



Object Oriented Programming with C++

WELCOME TO THE COURSE

Robert Lafore

Object-Oriented Programming in C++

Fourth
Edition



Object-Oriented Programming in C++ (4th Edition)
4th Edition

By Robert Lafore

What IS Object Oriented Programming ?

- Object-oriented programming (OOP) is a programming paradigm based on the concept of "objects"
- A programming paradigm : is a style of programming, a way of thinking about software construction.
- A programming paradigm does not refer to a specific language but rather to a way to build a program or a methodology to apply.
- Some languages make it easy to write in some paradigms but not others.
- Some Programming Languages allow the programmer to apply more than one Paradigm.

Example of Programming Paradigms

MIS 315 - Bsharah

Programming Paradigms

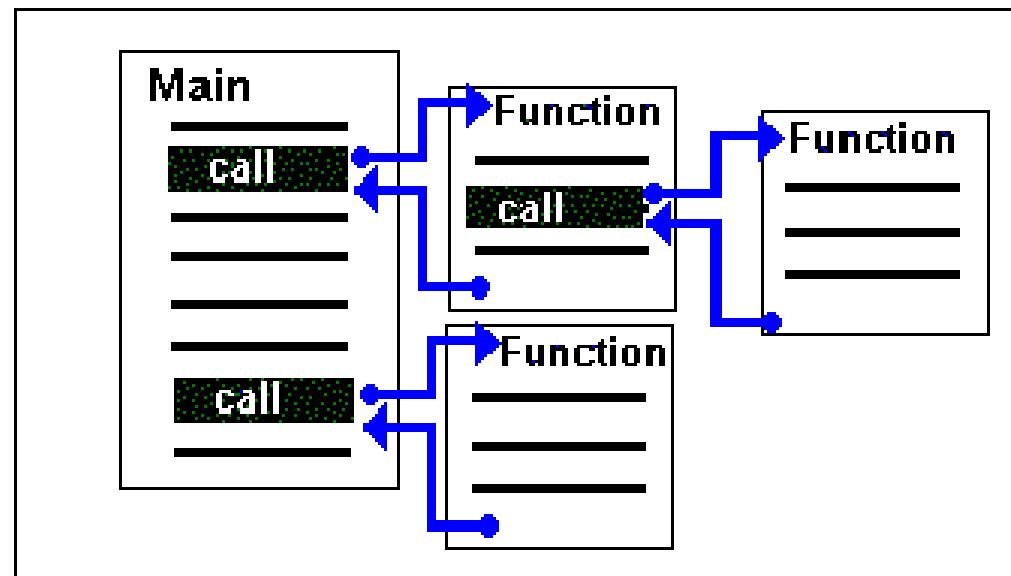
- The programming paradigm refers to a way of conceptualizing and structuring the tasks a computer performs.

Paradigm	Languages	Description
Procedural	BASIC, Pascal, COBOL, FORTRAN, Ada	Emphasizes linear steps that provide the computer with instructions on how to solve a problem or carry out a task
Object-oriented	Smalltalk, C++, Java	Formulates programs as a series of objects and methods that interact to perform a specific task
Declarative	Prolog	Focuses on the use of facts and rules to describe a problem
Functional	LISP, Scheme, Haskell	Emphasizes the evaluation of expressions, called functions
Event-driven	Visual Basic, C#	Focuses on selecting user interface elements and defining event-handling routines that are triggered by various mouse or keyboard activities

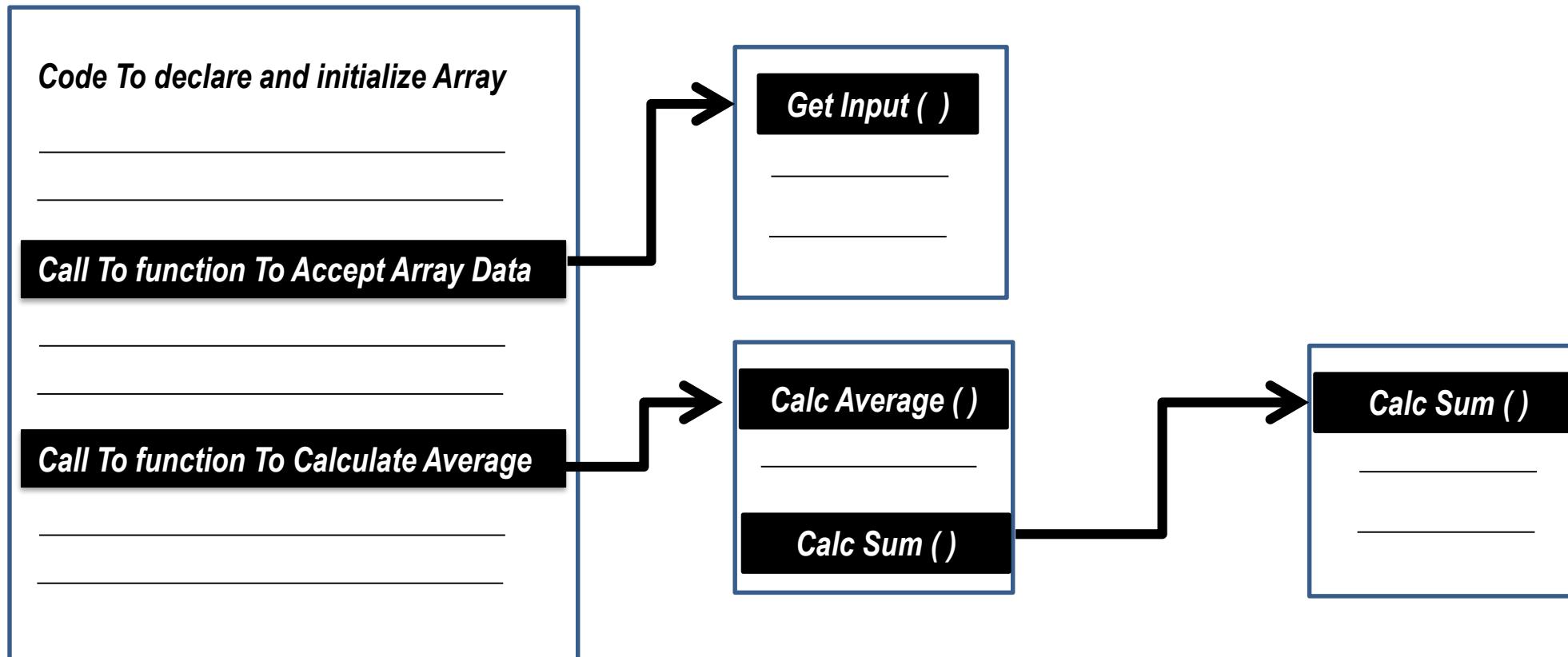
Example of Previous Programming Paradigm

Procedural Programming

Procedural programming (PP), also known as inline programming takes a top-down approach. It is about writing a list of instructions to tell the computer what to do step by step. It relies on procedures or routines.



Procedural Programming Example : Program to Calculate Average of Array Items



- Object-oriented programming (OOP) is a programming paradigm based on the concept of "objects"

Object : is a thing (Tangible – Intangible)



