



## Operator Overloading

It is possible to overload the built-in C++ operators such as +, >=, and ++ so that they invoke different functions, depending on their operands.

That is, the + in a+b will call one function if a and b are integers, but will call a different function if a and b are objects of a class you've created.

Overloading doesn't actually add any capabilities to C++. Everything you can do with an overloaded operator you can also do with a function.

However, overloaded operators make your programs easier to write, read, understand and maintain.

Operator overloading is only another way of calling a function.

Looking at it this way, you have no reason to overload an operator except if it will make the code involving your class easier to write and especially easier to read.

Remember, code is read much more than it is written.

## Limitations

- You can't overload operators that don't already exist in C++.

You can't make up a \*\* operator for (say) exponentiation.

**You can overload only the built-in operators.**

Even a few of these, such as the dot operator (.), the scope resolution operator (::), the conditional operator (?:), and sizeof, can't be overloaded.

- The C++ operators can be divided roughly into binary and unary. Binary operators take two arguments. Examples are a+b, a-b, a/b, and so on. Unary operators take only one argument: -a, ++a, a--.

If a built-in operator is binary, then all overloads of it remain binary. It is also true for unary operators.

- Operator precedence and syntax (number of arguments) cannot be changed through overloading. For example operator \* has always higher precedence than operator +.

- All the operators used in expressions that contain only built-in data types cannot be changed. For example you can never overload operator '+' for integers so that `a = 1 + 7;` behaves differently.

At least one operand must be of a user defined type (class).

**Overloading the + operator for ComplexT objects**

```
// A class to define complex numbers
class ComplexT{
    double re, im;
public:
    :                               // Member functions
    ComplexT operator+(const ComplexT&) const; // prototype of operator+ function
};

// The Body of the function for operator +
ComplexT ComplexT::operator+(const ComplexT& z) const
{
    double re_new, im_new;
    re_new = re + z.re;
    im_new = im + z.im;
    return ComplexT(re_new, im_new); // constructor ComplexT(double,double) is needed
}

int main()
{
    ComplexT z1(1,1), z2(2,2) , z3;
    :
    z3 = z1 + z2; // Other operations
                  // like   z3 = z1.operator+(z2);
}
```

See Example: e51.cpp

**Overloading the Assignment Operator (=)**

Because assigning an object to another object of the same type is an activity most people expect to be possible, the compiler will automatically create a `type::operator=(const type &)` if you don't make one.

The behavior of this operator is member wise assignment. It assigns (copies) each member of an object to members of another object.

If this operation is sufficient you don't need to overload the assignment operator.

For example, overloading of assignment operator for complex numbers is not necessary.

```
void ComplexT::operator=(const ComplexT& z) // unnecessary
{
    re = z.re; // Member wise assignment
    im = z.im;
}
```

You don't need to write such an assignment operator function, because the operator provided by the compiler does the same thing.

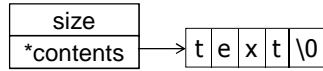
See Example: e52.cpp

With classes of any sophistication (especially if they contain pointers!) you have to explicitly create an operator=.

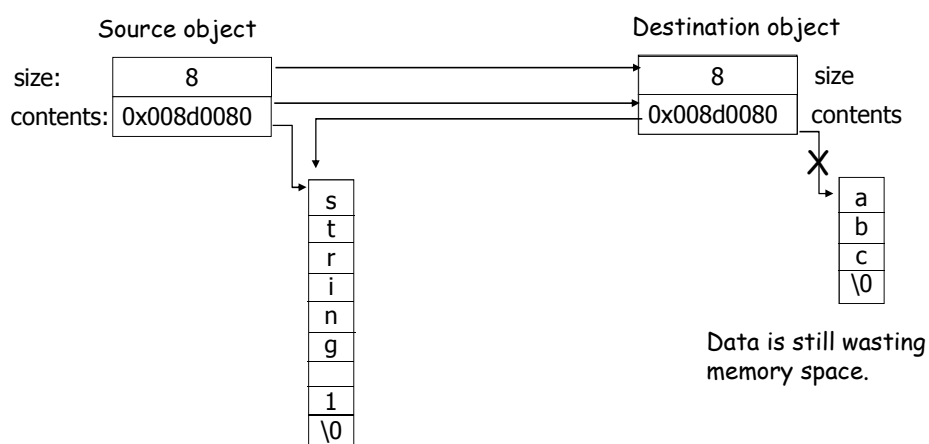
**Example**, the String class:

```
class String{
    int size;
    char *contents;
public:
    void operator=(const String &);           // assignment operator
    :                                           // Other methods
};

void String::operator=(const String &in_object)
{
    if (size != in_object.size){             // if the sizes of the source and destination
        size = in_object.size;              // objects are different
        delete [] contents;                  // The old contents is deleted
        contents = new char[size+1];        // Memory allocation for the new contents
    }
    strcpy(contents, in_object.contents);    // not a memberwise assignment
}
```

