UNIT II

OVERVIEW OF C++

UNITII OVERVIEW OF C++

- **Topics to be discussed,**

➢Classes and Objects

➢Constructors and Destructors

➢Operator Overloading and Type Conversions

➢Function object

➢Dynamic Memory Management

UNITII OVERVIEW OF C++

- **Topics to be discussed,**

➢Classes and Objects

➢Constructors and Destructors

➢Operator Overloading and Type Conversions

➢Function object

➢Dynamic Memory Management

C++ Classes and Objects

- In object-oriented programming languages like C++, the data and functions (procedures to manipulate the data) are bundled together as a self-contained unit called an object.
- *This programming paradigm is known* ***as object-oriented programming*****.**
- A class is an extended concept similar to that of structure in C programming language; which describes the data properties alone.
- In C++, a class describes both the properties (data) and behaviors (functions) of objects.
- Classes are not objects, but they are used to instantiate objects.
- But before we can create objects and use them in C++, we first need to learn about classes.

C++ Classes and Objects –Cont'd

- **C++ Class:**
  - A class in C++ is the **building block**, that leads to Object-Oriented programming.
  - A C++ class is like a **blueprint for an object**.
  - A class is basically a **logical entity** and mainly a **collection of objects.**
  - It *is similar to structures in C language*.
  - Class can also be defined as **user defined data type** but **it also contains data members and member functions.**
  - *To access the class members, we use an instance of the class*.
  - It declare & defines what data variables the object will have and what operations can be performed on the class's object.
  - The Class representation of objects and the sets of operations that can be applied to such objects

C++ Classes and Objects –Cont'd

- **Create a Class/Class Declaration**
  - The **class declaration describes the type and scope of its members**.
  - The class declaration is *similar to a struct declaration*.
  - A class is defined in C++ using **keyword class followed by the name of the class**.
  - The keyword class specifies, that what follows is an abstract data of type class name.
  - The body of the class is defined inside the curly brackets and terminated by a semicolon at the end.
- The general form of declaring a class is

```
class class_name {
        private:
                variable declarations ;
                function declarations ;
        public:
                variable declarations ;
                function declarations ;
} ;
```

Example:

```
class Rectangle
{
  private:
     double length;
     double breadth;
  public:
     double calculateArea()
   {
        return length * breadth;
     }
};
```

C++ Classes and Objects –Cont'd

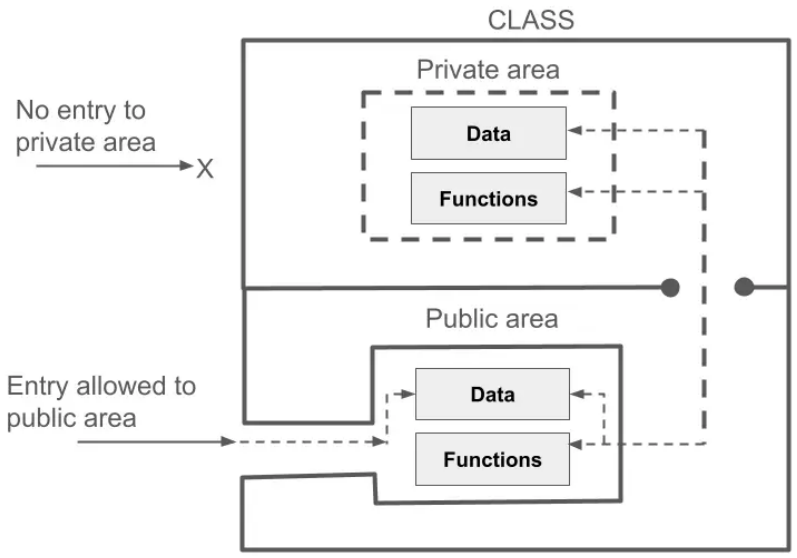
- The class body contains the declaration of variables and functions.
- These **functions and variables are collectively called class members**.
- They are usually grouped under two sections, namely, private and public to denote which of the members are private and which of them are public.
- **The keywords private and public are known as visibility labels**.
- Note that these keywords are followed by a colon

C++ Classes and Objects –Cont'd

- The keyword **public / private / protected** determines the access attributes of the members of the class that follow it.
- ***A private member cannot be directly accessed by the object*** and in order to access the data members, we have to define member functions which generally have public access specifier.
- We can also define the data members and member functions as public, private or protected.

C++ Classes and Objects –Cont'd

Data hiding in classes



- The class members that have been declared as private can be accessed only from within the class.
- On the other hand, public members can be accessed from outside the class also.
- The data hiding (using private declaration) is the key feature of object-oriented programming.
- The use of the keyword private is optional.
- **By default, the members of a class are private.**
- If both the labels are missing, then, by default, all the members are private.
- Such a class is completely hidden from the outside world and does not serve any purpose.

C++ Classes and Objects –Cont'd

```
class Test
{
   private:
      int a;
      void function1() { }
   public:
      int b;
      void function2() { }
}
```

- Here, **a and function1() are private** so that they **cannot be accessed from outside the class**.
- On the other hand, b and function2() are accessible from everywhere in the program.

- **Define Data Member and Member Function:**
  - The variables declared inside the class are known as **data members** It may be private or public. the functions are known as **member functions**.
  - **The private members cannot be accessed directly from outside of the class.**
  - The private data of class can be accessed only by the member functions of that class.
  - The public member can be accessed outside the class from the main function.

C++ Classes and Objects –Cont'd

Example:

```
class xyz
{
      int x ;
      int y ;
   public:
      int z ;
} ;

void main( )
{
 - - - - - - -
 - - - - - - -
 xyz p ;
 p.z=10 ; // ok, z is public
 p.x=0 ; // error, x is private
  - - - - - -
 - - - - - -
}
```

C++ Classes and Objects –Cont'd

•C++ Objects

- **An Object is an instance of a Class**.
- When a class is defined, no memory is allocated but when it is instantiated (i.e. an object is created) memory is allocated.
- In C++, **Object is a real world entity**, for example, chair, car, pen, mobile, laptop etc.
- In other words, **object is an entity that has state and behavior**.
- Here, **state means data and behavior means functionality**.
- Object is a runtime entity, it is created at runtime.
- **All the members of the class can be accessed through object**.
- To use the data and access functions defined in the class, we need to create objects.

C++ Classes and Objects –Cont'd

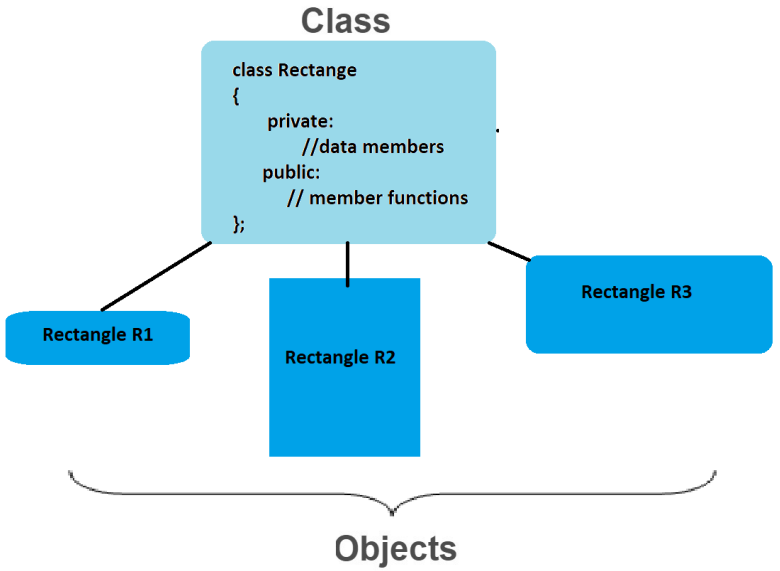
- We declare objects of a class with exactly the same sort of declaration that we declare variables of basic types
- **Syntax to Define Object:**

  className objectVariableName;
  - The class-name is the name of the class from which an object is to be created.
  - The object-name is the name to be assigned to the new object.
- **Example:**

  Rectangle R1;
- **we can also create multiple objects of a single class,** because the objects have different attribute/property values.

Example:

Rectangle R1,R2,R3;



C++ Classes and Objects –Cont'd

- **Define Data Member and Member Function:**
- The data member of a class is declared within the body of the class but **member function** can be defined in two ways
  - Inside the class definition
  - Outside the class definition

C++ Classes and Objects –Cont'd

Syntax:

```
class  class_name
{
   public:
   return type member function (arg.....)
    {
   // function body
    }
} ;
```

- **Member function defined Inside the class definition**
  - Normally, only small functions are defined inside the class definition.
  - Defining a member function is to replace the function declaration with the actual function definition inside the class.
  - When a function is defined inside a class, it is treated as an inline function.
  - Therefore, all the restrictions and limitations that apply to an inline function are also applied here.

C++ Classes and Objects –Cont'd

Syntax:

```
class class_name
{
    // ---------
   public:
   return type member function(arg);
   //----------
};
return type class_name :: member function(arg)
{
   // function body
}
```

- **Member function defined Outside the class definition:**
  - *Member functions that are **declared inside** a **class have to be defined separately outside the class***.
  - Their definitions are very much like normal functions.
  - They should have a function header and a function body.
  - If the member function is defined inside the class definition it can be defined directly
  - But if its defined outside the class, then we have to use the scope resolution **:** **:** operator along with class name along with function name.

