C++ Overloading (Operator and Function)

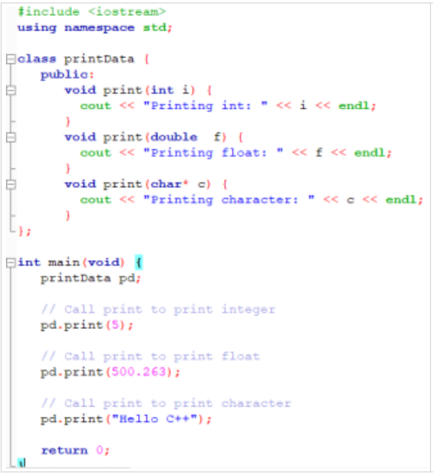
Saturday, May 1, 2021 8:48 AM

What is Overloading in C++ ?

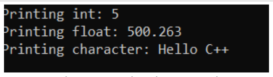
- C++ allows to specify more than 1 definition for a function name or an operator in one scope. This is called function overloading and operator overloading
- An overloaded declaration is a declaration that declared with the same name as a previously declaration in the same scope (excep both declarations have different arguments and different definition)

int find_sum(int x, int y)	int find_sum(int x)

- when calling an overloading function or operator, the compiler determines the most appropriate definition to use, by comparing the argument we use to call out that function or operator with same parameters
- this process called overload resolution



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

class printData {
    public:
        void print(int i) {
          cout << "Printing int: " << i << endl;
        }
        void print(double  f) {
          cout << "Printing float: " << f << endl;
        }
        void print(char* c) {
          cout << "Printing character: " << c << endl;
        }
};

int main(void) {
    printData pd;

    // Call print to print integer
    pd.print(5);

    // Call print to print float
    pd.print(500.263);

    // Call print to print character
    pd.print("Hello C++");

    return 0;
}
```



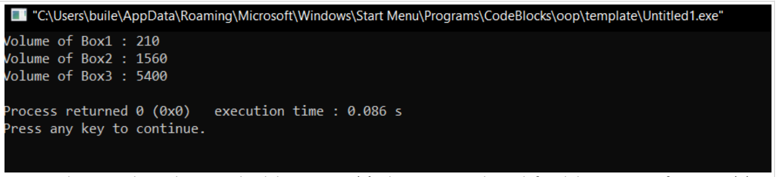
```
Printing int: 5
Printing float: 500.263
Printing character: Hello C++
```

- in this example, the compiler receives a lot of calls to function print but with different arguments
- for example, print(int) will call to the print() function which accepts integer parameter

Operators overloading

- We can redefind or overload most of the operators available in C++
- Overloaded operators are function with special name **operator**
- `Box operator+(const Box&);`

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

class Box {
   public:
      double getVolume(void) {
         return length * breadth * height;
      }
      void setLength( double len ) {
         length = len;
      }
      void setBreadth( double bre ) {
         breadth = bre;
      }
      void setHeight( double hei ) {
         height = hei;
      }

      // Overload + operator to add two Box objects.
      Box operator+(const Box& b) {
         Box box;
         box.length = this->length + b.length;
         box.breadth = this->breadth + b.breadth;
         box.height = this->height + b.height;
         return box;
      }

   private:
      double length;      // Length of a box
      double breadth;     // Breadth of a box
      double height;      // Height of a box
};

// Main function for the program
int main() {
   Box Box1;                // Declare Box1 of type Box
   Box Box2;                // Declare Box2 of type Box
   Box Box3;                // Declare Box3 of type Box
   double volume = 0.0;     // Store the volume of a box
here

   // box 1 specification
   Box1.setLength(6.0);
   Box1.setBreadth(7.0);
   Box1.setHeight(5.0);

   // box 2 specification
   Box2.setLength(12.0);
   Box2.setBreadth(13.0);
   Box2.setHeight(10.0);

   // volume of box 1
   volume = Box1.getVolume();
   cout << "Volume of Box1 : " << volume <<endl;

   // volume of box 2
```



```
"C:\Users\buile\AppData\Roaming\Microsoft\Windows\Start Menu\Programs\CodeBlocks\oop\template\Untitled1.exe"
Volume of Box1 : 210
Volume of Box2 : 1560
Volume of Box3 : 5400

Process returned 0 (0x0)   execution time : 0.086 s
Press any key to continue.
```

- IN this example, we have overload the operator '**+**' , that means we have defined the meaning of operator '+' between two object from class Box
- Box 1 + Box 2 = each element of box 1 plus each corresponding element from box 2 and return the box 3 reference

```
    volume = Box2.getVolume();
    cout << "Volume of Box2 : " << volume <<endl;

    // Add two object as follows:
    Box3 = Box1 + Box2;

    // volume of box 3
    volume = Box3.getVolume();
    cout << "Volume of Box3 : " << volume <<endl;

    return 0;
}
```