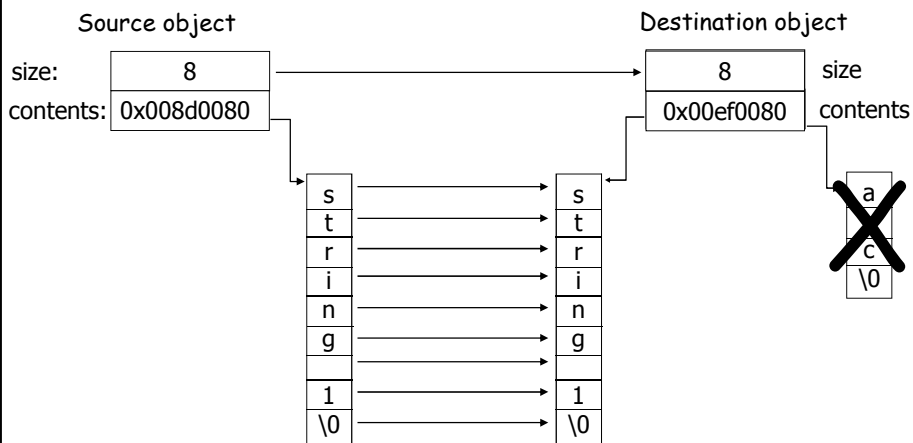


### Operator, provided by the compiler:





## Operator of the programmer:



## Return value of the assignment operator function

When there's a void return value, as shown in the previous example, you can't chain the assignment operator (as in `a = b = c`).

To fix this, the assignment operator must return a reference to the object that called the operator function (its address).

*// Assignment operator, can be chained as in `a = b = c`*

```
const String& String::operator=(const String &in_object)
```

```
{
    if (size != in_object.size){           // if the sizes of the source and destination
        size = in_object.size;           // objects are different
        delete [] contents;               // The old contents is deleted
        contents = new char[size+1];      // Memory allocation for the new contents
    }
    strcpy(contents, in_object.contents);
    return *this;                        // returns a reference to the object
}
```

See Example: e53.cpp

The difference between the **assignment operator** and the **copy constructor** is that the **copy constructor** actually **creates a new object** before copying data from another object into it, whereas the **assignment operator** copies data into an **already existing object**.

**Overloading the Subscript Operator "[ ]"**

Same rules apply to all operators. So we don't need to discuss each operator. However, we will examine some interesting operators.

One of the interesting operators is the subscript operator.

It is declared usually, in two different ways:

```
class C{
    returntype & operator [] (paramtype);    // for the left side of an assignment
    or
    const returntype & operator [] (paramtype) const; // for the right side
};
```

The first declaration can be used when the overloaded subscript operator modifies the object.

The second declaration is used with a const object; in this case, the overloaded subscript operator can access but not modify the object.

If *c* is an object of class *C*, the expression

**c[i];**

is interpreted as

**c.operator[ ](i);**

**Example:** Overloading of the subscript operator for the String class.

The operator will be used to access the *i*<sup>th</sup> character of the string.

