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.

The + in a+b will perform an integer addition if a and b are fundamental integers but will call a programmer-defined function if a or b is an object of a class you have created, e.g., `complex3 = complex1 + complex2`.

In this way, the types you define will behave more like fundamental data types, allowing you to express operations between objects more naturally.

Overloading does not actually add any capabilities to C++. Everything you can do with an overloaded operator, you can also do with a function.

However, overloaded operators (should) 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.

Code is read much more than it is written.

Avoid overloaded operators that do not behave as expected from their built-in counterparts.

Limitations of Operator Overloading

- You **cannot** overload operators that do not already exist in C++.
  You cannot 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`, cannot 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, e.g., `-a`, `++a`, `a--`, etc.
  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 * always has 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 the operator '+' for integers so that `a = 1 + 7;` behaves differently.
  At least one operand must be of a programmer-defined type (class).

Example: Comparing complex numbers

Assume that we design a class `ComplexNumber` to define complex numbers.

Remember: Complex numbers can be expressed as `a + bi`, where a and b are real numbers.

For the complex number `z = + bi`, a is called the real part, and b is called the imaginary part.

The size of a complex number is measured by its absolute value, defined by

$$|z| = |a + bi| = \sqrt{a^2 + b^2}$$

Requirement:

We want to use the greater than operator > to compare two programmer-defined complex number objects.

```
// ComplexNumber is a programmer-defined type
ComplexNumber complex1{ 1.1, 2.3 };
ComplexNumber complex2{ 2.5, 3.7 };

if (complex1 > complex2) ...
else ...
```


Example: Overloading the greater-than operator > for complex numbers

```
class ComplexNumber {
public:
 ComplexNumber(double, double);    // Constructor to initialize re. and im.
 bool operator>(const ComplexNumber&) const; // Overloading the operator >
   :
private:
 double m_re{}, m_im{1.0};          // real and imaginary parts are initialized
};

bool ComplexNumber::operator>(const ComplexNumber& in_number) const {
 return (m_re * m_re + m_im * m_im) >
        (in_number.m_re * in_number.m_re + in_number.m_im * in_number.m_im);
}
```

If the `ComplexNumber` class contains a `getSize()` method, then we can write the operator > method as follows:

```
bool ComplexNumber::operator>(const ComplexNumber& in_number) const {
   return getSize() > in_number.getSize();
}
```


Example: Overloading the greater-than operator > for complex numbers (contd)

Since the operator > is defined in the class `ComplexNumber`, we can use it to compare the sizes of two complex numbers.

```
int main() {
  ComplexNumber complex1{ 1.1, 2.3 };
  ComplexNumber complex2{ 2.5, 3.7 };
  if (complex1 > complex2)
      cout << "complex1 is greater than complex2" << endl;
  else
     cout << "complex1 is NOT greater than complex2" << endl;
```

The object for which the operator function runs. `this` points to this object.

The argument to the operator function. `complex1.operator>(complex2);`

We can assign the addres of the complex number that has a larger size to a pointer.

```
ComplexNumber *ptrComplex;           // Pointer to complex numbers
if (complex1 > complex2) ptrComplex = &complex1;
else ptrComplex = &complex2
ptrComplex->print();       // prints the number that has the larger size
   :
```

See Example: e05_1.cpp

Example: Comparing a complex number with a double literal

A class may contain multiple functions with different signatures for the same operator.

Assume that we want to compare the size of a complex number directly with a double literal.

```
  if (complex1 > 5.7) ...
```

We should write a proper operator> function.

```
  bool operator>(double) const; // Overloading the operator

 bool ComplexNumber::operator>(double in_size) const {
  return sqrt(m_re * m_re + m_im * m_im) > in_size;
 }
```

If the class `ComplexNumber` contains a method `getSize()` that returns the size of the complex number, we can call in the operator function.

```
bool ComplexNumber::operator>(double in_size) const {
 return getSize() > in_size;
}
```

See Example: e05_2.cpp

Defaulting the equality operator ==

If you only want to compare members of two objects, you do not need to write the body of the overloading function for the operator ==.

In C++20, you can default the equality operator ==.

In this case, the compiler will generate and maintain a member function that performs the member-wise comparison.

In other words, the de default equality operator compares all corresponding member variables of the objects in the order of their declaration.

```
class ComplexNumber {
    :
// Default equality operator, member-wise comparison
bool operator==(const ComplexNumber&) const = default;
```

See Example: e05_3.cpp

If you want to compare the sizes of the complex numbers using the equality operator, you should provide a new method to overload the operator ==.

If your class contains a pointer, the default equality operator will compare the addresses in the pointers, not the contents of the memory locations pointed to by the pointer.

If you want to compare the contents of memory locations, then you must write your method for the equality operator (remember the programmer-defined `String`).