Here are somes overloadable and unoverloadable operators:

Overloadable/Non-overloadableOperators

Following is the list of operators which can be overloaded −

+	-	*	/	%	^
&	|	~	!	,	=
<	>	<=	>=	++	--
<<	>>	==	!=	&&	||
+=	-=	/=	%=	^=	&=
|=	*=	<<=	>>=	[]	()
->	->*	new	new []	delete	delete []

Following is the list of operators, which can not be overloaded −

::	.*	.	?:

Operator Overloading Examples

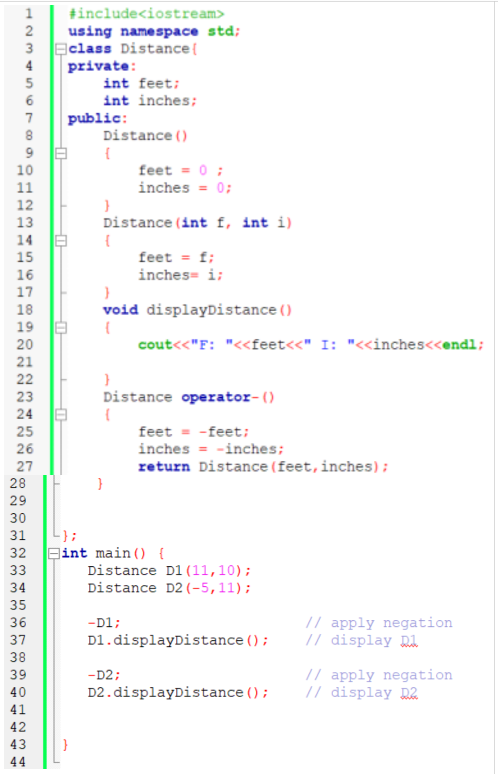
Unary Operators Overloading in C++

The unary operators operate on a single operand and following are the examples of Unary operators −

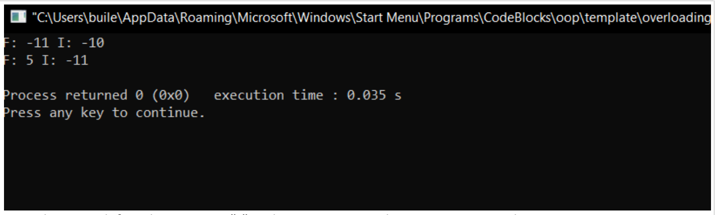
- The increment (++) and decrement (--) operators.
- The unary minus (-) operator.
- The logical not (!) operator.

How to use this operators overloading

- The unary operators operate on the object for which they were called and normally
- this operator appears on the left side of the object, as in !obj, -obj, and ++obj but sometime they can be used as postfix as well like obj++ or obj--.



```
#include<iostream>
using namespace std;
class Distance{
private:
    int feet;
    int inches;
public:
    Distance()
    {
        feet = 0 ;
        inches = 0;
    }
    Distance(int f, int i)
    {
        feet = f;
        inches= i;
    }
    void displayDistance()
    {
        cout<<"F: "<<feet<<" I: "<<inches<<endl;

    }
    Distance operator-()
    {
        feet = -feet;
        inches = -inches;
        return Distance(feet,inches);
   }


};
int main() {
   Distance D1(11,10);
   Distance D2(-5,11);

   -D1;                    // apply negation
   D1.displayDistance();   // display D1

   -D2;                    // apply negation
   D2.displayDistance();   // display D2


}
```

