CHAPTER 2

CLASSES AND OBJECTS

Er. Ganga Gautam

Outline:

1. Introduction to C++: Origin of C++, Basic C++ Program Structure, Console Input/output Streams and Manipulators
2. Structure in C and C++
3. Classes and Objects
4. Array of Objects
5. Class Diagram and Object Diagram
6. Access Specifiers and Visibility Mode
7. State and Behavior, Methods and Responsibilities
8. Implementation of Data Abstraction, Encapsulation, Message Passing and Data Hiding
9. Memory Allocation for Objects
10. Constructor: Default Constructor, Parameterized Constructor, Copy Constructor
11. Constructor Overloading
12. Destructors
13. Dynamic Memory Allocation: new and delete.
14. Dynamic Constructor
15. Functions: Inline function, Default argument, Passing and Returning by Value, Pointer and Reference, Static Data Member and Static Member Function
16. Friend Function and Friend Class

Introduction to C++

- C++ is a powerful and widely used high level programming language.
- C++ is an extension of the C programming language i.e. an attempt to add object-oriented features (plus other improvements) to C, earlier it was called as “C with classes”.
- It is a compiled, which means that code is translated into machine-readable language instructions before execution.
- C++ is known for its efficiency, performance, and flexibility.
- It is used in a variety of applications, including system software, game development, and high-performance computing.
- C++ has a large and active community of developers, making it a valuable skill to have.

Origin of C++

- C++ is an extension of the C programming (Dennis Ritchie, Bell Labs ,1970s).
- In the late 1970s, Bjarne Stroustrup, a Danish computer scientist, began developing an extension of the C language.
- He named it "C with Classes" and intended to add object-oriented programming features to C.
- The first version of C with Classes was implemented in 1979.
- Stroustrup continued to refine and expand the language.
- In 1983, he renamed it "C++" to reflect the added features and improvements.
- The name "C++" is a reference to the increment operator in C, indicating an enhanced version of C.
- The standardization of C++ began in the late 1980s to ensure compatibility and portability of the language.
- The first official standard, known as "C++98" was published in 1998.
- Subsequent standards like C++03, C++11, C++14, C++17, C++20 and C++23 have brought further enhancements.
- Its wide adoption and continued development make it an essential tool for many developers today.

Review of Structure

- Structure: A mechanism to store data of dissimilar datatype
- Declare with keyword: struct
- E.g.:

```
1  struct Student
2  {
3      char name[20];
4      int roll;
5      float fee, marks;
6  }st;
```

Sample program

Review of Structure

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

struct Student
{
    char name[20];
    int roll;
    float fee, marks;
}st;
```

```
main()
{
    cout<<"Enter name,rollno,fee and marks of a student"<<endl;
    cin>>st.name>>st.roll>>st.fee>>st.marks;

    cout<<"The name, rollno, fee and marks are:"<<endl;
    cout<<st.name<<endl;
    cout<<st.roll<<endl;
    cout<<st.fee<<endl;
    cout<<st.marks<<endl;
}
```

Output →

```
Enter name,rollno,fee and marks of a student
Arjun
25
5250.25
95.5
The name, rollno, fee and marks are:
Arjun
25
5250.25
95.5
```

Limitation of Structure

1. Does not allow the struct data type to be treated like built-in datatypes.

```
1  struct Student
2  {
3          float mark;
4  }st1,st2,st3;
5
6  int main()
7  {
8      st1.mark=55;
9      st2.mark=95
10     st3=st1+st2; // This is invalid
11 }
```

Here Student st1 and st2 are assigned with respective marks using dot operator. But we cannot add or subtract these two marks using structure variables.

2. No data hiding
3. No functions and constructors can be placed inside structure
4. Cannot have static members inside it

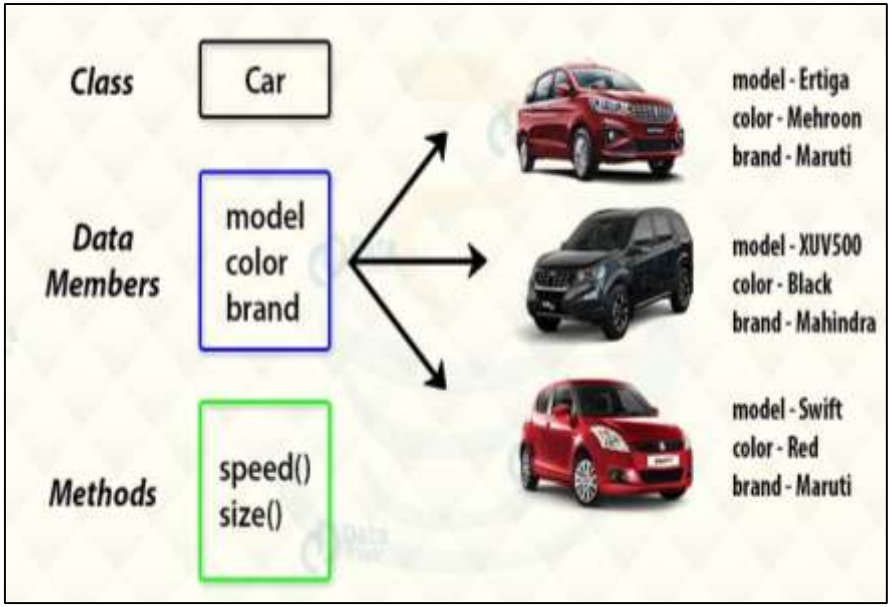
2.2 Concept of Class and Object

Class

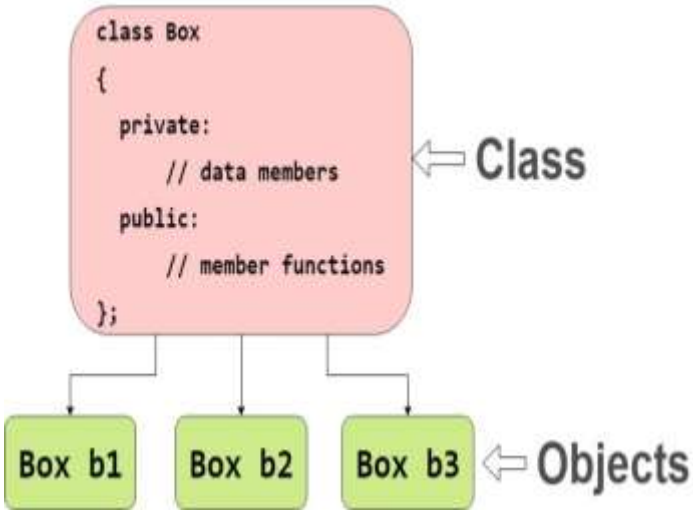
- Class is a container that holds
  - Data members
  - Function(or methods) members
- It is like a blueprint for an object

Object

- Is an instance of class



How to write a program with class and object




```
1  class Box
2  {
3      private:
4          int len, br, ht;
5      public:
6          void showVolume()
7          {
8          ..........
9          }
10 } b1, b2, b3;
```

How to declare class and object

```
1  class Box          //Class declaration
2  {
3      private:
4          int len, br, ht;
5      public:
6          void showVolume()
7          {
8          ……..
9          }
10 } b1, b2, b3;  // Objects creation just after class definition
```

OR

```
1  class Box //Class declaration
2  {
3      private:
4          int len, br, ht;
5      public:
6          void showVolume()
7          {
8          ……
9          }
10 }
11
12 Box b1, b2, b3;  // Objects creation beyond class definition
```

Sample program 2.2

WAP using OOP to find area of rectangle

```
1  #include<iostream>
2  using namespace std;
3  class Rectangle      // class declaration
4  {
5      private:
6          int l, b, a;      // data members
7      public:
8          void getdata()    // function member
9          {
10             cout<<"Enter length and breadth"<<endl;
11             cin>>l>>b;
12         }
13         void showdata()   // function member
14         {
15             a=l*b;
16             cout<<"Area is "<<a<<endl;
17         }
18 };
```

```
20 main()
21 {
22     Rectangle ob;
23     ob.getdata();
24     ob.showdata();
25 }
```

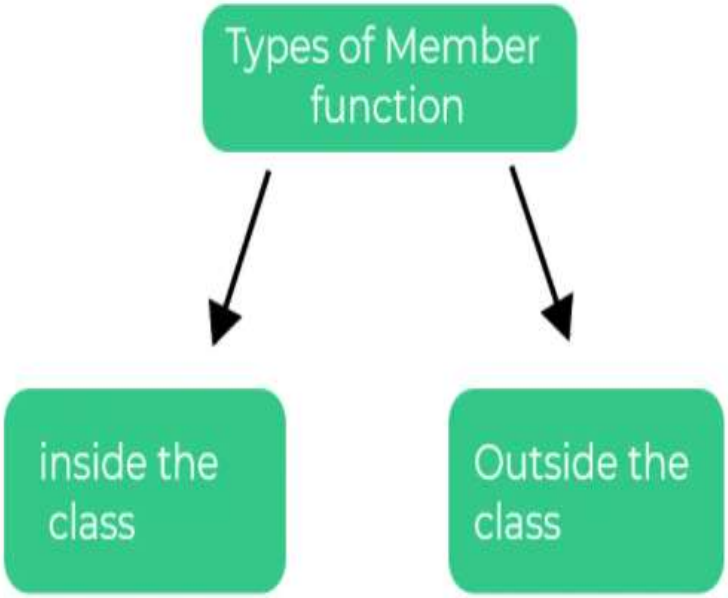
```
Enter length and breadth
10
20
Area is 200
```

Accessing data members and member functions

- The data members and member functions of class can be accessed using the dot('.') operator with the object.
- For example if the name of object is *obj*,
  - you want to access the member function with the name *printName()* then you will have to write *obj.printName()* .
  - you want to access the data member with the name *length* then you will have to write *obj.length*.

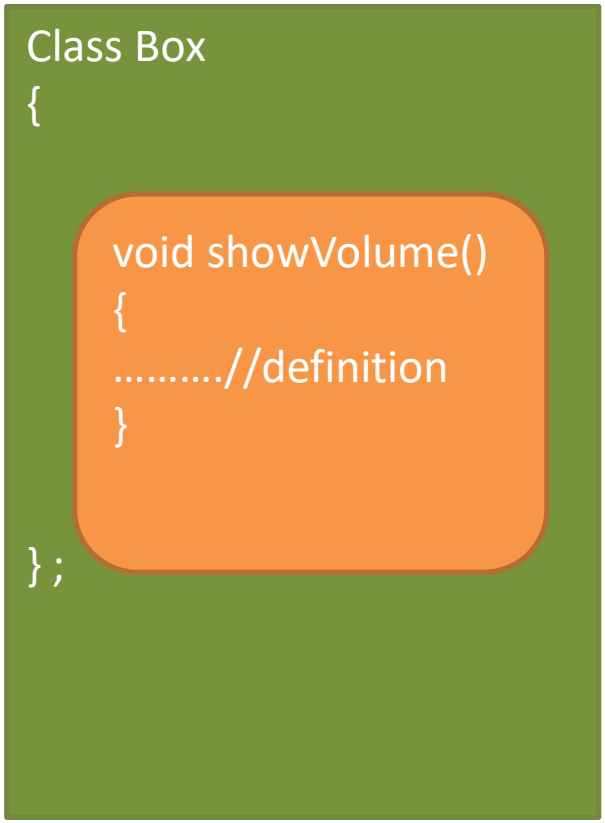
2.4 Ways of Defining Member functions

- Two ways of defining class function members
  1. Inside the class
  2. Outside the class



Defining Member functions inside the class

- Function declaration and definition both are done inside the class.



Sample program 2.3

