}
```

## OUTPUT

```
x = 7
x = -7
```

# Using a Friend Function to Overload a Unary Operator

- The function will take one operand as an argument.
- This operand will be an object of the class.
- The function will use the private members of the class only with the object name.
- The function may take the object by using value or by reference.
- The function may or may not return any value.
- The friend function does not have access to the this pointer.

## Example:- Use of friend function to overload a unary operator

```
#include<iostream.h>
class Number
{ private:
    int x;
public:
    Number()
    {   x = 0;   }
    Number(int n)
    {   x = n;   }
    void show_data()
    {   cout<<"\n x = "<<x;   }
    friend Number & operator-(Number &); //friend function declared
};

Number & operator -(Number &N)
{   N.x = -N.x;           //use objectname with data member
    return N;             //return object
}

main()
{   Number N1(100), N2;
    N2 = -N1;           // overloaded operator function called
    N2.show_data();
}
```

### OUTPUT

x = -100

# UNARY OPERATOR OVERLOADING

## Example Program:

**Write a program which will convert all positive values in an object to negative value.**

### Code:

```
#include <iostream>
class demo
{
int
x,y,z;
public:
void getdata (int a, int b,int c)
{
x=a;
y=b;
z=c;
}
```

**Contd...,**

```
void display();
void operator -();
};

void demo::display()
{
cout<<“x=“<<x<<“\ny=“<<y<<“\nz=“<<z<<endl;
}

void demo::operator -()
{
x=-x;
y=-y;
z=-z;
}
int main()
{
demo obj1;
```

**CONTD...,**

```
obj1.getdata(10,20,30);
obj1.display();
-obj1;
obj1.display();
return 0;
}
```

**Output:**

x=10  
y=20  
z=30

x=-10  
y=-20  
z=-30

# **BINARY OPERATOR OVERLOADING IN C++**

# BINARY OPERATOR

## INTRODUCTION

- In Binary operator overloading function, there should be one argument to be passed.
- It is overloading of an operator operating on two operands.

# BINARY OPERATOR OVERLOADING

```
#include<iostream>
class multiply
{
int first,second;
public:
void getdata(int a,int b)
{
first=a;
second=b;
}
```

Contd...,

```
void display()
{
cout<<“first=“<<first<<“second=“<<secon<<endl;
}
multiply operator *(multiply c);
};

void multiply::operator *(multiply c)
{
multiply temp;
temp.first=first*c.first;
temp.second=second*c.second;
return temp;
}
```

**Contd..,**

```
int main()
{
    multiply obj1,obj2,obj3;
    obj1.getdata(15,20);
    obj2.getdata(3,45);
    obj3=obj1*obj2;
    obj3.display();
    return 0;
}
```

### **Output:**

45  
900

# ACTIVITY DIAGRAM

# Activity Diagram

- ❑ Activity diagram is UML behavior diagram which emphasis on the sequence and conditions of the flow
- ❑ It shows a sequence of actions or flow of control in a system.
- ❑ It is like to a flowchart or a flow diagram.
- ❑ It is frequently used in business process modeling. They can also describe the steps in a use case diagram.
- ❑ The modeled Activities are either sequential or concurrent.

# Benefits

- It illustrates the logic of an algorithm.
- It describes the functions performed in use cases.
- Illustrate a business process or workflow between users and the system.
- It Simplifies and improves any process by descriptive complex use cases.
- Model software architecture elements, such as method, function, and operation.

# Symbols and Notations

## Activity

- Is used to illustrate a set of actions.
- It shows the non-interruptible action of objects.



# Symbols and Notations

## Action Flow

- It is also called edges and paths
- It shows switching from one action state to another. It is represented as an arrowed line.

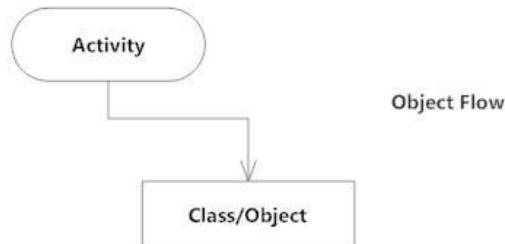


Action Flow

# Symbols and Notations

## Object Flow

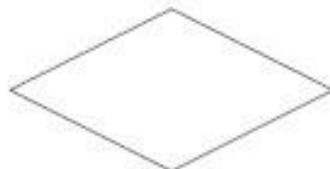
- Object flow denotes the making and modification of objects by activities.
- An object flow arrow from an action to an object means that the action creates or influences the object.
- An object flow arrow from an object to an action indicates that the action state uses the object.



# Symbols and Notations

## Decisions and Branching

- A diamond represents a decision with alternate paths.
- When an activity requires a decision prior to moving on to the next activity, add a diamond between the two activities.
- The outgoing alternates should be labeled with a condition or guard expression. You can also label one of the paths "else."



Decision Symbol

# Symbols and Notations

## Guards

- In UML, guards are a statement written next to a decision diamond that must be true before moving next to the next activity.
- These are not essential, but are useful when a specific answer, such as "Yes, three labels are printed," is needed before moving forward.



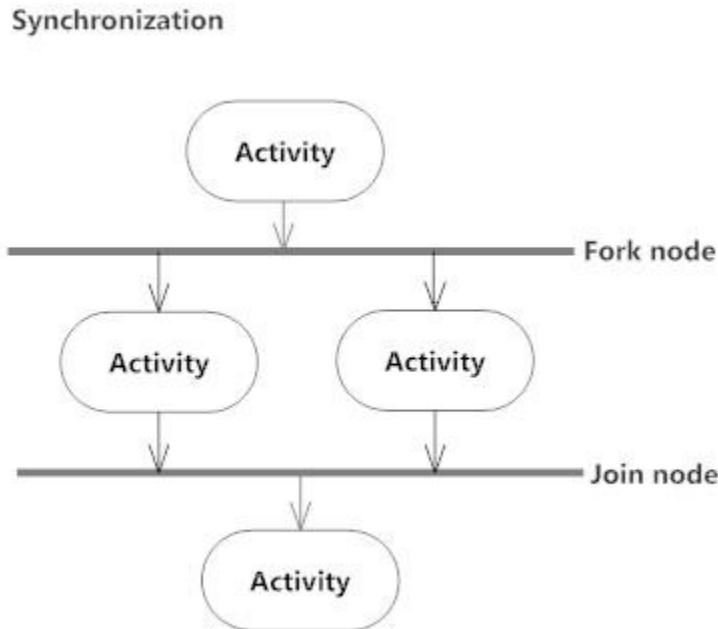
# Symbols and Notations

## Synchronization

- A fork node is used to split a single incoming flow into multiple concurrent flows. It is represented as a straight, slightly thicker line in an activity diagram.
- A join node joins multiple concurrent flows back into a single outgoing flow.
- A fork and join mode used together are often referred to as synchronization.

# Symbols and Notations

## Synchronization



# Symbols and Notations

## Time Event

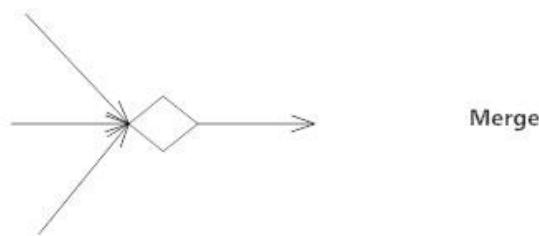
- This refers to an event that stops the flow for a time; an hourglass depicts it.



# Symbols and Notations

## Merge Event

- A merge event brings together multiple flows that are not concurrent.



## Final State or End Point

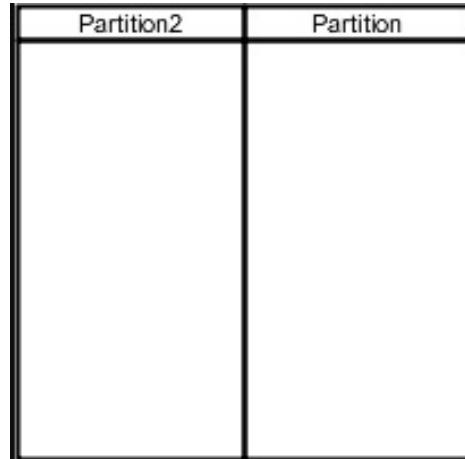
- An arrow pointing to a filled circle nested inside another circle represents the final action state.



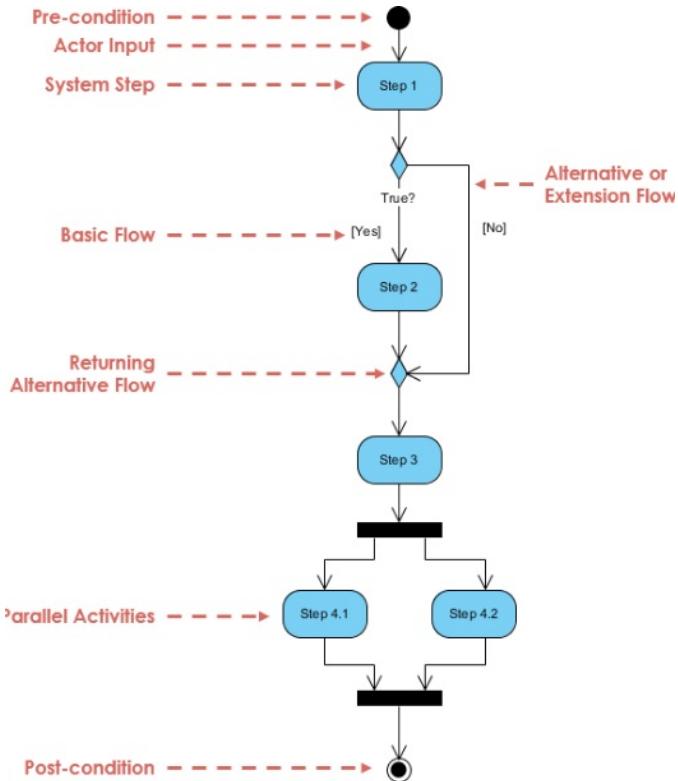
# Symbols and Notations

## Swimlane and Partition

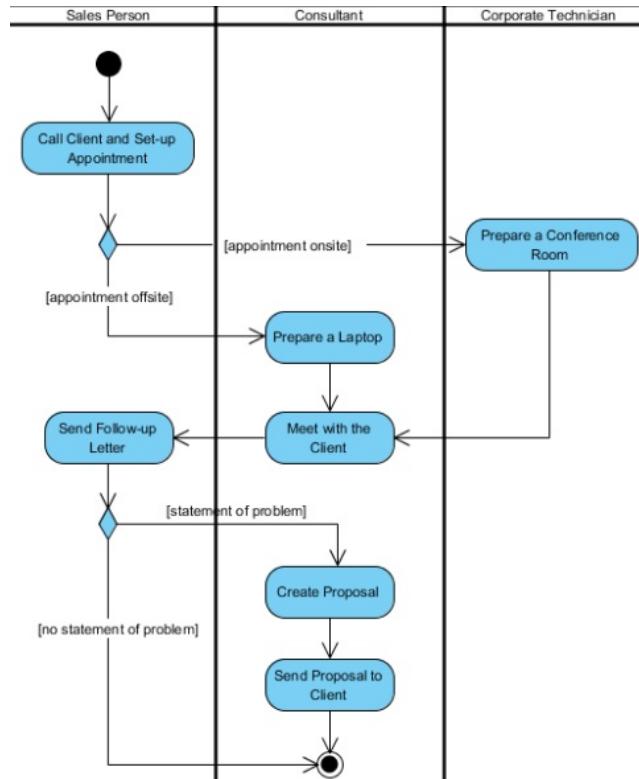
- A way to group activities performed by the same actor on an activity diagram or to group activities in a single thread



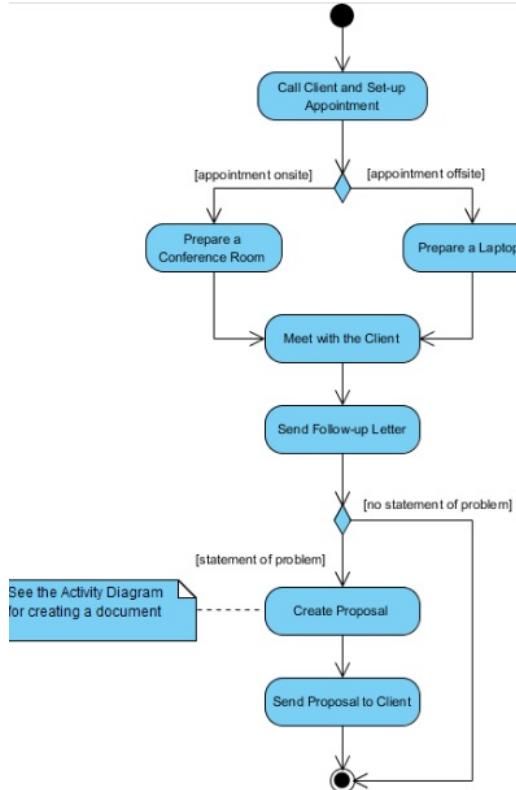
# Activity Diagram



# Activity Diagram with Swimlane



# Activity Diagram without Swimlane



# Friend Function

1. The main concepts of the object oriented programming paradigm are data hiding and data encapsulation.
2. Whenever data variables are declared in a private category of a class, these members are restricted from accessing by non – member functions.
3. The private data values can be neither read nor written by non – member functions.
4. If any attempt is made directly to access these members, the compiler will display an error message as “inaccessible data type”.
5. The best way to access a private data member by a non – member function is to change a private data member to a public group.
6. When the private or protected data member is changed to a public category, it violates the whole concept of data hiding and data encapsulation.
7. To solve this problem, a friend function can be declared to have access to these data members.
8. Friend is a special mechanism for letting non – member functions access private data.
9. The keyword friend inform the compiler that it is not a member function of the class.

# Friend Function

## Granting Friendship to another Class

1. A class can have friendship with another class.

2. For Example, let there be two classes, first and second. If the class first grants its friendship with the other class second, then the private data members of the class first are permitted to be accessed by the public members of the class second. But on the other hand, the public member functions of the class first cannot access the private members of the class second.

01

### Syntax

class second;forward declaration

class first

{

private:

-----

# Friend Function

## Two classes having the same Friend

1. A non – member function may have friendship with one or more classes.
2. When a function has declared to have friendship with more than one class, the friend classes should have forward declaration.
3. It implies that it needs to access the private members of both classes.

01

### Syntax

```
friend return_type  
function_name(parameters);
```

02

### Example

```
friend return_type fname(first one,  
second two)  
{}
```

03

### Note:

where friend is a keyword used as a function modifier. A friend declaration is valid only within or outside the class definition.

# Friend Function

Syntax:

```
class second;forward declaration  
class first  
{  
    private:  
    -----  
    public:  
        friend return_type fname(first one,  
second two);  
};  
class second  
{  
    private:  
    -----  
    public:  
        friend return_type fname(first one,  
second two);  
};
```

Case 2:

```
class sample  
{  
    private:  
        int x;  
        float y;  
    public:  
        virtual void display();  
        virtual static int sum(); //error  
    }  
    int sample::sum()  
    { }
```

# Friend Function Example

```
class sample
{
    private:
        int x;
    public:
        void getdata();
        friend void display(sample abc);
};

void sample::getdata()
{
    cout<<"Enter a value for x\n"<
```

Example:

# Friend Function Example

```
class first
{
    friend class second;
private:
    int x;
public:
    void getdata();
};

class second
{
public:
    void disp(first temp);
};

void first::getdata()
{
    cout<<"Enter a Number ?"<<endl;
    cin>>x;
}
```

Example:

# Friend Function Example

```
class second;//Forward Declaration
class first
{
    private:
        int x;
    public:
        void getdata();
        void display();
        friend int sum(first one,second two);
};

class second
{
    private:
        int y;
    public:
        void getdata();
        void display();
        friend int sum(first one,second two);
};

void first::getdata()
{
```

# Inline Member Function

**Inline functions are used in C++ to reduce the overhead of a normal function call.**

**A member function that is both declared and defined in the class member list is called an inline member function.**

**The inline specifier is a hint to the compiler that inline substitution of the function body is to be preferred to the usual function call implementation.**

**The advantages of using inline member functions are:**

- 1. The size of the object code is considerably reduced.**
- 2. It increases the execution speed, and**
- 3. The inline member function are compact function calls.**

# Inline Member Function

Syntax:

```
class user_defined_name  
{  
    private:  
        -----  
    public:  
        inline return_type function_name(parameters);  
        inline retrun_type function_name(parameters);  
        -----  
        -----  
};
```

Syntax

```
inline return_type function_name(parameters)  
{  
    -----  
    -----  
}
```

# Virtual function

- Virtual Function is a function in base class, which is overridden in the derived class, and which tells the compiler to perform Late Binding on this function.
- Virtual Keyword is used to make a member function of the base class Virtual. Virtual functions allow the most specific version of a member function in an inheritance hierarchy to be selected for execution. Virtual functions make polymorphism possible.

Key:

- Only the Base class Method's declaration needs the Virtual Keyword, not the definition.
- If a function is declared as virtual in the base class, it will be virtual in all its derived classes.

01

Syntax

```
virtual return_type function_name (arg);
```

02

Example

```
virtual void show()
{
    cout << "Base class\n";
}
```

03

Note:

We can call private function of derived class from the base class pointer with the help of virtual keyword. Compiler checks for access specifier only at compile time. So at run time when late binding occurs it does not check whether we are calling the private function or public function.

# Virtual function features

Case 1:

```
class sample
{
private:
    int x;
    float y;
public:
    virtual void display();
    virtual int sum();
}
virtual void sample::display() //Error
{ }
```

Case 2:

```
class sample
{
private:
    int x;
    float y;
public:
    virtual void display();
    virtual static int sum(); //error
}
int sample::sum()
{ }
```

# Virtual function features

Case 3:

```
class sample
{
    private:
        int x;
        float y;
    public:
        virtual sample(int x,float y);
//constructor
        void display();
        int sum();
}
```

Case 4:

```
class sample
{
    private:
        int x;
        float y;
    public:
        virtual ~sample(int x,float y); //invalid
        void display();
        int sum();
}
```

# Virtual function features

Case 5:

```
class sample_1
{
private:
    int x;
    float y;
public:
    virtual int sum(int x,float y); //error
};

class sample_2:public sample_1
{
private:
    int z;
public:
    virtual float sum(int xx,float yy);
};
```

Case 6:

```
virtual void display() //Error, non member
function
{  
-----  
-----  
}
```

# Virtual Function Example

```
class Point
{
protected:
    float length,breath,side,radius,area,height;
};

class Shape: public Point
{
public:
    virtual void getdata()=0;
    virtual void display()=0;
};

class Rectangle:public Shape
{
public:
    void getdata()
    {
        cout<<"Enter the Breadth Value:"<<endl;
        cin>>breath;
        cout<<"Enter the Length Value:"<<endl;
        cin>>length;
    }
    void display()
    {
        area = length * breath;
        cout<<"The Area of the Rectangle is:"<<area<<endl;
    }
};

class Square:public Shape
{
public:
```

## Example:

# Difference in invocation for virtual and non virtual function

```
class Base
{
public:
void show()
{
    cout << "Base class";
}
};

class Derived:public Base
{
public:
void show()
{
    cout << "Derived Class";
}
};

int main()
{
    Base* b; //Base class pointer
    Derived d; //Derived class object
    b = &d;
    b->show(); //Early Binding Occurs
}
```

Example:

# Difference in invocation for virtual and non virtual function

Example:

```
#include<iostream>
using namespace std;
class Base {
public:
    void foo()
    {
        std::cout << "Base::foo\n";
    }
    virtual void bar()
    {
        std::cout << "Base::bar\n";
    }
};

class Derived : public Base {
public:
    void foo()
    {
        std::cout << "Derived::foo\n";
    }
};
```

# Override

## Example:

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

class Base {
public:

    // user wants to override this in the derived class
    virtual void func()
    {
        cout << "I am in base" << endl;
    }
};

class derived : public Base {
public:

    // did a mistake by putting an argument "int a"
    void func(int a) override
```

# Pure Virtual function

- Pure virtual Functions are virtual functions with no definition. They start with virtual keyword and ends with = 0. Here is the syntax for a pure virtual function.
- Pure Virtual functions can be given a small definition in the Abstract class, which you want all the derived classes to have. Still you cannot create object of Abstract class.
- Also, the Pure Virtual function must be defined outside the class definition. If you will define it inside the class definition, complier will give an error. Inline pure virtual definition is Illegal.
- 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.

01

## Syntax

```
virtual void f() = 0;
```

02

## Example

```
class Base
```

```
{
```

```
public:
```

```
virtual void show() = 0; // Pure
```

```
Virtual Function
```

```
};
```

## Note:

We can call private function of derived class from the base class pointer with the help of virtual keyword. Compiler checks for access specifier only at compile time. So at run time when late binding occurs it does not check whether we are calling the private function or public function.

03

# Pure Virtual function

```
class pet
{
private:
    char name[5];
public:
virtual void getdata()=0;
virtual void display()=0;
};
class fish:public pet
{
private:
    char environment[10];
    char food[10];
public:
    void getdata();
    void display();
};
class dog: public pet
{
private:
```

Example:

# Pure Virtual function

```
void dog::getdata()
{
    cout<<"Enter the Dog Environment required"<<endl;
    cin>>environment;
    cout<<"Enter the Dog Food require"<<endl;
    cin>>food;
}

void dog::display()
{
    cout<<"Dog Environment="<<environment<<endl;
    cout<<"Dog Food="<<food<<endl;
    cout<<"-----"<<endl;
}

void main()
{
    pet *ptr;
    fish f;
    ptr=&f;
    ptr->getdata();
    ptr->display();
}
```

Example:

# Abstract Class

- Abstract class cannot be instantiated, but pointers and references of Abstract class type can be created.
- Abstract class can have normal functions and variables along with a pure virtual function.
- Abstract classes are mainly used for Upcasting, so that its derived classes can use its interface.
- Classes inheriting an Abstract Class must implement all pure virtual functions, or else they will become Abstract too.

