

C++ OOPs Concepts

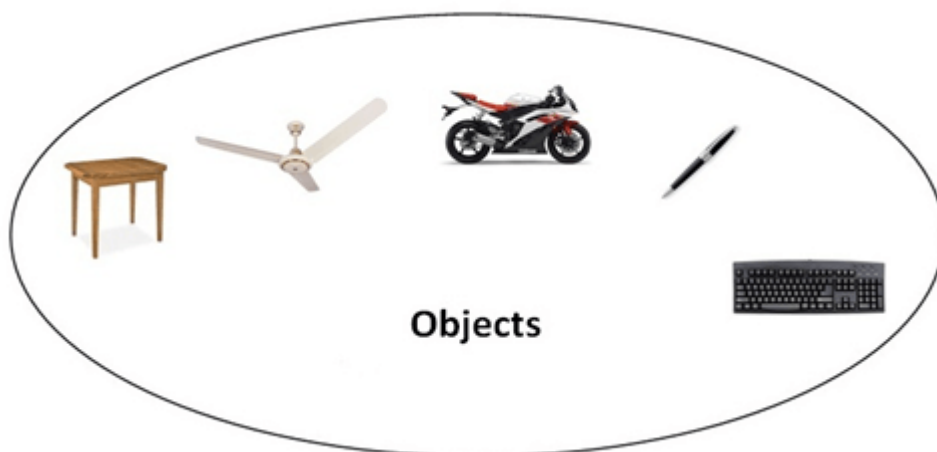
The major purpose of C++ programming is to introduce the concept of object orientation to the C programming language.

Object Oriented Programming is a paradigm that provides many concepts such as **inheritance, data binding, polymorphism etc.**

The programming paradigm where everything is represented as an object is known as truly object-oriented programming language. **Smalltalk** is considered as the first truly object-oriented programming language.

OOPs (Object Oriented Programming System)

Object means a real word entity such as pen, chair, table etc. **Object-Oriented Programming** is a methodology or paradigm to design a program using classes and objects. It simplifies the software development and maintenance by providing some concepts:



- Object
- Class
- Inheritance
- Polymorphism
- Abstraction
- Encapsulation

Object

Any entity that has state and behavior is known as an object. For example: chair, pen, table, keyboard, bike etc. It can be physical and logical.

Class

Collection of objects is called class. It is a logical entity.

Inheritance

When one object acquires all the properties and behaviours of parent object i.e. known as inheritance. It provides **code reusability**. It is used to achieve runtime polymorphism.

Polymorphism

When **one task is performed by different ways** i.e. known as polymorphism. For example: to convince the customer differently, to draw something e.g. shape or rectangle etc.

In C++, we use **Function overloading** and Function overriding to achieve polymorphism.

Abstraction

Hiding internal details and showing functionality is known as abstraction. For example: phone call, we don't know the internal processing.

In C++, we use abstract class and interface to achieve abstraction.

Encapsulation

Binding (or wrapping) code and data together into a single unit is known as encapsulation. For example: capsule, it is wrapped with different medicines.

Advantage of OOPs over Procedure-oriented programming language

1. OOPs makes development and maintenance easier where as in Procedure-oriented programming language it is not easy to manage if code grows as project size grows.
2. OOPs provide data hiding whereas in Procedure-oriented programming language a global data can be accessed from anywhere.

3. OOPs provide ability to simulate real-world event much more effectively. We can provide the solution of real word problem if we are using the Object-Oriented Programming language.
4. The prime purpose of C++ programming was to add object orientation to the C programming language, which is in itself one of the most powerful programming languages.
5. The core of the pure object-oriented programming is to create an object, in code, that has certain properties and methods. While designing C++ modules, we try to see whole world in the form of objects. For example a car is an object which has certain properties such as color, number of doors, and the like. It also has certain methods such as accelerate, brake, and so on.

There are a few principle concepts that form the foundation of object-oriented programming –

Object

This is the basic unit of object oriented programming. That is both data and function that operate on data are bundled as a unit called as object.

Class

When you define a class, you define a blueprint for an object. This doesn't actually define any data, but it does define what the class name means, that is, what an object of the class will consist of and what operations can be performed on such an object.

Abstraction

Data abstraction refers to, providing only essential information to the outside world and hiding their background details, i.e., to represent the needed information in program without presenting the details.

For example, a database system hides certain details of how data is stored and created and maintained. Similar way, C++ classes provides different methods to the outside world without giving internal detail about those methods and data.

Encapsulation

Encapsulation is placing the data and the functions that work on that data in the same place. While working with procedural languages, it is not always clear which functions work on which variables but object-oriented programming provides you framework to place the data and the relevant functions together in the same object.

Inheritance

One of the most useful aspects of object-oriented programming is **code reusability**. As the name suggests Inheritance is the process of forming a new class from an existing class that is from the existing class called as base class, new class is formed called as derived class.

This is a very important concept of object-oriented programming since this feature helps to reduce the code size.

Polymorphism

The ability to use an operator or function in different ways in other words giving different meaning or functions to the operators or functions is called polymorphism. Poly refers to many. That is a single function or an operator functioning in many ways different upon the usage is called polymorphism.

Overloading

The concept of overloading is also a branch of polymorphism. When the exiting operator or function is made to operate on new data type, it is said to be overloaded.

Friend Functions

It is possible to grant a **nonmember function access to the private members** of a class by using a **friend**. A friend function has access to all private and protected members of the class for which it is a friend.

To declare a friend function, include its prototype within the class, preceding it with the keyword **friend**. Consider this program:

```
#include <iostream>

using namespace std;

class myclass {
    int a, b;
public:
    friend int sum(myclass x);
    void set_ab(int i, int j);
};

void myclass::set_ab(int i, int j)
{
    a = i;
    b = j;
}

// Note: sum() is not a member function of any class.
int sum(myclass x)
{
    /* Because sum() is a friend of myclass, it can
    directly access a and b. */
    return x.a + x.b;
}

int main()
{
    myclass n;
    n.set_ab(3, 4);
    cout << sum(n);
}
```

```
return 0;
}
```

Friend Class

It is possible for **one class to be a friend of another class**. When this is the case, the friend class and all of its member functions have access to the private members defined within the other class. For example,

```
// Using a friend class.
```

```
#include <iostream>
```

```
using namespace std;
```

```
class TwoValues {
```

```
int a;
```

```
int b;
```

```
public:
```

```
TwoValues(int i, int j)
```

```
{
```

```
    a = i;
```

```
    b = j;
```

```
}
```

```
friend class Min;
```

```
};
```

```
class Min {
```

```
public:int min(TwoValues x);
```

```
};
```

```
int Min::min(TwoValues x)
```

```
{
```

```
return x.a < x.b ? x.a : x.b;
```

```
}
```

```
int main()
```

```
{
```

```
TwoValues ob(10, 20);
```

```
Min m;  
cout << m.min(ob);  
return 0;  
}
```

In this example, class Min has access to the private variables a and b declared within the TwoValues class. It is critical to understand that when one class is a friend of another, it only has access to names defined within the other class. It does not inherit the other class. Specifically, the members of the first class do not become members of the friend class. Friend classes are seldom used. They are supported to allow certain special case situations to be handled.

Data hiding is a fundamental concept of object-oriented programming. It restricts the access of private members from outside of the class.

Similarly, protected members can only be accessed by derived classes and are inaccessible from outside. For example,

```
class MyClass {  
    private:  
        int member1;  
}  
  
int main() {  
    MyClass obj;  
  
    // Error! Cannot access private members from here.  
    obj.member1 = 5;  
}
```

However, there is a feature in C++ called **friend functions** that break this rule and allow us to access member functions from outside the class. Similarly, there is a **friend class** as well, which we will learn later in this tutorial.

friend Function in C++

A **friend function** can access the **private** and **protected** data of a class. We declare a friend function using the `friend` keyword inside the body of the class.

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

Example 1: Working of friend Function

```
// C++ program to demonstrate the working of friend function  
  
#include <iostream>  
using namespace std;  
  
class Distance {  
    private:  
        int meter;  
  
        // friend function  
        friend int addFive(Distance);  
  
    public:  
        Distance() : meter(0) {}  
};  
  
// friend function definition  
int addFive(Distance d) {  
  
    //accessing private members from the friend function
```



```

        d.meter += 5;
        return d.meter;
    }

    int main() {
        Distance D;
        cout << "Distance: " << addFive(D);
        return 0;
    }

```

Output

```
Distance: 5
```

Here, `addFive()` is a friend function that can access both **private** and **public** data members.

Though this example gives us an idea about the concept of a friend function, it doesn't show any meaningful use.

A more meaningful use would be operating on objects of two different classes. That's when the friend function can be very helpful.