```
1  #include<iostream>
2  using namespace std;
3
4  class Box
5  {
6      private:
7          int len, br, ht, vol;
8      public:
9          void showVolume()
10         {
11             cout<<"Enter length, breadth and height"<< endl;
12             cin>>len>>br>>ht;
13             vol=len * br * ht;
14             cout<< "Volume is: " << vol;
15         }
16 };
17
18 main( )
19 {
20     Box b1;
21     b1.showVolume();
22 }
```

```
Enter length, breadth and height
10
20
10
Volume is: 2000
```

Defining Member functions outside the class

- Function declaration done inside the class
- But definition is outside the class

```
Class Box
{

   void showVolume();   // only function declaration

};
```

```
void showVolume()
{
……..//definition
}
```

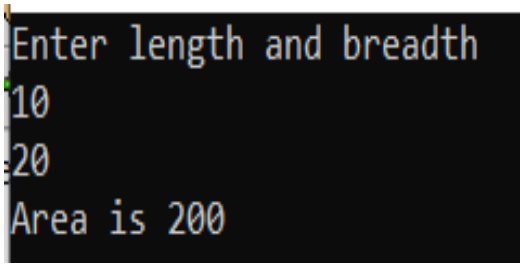
Sample program 2.4

```
1  #include<iostream>
2  using namespace std;
3
4  class Box
5  {
6      private:
7          int len, br, ht, vol;
8      public:
9          void showVolume();
10
11 };
12
13 void Box:: showVolume()
14     {
15         cout<<"Enter length, breadth and height"<< endl;
16         cin>>len>>br>>ht;
17         vol=len * br * ht;
18         cout<< "Volume is: " << vol;
19     }
20
21 main( )
22 {
23     Box b1;
24     b1.showVolume();
25 }
```

```
Enter length, breadth and height
10
20
10
Volume is: 2000
```

Sample program 1

```
#include <iostream>

using namespace std;

class Rectangle{
   private:
   int l,b,a;

   public:
   //Function defination inside class
   void getData(){
      cout<<"Enter length and breadth"<<endl;
      cin>>l>>b;
   }
   // function declaration for function defined outside class
   void showData();
};
//Function defination outside class
void Rectangle::  showData(){
      a=l*b;
      cout<<"Area is "<<a<<endl;
}

int main() {
  Rectangle ob;
  ob.getData();
  ob.showData();
}
```



Sample program 2

WAP to declare a class *Item* having two member variables (*number* and *cost*) and two member functions: *getdata()* which receives the arguments whenever called and *putdata()* which displays the information.

```
#include<iostream>
using namespace std;
class Item
{
    int num;
    float cost;
    public:
        void getdata(int a,float b)
        {
            num=a;
            cost=b;
        }
        void putdata()
        {
            cout<<"Number="<<num<<endl;
            cout<<"Cost="<<cost<<endl;
        }
};
```

```
main()
{
    Item x;
    x.getdata(100,75.5);
    x.putdata();
}
```

```
Number=100
Cost=75.5
```

Sample program 3

WAP to declare a class Test having three member variables: *n1, n2* and *sm*. And two member functions *getdata()* which receives the data and *display()* that displays information to calculate the sum of two numbers. All the member function should be defined outside the class.

```
1  #include<iostream>
2  using namespace std;
3  class Test
4  {
5     int n1,n2,sm;
6     public:
7        void getdata();
8        void display();
9  };
10 main()
11 {
12    Test t;
13    t.getdata();
14    t.display();
15 }
```

```
16 void Test::getdata()
17 {
18    cout<<"Enter any two numbers:"<<endl;
19    cin>>n1>>n2;
20 }
21 void Test::display()
22 {
23    sm=n1+n2;
24    cout<<"The Sum is ="<<sm<<endl;
25 }
```

```
Enter any two numbers:
5
10
The Sum is =15
```

Sample program 4

WAP to create a class Student having three member variables (*age*, *name* and *address*) and two member function: *getdata()* which receives the data from user and *showdata()* that displays information.

```
#include<iostream>
using namespace std;
class Student
{
    int age;
    char name[20], add[30];
    public:
        void getdata()
        {
            cout<<"Enter age, name and address"<<endl;
            cin>>age>>name>>add;
        }
        void showdata()
        {
            cout<<"The age, name and address are:"<<endl;
            cout<<age<<endl;
            cout<<name<<endl;
            cout<<add<<endl;
        }
};
```

```
main()
{
    Student st;
    st.getdata();
    st.showdata();
}
```

```
ENter age, name and address
25
umesh
Pokhara
The age, name and address are:
25
umesh
Pokhara
```

Input/output streams

- Data for a program may come from several sources.
- The connection between a program and a data source or destination is called a **stream**.
- An input stream handles data flowing into a program.
- An output stream handles data flowing out of a program.

Array of Objects

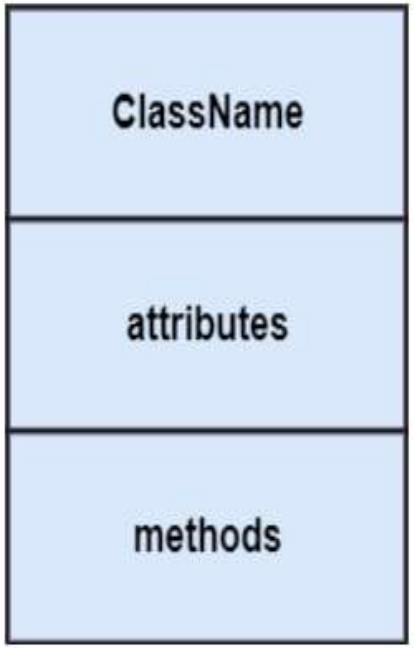
- An array of objects is created by declaring an array where each element is an object of a specific class.

```
#include <iostream>
class Person {
public:
    string name;
    int age;

    void display() {
        cout << "Name: " << name << ", Age: " << age << endl;
    }
};

int main() {
    const int arraySize = 3;
    Person p[arraySize];

    p[0].name = "Alice";
    p[0].age = 25;

    p[1].name = "Bob";
    p[1].age = 30;

    p[2].name = "Charlie";
    p[2].age = 35;