Objects in College Management Program

College Environment

Student

Course

Teacher

Section

Hall

Office

Objects in Super market Program

Super Market Environment

Product

Customer

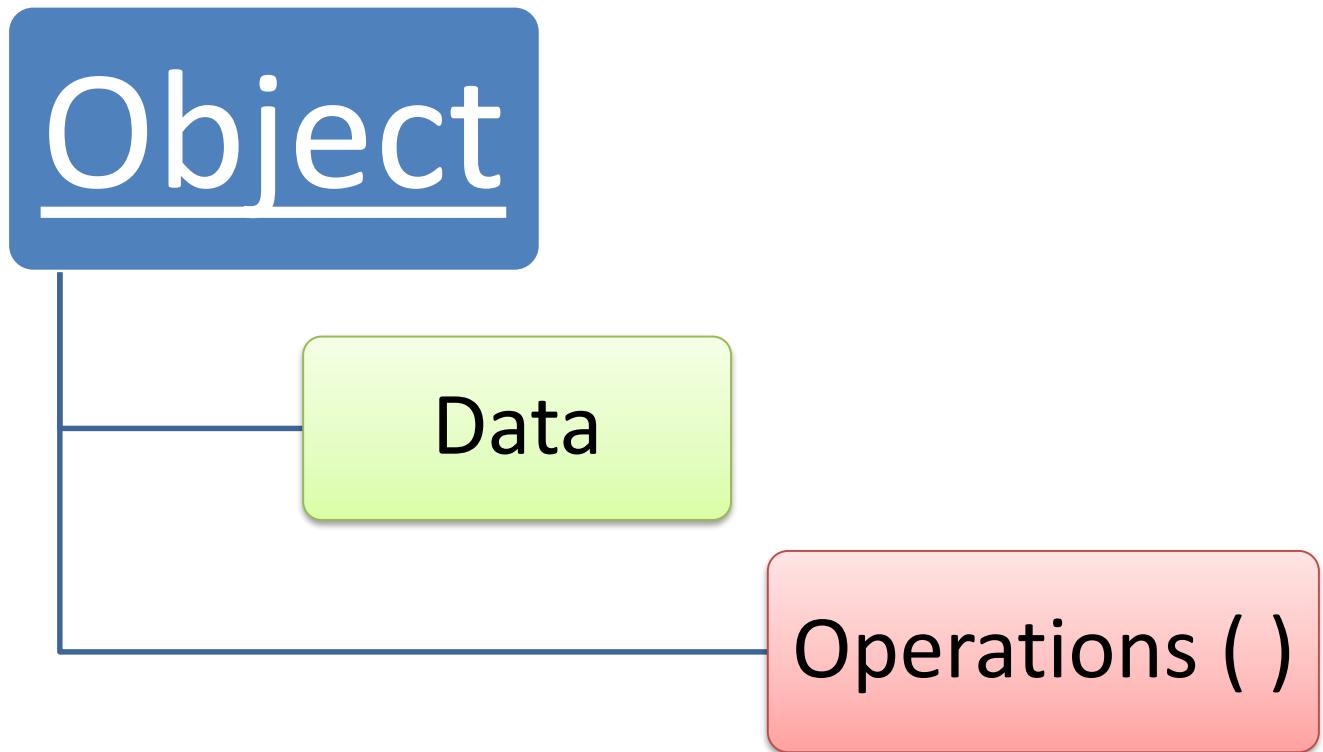
Cashier

Cart

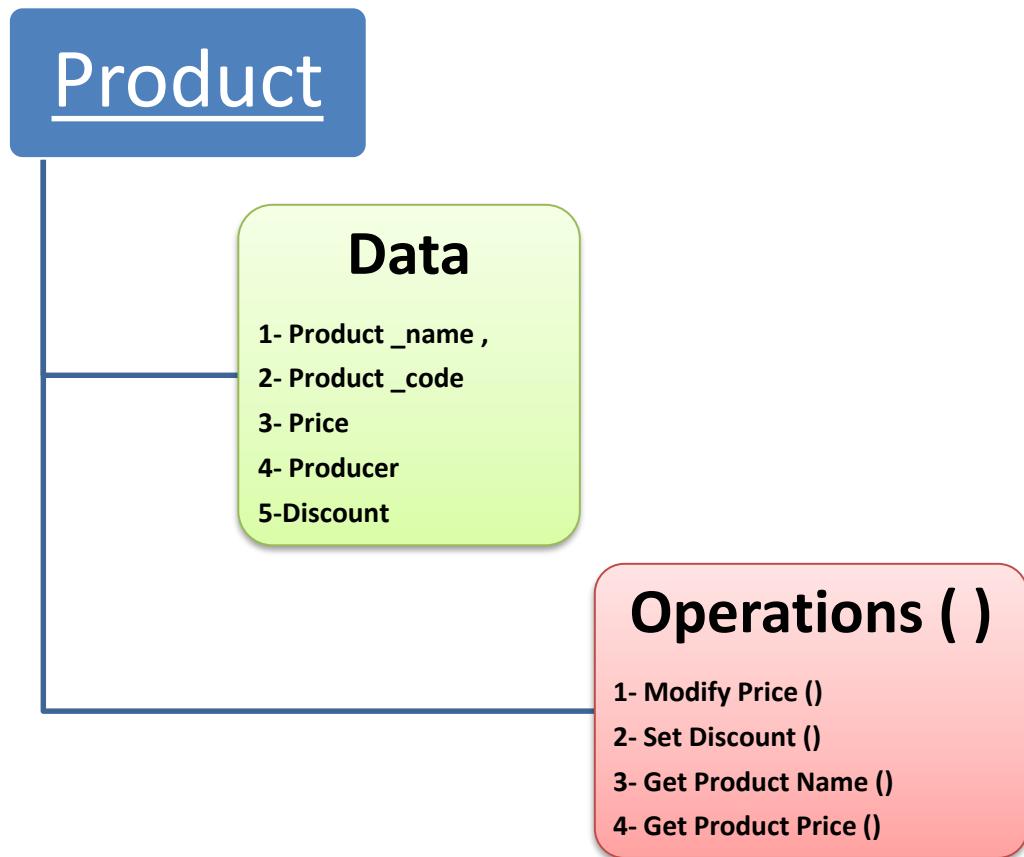
Bager

Loyalty
Card

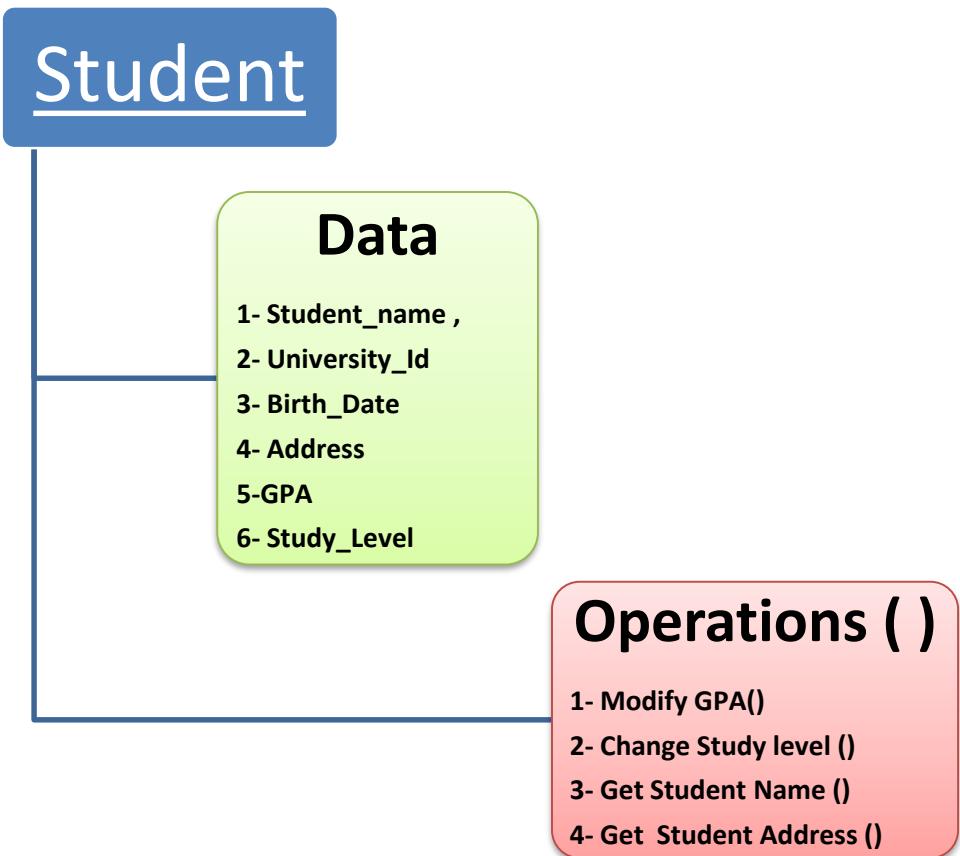
Object Is comprised Of ?



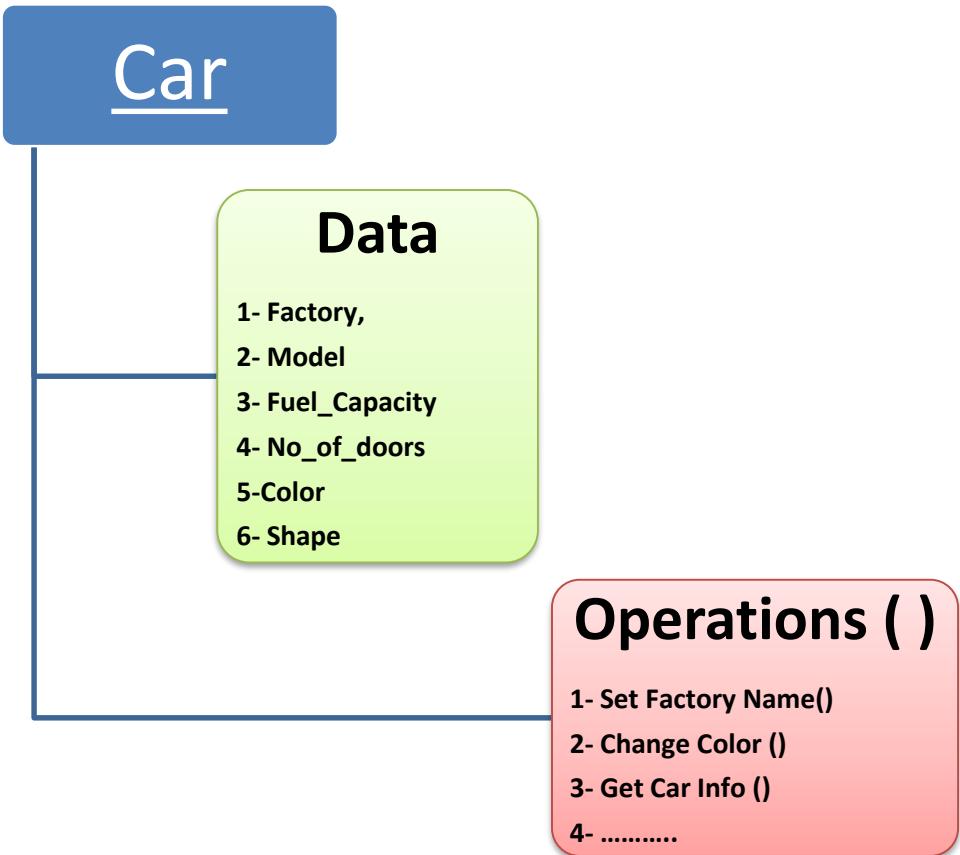
Object Is comprised Of ?



Object Is comprised Of ?



Object Is comprised Of ?



What is Class ? Why we need It ?

Student 1

Data:

- 1- Student_name ,
- 2- University_Id
- 3- Birth_Date
- 4- Address
- 5-GPA
- 6- Study_Level

Operations ()

- 1- Modify GPA()
- 2- Change Study level ()
- 3- Get Student Name ()
- 4- Get Student Address ()

Student 2

Data:

- 1- Student_name ,
 - 2- University_Id
 - 3- Birth_Date
 - 4- Address
 - 6- Study_Level
- GPA
not included

Operations ()

- 1- Modify GPA()
- 2- Change Study level ()
- 4- Get Student Address ()

Student 3

Data:

- 1- Student_name ,
 - 2- University_Id
 - 5-GPA
 - 6- Study_Level
- Birth Date
and
Address

Operations ()

- 1- Modify GPA()
- 2- Change Study level ()
- 3- Get Student Name ()
- 4- Get Student Address ()

are not included

Class is the Blueprint, Standard for attributes and Methods

What is Class ? Why we need It ?

cl || Same

Class Student

Data:

- 1- Student_name ,
- 2- University_Id
- 3- Birth_Date
- 4- Address
- 5-GPA
- 6- Study_Level

Operations ()

- 1- Modify GPA()
- 2- Change Study level ()
- 3- Get Student Name ()
- 4- Get Student Address ()

Student 1

Data:

- 1- Student_name
- 2- University_Id
- 3- Birth_Date
- 4- Address
- 5-GPA
- 6- Study_Level

Operations ()

- 1- Modify GPA()
- 2- Change Study level ()
- 3- Get Student Name ()
- 4- Get Student Address ()

Student 3

Data:

- 1- Student_name
- 2- University_Id
- 3- Birth_Date
- 4- Address
- 5-GPA
- 6- Study_Level

Operations ()

- 1- Modify GPA()
- 2- Change Study level ()
- 3- Get Student Name ()
- 4- Get Student Address ()

Student 2

Data:

- 1- Student_name
- 2- University_Id
- 3- Birth_Date
- 4- Address
- 5-GPA
- 6- Study_Level

Operations ()

- 1- Modify GPA()
- 2- Change Study level ()
- 3- Get Student Name ()
- 4- Get Student Address ()

What is Class ? Why we need It ?

Let's say we wanna add one more attribute to the class.

For example it's Email.

We would have to add it to the class, so it can be a stander for all the objects

Class Student

Data:

- 1- Student_name ,
- 2- University_Id
- 3- Birth_Date
- 4- Address
- 5-GPA
- 6- Study_Level

7- Email

Operations ()

- 1- Modify GPA()
- 2- Change Study level ()
- 3- Get Student Name ()
- 4- Get Student Address ()

5- Print Student Info ()

Student 1

Data:

- 1- Student_name
- 2- University_Id
- 3- Birth_Date
- 4- Address
- 5-GPA
- 6- Study_Level

7- Email

Operations ()

- 1- Modify GPA()
- 2- Change Study level ()
- 3- Get Student Name ()
- 4- Get Student Address ()

5- Print Student Info ()

Student 3

Data:

- 1- Student_name
- 2- University_Id
- 3- Birth_Date
- 4- Address
- 5-GPA
- 6- Study_Level

7- Email

Operations ()

- 1- Modify GPA()
- 2- Change Study level ()
- 3- Get Student Name ()
- 4- Get Student Address ()

5- Print Student Info ()

Student 2

Data:

- 1- Student_name
- 2- University_Id
- 3- Birth_Date
- 4- Address
- 5-GPA
- 6- Study_Level

7- Email

Operations ()

- 1- Modify GPA()
- 2- Change Study level ()
- 3- Get Student Name ()
- 4- Get Student Address ()

5- Print Student Info ()

What is Class ? Why we need It ?

Class Student

Data:

- 1- Student_name ,
- 2- University_Id
- 5-GPA
- 6- Study_Level

Operations ()

- 1- Modify GPA()
- 2- Change Study level ()
- 3- Get Student Name ()
- 4- Get Student GPA ()

Student 1

Data:

- 1- Student_name = Ahmed
- 2- University_Id = 1050
- 5-GPA = 3.75
- 6- Study_Level = 5

Operations ()

- 1- Modify GPA()
- 2- Change Study level ()
- 3- Get Student Name ()
- 4- Get Student GPA ()

Objects and Classes

- **Classes:** Where Objects Come From
 - A *class* is code that describes a particular type of object. It specifies the data that an object can hold (the object's attributes), and the actions that an object can perform (the object's methods).
 - You can think of a class as a code "blueprint" that can be used to create a particular type of object.

"Fields", "class variables", and "attributes" are more-or-less the same

Objects and Classes

- When a program is running, it can use the class to create, in memory, as many objects of a specific type as needed.
- Each object that is created from a class is called an *instance* of the class.



Classes & Objects

- A class is defined (declared) and used as follows:

```
class MyClass
{
    [private:]
        variables (data members)
        ...
        functions (methods)
        ...

    public:
        variables (data members)
        ...
        functions (methods)
        ...
};
```

```
void main()
{
    // define objects of type
    // class_name
    MyClass MyObject1;
    MyClass MyObject2;

    // call a member function
    MyObject1.func1(...);
    // assign value to data members
    MyObject1.Index = 12;
}
```

Classes & Objects

- The class CPoint represents a point in the 2D space...

```
class CPoint
{
    int x, y;

public:
    void Init()
    {
        x = 0;
        y = 0;
    }

    void Set (int ax, int ay)
    {
        x = ax;
        y = ay;
    }

    void Print()
    {
        cout<<"x = "<<m_x<<, y = "<<m_y<<endl;
    }
};
```

```
#include <iostream.h>

void main()
{
    CPoint p1, p2;

    p1.Init();
    p2.Set(4,6);

    p1.Print();
    p2.Print();
}
```

What is Class ? Why we need It ?