Example 2: Add Members of Two Different Classes

```

// Add members of two different classes using friend functions

#include <iostream>
using namespace std;

// forward declaration
class ClassB;

class ClassA {

    public:
        // constructor to initialize numA to 12
        ClassA() : numA(12) {}

```

```

    private:
        int numA;

        // friend function declaration
        friend int add(ClassA, ClassB);
};

class ClassB {

    public:
        // constructor to initialize numB to 1
        ClassB() : numB(1) {}

    private:
        int numB;

        // friend function declaration
        friend int add(ClassA, ClassB);
};

// access members of both classes
int add(ClassA objectA, ClassB objectB) {
    return (objectA.numA + objectB.numB);
}

int main() {
    ClassA objectA;
    ClassB objectB;
    cout << "Sum: " << add(objectA, objectB);
    return 0;
}

```

Output

```
Sum: 13
```

In this program, `ClassA` and `ClassB` have declared `add()` as a friend function. Thus, this function can access **private** data of both classes. One thing to notice here is the friend function inside `ClassA` is using the `ClassB`. However, we haven't defined `ClassB` at this point.

```
// inside classA
```

```
friend int add(ClassA, ClassB);
```

For this to work, we need a forward declaration of `ClassB` in our program.

```
// forward declaration  
class ClassB;
```

friend Class in C++

We can also use a friend Class in C++ using the `friend` keyword. For example,

```
class ClassB;  
  
class ClassA {  
    // ClassB is a friend class of ClassA  
    friend class ClassB;  
    ... ..  
}  
  
class ClassB {  
    ... ..  
}
```

When a class is declared a friend class, all the member functions of the friend class become friend functions.

Since `ClassB` is a friend class, we can access all members of `ClassA` from inside `ClassB`.

However, we cannot access members of `ClassB` from inside `ClassA`. It is because friend relation in C++ is only granted, not taken.

Example 3: C++ friend Class

```
// C++ program to demonstrate the working of friend class

#include <iostream>
using namespace std;

// forward declaration
class ClassB;

class ClassA {
private:
    int numA;

    // friend class declaration
    friend class ClassB;

public:
    // constructor to initialize numA to 12
    ClassA() : numA(12) {}
};

class ClassB {
private:
    int numB;

public:
    // constructor to initialize numB to 1
    ClassB() : numB(1) {}

    // member function to add numA
    // from ClassA and numB from ClassB
    int add() {
        ClassA objectA;
        return objectA.numA + numB;
    }
};

int main() {
    ClassB objectB;
    cout << "Sum: " << objectB.add();
    return 0;
}
```

Output

Sum: 13

Here, `ClassB` is a friend class of `ClassA`. So, `ClassB` has access to the members of `ClassA`.

In `ClassB`, we have created a function `add()` that returns the sum of `numA` and `numB`.

Since `ClassB` is a friend class, we can create objects of `ClassA` inside of `ClassB`.

Inline function

There is an important feature in C++, called an inline function, that is commonly used with classes. will make heavy use of it, inline functions are examined here. In C++, you can create **short functions** that are not actually called; rather, their code is expanded in line at the point of each invocation. This process is similar to using a function-like macro.

To cause a function to be expanded in line rather than called, precede its definition with the inline keyword. For example, in this program, the function `max()` is expanded in line instead of called:

```
#include <iostream>

using namespace std;

inline int max(int a, int b)
{
    return a>b ? a : b;
}

int main()
{
    cout << max(10, 20);
    cout << " " << max(99, 88);
    return 0;
}
```

The reason that inline functions are an important addition to C++ is that they allow you to create very efficient code. Since classes typically require several frequently executed interface functions (which provide access to private data), the efficiency of these functions is of critical concern. As you probably know, each time a function is called, a significant amount of overhead is generated by the calling and return mechanism. Typically, arguments are pushed onto the stack and various registers are saved when a function is called, and then restored when the function returns. The trouble is that these instructions take time. However, when a function is expanded in line, none of those operations occur. Although expanding function calls in line can produce faster run times, it can also result in larger code size because of duplicated code. For this reason, it is best to inline only very small functions. Further, it is also a good idea to inline only those functions that will have significant impact on the performance of your program.

In C++, inheritance is supported by allowing one class to incorporate another class into its declaration. Inheritance allows a hierarchy of classes to be built, moving from most general to most specific.

The process involves first defining a **base class**, which defines those qualities common to all objects to be **derived from the base**. The base class represents the most general description. The classes derived from the base are usually referred to as derived classes.

A derived class includes all features of the generic base class and then adds qualities specific to the derived class.

the building class is declared, as shown here. It will serve as the base for two derived classes

```
class building {
    int rooms;
    int floors;
    int area;

public:
    void set_rooms(int num);
    int get_rooms();
    void set_floors(int num);
    int get_floors();
    void set_area(int num);
    int get_area();
};

// house is derived from building
class house : public building {
    int bedrooms;
    int baths;
public:
    void set_bedrooms(int num);
    int get_bedrooms();
    void set_baths(int num);
    int get_baths();
};
```

Notice how **building** is inherited. The general form for inheritance is

```
class derived-class : access base-class {
    // body of new class
}
```

A derived class has direct access to both its own members and the public members of the base class

Here is a program illustrating inheritance. It creates two derived classes of building using inheritance; one is house, the other, school


```
#include <iostream>

using namespace std;

class building { //main class//
int rooms; //private//
int floors;
int area;
public:
void set_rooms(int num);
int get_rooms();
void set_floors(int num);
int get_floors();
void set_area(int num);
int get_area();
};

// house is derived from main building class//
class house : public building {
int bedrooms;
int baths;
public:
void set_bedrooms(int num);
int get_bedrooms();
void set_baths(int num);
int get_baths();
};

// school is also derived from main class or base class building//
class school : public building {
int classrooms;
int offices;
public:
```

```
void set_classrooms(int num);
int get_classrooms();
void set_offices(int num);
int get_offices();
};

void building::set_rooms(int num) //this for base class//
{
rooms = num;
}

void building::set_floors(int num)
{
floors = num;
}

void building::set_area(int num)
{
area = num;
}

int building::get_rooms()
{
return rooms;
}

int building::get_floors()
{
return floors;
}

int building::get_area()
{
return area;
}

void house::set_bedrooms(int num) //this is for derived class house//
```

```
{
bedrooms = num;
}
void house::set_baths(int num)
{
    baths = num;
}
int house::get_bedrooms()
{
return bedrooms;
}
int house::get_baths()
{
return baths;
}
void school::set_classrooms(int num) // this is for derived class school//
{
classrooms = num;
}
void school::set_offices(int num)
{
offices = num;
}
int school::get_classrooms()
{
return classrooms;
}
int school::get_offices()
{
return offices;
```

```
}  
  
int main()  
{  
    house h; //object h for house class//  
    school s; //object s for school class//  
    h.set_rooms(12);  
    h.set_floors(3);  
    h.set_area(4500);  
    h.set_bedrooms(5);h.set_baths(3);  
    cout << "house has " << h.get_bedrooms();  
    cout << " bedrooms\n";  
    s.set_rooms(200);  
    s.set_classrooms(180);  
    s.set_offices(5);  
    s.set_area(25000);  
    cout << "school has " << s.get_classrooms();  
    cout << " classrooms\n";  
    cout << "Its area is " << s.get_area();  
    return 0;  
}
```

Operator Overloading

The ability to overload operators is one of C++'s most powerful features. It allows the full integration of new class types into the programming environment.

After overloading the appropriate operators, you can use objects in expressions in just the same way that you use C++'s built-in data types. Operator overloading also forms the basis of C++'s approach to I/O.

You overload operators by creating **operator functions**. An operator function defines the operations that the overloaded operator will perform relative to the class upon which it will work. An operator function is created using the keyword **operator**.

Operator functions can be either members or nonmembers of a class. Nonmember operator functions are almost always friend functions of the class, however. The way operator functions are written differs between member and nonmember functions. Therefore, each will be examined separately, beginning with member operator functions.

Operator overloading

Operator overloading is a specific case of polymorphism in which some or all operators like +, = or == are treated as polymorphic functions and as such have different behaviors depending on the types of its arguments..

You can overload any of the following operators:

+	-	*	/	%	^	&		~
!	=	<	>	+=	-=	*=	/=	%=
^=	&=	=	<<	>>	<<=	>>=	==	!=
<=	>=	&&		++	--	,	->*	->

- You cannot overload the following operators:
- ., *, ::, ?:
- and the preprocessor symbols # and ##.

Creating a Member Operator Function

A member operator function takes this general form:

```
ret-type class-name::operator#(arg-list)
{
    // operations
}
```

Often, operator functions return an object of the class they operate on, but *ret-type* can be any valid type. The # is a placeholder. When you create an operator function, substitute the operator for the #. For example, if you are overloading the / operator, use **operator/**. When you are overloading a unary operator, *arg-list* will be empty. When you are overloading binary operators, *arg-list* will contain one parameter. (The reasons for this seemingly unusual situation will be made clear in a moment.)