Overloading the + operator for `ComplexNumber` objects

```
class ComplexNumber{
   :
 // Signature of the method for operator +
 ComplexNumber operator+(const ComplexNumber&) const;
   :
};

// The Body of the function for operator +
ComplexNumber ComplexNumber::operator+(const ComplexNumber& in_number) const
{
   double result_re, result_im; // Local variables to store the results
   result_re = m_re + in_number.m_re;
   result_im = m_im + in_number.m_im;
   return ComplexNumber(result_re, result_im);  // constructor is called
}                                               // creates a local object
int main()
{
   ComplexNumber complex0;
   ComplexNumber complex1{ 1.1, 2.3 };
   ComplexNumber complex2{ 0, 1.0 };
   complex0 = complex1 + complex2;
                          // like complex0 = complex1.operator+(complex2)
```

Returns by-value because it returns a local object

See Example: e05_4.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 an assignment operator method `type::operator=(const type &)` if you don't make one.

The behavior of this operator is a member-wise assignment. It copies each member of an object to members of another object.

It is called the ***default* copy assignment operator**.

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 ComplexNumber::operator=(const ComplexNumber& in) // unnecessary
{
    m_re = in.m_re;          // Member-wise assignment
    m_im = in.m_im;
}
```

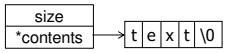
You do not need to write such an assignment operator function because the operator provided by the compiler does the same thing.

See Example: e05_5.cpp

Overloading the Copy Assignment Operator "="

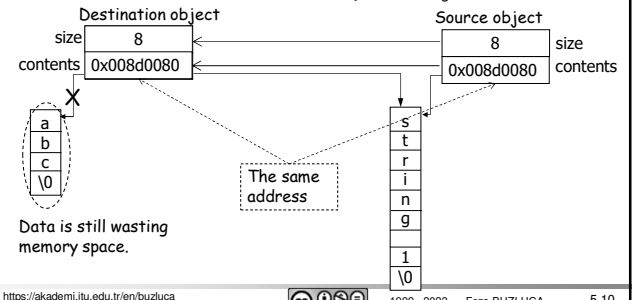
With classes of any sophistication (especially if they contain pointers!), you must create an `operator=` explicitly.

Example: The programmer-defined `String`:



Default copy assignment operator provided by the compiler:

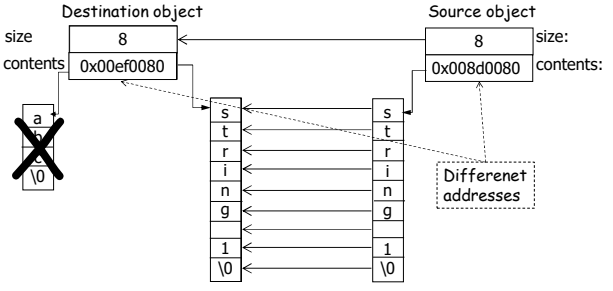
```
destination = source;  // Assignment
```



Example:
Overloading the copy assignment operator for the programmer-defined `String` class:

Copy assignment operator of the programmer:

```
destination = source;  // Assignment
```



Example:
Overloading the copy assignment operator for the programmer-defined `String`:

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

void String::operator=(const String &in_object)
{
   if (this != &in_object) {         // checking for self-assignment
     :                               // Assignment operations
   }
}
```

A programmer-defined copy assignment operator should start by checking for **self-assignment** if the class contains pointers.

Forgetting to do so may lead to fatal errors when accidentally assigning an object to itself, e.g., `string1 = string1;`

Return value of the assignment operator function

If the return value of the operator function is `void`, you cannot chain the assignment operator (as in `a = b = c`).

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

Example:
Overloading the copy assignment operator for the programmer-defined `String` class:

```
// Assignment operator can be chained as in  a = b = c
const String& String::operator=(const String& in_object)
{
  if (this != &in_object) {            // checking for self-assignment
    if (size != in_object.size) {      // if the sizes are different
       size = in_object.size;
       delete[] contents;              // The old contents is deleted
       contents = new char[size + 1];  // Memory allocation
    }
    strcpy_s(contents, size+1,in_object.contents);
  }
  return *this;                        // returns a reference to the object
}
```


The difference between the assignment operator and the copy constructor

- The **copy constructor creates a new object** before copying data from another object.
- **The copy assignment operator** copies data into **an already existing** object.

```
String firstString{ "First String" };  // Constructor is called
String secondString{ firstString };    // Copy constructor
String thirdString = secondString;     // Copy constructor
                                       // This is NOT an assignment!

secondString = firstString = thirdString;   // Assignment
```

See Example: e05_6.cpp

Deleting the Copy Assignment Operator

Just like with the copy constructor, you may not always want the compiler to generate an assignment operator for your class.

Design patterns, such as Singleton, for example, rely on objects that may not be copied.