Class Student

Data:

- 1- Student_name ,
- 2- University_Id
- 3- Birth_Date
- 4- Address
- 5-GPA
- 6- Study_Level

7- Email

Operations ()

- 1- Modify GPA()
- 2- Change Study level ()
- 3- Get Student Name ()
- 4- Get Student Address ()

5- Print Student Info ()

Student 1

Data:

- 1- Student_name
- 2- University_Id
- 3- Birth_Date
- 4- Address
- 5-GPA
- 6- Study_Level

7- Email

Operations ()

- 1- Modify GPA()
- 2- Change Study level ()
- 3- Get Student Name ()
- 4- Get Student Address ()
- 5- Print Student Info ()

Student 3

Data:

- 1- Student_name
- 2- University_Id
- 3- Birth_Date
- 4- Address
- 5-GPA
- 6- Study_Level

7- Email

Operations ()

- 1- Modify GPA()
- 2- Change Study level ()
- 3- Get Student Name ()
- 4- Get Student Address ()
- 5- Print Student Info ()

Student 2

Data:

- 1- Student_name
- 2- University_Id
- 3- Birth_Date
- 4- Address
- 5-GPA
- 6- Study_Level

7- Email

Operations ()

- 1- Modify GPA()
- 2- Change Study level ()
- 3- Get Student Name ()
- 4- Get Student Address ()
- 5- Print Student Info ()

What is Class ? Why we need It ?

Student 1

Data:

- 1- Student_name ,
- 2- University_Id
- 5-GPA
- 6- Study_Level

Operations ()

- 1- Modify GPA()
- 2- Change Study level ()
- 3- Get Student Name ()
- 4- Get Student GPA ()

= Ahmed

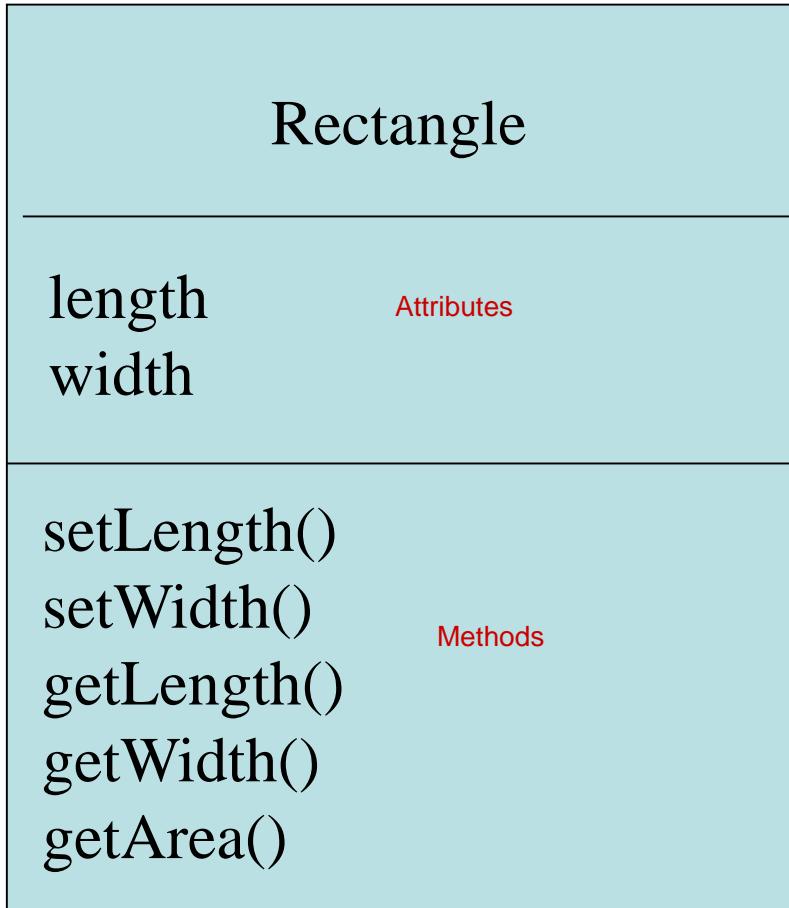
= 1050

=3.75

= 5

Writing a Class, Step by Step

- A Rectangle object will have the following fields:



Writing the Code

```
public class Rectangle  
{  
    private:  
        float length;  
        float width;  
}
```

Rectangle

length
width

setLength()
setWidth()
getLength()
getWidth()
getArea()

Access Modifiers

- An access modifier is a C++ keyword that indicates how an attribute or method can be accessed.
- **public**
 - When the `public` access modifier is applied to a class member, the member can be accessed by code inside the class or outside.
- **private**
 - When the `private` access modifier is applied to a class member, the member cannot be accessed by code outside the class.
 - The member can be accessed only by methods that are members of the same class.

Access Specifiers in C++

Specifier	Within Same Class	In Derived Class	Outside the Class
Private	Yes	No	No
Protected	Yes	Yes	No
Public	Yes	Yes	Yes

Data Hiding

- An object hides its internal, private attributes from code that is outside the class that the object is an instance of.
- Only the class's methods may directly access and change the object's internal data.
- Code outside the class must use the class's public methods to operate on an object's private fields.
- Data hiding is important because classes are typically used as components in large software systems, involving a team of programmers.
- Data hiding helps enforce the integrity of an object's internal data.

Access specifier
Public:

Return Type

Method Name

void setLength(float len)

Parameter variable declaration

```
graph TD; A[Access specifier] --> B[Public:]; C[Return Type] --> D[void setLength(float len)]; E[Method Name] --> F[setLength]; G[Parameter variable declaration] --> H[ ];
```

Rectangle
- width : float
- length : float
+ setWidth(w : float) : void
+ setLength(len : float) : void
+ getWidth() : float
+ getLength() : float
+ getArea() : float

```
public class Rectangle
{
    private:
        float length;
        float width;
```

Public:

```
void setLength(float len)
{
    If (len >=0)
        length = len;
    Else cout<<"Error , Please Enter positive value";
}
}
```

```
class Rectangle
{
private:
    float length;
    float width;

Public:
    void setLength(float len)
    {
        If (len >=0)
            length = len;
        Else cout <<"Error , Please Enter positive value";
    }
}
```

Rectangle

- width : float
- length : float

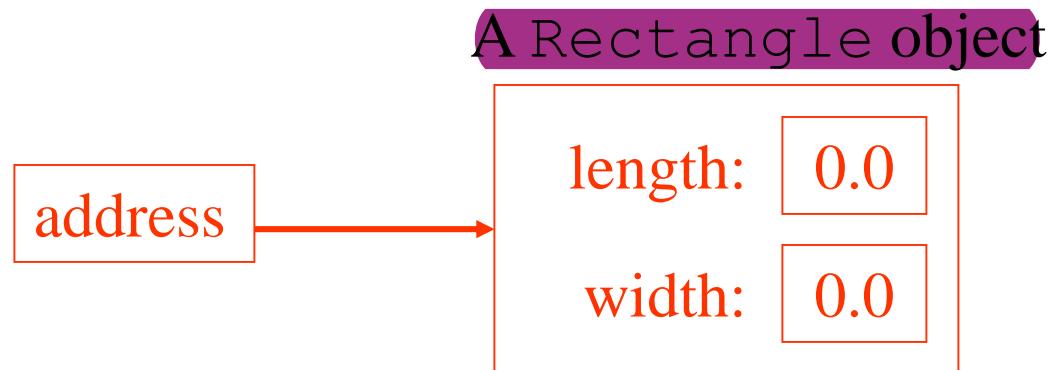
+ setWidth(w : float) : void
+ setLength(len : float): void
+ getWidth() : float
+ getLength() : float
+ getArea() : float

Creating a Rectangle object

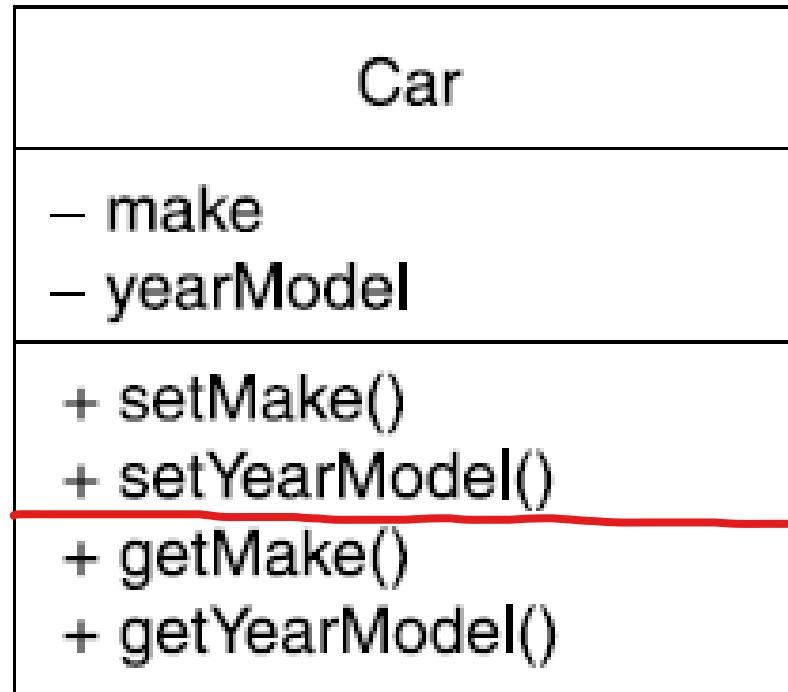
Rectangle box;

Int x; String name;

The **box** variable holds the address of the Rectangle object.



More Examples



Setter and Mutator are the same and used interchangeably.

Setter , Mutator

Getter, Accessor

Getter and Accessor are the same and used interchangeably.

Separating Class Code into 2 files.

The class code can be separated into 2 files:

Not necessary but advised.

Header File - .h

- Contains the declaration of all the class members.
- Only attributes declaration and methods prototypes

Implementation File - .cpp

- Contains the implementation of the class methods.
initialization

Client Code

- client code, is the one that includes the main function. This file should be stored by the name main.cpp

```
public class Rectangle
{
private:
    float width;
    float length;

public :
    void setWidth(float w)
    {
        width = w;
    }
    void setLength(float len)
    {
        length = len;
    }
    float getWidth()
    {
        return width;
    }
    float getLength()
    {
        return length;
    }

    float getArea()
    {
        return length * width;
    }
}
```

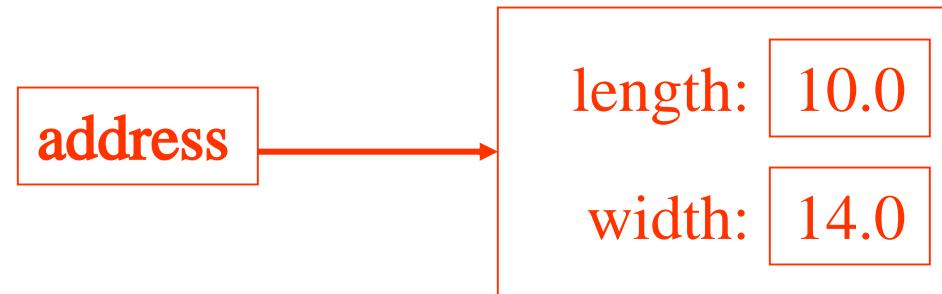
Instance attributes and Methods

- **Instance attributes** and **instance methods** require an object to be created in order to be used.
- For example, every room can have different dimensions.