Here is a simple first example of operator overloading. This program creates a class called **loc**, which stores longitude and latitude values. It overloads the + operator relative to this class. Examine this program carefully, paying special attention to the definition of **operator+()**:

```
#include <iostream>

using namespace std;

class loc {
    int longitude, latitude;
public:
    loc() {}
    loc(int lg, int lt) {
        longitude = lg;
        latitude = lt;
    }
    void show() {
        cout << longitude << " ";
        cout << latitude << "\n";
    }
    loc operator+(loc op2); //operator function//
};

// Overload + for loc.
loc loc::operator+(loc op2)
{
```

```

loc temp;
temp.longitude = op2.longitude + longitude;
temp.latitude = op2.latitude + latitude;
return temp;
}
int main()
{
loc ob1(10, 20), ob2( 5, 30);
ob1.show(); // displays 10 20
ob2.show(); // displays 5 30
ob1 = ob1 + ob2;
ob1.show(); // displays 15 50
return 0;
}

```

As you can see, **operator+()** has only one parameter even though it overloads the binary **+** operator. (You might expect two parameters corresponding to the two operands of a binary operator.) The reason that **operator+()** takes only one parameter is that the operand on the left side of the **+** is passed implicitly to the function through the **this** pointer. The operand on the right is passed in the parameter **op2**. The fact that the left operand is passed using **this** also implies one important point: When binary operators are overloaded, it is the object on the left that generates the call to the operator function.

As mentioned, it is common for an overloaded operator function to return an object of the class it operates upon. By doing so, it allows the operator to be used in larger expressions. For example, if the **operator+()** function returned some other type, this expression would not have been valid:

```
ob1 = ob1 + ob2;
```

In order for the sum of **ob1** and **ob2** to be assigned to **ob1**, the outcome of that operation must be an object of type **loc**.

Further, having **operator+()** return an object of type **loc** makes possible the following statement:

```
(ob1+ob2).show(); // displays outcome of ob1+ob2
```

In order for the sum of **ob1** and **ob2** to be assigned to **ob1**, the outcome of that operation must be an object of type **loc**.

Further, having **operator+()** return an object of type **loc** makes possible the following statement:

```
(ob1+ob2).show(); // displays outcome of ob1+ob2
```

In this situation, **ob1+ob2** generates a temporary object that ceases to exist after the call to **show()** terminates.

It is important to understand that an operator function can return any type and that the type returned depends solely upon your specific application. It is just that, often, an operator function will return an object of the class upon which it operates.

One last point about the **operator+()** function: It does not modify either operand. Because the traditional use of the **+** operator does not modify either operand, it makes sense for the overloaded version not to do so either. (For example, 5+7 yields 12, but neither 5 nor 7 is changed.) Although you are free to perform any operation you want inside an operator function, it is usually best to stay within the context of the normal use of the operator.

The next program adds three additional overloaded operators to the **loc** class: the **-**, the **=**, and the unary **++**. Pay special attention to how these functions are defined.

```
#include <iostream>

using namespace std;

class loc {
int longitude, latitude;public:
loc() {} // needed to construct temporaries
loc(int lg, int lt) {
longitude = lg;
latitude = lt;
}
void show() {
cout << longitude << " ";
cout << latitude << "\n";
}
loc operator+(loc op2);
loc operator-(loc op2);
loc operator=(loc op2);
loc operator++();
};
```



```

// Overload + for loc.
loc loc::operator+(loc op2)
{
    loc temp;
    temp.longitude = op2.longitude + longitude;
    temp.latitude = op2.latitude + latitude;
    return temp;
}

// Overload - for loc.
loc loc::operator-(loc op2)
{
    loc temp;
    // notice order of operands
    temp.longitude = longitude - op2.longitude;
    temp.latitude = latitude - op2.latitude;
    return temp;
}

// Overload assignment for loc.
loc loc::operator=(loc op2)
{
    longitude = op2.longitude;
    latitude = op2.latitude;
    return *this; // i.e., return object that generated call
}

// Overload prefix ++ for loc.
loc loc::operator++()
{
    longitude++;
    latitude++;
    return *this;
}

```

```

}

int main()
{
    loc ob1(10, 20), ob2( 5, 30), ob3(90, 90);
    ob1.show();
    ob2.show();
    ++ob1;
    ob1.show(); // displays 11 21
    ob2 = ++ob1;
    ob1.show(); // displays 12 22
    ob2.show(); // displays 12 22
    ob1 = ob2 = ob3; // multiple assignment
    ob1.show(); // displays 90 90
    ob2.show(); // displays 90 90
    return 0;
}

```

from the data pointed to by **this**. It is important to remember which operand generates the call to the function.

In C++, if the **=** is not overloaded, a default assignment operation is created automatically for any class you define. The default assignment is simply a member- by-member, bitwise copy. By overloading the **=**, you can define explicitly what the assignment does relative to a class. In this example, the overloaded **=** does exactly the same thing as the default, but in other situations, it could perform other operations. Notice that the **operator=()** function returns ***this**, which is the object that generated the call. This arrangement is necessary if you want to be able to use multiple assignment operations such as this:

```

ob1 = ob2 = ob3; // multiple assignment

```

Now, look at the definition of **operator++()**. As you can see, it takes no parameters. Since **++** is a unary operator, its only operand is implicitly passed by using the **this** pointer.

Notice that both **operator=()** and **operator++()** alter the value of an operand. In the case of assignment, the operand on the left (the one generating the call to the **operator=()** function) is assigned a new value. In the case of the **++**, the operand is incremented. As stated previously, although you are free to make these operators do anything you please, it is almost always wisest to stay consistent with their original meanings.

Types of Operator Overloading in C++

Operator Overloading:

C++ provides a special function to change the current functionality of some operators within its class which is often called as **operator overloading**. Operator Overloading is the method by which we can change the function of some specific operators to do some different task.

This can be done by declaring the function, its syntax is,

```
Return_Type classname :: operator op(Argument list)
```

```
{
```

```
    Function Body
```

```
}
```

In the above syntax **Return_Type** is value type to be returned to another object, operator op is the function where the operator is a keyword and op is the operator to be overloaded.

Operator function must be either non-static (member function) or friend function.

Operator Overloading can be done by using **three approaches**, they are

1. Overloading unary operator.
2. Overloading binary operator.
3. Overloading binary operator using a friend function.

Below are some criteria/rules to define the operator function:

- In case of a non-static function, the binary operator should have only one argument and unary should not have an argument.
- In the case of a friend function, the binary operator should have only two argument and unary should have only one argument.
- All the class member object should be public if operator overloading is implemented.
- Operators that cannot be overloaded are **.,* :: ?:**
- Operator cannot be used to overload when declaring that function as friend function **= () [] ->**.

Refer this, for more rules of [Operator Overloading](#)

1. **Overloading Unary Operator:** Let us consider to overload (-) unary operator. In unary operator function, no arguments should be passed. It works only with one class objects. It is a overloading of

an operator operating on a single operand.

Example:

Assume that class Distance takes two member object i.e. feet and inches, create a function by which Distance object should decrement the value of feet and inches by 1 (having single operand of Distance Type).

```
// C++ program to show unary operator overloading
#include <iostream>

using namespace std;

class Distance {
public:

    // Member Object
    int feet, inch;

    // Constructor to initialize the object's value
    Distance(int f, int i)
    {
        this->feet = f;
        this->inch = i;
    }

    // Overloading(-) operator to perform decrement
    // operation of Distance object
    void operator-()
    {
        feet--;
        inch--;
        cout << "\nFeet & Inches(Decrement): " << feet <<
"" << inch;
    }
};

// Driver Code
int main()
{
    // Declare and Initialize the constructor
    Distance d1(8, 9);

    // Use (-) unary operator by single operand
    -d1;
    return 0;
}
```

Output:

Feet & Inches(Decrement): 7'8

1. In the above program, it shows that no argument is passed and no return_type value is returned, because unary operator works on a single operand. (-) operator change the functionality to its member function.

Note: d2 = -d1 will not work, because operator-() does not return any value.

2. **Overloading Binary Operator:** In binary operator overloading function, there should be one argument to be passed. It is overloading of an operator operating on two operands. Let's take the same example of class Distance, but this time, add two distance objects.