# Pure virtual function

/Abstract base class

class Base

{

public:

virtual void show() = 0; // Pure Virtual Function

};

class Derived:public Base

{

public:

void show()

{

cout << "Implementation of Virtual Function in Derived class\n";

}

};

Example:

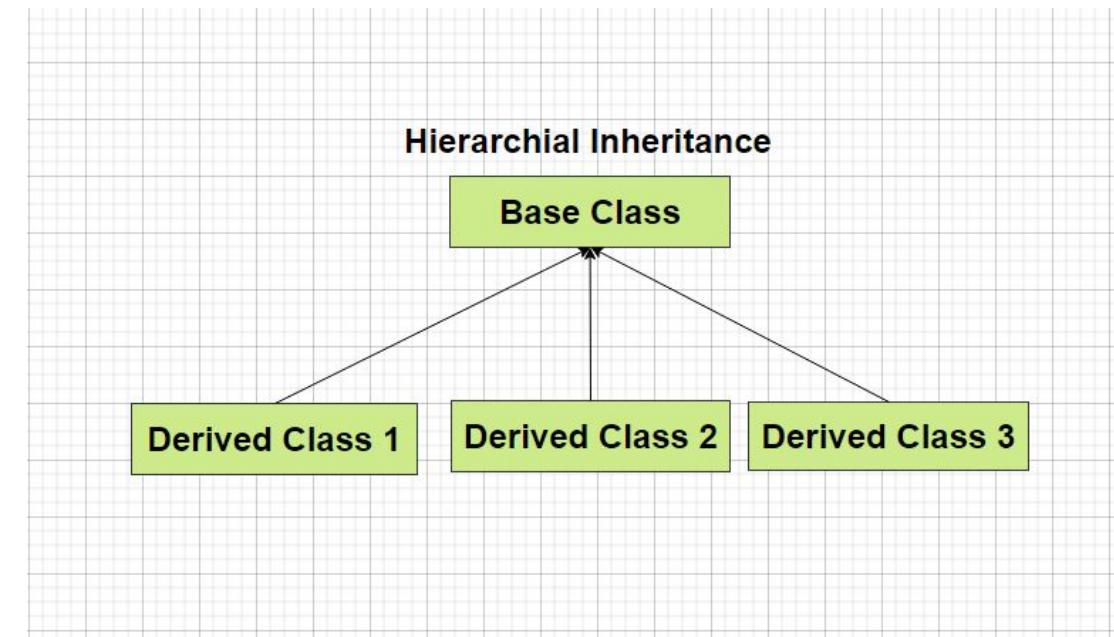
int main()

# Hierarichal Inheritance

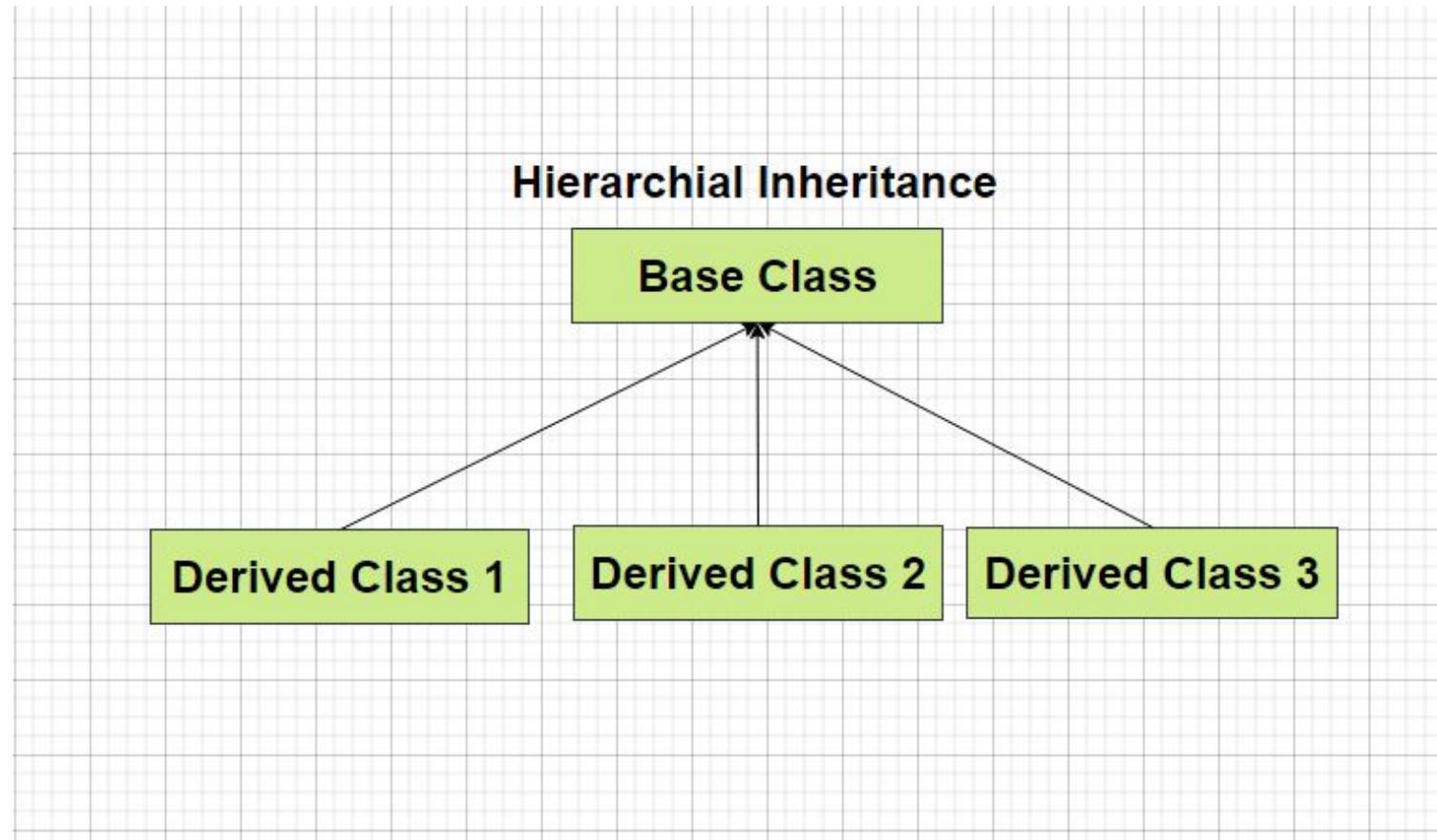


# Hierarichal Inheritance

- In Hierarichal Inheritance we have several classes that are derived from a common base class (or parent class).
- Here in the diagram Class 1, Class 2 and Class 3 are derived from a common parent class called Base Class.



# Diagrammatic Representation of Hierarchical Inheritance



# How to implement Hierarchical Inheritance in C++

```
class A {  
    // Body of Class A  
}; // Base Class
```

```
class B : access_specifier A  
{  
    // Body of Class B  
}; // Derived Class
```

```
class C : access_specifier A  
{  
    // Body of Class C  
}; // Derived Class
```

- In the example present in the left we have class A as a parent class and class B and class C that inherits some property of Class A.
- While performing inheritance it is necessary to specify the access\_specifier which can be public, private or protected.

# Access Specifiers

In C++ we have basically three types of access specifiers :

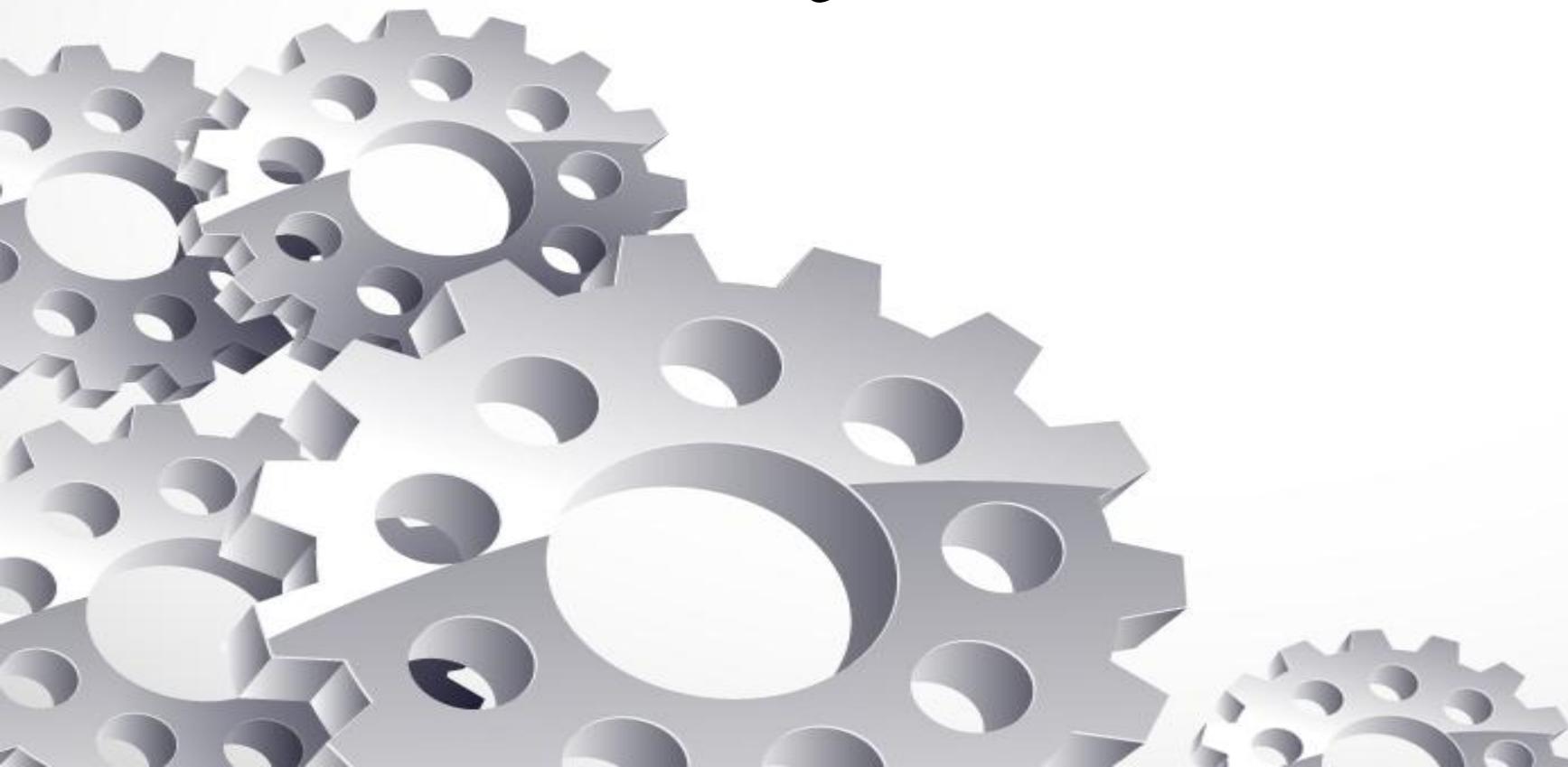
- Public : Here members of the class are accessible outside the class as well.
- Private : Here members of the class are not accessible outside the class.
- Protected : Here the members cannot be accessed outside the class, but can be accessed in inherited classes.

# Example of Inheritance

```
class Car {  
public:  
    int wheels = 4;  
    void show() {  
        cout << "No of wheels : "<<wheels ;  
    }  
};  
  
class Audi: public Vehicle {  
public:  
    string brand = "Audi";  
    string model = "A6";  
};
```

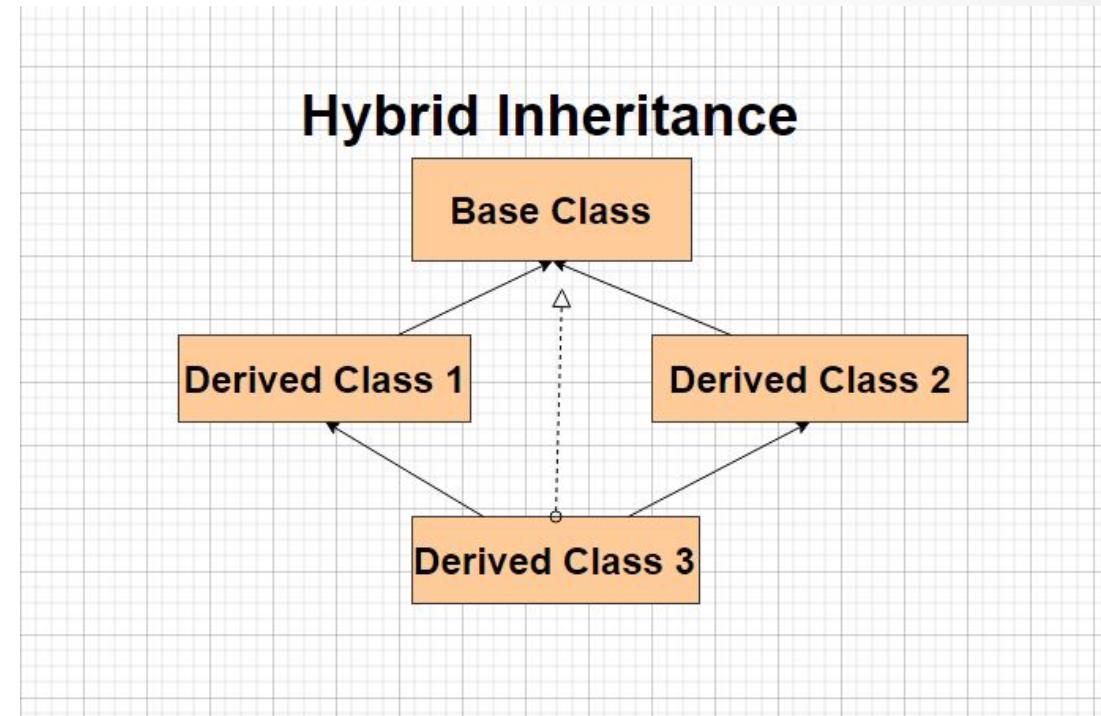
```
class Volkswagen: public Car {  
public:  
    string brand = "Volkswagen";  
    string model = "Beetle";  
};  
  
class Honda: public Car {  
public:  
    string brand = "Honda";  
    string model = "City";  
};
```

# Hybrid Inheritance

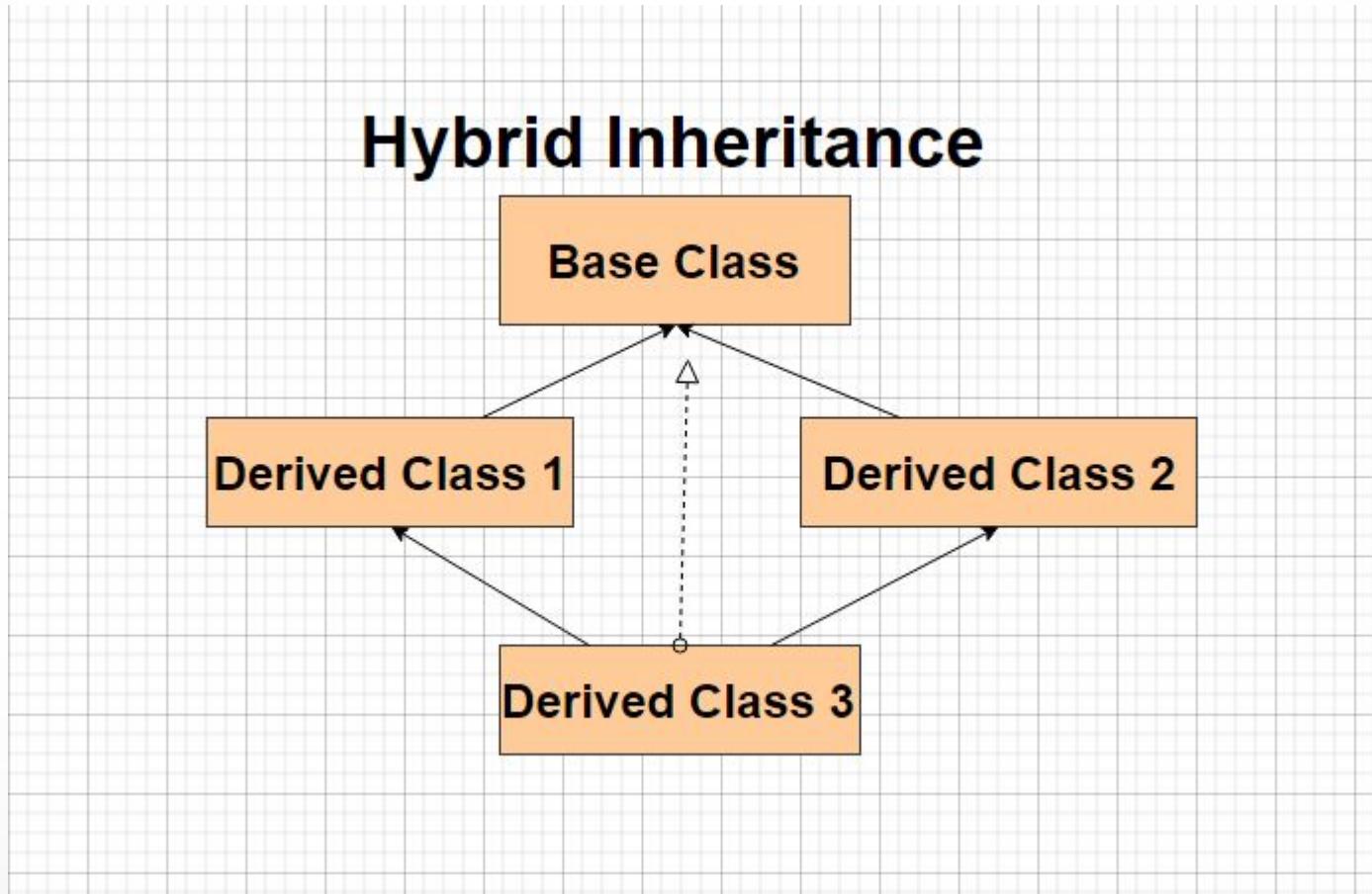


# Hybrid Inheritance

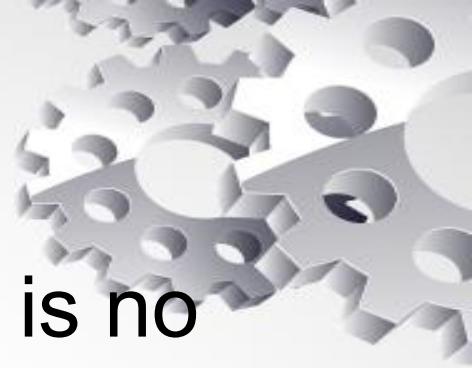
- Hybrid Inheritance involves derivation of more than one type of inheritance.
- Like in the given image we have a combination of hierarchical and multiple inheritance.
- Likewise we can have various combinations.



# Diagrammatic Representation of Hybrid Inheritance



# How to implement Hybrid Inheritance in C++



```
class A
{
    // Class A body
};

class B : public A
{
    // Class B body
};

class C
{
    // Class C body
};

class D : public B, public C
{
    // Class D body
};
```

- Hybrid Inheritance is no different than other type of inheritance.
- You have to specify the access specifier and the parent class in front of the derived class to implement hybrid inheritance.

# Access Specifiers



In C++ we have basically three types of access specifiers :

- Public : Here members of the class are accessible outside the class as well.
- Private : Here members of the class are not accessible outside the class.
- Protected : Here the members cannot be accessed outside the class, but can be accessed in inherited classes.

# Example of Hybrid Inheritance



```
class A
{
    public:
        int x;
};

class B : public A
{
    public:
        B()
        {
            x = 10;
        }
};
```

```
class C
{
    public:
        int y;
        C()
        {
            y = 4;
        }
};

class D : public B, public C
{
    public:
        void sum()
        {
            cout << "Sum= " << x + y;
        }
};
```

# State chart diagram

18CS202J OBJECT ORIENTED DESIGN AND PROGRAMMING

# **State diagram**

- A **state diagram** is used to represent the condition of the system or part of the system at finite instances of time. It's a **behavioural** diagram and it represents the behaviour using finite state transitions. State diagrams are also referred to as **State machines** and **State-chart Diagrams**.

# **Uses of state chart diagram**

- State chart diagrams are useful to model reactive systems
  - Reactive systems can be defined as a system that responds to external or internal events.
- State chart diagram describes the flow of control from one state to another state.

# Purpose

Following are the main purposes of using State chart diagrams:

- To model dynamic aspect of a system.
- To model life time of a reactive system.
- To describe different states of an object during its life time.
- Define a state machine to model states of an object.

# Difference between state diagram and flowchart

- The basic purpose of a **state diagram** is to portray various changes in state of the class and not the processes or commands causing the changes.
- However, a **flowchart** on the other hand portrays the processes or commands that on execution change the state of class or an object of the class.

# When to use State charts

- So the main usages can be described as:
  - To model object states of a system.
  - To model reactive system. Reactive system consists of reactive objects.
  - To identify events responsible for state changes.
  - Forward and reverse engineering.

# How to draw state charts

Before drawing a State chart diagram we must have clarified the following points:

- ✓ Identify important objects to be analysed.
- ✓ Identify the states.
- ✓ Identify the events.

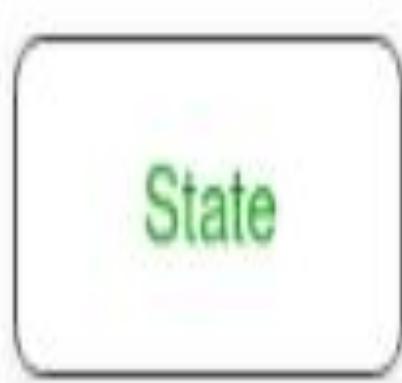
# Elements of state chart diagrams

- Initial State: This shows the starting point of the state chart diagram that is where the activity starts.



# Elements of state chart diagrams

- State: A state represents a condition of a modelled entity for which some action is performed. The state is indicated by using a rectangle with rounded corners and contains compartments



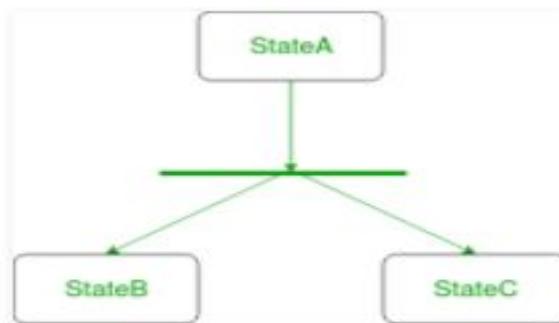
# Elements of state chart diagrams

- **Composite state** – We use a rounded rectangle to represent a composite state also. We represent a state with internal activities using a composite state.



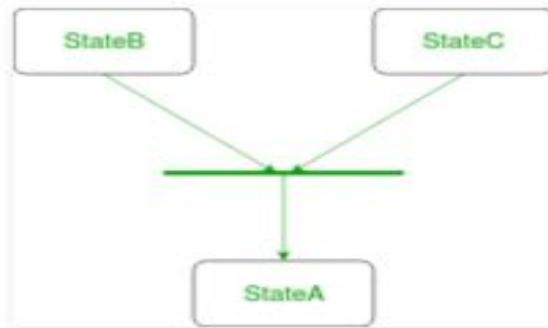
# Elements of state chart diagrams

- **Fork** – We use a rounded solid rectangular bar to represent a Fork notation with incoming arrow from the parent state and outgoing arrows towards the newly created states. We use the fork notation to represent a state splitting into two or more concurrent states.



# Elements of state chart diagrams

- **Join** – We use a rounded solid rectangular bar to represent a Join notation with incoming arrows from the joining states and outgoing arrow towards the common goal state. We use the join notation when two or more states concurrently converge into one on the occurrence of an event or events.



# Elements of state chart diagrams

- Transition: It is indicated by an arrow. Transition is a relationship between two states which indicates that Event/ Action an object in the first state will enter the second state and performs certain specified actions.



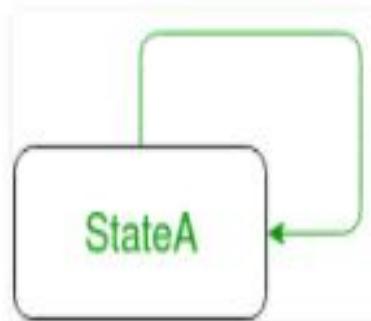
# Elements of state chart diagrams

- **Transition** – We use a solid arrow to represent the transition or change of control from one state to another. The arrow is labelled with the event which causes the change in state.



# Elements of state chart diagrams

- **Self transition** – We use a solid arrow pointing back to the state itself to represent a self transition. There might be scenarios when the state of the object does not change upon the occurrence of an event. We use self transitions to represent such cases.