C++ Classes and Objects –Cont'd

- **Create object and access Member Function:**
  - The main function for both the function definition will be same.
  - Inside main() we will create object of class, and will call the member function using dot **.** operator.
  - ***Syntax of accessing data member is:***

    object_name . data_member
  - ***Syntax of accessing member function:***

    object_name . function_name

Member function defined Inside the class definition - Example

```
#include<iostream>
using namespace std;
class item
{
      int qty;
      int cost;
   public:
      void getdata(int q, int c)
       {
          qty = q;
          cost = c;
       }
   void display()
   {
      cout<<"\nQuantity:"<<qty<<"\nCost:"<<cost;
      int total=qty*cost;
      cout<<"\nTotal cost="<<total<<endl;
   }
};
```

```
int main()
{
   item it1,it2;
   cout<<"\nItem1:";
   it1.getdata(10,11);
   it1.display();
   cout<<"\nItem2:";
   it2.getdata(10,8);
   it2.display();
}
```

Output:

```
Item1:
Quantity:10
Cost:11
Total cost=110

Item2:
Quantity:10
Cost:8
Total cost=80
```

Member function defined Outside the class definition - Example

```
#include<iostream>
using namespace std;
class College
{
   private:
      string name;
      string city;
   public:
      void getdata( );
      void displaydata();
};
void College::getdata()
{
   cout<<"\nEnter the college name:";
   cin>>name;
   cout<<"\nEnter the city name:";
   cin>>city;
}
```

```
void College:: displaydata()
{
   cout<<"\nCollege name:"<<name;
   cout<<"\nCity name:"<<city;
}
int main()
 {
   College c1;
   c1.getdata();
   c1.displaydata();
}
```

Output:

```
Enter the college name:MIT

Enter the city name:Chennai

College name:MIT
City name:Chennai
```

C++ Classes and Objects –Cont'd

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

C++ Classes and Objects –Cont'd

- One of the objectives of OOP is to separate the details of implementation from the class definition.
- It is therefore good practice to define the member functions outside the class.
- We can define a member function outside the class definition and still make it inline by just using the qualifier inline in the header line of the function definition..
- It uses keyword **' inline '**
- ***The syntax for defining the function inline:***

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

Making outer function inline - Example

```
#include<iostream>
using namespace std;
class item
{
      int qty;
      int cost;
   public:
      void getdata(int q, int c);
      void display();
};
inline void item::getdata(int q, int c)
      {
         qty = q;
         cost = c;
      }
inline void item::display()
   {
      cout<<"\nQuantity:"<<qty<<"\nCost:"<<cost;
      int total=qty*cost;
      cout<<"\nTotal cost="<<total<<endl;
   }
```

```
int main()
{
   item it1,it2;
   cout<<"\nItem1:";
   it1.getdata(10,11);
   it1.display();
   cout<<"\nItem2:";
   it2.getdata(10,8);
   it2.display();
}
```

Output:

```
Item1:
Quantity:10
Cost:11
Total cost=110

Item2:
Quantity:10
Cost:8
Total cost=80
```

C++ Classes and Objects –Cont'd

- **Array with in Class:**
  - C++ provides the array, which stores a fixed-size sequential collection of elements of the same type.
  - An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.
  - Array can be declared as the member of a class.
  - Arrays can be declared as private, public or protected members of class.
- **Declaring Arrays**
  - To declare an array in C++, the programmer specifies the type of the elements and the number of elements required by an array as follows

  type arrayName [arraySize];

Example of Array as a class member

```
#include<iostream>
using namespace std;
const int size=5;
class student
{
   int roll_no;
   int marks[size];
   public:
      void getdata ();
      void tot_marks ();
} ;
 void student :: getdata ()
 {
   cout<<"\nEnter roll no: ";
   cin>>roll_no;
   for(int i=0; i<size; i++)
   {
      cout<<"Enter marks in the subject "<<(i+1)<<": ";
      cin>>marks[i] ;
   }
}
```

```
void student :: tot_marks() //calculating total marks
{
   int total=0;
   for(int i=0; i<size; i++)
      total+= marks[i];
   cout<<"\nTotal marks "<<total;
}
int main()
{
      student stu;
      stu.getdata() ;
      stu.tot_marks() ;
}
```

Output:

```
Enter roll no: 111
Enter marks in the subject 1: 99
Enter marks in the subject 2: 95
Enter marks in the subject 3: 100
Enter marks in the subject 4: 89
Enter marks in the subject 5: 99

Total marks 482
```

C++ Classes and Objects –Cont'd

- **Array of Objects**
  - We know that an array can be of any data type including structure
  - Similarly, we can also have arrays of variables that are of the type class.
  - Such variable are called arrays of objects.
  - Consider the following class definition:

```
class employee
{
        char name[30];
        float age;
 public:
        void getdata(void);
        void putdata(void);
};
```

•**The identifier employee is a user-defined data type and can be used to create objects** that relate to different categories of the employees.

- •employee manager[3];
- •employee foreman[15];
- •employee worker[75];

Array of Objects - Example

```
#include<iostream>
using namespace std;
class Employee
{
   int id;
   char name[30];
   public:
      void getData();
      void putData();
};
void Employee::getData()
{
   cout << "Enter Id : ";
   cin >> id;
   cout << "Enter Name : ";
   cin >> name;
}
void Employee::putData()
{
   cout <<"ID:"<< id << " ";
   cout << "NAME:"<<name << "\n ";
}
```

```
int main()
{
   Employee emp[10];
   for( int i=0; i<5; i++ )
   {
      cout << "Employee:" << i + 1 << endl;
      emp[i].getData();
   }
   for( int i=0; i<5; i++ )
   {
      cout << "Employee:" << i + 1<<" ";
      emp[i].putData();
   }
   return 0;
}
```

Output:

```
Employee:1
Enter Id : 111
Enter Name : Balu
Employee:2
Enter Id : 222
Enter Name : Devi
Employee:3
Enter Id : 333
Enter Name : Ajay
Employee:4
Enter Id : 444
Enter Name : Akash
Employee:5
Enter Id : 555
Enter Name : Swetha
Employee:1 ID:111 NAME:Balu
 Employee:2 ID:222 NAME:Devi
 Employee:3 ID:333 NAME:Ajay
 Employee:4 ID:444 NAME:Akash
 Employee:5 ID:555 NAME:Swetha
```

C++ Classes and Objects –Cont'd

- **Passing and Returning Objects in C++:**
  - In C++ we can pass class's objects as arguments and also return them from a function the same way we pass and return other variables.
  - No special keyword or header file is required to do so.
- **Passing an Object as argument**
  - To pass an object as an argument we write the object name as the argument while calling the function the same way we do it for other variables.

  Syntax:

  function_name(object_name);
- **Returning Object as argument**

  Syntax:

  object = return object_name;

Passing an Object as argument - Example

```
#include <iostream>
using namespace std;
class Number
{
   public:
      int n;
      // This function will take
      // an object as an argument
      void add(Number N)
      {
         n = n + N.n;
      }
};
```

```
int main()
{
   Number N1,N2;
   N1.n = 25;
   N2.n = 100;
   cout << "Initial Values: \n";
   cout << "Value of object 1: " << N1.n
      << "\nValue of object 2: " << N2.n<< endl;
   // Passing object as an argument to function add()
   N1.add(N2);
   // Changed values after passing object as argument
   cout << "New values: \n";
   cout << "Value of object 1: " << N1.n
      << "\nValue of object 2: " << N2.n<<endl;
    return 0;
}
```

Output:

```
Initial Values:
Value of object 1: 25
Value of object 2: 100
New values:
Value of object 1: 125
Value of object 2: 100
```

Returning Object as argument - Example

```
#include <iostream>
using namespace std;
class Number
{
   public:
      int n;
      // This function will take two objects
      // as argument and return an object
      Number add(Number N1,Number N2)
      {
         Number N;
         N.n = N1.n + N2.n;
         return N;
      }
};
```

```
int main()
{
   Number N1,N2,N3;
   N1.n = 25;
   N2.n = 100;
   cout << "Initial Values: \n";
   cout << "Value of object 1: " << N1.n
      << "\nValue of object 2: " << N2.n<< endl;
   // Passing 2 objects as argument to function
   //and receiving the returned object
   N3=N3.add(N1,N2);
   cout << "New values: \n";
   cout << "Value of object 1: " << N1.n
        <<"\nValue of object 2: " << N2.n
      << "\nSum of object 1 and object 2 (object 3): " << N3.n<<endl;
   return 0;
}
```

Output:

```
Initial Values:
Value of object 1: 25
Value of object 2: 100
New values:
Value of object 1: 25
Value of object 2: 100
Sum of object 1 and object 2 (object 3): 125
```

UNITII OVERVIEW OF C++

- **Topics to be discussed,**

- **Classes and Objects**
  - **this pointer**
  - **static**
  - **Friend**
- Constructors and Destructors
- Operator Overloading and Type Conversions
- Function object
- Dynamic Memory Management

this Pointer

- The **this** pointer holds the address of current object, in simple words we can say that this pointer points to the current object of the class
- This keyword is *only* accessible within the *nonstatic member functions* of a class/struct/union type.
- There are many ways to use the this pointer,
  - When the parameter name is the same as the private variable's name, we use the this keyword inside the method to distinguish between the two when assigning values.
  - We can also use this to return the reference to the calling object.

When local variable's name is same as member's name - Example

```
#include<iostream>
using namespace std;
class Point
{
   private:
      int x;
      int y;
   public:
      Point (int x, int y)
      {
         this->x = x;
         this->y = y;
      }
      void printPoint()
       {
         cout<<"(x, y) = ("<<this->x<<", "<<this->y<<")"<<endl;
      }
};
int main ()
{
   Point pointA(2, 3), pointB(5, 6);
   pointA.printPoint();
   pointB.printPoint();
   return 0;
}
```

Output:

```
(x, y) = (2, 3)
(x, y) = (5, 6)
```

return reference to the calling object - Example

```
#include<iostream>
using namespace std;
class Point
{
   private:
      int x;
      int y;
   public:
      Point ()
      {
         this->x = 0; this->y = 0;
      }
      Point& setX(int x)
      {
         this->x=x;
         return *this;
      }
      Point& setY(int y)
      {
         this->y=y;
         return *this;
      }
      void printPoint()
      {
         cout<<"(x, y) = ("<<this->x<<", "<<this->y<<")"<<endl;
      }
};
```

```
int main ()
{
   Point pointA;
   pointA.printPoint();
   Point pointB;
   pointB.setX(5).setY(6);
   pointB.printPoint();
   return 0;
}
```

Output:

```
(x, y) = (0, 0)
(x, y) = (5, 6)
```

C++ Classes and Objects –Cont'd

- **Static Data Member:**
  - In C++, **static members are class members that belong to the class itself rather than individual instances (objects) of the class**.
  - In other words, Static data members (or static variables) are class-level variables that are shared by all instances of the class, not with any particular instance of the class.
  - They are *declared using the **static** keyword* and are typically used to store data that is common to all objects of the class.
- **The syntax for declaring a static data member**:

  static data_type data_member_name;

C++ Classes and Objects –Cont'd

- Note that here, we have merely declared the static data member.
- It doesn't allocate memory for the static data member or provide an initial value.
- It merely tells the compiler that such a member exists within the class.
- All static data is **initialized** to **zero** when the first object is created, if no other initialization is present.
- Static data members can be initialized outside the class using the **scope resolution operator (::)** to identify which class it belongs to.
- We initialize static data members after the class declaration and before the main function.
- The memory block of static data members is shared by every object of the class.
- So, if there is any change made in the static data member, it will be **updated for every other object of the class**.