- CPP

```
// C++ program to show binary operator overloading
#include <iostream>
```

```
using namespace std;
```

```
class Distance {
public:
```

```
    // Member Object
    int feet, inch;
    // No Parameter Constructor
    Distance()
    {
        this->feet = 0;
        this->inch = 0;
    }
```

```
    // Constructor to initialize the object's value
    // Parameterized Constructor
    Distance(int f, int i)
    {
        this->feet = f;
        this->inch = i;
    }
```

```
    // Overloading (+) operator to perform addition of
    // two distance object
    Distance operator+(Distance& d2) // Call by
reference
    {
        // Create an object to return
```

```

        Distance d3;

        // Perform addition of feet and inches
        d3.feet = this->feet + d2.feet;
        d3.inch = this->inch + d2.inch;

        // Return the resulting object
        return d3;
    }
};

// Driver Code
int main()
{
    // Declaring and Initializing first object
    Distance d1(8, 9);

    // Declaring and Initializing second object
    Distance d2(10, 2);

    // Declaring third object
    Distance d3;

    // Use overloaded operator
    d3 = d1 + d2;

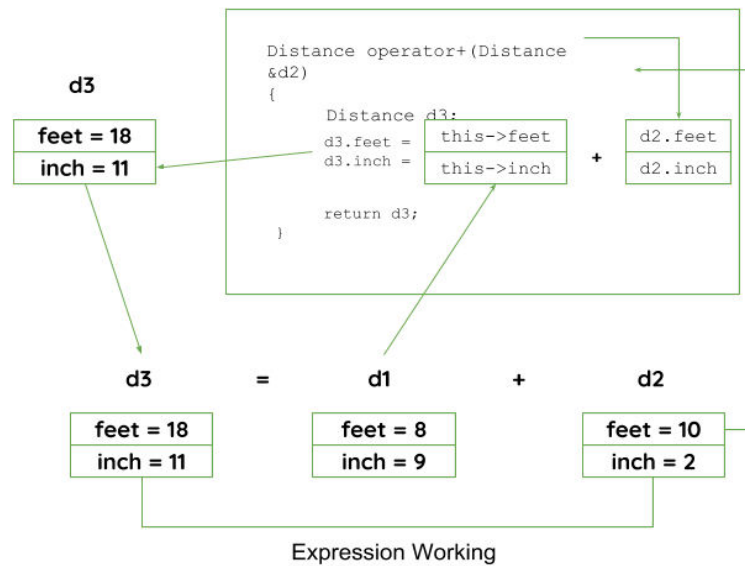
    // Display the result
    cout << "\nTotal Feet & Inches: " << d3.feet << " " <<
d3.inch;
    return 0;
}

```

Output:

Total Feet & Inches: 18'11

1. Here in the above program,
See Line no. 26, Distance operator+(Distance &d2), here return type of function is distance and it uses call by references to pass an argument.
See Line no. 49, d3 = d1 + d2; here, d1 calls the operator function of its class object and takes d2 as a parameter, by which operator function return object and the result will reflect in the d3 object.
Pictorial View of working of Binary Operator:



- Overloading Binary Operator using a Friend function:** In this approach, the operator overloading function must precede with friend keyword, and declare a function class scope. Keeping in mind, friend operator function takes two parameters in a binary operator, varies one parameter in a unary operator. All the working and implementation would same as binary operator function except this function will be implemented outside of the class scope. Let's take the same example using the friend function.

```
// C++ program to show binary operator overloading
#include <iostream>
```

```
using namespace std;
```

```
class Distance {
public:
```

```
    // Member Object
    int feet, inch;
```

```
    // No Parameter Constructor
    Distance()
    {
        this->feet = 0;
        this->inch = 0;
    }
```

```
    // Constructor to initialize the object's value
    // Parameterized Constructor
    Distance(int f, int i)
    {
```

```

        this->feet = f;
        this->inch = i;
    }

    // Declaring friend function using friend keyword
    friend Distance operator+(Distance&, Distance&);
};

// Implementing friend function with two parameters
Distance operator+(Distance& d1, Distance& d2) //
Call by reference
{
    // Create an object to return
    Distance d3;

    // Perform addition of feet and inches
    d3.feet = d1.feet + d2.feet;
    d3.inch = d1.inch + d2.inch;

    // Return the resulting object
    return d3;
}

// Driver Code
int main()
{
    // Declaring and Initializing first object
    Distance d1(8, 9);

    // Declaring and Initializing second object
    Distance d2(10, 2);

    // Declaring third object
    Distance d3;

    // Use overloaded operator
    d3 = d1 + d2;

    // Display the result
    cout << "\nTotal Feet & Inches: " << d3.feet << " " <<
d3.inch;
    return 0;
}

```

Output:

Total Feet & Inches: 18'11

1. Here in the above program, operator function is implemented outside of class scope by declaring that function as the friend function.

In these ways, an operator can be overloaded to perform certain tasks by changing the functionality of operators.

Operator Overloading Using a Friend Function

You can overload an operator for a class by using a nonmember function, which is usually a friend of the class. Since a friend function is not a member of the class, it does not have a this pointer. Therefore, an overloaded friend operator function is passed the operands explicitly. This means that a friend function that overloads a binary operator has two parameters, and a friend function that overloads a unary operator has one parameter. When overloading a binary operator using a friend function, the left operand is passed in the first parameter and the right operand is passed in the second parameter. In this program, the operator+() function is made into a friend:

```
#include <iostream>

using namespace std;

class loc {
    int longitude, latitude;

public:loc() {} // needed to construct temporaries
    loc(int lg, int lt) {
        longitude = lg;
        latitude = lt;
    }

    void show() {
        cout << longitude << " ";
        cout << latitude << "\n";
    }

    friend loc operator+(loc op1, loc op2); // now a friend
    loc operator-(loc op2);
    loc operator=(loc op2);
    loc operator++();
};
```

```

// Now, + is overloaded using friend function.
loc operator+(loc op1, loc op2)
{
    loc temp;
    temp.longitude = op1.longitude + op2.longitude;
    temp.latitude = op1.latitude + op2.latitude;
    return temp;
}

// Overload - for loc.
loc loc::operator-(loc op2)
{
    loc temp;
    // notice order of operands
    temp.longitude = longitude - op2.longitude;
    temp.latitude = latitude - op2.latitude;
    return temp;
}

// Overload assignment for loc.
loc loc::operator=(loc op2)
{
    longitude = op2.longitude;
    latitude = op2.latitude;
    return *this; // i.e., return object that generated call
}

// Overload ++ for loc.
loc loc::operator++()
{
    longitude++;
    latitude++;
    return *this;
}

```

```
}  
int main()  
{  
loc ob1(10, 20), ob2( 5, 30);  
ob1 = ob1 + ob2;  
ob1.show();  
return 0;  
}
```

Exception Handling Fundamentals

When executing C++ code, different errors can occur: coding errors made by the programmer, errors due to wrong input, or other unforeseeable things.

When an error occurs, C++ will normally stop and generate an error message. The technical term for this is: C++ will throw an **exception** (throw an error).

C++ try and catch

Exception handling in C++ consist of three keywords: **try, throw and catch:**

The **try** statement allows you to define a block of code to be tested for errors while it is being executed.

The **throw** keyword throws an exception when a problem is detected, which lets us create a custom error.

The **catch** statement allows you to define a block of code to be executed, if an error occurs in the try block.

C++ exception handling is built upon three keywords: **try**, **catch**, and **throw**. In the most general terms, program statements that you want to monitor for exceptions are contained in a **try** block. If an exception (i.e., an error) occurs within the **try** block, it is thrown (using **throw**). The exception is caught, using **catch**, and processed. The following discussion elaborates upon this general description.

Code that you want to monitor for exceptions must have been executed from within a **try** block. (Functions called from within a **try** block may also throw an exception.) Exceptions that can be thrown by the monitored code are caught by a **catch** statement, which immediately follows the **try** statement in which the exception was thrown. The general form of **try** and **catch** are shown here.

```

try {
    // try block
}
catch (type1 arg) {
    // catch block
}
catch (type2 arg) {
    // catch block
}
catch (type3 arg) {
    // catch block
}
.
.
.
catch (typeN arg) {
    // catch block
}

```

The **try** can be as short as a few statements within one function or as all, encompassing as enclosing the **main()** function code within a **try block** (which effectively causes the entire program to be monitored).

When an exception is thrown, it is caught by its corresponding **catch** statement, which processes the exception. There can be more than one **catch** statement associated with a **try**. Which **catch** statement is used is determined by the type of the exception. That is, if the data type specified by a **catch** matches that of the exception, then that **catch** statement is executed (and all others are bypassed). When an exception is caught, *arg* will receive its value. Any type of data may be caught, including classes that you create. If no exception is thrown (that is, no error occurs within the **try** block), then no **catch** statement is executed.

The general form of the **throw** statement is shown here:

```
throw exception;
```

throw generates the exception specified by *exception*. If this exception is to be caught, then **throw** must be executed either from within a **try** block itself, or from any function called from within the **try** block (directly or indirectly).

If you throw an exception for which there is no applicable catch statement, an abnormal program termination may occur. Throwing an unhandled exception causes the standard library function **terminate()** to be invoked. By default, **terminate()** calls **abort()** to stop your program, but you can specify your own termination handler.

// A simple exception handling example.

```

#include <iostream>
using namespace std;
int main()
{
    cout << "Start\n";
    try { // start a try block
        cout << "Inside try block\n";
        throw 100; // throw an error
        cout << "This will not execute";
    }
    catch (int i) { // catch an error
        cout << "Caught an exception -- value is: ";
        cout << i << "\n";
    }
    cout << "End";
    return 0;
}
.....