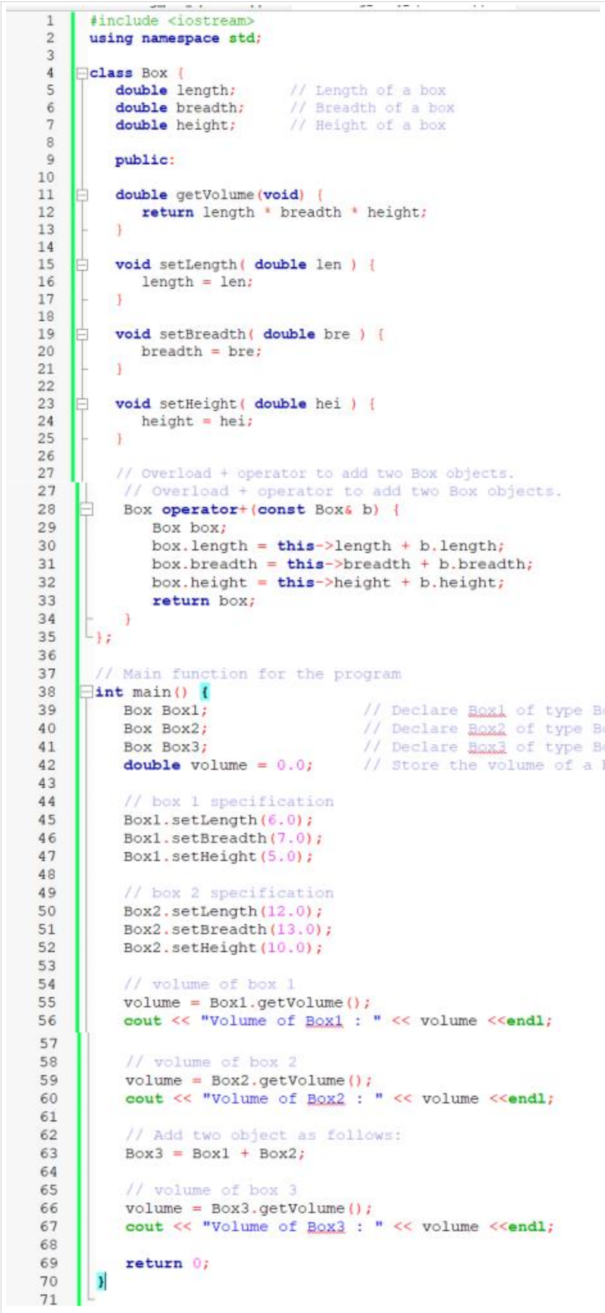


```
"C:\Users\buile\AppData\Roaming\Microsoft\Windows\Start Menu\Programs\CodeBlocks\oop\template\overloading
F: -11 I: -10
F: 5 I: -11

Process returned 0 (0x0)   execution time : 0.035 s
Press any key to continue.
```

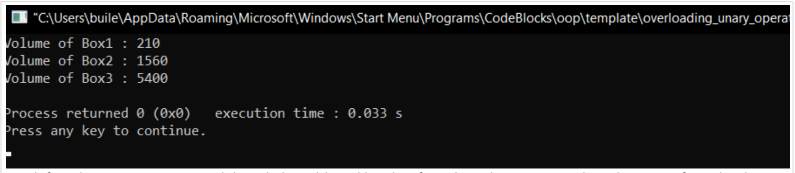
- here we define the operator "-" and put it previous object to negate its elements.

Binary Operators Overloading in C++

The binary operators take two arguments and following are the examples of Binary operators. You use binary operators very fre quently like addition (+) operator, subtraction (-) operator and division (/) operator.



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

class Box {
    double length;      // Length of a box
    double breadth;     // Breadth of a box
    double height;      // Height of a box

    public:

    double getVolume(void) {
        return length * breadth * height;
    }

    void setLength( double len ) {
        length = len;
    }

    void setBreadth( double bre ) {
        breadth = bre;
    }

    void setHeight( double hei ) {
        height = hei;
    }

    // Overload + operator to add two Box objects.
     // Overload + operator to add two Box objects.
     Box operator+(const Box& b) {
        Box box;
        box.length = this->length + b.length;
        box.breadth = this->breadth + b.breadth;
        box.height = this->height + b.height;
        return box;
     }
};

// Main function for the program
int main() {
    Box Box1;                // Declare Box1 of type Box
    Box Box2;                // Declare Box2 of type Box
    Box Box3;                // Declare Box3 of type Box
    double volume = 0.0;     // Store the volume of a box here

    // box 1 specification
    Box1.setLength(6.0);
    Box1.setBreadth(7.0);
    Box1.setHeight(5.0);

    // box 2 specification
    Box2.setLength(12.0);
    Box2.setBreadth(13.0);
    Box2.setHeight(10.0);

    // volume of box 1
    volume = Box1.getVolume();
    cout << "Volume of Box1 : " << volume <<endl;

    // volume of box 2
    volume = Box2.getVolume();
    cout << "Volume of Box2 : " << volume <<endl;

    // Add two object as follows:
    Box3 = Box1 + Box2;

    // volume of box 3
    volume = Box3.getVolume();
    cout << "Volume of Box3 : " << volume <<endl;

    return 0;
}
```



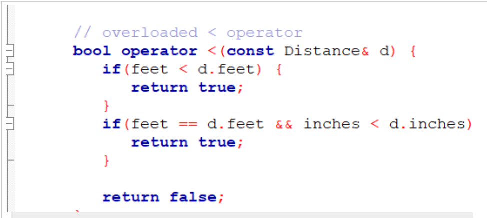
```
"C:\Users\buile\AppData\Roaming\Microsoft\Windows\Start Menu\Programs\CodeBlocks\oop\template\overloading_unary_operat
Volume of Box1 : 210
Volume of Box2 : 1560
Volume of Box3 : 5400