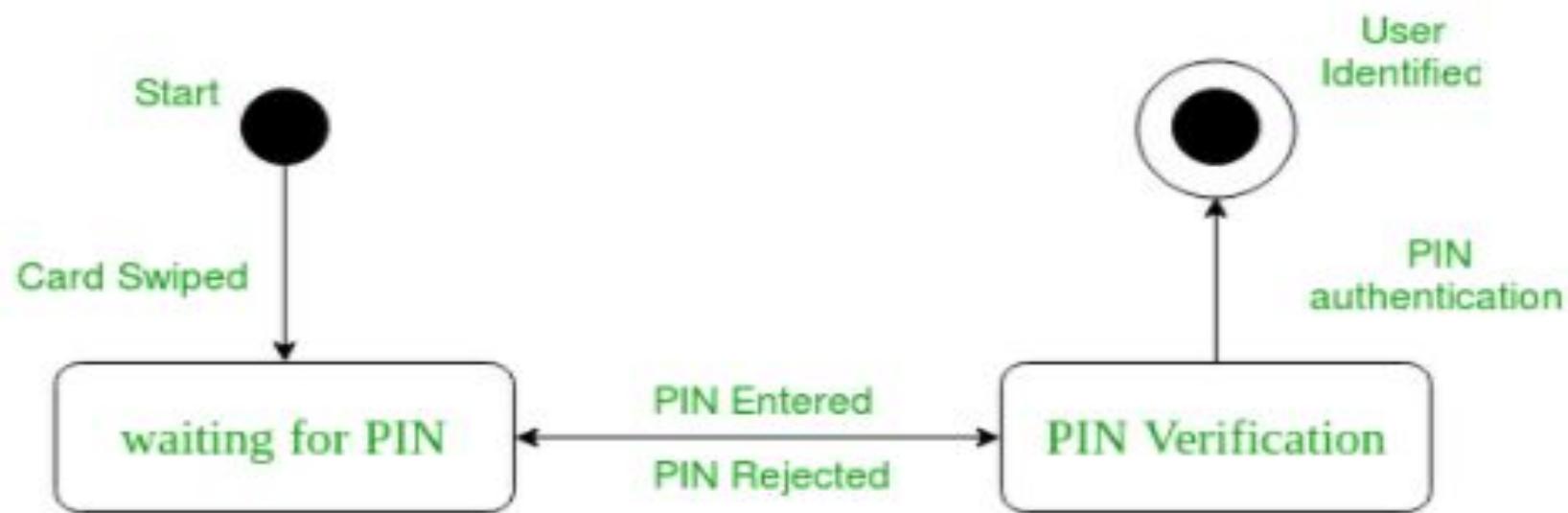


# Elements of state chart diagrams

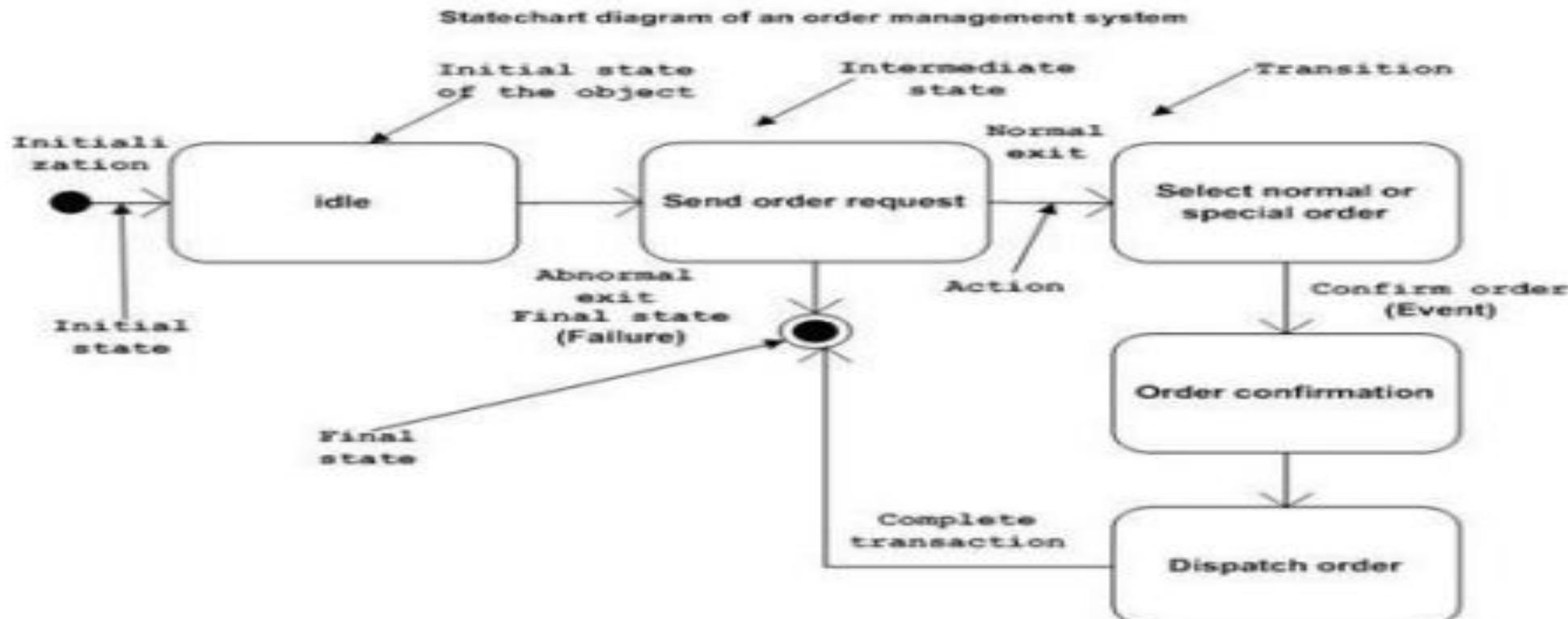
- Final State: The end of the state chart diagram is represented by a solid circle surrounded by a circle.



# Example state chart for ATM card PIN Verification



# Example state chart for order management system



# **18CSC202J - Syllabus**

## **Unit 3 :**

Single and Multiple Inheritance - Multilevel inheritance -  
Hierarchical - Hybrid Inheritance - Advanced Functions -  
Inline - Friend - Virtual -Overriding - Pure virtual function  
-Abstract class and Interface -UML State Chart Diagram -  
UML Activity Diagram

# Inheritance

01 Definition

Inheritance is the capability of one class to acquire properties and characteristics from another class. The class whose properties are inherited by other class is called the **Parent** or **Base** or **Super** class.

And, the class which inherits properties of other class is

```
class Subclass_name : access_mode Superclass_name
```

02 Syntax

Single Inheritance, Multiple Inheritance, Hierarchical Inheritance, Multilevel Inheritance, and Hybrid Inheritance (also known as Virtual Inheritance)

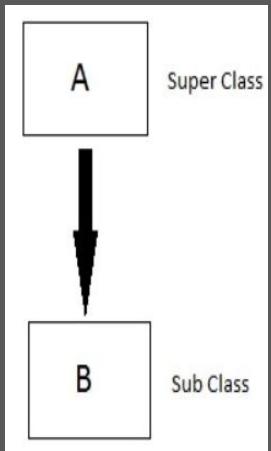
**Note :** All members of a class except Private, are inherited

1. Code Reusability
2. Method Overriding (Hence, Runtime Polymorphism.)
3. Use of Virtual Keyword

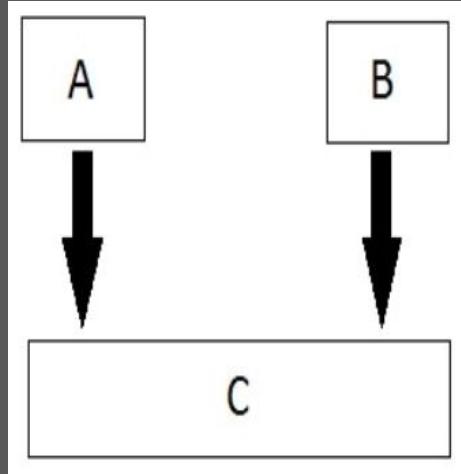
04 Advantages

# Inheritance Types

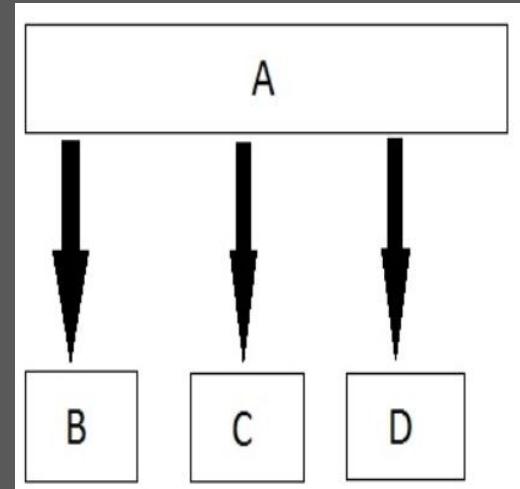
01 Single



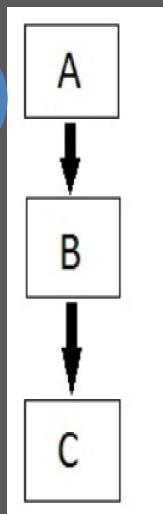
02 Multiple



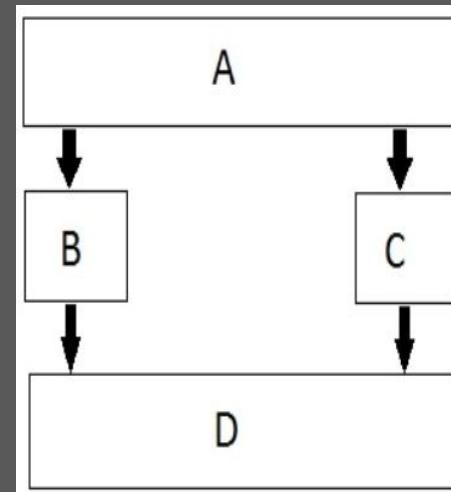
03 Hierarchical



04 Multilevel



05 Hybrid



# Modes of Inheritance

01 Public

If we derive a sub class from a public base class. Then the public member of the base class will become public in the derived class and protected members of the base class will become protected in derived class.

02 Protected

If we derive a sub class from a Protected base class. Then both public member and protected members of the base class will become protected in derived class.

03 private

If we derive a sub class from a Private base class. Then both public member and protected members of the base class will become Private in derived class.

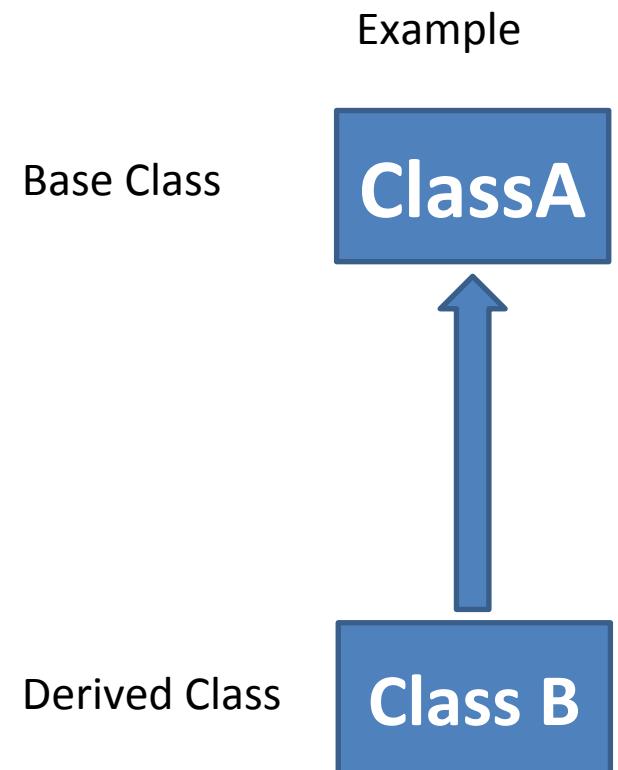
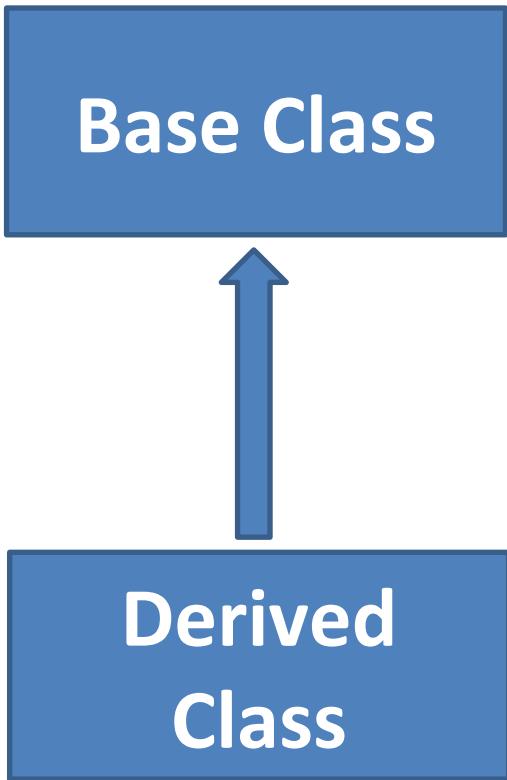
Note:

The private members in the base class cannot be directly accessed in the derived class, while protected members can be directly accessed. For example, Classes B, C and D all contain the variables x, y and z in below example

# Inheritance Access Matrix

Base class member access specifier	Type of Inheritance		
	Public	Protected	Private
Public	Public	Protected	Private
Protected	Protected	Protected	Private
Private	Not accessible (Hidden)	Not accessible (Hidden)	Not accessible (Hidden)

# SINGLE INHERITANCE



# Single Inheritance

In single inheritance, a class is allowed to inherit from only one class. i.e. one sub class is inherited by one base class only. Based on the visibility mode used or access specifier used while deriving, the properties of the base class are derived. Access specifier can be private, protected or public.

## Syntax:

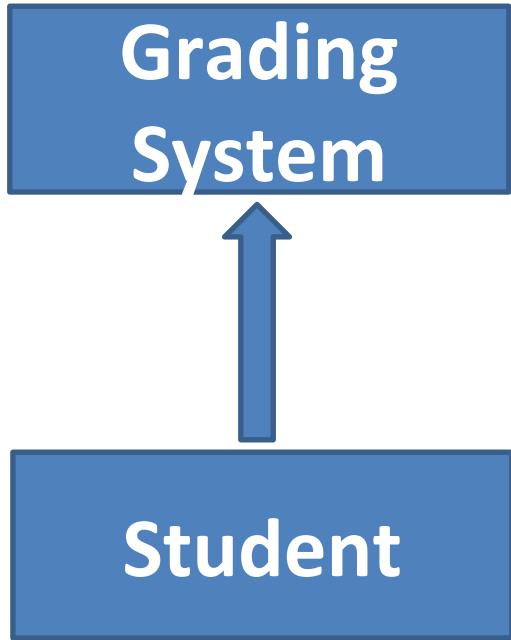
```
class Classname // base class
{
    .....
};

class classname: accessSpecifier baseclassname
{
    ...
};
```

# Example

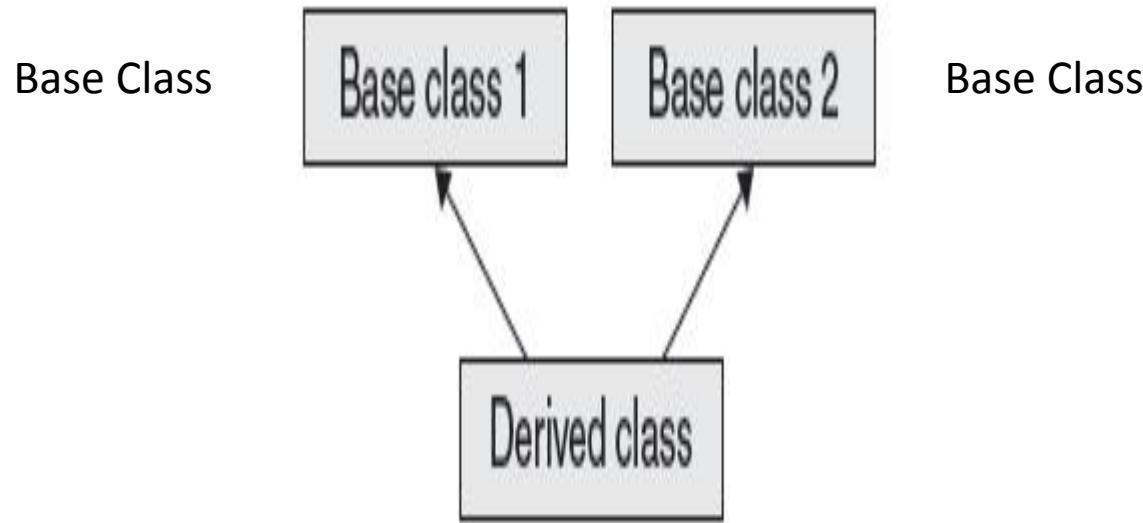
```
#include <iostream>
using namespace std;
class base //single base class
{ public:
    int x;
    void getdata()
    {
        cout << "Enter the value of x = ";
        cin >> x;
    }
};
class derived : public base //single derived class
{
    int y;
public:
    void readdata()
    {
        cout << "Enter the value of y = ";
        cin >> y;
    }
}
```

# Applications of Single Inheritance



1. University Grading System
2. Employee and Salary

# Multiple Inheritance



# Multiple Inheritance

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

## Syntax:

```
class A // base class
{
    .....
};

class B
{
    .....
}

class C : access_specifier A, access_specifier B // derived class
{
    .....
};
```

## Example:

```
#include using namespace std;
class sum1
{
    protected: int n1;
};

class sum2
{
    protected: int n2;
};

class show : public sum1, public sum2
{
public: int total()
{
    cout<<"enter n1";
    cin>>n1;

    cout<<"enter n2";
    cin>>n2;

    cout<<"enter n3";
    cin>>n3;
}

int sum()
{
    return n1+n2+n3;
}
};
```

# Applications of Multiple Inheritance

- Distributed Database

# Multilevel Inheritance

A derived class can be derived from another derived class. A child class can be the parent of another class.

## Syntax:

```
class A // base class
{
    .....
};

class B
{
    .....
}

class C : accessSpecifier B
// derived class
{
    .....
};

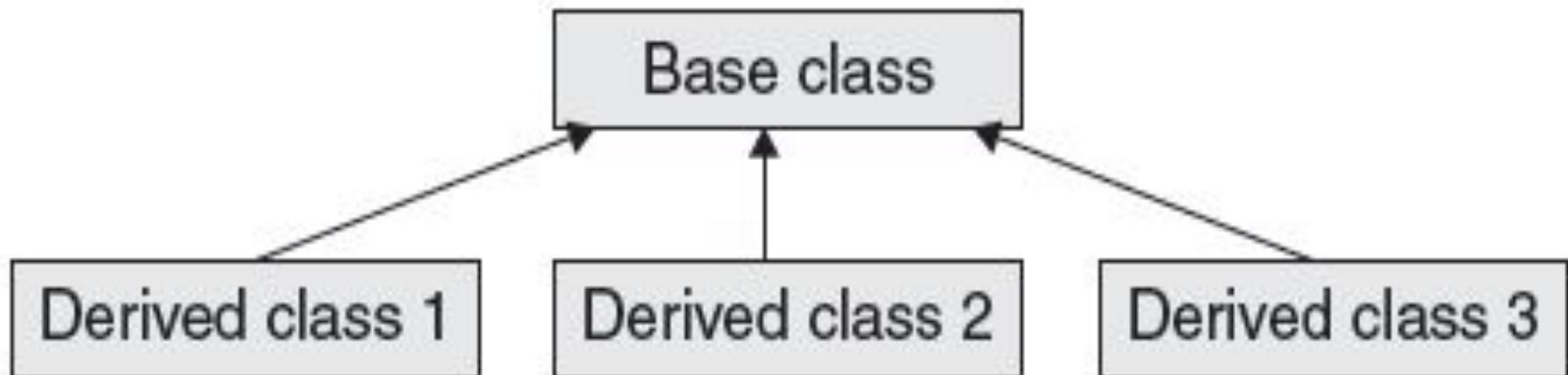
};
```

```
// base class
class Vehicle
{
public:
    Vehicle()
    {
        cout << "This is a Vehicle";
    }
};

class fourWheeler: public Vehicle
{ public:
    fourWheeler()
    {
        cout<<"Objects with 4 wheels are vehicles" << endl;
    }
};

// sub class derived from two base classes
class Car: public fourWheeler{
public:
    car()
{
```

# Hierarchical Inheritance



# Hierarchical Inheritance

In this type of inheritance, more than one sub class is inherited from a single base class. i.e. more than one derived class is created from a single base class.

**Syntax:**

```
class A // base class
{
};

class B : accessSpecifier A
{
};

class C : accessSpecifier A
{
};

class D : accessSpecifier A
{
};
```

```
#include <iostream>
using namespace std;
class A //single base class
{
public:
    int x, y;
    void getdata()
    {
        cout << "\nEnter value of x and y:\n";
        cin >> x >> y;
    }
};
class B : public A //B is derived from class base
{
public:
    void product()
    {
        cout << "\nProduct= " << x * y;
    }
};
```

## Example

```
class C : public A //C is also derived from
class base
{
public:
    void sum()
    {
        cout << "\nSum= " << x + y;
    }
};

int main()
{
    B obj1;      //object of derived class B
    C obj2;      //object of derived class C
    obj1.getdata();
    obj1.product();
    obj2.getdata();
    obj2.sum();
    return 0;
}
```

# Order of Constructor Call

Base class constructors are always called in the derived class constructors. Whenever you create derived class object, first the base class default constructor is executed and then the derived class's constructor finishes execution.

## Points to Remember

- Whether derived class's default constructor is called or parameterised is called, base class's default constructor is always called inside them.
- To call base class's parameterised constructor inside derived class's parameterised constructor, we must mention it explicitly while declaring derived class's parameterized constructor.

# Example

```
class Base
{
    int x;
    public:
Base()
{
cout<<"Base default constructor";
}
};

class Derived : public Base
{
    int y;
    public:
Derived()
{
cout<"Derived def. constructor";
}
```

# Order of Constructor Call

```
class Base
{
    int x;
public:
// parameterized constructor
Base(int i)
{
    x = i;
    cout<<"BaseParameterized Constructor\n";
}
};

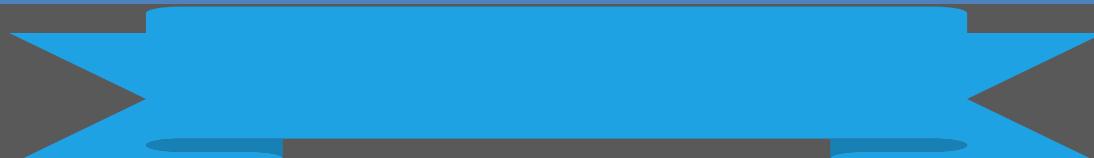
class Derived : public Base
{
    int y;
public:
// parameterized constructor
```

## Example

:

# Note:

Constructors have a special job of initializing the object properly. A Derived class constructor has access only to its own class members, but a Derived class object also have inherited property of Base class, and only base class constructor can properly initialize base class members. Hence all the constructors are called, else object wouldn't be constructed properly.

- 
- 01** Concept
- Constructors and Destructors are never inherited and hence never overridden.
  - Also, assignment operator = is never inherited. It can be overloaded but can't be inherited by sub class.
    - They are inherited into the derived class.
    - If you redefine a static member function in derived class, all the other overloaded functions in base class are hidden.
    - Static Member functions can never be virtual.

**02** Static Function

Derived class can inherit all base class methods except:

- 03** Limitation
- Constructors, destructors and copy constructors of the base class.
  - Overloaded operators of the base class.
  - The friend functions of the base class.