    for (int i = 0; i < arraySize; ++i) {
        p[i].display();
    }
    return 0;
}
```

Class Diagram

- Describes the structure of a system by showing the system's classes, their attributes, operations (or methods) and their relationships between the classes.

- The upper section encompasses class name
- Middle section constitutes the attributes.
- Lower section contains methods or operations.

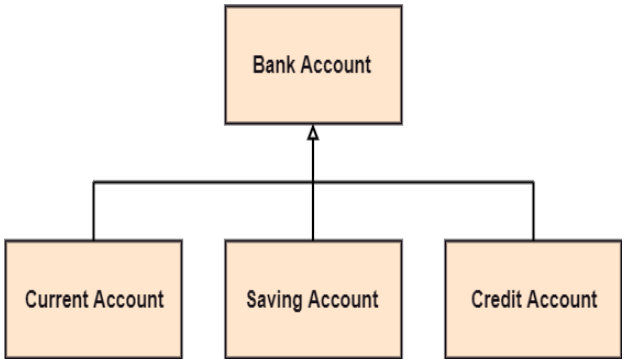


Class Diagram Notations:

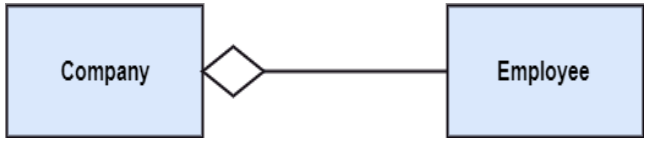
Relationship

- **Generalization :** Relationship between parent class and child class
- **Association :** describes connection between two or more objects
- **Aggregation :** subset of association. Eg. Company encompasses a number of employees.
- **Composition:** It represents a whole-part relationship.

Generalization:

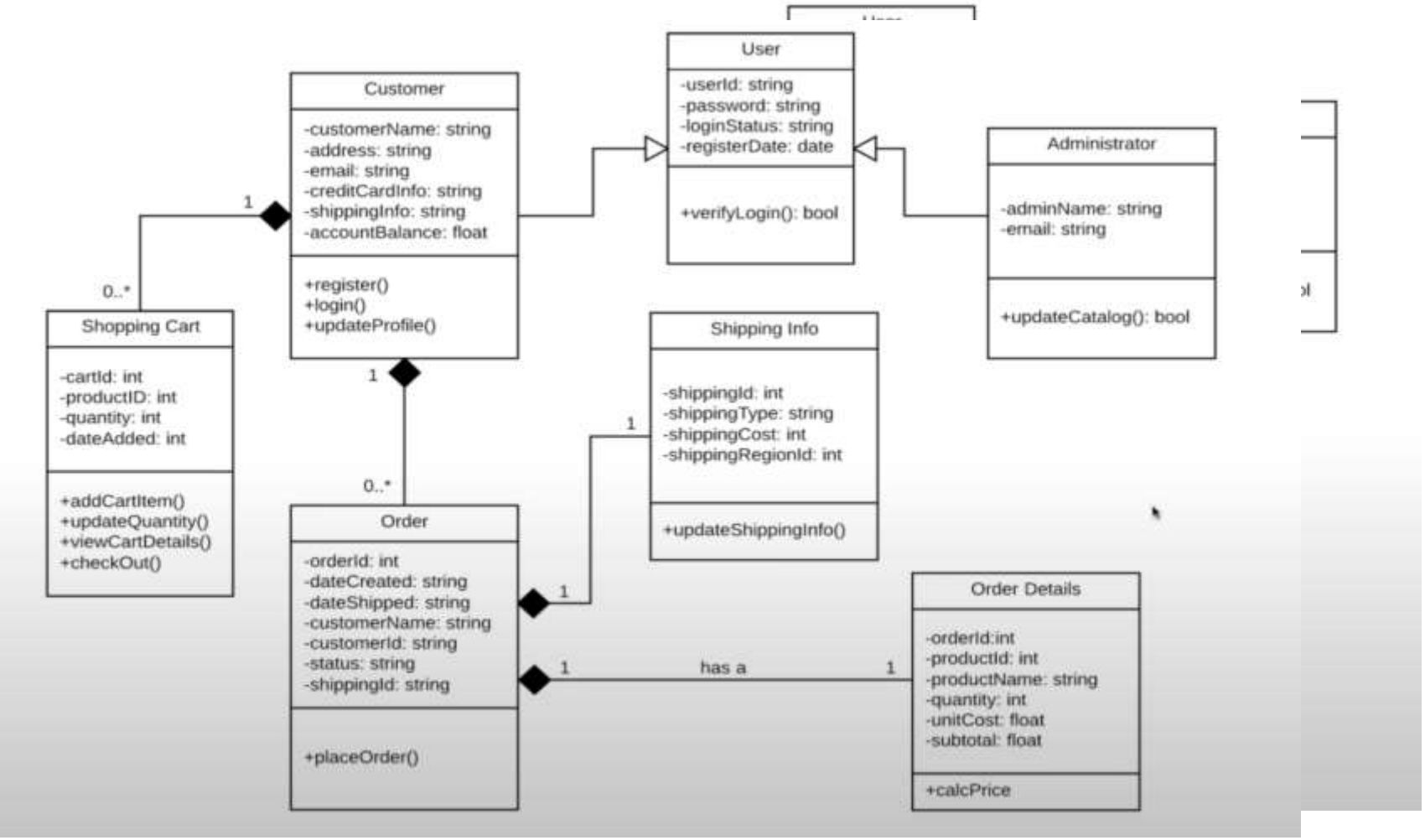


Aggregation:

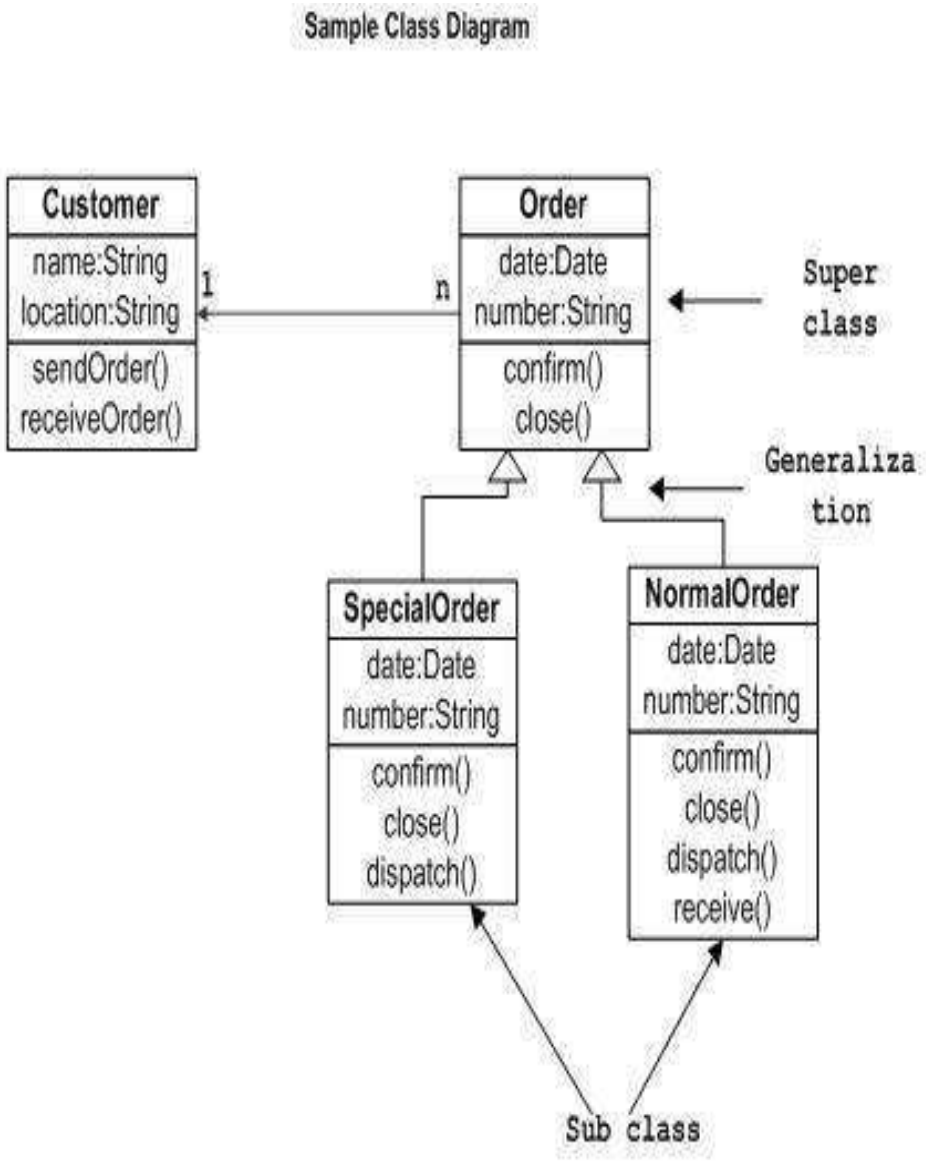


Association:

Composition:

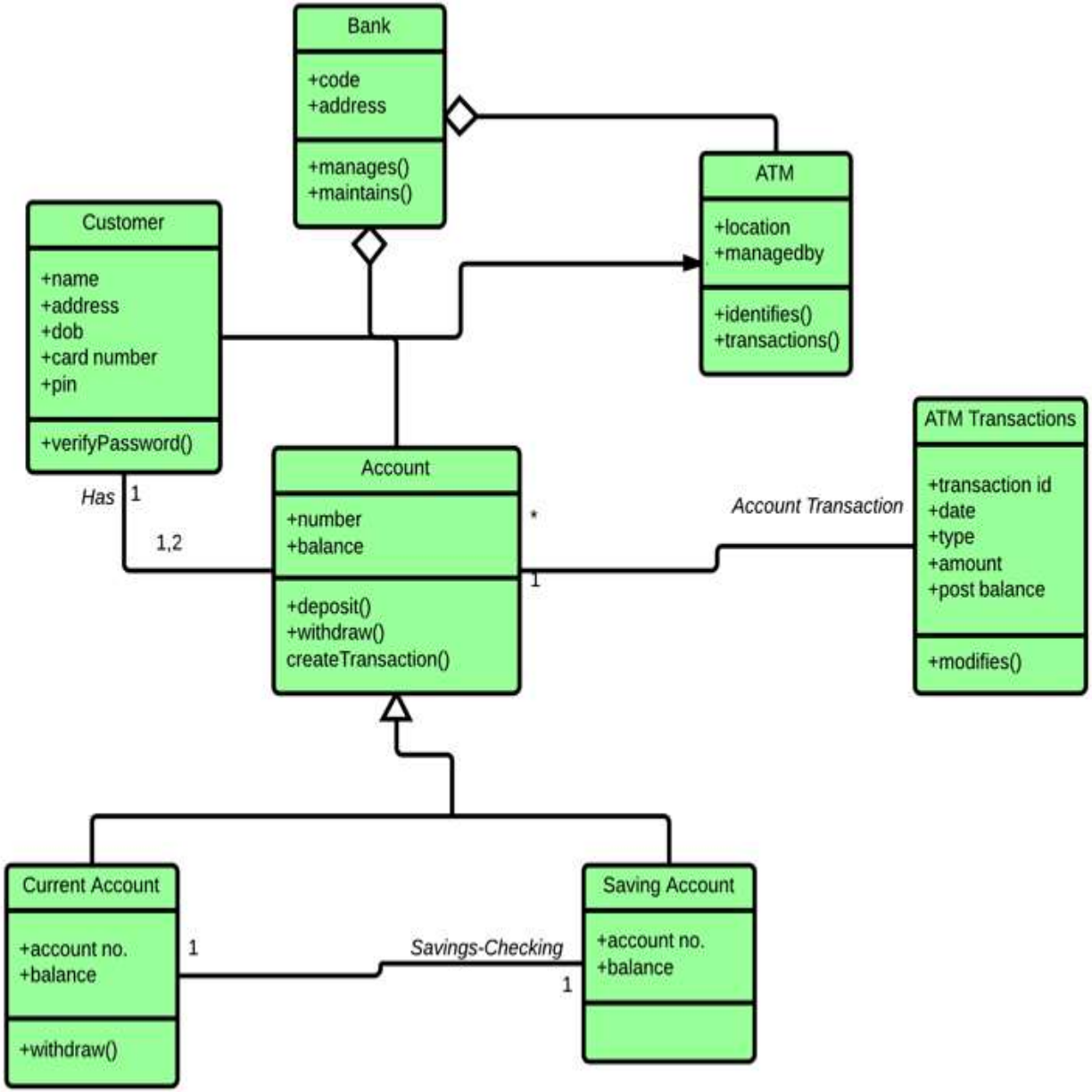


Visit site for tutorial: https://www.youtube.com/watch?v=UI6lqHOVHic



Order Management System

Source:
https://www.tutorialspoint.com/uml/uml_class_diagram.htm



ATM system Source:https://www.guru99.com/uml-class-diagram.html

Object Diagram

Object is an instance of class.

It has its own state and data values at a moment.

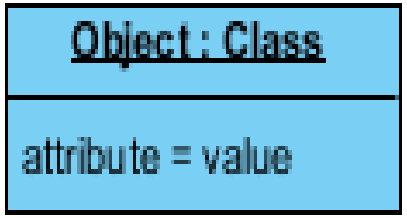
Notations in Object Diagram

- **Object Names**:

  Every object is actually symbolized like a rectangle, that offers the name from the object and its class underlined as well as divided with a colon.

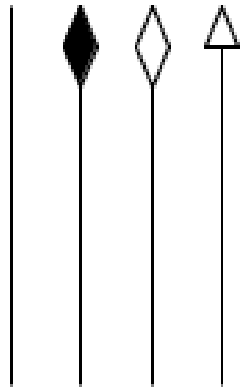
- **Object Attributes:**

  Similar to classes, you are able to list object attributes inside a separate compartment. However, unlike classes, object attributes should have values assigned for them.

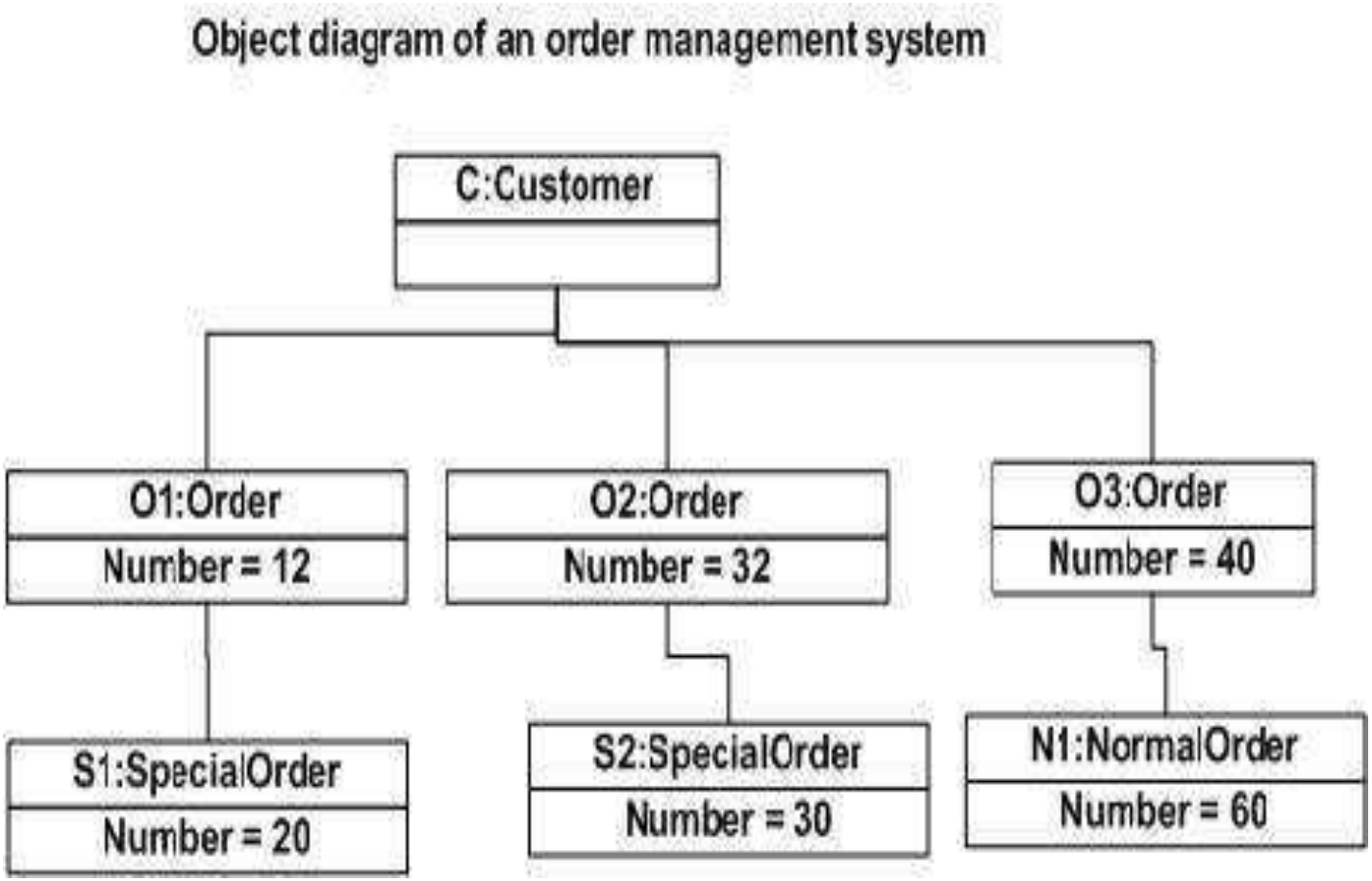


- **Links:**

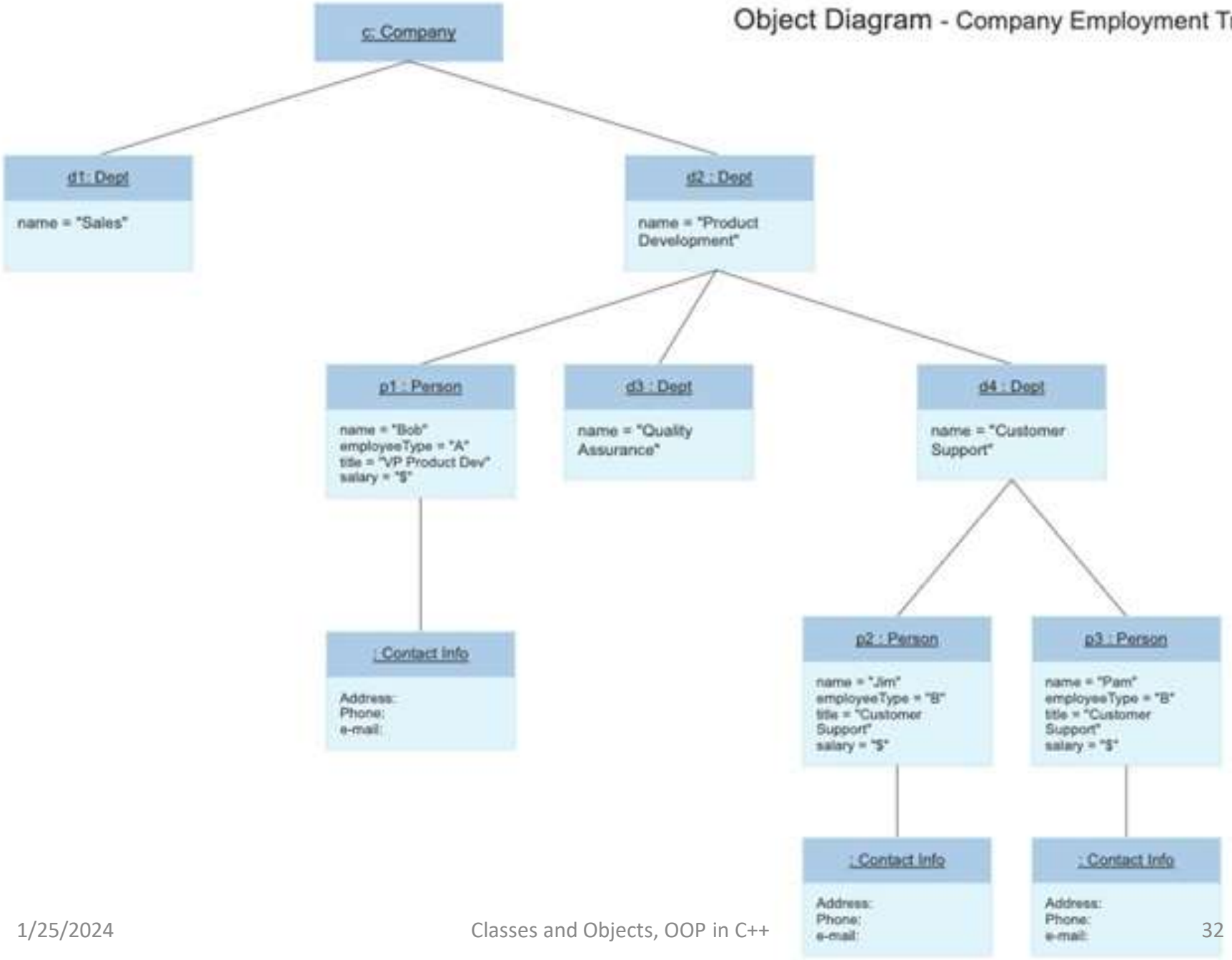
  Links tend to be instances associated with associations. You can draw a link while using the lines utilized in class diagrams.



Sample Object Diagrams



Object Diagram - Company Employment Tree



Practice for Students:

Class Diagram for Library management system