Process returned 0 (0x0)   execution time : 0.033 s
Press any key to continue.
```

we define the operators + as each length, breadth and height of one box plus corresponding elements of another box.

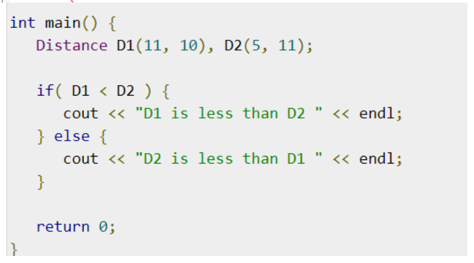
Relational Operators Overloading in C++

(<, >, <=, >=, ==, etc.)

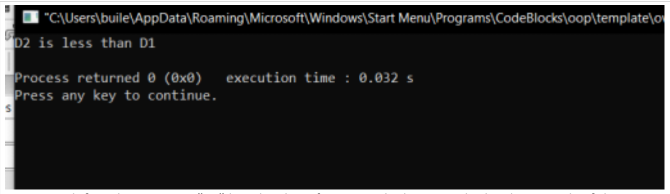
can overload any of these operators, which can be used to compare the objects of a class.



```cpp
      // overloaded < operator
      bool operator <(const Distance& d) {
         if(feet < d.feet) {
            return true;
         }
         if(feet == d.feet && inches < d.inches) {
            return true;
         }

         return false;
```



```cpp
int main() {
   Distance D1(11, 10), D2(5, 11);

   if( D1 < D2 ) {
      cout << "D1 is less than D2 " << endl;
   } else {
      cout << "D2 is less than D1 " << endl;
   }

   return 0;
}
```

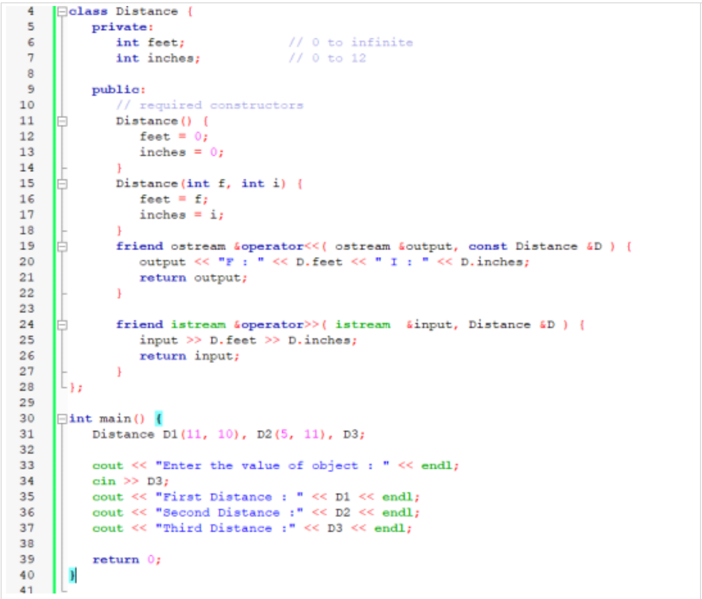


```
"C:\Users\buile\AppData\Roaming\Microsoft\Windows\Start Menu\Programs\CodeBlocks\oop\template\o
D2 is less than D1

Process returned 0 (0x0)   execution time : 0.032 s
Press any key to continue.
```

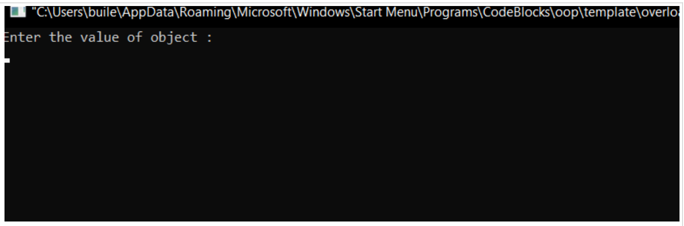
- we define the operator " < " by a boolean function which return the boolean result of the comparison between element "feet" from 2 objects

Input/Output Operators Overloading in C++

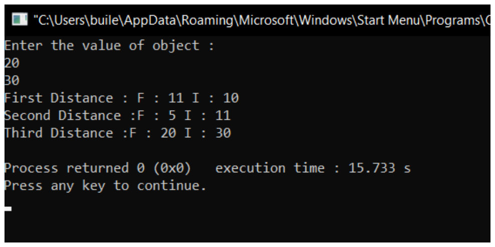
- C++ is able to input and output the built-in data types using the stream extraction operator >> and the stream insertion operator <<.
- The stream insertion and stream extraction operators also can be overloaded to perform input and output for user-defined types like an object.