```

Look carefully at this program. As you can see, there is a **try** block containing three statements and a **catch(int i)** statement that processes an integer exception. Within the **try** block, only two of the three statements will execute: the first **cout** statement and the **throw**. Once an exception has been thrown, control passes to the **catch** expression and the **try** block is terminated. That is, **catch** is *not* called. Rather, program execution is transferred to it. (The program's stack is automatically reset as needed to accomplish this.) Thus, the **cout** statement following the **throw** will never execute.

Usually, the code within a **catch** statement attempts to remedy an error by taking appropriate action. If the error can be fixed, execution will continue with the statements following the **catch**. However, often an error cannot be fixed and a **catch** block will terminate the program with a call to **exit()** or **abort()**.

As mentioned, the type of the exception must match the type specified in a **catch** statement. For example, in the preceding example, if you change the type in the **catch** statement to **double**, the exception will not be caught and abnormal termination will occur. This change is shown here.

```

// This example will not work.
#include <iostream>
using namespace std;


```

```

int main()
{
    cout << "Start\n";
    try { // start a try block
        cout << "Inside try block\n";
        throw 100; // throw an error
        cout << "This will not execute";
    }
    catch (double i) { // won't work for an int exception
        cout << "Caught an exception -- value is: ";
        cout << i << "\n";
    }
    cout << "End";
    return 0;
}

```

This program produces the following output because the integer exception will not be caught by the **catch(double i)** statement. (Of course, the precise message describing abnormal termination will vary from compiler to compiler.)



```

Start
Inside try block
Abnormal program termination

```

Using Multiple catch Statements

As stated, you can have more than one catch associated with a try. In fact, it is common to do so. However, each catch must catch a different type of exception. For example, this program catches both integers and strings.

```

#include <iostream>
using namespace std;

// Different types of exceptions can be caught.
void Xhandler(int test)
{
    try{
        if(test) throw test;
    }
}

```

```

else throw "Value is zero";
}
catch(int i) {
cout << "Caught Exception #: " << i << '\n';
}
catch(const char *str) {
cout << "Caught a string: ";
cout << str << '\n';
}
}
int main()
{
cout << "Start\n";
Xhandler(1);
Xhandler(2);
Xhandler(0);
Xhandler(3);
cout << "End";
return 0;
}

```

Practice programs

// This example uses catch(...) as a default.

```
#include <iostream>
```

```
using namespace std;
```

```
void Xhandler(int test)
```

```
{
```

```
try{
```

```
if(test==0) throw test; // throw int
```

```
if(test==1) throw 'a'; // throw char
```

```
if(test==2) throw 123.23; // throw double
```



```

}
catch(int i) { // catch an int exception
cout << "Caught an integer\n";
}
catch(...) { // catch all other exceptions
cout << "Caught One!\n";
}
}

int main()
{
cout << "Start\n";
Chapter 19: Exception Handling 499 C++Xhandler(0);
Xhandler(1);
Xhandler(2);
cout << "End";
return 0;
}
.....
// Restricting function throw types.
#include <iostream>
using namespace std;
// This function can only throw ints, chars, and doubles.
void Xhandler(int test) throw(int, char, double)
{
if(test==0) throw test; // throw int
if(test==1) throw 'a'; // throw char
if(test==2) throw 123.23; // throw double
}

int main()
{
cout << "start\n";

```

```

try{
Xhandler(0); // also, try passing 1 and 2 to Xhandler()
}
catch(int i) {
cout << "Caught an integer\n";
}
catch(char c) {
cout << "Caught char\n";
}
catch(double d) {
cout << "Caught double\n";
}
cout << "end";
return 0;
}

```

Understanding **terminate()** and **unexpected()**

As mentioned earlier, **terminate()** and **unexpected()** are called when something goes wrong during the exception handling process. These functions are supplied by the Standard C++ library. Their prototypes are shown here:

```

void terminate( );
void unexpected( );

```

These functions require the header **<exception>**.

The **terminate()** function is called whenever the exception handling subsystem fails to find a matching **catch** statement for an exception. It is also called if your program attempts to rethrow an exception when no exception was originally thrown. The **terminate()** function is also called under various other, more obscure circumstances. For example, such a circumstance could occur when, in the process of unwinding the stack because of an exception, a destructor for an object being destroyed throws an exception. In general, **terminate()** is the handler of last resort when no other handlers for an exception are available. By default, **terminate()** calls **abort()**.

Pointers to Objects

Just as you can have pointers to other types of variables, you can have pointers to objects. When accessing members of a class given a pointer to an object, **use the arrow (->) operator instead of the dot operator**. The next program illustrates how to access an object given a pointer to it:

```
#include <iostream>
using namespace std;
class cl {
int i;
public:
cl(int j) { i=j; }
int get_i() { return i; }
};
int main()
{
cl ob(88), *p;
p = &ob; // get address of ob
cout << p->get_i(); // use -> to call get_i()
return 0;
}
.....

#include <iostream>
using namespace std;
class cl {
int i;
public:
cl() { i=0; }
cl(int j) { i=j; }
int get_i() { return i; }
};
```

```

int main()
{
    cl ob[3] = {1, 2, 3};
    cl *p;
    int i;
    p = ob; // get start of array
    for(i=0; i<3; i++) {
        cout << p->get_i() << "\n";
        p++; // point to next object
    }
    return 0;
}
.....

```

```

#include <iostream>
using namespace std;
class cl {
public:
    int i;
    cl(int j) { i=j; }
};
int main()
{
    cl ob(1);
    int *p;
    p = &ob.i; // get address of ob.i
    cout << *p; // access ob.i via p
    return 0;
}

```

Arrays: Arrays of Objects

In C++, it is possible to have **arrays of objects**. The syntax for declaring and using an object array is exactly the same as it is for any other type of array. For example, this program uses a three-element array of objects:

```
#include <iostream>

using namespace std;

class cl {
int i;
public:
void set_i(int j) { i=j; }
int get_i() { return i; }
};

int main()
{
cl ob[3];
int i;
for(i=0; i<3; i++) ob[i].set_i(i+1);
for(i=0; i<3; i++)
cout << ob[i].get_i() << "\n";
return 0;
}
.....
#include <iostream>
using namespace std;
class cl {
int i;
public:
cl(int j) { i=j; } // constructor
int get_i() { return i; }
};

int main()
```

```

{
cl ob[3] = {1, 2, 3}; // initializers
int i;
for(i=0; i<3; i++)
cout << ob[i].get_i() << "\n";
return 0;
}
.....
#include <iostream>
using namespace std;
class cl {
int h;
int i;
public:
cl(int j, int k) { h=j; i=k; } // constructor with 2 parameters
int get_i() {return i;}
int get_h() {return h;}
};
int main()
{
cl ob[3] = {
cl(1, 2), // initialize
cl(3, 4),
cl(5, 6)
};
int i;
for(i=0; i<3; i++) {
cout << ob[i].get_h();
cout << ", ";
cout << ob[i].get_i() << "\n";
}
}

```

```
return 0;
```

```
}
```

```
.....
```

Problem statement: Write a program to assign passengers seats in an airplane. Assume a small airplane with seat numbering as follows:

```
1 A B C D
2 A B C D
3 A B C D
4 A B C D
5 A B C D
6 A B C D
7 A B C D
```

The program should display the seat pattern, with an 'X' marking the seats already assigned. After displaying the seats available, the program prompts the seat desired, the user types in a seat, and then the display of available seats is updated. This continues until all seats are filled or until the user signals that the program should end. If the user types in a seat that is already assigned, the program should say that the seat is occupied and ask for another choice.

Input Example:

For example, after seats 1A, 2B, and 4C are taken, the display should look like:

```
1 X B C D
2 A X C D
3 A B C D
4 A B X D
5 A B C D
6 A B C D
7 A B C D
```

Solution

The whole problem can be implemented with help of 4 major functions:

1. **getData()**
2. **display()**
3. **check()**
4. **update()**

The entire problem is discussed dividing into parts focusing on functional modularity.

1. Initialize a 2-D array to represent the seat matrix

A 2-D character array is used to represent the seat matrix where the first column have the row number of each seat & the rest of the columns have four seat A,B,C,D respectively. Thus it's a 7X5 2-D array to represent the airplane seat matrix which looks like following:

2. Take user input for seat no

User is requested to input the desired seat no by giving corresponding no of seat. Like a valid seat no is 1A, 3C, 7D and so on, whereas, an invalid seat request can be 6F, 0B so on.

The input is taken by our function **getData()** which takes user input & returns the string.

3. Check the seat no request (check())

Our major function for this problem is to check the validity of the seat request & update seat matrix status as per request.