To prevent copying, always delete both copy members. Deleting only the copy constructor or copy assignment operator is generally not a good idea.

```
String(const String&) = delete;              // Delete the copy construtor
const String& operator=(const String&) = delete;   // Delete assignement
```

The Move Assignment Operator:

Move assignment operators typically "steal" the resources the argument holds (e.g., pointers to dynamically allocated objects) rather than making copies of them.

For example, the move assignment operator for the `String` class will copy the size and contents of the source object to the destination and then assign zero to the size and `nullptr` to the contents of the source.

The source object is left empty.

Declaration for the move assignment operator:

Not constant r-value reference

Details are out of the scope of the course.

```
const String& operator=(String&&);       // Move assignment operator
```


Overloading the Subscript Operator "[ ]"

The same rules apply to all operators. So we do not need to discuss each operator. However, we will examine some interesting operators.

One of the interesting operators is the subscript operator.

It is usually declared in two different ways:

```
class AnyClass{
  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 `obj` is an object of class `AnyClass`, the expression

`obj[i];`

is interpreted as

`obj.operator[](i);`

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

The operator will be used to access the i^{th} character of the string.
If index `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 index)
{
    if(index < 0)
       return contents[0];                  // return first character
    if(index >= size)
       return contents[size-1];             // return last character
    return contents[index];                 // return i^th character
}
int main()
{
    String string1("String");
    string1[1] = 'p';        // modifies an element of the contents
    string1.print();
    cout << " 5 th character of the string is: " << string1[5] << endl;
    return 0;
}
```

See Example: e05_7.cpp

Overloading the Function Call Operator ()

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

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

If `obj` is an object of class `AnyClass`, the expression

`obj(p1, p2, p3);`

is interpreted as

`obj.operator()( p1, p2, p3 );`

Example:

The function call operator is overloaded to move the objects of the class `Point`.

In this example, the function call operator takes two arguments, i.e., coordinates.

```
// The function call operator to move point objects
bool Point::operator()(int new_x, int new_y){
    ...
}
```

See Example: e05_8.cpp

Function Objects

A **function object** is an object of a class that overloads the function call operator.

Function objects can be passed as arguments providing a powerful method to pass functions.

We will use them after we have covered templates.

Example:

`CalculateDistance` is a class that contains functions to calculate the distance of points from (0,0).

```
class CalculateDistance {
public:
  double operator()(int x, int y) const {   // Takes the coordinates
    return sqrt(x * x + y * y);              // distance from (0,0)
  }
  double operator()(const Point& in_point) const { //Takes a Point object
    return in_point.distanceFromZero();
  }
};
```


Example: Function Object (contd)

In main function we can define a function object of `CalculateDistance` and use its functions for distance calculation

```
int main()
{
  CalculateDistance calculateDistance; // A function object
  cout << "Distance of (30,40): " << calculateDistance(30, 40);
  Point point1{ 10, 20 };
  cout << "Distance of the point1 from Zero: " << calculateDistance(point1);
  return 0;
}
```

See Example: e05_9.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 and 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 ComplexNumber to increment the real part of a complex number by 0.1 .

```
void ComplexNumber::operator++()
{
     m_re = m_re + 0.1;
}

int main()
{
    ComplexNumber complex1{ 1.2, 0.5 };
    ++complex1;                                  // z.operator++()
    complex1.print();
    return 0;
}
```


Returning the `this` pointer from the overloading function:

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 ComplexNumber & ComplexNumber::operator++()
{
   m_re = m_re + 0.1;
   return *this;
}

int main()
{
  ComplexNumber complex0;
  ComplexNumber complex1{ 1.1, 2.3 };
  complex0 = ++complex1;     // operator ++ is called
    :
  return 0;
}
```

See Example: e05_10.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, different forms have different meanings.

```
z2 = ++z1;     // pre-increment. Firstly increment, then assign
z2 = z1++;     // post-increment Firstly assign, then increment
```

The declaration **operator++() w**ith no parameters overloads the pre-increment operator.

The declaration **operator++(int)** with a single `int` parameter overloads the post-increment operator.

Here, the int parameter serves to distinguish the post-increment form from the pre-increment form. This parameter is not used.

"Pre" and "post" form of operators ++ and -- (contd)

Example:

Overloading pre- and post-increment operators for the `ComplexNumber` class.

```
class ComplexNumber {
public:
   :
  const ComplexNumber& operator++();   // pre-increment ++ operator
  ComplexNumber operator++(int);       // post-increment ++ operator
   :
}
```

Return-by value because it returns a local object.

Temporary local object

```
// post-increment ++ operator
// increments the real part of a complex number by 0.1
ComplexNumber ComplexNumber::operator++(int)
{
  ComplexNumber temp{ *this };  // creates a copy of the original object
  m_re = m_re + 0.1;            // increment operation
  return temp;                  // returns the copy of the original object
}
```

See Example: e05_11.cpp