State and Behavior

- State and behavior are the basic properties of an object.
- State tells us about the type or the value of that object, whereas

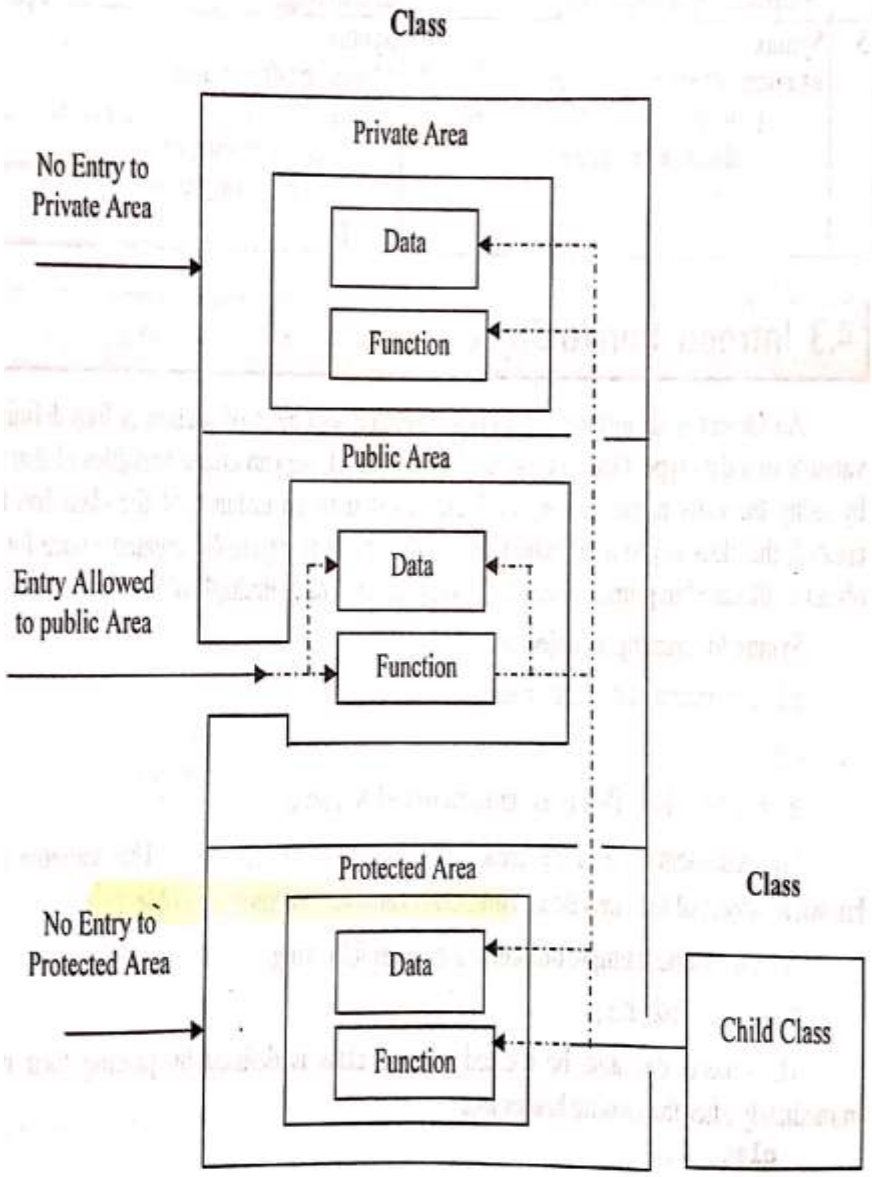
  behavior tells us about the operations or things that the object can perform.
- For example, lets us consider an object called car.
  - So, car object will have color, engine type, wheels, etc. are its state.
  - This car can run at 180 kmph, it can turn right or left, it can go back or forth, it can carry four people, etc. These are its behaviors.
- Hence, in OOP, states are the instance variable that we use in the class. And behavior are the functions that we use in the class

Method and Responsibility

- A method in object-oriented programming is a procedure associated with a class.
- A method defines the behavior of the objects that are created from the class.
- Another way to say this is that a method is an action that an object is able to perform.
- One of the most important idea used in object-oriented design is that of "*an object must be responsible for itself*".
- *An object must contain the data (attributes) and code (methods) necessary to perform any and all services that are required by the object.*
- This means that the object must have the capability to perform required services itself or at least know how to find and invoke these services.

Data Hiding

- Data hiding is the mechanism to hide the data members of a class and providing the access to only some of them.
- The class can secure its members from outside of the world (i.e. outside of class definition).
- It means that data is concealed (masked) within a class so that it cannot be accessed by functions outside the class even by mistake.
- The mechanism used to hide data is to put it in a class & make it private.
- And this is done by using the Access Specifiers



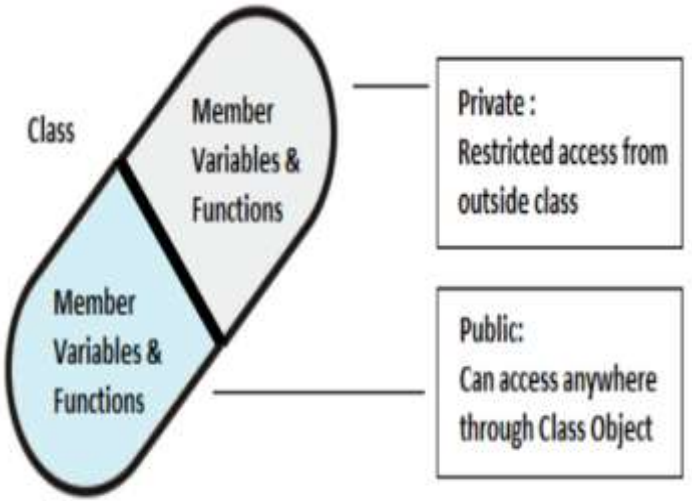
Access specifiers:

- Private members
  - Cannot be accessed by outside class not even the objects
- Public members
  - Can be accessed from outside the class
- Protected members
  - Acts like as private members
  - Can only be accessed by derived class.