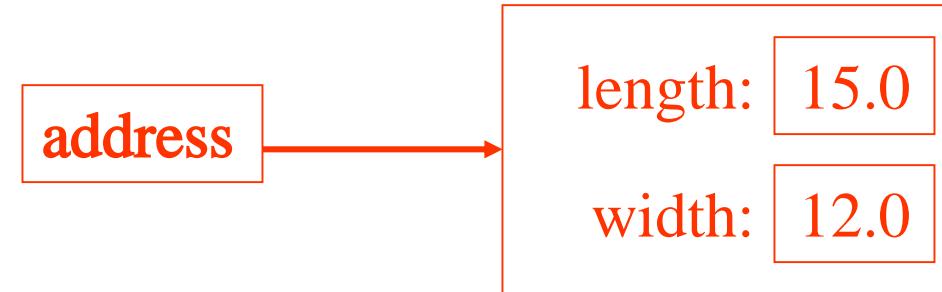
```
Rectangle    kitchen    =    new    Rectangle();  
Rectangle    bedroom    =    new    Rectangle();  
Rectangle    den        =    new    Rectangle();
```

States of Three Different Rectangle Objects

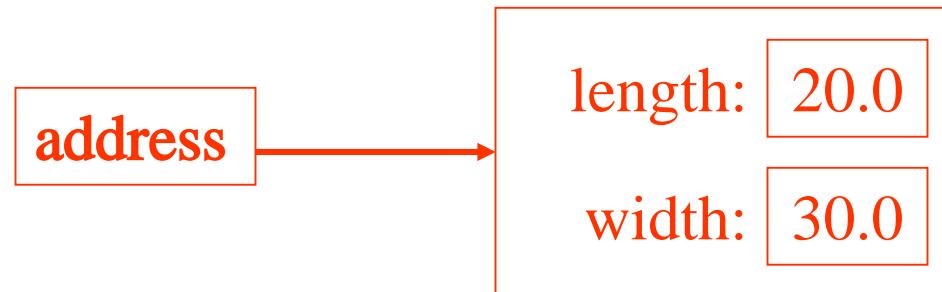
The `kitchen` variable holds the address of a Rectangle Object.



The `bedroom` variable holds the address of a Rectangle Object.



The `den` variable holds the address of a Rectangle Object.



Constructors

- Classes can have special methods called *constructors*.
- A constructor is a method that is automatically called when an object is created.

Recatngle r1; ----- Car c1;

- Constructors typically initialize object attributes and perform other object initialization tasks.
- Constructors are used to perform operations at the time an object is created.

A constructor is a special method that is automatically called when an object of a class is created.

Constructors

- Constructors have a few special properties that set them apart from normal methods.
 - Constructors have the same name as the class.
 - Constructors have no return type (not even `void`).
 - Constructors may not return any values.
 - Constructors are typically public.

```
public : Constructor  
        Rectangle( )  
{  
    length = 0;  
    width = 0;  
}
```

Constructors – Initialization list

```
public :  
    Rectangle( ): length(0), width (0)  
{  
    Cout <<"The Rectangle Length and width are initialized";  
}
```

```
public :  
    Rectangle(float len, float w)  
{  
    length = len;  
    width = w;  
}
```

```
public :  
    Rectangle(float len, float ):  
        length(len),width(w)  
{  
}
```

Destructor

- A destructor is a special method that is automatically called when an object life time is ended.
- Like constructors, destructors do not have a return value.
- The most common use of destructors is to deallocate memory that was allocated for the object by the constructor

```
public :  
    ~Rectangle( )  
{  
}  
}
```

Overloading Methods and Constructors

Overloading means different methods with same name

- Two or more methods in a class may have the same name as long as their signatures are different.
- Method signature (No of Args – Types of Args – Order of Args)
- When this occurs, it is called *method overloading*. This also applies to constructors.
- Method overloading is important because sometimes you need several different ways to perform the same operation.

Constructor Overloading

- ❖ Constructors can be overloaded:
 - An overloaded constructor provides multiple ways to set up a new object
 - The overloaded constructors differ by the number and type of parameters they get.
- ❖ When we construct an object, the compiler decides which constructor to invoke according to the type of the actual parameters
- ❖ A constructor with no parameters is called a *default constructor*

Constructor Overloading

In addition to overloading normal methods, you can also overload constructor methods.

Constructor overloading: Same constructor declared with different parameters in the same class.

Compiler differentiates which constructor is to be called depending upon the **number of parameters and their sequence of data types**.

Overloaded Method add

2 int parameters

```
int add(int num1, int num2)
{
    int sum = num1 + num2;
    return sum;
}
```

3 int parameters

```
int add(int num1, int num2, int num3)
{
    int sum = num1 + num2 + num3 ;
    return sum;
}
```

2 float parameters

```
Float add(float num1, float num2)
{
    float sum = num1 + num2;
    return sum;
}
```

Constructor Overloading

```
Rectangle::Rectangle () :length(0),width(0)
```

```
{  
}
```

```
Rectangle::Rectangle(float l , float w) :length(l),width(w)
```

```
{  
}
```

Rectangle Class Constructor Overload

```
Rectangle box1();
```

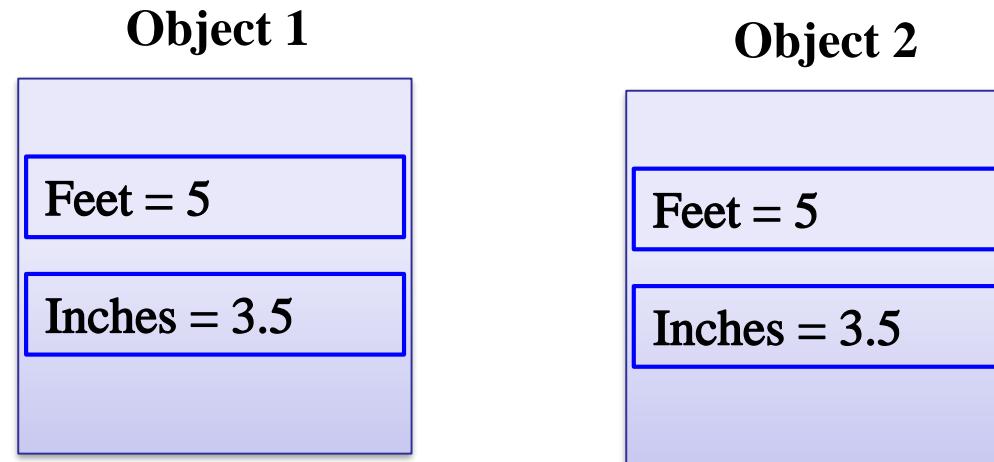
```
Rectangle box2(5.0, 10.0);
```

The Default Copy Constructor

Vid #11

- It is another way to initialize an object:
- Used to initialize an object with *another object of the same type*.
- No need to create a special constructor for this; one is already built into all classes

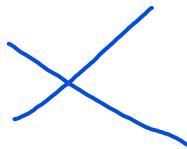
Class : Distance
- Feet - Inches
Distance () ~Distance ()



The Default Copy Constructor

```
class Distance { //English Distance class  
private:  
  
int feet;  
float inches;  
  
public:  
Distance():feet(0),inches(0.0)    Constructors  
{}  
Distance(int ft, float in):feet(ft),inches(in){ }  
}
```

```
int main() {  
Distance d1; default Constructor  
Distance dist2(11, 6.25);  
Distance dist3(dist2); copy of dist2 object  
Distance dist4 = dist2; same but another way of writing
```



The Default Constructor

- When an object is created, its constructor is always called.
- If you do not write a constructor, C++ provides one when the class is compiled.
The constructor that C++ provides is known as the *default constructor*.

Default Constructor

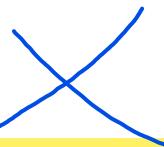
- A default constructor is a constructor that is used to create an object when you don't provide initialization values.
 - Time t1; //calling default constructor
- What if we do not write any constructor?
 - C++ automatically creates a **default constructor** for each class.
 - It creates an uninitialized object.
- For the Time class the default constructor looks like this:
 - Time::Time() { }

Important Thing about Default Constructor

- Compiler creates a default constructor only if you don't define any constructor.
- If you define any constructor, you need to define the default constructor too.
- If your class has non-default constructor but no default constructor then a declaration like this becomes an error.
 - Time t;

Define Default Constructor

- There are two ways to define a default constructor:
 - Define a constructor with no arguments
 - Time();
 - Provide default values for all arguments
 - Time(int h=12, int m=0, int s=0);
- There can be only one default constructor, so don't use both in a class.



The Default Constructor

- The default constructor is a constructor with no parameters, used to initialize an object in a default configuration.
- The only time that Java provides a default constructor is when you do not write any constructor for a class.
- A default constructor is not provided by Java if a constructor is already written.

Writing Your Own No-Arg Constructor

- A constructor that does not accept arguments is known as a *no-arg constructor*.
- The default constructor (provided by Java) is a no-arg constructor.
- We can write our own no-arg constructor

```
public Rectangle()  
{  
    length = 1.0;  
    width = 1.0;  
}
```

Passing Objects as Arguments

~~#12~~

- Objects can be passed to methods as arguments.
- When an object is passed as an argument, the value of the reference variable is passed.
- The value of the reference variable is an address or reference to the object in memory.
- A *copy* of the object is *not passed*, just a pointer to the object.
- When a method receives a reference variable as an argument, it is possible for the method to modify the contents of the object referenced by the variable.

```
Class Calculator
{
    Float add(float num1, float num2) Method
    {
        return num1 + num2;
    }
}
```

```
string add(string a, string b) Method
{
    return a + " " + b;
}
}
```

```
Distance Add_distances (Distance d1 , Distance d2)
{
}
```

```
Rectangle Merge (Rectangle r1 , Rectangle r2)
{
}
```

```
Main()
{
    Calculator calc;
    Float x = 50.0 ;
    Float Y = 10.0;

    Calc.add (x , y);
}
```

object

Class : Distance

- Feet
- Inches

Distance ()

Distance Add_distance(Distance d2)

the same obj datatype

~Distance ()

```
Distance Distance::add_distance(Distance d2)
{
    Distance result;
    result.feet = feet + d2.feet;
    result.inches=inches + d2.inches;
    return result;
}
```