```cpp
class Distance {
   private:
      int feet;             // 0 to infinite
      int inches;           // 0 to 12

   public:
      // required constructors
      Distance() {
         feet = 0;
         inches = 0;
      }
      Distance(int f, int i) {
         feet = f;
         inches = i;
      }
      friend ostream &operator<<( ostream &output, const Distance &D ) {
         output << "F : " << D.feet << " I : " << D.inches;
         return output;
      }

      friend istream &operator>>( istream  &input, Distance &D ) {
         input >> D.feet >> D.inches;
         return input;
      }
};

int main() {
   Distance D1(11, 10), D2(5, 11), D3;

   cout << "Enter the value of object : " << endl;
   cin >> D3;
   cout << "First Distance : " << D1 << endl;
   cout << "Second Distance :" << D2 << endl;
   cout << "Third Distance :" << D3 << endl;

   return 0;
}
```



```
"C:\Users\buile\AppData\Roaming\Microsoft\Windows\Start Menu\Programs\CodeBlocks\oop\template\overlo
Enter the value of object :
```

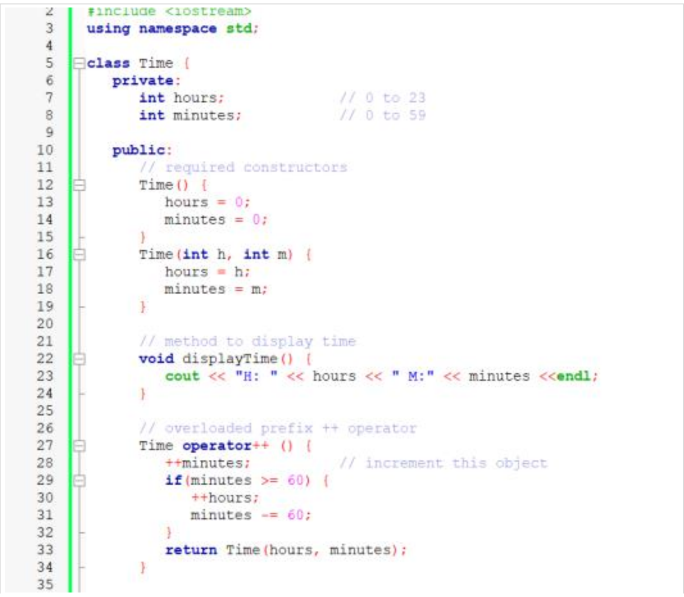


```
"C:\Users\buile\AppData\Roaming\Microsoft\Windows\Start Menu\Programs\C
Enter the value of object :
20
30
First Distance : F : 11 I : 10
Second Distance :F : 5 I : 11
Third Distance :F : 20 I : 30

Process returned 0 (0x0)   execution time : 15.733 s
Press any key to continue.
```

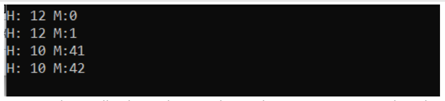
- This program allows us to input an object by the keyboard
-
```cpp
friend ostream &operator<<( ostream &output, const Distance &D ) {
   output << "F : " << D.feet << " I : " << D.inches;
   return output;
}
```
- The above lines have defined the operator "<<" return a result type called "ostream", by return the reference to the output as "cout<<" commands. IN this operators, feet and inches in the object would be outputted to the screen by a likely method as a "cout<<" command.
-
```cpp
friend istream &operator>>( istream  &input, Distance &D ) {
   input >> D.feet >> D.inches;
   return input;
}
```
- These above lines show the rule to input data amd create a new object in the class Distance, interesting :D.

Overloading Increment ++ & Decrement --

The increment (++) and decrement (--) operators are two important unary operators available in C++.

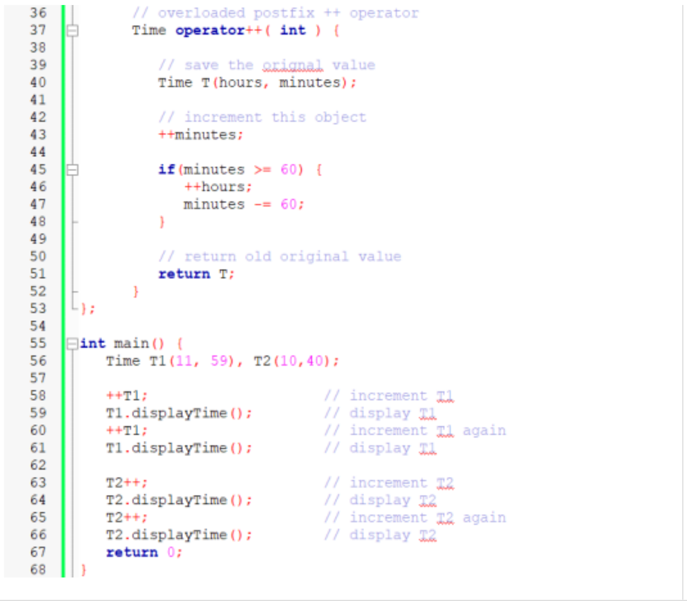