```
1  #include<iostream>
2  using namespace std;
3  class A
4  {
5     public:
6        int n1; //accesssible by ob
7  };
8  /*
9  class A
10 {
11    protected: //not accesssible by ob
12       int n1;
13 };
14 class A
15 {
16    private: //not accesssible by ob
17       int n1;
18 };
19 */
20 main()
21 {
22    A ob;
23    ob.n1=5;
24    cout<<ob.n1;
25 }
```

Encapsulation

- Encapsulation is the process of combining data and function into a single unit called class.
- This is to prevent the access to the data directly, the access to them is provided through the functions of the class.
- It is one of the popular features of OOP that helps in **data hiding**.
- In class, member variables and function can be declared as Private so that they cannot be accessed from outside the class.
- This makes the data become more secure



Encapsulation

- Let us consider the program below:
- The variables l, b, and h are private.
- This means that they can be accessed only by other members of the ***Cuboid*** class, and not by any other part of our program.
- This is one way, encapsulation is achieved.

```
class Cuboid
{
    private:
        int l;
        int b;
        int h;

    public:
        int getVol()
        {
            return l * b * h;
        }

};
```

Abstraction

- Abstraction means displaying only essential information and hiding the details.
- Data abstraction refers to providing only essential information about the data to the outside world, hiding the background details or implementation.
- Consider a real life example of a man driving a car.
  - The man only knows that pressing the accelerators will increase the speed of car or applying brakes will stop the car but he does not know about how on pressing accelerator the speed is actually increasing, he does not know about the inner mechanism of the car or the implementation of accelerator, brakes etc in the car.
  - This is what abstraction is.

Sample Program 2.8

Illustration of Abstraction

```
1  #include <iostream>
2  using namespace std;
3  class AbstractionDemo
4  {
5      private:
6          int a, b;
7
8      public:
9          // method to set values of  private members
10         void set(int x, int y)
11         {
12             a = x;
13             b = y;
14         }
15         void display()
16         {
17             cout<<"a = " <<a << endl;
18             cout<<"b = " << b << endl;
19         }
20 };
```

```
21 main()
22 {
23     AbstractionDemo obj;
24     obj.set(10, 20);
25     obj.display();
26 }
```

- In the above program we are not allowed to access the variables a and b directly, however one can call the function set() to set the values in a and b and the function display() to display the values of a and b.

Message passing formalization

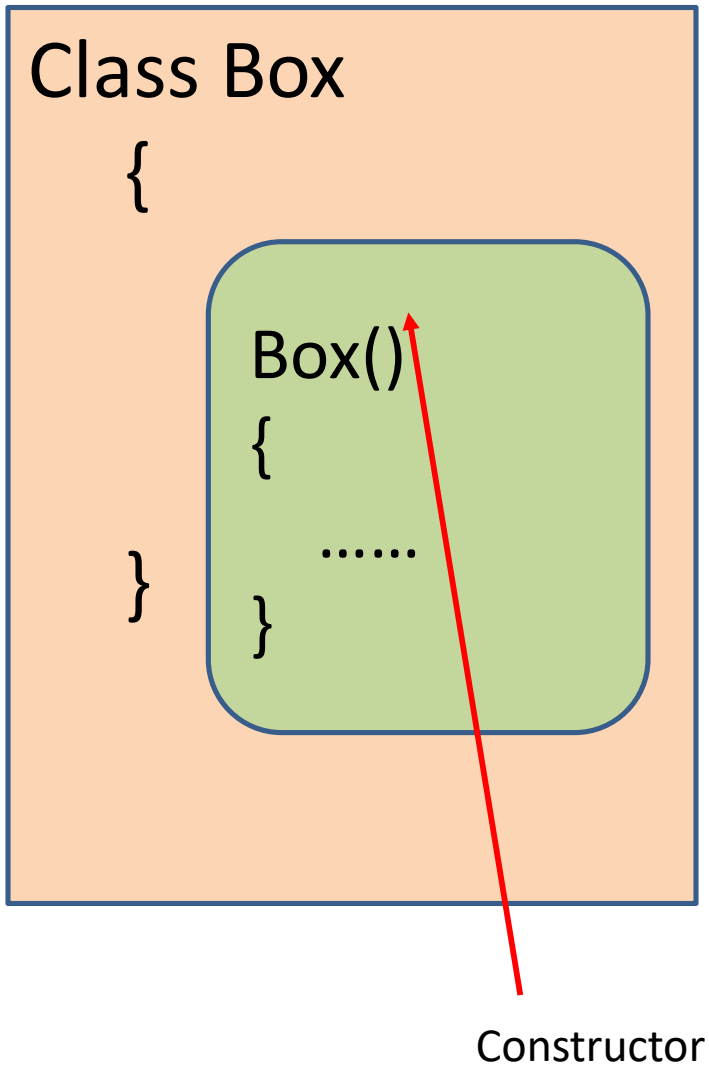
- Aka *Method lookup*
- The process of passing argument to objects is called message passing.
- Objects communicate with one another by sending and receiving information to each other.
- Message passing involves specifying the name of the object, the name of the function and the information to be sent.
- For example:

```
st.mark(name)
```

Here, ***st*** is object, ***mark*** is function, ***name*** is information.

Constructors and its types

- A special type of function whose name is the same as the class name.
- It initializes objects of a class.
- Constructor is automatically called when an object is created.
- It has no return type, not even **void**
- Named so because it constructs the values of data members of the class.

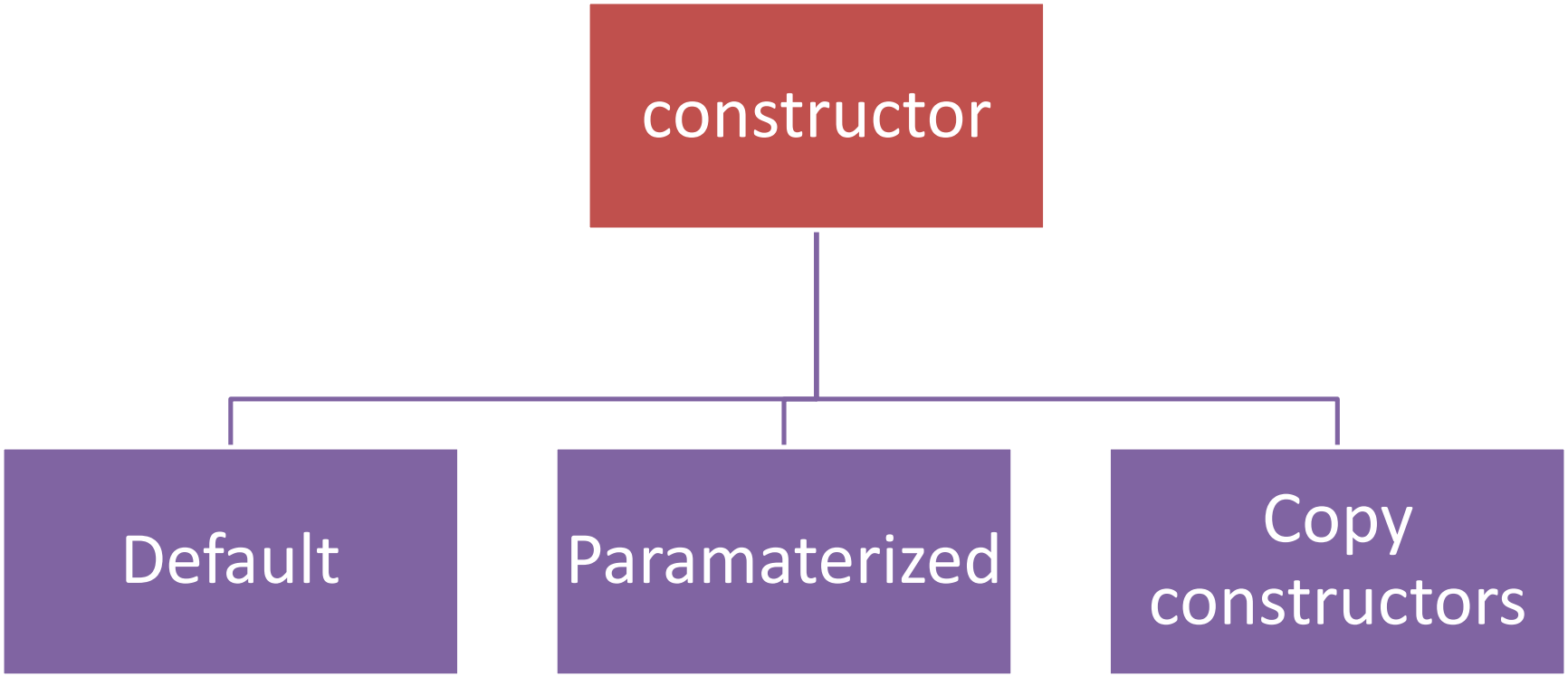


Constructors : Sample program

```
1  //sample program to illustrate constructor
2  #include<iostream>
3  using namespace std;
4  class Box
5  {
6      int m;
7      public:
8          Box(int x)  //constructor
9          {
10             m=x;
11             cout<<"Value of m="<<m<<endl;
12             cout<<"constructor created"<<endl;
13         }
14 };
15 main()
16 {
17     Box b1(5);
18 }
```

```
Value of m=5
constructor created
```

Constructors : Types



i) Default Constructor

- It is the automatically generated constructor by the compiler if there is no user-defined constructor for that class.
- does not initialize data members to any values: neither to zero , but to random.

i) Default constructor: sample program 1

```
1  //sample program to illustrate default constructor
2  #include<iostream>
3  using namespace std;
4  class Box
5  {
6      int len;
7      public:
8          Box()   //default constructor
9          {
10             len=10;
11             cout<<"Value of length="<<len<<endl;
12             cout<<"constructor created"<<endl;
13         }
14 };
15
16 main()
17 {
18     Box b1;
19 }
```

```
Value of length=10
constructor created

------------------------
```

i) Default constructor: sample program 2

```
1  //sample program to illustrate default constructor
2  #include<iostream>
3  using namespace std;
4  class Box
5  {
6      int len,br,a;
7      public:
8          Box(int x=5) //default constructor with default agrument
9          {
10             len=10;
11             br=x;
12             a=len*br;
13             cout<<"Area="<<a<<endl;
14             cout<<"constructor created"<<endl;
15         }
16 };
17
18 main()
19 {
20     Box b1;
21 }
```

```
Area=50
constructor created

--------------------------------
Process exited after 0.0872 seconds with return value 0
Press any key to continue . . .
```

ii) Parameterized Constructor

- Parameterized constructors are those constructors that have parameters or arguments.
- They initialize data members with different values when the objects are created.

ii) Parameterized Constructor: Sample program

```
1  //sample program to illustrate parameterized constructor
2  #include<iostream>
3  using namespace std;
4  class Box
5  {
6      float l, b, h;
7      public:
8          Box(float x, float y, float z)  //parameterized constructor
9          {
10             l=x; b=y ;h=z;
11         }
12         void Display()
13         {
14             cout<< "Length="<<l<<endl;
15             cout<< "Breadth="<<b<<endl;
16             cout<< "Height="<<h<<endl;
17         }
18
19         float showVolume()
20         {
21             return l*b*h;
22         }
23 };
```

```
25 main()
26 {
27     Box ob(10,5.5,3);
28     ob.Display();
29     cout<<"Volume="<<ob.showVolume();
30 }
```

```
Length=10
Breadth=5.5
Height=3
Volume=165
```

iii) Copy Constructor

- A copy constructor is a member function which initializes an object using another object of the same class.
- It constructs an object by copying the state from another object of the same class.
- It is used to initialize an object form another object of the same type.
- When an object is copied, another object is created and, in this process,, the copy constructor is invoked.

iii) Copy Constructor

```
1  //sample program to illustrate copy constructor
2  #include<iostream>
3  using namespace std;
4  class Box
5  {
6      float l,b,h;
7      public:
8          Box(float x, float y, float z)  //paramterized constructor
9          {
10             l=x; b=y; h=z;
11         }
12         Box( Box &b2)   //copy constructor
13         {
14             l=b2.l+10;
15             b=b2.b+10;
16             h=b2.h+10;
17         }
18         void Display()
19         {
20             cout<< "Length="<<l<<endl;
21             cout<< "Breadth="<<b<<endl;
22             cout<< "Height="<<h<<endl;
23         }
24
25         float showVolume()
26         {
27             return l*b*h;
28         }
29 };
```