adding to objects together

d1

Feet = 5

Inches = 3.5

d2

Feet = 3

Inches = 4.25

Result.feet = d1.feet + d2.feet

Result.inches = d1.inches + d2.inches

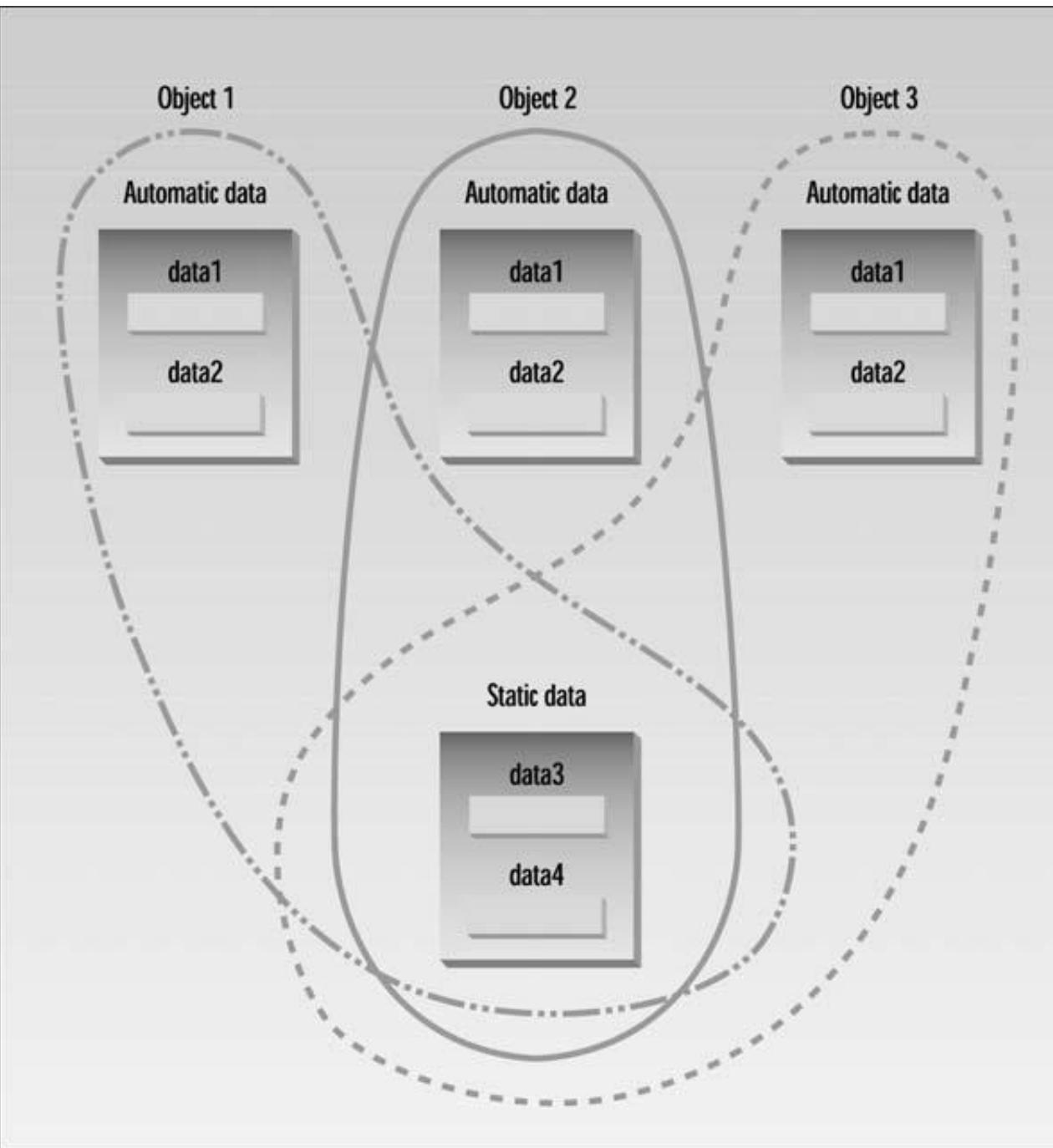
Feet = 8

Inches = 7.75

Result

Static Class Members

- *Static fields* and *static methods* do not belong to a single instance of a class.
- A static data item is useful when all objects of the same class must share a common item of information.
- Its lifetime is the entire program. It continues to exist even if there are no objects of the class.
- To invoke a static method or a static field, use the class name, rather than the instance name.

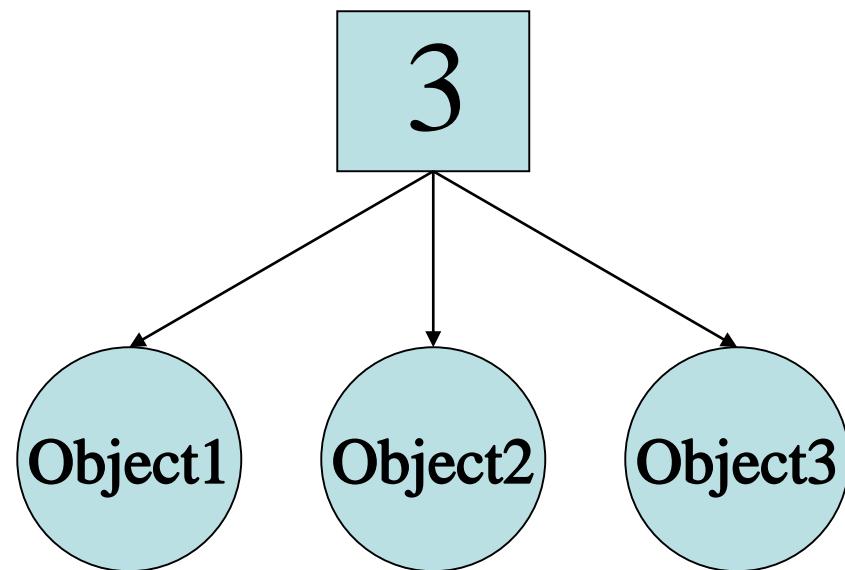


```
class Car{  
  
    string Maker ;  
    int model ;  
    static int count;  
    public:  
        Car( ) //increments count when object created  
        { count++; }  
        int getcount( ) //returns count  
        { return count; }  
};
```

```
int Car::count = 0;  
  
int main(){  
    Car c1, c2, c3; //create three objects  
    cout << "count is " << c1.getcount() << endl;  
    cout << "count is " << c2.getcount() << endl;  
    cout << "count is " << c3.getcount() << endl;  
    return 0;  
}
```

Static Fields

instanceCount field
(static)



Static Methods

- Static methods are convenient because they may be called at the class level.
- They are typically used to create utility classes.
- Static methods may not communicate with instance fields, only static fields.

```
Class Calc{  
    Public:  
  
        Static int add(int num1 , int num2){  
            return num1 + num2;  
        }  
        Static int multiply (int num1 , int num2){  
            return num1 * num2;  
        }  
};
```

Operators in C++

Unary operator



Operator

Type

`++`, `--`

Unary operator

`+, -, *, /, %`

Arithmetic operator

`<, <=, >, >=, ==, !=`

Relational operator

`&&, ||, !`

Logical operator

`&, |, <<, >>, ~, ^`

Bitwise operator

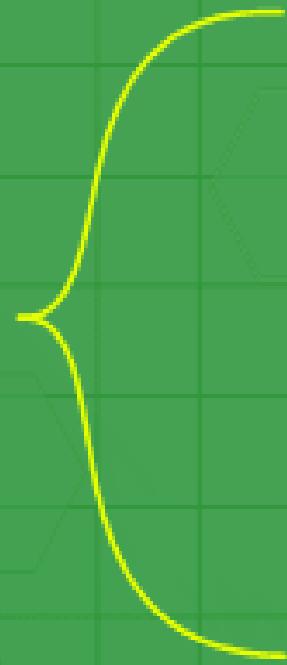
`=, +=, -=, *=, /=, %=`

Assignment operator

`?:`

Ternary or
conditional operator

Binary operator



Ternary operator



Operator overloading

such as + , * , <= , +=

- The term *operator overloading* refers to giving the normal C++ operators additional meanings when they are applied to user-defined data types.
- Operator overloading is one of the most exciting features of object-oriented programming.
- It can transform complex program listings into easy ones.

```
int a , b, c;
```

```
c= a + b ;
```

```
Counter c1, c2 , c3;
```

```
c3 = c1+ c2;
```

The **operator** Keyword is used to overload operators

Overloading Unary Operators

```
class Counter
{
private:
    unsigned int count;
public:
    Counter() : count(0)
    {
    }
    Counter(int c) : count(c)
    {
    }
    unsigned int get_count()
    {
        return count;
    }
    Counter operator ++ ()
    {
        ++count;
        return Counter(count);
    }
};
```

```
int main()
{
    Counter c1, c2;
    cout << c1.get_count();
    cout << c2.get_count();

    //Operator Overloading

    ++c1; //increment c1
    ++c2; //increment c2
    ++c2; //increment c2

    cout << c1.get_count();
    cout << c2.get_count();
```

```
class Counter
{
private:
    unsigned int count;
public:
    Counter() : count(0)
    {
    }
    unsigned int get_count()
    { return count; }
    Counter operator ++ ()
    {
        ++count;
        Counter temp;
        temp.count = count;
        return temp;
    }
};
```

```
class Counter
{
private:
    unsigned int count;
public:
    Counter() : count(0)
    {
    }
    Counter(int c) : count(c)
    {
    }
    unsigned int get_count()
    { return count; }
    Counter operator ++ ()
    {
        ++count;
        return Counter(count);
    }
};
```

Overloading Unary Operators – PostFix Notation

```
class Counter
{
private:
    unsigned int count;           //count
public:
    Counter() : count(0)         //constructor no args
    {
    }
    Counter(int c) : count(c)   //constructor, one arg
    {
    }
    unsigned int get_count() const //return count
    {
        return count;
    }

    Counter operator ++ ()      //increment count (prefix)
    {
        //increment count, then return
        return Counter(++count); //an unnamed temporary object
    }
    //initialized to this count

    Counter operator ++ (int)   //increment count (postfix)
    {
        //return an unnamed temporary
        return Counter(count++); //object initialized to this
    }
    //count, then increment count
};
```

```
int main()
{
    Counter c1, c2;

    cout << "\nc1=" << c1.get_count();
    cout << "\nc2=" << c2.get_count();

    ++c1;
    c2 = ++c1;

    cout << "\nc1=" << c1.get_count();
    cout << "\nc2=" << c2.get_count();

    c2 = c1++;
}
```

Overloading Binary Operators



Distance d_1, d_2, d_3 ;
 $d_3 = d_1 + d_2$

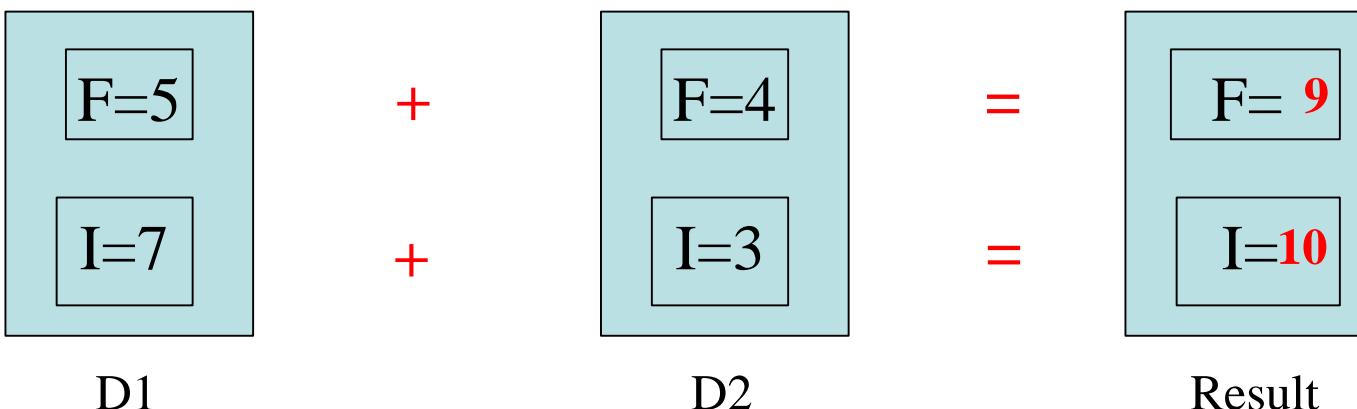


Counter c_1, c_2, c_3 ;
 $c_3 = c_1 + c_2$



Overloading Binary Operators

```
Distance Distance::operator + (Distance d2) const //return sum
{
    int f = feet + d2.feet;                      //add the feet
    float i = inches + d2.inches;                 //add the inches
    if(i >= 12.0)                                //if total exceeds 12.0,
    {
        i -= 12.0;                               //then decrease inches
        f++;                                    //by 12.0 and
    }                                              //increase feet by 1
    return Distance(f,i);                         //return a temporary Distance
}
```



What is Inheritance?

- Inheritance is probably the most powerful feature of object-oriented programming, after classes themselves.
- Inheritance is the process of creating new classes, called *derived classes*, from existing or *base classes*.
- The derived class inherits all the capabilities of the base class but can add its own features. And the base class is unchanged by this process.

Inheritance is a relationship between two or more classes where derived class inherits properties of pre-existing (base) classes.

The mechanism of deriving a new class from an old class is called inheritance.