C++ Classes and Objects –Cont'd

- **Static Member Functions:**
  - As we have discussed about the static data members, similarly we have **static member functions which are defined using the static keyword.**
  - Since, we have defined these functions as static, so they consists of class properties rather than object properties.
  - We can access static member functions by using the class name and scope resolution operator.
  - **Syntax :**

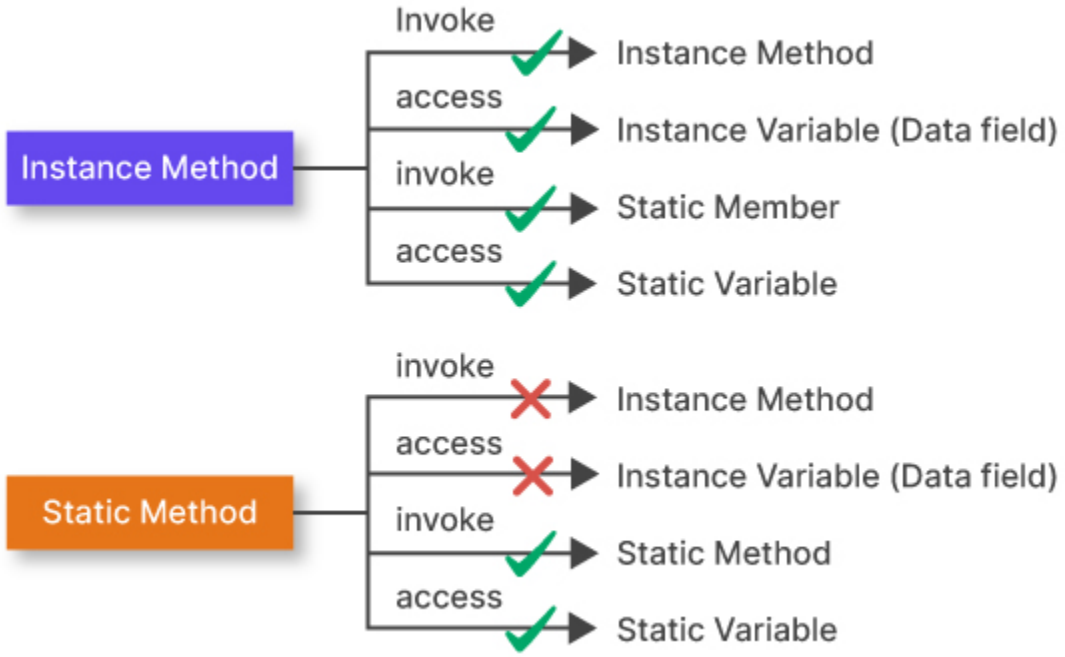
    static class_name :: function_name()
  - As we can call static member function by just using class name and function name, they can be called even if no object of class be created.
  - **A static member function can only access static data members, other static member functions, and any other functions from outside the class.**
  - Static member functions do not have access to this pointer because they have class scope.

C++ Classes and Objects –Cont'd

- **Ways To Call Static Member Function In C++**
- In C++, you can call static member functions using the following methods:
  - ***Using the Scope Resolution Operator (::):***
    - This is the most common and recommended way to call static member functions.
    - It provides a clear and unambiguous syntax for accessing static functions associated with a class.
  - ***Using an Object of the Class:***
    - It's less common and not the recommended practice
    - This method can be confusing and is generally discouraged because it can mislead others into thinking that we are calling a non-static member function.

C++ Classes and Objects –Cont'd

Relationship between instance and static methods



C++ Classes and Objects –Cont'd

Regular Member Function Vs. Static Member Function In C++

Property	Regular Member Function	Static Member Function
Association	Associated with instances of the class.	Associated with the class itself.
Access	Using object instances	Using the class name and scope resolution (::) operator
Can access static data members	Yes (via object or class name)	Yes (via class name and scope resolution operator)
Can access non-static data members	Yes (via object)	No (unless passed as a parameter)
Can access regular member functions	Yes (via object)	No
Can access static member functions	Yes (via object or class name)	Yes (via class name and scope resolution operator)
Access to 'this' pointer	Yes	No
Lifetime	Tied to the object's lifetime.	Tied to the class's lifetime.
Initialization	Per instance	Typically, at class declaration
Virtual Function	Can be declared virtual	Cannot be declared virtual
Polymorphism	Participates in polymorphism	Does not participate in polymorphism

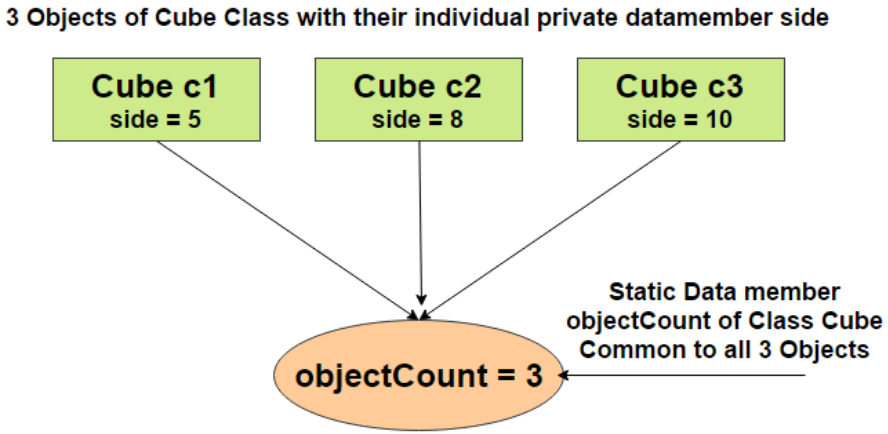
Static Data Member and Static Function - Example

```
#include <iostream>
using namespace std;
class Cube
{
   private:
    int side; // normal data member
   public:
    static int objectCount;// static data member
    void setSide(int s)
    {
       side=s;
    }
    void printSide()
    {
       cout<<"\nside:"<<side;
    }

    static int displayObjectCount() // Constructor definition
      {
         objectCount+=1;
          return objectCount; // Increase every time object is created
      }
};
int Cube::objectCount = 0; // Initialize static member of class Box
```

```
int main(void)
{
  Cube c1;// Object 1.
  cout<<"\nCUBE 1:";
  c1.setSide(5);
  c1.printSide();
  cout << "\nTotal objects: " << Cube::displayObjectCount() << endl;
  Cube c2; // Object 2
  cout<<"\nCUBE 2:";
  c2.setSide(8);
  c2.printSide();
  cout << "\nTotal objects: " << Cube::displayObjectCount() << endl;
  Cube c3; // Object 3
  cout<<"\nCUBE 3:";
  c3.setSide(10);
  c3.printSide();
  cout << "\nTotal objects: " << c3.displayObjectCount() << endl;
  return 0;
}
```

Output:

```
CUBE 1:
side:5
Total objects: 1

CUBE 2:
side:8
Total objects: 2

CUBE 3:
side:10
Total objects: 3
```



C++ Classes and Objects –Cont'd

- **Friends Functions:**
  - A friend function is a **non-member function that has access to the private members of the class.**
  - The outside functions can't access the private data of a class. But there could be a situation where we could like two classes to share a particular data.
  - C++ allows the common function to have access to the private data of the class.
  - Such a function may or may not be a member of any of the classes.
  - To make an outside function "friendly" to a class, we have to simply declare this function as a friend of the class as below.
  - As they could weaken encapsulation and potentially make code more difficult to maintain, friend functions should only be used selectively.

C++ Classes and Objects –Cont'd

```
class ABC
{
 ------
 ------
 public :
 ------
 ------
 friend void xyz(void) ; //declaration
} ;

 void xyz(void) //function definition
{
    //function body
}
```

- The function declaration should be preceded by the keyword **friend**.
- The function definition does not use either the keyword friend or the scope operator (**: :**).
- The function that is declared with the keyword friend are known as **friend functions**.
- A function can be declared as a **friend** in any number of classes.

C++ Classes and Objects –Cont'd

- **Special Characteristics of Friend Function**
  - It is not in the scope of the class to which it has been declared as friend.
  - That is why, it cannot be called using object of that class.
  - It can be invoked like a normal function without the help of any object.
  - It cannot access member names (member data) directly.
  - It has to use an object name and dot membership operator with each member name.
  - It can be declared in the public or the private part of a class without affecting its meaning.
  - Usually, it has the objects as arguments.

C++ Classes and Objects –Cont'd

- **Implementation Of Friend Function**
- A friend function in C++ can be implemented in two ways:
  - As a global function
  - As the member function of another class

C++ Classes and Objects –Cont'd

- **Global Friend Function (Global Function as Friend Function)**
  - The functions that are declared outside of a class or namespace and are accessible from any place in the program are known as global functions.
  - The main() function can invoke these global functions without requiring an object.
  - A global function can be attached as a friend function to one or more classes.
  - Once attached, the global function has direct access to the class's private and protected members.