```
30
31 main()
32 {
33     Box b1(5,10.0,15);
34     b1.Display();
35     cout<<"Volume of Box in cubic cm="<<b1.showVolume()<<endl;
36     Box b2(b1);
37     b2.Display();
38     cout<<"Volume of Boxin cubic mm="<<b2.showVolume()<<endl;
39
40 }
```

```
C:\Users\User\OneDrive\Documents\cc.exe
Length=5
Breadth=10
Height=15
Volume of Box in cubic cm=750
Length=15
Breadth=20
Height=25
Volume of Boxin cubic mm=7500

--------------------------------
Process exited after 0.07943 seconds with return value 0
Press any key to continue . . .
```

Constructor Overloading

- When a class has more than one constructor, then it is termed as constructor overloading.
- In C++, we can have more than one constructor in a class with same name, as long as each has a different list of arguments. This concept is known as Constructor Overloading and is quite similar to function overloading.
- Overloaded constructors essentially have the same name (name of the class) and different number of arguments.
- A constructor is called depending upon the number and type of arguments passed.
- While creating the object, arguments must be passed to let compiler know, which constructor needs to be called.

Constructor Overloading (Sample Program)

```
1   //sample program to illustrate constructor overloading
2   #include<iostream>
3   using namespace std;
4   class Box
5   {
6       int l,b,h;
7       public:
8           Box(float x, float y)   //constructor with two parameters
9           {
10              l=x; b=y;
11          }
12          Box(float x, float y,float z)   //constructor with three parameters
13          {
14              l=x; b=y ;h=z;
15          }
16
17          float DisplayArea()
18          {
19              return l*b;
20          }
21
22          float DisplayVolume()
23          {
24              return l*b*h;
25          }
26  };
```

```
28  main()
29  {
30      Box b1(10,5), b2(4,3,2);
31      cout<<"Area of base of Box= "<<b1.DisplayArea()<<endl;
32      cout<<"Volume of Box="<<b2.DisplayVolume()<<endl;
33
34  }
```

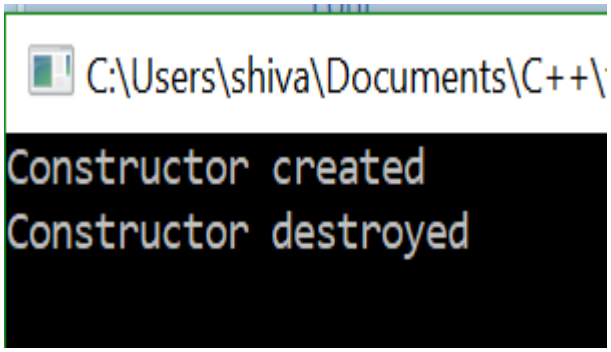
```
C:\Users\shiva\Documents\C++\constructorsample.exe
Area of base of Box= 50
Volume of Box=24

--------------------------------
Process exited after 0.0599 seconds with return value 0
Press any key to continue . . .
```

Destructor

- Destructor is the special member function which destroys or deallocates the memory space allocated by the constructors.
- When the constructor is no longer needed, the destructor destroys it (i,e, memory used by constructor).
- Due to the use of destructor, memory space is released which can be used in future.
- Therefore, the destructor is used to save memory space in the program.
- Some properties of destructor are:
  - Has same name as class, but a tilde(~) sign preceeds it.
  - Cannot take arguments
  - Does not return any value.
  - Only one destructor can be declared in a class.
  - Defined within public or protected section of class.

Destructor : Sample program

```
1   using namespace std;
2   class Rectangle
3   {
4       public:
5           Rectangle()     //constructor
6           {
7               cout<<"Constructor created"<<endl;
8           }
9
10          ~ Rectangle()   //destructor
11          {
12              cout<<"Constructor destroyed"<<endl;
13          }
14  };
15
16  main()
17  {
18      Rectangle ob;
19  }
```



Dynamic Memory Allocation

- Have you ever thought, how memory is allocated during the run time of a program?
- How can we determine the amount of memory that will be used in a program?
- Don’t worry !!!
- All this is done with the help of Dynamic Memory Allocation in C++.

Dynamic Memory Allocation

- Dynamic memory allocation in C/C++ refers to performing memory allocation manually by programmer.
- Dynamically allocated memory is allocated on Heap and non-static and local variables get memory allocated on Stack
- We use DMA technique when it is not known how much memory space is required.
- While **C** uses functions like **malloc()**, **calloc()**, **realloc()** and **free()** to handle operations based on DMA, C++ also uses all the 4 functions in addition to 2 different operators called **new and delete** to allocate memory dynamically.
- C++ supports DMA by using two operators:
  - ***new:*** to create dynamic memory
  - ***delete:*** to release allocated memory.

i) new operator

- 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 memory and initialized memory to the pointer variable.
- **Syntax**

  pointer-variable = **new** data-type;
- Eg:

```
int *p = new int;
```

i) new operator (contd.)

Initialize memory:

- We can also initialize the memory using new operator:

  pointer-variable = **new** datatype(value);

- Eg:

```
int *p = new int(25);
```

```
float *q = new float(75.25);
```

i) new operator (contd.)

Allocate block of memory:

- ***new*** operator is also used to allocate a block(an array) of memory of type *data-type*.

```
pointer-variable = new data-type[size];
```

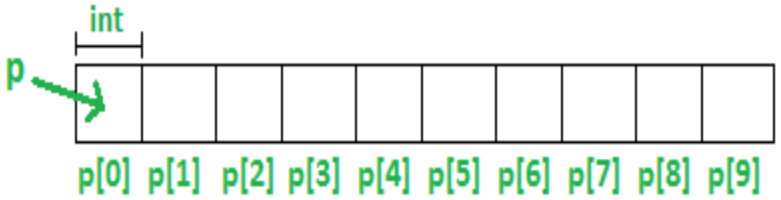
  - where size specifies the number of elements in an array.

- Eg:

```
int *p = new int[10]
```

This allocates memory for 10 integers continuously of type int and returns pointer to the first element of the sequence, which is assigned to p(a pointer).

p[0] refers to first element, p[1] refers to second element and so on.



ii) delete operator

- It is programmer's responsibility to deallocate memory.
- Once the memory is no longer needed, it should be freed so that the memory becomes available for other request of dynamic memory.
- We use the "delete" operator in C++ for dynamic memory deallocation.
- **Syntax:**

  delete pointer-variable;
- Eg:

  delete p;
- To free the dynamically allocated array pointed by pointer-variable, use following form of *delete*:
- Syntax:

  delete[ ] pointer_variable
- Eg:

  delete[] p;
- Also can be written as:

  delete p[];

DMA : Sample Program

```
Value of p: 29
Value of r: 75.25
Value store in block of memory: 10 11 12 13 14
--------------------------------
```

```cpp
#include <iostream>
using namespace std;
main ()
{
    int i;
    int* p = NULL;      // Pointer initialization to null
    p = new int;        // Request memory for the variable using new operator
    *p = 29;            // Store value at allocated address
    cout << "Value of p: " << *p << endl;

    float *r = new float(75.25);     // Initialization of memory using new operator
    cout << "Value of r: " << *r << endl;

    int *q = new int[5];     // Request block of memory of size 5
    for (i = 0; i < 5; i++)
        {
         q[i] = i+10;
        }

    cout << "Value store in block of memory: ";
    for (int i = 0; i < 5; i++)
        {
        cout << q[i] << " ";
        }

    delete p;     // freed the allocated memory
    delete r;    // freed the allocated memory
    delete[] q;      // freed the block of allocated memory
}
```

Dynamic Constructor

- Dynamic constructor is used to allocate the memory to the objects at the run time.
- Memory is allocated at run time with the help of 'new' operator.
- By using this constructor, we can dynamically initialize the objects.

Dynamic Constructor: Sample program

```
#include<iostream>
using namespace std;
class Test
{
    int *ptr;
    public:
        Test()
        {
            ptr=new int();
            *ptr=100;
        }

        Test(int v)
        {
            ptr=new int;
            *ptr=v;
        }

        int Display()
        {
            return (*ptr);
        }
};
```

```
main()
{
    Test ob1;
    Test ob2(200);
    cout<<"Value of First pointer="<<ob1.Display()<<endl;
    cout<<"Value of Second pointer="<<ob2.Display()<<endl;
}
```

```
Value of First pointer=100
Value of Second pointer=200
```

Scope resolution operator

- The scope resolution operator is used to reference the global variable or member function that is out of scope.
- Therefore, we use the scope resolution operator to access the hidden variable or function of a program.
- The operator is represented as the double colon (::) symbol.
- Syntax:

  :: variableName or functionName;

- Eg:

```
:: x ;
void Box :: calculate();
```

Application of Scope Resolution Operator

It is used for following purposes:

- **To access a global variable when there is a local variable with same name.**
- **To define a function outside a class.**
- **To access a class's static variables.**
- **In case of multiple Inheritance**

To access a global variable when there is a local variable with same name.

```
1  //program to access global variable
2  #include<iostream>
3  using namespace std;
4  int x=20;  // Global x
5  int main()
6  {
7      int x = 10;   // Local x
8     cout << "Value of global x is " << ::x;
9     cout << "\nValue of local x is " << x;
10    return 0;
11 }
```

```
Value of global x is 20
Value of local x is 10
--------------------------------
Process exited after 0.1329 seconds with return value 0
Press any key to continue . . .
```

Sample programs

WAP to create a class Rectangle that has two data members: length and breadth. Use constructor to initialize the data member of a class and then calculate the area of rectangle.

```cpp
#include<iostream>
using namespace std;
class Rectangle
{
    float l,b;
    public:
        Rectangle(float x, float y)
        {
            l=x;
            b=y;
        }

        void Display_member()
        {
            cout<<"Length is : "<<l<<endl;
            cout<<"Breadth is : "<<b<<endl;
        }

        float Get_area()
        {
            return l*b;
        }
};
```

```cpp
main()
{
    Rectangle r(10,8);
    float area;
    r.Display_member();
    area=r.Get_area();
    cout<<"Area is : "<<area<<endl;
}
```

```
Length is : 10
Breadth is : 8
Area is : 80
```

WAP to create a class Time that has data members: hours, minutes and seconds. Use constructor to initialize the data members. Also calculate the total given time in seconds.

```
#include<iostream>
using namespace std;
class Time
{
    float h,m,s;
    public:
        Time(float x, float y,float z)
        {
            h=x;
            m=y;
            s=z;
        }

        void display_member()
        {
            cout<<"Hour is : "<<h<<endl;
            cout<<"Minute is : "<<m<<endl;
            cout<<"Second is : "<<s<<endl;
        }

        float get_second()
        {
            return ((h*3600)+(m*60)+s);
        }
};
```

```
main()
{
    Time t(2,50,30);
    float second;
    t.display_member();
    second=t.get_second();
    cout<<"Total seconds is : "<<second<<endl;
}
```

```
Hour is : 2
Minute is : 50
Second is : 30
Total seconds is : 10230

--------------------------------
```

WAP to create a class *Area* having two constructors. One constructor receives two arguments and another constructor receives one argument. Calculate the area of rectangle, area of square and display them.

```
#include<iostream>
using namespace std;
class Area
{
    float length, breadth, length_sq;
    public:
        Area(){}

        Area(float a, float b)
        {
            length=a;
            breadth=b;
        }

        Area(float c)
        {
            length_sq=c;
        }
        void display_area_rec()
        {
            cout<< "Area of Rectangle="<<length*breadth<<endl;
        }
        void display_area_sq()
        {
            cout<< "Area of Square="<<length_sq*length_sq<<endl;
        }
};

main()
{
    Area a1(25,20);
    Area a2(22);
    a1.display_area_rec();
    a2.display_area_sq();
}
```

```
Area of Rectangle=500
Area of Square=484
```

WAP to read number of students and then marks of each student. Display entered marks and their average. Use DMA to reserve memory to store marks of each student.