If *i* is less than zero then the first character and if *i* is greater than the size of the string the last character will be accessed.

```
// Subscript operator
char & String::operator[](int i)
{
    if(i < 0)
        return contents[0];           // return first character
    if(i >= size)
        return contents[size-1];     // return last character
    return contents[i];              // return i th character
}
int main()
{
    String s1("String 1");
    s1[1] = 'p';                       // modifies an element of the contents
    s1.print();
    cout << " 5 th character of the string s1 is: " << s1[5] << endl;
    return 0;
}
```

See Example: e54.cpp

### Overloading the Function Call Operator ( )

The function call operator is unique in that it allows any number of arguments.

```
class C{
    returntype operator ( ) (paramtypes);
};
```

If c is an object of class C, the expression

**c(i, j, k);**

is interpreted as

**c.operator( )( i, j, k );**

**Example:** The function call operator is overloaded to print complex numbers on the screen. In this example the function call operator does not take any arguments.

```
// The function call operator without any argument, it prints a complex number
void ComplexT::operator( ) const
{
    cout << re << " , " << im << endl ;
}
```

See Example: e55.cpp

**Example:** The function call operator is overloaded to copy a part of the contents of a string into a given memory location.

In this example the function call operator takes two arguments: the address of the destination memory and the numbers of characters to copy.

```
// The function call operator with two arguments
void String::operator( ) ( char * dest, int num) const
{
    if (num > size) num=size; // if num is greater the size of the string
    for (int k=0; k < num; k++) dest[k]=contents[k];
}
```

// ----- Main function -----

```
int main( )
```

```
{
```

```
String s1("Example Program");
```

```
char * c = new char[8];
```

```
s1(c,7);
```

```
c[7] = '\0';
```

```
cout << c;
```

```
delete [] c;
```

```
return 0;
```

```
}
```

Is this statement understandable??

```
// Destination memory
// First 7 letters of string1 are copied into c
// End of string (null) character
```

See Example: e56.cpp



### Overloading Unary Operators

Unary operators operate on a single operand. Examples are the increment (++) and decrement (--) operators; the unary minus, as in -5; and the logical not (!) operator.

Unary operators take no arguments, they operate on the object for which they were called.

Normally, this operator appears on the left side of the object, as in !obj, -obj, and ++obj.

**Example:** We define ++ operator for class ComplexT to increment the real part of the complex number by 0.1 .

```
void ComplexT::operator++()
{
    re=re+0.1;
}

int main()
{
    ComplexT z(1.2, 0.5);
    ++z;                // z.operator++()
    z.print();
    return 0;
}
```

To be able to assign the incremented value to a new object, the operator function must return a reference to the object.

```
// ++ operator
// increments the real part of a complex number by 0.1
const ComplexT & ComplexT::operator++()
{
    re=re+0.1;
    return *this;
}

int main()
{
    ComplexT z1(1.2, 0.5), z2;
    z2 = ++z1;           // ++ operator is called, incremented value is assigned to z2
    z2.print();
    return 0;
}
```

See Example: e57.cpp

**"Pre" and "post" form of operators ++ and --**

Recall that ++ and -- operators come in a "pre" and "post" form. If these operators are used with an assignment statement than different forms have different meanings.

```
z2= ++ z1; // preincrement
```

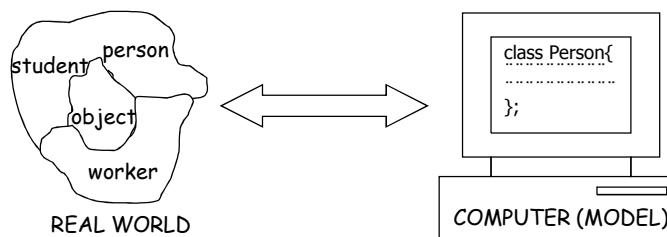
```
z2 = z1++; // postincrement
```

The declaration, **operator ++ ( )** with no parameters overloads the preincrement operator.

The declaration, **operator ++ (int)** with a single int parameter overloads the postincrement operator. Here, the int parameter serves to distinguish the postincrement form from the preincrement form. This parameter is not used.

```
ComplexT ComplexT::operator++(int) // postincrement operator
{
    ComplexT temp;
    temp = *this; // old value (original object)
    re= re + 0.1; // increment the real part
    return temp; // return old value
}
```

See Example: e58.cpp

**SUMMARY:****Benefits of Classes**

- Object-oriented programming gives us a natural and intuitive way to view the programming, namely by modeling real-world objects. There is a one to one relation between objects in the real world and objects in the program.
- Programs are easy to read and understand. The data and functions of an object are intimately tied together.
- Information hiding: Objects know how to communicate with other objects. But objects normally are not allowed to know how other objects are implemented. This property prevents data corruption. It is easy to find errors.
- Objects are active data structures. They can receive messages and perform some actions according to these messages.