# Calling base and derived class method using base reference

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

## Example :

```
class Foo
{
public:
    int x;

    virtual void printStuff()
    {
        cout<<"BaseFoo";
    }
};

class Bar : public Foo
{
public:
    int y;

    void printStuff()
    {
        cout<<"DerivedBar";
    }
};
```

```
printStuff called"
```

# Calling base and derived class method using derived reference

## Example

:

```
#include <iostream>

class Base{
public:
    void foo()
    {
        std::cout<<"base";
    }
};

class Derived : public Base
{
public:
    void foo()
    {
        std::cout<<"derived";
    }
};
```

# 18CSC202J - OBJECT ORIENTED DESIGN AND PROGRAMMING

## Session 1

**Topic :Generic - Templates :  
Introduction**

# Templates-Introduction

- Allows functions and classes to operate with **generic types**.
- Allows a function or class to work on many **different data types without being rewritten** for each one.
- Great utility when combined with **multiple inheritance and operator overloading**
- The **C++ Standard Library** is based upon conventions introduced by the Standard Template Library (STL)

# Types of Templates

- **Function Template**

A *function template* behaves like a function except that the template can have arguments of many different types

- **Class Template**

A class template provides a specification for generating classes based on parameters.

Class templates are generally used to implement containers.

# Function Template

- A function templates work in similar manner as function but with one key difference.
- A single function template can work on different types at once but, different functions are needed to perform identical task on different data types.
- If you need to perform identical operations on two or more types of data then, you can use function overloading. But better approach would be to use function templates because you can perform this task by writing less code and code is easier to maintain.

# Cont.

- A generic function that represents several functions performing same task but on different data types is called function template.
- For example, a function to add two integer and float numbers requires two functions. One function accept integer types and the other accept float types as parameters even though the functionality is the same. Using a function template, a single function can be used to perform both additions.
- It avoids unnecessary repetition of code for doing same task on various data types.

# Cont.

## Why Function Templates?

- Templates are instantiated at compile-time with the source code.
- Templates are used less code than overloaded C++ functions.
- Templates are type safe.
- Templates allow user-defined specialization.
- Templates allow non-type parameters.

# Function Template

- A function template starts with keyword template followed by template parameter/s inside <> which is followed by function declaration.
- T is a template argument and class is a keyword.
- We can also use keyword **typename** instead of class.
- When, an argument is passed to some\_function( ), compiler generates new version of some\_function() to work on argument of that type.

# Template Syntax

```
template <class T>
T some_function(T argument)
{
.... .... ....
}
```

The template type keyword specified can be either "class" or "typename":

**template<class T>**

**or**

**template<typename T>**

Both are valid and behave exactly the same.

# 18CSC202J - OBJECT ORIENTED DESIGN AND PROGRAMMING

## Session 2

**Topic :Example Program Function  
Template, Class Template**

# Example program Function Templates

```
#include<iostream.h>
template <typename T>
T Sum(T n1, T n2)                                // Template function
{
    T rs;
    rs = n1 + n2;
    return rs;
}
int main()
{
    int A=10,B=20,C;
    long I=11,J=22,K;
    C = Sum(A,B);
    cout<<"nThe sum of integer values : "<<C;
    K = Sum(I,J);
    cout<<"nThe sum of long values : "<<K;
}
```

# More than One Template Argument

```
template<class T , class U>
void multiply(T a , U b)
{
    cout<<"Multiplication= "<<a*b<<endl;
}
int main()
{
    int a, b;
    float x, y;
    cin>>a>>b;
    cin>>x>>y;
    multiply(a,b);                                // Multiply two integer type data
    multiply(x,y);                                // Multiply two float type data
    multiply(a,x);                                // Multiply a float and integer type data
    return 0;
}
```

# Class Template

- Like function template, a class template is a common class that can represent various similar classes operating on data of different types.
- Once a class template is defined, we can create an object of that class using a specific basic or user-defined data types to replace the generic data types used during class definition.

# Syntax for Class Template

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

# Example Program

```
#include <iostream.h>
using namespace std;
const int MAX = 100; //size of array
template <class Type>
class Stack
{
private:
    Type st[MAX]; //stack: array of any type
    int top; //number of top of stack
public:
    Stack() //constructor
        { top = -1; }
    void push(Type var) //put number on stack
        { st[++top] = var; }
    Type pop() //take number off stack
        { return st[top--]; }
};
```

# Cont.

```
int main()
{
    Stack<float> s1;                                //s1 is object of class Stack<float>
    s1.push(1111.1F);                               //push 3 floats, pop 3 floats
    s1.push(2222.2F);
    s1.push(3333.3F);
    cout << "1: " << s1.pop() << endl;
    cout << "2: " << s1.pop() << endl;
    cout << "3: " << s1.pop() << endl;

    Stack<long> s2;                                //s2 is object of class Stack<long>
    s2.push(123123123L);                            //push 3 longs, pop 3 longs
    s2.push(234234234L);
    s2.push(345345345L);
    cout << "1: " << s2.pop() << endl;
    cout << "2: " << s2.pop() << endl;
    cout << "3: " << s2.pop() << endl;

    return 0;
}
```

# 18CSC202J - OBJECT ORIENTED DESIGN AND PROGRAMMING

## Session 3

**Topic :Class Template, Example  
Program for Class and Function  
Template**

# Class Templates with Multiple parameter

- We can use more than one generic data type in a class template.
- Syntax:

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

# Example Program

```
template<class T1, classT2>
class Test
{
    T1 a;
    T2 b;
}
void show()
{
cout<<a;
cout<<b;
}
};
```

```
int main()
{
    test<float, int> test1(1.23, 123);
    test<int, char> test2(100,'w');
    test1.show();
    test2.show();
    return 0;
}
```

Output:  
1.23  
123  
100  
w

# Class Template Object

To create a class template object, you need to define the data type inside a < > when creation.

## Syntax:

```
className<dataType> classObject;
```

## Example:

```
className<int> classObject;
```

```
className<float> classObject;
```

```
className<string> classObject;
```

# Program

1. Program to display largest among two numbers using function templates.
2. Program to swap data using function templates.
3. Program to add, subtract, multiply and divide two numbers using class template.

# Solution:1

```
#include <iostream>
using namespace std;
template <class T>
T Large(T n1, T n2)
{
    return (n1 > n2) ? n1 : n2;
}
int main()
{
    int i1, i2;
    float f1, f2;
    char c1, c2;
```

```
cout << "Enter two integers:\n";
cin >> i1 >> i2;
cout << Large(i1, i2) << " is larger." << endl;
cout << "\nEnter two floating-point numbers:\n";
cin >> f1 >> f2;
cout << Large(f1, f2) << " is larger." << endl;
cout << "\nEnter two characters:\n";
cin >> c1 >> c2;
cout << Large(c1, c2) << " has larger ASCII value.";
return 0;
}
```

# Output

Enter two integers:

5

10

10 is larger.

Enter two floating-point numbers:

12.4

10.2

12.4 is larger.

Enter two characters:

z

Z

z has larger ASCII value.

# Solution: 2

```
#include <iostream>
using namespace std;
template <typename T>
void Swap(T &n1, T &n2)
{
    T temp;
    temp = n1;
    n1 = n2;
    n2 = temp;
}
int main()
{
    int i1 = 1, i2 = 2;
    float f1 = 1.1, f2 = 2.2;
    char c1 = 'a', c2 = 'b';
    cout << "Before passing data to function template.\n";
    cout << "i1 = " << i1 << "\ni2 = " << i2;
    cout << "\nf1 = " << f1 << "\nf2 = " << f2;
    cout << "\nc1 = " << c1 << "\nc2 = " << c2;
    Swap(i1, i2);
    Swap(f1, f2);
    Swap(c1, c2);
    cout << "\nAfter passing data to function template.\n";
    cout << "i1 = " << i1 << "\ni2 = " << i2;
    cout << "\nf1 = " << f1 << "\nf2 = " << f2;
    cout << "\nc1 = " << c1 << "\nc2 = " << c2;
    return 0;
}
```

# Output

Before passing data to function template.

i1 = 1

i2 = 2

f1 = 1.1

f2 = 2.2

c1 = a

c2 = b

After passing data to function template.

i1 = 2

i2 = 1

f1 = 2.2

f2 = 1.1

c1 = b

c2 = a

# Solution: 3

```
#include <iostream>
using namespace std;
template <class T>
class Calculator
{
private: T num1, num2;
public: Calculator(T n1, T n2)
{
    num1 = n1;
    num2 = n2;
}
void displayResult()
{
    cout << "Numbers are: " << num1 << " and " << num2 << ".";
    cout << "Addition is: " << add() << endl;
    cout << "Subtraction is: " << subtract() << endl;
    cout << "Product is: " << multiply() << endl;
    cout << "Division is: " << divide() << endl;
}
T add()
{
    return num1 + num2;
}
```

```
T subtract()
{
    return num1 - num2;
}
T multiply()
{
    return num1 * num2;
}
T divide()
{
    return num1 / num2;
};
int main()
{
    Calculator<int> intCalc(2, 1);
    Calculator<float> floatCalc(2.4, 1.2);
    cout << "Int results:" << endl;
    intCalc.displayResult();
    cout << endl << "Float results:" << endl;
    floatCalc.displayResult();
    return 0;
}
```

# Output

Int results:

Numbers are: 2 and 1.

Addition is: 3

Subtraction is: 1

Product is: 2

Division is: 2

Float results:

Numbers are: 2.4 and 1.2.

Addition is: 3.6

Subtraction is: 1.2

Product is: 2.88

Division is: 2

# 18CSC202J - OBJECT ORIENTED DESIGN AND PROGRAMMING

## Session 6

**Topic :Exceptional Handling: try and  
catch, multilevel exceptional**

# Exceptions

- Indicate problems that occur during a program's execution
- Occur infrequently
- Exceptions provide a way to transfer control from one part of a program to another.
- 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.

# Exception handling

- Can resolve exceptions
  - Allow a program to continue executing or
  - Notify the user of the problem and
  - Terminate the program in a controlled manner
- Makes programs robust and fault-tolerant
- Types
  - Synchronous exception (out-of-range index, overflow)
  - Asynchronous exception (keyboard interrupts)

# Types of Exception

Two types of exception:

Synchronous Exceptions  
Asynchronous Exceptions

# Synchronous Exceptions

- Occur during the program execution due to some fault in the input data or technique that is not suitable to handle the current class of data, within the program .
- For example:
  - errors such as out of range
  - Overflow
  - underflow and so on

# Asynchronous Exceptions

- Caused by events or faults unrelated (external) to the program and beyond the control of the program.
- For example
  - errors such as keyboard interrupts
  - hardware malfunctions
  - disk failure and so on

The exception handling mechanism of C++ is designed to **handle only synchronous exceptions** within a program.

# Exception levels

Exceptions can occur at many levels:

1. Hardware/operating system level.
  - Arithmetic exceptions; divide by 0.
  - Memory access violations; stack over/underflow.
2. Language level.
  - Type conversion; illegal values, improper casts.
  - Bounds violations; illegal array indices.
  - Bad references; null pointers.
3. Program level.
  - User defined exceptions.

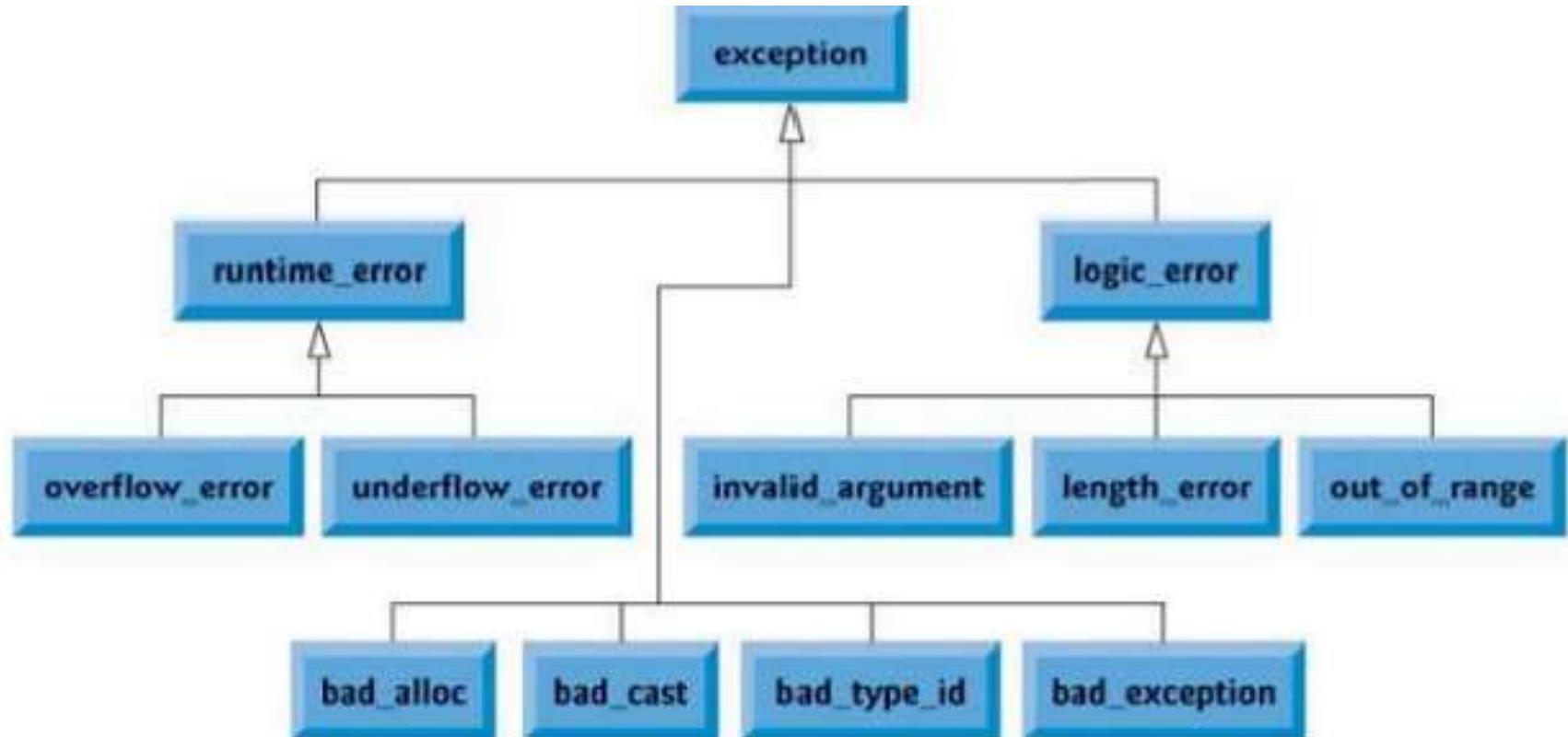
# Need of Exceptions

- Detect and report an “exceptional circumstance”
- Separation of error handling code from normal code
- Functions/ Methods can handle any exception they choose
- Grouping of Error types

# Mechanism

1. Find the problem (*Hit* the exception)
2. Inform that an error has occurred (*Throw* the exception)
3. Receive the error information (*Catch* the exception)
4. Take corrective actions (*Handle* the exception)

# C++ Standard Exceptions



# C++ Standard Exceptions

Exception	Description
<b>std::exception</b>	An exception and parent class of all the standard C++ exceptions.
<b>std::bad_alloc</b>	This can be thrown by <b>new</b> .
<b>std::bad_cast</b>	This can be thrown by <b>dynamic_cast</b> .
<b>std::bad_exception</b>	This is useful device to handle unexpected exceptions in a C++ program
<b>std::bad_typeid</b>	This can be thrown by <b>typeid</b> .

# C++ Standard Exceptions

Exception	Description
<b>std::logic_error</b>	An exception that theoretically can be detected by reading the code.
<b>std::domain_error</b>	This is an exception thrown when a mathematically invalid domain is used
<b>std::invalid_argument</b>	This is thrown due to invalid arguments.
<b>std::length_error</b>	This is thrown when a too big std::string is created
<b>std::out_of_range</b>	This can be thrown by the at method from for example a std::vector and std::bitset<>::operator[]().

# C++ Standard Exceptions

Exception	Description
std::runtime_error	An exception that theoretically can not be detected by reading the code.
std::overflow_error	This is thrown if a mathematical overflow occurs.
std::range_error	This occurred when you try to store a value which is out of range.
std::underflow_error	This is thrown if a mathematical underflow occurs.

# Exceptions: keywords

- Handling the Exception is nothing but converting system error message into user friendly error message
- Exception handling use three keywords for handling the exception

- try
- catch
- throw

# Simple Exceptions : syntax

```
.....  
.....  
try  
{  
    .....  
    throw exception;  
    .....  
    .....  
}  
catch(type arg)  
{  
    .....  
    .....  
}  
.....  
.....
```

# Exceptions

```
void Func3()
{
    try
    {
        Func4();
    }
    catch ( ErrType )
    {
    }
}
```

Function call

Normal return

Return from  
thrown  
exception

```
void Func4()
{
    if ( error )
        throw ErrType();
}
```

# Simple Exceptions : Example

```
#include<iostream>
using namespace std;
int main()
{
    int a,b;
    cin >> a >> b;
    try
    {
        if (b!=0)
        {
            cout<<"result (a/b) ="<<a/b;
        }
    }
    else
    {
        throw(b);
    }
}
catch(int i)
{
    cout <<"exception caught";
}
```

# Nested try blocks

```
try
{
.....
try
{
.....
}
catch (type arg)
{
.....
}
}
catch(type arg)
{
.....
.....
}
}
```

# Multiple Catch Exception

- Used when a user wants to handle different exceptions differently.
- For this, a user must include catch statements with different declaration.

# Multiple Catch Exception

- It is possible to design a separate catch block for each kind of exception
- Single catch statement that catches all kind of exceptions
- Syntax

```
catch(...)
{
    .....
}
```

Note :

A better way to use this as a default statement along with other catch statement so that it can catch all those exception which are not handle by other catch statement

# Multiple catch statement : Syntax

```
try
{
.....
}
catch (type1 arg)
{
.....
}
catch (type2 arg)
{
.....
}
.....
}
.....
catch(typeN arg)
{
.....
}
```

# Multiple Exceptions : Example

```
#include<iostream>
using namespace std;
int main()
{
    int a,b;
    cin >> a >> b;
    try
    {
        if (b!=a)
        {
            float div = (float) a/b;
            if(div <0)
                throw 'e';
            cout<<div;
        }
        else
            throw b;
    }
    catch(int i)
    {cout <<“exception caught”;
    }
```

```
catch(int i)
{
    cout <<“exception caught : Division by zero”;
}
catch (char st)
{
    cout << “exception caught : Division is less
than 1”;
}
catch(...)
{
    cout << “Exception : unknown”;
}
```

# 18CSC202J - OBJECT ORIENTED DESIGN AND PROGRAMMING

## Session 7

**Topic :throw, throws and finally**

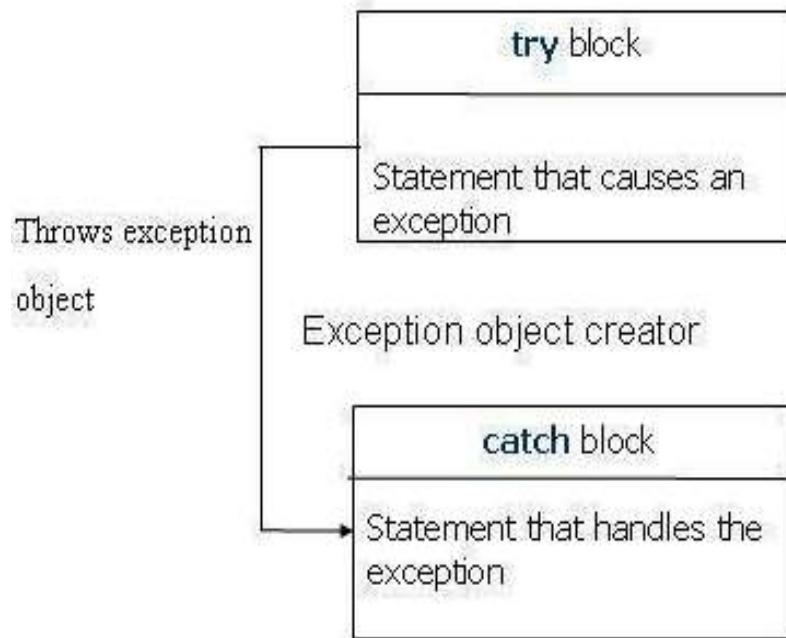
## throwing Exception

- When an exception is detected, it is thrown using throw statement in the try block
- It is also possible, where we have nested try-catch statement

throw;

- It cause the current exception to be thrown to the next enclosing try/catch sequence and is caught by a catch statement listed after that enclosing try block.

# throwing Exception





# Rethrowing Exception

```
try
{
    -----
    try
    {
        -----
        throw val;
        -----
    }
    catch(data-type arg)
    {
        -----
        throw;
        -----
    }
    -----
}
catch(data-type arg)
{
    -----
}
```

throws exception value

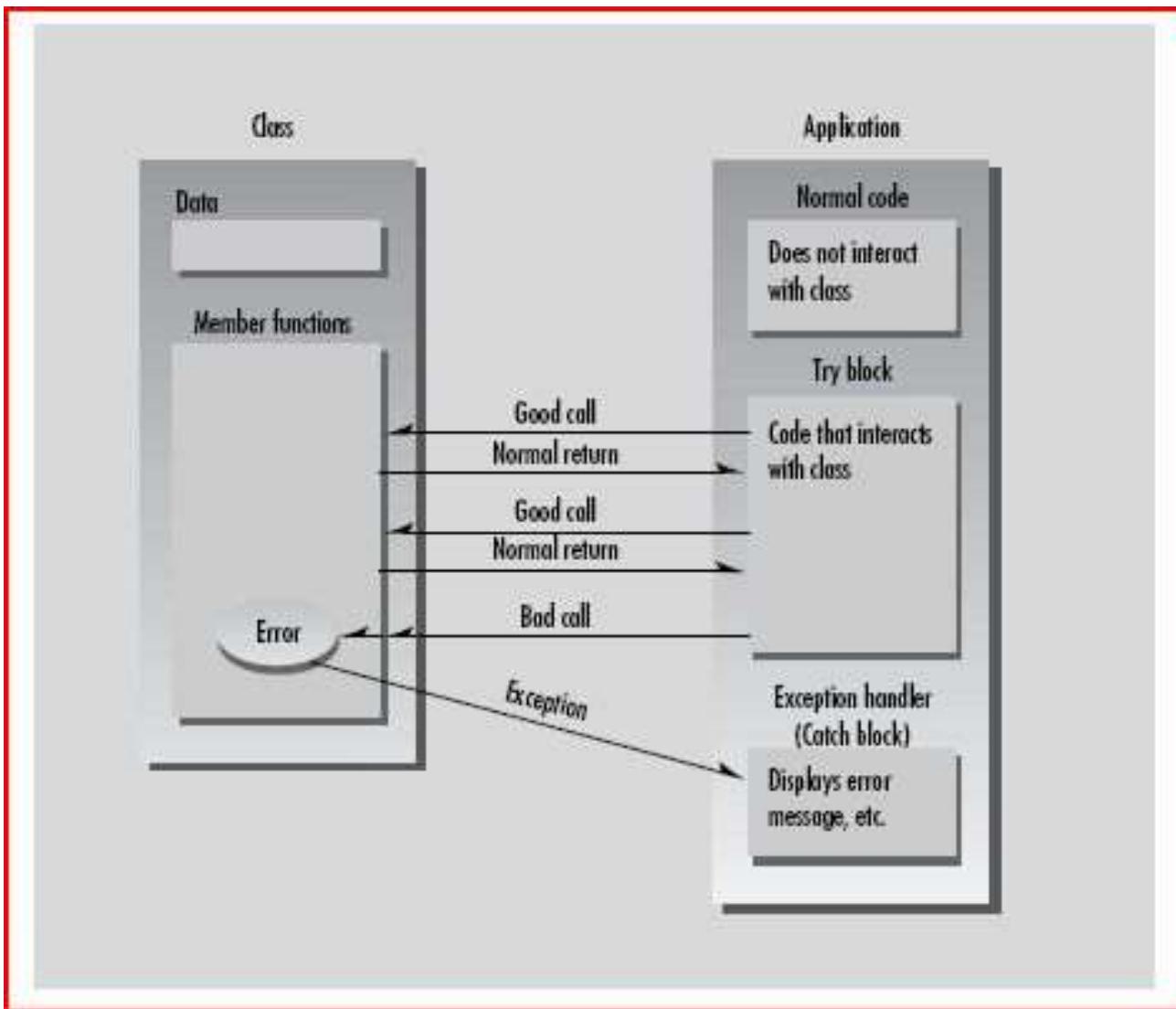
Rethrows exception value

# The *throw* point

- Used to indicate that an exception has occurred
- It is called “throwing an exception”
- Throw normally specifies an operand:
- Will be caught by closest exception handler

# The throw point: Example

```
try
{
    if(denominator == 0)
    {
        throw denominator;
    }
    result = numerator/denominator;
    cout<<"\nThe result of division is:" <<result;
}
```



# Finally

- The application always executes any statements in the **finally** part, even if an exception occurs in the **try** block. When any code in the **try** block raises an exception, execution halts at that point.
- Once an exception handler is found, execution jumps to the **finally** part. After the **finally** part executes, the exception handler is called.
- If no exception occurs, the code in the **finally** block executes in the normal order, after all the statements in the **try** block.