```
#include<iostream>
using namespace std;
class Exam
{
    int n,i;
    float *ptr,avg,sum;
    public:
        void getnum()
        {
            cout<<"Enter number of students"<<endl;
            cin>>n;
        }

        void getmarks()
        {
            ptr=new float[n];
            cout<<"Enter marks of each student"<<endl;
            for(i=0;i<n;i++)
            {
                cin>>*(ptr+i);
                sum+=*(ptr+i);
            }
        }
        void showmarks()
        {
            cout<<"Students marks are:"<<endl;
            for(i=0;i<n;i++)
            {
                cout<< *(ptr+i)<<endl;
            }
        }

        void showaverage()
        {
            avg=sum/n;
            cout<<"Average of marks:"<<avg<<endl;
        }
};

main()
{
    Exam e;
    e.getnum();
    e.getmarks();
    e.showmarks();
    e.showaverage();
}
```

```
Enter number of students
5
Enter marks of each student
60
60
80
80
100
Students marks are:
60
60
80
80
100
Average of marks:76
```

2.14. Inline Function

- Inline functions are those functions declared with keyword ***inline***, which contains small number of codes inside its function.
- The main purpose of inline function is to speed up the program execution.
- To save execution time in short functions, the code in function body is made directly inline with the code in the calling program.
- When the function is called, the actual code from the function is inserted or substituted instead of a jump to the function.
- This substation is performed by the C++ compiler at compile time. However, inlining is only a request to the compiler, not a command.
- Compiler can ignore the request of inlinig.

2.14. Inline Function (Contd.)

Syntax:

```
inline return-type function-name(parameters)
{
    // function code
}
```

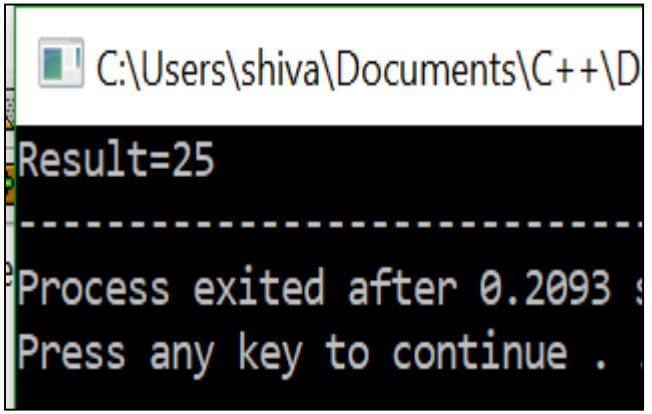
Example:

```
inline int display(int x)
   {
      n=x;
      return n*n;
   }
```

Sample program 2.17

Illustration of Inline function

```
1  #include<iostream>
2  using namespace std;
3  class Test
4  {
5     int n;
6     public:
7        inline int display(int x)  // inline function
8        {
9           n=x;
10          return n*n;
11       }
12 };
13
14 main()
15 {
16    Test ob;
17    cout<<"Result="<<ob.display(5);
18 }
```



2.10. Static Data Member

- It is declared with keyword **static**.
- All the objects of the class share it as a common variable.
- It has its lifetime throughout the entire program.
- It is initialized to zero when the first object of the class is created. No other initialization is permitted.

2.10. Static Data Member (Contd.)

- **Declaration**

```
static data_type member_name;
```

- **Defining the static data member**
  It should be defined outside of the class following this syntax:

```
data_type class_name :: member_name =value;
```

Sample program 2.10

Illustration of Static data member

```
1  #include<iostream>
2  using namespace std;
3  class Demo
4  {
5     static int ctr;    //static data member declaration
6     public:
7        void count()
8        {
9           cout<< "Count No:"<<ctr<<endl;
10          ctr++;
11       }
12 };
13 int Demo::ctr; //static data member definition
```

```
15 main()
16 {
17    Demo ob1,ob2,ob3;
18    ob1.count();
19    ob2.count();
20    ob3.count();
21 }
```

```
Count No:0
Count No:1
Count No:2
```

2.11. Static Function Member

- A static member function is a special member function, which is used to access only static data members, any other normal data member cannot be accessed through static member function.
- Just like static data member, static member function is also a class function; it is not associated with any class object.
- A static member function is always declared with the ***static*** keyword.
- It has its lifetime throughout the entire program.

2.11. Static Function Member

- **Syntax:**

```
static return_type function_name(argument_list)
{
.....
}
```

- We ss name, by using following syntax:

```
class_name:: function_name(perameter);
```

Sample program 2.11

```
Initial Value of num:0
Final Value of num: 3
```

```cpp
#include<iostream>
using namespace std;
class Demo
{
   static int num;   //static data member declaration
   public:
      static int Display()
      {
         return num;
      }
      static int Increase()
      {
         num++;
      }
};
int Demo::num; //static data member definition
```

```cpp
main()
{
   cout<<"Initial Value of num:"<<Demo::Display()<<endl;
   Demo ob1,ob2,ob3;
   Demo::Increase();
   Demo::Increase();
   Demo::Increase();
   cout<<"Final Value of num: "<<Demo::Display()<<endl;
}
```

2.12. Friend Function

- A friend function is a function, declared with the keyword ***friend***, which can access Private or Protected data members of a class from outside the class in which it is declared.
- If a function is defined as a friend function in C++, then the protected and private data of a class can be accessed using the function.
- By using the keyword **friend,** compiler knows the given function is a friend function.
- For accessing the data, the declaration of a friend function should be done inside the body of a class starting with the keyword friend.
- Friend function of the class is declared inside the class but is defined outside the class but it isn't the member function of the class.

2.12. Friend Function

- A friend function is a function, declared with the keyword ***friend***, which can access Private or Protected data members of a class from outside the class in which it is declared.
- If a function is defined as a friend function in C++, then the protected and private data of a class can be accessed using the function.
- By using the keyword **friend,** compiler knows the given function is a friend function.
- For accessing the data, the declaration of a friend function should be done inside the body of a class starting with the keyword friend.
- Friend function of the class is declared inside the class but is defined outside the class but it isn't the member function of the class.

2.12 Declaration of friend function

```
class class_name
{
        friend return_type function_name(arguments);    // friend function declaration
};
```

Merits and Demerits of Friend Function

Merits

1. It acts as the bridge between two classes by operating on their private data's.
2. It is able to access members without need of inheriting the class.
3. It can be used to increase the versatility of overloading operator.
4. It provides functions that need data which isn't normally used by the class.
5. Allows sharing private class information by a non-member function.

Demerits

1. It violates the law of data hiding by allowing access to private members of the class from outside the class.
2. Breach of data integrity.
3. Conceptually messy
4. Runtime polymorphism in the member cannot be done.
5. Size of memory occupied by objects will be maximum

Sample Program 2.12

Program with Friend function

```
#include<iostream>
using namespace std;
class DemoFriend
{
    int n;
    public:
        void getdata()
        {
            cout<<"Enter an Integer:"<<endl;
            cin>>n;
        }
        friend void FindSquare(DemoFriend); // friend function declaration
};
void FindSquare(DemoFriend d)     //Function outside the class
{
    int sq;
    sq = d.n*d.n;  //accessing private data form non-member function
    cout<<"Square is: "<<sq;
}
```

```
main()
{
    DemoFriend ob;
    ob.getdata();
    FindSquare(ob);
}
```

```
Enter an Integer:
5
Square is: 25
--------------------
```

Sample Program 2.13:

Create class called *One* and *Two* with each having one private member. Add member function to set a value (say *setValue)* on each class. Add one more function *max()* that is friendly to both classes. *max()* function should compare two private member of two classes and show maximum among them. Create one-one object of each class. Then set a value to them. Display the maximum number among them. [PU 2016 Fall, 2015 Fall]

```
include<iostream>
using namespace std;
class Two;  //class declaration
class One
{
   int x;
   public:
      void setValue(int i)
      {
         x=i;
      }

      friend void max(One, Two);
};
class Two
{
   int y;
   public:
      void setValue(int i)
      {
         y=i;
      }

      friend void max(One, Two);
};
```

```
void max(One m, Two n)
{
   if (m.x>=n.y)
      cout<<"Maximum number is: "<<m.x<<endl;
   else
      cout<<"Maximum number is: "<<n.y<<endl;
}
main()
{
   One ob1;
   Two ob2;
   ob1.setValue(15);
   ob2.setValue(12);
   max(ob1,ob2);
}
```

```
Maximum number is: 15

--------------------------------
Process exited after 0.08307 seconds with return value 0
Press any key to continue . . .
```

Friend Class

- A friend class is a technique in OOP, by which a class can access to all data and function members of another class.
- For e.g, if a class ABC declares another class XYZ as its friend, then XYZ can access all the data and function members of class ABC.
- Friendship is not mutual. If a class A is friend of B, then B doesn't become friend of A automatically.

- **Friend class declaration:**

```
class class_name
{
    friend class friend_class;// declaring friend class
};

class friend_class
{

};
```

Sample program 2.14

Illustration of Friend class

```
1  #include<iostream>
2  using namespace std;
3  class ABC
4  {
5     int a,b, c=10;
6     public:
7        int getdata()
8        {
9           cout<<"Enter two Integers:"<<endl;
10          cin>>a>>b;
11       }
12       friend class XYZ; // friend class declaratio
13 };
14 class XYZ
15 {
16    public:
17       void display(ABC m)
18       {
19          cout<<"First number is:"<<m.a<<endl;
20          cout<<"Second number is:"<<m.b<<endl;
21          cout<<"Value of c is:"<<m.c<<endl;
22       }
23 };
```

```
24 main()
25 {
26    ABC ob1;
27    XYZ ob2;
28    ob1.getdata();
29    ob2.display(ob1);
30 }
```

```
Enter two Integers:
4
6
First number is:4
Second number is:6
Value of c is:10

--------------------------
```

Sample program 2.15

Create two classes *One* and *Two* each has one private members and a function member *getdata()*. Make both the classes as friend class. and display both the data by both the classes. Make necessary functions and objects as required to support your program.

```cpp
#include<iostream>
using namespace std;
class Two;
class One
{
    int a;
    public:
        void getdata1()
        {
            cout<<"enter value of a"<<endl;
            cin>>a;
        }
        friend class Two; // friend class declaration
        void display (Two);

};
class Two
{
    int b;
    public:
        void getdata2()
        {
            cout<<"enter value of b"<<endl;
            cin>>b;
        }
        friend class One;
        void display(One);

};
```

```cpp
void One::display(Two t1)
    {
        cout<<"Displaying through class One: a="<<a<<"b="<<t1.b<<endl
    }
void Two::display(One ob)
    {
        cout<<"Displaying through class Two: a="<<ob.a<<"b="<<b<<endl
    }
```

```cpp
main()
{
    One ob1;
    Two ob2;
    ob1.getdata1();
    ob2.getdata2();
    ob1.display(ob2);
    ob2.display(ob1);
}
```

```
enter value of a
5
enter value of b
10
Displaying through class One:
a=5 b=10
Displaying through class Two:
a=5 b=10
```

Create a class called One and Two with each having a private member. Add member function to set value (sayset_value) on each class Add one more function max() that is friendly to both classes. max() function should compare two private members of two classes and show maximum among them. Create one object of each class then set a value on them. Display the maximum among them.