```cpp
#include <iostream>
using namespace std;

class Time {
   private:
      int hours;             // 0 to 23
      int minutes;           // 0 to 59

   public:
      // required constructors
      Time() {
         hours = 0;
         minutes = 0;
      }
      Time(int h, int m) {
         hours = h;
         minutes = m;
      }

      // method to display time
      void displayTime() {
         cout << "H: " << hours << " M:" << minutes <<endl;
      }

      // overloaded prefix ++ operator
      Time operator++ () {
         ++minutes;          // increment this object
         if(minutes >= 60) {
            ++hours;
            minutes -= 60;
         }
         return Time(hours, minutes);
      }

```

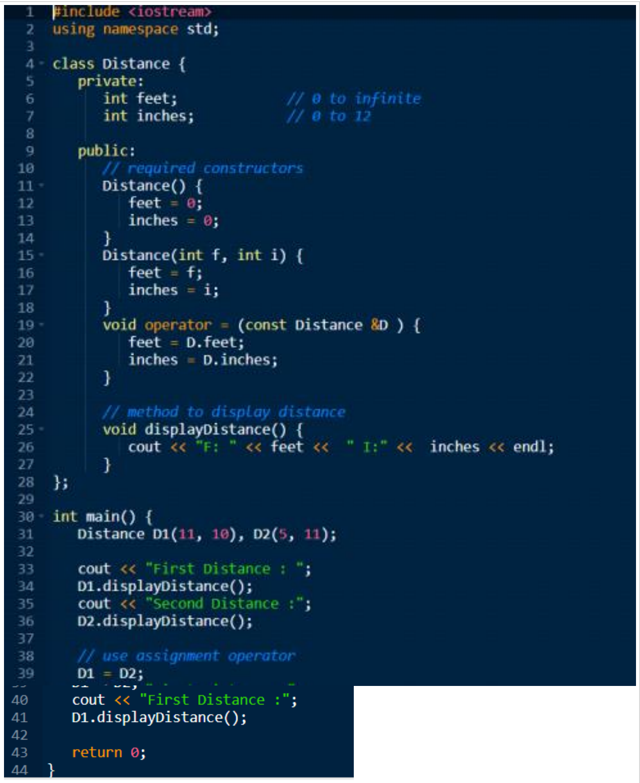


```
H: 12 M:0
H: 12 M:1
H: 10 M:41
H: 10 M:42
```

Not much to talk a bout this, similar to the unary operators, but this overload the "++" operator


```
    // overloaded postfix ++ operator
    Time operator++( int ) {

       // save the orignal value
       Time T(hours, minutes);

       // increment this object
       ++minutes;

       if(minutes >= 60) {
          ++hours;
          minutes -= 60;
       }

       // return old original value
       return T;
    }
};

int main() {
   Time T1(11, 59), T2(10,40);

   ++T1;                    // increment T1
   T1.displayTime();        // display T1
   ++T1;                    // increment T1 again
   T1.displayTime();        // display T1

   T2++;                    // increment T2
   T2.displayTime();        // display T2
   T2++;                    // increment T2 again
   T2.displayTime();        // display T2
   return 0;
}
```

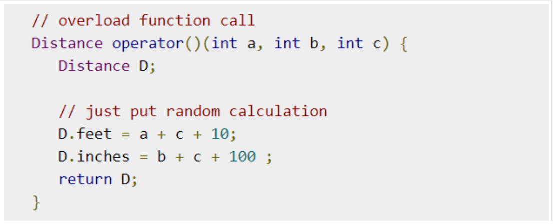
Assignment Operators Overloading in C++


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

class Distance {
   private:
      int feet;             // 0 to infinite
      int inches;           // 0 to 12

   public:
      // required constructors
      Distance() {
         feet = 0;
         inches = 0;
      }
      Distance(int f, int i) {
         feet = f;
         inches = i;
      }
      void operator = (const Distance &D ) {
         feet = D.feet;
         inches = D.inches;
      }

      // method to display distance
      void displayDistance() {
         cout << "F: " << feet <<  " I:" <<  inches << endl;
      }
};

int main() {
   Distance D1(11, 10), D2(5, 11);

   cout << "First Distance : ";
   D1.displayDistance();
   cout << "Second Distance :";
   D2.displayDistance();

   // use assignment operator
   D1 = D2;
   cout << "First Distance :";
   D1.displayDistance();

   return 0;
}
```

```
$main
First Distance : F: 11 I:10
Second Distance :F: 5 I:11
First Distance :F: 5 I:11
```

You can overload the assignment operator (=) just as you can other operators and it can be used to create an object just like the copy constructor.