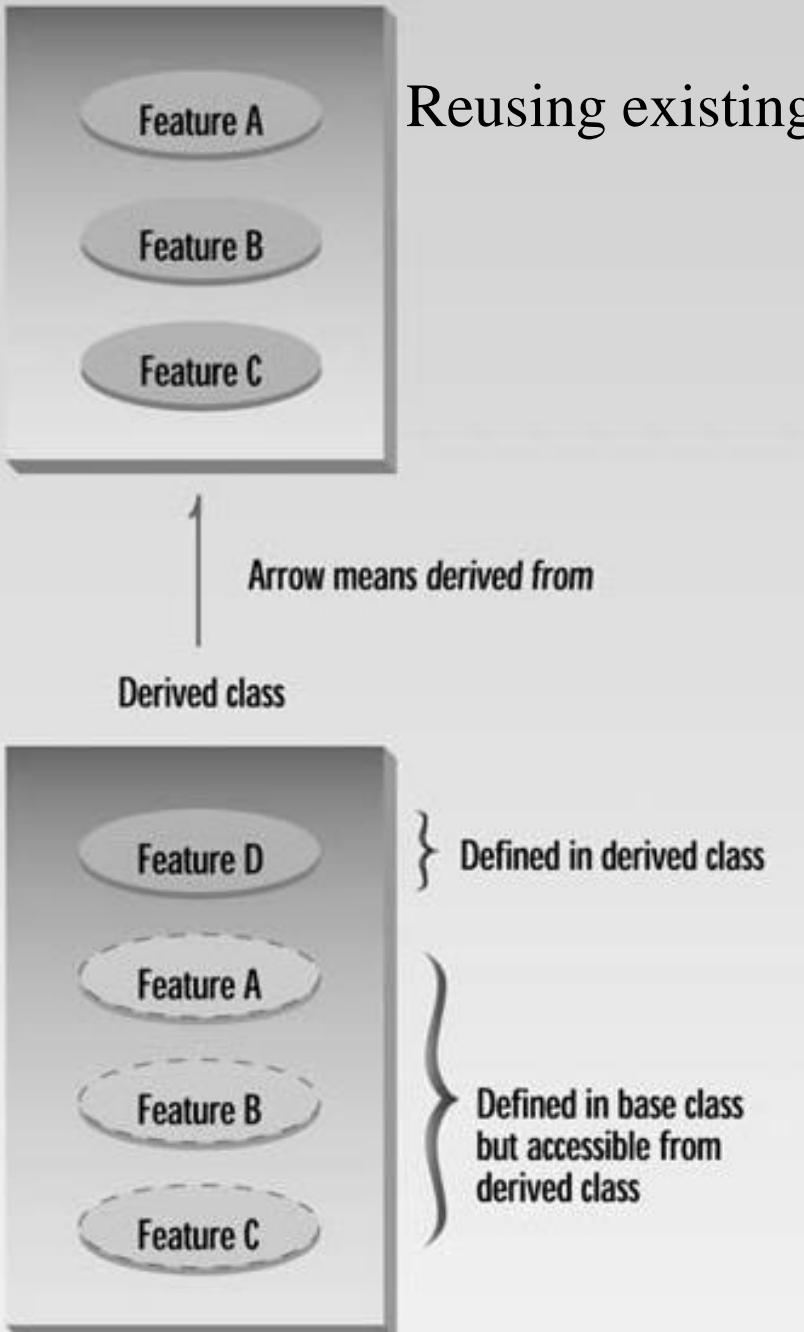
The old class is referred as base class and new one is called as derived class.

A new class inherits some or all properties from the base class.

Inheritance enables reusability.

Base class

Inheritance permits code *reusability*.



Reusing existing code saves time and money and increases a program's reliability.

```
class Counter //base class
{
protected: //NOTE: not private
int count;
public:
Counter() : count(0)
{ }
Counter(int c) : count(c)
{ }
int get_count()
{ return count; }
Counter operator ++ ()
{
    return Counter(++count);
}
};
```



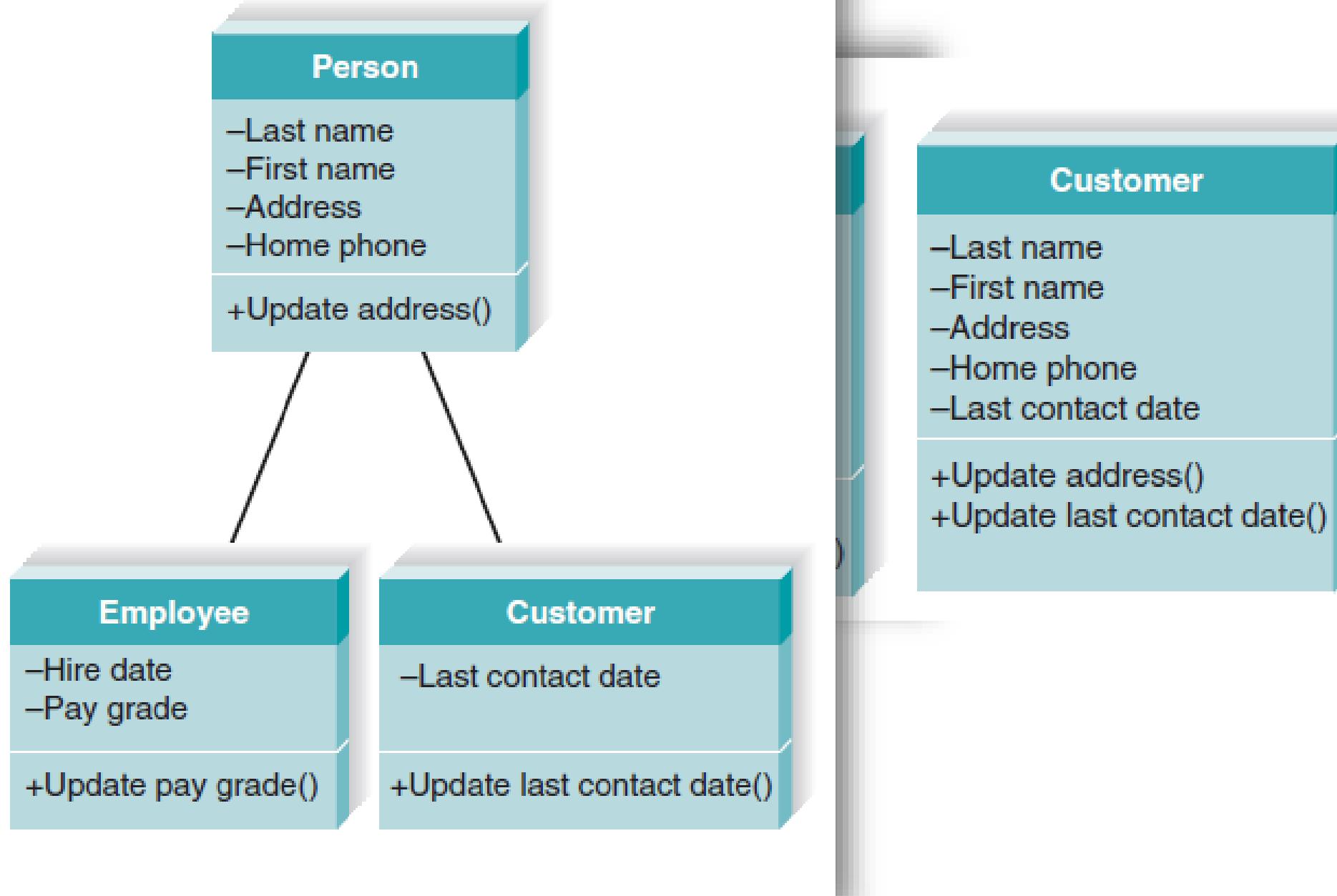
```
int main()
{
CountDn c1; //c1 of class CountDn

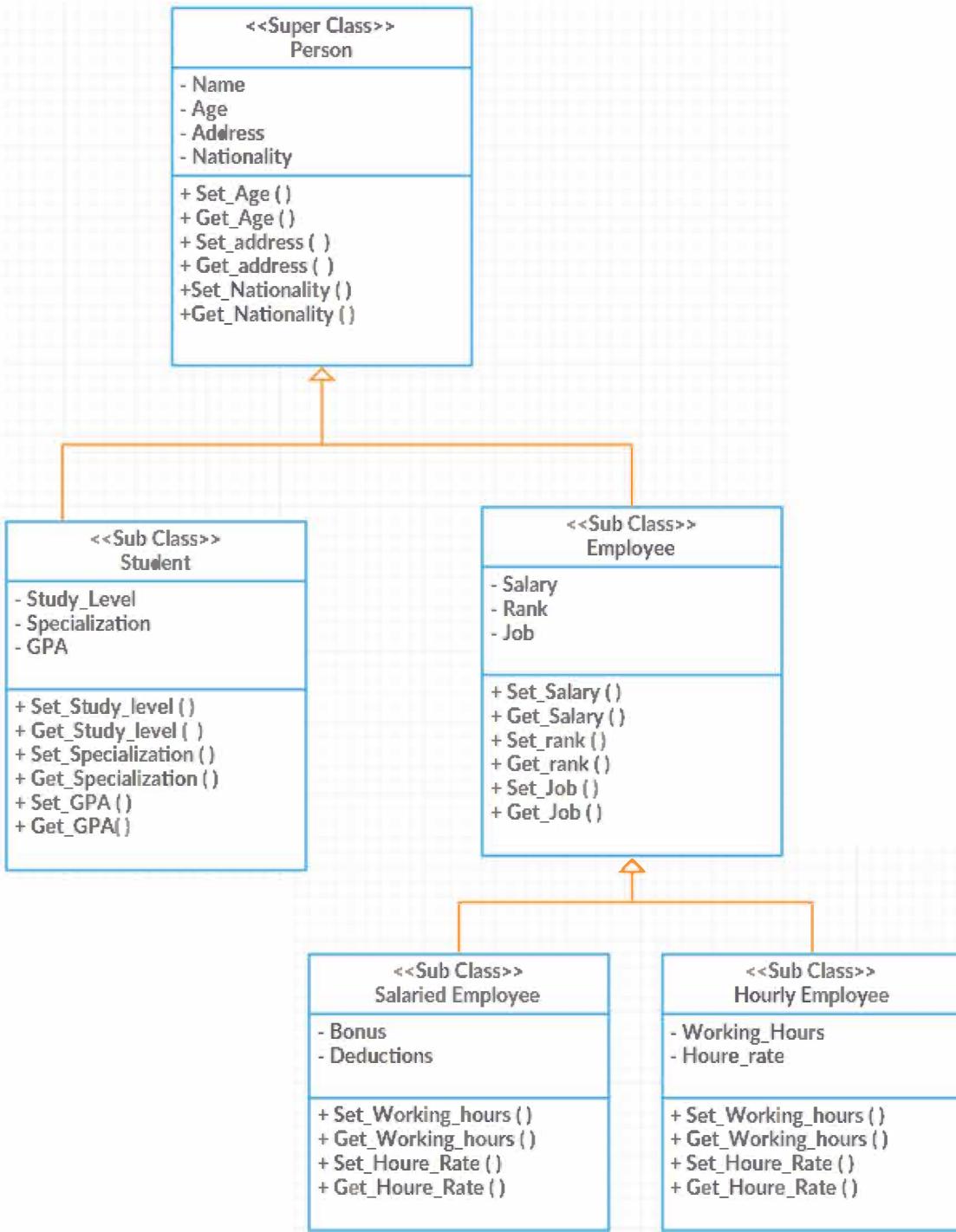
++c1; ++c1; ++c1; //increment c1, 3 times

--c1; --c1; //decrement c1, twice
}
```

```
class CountDn : public Counter //derived class
{
public:
Counter operator -- ()
{
    return Counter(--count);
}
};
```

With Inheritance





The “is a” Relationship

- The relationship between a **Base Class** and **a derived class** is called an **“is a” relationship**.
 - A post graduate student **“is a”** Student.
 - An Employee **“is a”** Person.
 - Salaried Employee **“is a”** Employee.
 - A car **“is a”** vehicle.
- A **specialized object** has:
 - All of the characteristics of the general object, plus additional characteristics that make it special.
- In object-oriented programming, **inheritance** is used to create an **“is a”** relationship among classes.

Base Class: It is the class whose properties are inherited by another class. It is also called Super Class or Parent Class.