```
#include<iostream>
using namespace std;
class Two;
class One
{
    int x;
    public:
        void setvalue(int i)
        {
            x=i;
        }

        friend void max(One, Two);
};

class Two
{
    int a;
    public:
        void setvalue(int i)
        {
            a=i;
        }
        friend void max(One, Two);
};
```

```
void max(One m, Two n)
{
    if (m.x>=n.a)
        cout<<"Maximum number is: "<<m.x<<endl;
    else
        cout<<"Maximum number is: "<<n.a<<endl;
}

main()
{
    One ob1;
    Two ob2;
    ob1.setvalue(10);
    ob2.setvalue(12);
    max(ob1,ob2);
}
```

```
Maximum number is: 12
```

WAP to add two complex number using concept of constructor.

```
#include<iostream>
using namespace std;
class Complex
{
    float x,y;
    public:
        Complex(){}

        Complex(float real, float imag)
        {
            x=real;
            y=imag;
        }
        friend Complex sum(Complex, Complex);
        friend void show(Complex);
};
```

```

Complex sum(Complex c1, Complex c2)
{
    Complex temp;
    temp.x=c1.x+c2.x;
    temp.y=c1.y+c2.y;
    return temp;
}

void show(Complex c4)
{
    cout<<c4.x<<"c+j"<<c4.y<<endl;
}

```

```
main()
{
    Complex A(2.5,3.5);
    Complex B(3.5,4.6);
    show(A);
    show(B);
    Complex C;
    C=sum(A,B);
    show(C);
}
```

```
2.5c+j3.5
3.5c+j4.6
6c+j8.1
```

Sample program 2.16

Write base class that ask the user to enter Time (hour, minute and second) and derived class adds the Time of its own with the base. Finally make third class that is friend of derived and calculate the difference of base class time and its own time.

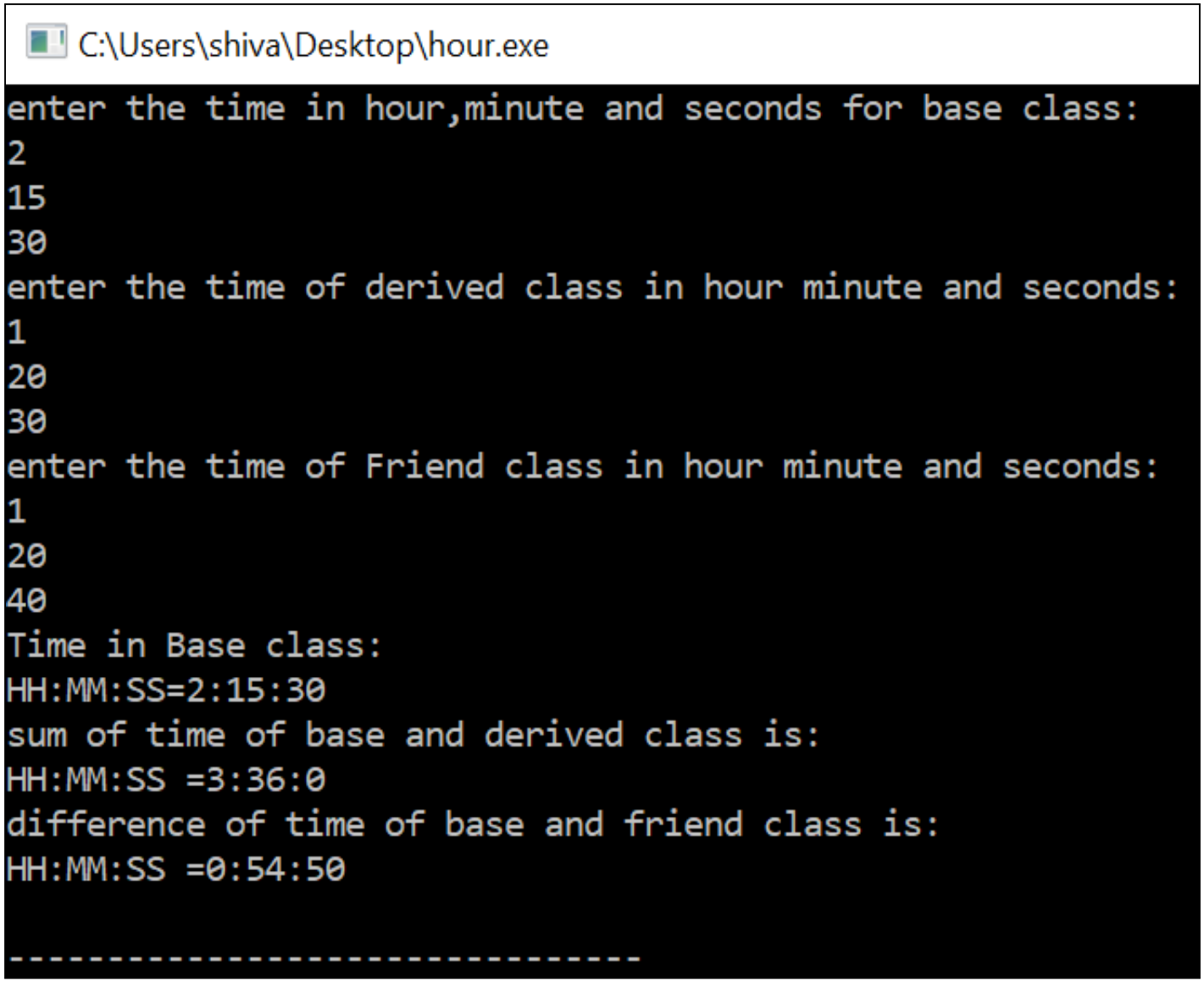
```
#include<iostream>
#include<math.h>
using namespace std;
class Time
{
   protected:
   int h,m,s,tot_sec1;
   public:
         void gettime1()
         {
            cout<<"enter the time in hour,minute and seconds for base class:"<<endl;
            cin>>h>>m>>s;
            tot_sec1=(h*3600)+(m*60)+s;
         }
         void display1()
         {
            cout<<"Time in Base class:"<<endl;
            cout<<"HH:MM:SS="<<h<<":"<<m<<":"<<s<<endl;
         }
   };
```

```
21  class Add:public Time
22  {
23     int x,y,z,sum,tot_sec2;
24     int h2,m2,s2;
25     public:
26        void gettime2()
27        {
28           cout<<"enter the time of derived class in hour minute and seconds:"<<endl;
29           cin>>x>>y>>z;
30           tot_sec2=(x*3600)+(y*60)+z;
31        }
32
33        void displaysum()
34        {
35           sum=tot_sec1+tot_sec2;
36           h2=sum/3600;
37           sum=sum%3600;
38           m2=sum/60;
39           sum=sum%60;
40           s2=sum;
41           cout<<"sum of time of base and derived class is:"<<endl;
42           cout<<"HH:MM:SS ="<<h2<<":"<<m2<<":"<<s2<<endl;
43        }
44        friend class Subtract;
45  };
```

```
47  class Subtract
48  {
49     int diff, h3, m3, s3, a, b, c, tot_sec3;
50     public:
51        void gettime3()
52        {
53           cout<<"enter the time of Friend class in hour minute and seconds:"<<endl;
54           cin>>a>>b>>c;
55           tot_sec3=(a*3600)+(b*60)+c;
56        }
57        void displaydiff(Add d)
58        {
59           diff=d.tot_sec1-tot_sec3;
60           diff=abs(diff);      //gives absolute value
61           h3=diff/3600;
62           diff=diff%3600;
63           m3=diff/60;
64           diff=diff%60;
65           s3=diff;
66           cout<<"difference of time of base and friend class is:"<<endl;
67           cout<<"HH:MM:SS ="<<h3<<":"<<m3<<":"<<s3<<endl;
68        }
69  };
```

```
70  main()
71  {
72     Add ad;
73     Subtract sb;
74     ad.gettime1();
75     ad.gettime2();
76     sb.gettime3();
77     ad.display1();
78     ad.displaysum();
79     sb.displaydiff(ad);
80  }
```

Output:

- 
C:\Users\shiva\Desktop\hour.exe
enter the time in hour,minute and seconds for base class:
2
15
30
enter the time of derived class in hour minute and seconds:
1
20
30
enter the time of Friend class in hour minute and seconds:
1
20
40
Time in Base class:
HH:MM:SS=2:15:30
sum of time of base and derived class is:
HH:MM:SS =3:36:0
difference of time of base and friend class is:
HH:MM:SS =0:54:50

Pointer

- A pointer variable in C++ is used to store the memory address of another variable. Pointers allow indirect access to variables and enable dynamic memory allocation and deallocation.
- **Declaration:** A pointer variable is declared using the '*' symbol before the variable name.

```
datatype *pointer_variable;
```

Pointer

Initialization:

- Pointers can be initialized with the memory address of an existing variable using the '&' operator.

Syntax: datatype variable;

datatype *pointer_variable = &variable;

Accessing Value:

- To access the value of the variable that a pointer is pointing to, the pointer needs to be dereferenced using the '*' operator.

Syntax: *pointer_variable;

Reference Variable

- A reference variable is an alias, that is, another name for an already existing variable.
- Once a reference is initialized with a variable, either the variable name or the reference name may be used to refer to the variable.
- When a variable is declared as a reference, it becomes an alternative name for an existing variable.
- A variable can be declared as a reference by putting '&' in the declaration.
- A reference variable must be initialized at the time of declaration.

Reference Variable

- **Declaration**
- ```
  [data_type]   &[reference_variable]=[regular_variable];
  ```
  - *regular_variable* is a variable that has already initialized,
  - *reference_variable* is an alternative name (alias) to represent the variable *regular_variable*
  ```

Sample program

Illustration of Reference variable

```
value of a :10
value of b :10

AFTER ADDING 10 INTO REFERENCE VARIABLE
value of a :20
value of b :20
```

```
1  #include<iostream>
2  using namespace std;
3  main()
4  {
5      int a=10;
6      int &b=a;    // reference variable
7      cout<< " value of a :"<< a << endl;
8      cout<< " value of b :"<< b << endl;
9      b=b+10;
10     cout<<"\nAFTER ADDING 10 INTO REFERENCE VARIABLE \n";
11     cout<< " value of a :"<< a << endl;
12     cout<< " value of b :"<< b << endl;
13 }
```

Default Arguments

- A default argument is a value provided in a function declaration that is automatically assigned by the compiler if the caller of the function doesn't provide a value for the argument with a default value.
- We can specify default values i.e. some fixed values for arguments in function so that when such function is called without specifying all its arguments, the C++ function assigns the default value to the parameter which does not have a matching argument in the function call.
- If the function is called with explicit value for default argument, the supplied value is used for that argument.
- But when the value is not passed for the default argument, the default value is used for that argument.
- Thus, defining default argument means like making optional argument. The default values are specified in function prototypes.

Default Arguments

- Syntax:

```
Return_type function_name ( ……..,   ,  data_type1 =default value1, dataype2 =default_value2);
```

Sample program

Illustration of default arguments.

```
1  #include<iostream>
2  using namespace std;
3  class Test
4  {
5     public:
6        float Interest (float p, float t, float r=20)   //default argument declaration
7        {
8           return(p*t*r/100);
9        }
10 };
11
12 main()
13 {
14    Test ob;
15    float si1, si2;
16    si1=ob.Interest(1000,2,30);
17    si2=ob.Interest(1000,2);
18    cout<< " Simple Interest with 30% rate :"<< si1 << endl;
19    cout<< " Simple Interest with default 20% rate :"<< si2 << endl;
20 }
```

```
Simple Interest with 30% rate :600
Simple Interest with default 20% rate :400


--------------------------------
```

End of chapter