# Syntax

```
try
{
    // statements that may raise an exception
}
finally
{
    // statements that are called even
    // if there is an exception in the try block
}
```

# Example

```
#include<iostream>
using namespace std;
int main()
{
    int a,b;
    cin >> a>> b;
    try
    {
        if (b!=0)
        {
            cout<<"result (a/b)="\b<<a/b;
        }
    }
    else
    {
        throw(b);
    }
}
catch(int i)
{
    cout <<"exception caught";
}
finally
{
    Cout<<"Division";
}
```

# 18CSC202J - OBJECT ORIENTED DESIGN AND PROGRAMMING

## Session 8

**Topic : Exceptional Handling:  
predefined exception**

# User Defined Exception

We can define your own exceptions by inheriting and overriding exception class functionality. Following is the example, which shows how you can use exception class to implement your own exception in standard way:

# Define New Exception

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

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

Here, what() is a public method provided by exception class and it has been overridden by all the child exception classes. This returns the cause of an exception.

# Practices Program

1. C++ program to divide two numbers using try catch block.
2. Simple C++ Program for Basic Exception Handling.
3. Simple Program for Exception Handling Divide by zero Using C++ Programming
4. Simple Program for Exception Handling with Multiple Catch Using C++ Programming
5. Simple C++ Program for Catch All or Default Exception Handling
6. Simple C++ Program for Rethrowing Exception Handling in Function
7. Simple C++ Program for Nested Exception Handling

# Program

```
#include <iostream>
#include <conio.h>
using namespace std;
int main() {
    int a,b;
    cout << "Enter 2 numbers:
";
    cin >> a >> b;
    try {
        if (b != 0)
        {
            float div = (float)a/b;
```

```
        if (div < 0)
            throw 'e';
        cout << "a/b = " << div;
    }
    else throw b;
} catch (int e)
{ cout << "Exception: Division by zero"; }
catch (char st)
{ cout << "Exception: Division is less than
1"; }
catch(...) { cout << "Exception: Unknown";
}
getch();
return 0;
}
```

# 18CSC202J - OBJECT ORIENTED DESIGN AND PROGRAMMING

## Session 11

**Topic : Dynamic Modelling: Package  
Diagram, UML Component Diagram**

# Dynamic Modelling

The dynamic model is used to express and model the behaviour of the system over time. It includes support for activity diagrams, state diagrams, sequence diagrams and extensions including business process modelling.

# Package Diagram

- All the interrelated classes and interfaces of the system when grouped together form a package.
- To represent all these interrelated classes and interface UML provides package diagram.
- Package diagram helps in representing the various packages of a software system and the dependencies between them.
- It also gives a high-level impression of use case and class diagram.

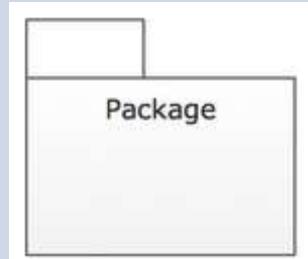
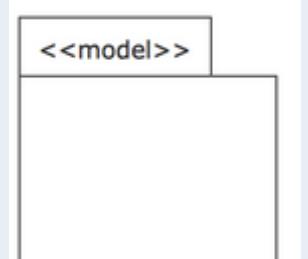
# Package Diagram: purpose

- To provide static models of modules, their parts and their relationships
- To present the architectural modelling of the system
- To group any UML elements
- To specify the logical distribution of classes
- To emphasize the logical structure of the system
- To offer the logical distribution of classes which is inferred from the logical architecture of the system

# Package Diagram: Uses

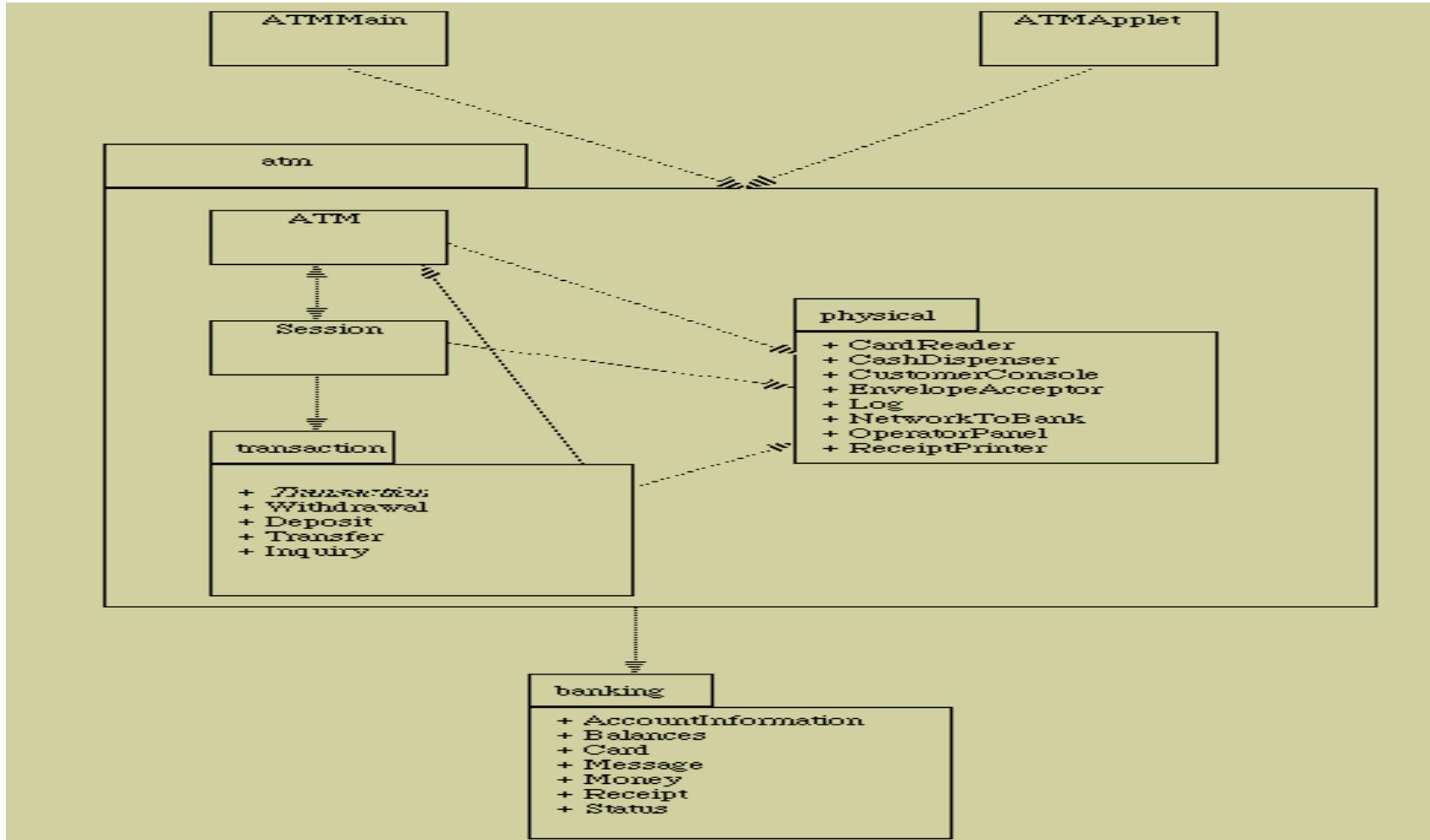
- To illustrate the functionality of a software system.
- To illustrate the layered architecture of a software system.
- The dependencies between these packages can be adorned with labels / stereotypes to indicate the communication mechanism between the layers.

# Notations

S.NO	NAME	SYMBOL	DESCRIPTION
1	Package		organize elements into groups to provide better structure for system model.
2	Mode		show only a subset of the contained elements according to some criterion.



# Example



# Component Diagram

- A component diagram shows the physical view of the system
- A component is an autonomous unit within a system.
- We combine packages or individual entities to form components.
- We can depict various components and their dependencies using a component diagram.
- Component diagram contain: component package, components, interfaces and dependency relationship.



# Component Diagram: Purpose

- It shows the structural relationship between the components of a system.
- It identifies the architectural perspective of the system as they enable the designer to model the high level software components with their interfaces to other components.
- It helps to organize source code into manageable chunks called components.
- It helps to specify a physical database.
- It can be easily developed by architects and programmers.
- It enables to model the high level software components and the interfaces to those components.
- The components and subsystem can be flexibly reused and replaced.

# Guidelines to Draw

- Based on the analysis of the problem description of the system, identify the major subsystem.
- Group the individual packages and other logical entities in the system to provide as separate components.
- Then identify the interfaces needed for components interaction.
- If needed, identify the subprograms which are part of each of the components and draw them along with their associated components.
- Use appropriate notations to draw the complete component diagram.

# 18CSC202J - OBJECT ORIENTED DESIGN AND PROGRAMMING

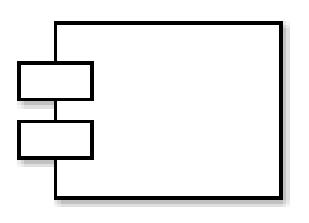
## Session 12

**Topic : UML Component Diagram,  
Deployment Diagram**

# Guidelines to Draw: Component Diagram

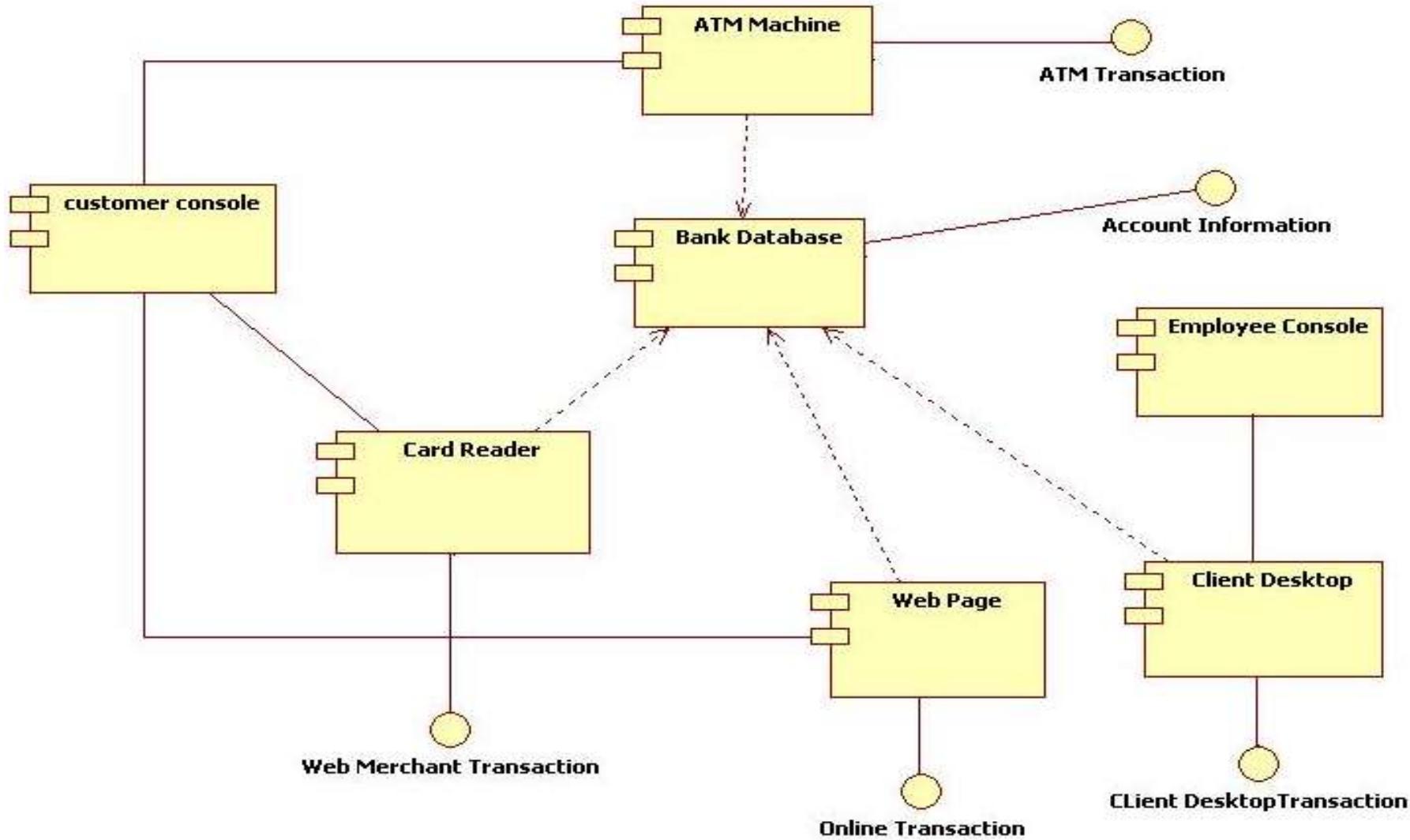
- Based on the analysis of the problem description of the system, identify the major subsystem.
- Group the individual packages and other logical entities in the system to provide as separate components.
- Then identify the interfaces needed for components interaction.
- If needed, identify the subprograms which are part of each of the components and draw them along with their associated components.
- Use appropriate notations to draw the complete component diagram.

# Notations

S.NO	NAME	SYMBOL	DESCRIPTION
1	Component		Component is used to represent any part of a system for which UML diagrams are made.
2	Association		A structural relationship describing a set of links connected between objects.



# Example



# Deployment Diagram

- A deployment diagram shows the physical placement of components in nodes over a network.
- A deployment diagram can be drawn by identifying nodes and components.
- A deployment diagram usually describes the resources required for processing and the installation of software components in those resources.

# Purpose

- It shows the relationship between software and hardware components in the target system.
- They are useful to show the system design that has subsystem, concurrent execution, compile time and execution time invocations, and hardware/software mapping by assigning the appropriate software components to the hardware devices.
- As they specify the distribution of software components in various devices and processors in the target environment, it will be easier for maintenance activities.
- Using this diagram it is easier to identify performance bottlenecks.

# 18CSC202J - OBJECT ORIENTED DESIGN AND PROGRAMMING

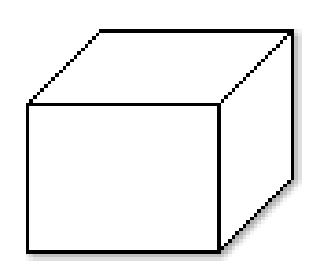
## Session 13

**Topic : UML Deployment Diagram,  
Examples**

# Guidelines to Draw: Deployment Diagram

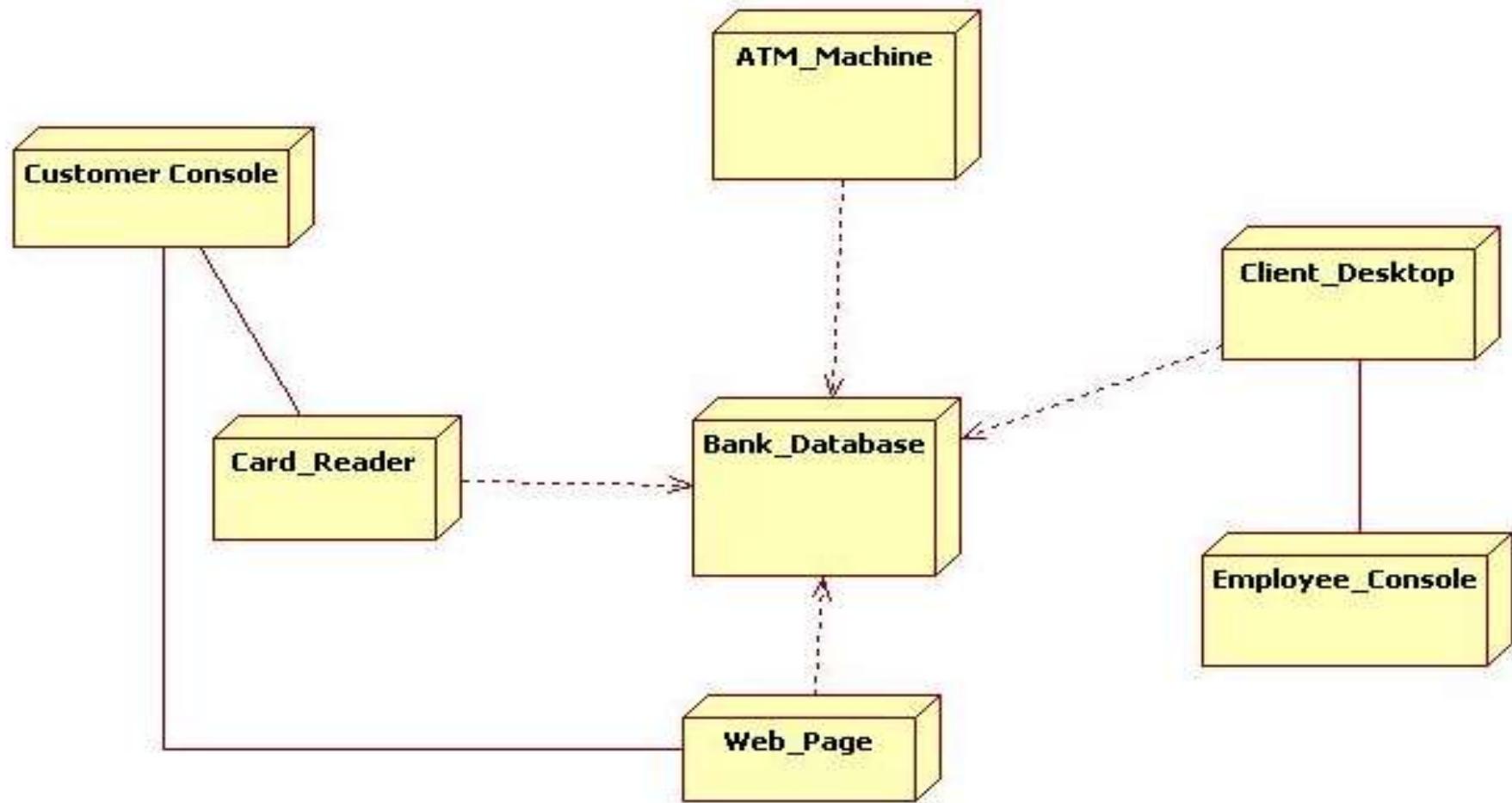
- Identify the hardware components and processing units in the target system.
- Analyze the software and find out the subsystem, parallel execution of modules, server side components, client side components, business logic components, backend database servers and software and hardware mapping mechanism to map the software components to be mapped with appropriate hardware devices.
- Draw the hardware components and show the software components inside them and also show the connectivity between them.

# Notations

S.NO	NAME	SYMBOL	DESCRIPTION
1	Node		A node represents a physical component of the system. Node is used to represent physical part of a system like server, network etc.
2	Association		A structural relationship describing a set of links connected between objects.



# Example



# Example

1. online shopping UML diagrams
2. Ticket vending machine UML diagrams
3. Bank ATM UML diagrams
4. Hospital management UML diagrams
5. Digital imaging and communications in medicine (DICOM) UML diagrams
6. Java technology UML diagrams
7. Application development for Android UML diagrams

# **18CSC202J - OBJECT ORIENTED DESIGN AND PROGRAMMING**

## **Session 1**

**Topic : Sequence Container:Vector List**

# What is stl???

- The Standard Template Library (STL) is a set of C++ template classes to provide common programming data structures and functions such as lists, stacks, arrays, etc.
- It is a library of container classes, algorithms, and iterators.
- It is a generalized library and so, its components are parameterized.
- A working knowledge of template classes is a prerequisite for working with STL.

**STL has four  
components**

Containers

Iterators

Algorithms

Functions

# STL COntainers

Sequence and Associative Containers

# Containers

- Containers or container classes store objects and data.
- There are in total seven standard “first-class” container classes and three container adaptor classes and only seven header files that provide access to these containers or container adaptors.
  - Sequence Containers
  - Container Adaptors
  - Associative Containers
  - Unordered Associative Containers

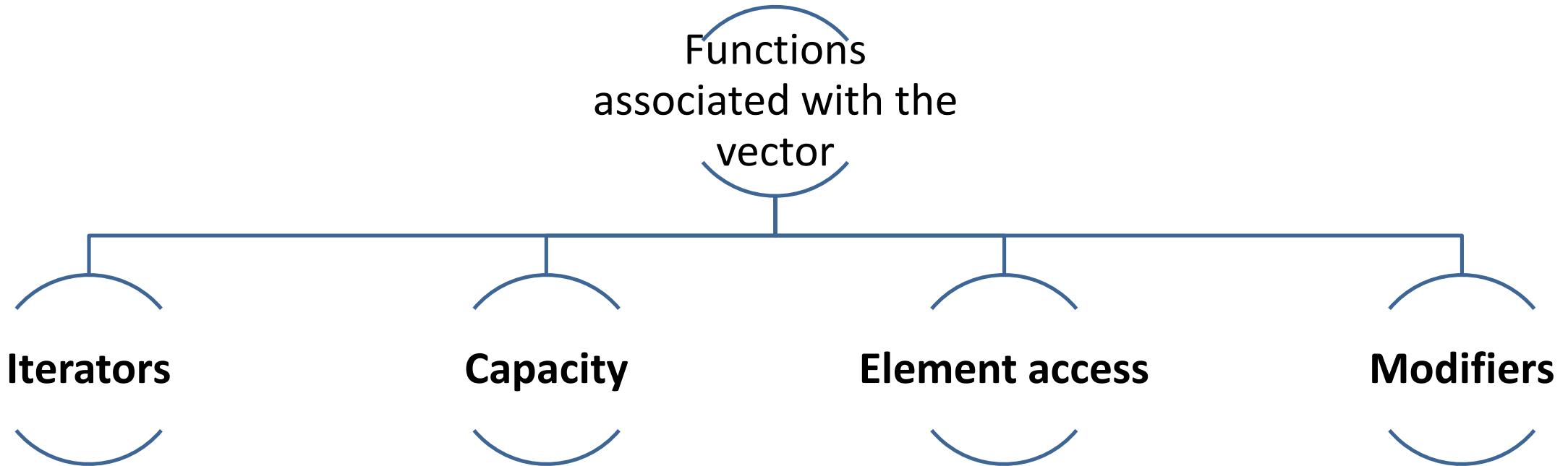
# Sequence Containers

Implement data structures which can be accessed in a sequential manner.