- Firstly it checks whether the user input is in the range 1A to 7D or not. If user input is out of range a prompt out **"invalid request"** & continue for further request.
- Check whether user input is **'N'** or not. If it's **'N'** then user wants to end the program. Terminate.
- if seat request is valid but already occupied
Then prompt a message stating "It's already occupied"
This checking can be done by founding the 2-D array row & column index for the corresponding seat.
Let, **row_index=r&column_index=c**
If(seat_matrix[row_index][column_index] == 'X')
Seat is occupied.
- **ELSE** seat request is a valid one and not occupied still
Update()

4. Update the valid entry

If the request is to update the valid seat we simple change its value to **'X'**. It can be done by finding row & column index and updating the value of **seat_matrix** at that location to **'X'**.

5. Special function to check whether all seats are occupied

The program also need to be terminated when all seats are occupied. Thus every time we keep a checking whether all the entry of **seat_matrix** is **'X'** or not.

```
//Airline Seat Reservation Problem//
```

```
#include <bits/stdc++.h>
```

```
using namespace std;
```

```
// to check whether all sits are occupied or not
```

```
int allOccupied(char arr[7][5]){
```

```
    int count=0;
```

```
    for(int i=0;i<7;i++){
```

```
        for(int j=1;j<5;j++){
```

```
            if(arr[i][j]=='X')
```

```
                count++;
```

```
        }
```

```
    if(count==28)
```

```
        return 1;
```

```
    return 0;
```

```
}
```

```
//to display the sits
```

```
void display(char arr[7][5]){
```

```
    for(int i=0;i<7;i++){
```

```
        for(int j=0;j<5;j++){
```

```
            cout<<arr[i][j]<<" ";
```

```
        }
```

```
        cout<<endl;
```

```
    }
```

```
    return;
```

```
}
```

```
//take user data
```

```
string getData(){
```

```

        string p;
        cout<<"enter valid seat no to check(like 1B) or N to end: ";
        cin>>p;
        return p;
    }

```

```

//update sit status
void update(char arr[7][5],int row,int col){
    cout<<"congrats, your seat is valid. Reserved for you\n";
    cout<<"updated seat status.....\n";
    arr[row][col]='X';
}

```

```

//checking whether user request for
//his sit no can be processed or not
int check(char arr[7][5],string s){
    //if user input is not in the range 1A to 7D
    if(s[0]>'7' || s[0]<'1' || s[1]>'D' || s[1]<'A'){
        cout<<"invalid seat no\n"; //invalid sit no
        return 0;
    }
}

```

```

int row=-1,col=-1;
//find the row no of the user sit
for(int i=0;i<7;i++){
    if(arr[i][0]==s[0])
        row=i;
}

//find the column no of user sit
for(int j=0;j<5;j++){

```

```

if(arr[row][j]==s[1])
col=j;
}

//check whether sit is already occupied or not.
if(col==-1){
cout<<"Seat is already occupied\n";
return 0;
}
else{
//if it's a valid sit & not occupied,
//process the requested & update the sit as occupied
update(arr,row,col);
}
return 1;
}

void airline(char arr[7][5]){
    // user can stop process by pressing 'N'
    cout<<"enter N if you are done!\n";
    string s;
    // continue if not interrepted by user or
    //there is valid sit in unoccupied state
    while(true){
        s=getData(); //get user input
        //if user input is to stop the process
        if(s[0]=='N')
            break; // break

        //process the request & check according to
        if(check(arr,s))

```

```

        display(arr);

        if(allOccupied(arr)){ // if all sits are occupied
            cout<<"sorry, no more seats left!!!!!!!!!!1..."<<endl;
            break; //break
        }
    }
    cout<<"Thanks, that's all"<<endl; //end of program
}

int main()
{
    //2-D array for storing sit number
    char arr[7][5];
    for(int i=0;i<7;i++){
        //forst column is row number
        arr[i][0]=i+1+'0';
        for(int j=1;j<5;j++){
            //to represent sit number A,B,C,D respectively
            arr[i][j]='A'+j-1;
        }
    }

    cout<<"initial seat arrangements.....\n";
    display(arr);

    airline(arr); //airline function

    return 0;
}

```

Virtual Functions

A virtual function is a member function that is declared within a base class and redefined by a derived class. To create a virtual function, precede the function's declaration in the base class with the keyword **virtual**. When a class containing a virtual function is inherited, the derived class redefines the virtual function to fit its own needs. In essence, virtual functions implement the "one interface, multiple methods" philosophy that underlies polymorphism. The virtual function within the base class defines the form of the interface to that function. Each redefinition of the virtual function by a derived class implements its operation as it relates specifically to the derived class. That is, the redefinition creates a specific method.

Virtual Function is a function in base class, which is overridden in the derived class, and which tells the compiler to perform **Late Binding** on this function.

Virtual Keyword is used to make a member function of the base class Virtual.

Late Binding

In Late Binding function call is resolved at runtime. Hence, now compiler determines the type of object at runtime, and then binds the function call. Late Binding is also called **Dynamic Binding** or **Runtime Binding**.

Problem without Virtual Keyword

```
class Base
{
public:
void show() // belongs to base class//
{
    cout << "Base class";
}
};

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

int main()
{
    Base* b;           //Base class pointer
```

```

Derived d;    //Derived class object
b = &d;
b->show();    //Early Binding Occurs
}

```

Output : Base class

When we use Base class's pointer to hold Derived class's object, base class pointer or reference will always call the base version of the function

Using Virtual Keyword

We can make base class's methods virtual by using **virtual** keyword while declaring them. Virtual keyword will lead to Late Binding of that method.

```

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

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

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

```

Output : Derived class

On using Virtual keyword with Base class's function, Late Binding takes place and the derived version of function will be called, because base class pointer points to Derived class object.

Using Virtual Keyword and Accessing Private Method of Derived class

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

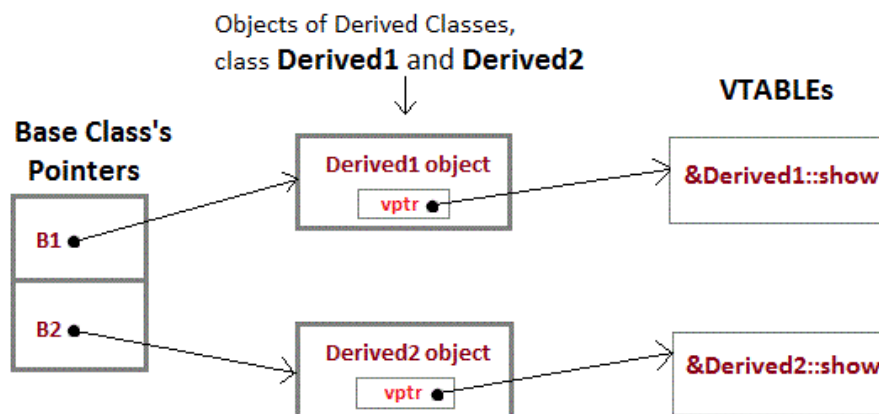
```
#include
using namespace std;
class A
{
    public:
    virtual void show()
    {
        cout << "Base class\n";
    }
};

class B: public A
{
    private:
    virtual void show()
    {
        cout << "Derived class\n";
    }
};

int main()
{
    A *a;
    B b;
    a = &b;
    a -> show();
}
```

Output : Derived class

Mechanism of Late Binding



vptr, is the vpointer, which points to the Virtual Function for that object.

VTABLE, is the table containing address of Virtual Functions of each class.

To accomplish late binding, Compiler creates **VTABLEs**, for each class with virtual function. The address of virtual functions is inserted into these tables. Whenever an object of such class is created the compiler secretly inserts a pointer called **vpointer**, pointing to VTABLE for that object. Hence when function is called, compiler is able to resolve the call by binding the correct function using the vpointer.

Important Points to Remember

1. Only the Base class Method's declaration needs the **Virtual** Keyword, not the definition.
2. If a function is declared as **virtual** in the base class, it will be virtual in all its derived classes.
3. The address of the virtual Function is placed in the **VTABLE** and the compiler uses **VPTR**(vpointer) to point to the Virtual Function.

Overriding

- 1)Methods name and signatures must be same.
- 2)Overriding is the concept of runtime polymorphism
- 3)When a function of base class is re-defined in the derived class called as Overriding
- 4)It needs inheritance.
- 5)Method should have same data type.

6)Method should be public.

Overloading

1)Having same method name with different Signatures.

2)Overloading is the concept of compile time polymorphism

3)Two functions having same name and return type, but with different type and/or number of arguments is called as Overloading

4)It doesn't need inheritance.

5)Method can have different data types

6)Method can be different access specifies

Example programs:

```
#include <iostream>

using namespace std;

class base {

public:

virtual void vfunc() {

cout << "This is base's vfunc().\n";

}

};

class derived1 : public base {

public:

void vfunc() {

cout << "This is derived1's vfunc().\n";

}

};

class derived2 : public base {

public:

void vfunc() {

cout << "This is derived2's vfunc().\n";
```

```

}

};

int main()
{
    base *p, b;

    derived1 d1;

    derived2 d2;

    // point to base

    p = &b;

    p->vfunc(); // access base's vfunc()

    // point to derived1

    p = &d1;

    p->vfunc(); // access derived1's vfunc()

    // point to derived2

    p = &d2;

    p->vfunc(); // access derived2's vfunc()

    return 0;
}

```

This program displays the following:

```

This is base's vfunc().
This is derived1's vfunc().
This is derived2's vfunc().