Syntax:

```
class class_name
{
   access_specifier:
      friend return_type global_function(arguments);
};
return_type global_function(arguments)
{
}
```

```
#include <iostream>
using namespace std;
class Travel
{
   private:
      int speed;
       int distance;
   public:
      void setData (int s, int d)
      {
         speed = s;
         distance = d;
      }
   friend double findTimeofTravel (Travel); // Friend function
};
// Global Function to find the time of Travel not tied to class.
double findTimeofTravel (Travel t)
{
   double time  = (double)t.distance / (double)t.speed;
   return time;
}
```

```
int main ()
{
   Travel t;
    t.setData(10, 30);
    cout << "Time of Travel: " << findTimeofTravel (t) << " hrs" << endl;
    return 0;
}
```

Output:

```
Time of Travel: 3 hrs
```

C++ Classes and Objects –Cont'd

- A friend function can be friendly to 2 or more classes.
- The friend function does not belong to any class, so it can be used to access private data of two or more classes.

```
#include<iostream>
using namespace std;
class B; //forward declaration.
class A
{
   int x;
   public:
       void setData (int i)
        {
          x=i;
        }
   friend void max (A, B); //friend function.
} ;
class B
{
   int y;
   public:
       void setData (int i)
        {
          y=i;
        }
   friend void max (A, B);
};
void max (A a, B b)
{
   cout<<"\na.x="<<a.x;
   cout<<"\nb.y="<<b.y;
  if (a.x >= b.y)
       cout<< "\nMax="<<a.x <<endl;
  else
       cout<<"\nMax="<< b.y << endl;
}
 int main ()
{
  A a;
  B b;
   a. setData (10);
   b. setData (20);
   max (a, b);
   return 0;
}
```

Output:

```
a.x=10
b.y=20
Max=20
```

C++ Classes and Objects –Cont'd

- **C++ Friend Class:**
  - Just like a friend function, a particular class can also have a friend class.
  - A friend class shares the same privilege, i.e., it can access the private and protected members of the class whose friend it has been declared.
  - This means that all the functions declared inside the friend class will also be able to access the private and protected members of the class.
  - **Syntax of Friend Class:**
  - To declare a class as a friend class in C++, it needs to be preceded by the keyword "friend" inside the body of the class, just like with the friend function.

```
// Creating a class named class_Name.
class class_Name
{
   // Declaration of class properties.

   friend class friendClassName;
}
```

  - Here friendClassName is the name of the class which is declared as the friend for the class_Name class.

C++ Classes and Objects –Cont'd

- **Friendship is *granted, not taken*** —for class B to be a friend of class A, class A must *explicitly* declare that class B is its friend.
- **Friendship is not *symmetric*** —if class A is a friend of class B, we cannot infer that class B is a friend of class A.
- **Friendship is not *transitive*** —if class A is a friend of class B and class B is a friend of class C, we cannot infer that class A is a friend of class C.

```
#include<iostream>
using namespace std;
class Shape;
class Square
{
   private:
     int side;
   public:
      void set_values (int s)
      {
         side = s;
      }
   friend class Shape; // friend class
};
class Shape
{
   public:
      void print_area (Square& s)
      {
      // Shape is a friend class of Square class.
      // It can access the side member of the Square class.
      // Calculate the area of the Square.
      int area = s.side*s.side;
      cout<<"\nSide:"<<s.side<<"\nThe area of the Square is: "<<area<<endl;
      }
};
int main ()
{
   Square s;
   s.set_values(5);
   Shape sh;
   sh.print_area(s);
   return 0;
}
```

Output:

```
Side:5
The area of the Square is: 25
```

C++ Classes and Objects –Cont'd

- **Class Members as Friend**
  - We can also make a function of one class as a friend of another class.
  - We do this in the same way as we make a function as a friend of a class.
  - The only difference is that we need to write class_name :: in the declaration before the name of that function in the class whose friend it is being declared.
  - The friend function is only specified in the class and its entire body is declared outside the class. It will be clear from the example given below.

```
class A; // forward declaration of A needed by B
class B
{
    display( A a ); //only specified. Body is not declared
};
class A
{
    friend void B::display( A );
};
void B::display(A a) //declaration here
{
}
```

Program to illustrate the use of friend keyword to make a member function of one class as friend function of another class.

```
#include<iostream>
using namespace std;
class NumberF;
class Number
{
    int i;
    public:
    Number()
    {
        i=5;
    }
    void show(NumberF);

};
class NumberF
{
    float f;
    public:
    NumberF()
    {
        f=123.7;
    }
     friend void Number::show(NumberF);
};
```

```
void Number::show(NumberF n)
      {
        cout<<"\ni="<<i;
        cout<<"\nf="<<n.f;
    }

int main()
{
    Number n1;
    NumberF n2;
    n1.show(n2);

}
```

Output:

```
i=5
f=123.7
```

UNITII OVERVIEW OF C++

- **Topics to be discussed,**

➢Classes and Objects

➢**Constructors and Destructors**

➢Operator Overloading and Type Conversions

➢Function object

➢Dynamic Memory Management

Constructors and Destructors

- **Constructor**
  - A Constructor is a **special member method** which will be **called implicitly (automatically) whenever an object of class is created**.
  - Constructors are special class functions which **performs initialization of every object**.
- **How constructors are different from a normal member function?**
  - Constructor has same name as the class itself.
  - Constructors don't have return type (not even void).
  - A constructor is automatically called when an object is created.
  - If we do not specify a constructor, C++ compiler generates a default constructor for us (expects no parameters and has an empty body).

Class without constructor - Example

```
#include <iostream>
using namespace std;
class sample
{
     int a, b ;
     public:
     void printData()
     {
          cout<<"a="<<a<<"   "<<"b="<<b;
     }
};
int main( )
{
 sample s ;
 s.printData();
}
```

Output:

```
a=16  b=0
```

Garbage values

Class with constructor - Example

```
#include <iostream>
using namespace std;
class sample
{
    int a, b ;
    public:
    sample( )
    {
        cout<< "constructor is called" << endl ;
        a=100; b=200 ;
    }
    void printData()
    {
        cout<<"a="<<a<<"   "<<"b="<<b;
    }
};
int main( )
{
 sample s ; //constructor called
 s.printData();
}
```

Output:

```
constructor is called
a=100   b=200
```

Constructors and Destructors – Cont'd

- **Characteristics of constructor**
  - Constructor has same name as that of its class name.
  - It should be declared in public section of the class.
  - It is invoked automatically when objects are created.
  - It cannot return any value because it does not have a return type even void.
  - It cannot be a virtual function and we cannot refer to its address.
  - It cannot be inherited.
  - It makes implicit call to operators new and delete when memory allocation is required.

Constructors and Destructors – Cont'd

- **C++ Types of Constructors**
  - Constructors are of three types,
    - Default Constructor
    - Parameterized Constructor
    - Copy Constructor

Constructors and Destructors – Cont'd

- **Default Constructor**
  - Default constructor is the constructor which doesn't take any argument.
  - It has no parameter.
  - It is invoked at the time of creating object.
  - If you don't create a default constructor, **the compiler provides a default constructor** by default.

```
#include <iostream>
using namespace std;
class Line
{
    int size;
    public:
        Line()//default constructor
        {
            size=30;
            cout<<"Line size is"<<" "<<size<<" "<<"cm";
        }
};
int main()
{
 //default constructor called when object is created
 Line l;
 return 0;
}
```

Output:

```
Line size is 30 cm
```

Constructors and Destructors – Cont'd

- **C++ Parameterized Constructor**
  - In C++, a constructor with parameters is known as a parameterized constructor.
  - Using this Constructor we can provide different values to data members of different objects, by passing the appropriate values as argument.
  - This is the preferred method to initialize member data.

```
#include <iostream>
using namespace std;
class Cube
{
        int side;
    public:
    Cube(int x)//Parameterized Constructor
    {
        side=x;
    }
    void putData()
    {
        cout<<"\nSide:"<< side<<endl;
    }
};
int main()
{
    Cube c1(10);
    Cube c2(20);
    Cube c3(30);
    c1.putData();
    c2.putData();
    c3.putData();
}
```

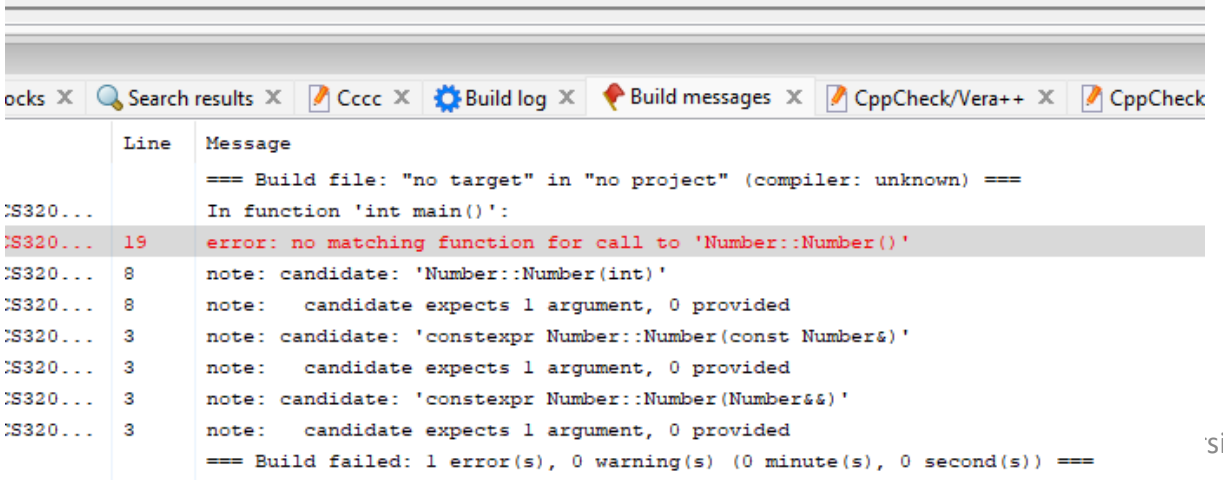
Output:

```
Side:10

Side:20

Side:30
```

Constructors and Destructors – Cont'd

- **Does C++ compiler create default constructor when we write our own?**
  - **NO,** the C++ compiler doesn't create a default constructor when we initialize our own.
  - If there is no Constructor present in a class, one Default Constructor is added at Compile time.
  - If there is any one parametrized Constructor present in a class, Default Constructor will not be added at Compile time.
  - So if our program has any constructor containing parameters and no default constructor is specified then we will not be able to create object of that class using Default constructor.

```cpp
#include <iostream>
using namespace std;
class Number
{
      private:
         int x;
      public:
        Number(int a) //Parameterized constructor
        {
              x =a;
        }
        void display()
        {
              cout << "x = " << x <<endl;
        }
};
int main()
{
    Number n1;
    n1.display();
    return 0;
 }
```