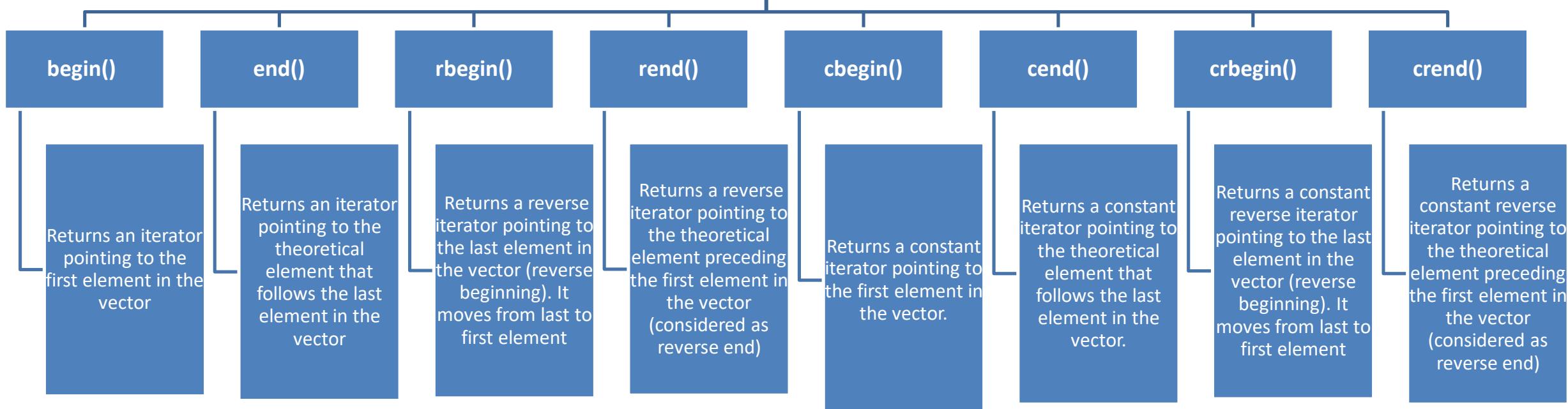
- vector
- list
- deque
- arrays
- forward-list

# Sequence Containers: Vector

- Vectors are same as dynamic arrays with the ability to resize itself automatically when an element is inserted or deleted, with their storage being handled automatically by the container.
- Vector elements are placed in contiguous storage so that they can be accessed and traversed using iterators. In vectors, data is inserted at the end.
- Inserting at the end takes differential time, as sometimes there may be a need of extending the array.
- Removing the last element takes only constant time because no resizing happens. Inserting and erasing at the beginning or in the middle is linear in time.



# Iterators



```
// C++ program to illustrate the iterators in vector
#include <iostream>
#include <vector>

using namespace std;

int main()
{
    vector<int> g1;

    for (int i = 1; i <= 5; i++)
        g1.push_back(i);

    cout << "Output of begin and end: ";
    for (auto i = g1.begin(); i != g1.end(); ++i)
        cout << *i << " ";

    cout << "\nOutput of cbegin and cend: ";
    for (auto i = g1.cbegin(); i != g1.cend(); ++i)
        cout << *i << " ";

    cout << "\nOutput of rbegin and rend: ";
    for (auto ir = g1.rbegin(); ir != g1.rend(); ++ir)
        cout << *ir << " ";

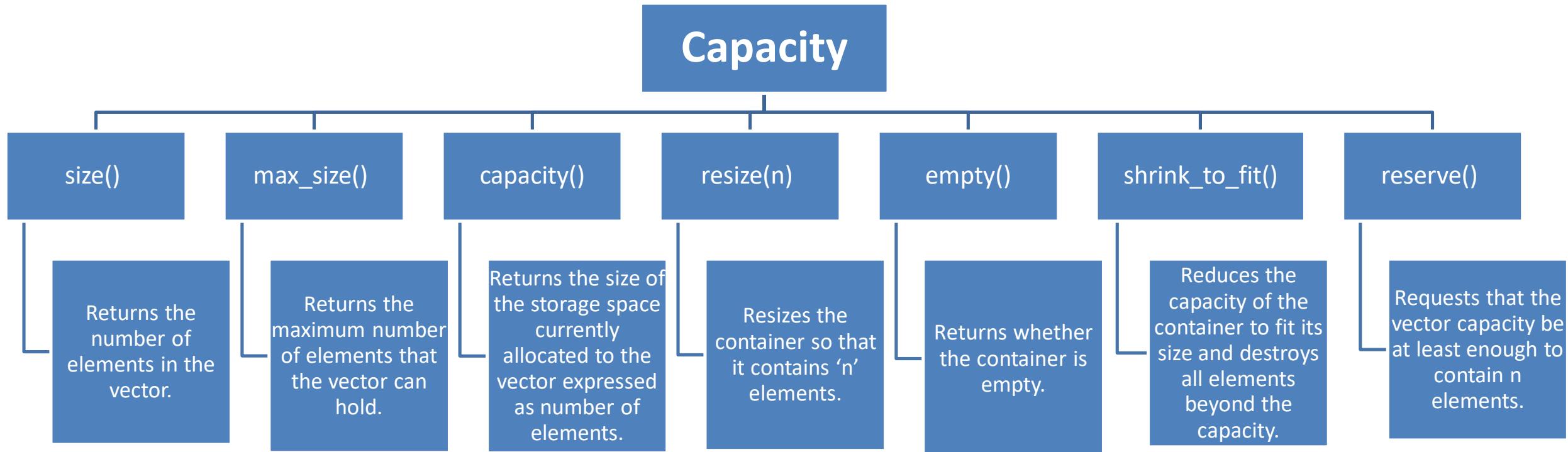
    cout << "\nOutput of crbegin and crend : ";
    for (auto ir = g1.crbegin(); ir != g1.crend(); ++ir)
        cout << *ir << " ";

    return 0;
}
```

### Output:

Output of begin and end: 1 2 3 4 5  
Output of cbegin and cend: 1 2 3 4 5  
Output of rbegin and rend: 5 4 3 2 1  
Output of crbegin and crend : 5 4 3 2 1

# functions associated with the vector



```
// C++ program to illustrate the capacity function in vector
#include <iostream>
#include <vector>

using namespace std;

int main()
{
    vector<int> g1;

    for (int i = 1; i <= 5; i++)
        g1.push_back(i);

    cout << "Size : " << g1.size();
    cout << "\nCapacity : " << g1.capacity();
    cout << "\nMax_Size : " << g1.max_size();

    // resizes the vector size to 4
    g1.resize(4);

    // prints the vector size after resize()
    cout << "\nSize : " << g1.size();

    // checks if the vector is empty or not
    if (g1.empty() == false)
        cout << "\nVector is not empty";
    else
        cout << "\nVector is empty";

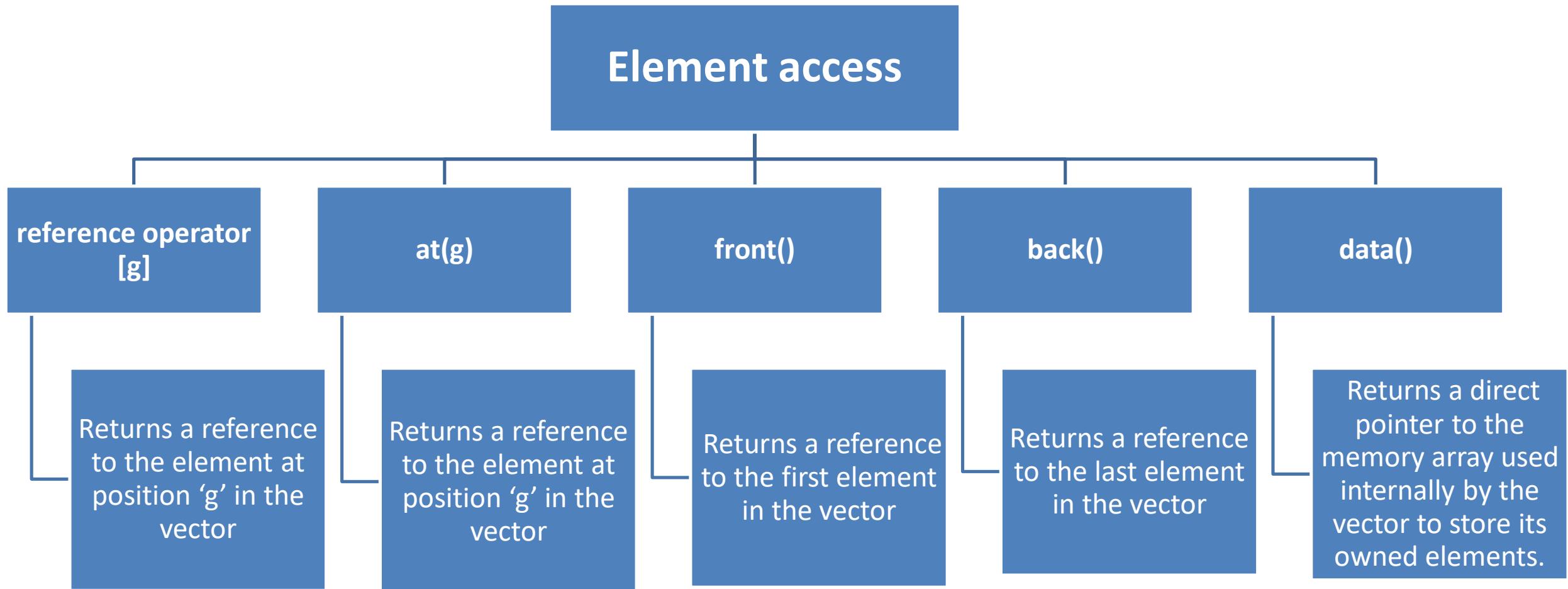
    // Shrinks the vector
    g1.shrink_to_fit();
    cout << "\nVector elements are: ";
    for (auto it = g1.begin(); it != g1.end(); it++)
        cout << *it << " ";

    return 0;
}
```

**Output:**

```
Size : 5
Capacity : 8
Max_Size : 4611686018427387903
Size : 4
Vector is not empty
Vector elements are: 1 2 3 4
```

# functions associated with the vector



```
// C++ program to illustrate the element accesser in vector
#include <bits/stdc++.h>
using namespace std;

int main()
{
    vector<int> g1;

    for (int i = 1; i <= 10; i++)
        g1.push_back(i * 10);

    cout << "\nReference operator [g] : g1[2] = " << g1[2];
    cout << "\nat : g1.at(4) = " << g1.at(4);
    cout << "\nfront() : g1.front() = " << g1.front();
    cout << "\nback() : g1.back() = " << g1.back();

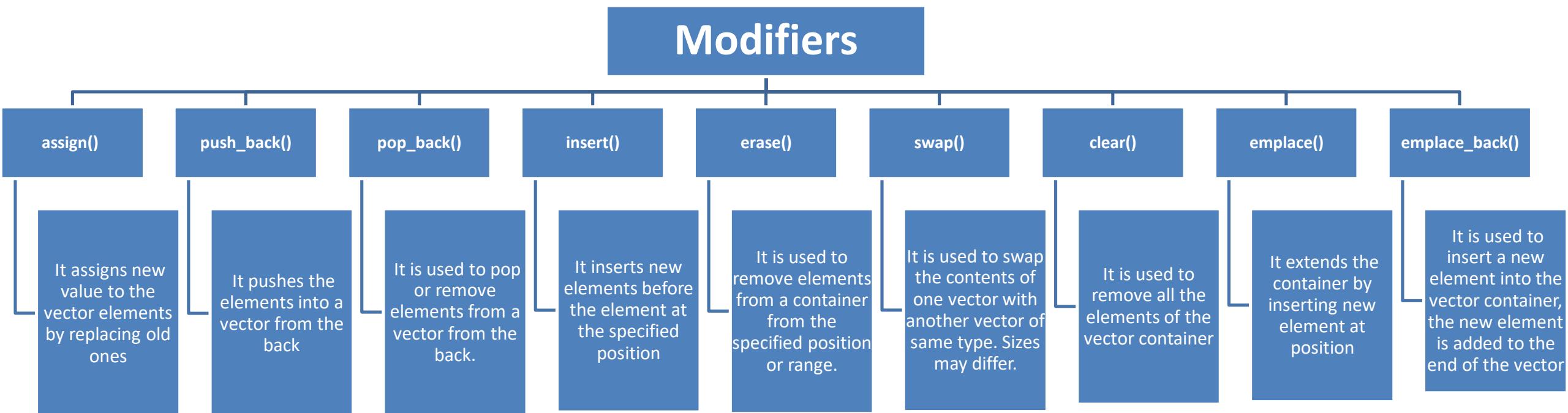
    // pointer to the first element
    int* pos = g1.data();

    cout << "\nThe first element is " << *pos;
    return 0;
}
```

**Output:**

Reference operator [g] : g1[2] = 30  
at : g1.at(4) = 50  
front() : g1.front() = 10  
back() : g1.back() = 100  
The first element is 10

# functions associated with the vector



# Sequence Container: List

- Lists are sequence containers that allow non-contiguous memory allocation.
- As compared to vector, list has slow traversal, but once a position has been found, insertion and deletion are quick.
- Normally, when we say a List, we talk about doubly linked list. For implementing a singly linked list, we use forward list.

```
#include <iostream>
#include <list>
#include <iterator>
using namespace std;
//function for printing the elements in a list
void showlist(list <int> g)
{
    list <int> :: iterator it;
    for(it = g.begin(); it != g.end(); ++it)
        cout << '\t' << *it;
    cout << '\n';
}
int main()
{
list <int> gqlist1, gqlist2;
for (int i = 0; i < 10; ++i)
{
    gqlist1.push_back(i * 2);
    gqlist2.push_front(i * 3);
}
cout << "\nList 2 (gqlist2) is : ";
cout << "\nList 1 (gqlist1) is : ";
showlist(gqlist1);
showlist(gqlist2);
cout << "\ngqlist1.front() : " << gqlist1.front();
cout << "\ngqlist1.back() : " << gqlist1.back();
cout << "\ngqlist1.pop_front() : ";
gqlist1.pop_front();
showlist(gqlist1);
cout << "\ngqlist2.pop_back() : ";
gqlist2.pop_back();
showlist(gqlist2);
cout << "\ngqlist1.reverse() : ";
gqlist1.reverse();
showlist(gqlist1);
cout << "\ngqlist2.sort(): ";
gqlist2.sort();
showlist(gqlist2);
return 0;
}
```

The output of the above program is :

```
List 1 (gqlist1) is : 0 2 4 6 8 10 12 14 16 18
List 2 (gqlist2) is : 27 24 21 18 15 12 9 6 3 0
gqlist1.front() : 0
gqlist1.back() : 18
gqlist1.pop_front() : 2 4 6 8 10 12 14 16 18
gqlist2.pop_back() : 27 24 21 18 15 12 9 6 3
gqlist1.reverse() : 18 16 14 12 10 8 6 4 2
gqlist2.sort(): 3 6 9 12 15 18 21 24 27
```

# functions associated with the Lists

•front()

back()

push\_front(g)

push\_back(g)

pop\_front()

pop\_back()

•Returns the value  
of the first element  
in the list.

Returns the value  
of the last element  
in the list .

Adds a new  
element 'g' at the  
beginning of the  
list .

Adds a new  
element 'g' at the  
end of the list.

Removes the first  
element of the list,  
and reduces size  
of the list by 1.

Removes the last  
element of the list,  
and reduces size  
of the list by 1

# functions associated with the Lists

begin()	end()	rbegin()	rend()	cbegin()	cend()	crbegin()	crend()
<p><b>begin()</b> function returns an iterator pointing to the first element of the list</p>	<p><b>end()</b> function returns an iterator pointing to the theoretical last element which follows the last element.</p>	<p>returns a reverse iterator which points to the last element of the list.</p>	<p>returns a reverse iterator which points to the position before the beginning of the list.</p>	<p>returns a constant random access iterator which points to the beginning of the list.</p>	<p>returns a constant random access iterator which points to the end of the list.</p>	<p>returns a constant reverse iterator which points to the last element of the list i.e reversed beginning of container.</p>	<p>returns a constant reverse iterator which points to the theoretical element preceding the first element in the list i.e. the reverse end of the list.</p>

# functions associated with the Lists

**empty()**

**insert()**

**erase()**

**assign()**

**remove()**

Returns whether the list is empty(1) or not(0).

Inserts new elements in the list before the element at a specified position.

Removes a single element or a range of elements from the list.

Assigns new elements to list by replacing current elements and resizes the list.

**Removes all the elements from the list, which are equal to given element.**

# functions associated with the Lists

**reverse()**

Reverses the list.

**size()**

Returns the number of elements in the list.

**list resize()**

Used to resize a list container.

**sort()**

Sorts the list in increasing order.

# functions associated with the Lists

**max\_size()**

**unique()**

**emplace\_front()**

**emplace\_back()**

**clear()**

Returns the maximum number of elements a list container can hold.

Removes all duplicate consecutive elements from the list.

function is used to insert a new element into the list container, the new element is added to the beginning of the list.

function is used to insert a new element into the list container, the new element is added to the end of the list.

function is used to remove all the elements of the list container, thus making it size 0.

# functions associated with the Lists

**operator=**

This operator is used to assign new contents to the container by replacing the existing contents.

**swap()**

This function is used to swap the contents of one list with another list of same type and size.

**splice()**

Used to transfer elements from one list to another.

**merge()**

Merges two sorted lists into one

**emplace()**

Extends list by inserting new element at a given position.

# 18CSC202J - OBJECT ORIENTED DESIGN AND PROGRAMMING

UNIT V

SESSION 1

TOPIC:STL CONTAINERS

SEQUENCE CONTAINER

**STL has four  
components**

Containers

Algorithms

Functions

Iterators

# Sequence Containers

Implement data structures which can be accessed in a sequential manner.

- vector
- list
- deque
- arrays
- forward-list

# Sequence Containers: Deque

- ▶ Double ended queues are sequence containers with the feature of expansion and contraction on both the ends.
- ▶ They are similar to vectors, but are more efficient in case of insertion and deletion of elements. Unlike vectors, contiguous storage allocation may not be guaranteed.
- ▶ Double Ended Queues are basically an implementation of the data structure double ended queue. A queue data structure allows insertion only at the end and deletion from the front.
- ▶ This is like a queue in real life, wherein people are removed from the front and added at the back. Double ended queues are a special case of queues where insertion and deletion operations are possible at both the ends.
- ▶ The functions for deque are same as vector, with an addition of push and pop operations for both front and back.

# Methods of Deque

## •deque insert()

•Inserts an element. And returns an iterator that points to the first of the newly inserted elements.

## •rbegin()

•Returns a reverse iterator which points to the last element of the deque (i.e., its reverse beginning).

## •rend()

• Returns a reverse iterator which points to the position before the beginning of the deque (which is considered its reverse end).

## •cbegin()

• Returns a constant iterator pointing to the first element of the container, that is, the iterator cannot be used to modify, only traverse the deque.

## •max\_size()

• Returns the maximum number of elements that a deque container can hold.

## •assign()

• Assign values to the same or different deque container.

## •resize()

• Function which changes the size of the deque.

```

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

void showdq(deque <int> g)
{
    deque <int> :: iterator it;
    for (it = g.begin(); it != g.end(); ++it)
        cout << '\t' << *it;
    cout << '\n';
}

int main()
{
    deque <int> gquiz;
    gquiz.push_back(10);
    gquiz.push_front(20);
    gquiz.push_back(30);
    gquiz.push_front(15);
    cout << "The deque gquiz is : ";
    showdq(gquiz);

    cout << "\ngquiz.size() : " << gquiz.size();
    cout << "\ngquiz.max_size() : " << gquiz.max_size();

    cout << "\ngquiz.at(2) : " << gquiz.at(2);
    cout << "\ngquiz.front() : " << gquiz.front();
    cout << "\ngquiz.back() : " << gquiz.back();

    cout << "\ngquiz.pop_front() : ";
    gquiz.pop_front();
    showdq(gquiz);

    cout << "\ngquiz.pop_back() : ";
    gquiz.pop_back();
    showdq(gquiz);

    return 0;
}

```

## **OUTPUT:**

```

The deque gquiz is : 15 20 10 30
gquiz.size() : 4
gquiz.max_size() : 4611686018427387903
gquiz.at(2) : 10
gquiz.front() : 15
gquiz.back() : 30
gquiz.pop_front() : 20 10 30
gquiz.pop_back() : 20 10

```

# Methods of Deque

•push_front()	•push_back()	•pop_front()	•pop_back()	•front()	•back()	•clear()	•erase()	•empty()	•size()
• This function is used to push elements into a deque from the front.	• This function is used to push elements into a deque from the back.	• Is used to pop or remove elements from a deque from the front.	• Is used to pop or remove elements from a deque from the back.	• Is used to reference the first element of the deque container.	• Is used to reference the last element of the deque container.	• Is used to remove all the elements of the deque container, thus making its size 0.	• Is used to remove elements from a container from the specified position or range.	• Is used to check if the deque container is empty or not.	• Is used to return the size of the deque container or the number of elements in the deque container.

# Sequence Containers: Array

- ▶ The introduction of array class from C++11 has offered a better alternative for C-style arrays. The advantages of array class over C-style array are :-
- ▶ Array classes knows its own size, whereas C-style arrays lack this property. So when passing to functions, we don't need to pass size of Array as a separate parameter.
- ▶ With C-style array there is more risk of array being decayed into a pointer. Array classes don't decay into pointers
- ▶ Array classes are generally more efficient, light-weight and reliable than C-style arrays.
- ▶

# Operations on array

at()

This function is used to access the elements of array.

get()

This function is also used to access the elements of array.

This function is not the member of array class but overloaded function from class tuple.

operator[]

This is similar to C-style arrays. This method is also used to access array elements.

```
/// C++ code to demonstrate working of array, to() and get()
#include<iostream>
#include<array> // for array, at()
#include<tuple> // for get()
using namespace std;
int main()
{
    // Initializing the array elements
    array<int,6> ar = {1, 2, 3, 4, 5, 6};

    // Printing array elements using at()
    cout << "The array elements are (using at()) : ";
    for ( int i=0; i<6; i++)
        cout << ar.at(i) << " ";
    cout << endl;

    // Printing array elements using get()
    cout << "The array elements are (using get()) : ";
    cout << get<0>(ar) << " " << get<1>(ar) << " ";
    cout << get<2>(ar) << " " << get<3>(ar) << " ";
    cout << get<4>(ar) << " " << get<5>(ar) << " ";
    cout << endl;

    // Printing array elements using operator[]
    cout << "The array elements are (using operator[]) : ";
    for ( int i=0; i<6; i++)
        cout << ar[i] << " ";
    cout << endl;

    return 0;
}
```

#### Output:

```
The array elements are (using at()) : 1 2 3 4 5 6
The array elements are (using get()) : 1 2 3 4 5 6
The array elements are (using operator[]) : 1 2 3 4
5 6
```

# Operations on array

**front()**

This returns the  
first element of  
array.

**back()**

This returns the  
last element of  
array.

```
// C++ code to demonstrate working of front() and back()
#include<iostream>
#include<array> // for front() and back()
using namespace std;
int main()
{
    // Initializing the array elements
    array<int,6> ar = {1, 2, 3, 4, 5, 6};

    // Printing first element of array
    cout << "First element of array is : ";
    cout << ar.front() << endl;

    // Printing last element of array
    cout << "Last element of array is : ";
    cout << ar.back() << endl;

    return 0;
}
```

Output:

First element of array is : 1  
Last element of array is : 6

# Operations on array

**size()**

It returns the number of elements in array. This is a property that C-style arrays lack.