```

As the program illustrates, inside base, the virtual function `vfunc()` is declared. Notice that the keyword `virtual` precedes the rest of the function declaration. When `vfunc()` is redefined by `derived1` and `derived2`, the keyword `virtual` is not needed. (However, it is not an error to include it when redefining a virtual function inside a derived class; it's just not needed.)

In this program, **base** is inherited by both **derived1** and **derived2**. Inside each class definition, **vfunc()** is redefined relative to that class. Inside **main()**, four variables are declared:

Name	Type
p	base class pointer
b	object of base
d1	object of derived1
d2	object of derived2

Next, **p** is assigned the address of **b**, and **vfunc()** is called via **p**. Since **p** is pointing to an object of type **base**, that version of **vfunc()** is executed. Next, **p** is set to the address of **d1**, and again **vfunc()** is called by using **p**. This time **p** points to an object of type **derived1**. This causes **derived1::vfunc()** to be executed. Finally, **p** is assigned the address of **d2**, and **p->vfunc()** causes the version of **vfunc()** redefined inside **derived2** to be executed. The key point here is that the kind of object to which **p** points determines which version of **vfunc()** is executed. Further, this determination is made at run time, and this process forms the basis for run-time polymorphism.

Calling a Virtual Function Through a Base Class Reference

```
#include <iostream>

using namespace std;

class base {

public:

virtual void vfunc() {

cout << "This is base's vfunc().\n";

}

};

class derived1 : public base {

public:

void vfunc() {

cout << "This is derived1's vfunc().\n";

}

};

class derived2 : public base {

public:

void vfunc() {
```

```

    cout << "This is derived2's vfunc().\n";

}

};

int main()

{

    base *p, b;

    derived1 d1;

    derived2 d2;

    // point to base

    p = &b;

    p->vfunc(); // access base's vfunc()

    // point to derived1

    p = &d1;

    p->vfunc(); // access derived1's vfunc()

    // point to derived2

    p = &d2;

    p->vfunc(); // access derived2's vfunc()

    return 0;

}

```

The Virtual Attribute Is Inherited

When a virtual function is inherited, its virtual nature is also inherited. This means that when a derived class that has inherited a virtual function is itself used as a base class for another derived class, the virtual function can still be overridden.

```

#include <iostream>

using namespace std;

class base {

public:

    virtual void vfunc() {

```

```

cout << "This is base's vfunc().\n";

}

};

class derived1 : public base {

public:

void vfunc() {

cout << "This is derived1's vfunc().\n";

}

};

/* derived2 inherits virtual function vfunc()
from derived1. */

class derived2 : public derived1 {

public:

// vfunc() is still virtual

void vfunc() {

cout << "This is derived2's vfunc().\n";

}

};

int main()

{

base *p, b;

derived1 d1;

derived2 d2;

// point to base

p = &b;

p->vfunc(); // access base's vfunc()

// point to derived1

```

```
p = &d1;

p->vfunc(); // access derived1's vfunc()

// point to derived2

p = &d2;

p->vfunc(); // access derived2's vfunc()

return 0;

}
```

Virtual Functions Are Hierarchical

when a function is declared as virtual by a base class, it may be overridden by a derived class. However, the function does not have to be overridden. When a derived class fails to override a virtual function, then when an object of that derived class accesses that function, the function defined by the base class is used. For example, consider this program in which derived2 does not override vfunc():

Early binding refers to events that occur at compile time. In essence, early binding occurs when all information needed to call a function is known at compile time. (Put differently, early binding means that an object and a function call are bound during compilation.) Examples of early binding include normal function calls (including standard library functions), overloaded function calls, and overloaded operators. The main advantage to early binding is efficiency. Because all information necessary to call a function is determined at compile time, these types of function calls are very fast. The opposite of early binding is late binding.

As it relates to C++, **late binding refers** to function calls that are not resolved until run time. **Virtual functions are used to achieve late binding.** As you know, when access is via a base pointer or reference, the virtual function actually called is determined by the type of object pointed to by the pointer. Because in most cases this cannot be determined at compile time, the object and the function are not linked until run time. The main advantage to late binding is flexibility. Unlike early binding, late binding allows you to create programs that can respond to events occurring while the program executes without having to create a large amount of "contingency code." Keep in mind that because a function call is not resolved until run time, late binding can make for somewhat slower execution times

A *pure virtual function* is a virtual function that has no definition within the base class. To declare a pure virtual function, use this general form:

```
virtual type func-name(parameter-list) = 0;
```

When a virtual function is made pure, any derived class must provide its own definition. If the derived class fails to override the pure virtual function, a compile-time error will result.

The following program contains a simple example of a pure virtual function. The base class, **number**, contains an integer called **val**, the function **setval()**, and the pure virtual function **show()**. The derived classes **hextype**, **dectype**, and **octtype** inherit **number** and redefine **show()** so that it outputs the value of **val** in each respective number base (that is, hexadecimal, decimal, or octal).

```
#include <iostream>
using namespace std;
class number {
protected:
int val;
public:
void setval(int i) { val = i; }
// show() is a pure virtual function
virtual void show() = 0;
};
class hextype : public number {
public:
void show() {
cout << hex << val << "\n";
}
};
class dectype : public number {
public:
void show() {
cout << val << "\n";
}
};
class octtype : public number {
public:
void show() {
cout << oct << val << "\n";
}
};
int main()
{
dectype d;
hextype h;
octtype o;
d.setval(20);
d.show(); // displays 20 - decimal
h.setval(20);
h.show(); // displays - hexadecimal
```



```
o.setval(20);  
o.show(); // displays - octal  
return 0;  
}
```

Generic Functions

A generic function defines a general set of operations that will be applied to various types of data.

The type of data that the function will operate upon is passed to it as a parameter

Through a generic function, **a single general procedure can be applied to a wide range of data.**

A generic function is created using the keyword `template`. The normal meaning of the word "template" accurately reflects its use in C++. It is used to create a template (or framework) that describes what a function will do, leaving it to the compiler to fill in the details as needed. The general form of a template function definition is shown here:

```
template <class Ttype> ret-type func-name(parameter list)
{
    // body of function
}
```

Here, *Ttype* is a placeholder name for a data type used by the function. This name may be used within the function definition. However, it is only a placeholder that the compiler will automatically replace with an actual data type when it creates a specific version of the function. Although the use of the keyword **class** to specify a generic type in a **template** declaration is traditional, you may also use the keyword **typename**.

// Function template example.

```
#include <iostream>
```

```
using namespace std;
```

```
// This is a function template.
```

```
template <class X> void swapargs(X &a, X &b)
```

```
{
```

```
    X temp;
```

```
    temp = a;
```

```
    a = b;
```

```
    b = temp;
```

```
}
```

```
int main()
```

```
{
```

```

int i=10, j=20;
double x=10.1, y=23.3;
char a='x', b='z';
cout << "Original i, j: " << i << ' ' << j << '\n';
cout << "Original x, y: " << x << ' ' << y << '\n';
cout << "Original a, b: " << a << ' ' << b << '\n';
swapargs(i, j); // swap integers
swapargs(x, y); // swap floats
swapargs(a, b); // swap chars
cout << "Swapped i, j: " << i << ' ' << j << '\n';
cout << "Swapped x, y: " << x << ' ' << y << '\n';
cout << "Swapped a, b: " << a << ' ' << b << '\n';
return 0;
}

```

tells the compiler two things: that a template is being created and that a generic definition is beginning. Here, X is a generic type that is used as a placeholder. After the template portion, the function `swapargs()` is declared, using X as the data type of the values that will be swapped.

In `main()`, the `swapargs()` function is called using three different types of data: ints, doubles, and chars. Because `swapargs()` is a generic function, the compiler automatically creates three versions of `swapargs()`: one that will exchange integer values, one that will exchange floating-point values, and one that will swap characters.