Constructors and Destructors – Cont'd

- **C++ Copy Constructor**
  - A copy constructor is a member function which initializes an object using another object of the same class.
  - These are special type of Constructors which takes an object as argument, and is used to copy values of data members of one object into other object

```
#include <iostream>
using namespace std;
class Cube
{
        int side;
    public:
    Cube(int x)//Parameterized Constructor
    {
        cout<<"\nParameterized Constructor called";
        side=x;
    }
    Cube(Cube &c)
    {
        cout<<"\nCopy Constructor called";
        side=c.side;
    }
    void putData()
    {
        cout<<"\nSide:"<< side<<endl;
    }
};
int main()
{
    Cube c1(10);
    Cube c2=c1;
    Cube c3(c1);
    c1.putData();
    c2.putData();
    c3.putData();
}
```

Output:

```
Parameterized Constructor called
Copy Constructor called
Copy Constructor called
Side:10

Side:10

Side:10
```

Constructors and Destructors – Cont'd

- **Constructor Overloading**
  - Overloading in Constructors are the constructors with the same name and different parameters (or arguments).
  - Hence, the constructor call depends upon data types and the number of arguments.

```
#include <iostream>
using namespace std;
class ABC
{
     private:
        int x,y;
     public:
        ABC ()//constructor 1 Default Constructor(with NO argume
        {
             x = y = 0;
        }
        ABC(int a) //constructor 2 with ONE argument
        {
              x = y = a;
        }
        ABC(int a,int b)     //constructor 3 with TWO arguments
        {
              x = a;
              y = b;
        }
        ABC(ABC &obj)// Copy Constructor
        {
            x=obj.x;
            y=obj.y;
        }
        void display()
        {
              cout << "x = " << x << " and " << "y = " << y << endl;
        }
};
```

```
int main()
{
     ABC obj1; //constructor 1
     ABC obj2(5); //constructor 2
     ABC obj3(10,20); //constructor 3
     ABC obj4=obj3;//Copy constructor
     obj1.display();
     obj2.display();
     obj3.display();
     obj4.display();
     return 0;
}
```

Output:

```
x = 0 and y = 0
x = 5 and y = 5
x = 10 and y = 20
x = 10 and y = 20
```

Constructors and Destructors – Cont'd

- **Destructor**
  - Destructor is a member function which deletes an object.
  - A destructor is called automatically by the compiler when the object goes out of scope.
  - A destructor is a special member function that works just opposite to constructor, unlike constructors that are used for initializing an object, destructors destroy (or delete) the object.
  - It can be defined only once in a class and must have same name as class.
  - Like constructors, it is invoked automatically. But it is prefixed with a tilde sign (~).

Syntax:

```
class Name_of _class
{
   public:
   ~Name_of_class() //this is known as Destructor
   {

   }
}
```

Constructors and Destructors – Cont'd

- **Properties of Destructor**
- A Destructor is different from normal functions in following ways:
  - Destructor function is automatically invoked when the objects are destroyed.
  - It cannot be declared static or const.
  - The destructor does not have arguments.
  - It has no return type not even void.
  - An object of a class with a Destructor cannot become a member of the union.
  - A destructor should be declared in the public section of the class.
  - The programmer cannot access the address of destructor.

Constructors and Destructors – Cont'd

- **When does the destructor get called?**
- The destructor will be called automatically when an object is no longer in scope and also in situations mentioned below
  - When the function ends.
  - When the program ends.
  - When a scope (the { } parenthesis) containing local variable ends.
  - When a delete operator is called.
- **How destructors are different from a normal member function?**
- A destructor is different from normal functions in following ways:
  - Destructors have same name as the class but is preceded by a tilde (~).
  - Destructors have no argument and no return value.

Destructor Example 1

```
#include <iostream>
using namespace std;
class Cube
{
   int side;
public:
   Cube()
   {
      side=5;
      cout<<"Constructor Called,side="<<side;
   }
   ~Cube()
   {
      cout<<"\nDestructor Called";
   }
};
int main()
{
   Cube c;
}
```

Output:

```
Constructor Called,side=5
Destructor Called
```

Destructor Example 2

```
#include<iostream>
using namespace std;
class Array
{
   private:
      int size;
      int *ptr;
   public:
     Array()
     {
         cout<<"\nInside constructor";
        size=0;
         ptr=NULL;
     }
     void getData()
     {
        cout <<"\nEnter the number of integers in the array:";
        cin >>size;
        ptr = new int[size];
        int i;
        for (i=0; i< size; i++)
        {
           cout <<"\nEnter a number: " ;
           cin >>*(ptr+i);
        }
     }
```

```
void printData()
  {
    int i =0;
    cout <<"\nThe array contains " <<size << " integers.";
    cout <<"\nThey are:\n";
     for(;i<size;i++)
       {
          cout <<*(ptr+i)<<endl;
       }
  }
~Array()       //destructor to delete the array
  {
     cout<<"\nInside Destructor";
     delete []ptr;
  }

};
 int main()
{
   Array array1;
   array1.getData();
   array1.printData();
}
```

Output:

```
Inside constructor
Enter the number of integers in the array:3

Enter a number: 12

Enter a number: 34

Enter a number: 23

The array contains 3 integers.
They are:
12
34
23

Inside Destructor
```

Constructors and Destructors – Cont'd

- C++ Destructor Overloading
  - The destructor cannot be overloaded.
  - In a class, we can only have one destructor.
  - In a class followed by a class name, there can only be one destructor with no parameters and no return type.
- Default Destructor and User-Defined C++ Destructor
  - If we do not write our own destructor in class, compiler creates a default destructor for us.
  - The default destructor works fine unless we have dynamically allocated memory or pointer in class.
  - When a class contains a pointer to memory allocated in class, we should write a destructor to release memory before the class instance is destroyed.
  - This must be done to avoid memory leak.

UNITII OVERVIEW OF C++

- **Topics to be discussed,**

➢Classes and Objects

➢Constructors and Destructors

➢**Operator Overloading and Type Conversions**

➢Function object

➢Dynamic Memory Management

UNITII OVERVIEW OF C++

- **Topics to be discussed,**

➢Classes and Objects

➢Constructors and Destructors

➢**Operator Overloading**

➢Function object

➢Dynamic Memory Management

Operator Overloading

- Operator overloading is a **compile-time polymorphism** in which the operator is overloaded to give user defined meaning to it.
- Operator overloading is used to overload or redefine most of the operators available in C++.
- **Operator overloading is to provide a special meaning or redefinition to an operator** (unary or binary), **so that it could work on user-defined data-type objects just in the same way as it works on built-in data-type type variables.**
- For example, we could add two built-in data type int variables by just using the + operator between these two int variables.

  int a = 10, b = 20;

  int result = a + b;
- And, C++ also allows us to use + operator on two objects of user-defined data-type by using the concept of operator overloading using member function or a non-member friend function

  ob3 = ob1 + ob2;
  - where, ob1, ob2 and ob3 are the objects of a class and operator + is overloaded in a way that we could just simple apply the + operator on the objects of class, in the same manner as we apply the + operator on any variables of built-in data-types.

Operator Overloading– Cont'd

- **For Example:-**
- Suppose we have created three objects c1, c2 and c3 for a class complex and if we want to add c1 and c2 and store in c3,Usually we write as , c3=ADD(c1,c2)
- Since operator overloading allows us to change how operators work, we can redefine how the + operator works and use it to add the complex numbers we can write as c3=c1+c2
- This makes our code intuitive and easy to understand.
- **Note:** We cannot use operator overloading for fundamental data types like int, float, char and so on.

Operator Overloading – Cont'd

- **Syntax:**

  returnType classname:: **Operator** OperatorSymbol(argument list)
  {
  // function body
  }
  - returnType is the return type of the function.
  - operator keyword.
  - symbol is the operator we want to overload. Like: +, <, -, ++, etc.
  - argument(s) can be passed to the operator function in the same way as function
- **What is the difference between operator functions and normal functions?**
  - Operator functions are same as normal functions.
  - The only differences are, name of an operator function is always operator keyword followed by symbol of operator and operator functions are called when the corresponding operator is used.

Operator Overloading – Cont'd

- **Implementing Operator Overloading**
- Operator overloading can be done by implementing a function which can be :
  - Member Function
  - Friend Function
- Operator overloading function can be a member function if the Left operand is an Object of that class, but if the Left operand is different, then Operator overloading function must be a non-member function.
- Operator overloading function can be made friend function if it needs access to the private and protected members of class.

Operator Overloading – Cont'd

- **What Operators we can Overload?**
  - Unary Operators (**-, +, !, ~**)
  - Binary Operators or Arithmetic Operators (**+, -, *, |, %**)
  - Relational / Comparison Operators (**<, >, <=, >=, !=** etc)
  - Increment / Decrement Operators (**++, —**)
  - Logical Operators (&&, ||)
  - Comma Operator (,)
  - Pointer to Member (– >)
  - Operator Assignment (**+=, -=, &=, /=, %=**)

- **What Operators we can not Overload?**
  - Scope Resolution Operator (**::**)
  - Pre-processed symbol operator (#,##)
  - Class member operators (**.**)
  - Size of operator (size of)
  - Conditional operator(?:)
  - Pointer to member operator (.*)

Operator Overloading– Cont'd

- **Rules of Operating Overloading**
  - Only the existing operators can be overloaded. New operators cannot be created
  - The overloaded operators must have at least one user-defined operand.
  - Overloaded operators follow the syntax rule of the original operators .i.e. We cannot redefine the plus(+) operator to subtract one value from the other.
  - Friend function cannot be used to overload certain operators like =, ( ), [ ] , ->
  - Some operators like the size of the operator(size of), membership operator (.), a pointer to a member operator (.*), scope resolution operator (: :), conditional operator (? :), etc cannot be overloaded.
  - Binary operators such as +, -, *, / etc must explicitly return a value.