Derived Class: It is the class that inherit properties from base class(es). It is also called Sub Class or Child Class.

```
class Person
{
private:
    string name;
    string gender;
    float age;

public:
    Person() { ... }
    Person(string n, string g, long s, float a)
    void set_age(float a) { ... }
    float get_age() { ... }
    void set_name(string n) { ... }
    string get_name() { ... }
    void set_gender(string g) { ... }
    string get_gender() { ... }
    void display() { ... }
};
```

```
class Student : public Person
{
};
```

#Modes of inheritance

#Public mode

If we derive a child class from a public parent class. Then the public member of the parent class becomes a public member for the child class and protected members of parent class becomes protected members of the child class.

#Protected mode

If we derive child class from a protected base class, then the public, as well as a protected member of the parent class, becomes the protected members of the child class.

#Private mode

If we derive a child class from a private base class, then the public, as well as protected members, become private for the derived class.

Private members of a base class cannot be directly accessed in the derived class in any circumstance.

Visibility Mode

The visibility mode specifies how the features of the base class are visible to the derived class.

Private : When a derived class privately inherits a base class, the protected and public members of base class become private members of the derived class.

Public : In Public mode, the protected and public members of base class become protected and public members of derived class respectively.

Protected : In Protected mode, the protected and public members of base class become protected members of the derived class.

Order of Constructor Call with Inheritance in C++

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

Function Overriding

- It is the redefinition of base class function in its derived class with same signature.

```
Class a
{
public:
    virtual void display(){ cout << "hello"; }
}

Class b:public a
{
public:
    void display(){ cout << "bye"; }
}
```

Function Overloading

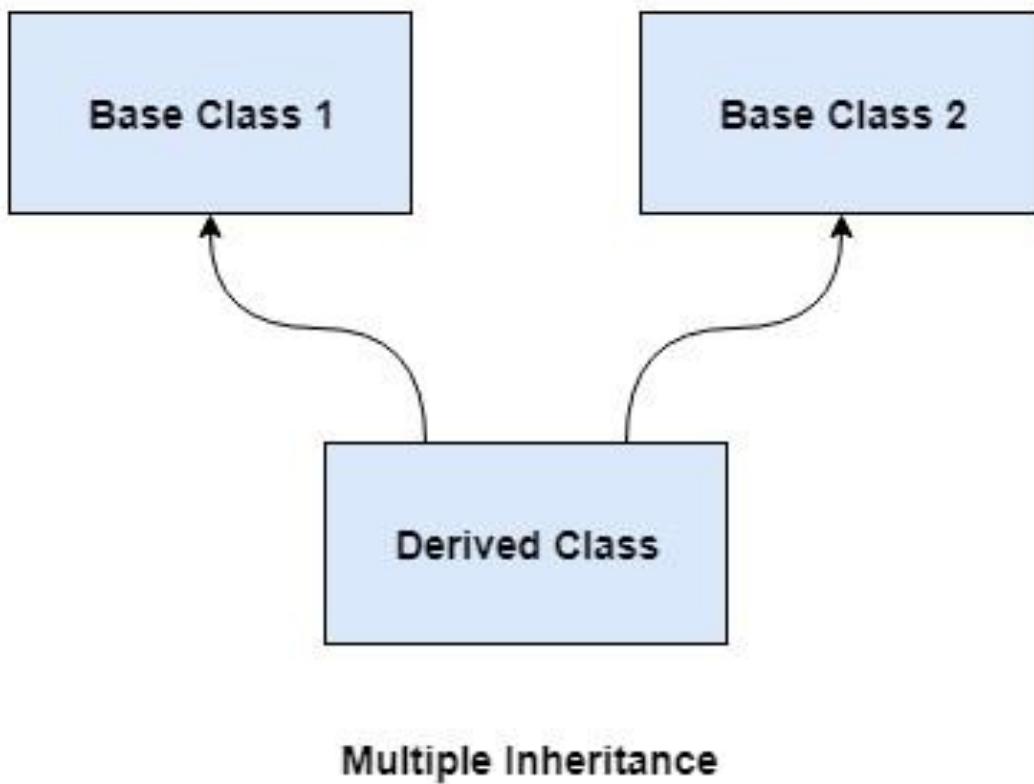
- It provides multiple definitions of the function by changing signature i.e changing number of parameters, change datatype of parameters.
- It can be done in base as well as derived class.

Example:

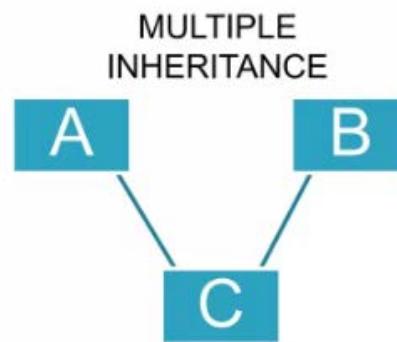
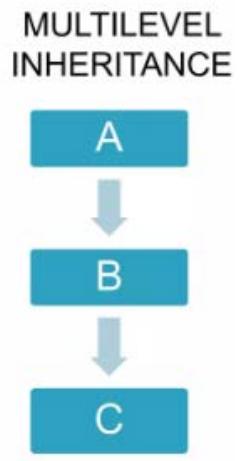
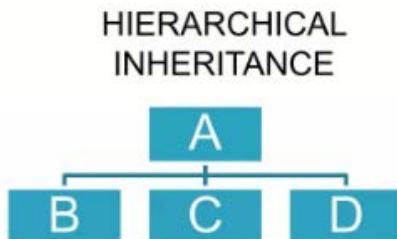
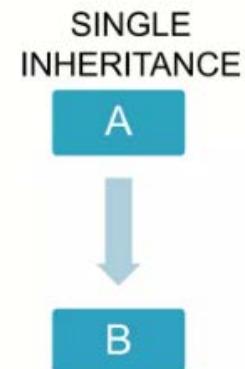
```
void area(int a);  
void area(int a, int b);
```

Multiple Inheritance in C++

- Multiple inheritance occurs when a class inherits from more than one base class.
So the class can inherit features from multiple base classes in the same time.
- Unlike other object oriented programming languages, C++ allow this important features to programmers.



TYPES OF INHERITANCE

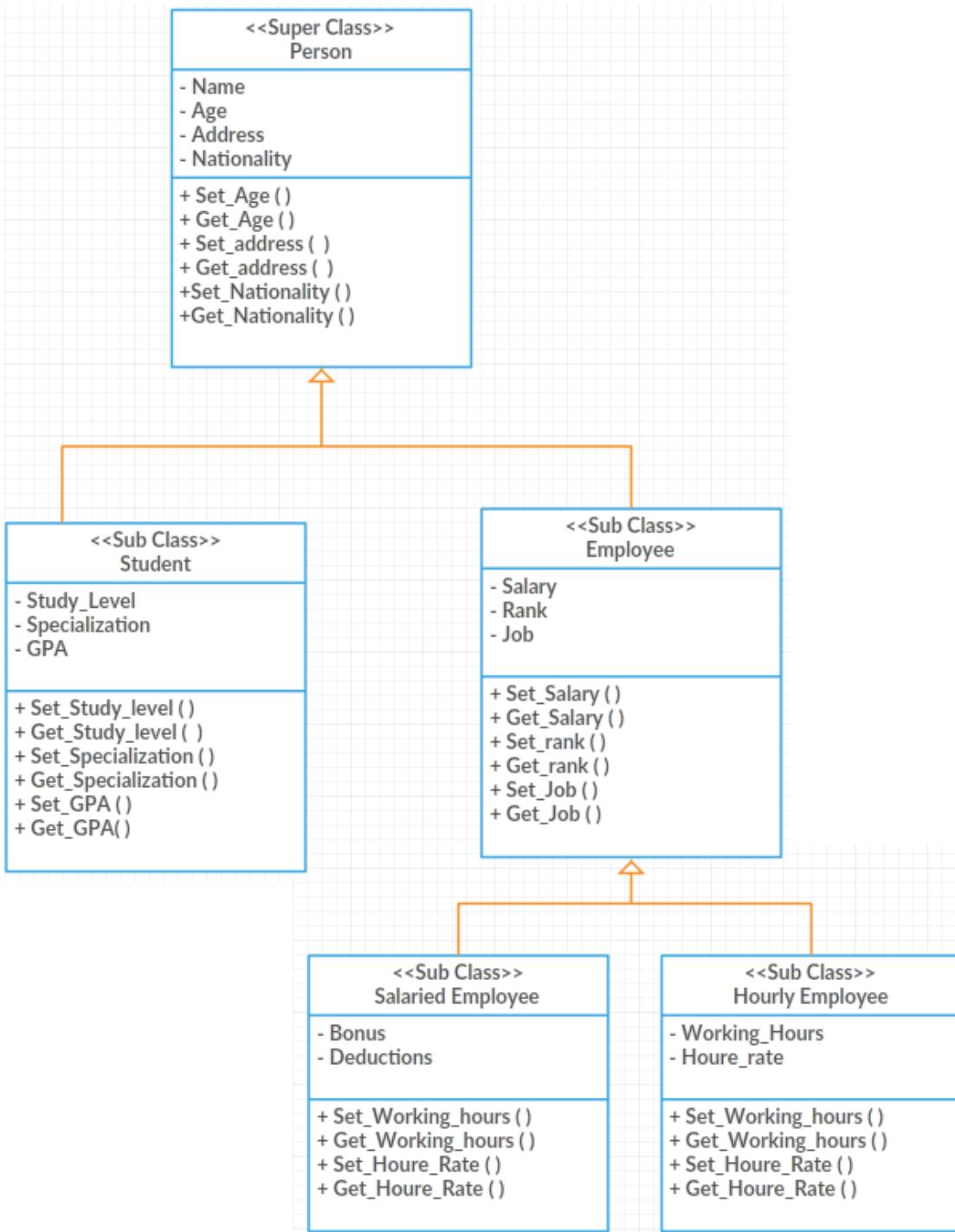


```
class A {  
public:  
    int a = 5;  
A() {  
    cout << "Constructor for class A" << endl;  
}  
};  
class B {  
public:  
    int b = 10;  
B() {  
    cout << "Constructor for class B" << endl;  
}  
};
```

```
class C: public A, public B {  
public:  
    int c = 20;  
C() {  
    cout << "Constructor for class C" << endl;  
    cout << "Class C inherits from class A and class B" << endl;  
}  
};
```

What is Polymorphism ?

- **Polymorphism** is an object-oriented programming concept that refers to the ability of a *variable*, *function* or *object* to take on *multiple* forms.
- with **polymorphism**, class objects belonging to the same hierarchical tree (inherited from a common **parent class**) may have functions with the same name, but with different behaviors.