Function Call Operator () Overloading in C++

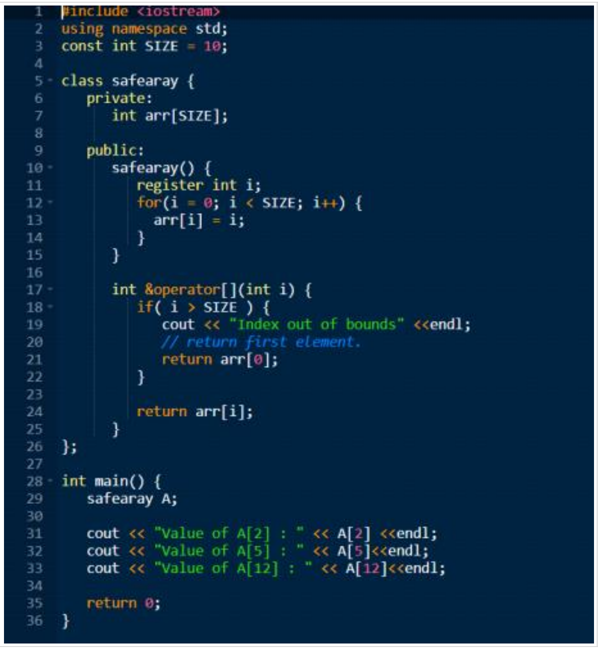

```
// overload function call
Distance operator()(int a, int b, int c) {
   Distance D;

   // just put random calculation
   D.feet = a + c + 10;
   D.inches = b + c + 100 ;
   return D;
}
```

The function call operator () can be overloaded for objects of class type.
When we overload ( ), we are not creating a new way to call a function.
Rather, we are creating an operator function that can be passed an arbitrary number of parameters.

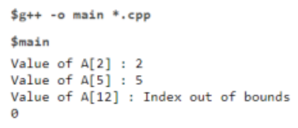
```
$g++ -o main *.cpp

$main
First Distance : F: 11 I:10
Second Distance :F: 30 I:120
```

```
int main() {
   Distance D1(11, 10), D2;

   cout << "First Distance : ";
   D1.displayDistance();

   D2 = D1(10, 10, 10); // invoke operator()
   cout << "Second Distance :";
   D2.displayDistance();

   return 0;
}
```

Subscripting [] Operator Overloading in C++



```
#include <iostream>
using namespace std;
const int SIZE = 10;

class safearay {
   private:
      int arr[SIZE];

   public:
      safearay() {
         register int i;
         for(i = 0; i < SIZE; i++) {
           arr[i] = i;
         }
      }

      int &operator[](int i) {
         if( i > SIZE ) {
            cout << "Index out of bounds" <<endl;
            // return first element.
            return arr[0];
         }

         return arr[i];
      }
};

int main() {
   safearay A;

   cout << "Value of A[2] : " << A[2] <<endl;
   cout << "Value of A[5] : " << A[5]<<endl;
   cout << "Value of A[12] : " << A[12]<<endl;

   return 0;
}
```

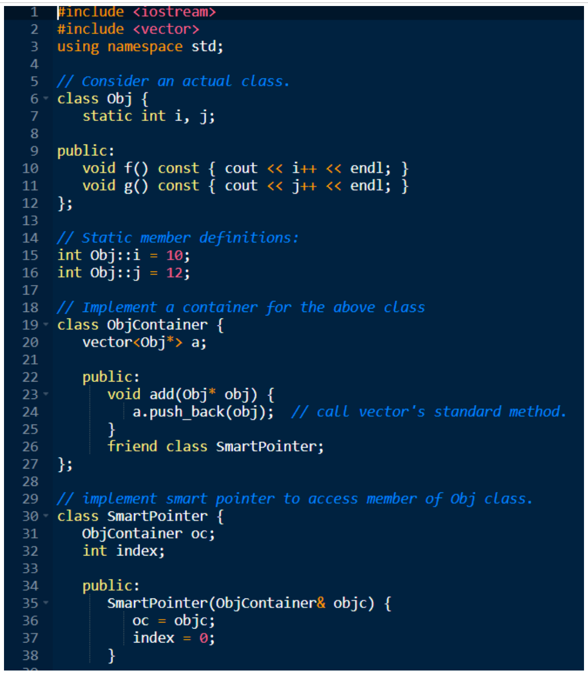


```
$g++ -o main *.cpp
$main
Value of A[2] : 2
Value of A[5] : 5
Value of A[12] : Index out of bounds
0
```