Operator Overloading – Cont'd

- **Overloading Binary Operator as Member function**
  - This takes two operands while overloading.
  - For example, c=a+b where a and b are two operands.
  - Following binary operators can be overloaded.
    - Arithmetic Operators (+, -, *, |, %)
    - Arithmetic assignment operator (+=, -=, *=, /=, %=)
    - Relational operator (>, <, >=, <=, !=, ==)
  - When we overload the binary operator for user-defined types by using the code:

    obj3 = obj1 + obj2;

    - The operator function is called using the obj1 object and obj2 is passed as an argument to the function.

Overloading Binary Operator as Member function - Example

```
#include<iostream>
using namespace std;
class complex
{
    float x,y;
    public:
    complex() { }
    complex (float real, float imag)
     {
        x = real;
        y= imag;
     }
    void display()
    {
        cout<<x<<"+"<<y<<"i\n";
    }
    complex operator+(complex c)
     {
        complex sum;
        sum.x = x+c.x;
        sum.y = y+c.y;
        return sum;
    }
};
```

```
int main()
{
    complex c1(2,3),c2(4,5),c3;
    c3=c1+c2;
    cout<<"Complex Number 1:";
    c1.display();
    cout<<"Complex Number 2:";
    c2.display();
    cout<<"Sum:";
    c3.display();
}
```

Output:

```
Complex Number 1:2+3i
Complex Number 2:4+5i
Sum:6+8i
```

Operator Overloading– Cont'd

In the previous program, instead of

```
complex operator+(complex c)
 {
     complex sum;
     sum.x = x+c.x;
     sum.y = y+c.y;
     return sum;
}
```

we also could have written this function like,

```
complex operator+(const complex &c)
 {
     complex sum;
     sum.x = x+c.x;
     sum.y = y+c.y;
     return sum;
}
```

- using & makes our code efficient by referencing the complex2 object instead of making a duplicate object inside the operator function.
- using const is considered a good practice because it prevents the operator function from modifying complex2.

Operator Overloading – Cont'd

- As we discussed earlier **in case of operator overloading first operand invokes the operator function and second one is passed as argument**.
- But in some circumstance, the first object may not be object and it may be some primitive data types or some other object.
- Then how do we make it work?
  - In this scenario friend functions can be of help to us.
  - As friend function is not a member of class, it does not require object for its invocation.
  - Therefore the first argument of operator need not be object of the class for which operator is being overloaded.
  - However at least one of the operands must be the object of the class for which operator is being overloaded.
  - Both Unary as well as binary operators can be overloaded through friend function.

Overloading Binary Operator as Friend function – Example 1

```
#include<iostream>
using namespace std;
class complex
{
    float x,y;
    public:
    complex() { }
    complex (float real, float imag)
     {
        x = real;
        y= imag;
     }
    void display()
    {
        cout<<x<<"+"<<y<<"i\n";
    }
    friend complex operator+(complex &c1,complex &c2)
     {
        complex sum;
        sum.x = c1.x+c2.x;
        sum.y = c1.y+c2.y;
        return sum;
    }
};
```

```
int main()
{
    complex c1(2,3),c2(4,5),c3;
    c3=c1+c2;
    cout<<"Complex Number 1:";
    c1.display();
    cout<<"Complex Number 2:";
    c2.display();
    cout<<"Sum:";
    c3.display();
}
```

Output:

```
Complex Number 1:2+3i
Complex Number 2:4+5i
Sum:6+8i
```

Overloading Binary Operator as Friend function – Example 2

```
#include<iostream>
using namespace std;
class complex
{
    float x,y;
    public:
    complex() { }
    complex (float real, float imag)
     {
        x = real;
        y= imag;
     }
    void display()
    {
        cout<<x<<"+"<<y<<"i\n";
    }
    friend complex operator+(int,complex);
    friend complex operator+(complex,int);
};
complex operator+(int r,complex c)
     {
        complex sum;
        sum.x = r+c.x;
        sum.y = c.y;
        return sum;
    }
complex operator+(complex c,int i)
     {
        complex sum;
        sum.x = c.x;
        sum.y = c.y+i;
        return sum;
    }
```

```
int main()
{
    complex c1(2,3),c2,c3;
    int r=5,i=10;
    c2=r+c1;
    c3=c1+i;
    cout<<"Complex Number 1:";
    c1.display();
    cout<<"Complex number 2:";
    c2.display();
    cout<<"Complex number 3:";
    c3.display();
}
```

Output:

```
Complex Number 1:2+3i
Complex number 2:7+3i
Complex number 3:2+13i
```

Operator Overloading– Cont'd

- **Unary Operator Overloading**
- Unary operators are those operators that operate (act) on a single operand. For example, ++, — are unary operators.
- Following unary operators can be overloaded.
  - Unary plus(+) & unary minus(-)
  - Increment(++) & decrement(--) [prefix and postfix]
  - Complement operator(~)
  - Logical NOT (!)
  - Address of (&)
  - Pointer deference (*)

Operator Overloading – Cont'd

- **Syntax for overloading Unary operator as member function is as follows:**

  return_type operator operator_to_be_overloaded ( );

  - Here unary operator has only one operand, which will be the invoking object, hence passed implicitly in the function.
  - Therefore it does not require any argument. If unary operator is overloaded using friend function which is not invoked on any object, the operand needs to be passed as argument.

- **The syntax for declaring function in class for overloading unary operator using friend function is as follows:**

friend return_type operator operator_to_be_overloaded (object);

Unary Operator Overloading using member function – Example 1

```
#include <iostream>
using namespace std;
class Count
 {
   private:
    int num;
   public:
    Count (){num=1;}
    void operator - ()
    {
        num = -num;
    }
    void display()
    {
        cout << "The Count: " << num << endl;
    }
};
int main()
{
    Count c;
    c.display();
    -c;
    c.display();
    return 0;
}
```

Output:

```
The Count: 1
The Count: -1
```

Unary Operator Overloading using member function – Example 1

```
#include <iostream>
using namespace std;
class Count
 {
   private:
    int num;
   public:
    Count(){num=2;}
    Count operator - ()
    {
        Count c;
        c.num = -num;
        return c;
    }
    void display()
    {
        cout << "The Count: " << num << endl;
    }
};
int main()
{
    Count c;
    c.display();
    c=-c;
    c.display();
    return 0;
}
```

Output:

```
The Count: 2
The Count: -2
```

Unary Operator Overloading using friend function – Example 1

```
#include <iostream>
using namespace std;
class Count
 {
   private:
    int num;
   public:
    Count(){num=5;}
    void display()
    {
        cout << "The Count: " << num << endl;
    }
    friend Count operator - (Count);
};
Count operator - (Count c)
    {
        c.num = -c.num;
        return c;
    }
int main()
{
    Count c;
    c.display();
    c=-c;
    c.display();
    return 0;
}
```

Output:

```
The Count: 5
The Count: -5
```

Unary Operator Overloading using friend function – Example 2

```
#include <iostream>
using namespace std;
class Count
 {
   private:
    int num;
   public:
    Count(){num=10;}
    void display()
    {
        cout << "The Count: " << num << endl;
    }
    friend void operator - (Count &);
};
void operator - (Count &c)
    {
        c.num = -c.num;
    }
int main()
{
    Count c;
    c.display();
    -c;
    c.display();
    return 0;
}
```

Output:

```
The Count: 10
The Count: -10
```

Operator Overloading– Cont’d

- Unary ++, -- Overloading :
  - The operator symbol for both **prefix(++i)** and **postfix(i++)** are the same.
  - Hence, we need two different function definitions to distinguish between them.
  - This is achieved by passing a dummy int parameter in the postfix version.

Unary ++ Overloading – Example 1

```
#include <iostream>
using namespace std;
class Integer
{
private:
    int i;
public:
    Integer(){}
    Integer(int i)
    {
        this->i = i;
    }
    // Overloading the prefix operator
    void operator++()
    {
        ++i;
    }
    // Overloading the postfix operator
    void operator++(int)
    {
        i++;
    }
    // Function to display the value of i
    void display()
    {
        cout << i << endl;
    }
};
```

```
int main()
{
    Integer i1(5),i2(10);
    cout << "Before increment:i1=";
    i1.display();
     // Using the pre-increment operator
    ++i1;
    cout << "After pre increment:\ni1=";
    i1.display();
    cout << "Before increment:i2=";
    i2.display();
    // Using the post-increment operator
    i2++;
    cout << "After post increment:\ni2=";
    i2.display();
}
```

Output:

```
Before increment:i1=5
After pre increment:
i1=6
Before increment:i2=10
After post increment:
i2=11
```

Unary ++ Overloading – Example 2

```
using namespace std;
class Integer
{
private:
    int i;
public:
    Integer(){}
    Integer(int i)
    {
        this->i = i;
    }
    // Overloading the prefix operator
    Integer& operator++()
    {
        ++i;
        // returned value should be a
        // reference to *this
        return *this;
    }
    // Overloading the postfix operator
    Integer operator++(int)
    {
        // returned value should be a copy
        //of the object before decrement
        Integer temp = *this;
        ++i;
        return temp;
    }
    // Function to display the value of i
    void display()
    {
        cout << i << endl;
    }
};
```

```
int main()
{
    Integer i1(5),i2(10),i3;
    cout << "Before increment:i1=";
    i1.display();
     // Using the pre-increment operator
    i3 = ++i1;
    cout << "After pre increment:\ni1=";
    i1.display();
    cout << "i3=";
    i3.display();
    cout << "Before increment:i2=";
    i2.display();
    // Using the post-increment operator
    i3 = i2++;
    cout << "After post increment:\ni2=";
    i2.display();
    cout << "i3=";
    i3.display();
}
```

Output:

```
Before increment:i1=5
After pre increment:
i1=6
i3=6
Before increment:i2=10
After post increment:
i2=11
i3=10
```

Operator Overloading– Cont'd

- **Overloading stream insertion (<<) and extraction (>>) operators in C++:**
  - C++ is able to input and output the built-in data types using the stream extraction operator >> and the stream insertion operator <<.
  - The stream insertion and stream extraction operators also can be overloaded to perform input and output for user-defined types like an object.
  - Here, it is important to make operator overloading function a friend of the class because it would be called without creating an object.