Wrong declaration:

```

// This will not compile.
template <class X>
int i; // this is an error
void swapargs(X &a, X &b)
{
    X temp;

    temp = a;
    a = b;
    b = temp;
}

```

A Function with Two Generic Types

```
//A Function with Two Generic Types//
#include <iostream>
using namespace std;
template <class type1, class type2>
void myfunc(type1 x, type2 y)
{
    cout << x << ' ' << y << '\n';
}
int main()
{
    myfunc(10, "I like C++");
    myfunc(98.6, 19L);
    return 0;
}

//Explicitly Overloading a Generic Function//
#include <iostream>
using namespace std;
template <class X> void swapargs(X &a, X &b)
{
    X temp;
    temp = a;
    a = b;
    b = temp;
    cout << "Inside template swapargs.\n";
}

// This overrides the generic version of swapargs() for ints.
void swapargs(int &a, int &b)
{
    int temp;
```

```

temp = a;
a = b;
b = temp;
cout << "Inside swapargs int specialization.\n";
}
int main()
{
int i=10, j=20;
double x=10.1, y=23.3;
char a='x', b='z';
cout << "Original i, j: " << i << ' ' << j << '\n';
cout << "Original x, y: " << x << ' ' << y << '\n';
cout << "Original a, b: " << a << ' ' << b << '\n';
swapargs(i, j); // calls explicitly overloaded swapargs()
swapargs(x, y); // calls generic swapargs()
swapargs(a, b); // calls generic swapargs()
cout << "Swapped i, j: " << i << ' ' << j << '\n';
cout << "Swapped x, y: " << x << ' ' << y << '\n';
cout << "Swapped a, b: " << a << ' ' << b << '\n';
return 0;
}

```

A Generic Sort

Sorting is exactly the type of operation for which generic functions were designed. Within wide latitude, a sorting algorithm is the same no matter what type of data is being sorted. The following program illustrates this by creating a generic bubble sort. While the bubble sort is a rather poor sorting algorithm, its operation is clear and uncluttered and it makes an easy-to-understand example. The bubble() function will sort any type of array. It is called with a pointer to the first element in the array and the number of elements in the array.

```
// A Generic bubble sort.
```

```
#include <iostream>
```

```

using namespace std;

template <class X> void bubble(
X *items, // pointer to array to be sorted
int count) // number of items in array
{
    register int a, b;
    X t;
    for(a=1; a<count; a++)
    for(b=count-1; b>=a; b--)
    if(items[b-1] > items[b]) {
        // exchange elements
        t = items[b-1];
        items[b-1] = items[b];
        items[b] = t;
    }
}

int main()
{
    int iarray[7] = {7, 5, 4, 3, 9, 8, 6};
    double darray[5] = {4.3, 2.5, -0.9, 100.2, 3.0};
    int i;
    cout << "Here is unsorted integer array: ";
    for(i=0; i<7; i++)cout << iarray[i] << ' ';
    cout << endl;
    cout << "Here is unsorted double array: ";
    for(i=0; i<5; i++)
    cout << darray[i] << ' ';
    cout << endl;
    bubble(iarray, 7);
    bubble(darray, 5);
}

```

```

cout << "Here is sorted integer array: ";
for(i=0; i<7; i++)
cout << iarray[i] << ' ';
cout << endl;
cout << "Here is sorted double array: ";
for(i=0; i<5; i++)
cout << darray[i] << ' ';
cout << endl;
return 0;
}

```

Generic Classes

In addition to generic functions, you can also define a generic class. When you do this, you create a class that defines all the algorithms used by that class; however, the actual type of the data being manipulated will be specified as a parameter when objects of that class are created.

Generic classes are useful when a class uses logic that can be generalized. For example, the same algorithms that maintain a queue of integers will also work for a queue of characters, and the same mechanism that maintains a linked list of mailing addresses will also maintain a linked list of auto part information. When you create a generic class, it can perform the operation you define, such as maintaining a queue or a linked list, for any type of data. The compiler will automatically generate the correct type of object, based upon the type you specify when the object is created.

The general form of a generic class declaration is shown here:

```

template <class Ttype> class class-name {
    .
    .
    .
}

```

// This function demonstrates a generic stack.

```
#include <iostream>
```

```
using namespace std;
```

```
const int SIZE = 10;
```

```
// Create a generic stack class
```

```

template <class StackType> class stack {
StackType stck[SIZE]; // holds the stack
int tos; // index of top-of-stack
public:
stack() { tos = 0; } // initialize stack
void push(StackType ob); // push object on stack
StackType pop(); // pop object from stack
};

// Push an object.
template <class StackType> void stack<StackType>::push(StackType ob)
{
if(tos==SIZE) {
cout << "Stack is full.\n";
return;
}
stck[tos] = ob;
tos++;
}

// Pop an object.
template <class StackType> StackType stack<StackType>::pop()
{
if(tos==0) {
cout << "Stack is empty.\n";
return 0; // return null on empty stack
}
tos--;
return stck[tos];
}

int main()
{

```



```

// Demonstrate character stacks.
stack<char> s1, s2; // create two character stacks
int i;
s1.push('a');
s2.push('x');
s1.push('b');
s2.push('y');
s1.push('c');
s2.push('z');
for(i=0; i<3; i++) cout << "Pop s1: " << s1.pop() << "\n";
for(i=0; i<3; i++) cout << "Pop s2: " << s2.pop() << "\n";
// demonstrate double stacks
stack<double> ds1, ds2; // create two double stacks
ds1.push(1.1);
ds2.push(2.2);
ds1.push(3.3);
ds2.push(4.4);
ds1.push(5.5);
ds2.push(6.6);
for(i=0; i<3; i++) cout << "Pop ds1: " << ds1.pop() << "\n";
for(i=0; i<3; i++) cout << "Pop ds2: " << ds2.pop() << "\n";
return 0;
}

```

```

//Two Generic Data Types of Calss//
/* This example uses two generic data types in a
class definition.
*/
#include <iostream>
using namespace std;
template <class Type1, class Type2> class myclass

```

```

{
Type1 i;
Type2 j;
public:
myclass(Type1 a, Type2 b) { i = a; j = b; }
void show() { cout << i << ' ' << j << '\n'; }
};

int main()
{
myclass<int, double> ob1(10, 0.23);
myclass<char, char *> ob2('X', "Templates add power.");
ob1.show(); // show int, double
ob2.show(); // show char, char *
return 0;
}

```

Applying Template Classes:

A Generic Array Class To illustrate the practical benefits of template classes, let's look at one way in which they are commonly applied. As you saw in Chapter 15, you can overload the [] operator. Doing so allows you to create your own array implementations, including "safe arrays" that provide run-time boundary checking. As you know, in C++, it is possible to overrun (or underrun) an array boundary at run time without generating a run-time error message. However, if you create a class that contains the array, and allow access to that array only through the overloaded [] subscripting operator, then you can intercept an out-of-range index.

// A generic safe array example.

```

#include <iostream>
#include <cstdlib>
using namespace std;
const int SIZE = 10;
template <class AType> class atype {
AType a[SIZE];
public:

```

```

atype() {
register int i;
for(i=0; i<SIZE; i++) a[i] = i;
}
AType &operator[](int i);
};

// Provide range checking for atype.
template <class AType> AType &atype<AType>::operator[](int i)
{
if(i<0 || i> SIZE-1) {
cout << "\nIndex value of ";
cout << i << " is out-of-bounds.\n";
exit(1);
}
return a[i];
}

int main()
{
atype<int> intob; // integer array
atype<double> doubleob; // double array
int i;
cout << "Integer array: ";
for(i=0; i<SIZE; i++) intob[i] = i;
for(i=0; i<SIZE; i++) cout << intob[i] << " ";
cout << '\n';
cout << "Double array: ";
for(i=0; i<SIZE; i++) doubleob[i] = (double) i/3;
for(i=0; i<SIZE; i++) cout << doubleob[i] << " ";
cout << '\n';
intob[12] = 100; // generates runtime error

```

```
return 0;
```

```
}
```

Using Non-Type Arguments with Generic Classes

In the template specification for a generic class, you may also specify non-type arguments. That is, in a template specification you can specify what you would normally think of as a standard argument, such as an integer or a pointer. The syntax to accomplish this is essentially the same as for normal function parameters: simply include the type and name of the argument. For example, here is a better way to implement the safe-array class presented in the preceding section. It allows you to specify the size of the array.

```
// A generic safe array example.
```

```
#include <iostream>
```

```
#include <cstdlib>
```

```
using namespace std;
```

```
const int SIZE = 10;
```

```
template <class AType> class atype {
```

```
    AType a[SIZE];
```

```
public:
```

```
    atype() {
```

```
        register int i;
```

```
        for(i=0; i<SIZE; i++) a[i] = i;
```

```
    }
```

```
    AType &operator[](int i);
```

```
};
```

```
// Provide range checking for atype.
```

```
template <class AType> AType &atype<AType>::operator[](int i)
```

```
{
```

```
    if(i<0 || i> SIZE-1) {
```

```
        cout << "\nIndex value of ";
```

```
        cout << i << " is out-of-bounds.\n";
```

```
        exit(1);
```

```
}  
return a[i];  
}  
int main()  
{  
    atype<int> intob; // integer array  
    atype<double> doubleob; // double array  
    int i;  
    cout << "Integer array: ";  
    for(i=0; i<SIZE; i++) intob[i] = i;  
    for(i=0; i<SIZE; i++) cout << intob[i] << " ";  
    cout << '\n';  
    cout << "Double array: ";  
    for(i=0; i<SIZE; i++) doubleob[i] = (double) i/3;  
    for(i=0; i<SIZE; i++) cout << doubleob[i] << " ";  
    cout << '\n';  
    intob[12] = 100; // generates runtime error  
    return 0;  
}
```