**max\_size()**

It returns the maximum number of elements array can hold i.e, the size with which array is declared.

```
// C++ code to demonstrate working of size() and max_size()
#include<iostream>
#include<array> // for size() and max_size()
using namespace std;
int main()
{
    // Initializing the array elements
    array<int,6> ar = {1, 2, 3, 4, 5, 6};

    // Printing number of array elements
    cout << "The number of array elements is : ";
    cout << ar.size() << endl;

    // Printing maximum elements array can hold
    cout << "Maximum elements array can hold is : ";
    cout << ar.max_size() << endl;

    return 0;
}
```

Output:

The number of array elements is : 6  
Maximum elements array can hold is : 6

# Operations on array

**size()**

It returns the number of elements in array. This is a property that C-style arrays lack.

**max\_size()**

It returns the maximum number of elements array can hold i.e, the size with which array is declared.

**swap()** :- The swap() swaps all elements of one array with other.

```
// C++ code to demonstrate working of swap()
#include<iostream>
#include<array> // for swap() and array
using namespace std;
int main()
{
    // Initializing 1st array
    array<int,6> ar = {1, 2, 3, 4, 5, 6};

    // Initializing 2nd array
    array<int,6> ar1 = {7, 8, 9, 10, 11, 12};

    // Printing 1st and 2nd array before swapping
    cout << "The first array elements before swapping are : ";
    for (int i=0; i<6; i++)
        cout << ar[i] << " ";
    cout << endl;
    cout << "The second array elements before swapping
are : ";

    for (int i=0; i<6; i++)
        cout << ar1[i] << " ";
    cout << endl;
    // Swapping ar1 values with ar
    ar.swap(ar1);

    // Printing 1st and 2nd array after swapping
    cout << "The first array elements after swapping are : ";
    for (int i=0; i<6; i++)
        cout << ar[i] << " ";
    cout << endl;
    cout << "The second array elements after swapping are : ";
    for (int i=0; i<6; i++)
        cout << ar1[i] << " ";
    cout << endl;
    return 0;
}
```

Output:

The first array elements before swapping are : 1 2 3 4 5 6

The second array elements before swapping are : 7 8 9 10 11 12

The first array elements after swapping are : 7 8 9 10 11 12

The second array elements after swapping are : 1 2 3 4 5 6

# Operations on array

**empty()**

This function returns true  
when the array size is zero  
else returns false.

**fill()**

This function is used to fill  
the entire array with a  
particular value.

```
// C++ code to demonstrate working of empty() and fill()
#include<iostream>
#include<array> // for fill() and empty()
using namespace std;
int main()
{
    array<int,6> ar;    // Declaring 1st array
    array<int,0> ar1;  // Declaring 2nd array
    ar1.empty()? cout << "Array empty":
    cout << "Array not empty";
    cout << endl;      // Checking size of array if it is empty
    // Filling array with 0
    ar.fill(0);
    // Displaying array after filling
    cout << "Array after filling operation is : ";
    for ( int i=0; i<6; i++)
        cout << ar[i] << " ";
    return 0;
}
```

Output:  
Array empty  
Array after filling operation is : 0 0 0 0 0 0

# **18CSC202J - OBJECT ORIENTED DESIGN AND PROGRAMMING**

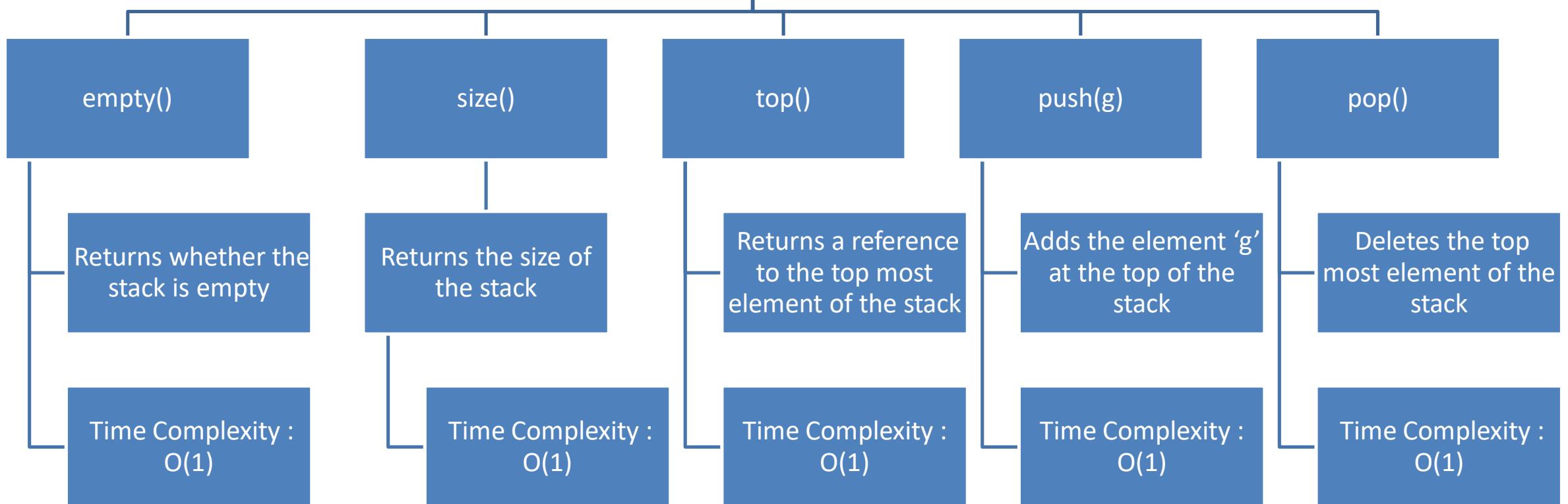
## **Session 3**

**Topic : Stack,  
Associative containers:Map,MultiMap**

# STL Stack

- Stacks are a type of container adaptors with LIFO(Last In First Out) type of working, where a new element is added at one end and (top) an element is removed from that end only.

## Functions associated with stack



```
// CPP program to demonstrate working of STL stack
#include <iostream>
#include <stack>
using namespace std;

void showstack(stack <int> s)
{
    while (!s.empty())
    {
        cout << '\t' << s.top();
        s.pop();
    }
    cout << '\n';
}

int main ()
{
    stack <int> s;
    s.push(10);
    s.push(30);
    s.push(20);
    s.push(5);
    s.push(1);

    cout << "The stack is : ";
    showstack(s);

    cout << "\ns.size() : " << s.size();
    cout << "\ns.top() : " << s.top();

    cout << "\ns.pop() : ";
    s.pop();
    showstack(s);

    return 0;
}
```

### Output:

The stack is : 1 5 20 30 10  
s.size() : 5  
s.top() : 1  
s.pop() : 5 20 30 10

# List of functions of Stack



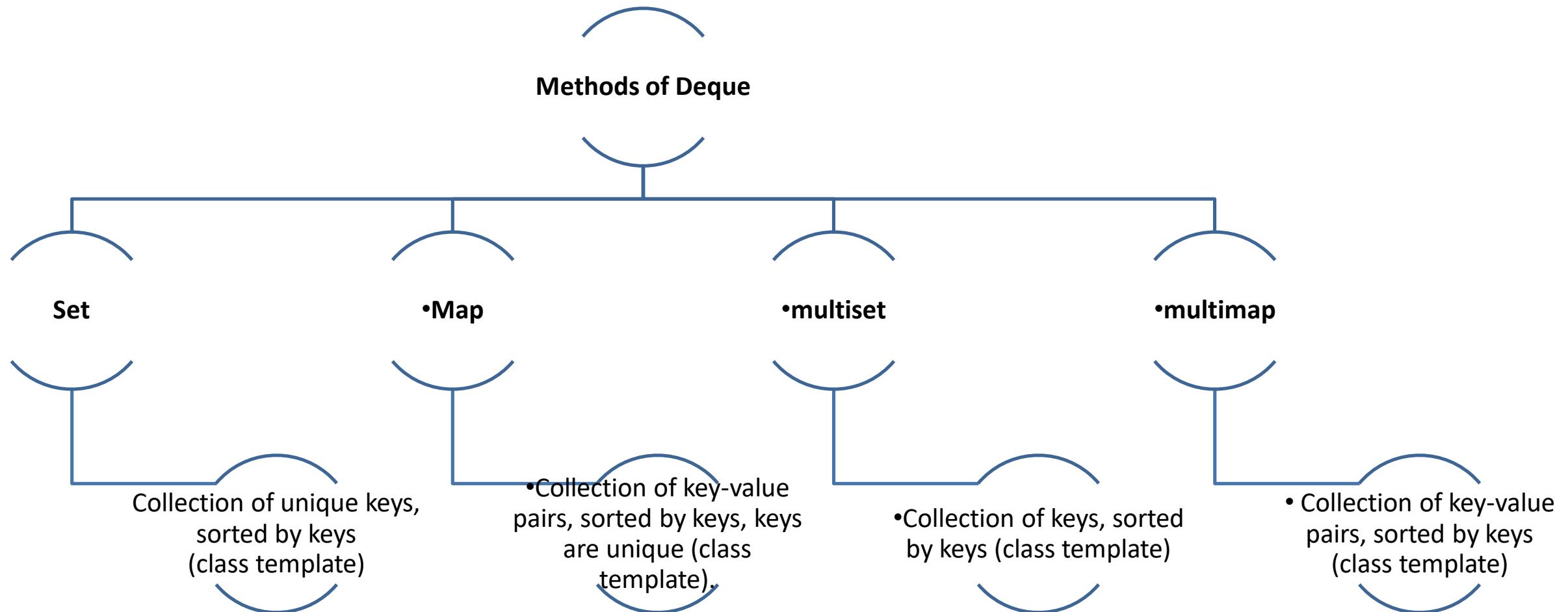
# Associative containers

Map and Multimap

# Associative containers

They implement sorted data structures that can be quickly searched ( $O(\log n)$  complexity).

# Methods of Deque



# Associative containers: Map

- Maps are associative containers that store elements in a mapped fashion.
- Each element has a key value and a mapped value. No two mapped values can have same key values.

# basic functions associated with Map

begin()	end()	size()	max_size()	empty()	pair insert(keyvalue, mapvalue)	erase(iterator position)	erase(const g)	clear()
Returns an iterator to the first element in the map	Returns an iterator to the theoretical element that follows last element in the map	Returns the number of elements in the map	Returns the maximum number of elements that the map can hold	Returns whether the map is empty	Adds a new element to the map	Removes the element at the position pointed by the iterator	Removes the key value 'g' from the map	Removes all the elements from the map

# Associative containers: Multimap

- Multimap is similar to map with an addition that multiple elements can have same keys. Rather than each element being unique, the key value and mapped value pair has to be unique in this case.

# basic functions associated with multiMap

•begin()

•end()

•size()

•max\_size()

•empty()

•pair<int,int>  
insert(keyvalue,multimap  
value)

•Returns an iterator to the  
first element in the  
multimap

•Returns an iterator to the  
theoretical element that  
follows last element in the  
multimap

• Returns the number of  
elements in the multimap

•Returns the maximum  
number of elements that  
the multimap can hold

•Returns whether the  
multimap is empty

•Adds a new element to  
the multimap

```

#include <iostream>
#include <map>
#include <iterator>

using namespace std;

int main()
{
    multimap <int, int> gquiz1; // empty multimap container
    // insert elements in random order
    gquiz1.insert(pair <int, int> (1, 40));
    gquiz1.insert(pair <int, int> (2, 30));
    gquiz1.insert(pair <int, int> (3, 60));
    gquiz1.insert(pair <int, int> (4, 20));
    gquiz1.insert(pair <int, int> (5, 50));
    gquiz1.insert(pair <int, int> (6, 50));
    gquiz1.insert(pair <int, int> (6, 10));

    // printing multimap gquiz1
    multimap <int, int> :: iterator itr;
    cout << "\nThe multimap gquiz1 is : \n";
    cout << "\tKEY\tELEMENT\n";
    for (itr = gquiz1.begin(); itr != gquiz1.end(); ++itr)
    {
        cout << '\t' << itr->first
            << '\t' << itr->second << '\n';
    }
    cout << endl;

    // assigning the elements from gquiz1 to gquiz2
    multimap <int, int> gquiz2(gquiz1.begin(),gquiz1.end());

    // print all elements of the multimap gquiz2
    cout << "\nThe multimap gquiz2 after assign from gquiz1 is
    : \n";
    cout << "\tKEY\tELEMENT\n";
    for (itr = gquiz2.begin(); itr != gquiz2.end(); ++itr)
    {
        cout << '\t' << itr->first
            << '\t' << itr->second << '\n';
    }
    cout << endl;
}

// remove all elements up to element with value 30 in gquiz2
key=3 : \n";
cout << "\ngquiz2 after removal of elements less than
key=3 : \n";
cout << "\tKEY\tELEMENT\n";
gquiz2.erase(gquiz2.begin(), gquiz2.find(3));
for (itr = gquiz2.begin(); itr != gquiz2.end(); ++itr)
{
    cout << '\t' << itr->first
        << '\t' << itr->second << '\n';
}

// remove all elements with key = 4
int num;
num = gquiz2.erase(4);
cout << "\ngquiz2.erase(4) : ";
cout << num << " removed \n" ;
cout << "\tKEY\tELEMENT\n";
for (itr = gquiz2.begin(); itr != gquiz2.end(); ++itr)
{
    cout << '\t' << itr->first
        << '\t' << itr->second << '\n';
}

cout << endl;

//lower bound and upper bound for multimap gquiz1 key =
5
cout << "gquiz1.lower_bound(5) : " << "\tKEY = ";
cout << gquiz1.lower_bound(5)->first << '\t';
cout << "\tELEMENT = " << gquiz1.lower_bound(5)-
>second << endl;
cout << "gquiz1.upper_bound(5) : " << "\tKEY = ";
cout << gquiz1.upper_bound(5)->first << '\t';
cout << "\tELEMENT = " << gquiz1.upper_bound(5)-
>second << endl;
return 0;
}

```

Output:

The multimap gquiz1 is :

KEY	ELEMENT
1	40
2	30
3	60
4	20
5	50
6	50
6	10

The multimap gquiz2 after assign from gquiz1 is :

KEY	ELEMENT
1	40
2	30
3	60
4	20
5	50
6	50
6	10

gquiz2 after removal of elements less than key=3 :

KEY	ELEMENT
3	60
4	20
5	50
6	50
6	10

gquiz2.erase(4) : 1 removed

KEY	ELEMENT
3	60
5	50
6	50
6	10

gquiz1.lower\_bound(5) : KEY = 5 ELEMENT = 50

gquiz1.upper\_bound(5) : KEY = 6 ELEMENT = 50

# 18CSC202J - OBJECT ORIENTED DESIGN AND PROGRAMMING

## Session 4

### Topic :STL Iterators

**STL has four  
components**

Containers

Iterators

Algorithms

Functions

# STL Iterators

- Are used to point at the memory addresses of STL containers.
- They are primarily used in sequence of numbers, characters etc.
- They reduce the complexity and execution time of program.

## Operations of iterators

`begin()`

`end()`

This function is used to return the **beginning position** of the container.

This function is used to return the **after end position** of the container.

```
// C++ code to demonstrate the working of iterator, begin() and end()

#include<iostream>
#include<iterator> // for iterators
#include<vector> // for vectors
using namespace std;

int main()
{
    vector<int> ar = { 1, 2, 3, 4, 5 };

    // Declaring iterator to a vector
    vector<int>::iterator ptr;

    // Displaying vector elements using begin() and end()
    cout << "The vector elements are :";

    for (ptr = ar.begin(); ptr < ar.end(); ptr++)
        cout << *ptr << " ";

    return 0;
}
```

**Output:**

The vector elements are : 1 2 3 4 5

**advance()** :- This function is used to **increment the iterator position** till the specified number mentioned in its arguments.

```
// C++ code to demonstrate the working of advance()
#include<iostream>
#include<iterator> // for iterators
#include<vector> // for vectors
using namespace std;
int main()
{
    vector<int> ar = { 1, 2, 3, 4, 5 };

    // Declaring iterator to a vector
    vector<int>::iterator ptr = ar.begin();

    // Using advance() to increment iterator position
    // points to 4
    advance(ptr, 3);

    // Displaying iterator position
    cout << "The position of iterator after advancing is : ";
    cout << *ptr << " ";

    return 0;
}
```

**Output:**

The position of iterator after advancing is : 4

## Operations of iterators

`next()`

`prev()`

This function **returns the new iterator** that the iterator would point after **advancing the positions** mentioned in its arguments.

This function **returns the new iterator** that the iterator would point after **decrementing the positions** mentioned in its arguments.

```
// C++ code to demonstrate the working of next() and prev()
#include<iostream>
#include<iterator> // for iterators
#include<vector> // for vectors
using namespace std;
int main()
{
    vector<int> ar = { 1, 2, 3, 4, 5 };

    // Declaring iterators to a vector
    vector<int>::iterator ptr = ar.begin();
    vector<int>::iterator ftr = ar.end();

    // Using next() to return new iterator
    // points to 4
    auto it = next(ptr, 3);

    // Using prev() to return new iterator
    // points to 3
    auto it1 = prev(ftr, 3);

    // Displaying iterator position
    cout << "The position of new iterator using next() is : ";
    cout << *it << " ";
    cout << endl;

    // Displaying iterator position
    cout << "The position of new iterator using prev() is : ";
    cout << *it1 << " ";
    cout << endl;

    return 0;
}
```

### Output:

The position of new iterator using next() is : 4  
The position of new iterator using prev() is : 3

**inserter()** :- This function is used to **insert the elements at any position** in the container. It accepts **2 arguments, the container and iterator to position where the elements have to be inserted.**

```
// C++ code to demonstrate the working of inserter()
#include<iostream>
#include<iterator> // for iterators
#include<vector> // for vectors
using namespace std;
int main()
{
    vector<int> ar = { 1, 2, 3, 4, 5 };
    vector<int> ar1 = {10, 20, 30};

    // Declaring iterator to a vector
    vector<int>::iterator ptr = ar.begin();

    // Using advance to set position
    advance(ptr, 3);

    // copying 1 vector elements in other using inserter()
    // inserts ar1 after 3rd position in ar
    copy(ar1.begin(), ar1.end(), inserter(ar,ptr));

    // Displaying new vector elements
    cout << "The new vector after inserting elements is : ";
    for (int &x : ar)
        cout << x << " ";

    return 0;
}
```

**Output:**

The new vector after inserting elements is : 1 2 3 10 20 30 4 5

# 18CSC202J - OBJECT ORIENTED DESIGN AND PROGRAMMING

## Session 5

**Topic : STL Algorithm  
Function Objects**

**STL has four  
components**

Containers

Iterators

Algorithms

Functions

# STL Algorithms

- Are used to point at the memory addresses of STL containers.
- They are primarily used in sequence of numbers, characters etc.
- They reduce the complexity and execution time of program.

# STL ALgorithms

- STL has an ocean of algorithms, for all < algorithm > library functions
- Some of the most used algorithms on vectors and most useful one's in Competitive Programming are mentioned as follows :
  - **sort(first\_iterator, last\_iterator)** – To sort the given vector.
  - **reverse(first\_iterator, last\_iterator)** – To reverse a vector.
  - **\*max\_element (first\_iterator, last\_iterator)** – To find the maximum element of a vector.
  - **\*min\_element (first\_iterator, last\_iterator)** – To find the minimum element of a vector.
  - **accumulate(first\_iterator, last\_iterator, initial value of sum)** – Does the summation of vector elements

```
// A C++ program to demonstrate working of sort(), reverse()
#include <algorithm>
#include <iostream>
#include <vector>
#include <numeric> //For accumulate operation
using namespace std;
```

```
int main()
{
    // Initializing vector with array values
    int arr[] = {10, 20, 5, 23 ,42 ,15};
    int n = sizeof(arr)/sizeof(arr[0]);
    vector<int> vect(arr, arr+n);

    cout << "Vector is: ";
    for (int i=0; i<n; i++)
        cout << vect[i] << " ";
}
```

```
// Sorting the Vector in Ascending order
sort(vect.begin(), vect.end());
```

```
cout << "\nVector after sorting is: ";
for (int i=0; i<n; i++)
    cout << vect[i] << " ";
```

```
// Reversing the Vector
reverse(vect.begin(), vect.end());

cout << "\nVector after reversing is: ";
for (int i=0; i<6; i++)
    cout << vect[i] << " ";

cout << "\nMaximum element of vector is: ";
cout << *max_element(vect.begin(), vect.end());

cout << "\nMinimum element of vector is: ";
cout << *min_element(vect.begin(), vect.end());

// Starting the summation from 0
cout << "\nThe summation of vector elements is: ";
cout << accumulate(vect.begin(), vect.end(), 0);

return 0;
}
```

Output:

Vector before sorting is: 10 20 5 23 42 15  
Vector after sorting is: 5 10 15 20 23 42  
Vector before reversing is: 5 10 15 20 23 42  
Vector after reversing is: 42 23 20 15 10 5  
Maximum element of vector is: 42

Minimum element of vector is: 5  
The summation of vector elements is: 115

**6.count(first\_iterator, last\_iterator,x)** – To count the occurrences of x in vector.

**7.find(first\_iterator, last\_iterator, x)** – Points to last address of vector ((name\_of\_vector).end()) if element is not present in vector.

```
// C++ program to demonstrate working of count() and find()
#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;

int main()
{
    // Initializing vector with array values
    int arr[] = {10, 20, 5, 23, 42, 20, 15};
    int n = sizeof(arr)/sizeof(arr[0]);
    vector<int> vect(arr, arr+n);

    cout << "Occurrences of 20 in vector : ";

    // Counts the occurrences of 20 from 1st to
    // last element
    cout << count(vect.begin(), vect.end(), 20);

    // find() returns iterator to last address if
    // element not present
    find(vect.begin(), vect.end(), 5) != vect.end()?
        cout << "\nElement found":
        cout << "\nElement not found";

    return 0;
}
```

**Output:**

Occurrences of 20 in vector: 2  
Element found

# merge() in C++ STL

- C++ offers in its STL library a merge() which is quite useful to **merge sort two containers** into a **single** container.  
It is defined in header “**algorithm**”. It is implemented in two ways.
- **Syntax 1 : Using operator “<”**

```
// C++ code to demonstrate the working of merge() implementation 1
```

```
#include <bits/stdc++.h>
using namespace std;

int main()
{
    // initializing 1st container
    vector<int> arr1 = { 1, 4, 6, 3, 2 };

    // initializing 2nd container
    vector<int> arr2 = { 6, 2, 5, 7, 1 };

    // declaring resultant container
    vector<int> arr3(10);

    // sorting initial containers
    sort(arr1.begin(), arr1.end());
    sort(arr2.begin(), arr2.end());

    // using merge() to merge the initial containers
    merge(arr1.begin(), arr1.end(), arr2.begin(), arr2.end(),
          arr3.begin());

    // printing the resultant merged container
    cout << "The container after merging initial containers is : ";

    for (int i = 0; i < arr3.size(); i++)
        cout << arr3[i] << " ";
    return 0;
}
```

Output:

The container after merging initial containers  
is : 1 1 2 2 3 4 5 6 6 7

# Functors in C++

# Function objects

- Consider a function that takes only one argument.
- However, while calling this function we have a lot more information that we would like to pass to this function, but we cannot as it accepts only one parameter. What can be done?
- One obvious answer might be global variables.
- However, good coding practices do not advocate the use of global variables and say they must be used only when there is no other alternative.
- **Functors** are objects that can be treated as though they are a function or function pointer.
- Functors are most commonly used along with STLs.

```
// A C++ program uses transform() in STL to add 1 to all elements of arr[]
#include <bits/stdc++.h>
using namespace std;

int increment(int x) { return (x+1);}

int main()
{
    int arr[] = {1, 2, 3, 4, 5};
    int n = sizeof(arr)/sizeof(arr[0]);

    // Apply increment to all elements of
    // arr[] and store the modified elements
    // back in arr[]
    transform(arr, arr+n, arr, increment);

    for (int i=0; i<n; i++)
        cout << arr[i] << " ";

    return 0;
}
```

Output:  
2 3 4 5 6

# **18CSC202J - OBJECT ORIENTED DESIGN AND PROGRAMMING**

## **Session 6**

**Topic : Streams and Files**

# Learning Objectives

- C++ I/O streams.
- Reading and writing sequential files.
- Reading and writing random access files.