Overloading stream insertion (<<) and extraction (>>) operators in C++ - Example

```
#include <iostream>
using namespace std;
class Distance
{
   private:
   int feet;
   int inch;
   public:
   Distance()
   {
      feet = 0;
      inch = 0;
   }
   Distance(int f, int i)
   {
      feet = f;
      inch = i;
   }
   friend ostream &operator<<(ostream &output,const Distance &D)
   {
      output << "\nFeet:" << D.feet << "\nInch:" << D.inch;
      return output;
   }
   friend istream &operator>>(istream &input, Distance &D)
   {
      input >> D.feet >> D.inch;
      return input;
   }
};
```

```
int main()
{
   Distance D1(10, 5),D2;
   cout << "First Distance : " << D1 << endl;
   cout << "Enter feet and inch:" << endl;
   cin >> D2;
   cout << "Second Distance :" << D2 << endl;
   return 0;
}
```

Output:

```
First Distance :
Feet:10
Inch:5
Enter feet and inch:
6 7
Second Distance :
Feet:6
Inch:7
```

Subscripting [] Operator Overloading in C++

```
#include <iostream>
using namespace std;
const int SIZE = 10;
class Array
{
   private:
      int arr[SIZE];
  public:
      Array()
      {
         int i;
         for(i = 0; i < SIZE; i++)
         {
           arr[i] = i;
         }
      }
      int operator[](int i)
      {
         if( i > SIZE )
         {
            cout << "Index out of bounds ";
            return -1;
         }
         return arr[i];
      }
};
int main()
{
   Array A;
   cout << "Value of A[3] : " << A[3] <<endl;
   cout << "Value of A[7] : " << A[7]<<endl;
   cout << "Value of A[15] : " << A[15]<<endl;
   return 0;
}
```

- The subscript operator [] is normally used to access array elements.
- This operator can be overloaded to enhance the existing functionality of C++ arrays.

Output:

```
Value of A[3] : 3
Value of A[7] : 7
Value of A[15] : Index out of bounds -1
```

UNITII OVERVIEW OF C++

- **Topics to be discussed,**

➢Classes and Objects

➢Constructors and Destructors

➢**Operator Overloading and Type Conversions**

➢Function object

➢Dynamic Memory Management

UNITII OVERVIEW OF C++

- **Topics to be discussed,**

➢Classes and Objects

➢Constructors and Destructors

➢**Type Conversions**

➢Function object

➢Dynamic Memory Management

Type Conversions

- **Data Type Conversion in C++: Basic and User Defined Data Types**
  - It is easier to convert same type of data from one to the other.
  - For instance if we have two integer variables say, int num1, num2;
  - The compiler will automatically convert them from one form to other if an equal's sign is used between these two.

    For example

    num1 = num2;
    - will be automatically compiled by the compiler and no explicit logic needs to be embedded in the code.
    - This is the case with basic data types.
  - In user defined data types (objects).
  - If we could create two objects of class1 like, class1 obj1, ob2;

    obj1=obj2
    - In that all the values from the variables of obj2 will be copied into the corresponding variables of obj1.
  - In both of our examples we are storing values between variables of same data type.
  - First we stored data from an integer to another integer and then we stored user defined objects of class1 to another object of class1.

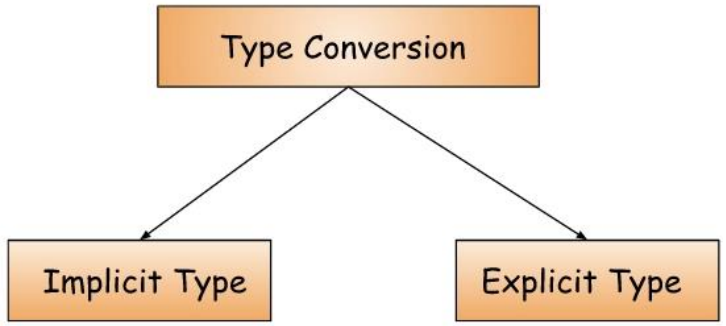
Type Conversions – Cont'd

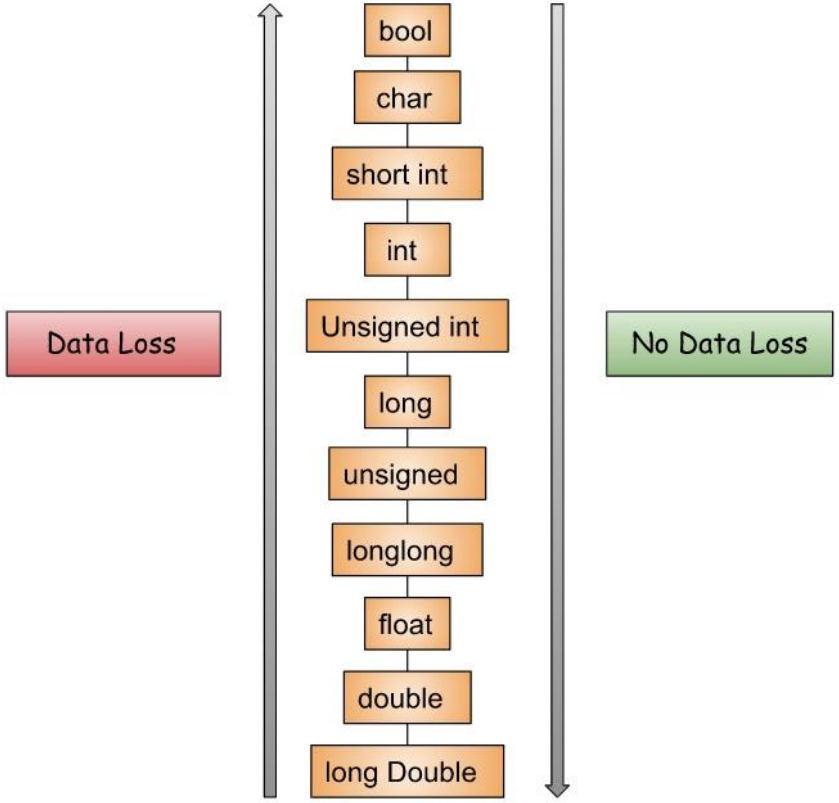
- If we want to store a float type data into an integer or an object of class1 into another object of class2 i.e. the data type of both of these objects will be different then we need to introduce some coding.
- **Conversion between basic data types:**
- Conversion between basic data types is a straight forward process and is often done implicitly.
- For example if you write
- Code:
- Int num1;
- float num2 = 0.516;
- Now, you want to assign the value of float to num1, you can do
- Code:
- num1= num2;
- Compiler will not generate any error in this case though; num1 is of integer type whereas num2 is of float type. This is implicit conversion where compiler will run some internal routine to convert the float value to integer.

- However, we can also direct the compiler to convert the float type to integer. This is called explicit conversion of basic type data.

Type Conversions – Cont'd

- **Conversion between basic data types:**





Type Conversions – Cont'd

- **Conversion between basic data types:**
  - Conversion between basic data types is a straight forward process and is often done implicitly.

Implicit Type conversion Example

```
#include <iostream>
using namespace std;
int main()
{
    int num1;
    float num2 = 25.987;
    num1=num2;
    cout<<"Float number= "<<num2;
    cout<<"\nInt number= "<<num1;
}
```

Output:

```
Float number= 25.987
Int number= 25
```

In the above example, float was implicitly converted into integer; truncating the decimal part.

Type Conversions – Cont'd

- In the previous example, float was implicitly converted into integer; truncating the decimal part.
- However, for the purpose of readability, we can also explicitly specify the conversion between float and integer.
- This is called explicit conversion of basic type data.

Explicit Type conversion Example

```
#include <iostream>
using namespace std;
int main()
{
    int num1;
    float num2 = 25.987;
    num1=static_cast<int>(num2);
    cout<<"Float number= "<<num2;
    cout<<"\nInt number= "<<num1;
}
```

Output:

```
Float number= 25.987
Int number= 25
```

- shows that num2 is being converted into integer type.
- If we want to convert it into any other type, we can simply replace <int> with the <type>, where type is the type of data into which we want the conversion to be done.

Type Conversions – Cont'd

- **Conversion between Basic Data Types and User Defined Data types**
  - Conversion between basic data types is extremely simple as we studied in the last section.
  - However; conversion between user defined and basic data types entails explicit programming logic because compiler doesn't know anything about the user defined data type.
  - Therefore, we have write code to convert basic data types into user defined data types and vice versa.
  - Three type of situation arise in user defined data type conversion.
    - Basic type to Class type
    - Class type to Basic type
    - Class type to Class type

Type Conversions – Cont'd

- **Conversion from basic type to class type**
  - The conversion from basic to class type can be **done by using the constructor**.

Basic type to class type – Example 1

```
#include<iostream>
using namespace std;
class Number
{
   int i;
   float f;
   public:
   Number()
   {
      i=0;
      f=0.0;
   }
   Number(int a,float b)
   {
      i=a;
      f=b;
   }
  void show()
   {
      cout<<"\ni="<<i;
      cout<<"\nf="<<f;
   }
   Number(int num)
   {
      cout<<"\nint constructor invoked";
      i=num;
   }
   Number(float num)
   {
      cout<<"\nfloat constructor invoked";
      f=num;
   }
};
int main()
{
   Number n1,n2;
   int x;
   cout<<"\nEnter an integer:";
   cin>>x;
   n1=x;
   cout<<"\nNumber n1:";
   n1.show();
   float y;
   cout<<"\nEnter a floating point number:";
   cin>>y;
   n2=y;
   cout<<"\nNumber n2:";
   n2.show();
}
```

Output:

```
Enter an integer:56

int constructor invoked
Number n1:
i=56
f=0
Enter a floating point number:89.678

float constructor invoked
Number n2:
i=16
f=89.678
```

Basic type to class type – Example 2

```
#include <iostream>
using namespace std;
class Time
{
        int hours;
        int minutes;
    public:
        Time(){}
        Time (int t) // constructor
        {
            hours = t / 60; //t is inputted in minutes
            minutes = t % 60;
        }
        void show()
        {
            cout<<"Time:"<<hours<<":"<<minutes<<endl;
        }
};
int main()
{
    Time t;
    int min;
    cout<<"\nEnter time period (minutes):";
    cin>>min;
    t=min;
    t.show();
}
```

Output:

```
Enter time period (minutes):130
Time:2:10
```

Type Conversions – Cont'd

- **Conversion from class type to basic type**
  - The constructor functions do not support conversion from a class to basic type.
  - C++ allows us to define a overloaded casting operator that convert a class type data to basic type.
  - The general form of an overloaded casting operator function, also referred to as a **conversion function**, is:
  - **Syntax:**

```
operator typename( )
{
        //Program statement.
}
```

    - This function converts a class type data to typename data.
  - The casting operator should satisfy the following conditions,
    - It must be a class member.
    - It must not specify a return type.
    - It must not have any arguments. Since it is a member function, it is invoked by the object and therefore, the values used for, Conversion inside the function belongs to the object that invoked the function. As a result function does not need an argument.