Shape

String : Color

Draw ()

Erase ()

Get_area ()

Rectangle

int : Length

int : width

Draw ()

Erase ()

Get_area ()

Box

int : SideLength

Draw ()

Erase ()

Get_area ()

Circle

int : radius

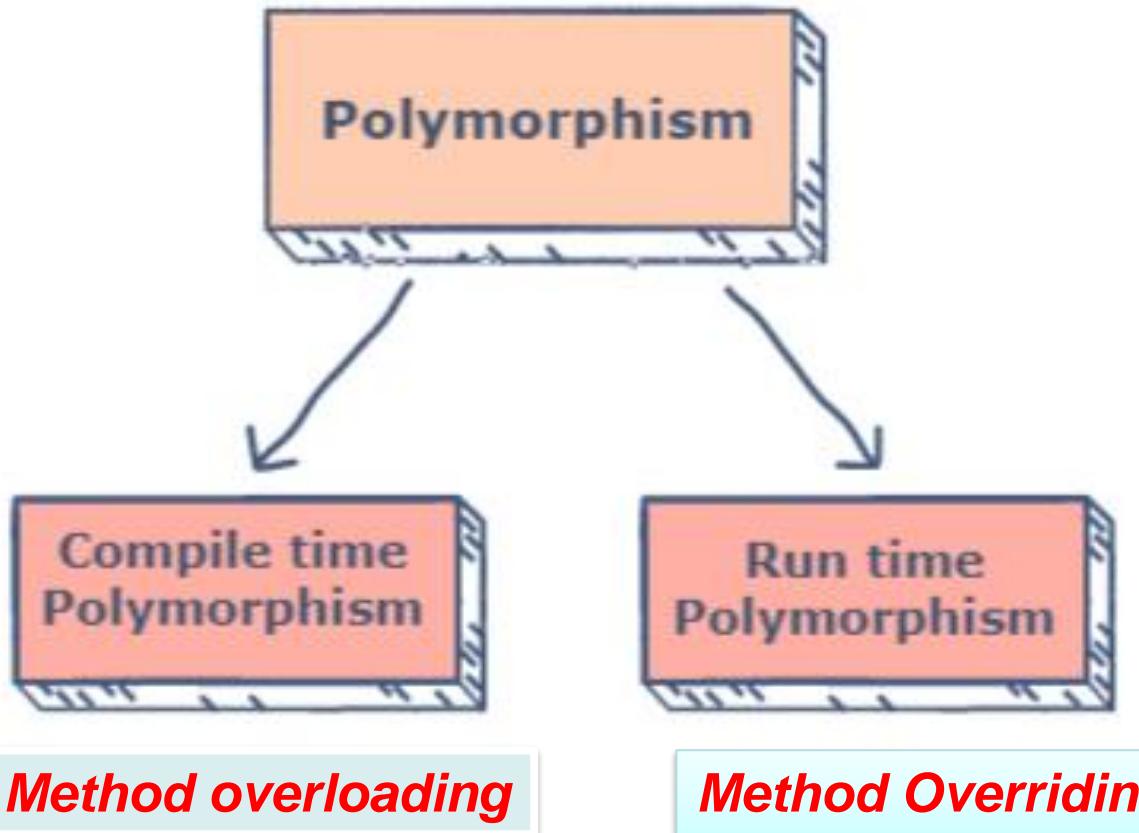
Draw ()

Erase ()

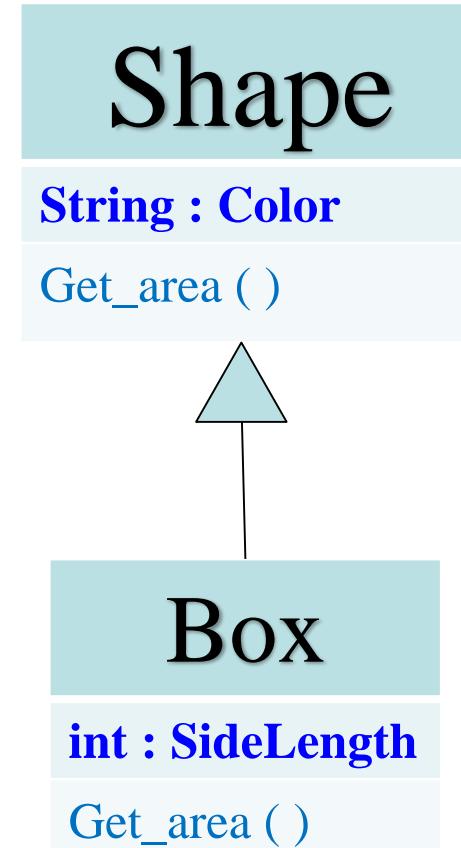
Get_area ()

Get_circumference ()

Types of Polymorphism



```
int      Add(int x , int y);  
double  Add (double x, double y);
```



```
Box b;  
b.Get_area ();
```

Functions Overriding using Virtual Functions

- A virtual function a member function which is declared within a base class and is re-defined(Overridden) by a derived class. When you refer to a derived class object using a pointer to the base class, you can call a virtual function for that object and execute the derived class's version of the function.
- Virtual functions ensure that the correct function is called for an object, regardless of the type of reference (or pointer) used for function call.

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

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

Shape

String : Color

Draw ()

Erase ()

Get_area ()

Rectangle R;

Rectangle

int : Length

int : width

Draw ()

Erase ()

Get_area ()

Box

int : SideLength

Draw ()

Erase ()

Get_area ()

Shape* shap_ptr ;

shap_ptr

Circle

int : radius

Draw ()

Erase ()

Get_area ()

Get_circumference ()

Pure Virtual Functions

```
class Shape {  
protected:  
    int width, height;  
  
public:  
    Shape(int a = 0, int b = 0) {  
        width = a;  
        height = b;  
    }  
  
    // pure virtual function  
    virtual int area() = 0;  
};
```

Abstract Classes - Interfaces

- An interface (Abstract Class) describes the behavior or capabilities of a C++ class without committing to a particular implementation of that class.
- The purpose of an **abstract class** is to provide the Desired base class Form which will be inherited by other classes in the class hierarchy.
- Abstract classes cannot be used to instantiate objects and serves only as an **interface**.
- A class is made abstract by declaring *at least one* of its functions as pure virtual function.

Shape

String : Color

Draw ()

Erase ()

Get_area ()

Rectangle

int : Length

int : width

Draw ()

Erase ()

Get_area ()

Box

int : SideLength

Draw ()

Erase ()

Get_area ()

Circle

int : radius

Draw ()

Erase ()

Get_area ()

Get_circumference ()

- A class is made abstract by declaring at least one of its functions as pure virtual function.

```
class Shape {  
protected:  
    int width, height;  
  
public:  
    Shape(int a = 0, int b = 0) {  
        width = a;  
        height = b;  
    }  
  
    // pure virtual function  
    virtual int area() = 0;  
};
```

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.
- The prototypes for friend functions appear in the class definition.
- friends are not member functions.

```
class className{  
    ....  
    friend returnType functionName(arg list);  
};
```

Friend Class

- Just like friend functions, we can also have a friend class.
- Friend class can access private and protected members of the class to which it is a friend.
- Note that the friendship is not mutual unless we make it so.
- The friendship of the class is not inherited. This means that as class B is a friend of class A, it will not be a friend of the subclasses of class A.

```
class A{  
    ....  
    friend class B;  
};  
class B{  
    ....  
};
```

Employee

String : Name

String : SSN

Double : Salary

Get_TotalSalary();

Print ();

Sales

Float : Goss_Sales

Float : Commission_Rate

Set_Gross_Sales()

Set_commission_Rate ()

Engineer

String : Speciality

Int : Experience

int: overtime_hours

Float:overtime_hour_rate

Set_OverTime_Hours()

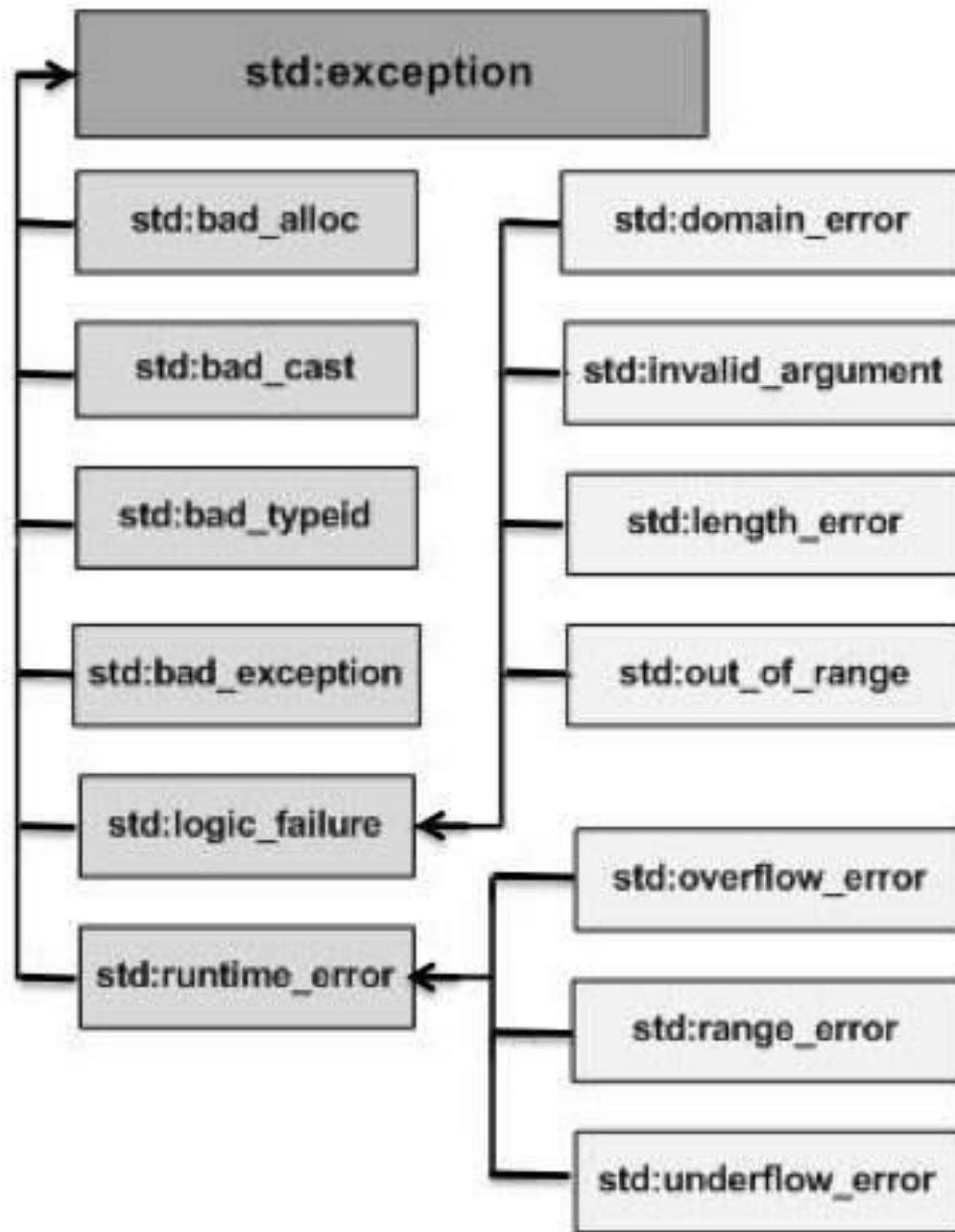
Set_OverTime_hour_rate ()

Introduction

- **Exceptions**
 - Indicate problems that occur during a program's execution
 - 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

Exception Handling

- An *exception* is a class
 - Usually derived from one of the system's exception base classes
- Exception Class is the standard C++ base class for all exceptions
- Provides derived classes with virtual function ***what ()***
 - Returns the exception's stored error message
- If an exceptional or error situation occurs, program *throws* an object of that class.



- Exceptions provide a way to transfer control from one part of a program to another. three keywords: **try**, **catch** , **throw** are used
- **try** – A **try** block identifies a block of code for which particular exceptions will be activated. It's followed by one or more catch blocks.
- **throw** – A program throws an exception when a problem shows up. This is done using a **throw** keyword.
- **catch** – A program catches an exception with an exception handler at the place in a program where you want to handle the problem. The **catch** keyword indicates the catching of an exception.

```
int x = 10, y = 2;  
try  
{  
if (y == 0)  
    throw exception ( );  
else  
cout << x / y << endl;  
}  
catch (exception e)  
{  
    cout << e.what ( ) << endl;  
}  
cout << "The Program Continued" << endl;
```

```
int x = 10, y = 2;  
try  
{  
if (y == 0)  
    throw "division by zero Exception";  
else  
cout << x / y << endl;  
}  
catch (const char* msg)  
{  
    cout << msg << endl;  
    cout << "Y must be greater than 0" << endl;  
}  
cout << "The Program Continued" << endl;
```

```
try {
    int age = 15;
    if (age > 18) {
        cout << "Access granted - you are old enough.";
    } else {
        throw (age);
    }
}
catch (int myNum) {
    cout << "Access denied - You must be at least 18 years old.\n";
    cout << "Age is: " << myNum;
}
```

Exception Handling – General Syntax

- ```
try {
 // Block of code to try
 throw exception; // Throw an exception when a problem arise
}
catch () {
 // Block of code to handle errors
}
```

# Exception Handling

```
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
}
...
```

**catch** clauses attempted  
in order; first match wins!

# Handle Any Type of Exceptions (...)

```
int x = 10, y = 2;
try
{
 if (y == 0)
 throw "Integer division by zero";
 else
 cout << x / y << endl;
}
catch (...)
{ cout << "An exception Caught" << endl;
}
cout << "The Program Continued" << endl;
```

# 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