# C++ Files and Streams

- C++ views each files as a sequence of bytes.
- Each file ends with an *end-of-file* marker.
- When a file is *opened*, an object is created and a stream is associated with the object.
- To perform file processing in C++, the header files **<iostream.h>** and **<fstream.h>** must be included.
- **<fstream.>** includes **<ifstream>** and **<ofstream>**

# Creating a sequential file

```
// Fig. 14.4: fig14_04.cpp D&D p.708
// Create a sequential file
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>
int main()
{
    // ofstream constructor opens file
    ofstream outClientFile( "clients.dat", ios::out );

    if ( !outClientFile ) { // overloaded ! operator
        cerr << "File could not be opened" << endl;
        exit( 1 ); // prototype in stdlib.h
    }
```

# Sequential file

```
cout << "Enter the account, name, and balance.\n"
      << "Enter end-of-file to end input.\n? ";
int account;
char name[ 30 ];
float balance;

while ( cin >> account >> name >> balance ) {
    outClientFile << account << ' ' << name
                  << ' ' << balance << '\n';
    cout << "? ";
}

return 0; // ofstream destructor closes file
}
```

# How to open a file in C++ ?

**Ofstream outClientFile("clients.dat", ios::out)**

**OR**

**Ofstream outClientFile;**

**outClientFile.open("clients.dat", ios::out)**

# File Open Modes

ios:: app - (append) write all output to the end of file

ios:: ate - data can be written anywhere in the file

ios:: binary - read/write data in binary format

ios:: in - (input) open a file for input

ios::out - (output) open a file for output

ios::trunc -(truncate) discard the files' contents if  
it exists

## File Open Modes contd.....

ios:nocreate - if the file does **NOT** exists, the open operation fails

ios:noreplace - if the file exists, the open operation fails

# How to close a file in C++?

The file is closed implicitly when a destructor for the corresponding object is called

OR

by using member function *close*:

**outClientFile.close();**

# Reading and printing a sequential file

```
// Reading and printing a sequential file
#include <iostream.h>
#include <fstream.h>
#include <iomanip.h>
#include <stdlib.h>
void outputLine( int, const char *, double );
int main()
{
    // ifstream constructor opens the file
    ifstream inClientFile( "clients.dat", ios::in );

    if ( !inClientFile ) {
        cerr << "File could not be opened\n";
        exit( 1 );
    }
```

```
int account;
char name[ 30 ];
double balance;

cout << setiosflags( ios::left ) << setw( 10 ) << "Account"
    << setw( 13 ) << "Name" << "Balance\n";

while ( inClientFile >> account >> name >> balance )
    outputLine( account, name, balance );

return 0; // ifstream destructor closes the file
}

void outputLine( int acct, const char *name, double bal )
{
    cout << setiosflags( ios::left ) << setw( 10 ) << acct
        << setw( 13 ) << name << setw( 7 ) << setprecision( 2 )
        << resetiosflags( ios::left )
        << setiosflags( ios::fixed | ios::showpoint )
        << bal << '\n';
}
```

## File position pointer

<iostream> and <ostream> classes provide member functions for repositioning the *file pointer* (the byte number of the next byte in the file to be read or to be written.)

These member functions are:

***seekg*** (seek get) for istream class

***seekp*** (seek put) for ostream class

# Examples of moving a file pointer

`inClientFile.seekg(0)` - repositions the file get pointer to the beginning of the file

`inClientFile.seekg(n, ios:beg)` - repositions the file get pointer to the n-th byte of the file

`inClientFile.seekg(m, ios:end)` -repositions the file get pointer to the m-th byte from the end of file

`nClientFile.seekg(0, ios:end)` - repositions the file get pointer to the end of the file

The same operations can be performed with `<ostream>` function member `seekp`.

# Updating a sequential file

Data that is formatted and written to a sequential file **cannot be modified easily** without the risk of destroying other data in the file.

If we want to modify a record of data, the new data may be longer than the old one and it could overwrite parts of the record following it.

## Problems with sequential files

Sequential files are inappropriate for so-called “instant access” applications in which a particular record of information must be located immediately.

These applications include banking systems, point-of-sale systems, airline reservation systems, (or any data-base system.)

# Random access files

Instant access is possible with random access files.

Individual records of a **random access file** can be accessed directly (and quickly) without searching many other records.

# Example of a Program that Creates a Random Access File

```
// Fig. 14.11: clntdata.h
// Definition of struct clientData used in
// Figs. 14.11, 14.12, 14.14 and 14.15.
#ifndef CLNTDATA_H
#define CLNTDATA_H
struct clientData {
    int accountNumber;
    char lastName[ 15 ];
    char firstName[ 10 ];
    float balance;
};
#endif
```

# Creating a random access file

```
// Creating a randomly accessed file sequentially
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>
#include "clntdata.h"
int main()
{
    ofstream outCredit( "credit1.dat", ios::out);
if ( !outCredit ) {
    cerr << "File could not be opened." << endl;
    exit( 1 );
}
```

```
clientData blankClient = { 0, "", "", 0.0 };

    for ( int i = 0; i < 100; i++ )
        outCredit.write
(reinterpret_cast<const char *>( &blankClient ),
     sizeof( clientData ) );
    return 0;
}
```

## <iostream> member function *write*

The <iostream> member function *write* outputs a fixed number of bytes beginning at a specific location in memory to the specific stream. When the stream is associated with a file, the data is written beginning at the location in the file specified by the “put” file pointer.

# Writing data randomly to a random file

```
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>
#include "clntdata.h"
int main()
{
    ofstream outCredit( "credit.dat", ios::ate );
    if ( !outCredit ) {
        cerr << "File could not be opened." << endl;
        exit( 1 );
    }
```

```
cout << "Enter account number "<< "(1 to 100, 0 to end  
input)\n? ";  
clientData client;  
cin >> client.accountNumber;  
  
while ( client.accountNumber > 0 &&  
client.accountNumber <= 100 )  
{  
    cout << "Enter lastname, firstname, balance\n? ";  
    cin >> client.lastName >> client.firstName>>  
client.balance;
```

```
outCredit.seekp( ( client.accountNumber - 1 ) *sizeof( clientData ) );
    outCredit.write(
        reinterpret_cast<const char *>( &client ),
        sizeof( clientData ) );
cout << "Enter account number\n? ";
cin >> client.accountNumber;
}

return 0;
}
```

# Reading data from a random file

```
#include <iostream.h>
#include <iomanip.h>
#include <fstream.h>
#include <stdlib.h>
#include "clntdata.h"
void outputLine( ostream&, const clientData & );
int main()
{
    ifstream inCredit( "credit.dat", ios::in );
    if ( !inCredit ) {
        cerr << "File could not be opened." << endl;
        exit( 1 );
    }
```

```
cout << setiosflags( ios::left ) << setw( 10 ) << "Account" << setw( 16 ) << "Last Name"
<< setw( 11 )
    << "First Name" << resetiosflags( ios::left ) << setw( 10 ) << "Balance" << endl;

clientData client;

inCredit.read( reinterpret_cast<char *>( &client ),
               sizeof( clientData ) );
```

```
while ( inCredit && !inCredit.eof() ) {  
    if ( client.accountNumber != 0 )  
        outputLine( cout, client );  
  
    inCredit.read( reinterpret_cast<char *>( &client ),  
                  sizeof( clientData ) );  
}  
  
return 0;  
}
```

```
void outputLine( ostream &output, const clientData &c )
{
    output << setiosflags( ios::left ) << setw( 10 )<< c.accountNumber <<
    setw( 16 ) << c.lastName
        << setw( 11 ) << c.firstName << setw( 10 )<< setprecision( 2 ) <<
    resetiosflags( ios::left )
        << setiosflags( ios::fixed | ios::showpoint )<< c.balance << '\n';
}
```

## The <iostream> function *read*

```
inCredit.read (reinterpret_cast<char *>(&client),sizeof(clientData));
```

The <iostream> function inputs a specified (by sizeof(clientData)) number of bytes from the current position of the specified stream into an object.

# **18CSC202J - OBJECT ORIENTED DESIGN AND PROGRAMMING**

## **Session 7**

**Topic : Classes and Errors**

# Outline

- What exceptions are and when to use them
- Using **try**, **catch** and **throw** to detect, handle and indicate exceptions, respectively
- To process uncaught and unexpected exceptions
- To declare new exception classes
- How stack unwinding enables exceptions not caught in one scope to be caught in another scope
- To handle new failures
- To understand the standard exception hierarchy

# Introduction

- Exceptions
  - Indicate problems that occur during a program's execution
  - Occur infrequently
- Exception handling
  - Can resolve exceptions
    - Allow a program to continue executing or
    - Notify the user of the problem and
    - Terminate the program in a controlled manner
  - Makes programs robust and fault-tolerant

# Exception Handling in C++

- A standard mechanism for processing errors
  - Especially important when working on a project with a large team of programmers
- *C++* exception handling is much like Java's
- *Java's* exception handling is much like *C++*

# Fundamental Philosophy

- Mechanism for sending an exception signal up the call stack
  - Regardless of intervening calls
- Note: there is a mechanism based on same philosophy in C
  - `setjmp()`, `longjmp()`
  - See man pages

# Traditional Exception Handling

- Intermixing program and error-handling logic
  - Pseudocode outline

*Perform a task*

*If the preceding task did not execute correctly*

*Perform error processing*

*Perform next task*

*If the preceding task did not execute correctly*

*Perform error processing*

*...*

- Makes the program difficult to read, modify, maintain and debug
- Impacts performance

Note:— In most large systems, code to handle errors and exceptions represents >80% of the total code of the system

# Fundamental Philosophy (continued)

- Remove error-handling code from the program execution's "main line"
- Programmers can handle any exceptions they choose
  - All exceptions
  - All exceptions of a certain type
  - All exceptions of a group of related types

# Fundamental Philosophy (continued)

- Programs can
  - Recover from exceptions
  - Hide exceptions
  - Pass exceptions up the “chain of command”
  - Ignore certain exceptions and let someone else handle them

# Fundamental Philosophy (continued)

- An *exception* is a class
  - Usually derived from one of the system's exception base classes
- If an exceptional or error situation occurs, program *throws* an object of that class
  - Object crawls up the call stack
- A calling program can choose to *catch* exceptions of certain classes
  - Take action based on the exception object

# Class exception

- The standard C++ base class for all exceptions
- Provides derived classes with virtual function **what**
  - Returns the exception's stored error message

# Example: Handling an Attempt to Divide by Zero

```
1 // Fig. 27.1: DivideByZeroException.h
2 // Class DivideByZeroException definition.
3 #include <stdexcept> // stdexcept header file contains runtime_error
4 using std::runtime_error; // standard C++ library class runtime_error
5
6 // DivideByZeroException objects should be thrown by functions
7 // upon detecting division-by-zero exceptions
8 class DivideByZeroException : public runtime_error
9 {
10 public:
11     // constructor specifies default error message
12     DivideByZeroException::DivideByZeroException()
13         : runtime_error( "attempted to divide by zero" ) {}
14 }; // end class DivideByZeroException
```

# Zero Divide Example

- Fig27-2
  - (1 of 2)

```
1 // Fig. 27.2: Fig27_02.cpp
2 // A simple exception-handling example that checks for
3 // divide-by-zero exceptions.
4 #include <iostream>
5 using std::cin;
6 using std::cout;
7 using std::endl;
8
9 #include "DivideByZeroException.h" // DivideByZeroException class
10
11 // perform division and throw DivideByZeroException object if
12 // divide-by-zero exception occurs
13 double quotient( int numerator, int denominator )
14 {
15     // throw DivideByZeroException if trying to divide by zero
16     if ( denominator == 0 )
17         throw DivideByZeroException(); // terminate function
18
19     // return division result
20     return static_cast< double >( numerator ) / denominator;
21 } // end function quotient
22
23 int main()
24 {
25     int number1; // user-specified numerator
26     int number2; // user-specified denominator
27     double result; // result of division
28
29     cout << "Enter two integers (end-of-file to end): ";
```

# Zero Divide Example

- Fig27-2
  - (2 of 2)

```
30
31 // enable user to enter two integers to divide
32 while ( cin >> number1 >> number2 )
33 {
34     // try block contains code that might throw exception
35     // and code that should not execute if an exception occurs
36     try
37     {
38         result = quotient( number1, number2 );
39         cout << "The quotient is: " << result << endl;
40     } // end try
41
42     // exception handler handles a divide-by-zero exception
43     catch ( DivideByZeroException &divideByZeroException )
44     {
45         cout << "Exception occurred: "
46             << divideByZeroException.what() << endl;
47     } // end catch
48
49     cout << "\nEnter two integers (end-of-file to end): ";
50 } // end while
51
52 cout << endl;
53 return 0; // terminate normally
54 } // end main
```

# try Blocks

- Keyword **try** followed by braces ({} )
- Should enclose
  - Statements that might cause exceptions
  - Statements that should be skipped in case of an exception

# Software Engineering Observation

- Exceptions may surface
  - through explicitly mentioned code in a `try` block,
  - through calls to other functions and
  - through deeply nested function calls initiated by code in a `try` block.

# Catch Handlers

- Immediately follow a **try** block
  - One or more **catch** handlers for each **try** block
- Keyword **catch**
- Exception parameter enclosed in parentheses
  - Represents the type of exception to process
  - Can provide an optional parameter name to interact with the caught exception object
- Executes if exception parameter type matches the exception thrown in the **try** block
  - Could be a base class of the thrown exception's class

# Catch Handlers (continued)

```
try {  
    // code to try  
}  
  
catch (exceptionClass1 &name1) {  
    // handle exceptions of exceptionClass1  
}  
  
catch (exceptionClass2 &name2) {  
    // handle exceptions of exceptionClass2  
}  
  
catch (exceptionClass3 &name3) {  
    // handle exceptions of exceptionClass3  
}  
  
...  
/* code to execute if  
   no exception or  
   catch handler handled exception*/
```

All other classes of exceptions  
are not handled here

**catch** clauses attempted  
in order; first match wins!

# Common Programming Errors

Syntax error to place code between a `try` block and its corresponding `catch` handlers

Each `catch` handler can have only a single parameter

- Specifying a comma-separated list of exception parameters is a syntax error
- Logic error to catch the same type in two different `catch` handlers following a single `try` block

# Fundamental Philosophy (continued)

- Termination model of exception handling
  - **try** block *expires* when an exception occurs
    - Local variables in try block go out of scope
  - Code within the matching catch handler executes
  - Control resumes with the first statement after the last catch handler following the try block
- Stack unwinding
  - Occurs if no matching **catch** handler is found
  - Program attempts to locate another enclosing **try** block in the calling function

*Control does not return to throw point*

# Stack Unwinding

- Occurs when a thrown exception is not caught *in a particular scope*
- *Unwinding a Function* terminates that function
  - All local variables of the function are destroyed
    - Invokes destructors
  - Control returns to point where function was invoked
- Attempts are made to catch the exception in outer **try...catch** blocks
- If the exception is never caught, the function **terminate** is called

# Observations

With exception handling, a program can continue executing (rather than terminating) after dealing with a problem.

This helps to support robust applications that contribute to *mission-critical* computing or *business-critical* computing

When no exceptions occur, there is no performance penalty

# Throwing an Exception

- Use keyword **throw** followed by an operand representing the type of exception
  - The **throw** operand can be of any type
  - If the **throw** operand is an object, it is called an **exception** object
- The **throw** operand initializes the exception parameter in the matching **catch** handler, if one is found

# Notes

- Catching an exception object by reference eliminates the overhead of copying the object that represents the **thrown** exception
- Associating each type of runtime error with an appropriately named exception object improves program clarity.

# When to Use Exception Handling

- To process synchronous errors
  - Occur when a statement executes
- Not to process asynchronous errors
  - Occur in parallel with, and independent of, program execution
- To process problems arising in predefined software elements
  - Such as predefined functions and classes
  - Error handling can be performed by the program code to be customized based on the application's needs

Don't use for routine stuff such as end-of-file or null string checking

# Software Engineering Notes

- Incorporate exception-handling strategy into system design from the start
  - Very difficult to retrofit after the system has been implemented!
- Exception handling provides uniform technique for processing problems
  - Helps with understandability of each other's error handling code
- Avoid using exception handling as an alternate form of flow of control
  - These “additional” exceptions can “get in the way” of genuine error handling

# Rethrowing an Exception

- Empty `throw;` statement
- Use when a `catch` handler cannot or can only partially process an exception
- Next enclosing `try` block attempts to match the exception with one of its `catch` handlers

# Common Programming Error

Executing an empty **throw** statement outside a **catch** handler causes a function call to terminate

- Abandons exception processing and terminates the program immediately

See D&D Fig 27.3

# Exception Specifications

- Also called **throw** lists
- Keyword **throw**
  - Comma-separated list of exception classes in parentheses
- Example
  - ```
int someFunction( double value )
    throw ( ExceptionA, ExceptionB,
            ExceptionC )
```

  
Optional!
  - Indicates **someFunction** can **throw** types **ExceptionA**, **ExceptionB** and **ExceptionC**

# Exception Specifications (continued)

- A function can **throw** only exceptions of types in its specification (or derived types)
  - If a function throws a non-specification exception, function **unexpected** is called
    - This normally terminates the program
- Absence of exception specification indicates that the function can **throw** any exception
- An empty exception specification, **throw()**, indicates the function *cannot throw* any exceptions

# Error Note

- The compiler will not generate a compilation error if a function contains a **throw** expression for an exception not listed in the function's exception specification.
- Error occurs only when that function attempts to **throw** that exception at run time.
- To avoid surprises at execution time, carefully check your code to ensure that functions do not **throw** exceptions not listed in their exception specifications

# Constructors and Destructors

- Exceptions and constructors
  - Exceptions enable constructors to report errors
    - Unable to return values
  - Exceptions thrown by constructors cause any already-constructed component objects to call their destructors
    - Only those objects that have already been constructed will be destructed
- Exceptions and destructors
  - Destructors are called for all automatic objects in the terminated **try** block when an exception is thrown
    - Acquired resources can be placed in local objects to automatically release the resources when an exception occurs
  - If a destructor invoked by stack unwinding throws an exception, function **terminate** is called

# Note

- When an exception is thrown from the constructor for an object that is created in a `new` expression, ...
- ... the dynamically allocated memory for that object is released.

# Exceptions and Inheritance

- New exception classes can be defined to inherit from existing exception classes
- A **catch** handler for a particular exception class can also catch exceptions of classes derived from that class
  - Enables **catching** related errors with a concise notation

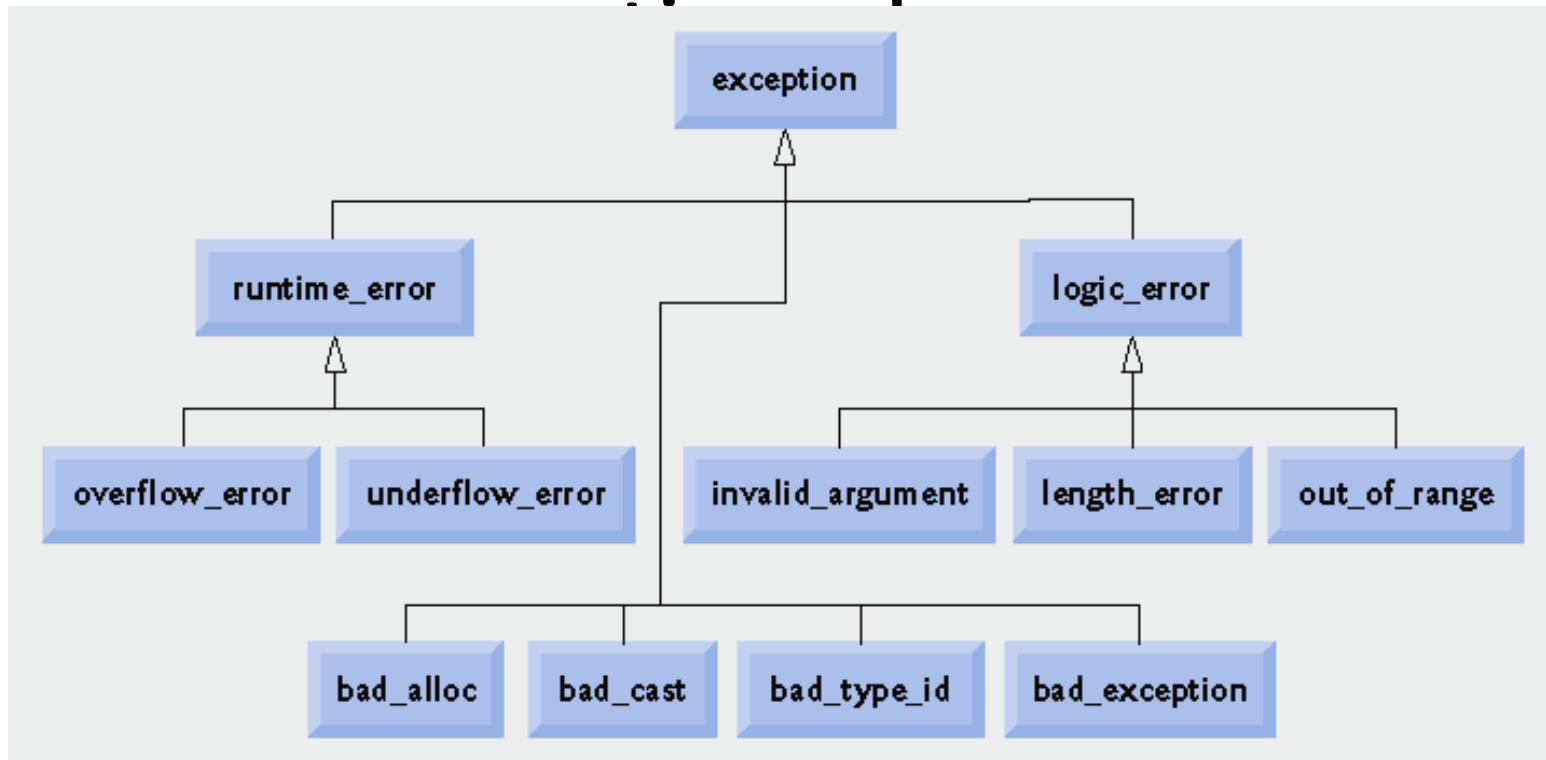
# Failure of calls to `new`

- Some compilers **throw** a `bad_alloc` exception
  - Compliant to the C++ standard specification
- Some compilers return 0
  - C++ standard-compliant compilers also have a version of `new` that returns 0
    - Use expression `new( noexcept )`, where `noexcept` is of type `noexcept_t`
- Some compilers **throw** `bad_alloc` if `<new>` is included

# Standard Library Exception Hierarchy

- Base-class **exception**
  - Contains **virtual** function **what** for storing error messages
- Exception classes derived from **exception**
  - **bad\_alloc** – thrown by **new**
  - **bad\_cast** – thrown by **dynamic\_cast**
  - **bad\_typeid** – thrown by **typeid**
  - **bad\_exception** – thrown by **unexpected**
    - Instead of terminating the program or calling the function specified by **set\_unexpected**
    - Used only if **bad\_exception** is in the function's **throw** list

# Fig. 27.11 | Standard Library



# Exception Handling Summary

- Exceptions are derived from class **exception**
- Exceptional or error condition is indicated by **throwing** an object of that class
  - Created by constructor in **throw** statement
- Calling programs can check for exceptions with **try...catch** construct
- Unified method of handling exceptions
  - Far superior to coding exception handling in long hand
- No performance impact when no exceptions