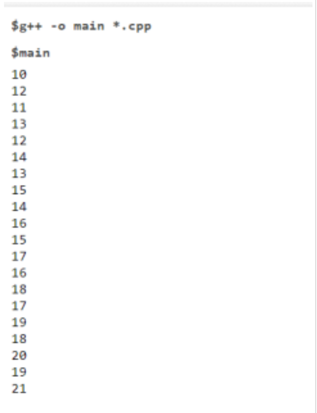
The subscript operator [] is normally used to access array elements. This operator can be overloaded to enhance the existing functionality of C++ arrays.
In this example, the an object safearray contains an array that has 10 element. And to access that array, we would like to define a new way to call subscript to that object by redefining it. Really cool way to use overloading operators !

Class Member Access Operator (->) Overloading in C++

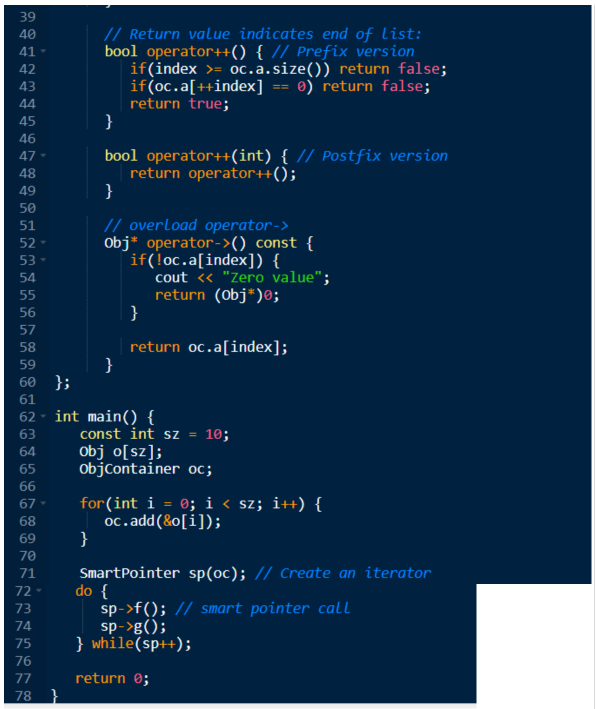
- The class member access operator (->) can be overloaded but it is bit trickier. It is defined to give a class type a "pointer-like" behavior. The operator -> must be a member function. If used, its return type must be a pointer or an object of a class to which you can apply.
- The operator-> is used often in conjunction with the pointer-dereference operator * to implement "smart pointers."
- These pointers are objects that behave like normal pointers except they perform other tasks when you access an object through them, such as automatic object deletion either when the pointer is destroyed, or the pointer is used to point to another object.



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

// Consider an actual class.
class Obj {
   static int i, j;

public:
   void f() const { cout << i++ << endl; }
   void g() const { cout << j++ << endl; }
};

// Static member definitions:
int Obj::i = 10;
int Obj::j = 12;

// Implement a container for the above class
class ObjContainer {
   vector<Obj*> a;

   public:
      void add(Obj* obj) {
         a.push_back(obj);  // call vector's standard method.
      }
      friend class SmartPointer;
};

// implement smart pointer to access member of Obj class.
class SmartPointer {
   ObjContainer oc;
   int index;

   public:
      SmartPointer(ObjContainer& objc) {
         oc = objc;
         index = 0;
      }
```



```
$g++ -o main *.cpp
$main
10
12
11
13
12
14
13
15
14
16
15
17
16
18
17
19
18
20
19
21
```



```cpp
        // Return value indicates end of list:
        bool operator++() { // Prefix version
            if(index >= oc.a.size()) return false;
            if(oc.a[++index] == 0) return false;
            return true;
        }

        bool operator++(int) { // Postfix version
            return operator++();
        }

        // overload operator->
        Obj* operator->() const {
            if(!oc.a[index]) {
                cout << "Zero value";
                return (Obj*)0;
            }

            return oc.a[index];
        }
};

int main() {
    const int sz = 10;
    Obj o[sz];
    ObjContainer oc;

    for(int i = 0; i < sz; i++) {
        oc.add(&o[i]);
    }

    SmartPointer sp(oc); // Create an iterator
    do {
        sp->f(); // smart pointer call
        sp->g();
    } while(sp++);

    return 0;
}
```