Conversion from class type to basic type – Example 1

```
#include<iostream>
using namespace std;
class Number
{
   int i;
   float f;
   public:
   Number()
   {
      i=0;
      f=0.0;
   }
   Number(int a,float b)
   {
      i=a;
      f=b;
   }
   void show()
   {
      cout<<"\ni="<<i;
      cout<<"\nf="<<f;
   }
   operator int()
   {
      int i;
      i=this->i;
      return i;
   }
   operator float()
   {
      float n;
      n=f;
      return n;
   }
};
int main()
{
   Number n1(2,3.6),n2(5,6.7);
   int x;
   float y;
   cout<<"\nNumber n1:";
   n1.show();
   x=n1;
   cout<<"\nx="<<x;
   cout<<"\nNumber n2:";
   n2.show();
   y=n2;
   cout<<"\ny="<<y;
}
```

Output:

```
Number n1:
i=2
f=3.6
x=2
Number n2:
i=5
f=6.7
y=6.7
```

Conversion from class type to basic type – Example 2

```
#include <iostream>
using namespace std;
class Time
{
        int hours;
        int minutes;
    public:
        Time(){}
        Time (int h,int m)
        {
            hours = h;
            minutes = m;
        }
        void show()
        {
            cout<<"Time:"<<hours<<":"<<minutes<<endl;
        }
        operator int()
        {
            int m;
            m=hours*60+minutes;
            return m;
        }
};
int main()
{
    Time t(2,30);
    t.show();
    int min;
    min=t;
    cout<<"\nMinutes="<<min;
}
```

Output:

```
Time:2:30

Minutes=150
```

Type Conversions – Cont'd

- **Conversion from one class type to another class type:**
  - We have just seen data conversion techniques from a basic to class type and a class to basic type.
  - But sometimes we would like to convert one class data type to another class type.
  - **Example**

    Obj1 = Obj2; //Obj1 and Obj2 are objects of different classes.
    - Obj1 is an object of class one and Obj2 is an object of class two.
    - The class two type data is converted to class one type data and the converted value is assigned to the Obj1.
    - Since the conversion takes place from class two to class one, two is known as the source and one is known as the destination class.
- Such conversion between objects of different classes can be carried out by either a **constructor** or a **conversion function**.
- Which form to use, depends upon where we want the type-conversion function to be located, whether in the source class or in the destination class.

Type Conversions – Cont'd

- We studied that the casting operator function ***Operator typename()*** Converts the class object of which it is a member to **typename.**
- The type name may be a built-in type or a user defined one (another class type).
- In the case of conversions between objects, ***typename*** refers to the destination class.
- Therefore, when a class needs to be converted, a casting operator function can be used.
- **The conversion takes place in the source class and the result is given to the destination class object.**

Conversion takes place in,

Conversion	Source class	Destination class
Basic to class	Not applicable	Constructor
Class to Basic	Casting operator	Not applicable
Class to class	Casting operator	Constructor

Conversion from one class type to another class type – using constructor

```
//using constructor
//change from  hours(class) to minutes (class)
#include<iostream>
using namespace std;
class hours
{
    int hr;
    public:
    hours()
    {
        hr=0;
    }
    void get()
    {
        cout<<"\nEnter hr:";
        cin>>hr;
    }
    int getHr(){return hr;}
    void show()
    {
        cout<<"\nHr="<<hr;
    }
};
```

```
class minutes
{
    int m;
    public:
    minutes()
    {
        m=0;
    }
    minutes(int x)
    {
        m=x;
    }
    minutes(hours H)
    {
        m=H.getHr()*60;
    }
    void show()
    {
        cout<<"\nminute:"<<m;
    }
};
```

```
int main()
{
    hours h;
    minutes m;
    h.get();
    h.show();
    m=h;
    m.show();
}
```

Output:

```
Enter hr:3

Hr=3
minute:180
```

Conversion from one class type to another class type – using conversion function

```
//using conversion function
//change from  hours(class) to minutes (class)
#include<iostream>
using namespace std;
class minutes
{
    int m;
    public:
    minutes()
    {
        m=0;
    }
    minutes(int x)
    {
        m=x;
    }

    void show()
    {
        cout<<"\nMinutes:"<<m;
    }
};
```

```
class hours
{
    int hr;
    public:
    hours()
    {
        hr=0;
    }
    hours(int h)
    {
        hr=h;
    }
    void get()
    {
        cout<<"\nEnter hr:";
        cin>>hr;
    }
    operator minutes()
    {
        int min=hr*60;
        return(minutes(min));
    }
    void show()
    {
        cout<<"\nHr="<<hr;
    }
};
```

```
int main()
{
    hours h;
    minutes m;
    h.get();
    h.show();
    m=h;
    m.show();
}
```

Output:

```
Enter hr:2

Hr=2
Minutes:120
```

UNITII OVERVIEW OF C++

- **Topics to be discussed,**

➢Classes and Objects

➢Constructors and Destructors

➢Operator Overloading and Type Conversions

➢**Function object**

➢Dynamic Memory Management

Function object

- A C++ **functor** (function object) is a class or struct **object that can be called like a function.**
- This can be achieved by overloading the function call operator ()
- Public access must be granted to the overloading of the () operator in order to be used as intended.
- Functors can simplify tasks and improve efficiency in many cases.
- **Syntax:** For an object to be a functor, the class body must contain the following:

```
class MyClass
{
  public:
    type operator()(...) {
      // Function body
    }
};
```

  - type is the data type to be returned
  - operator() overloads the function call operator () and (...) are the arguments required to execute the function body.

Function object – Cont'd

- To use the functor, an instance of the class is created.
- Then, it is called with arguments passed in:

  MyClass myclass;
  myclass(...);

```
#include<iostream>
using namespace std;
class Welcome
{
  public:
    void operator()()
    {
      cout << "Welcome to MIT!\n";
    }
};
int main()
{
  Welcome msg;
  cout<<"\nFirst:\n";
  msg();
  cout<<"\nSecond:\n";
  msg.operator()();
  return 0;
}
```

Output:

```
First:
Welcome to MIT!

Second:
Welcome to MIT!
```

```
#include<iostream>
using namespace std;
class Welcome
{
  public:
    void operator()(int n)
    {
        for(int i=1;i<=n;i++)
            cout << "Welcome to MIT!\n";
    }
};
int main()
{
  Welcome msg;
  cout<<"\nFirst:\n";
  msg(2);
  cout<<"\nSecond:\n";
  msg.operator()(3);
  return 0;
}
```

Output:

```
First:
Welcome to MIT!
Welcome to MIT!

Second:
Welcome to MIT!
Welcome to MIT!
Welcome to MIT!
```

Function object – Cont'd

- **C++ Functor With Return Type and Parameters:**
  - We can also define a functor that accepts parameters and returns a value.

```
#include <iostream>
using namespace std;
class MUL
{
  public:
     int operator() (int a, int b) //Functor With Return Type and Parameters
     {
       return a * b;
     }
};
int main()
{
     MUL obj;
     int res = obj(12,8);
     cout << "12 * 8 = "<< res;
     return 0;
}
```

Output:

```
12 * 8 = 96
```

Function object – Cont'd

C++ Functor With a Member Variable

```
#include<iostream>
using namespace std;
class AddVal
{
     private:
     int n;

     public:
     AddVal(){n=0;}
     int operator()(int num)
     {
       return n + num;
     }
};
int main()
{
     AddVal x;
     int res = x(100);
     cout << "result:"<<res; n;
     return 0;
}
```

Output:

```
result:100
```

Function object – Cont'd

- **C++ Predefined Functors with STL**
  - The C++ language has a wide collection of predefined useful function objects or simply functors, defined inside the functional header file.

    #include <functional>
  - C++ provides the library functors for arithmetic, relational, and logical operations.

Arithmetic Functors

Functors	Description
plus	returns the sum of two parameters
minus	returns the difference of two parameters
multiplies	returns the product of two parameters
divides	returns the result after dividing two parameters
modulus	returns the remainder after dividing two parameters
negate	returns the negated value of a parameter

Relational Functors

Functors	Description
equal_to	returns true if the two parameters are equal
not_equal_to	returns true if the two parameters are not equal
greater	returns true if the first parameter is greater than the second
greater_equal	returns true if the first parameter is greater than or equal to the second
less	returns true if the first parameter is less than the second
less_equal	returns true if the first parameter is less than or equal to the second

Logical Functors

Functors	Description
logical_and	returns the result of **Logical AND** operation of two booleans
logical_or	returns the result of **Logical OR** operation of two booleans
logical_not	returns the result of **Logical NOT** operation of a boolean

Bitwise Functors

Functors	Description
bit_and	returns the result of **Bitwise AND** operation of two parameters
bit_or	returns the result of **Bitwise OR** operation of two parameters
bit_xor	returns the result of **Bitwise XOR** operation of two parameters

UNITII OVERVIEW OF C++

- **Topics to be discussed,**

➢ Classes and Objects

➢ Constructors and Destructors

➢ Operator Overloading and Type Conversions

➢ Function object

➢ **Dynamic Memory Management**

– (Already discussed in Unit 1)