**Operator Overloading**

Use operators for customized semantics.

1. **Overload operators outside class** (actions are defined by types passed to +)

Text

Description automatically generated

**?** Operator precedence **STILL FOLLOW THE SAME PRECEDENCE**

🡪 Say we have objects a, b, c, d, and we do d = a + b \* c 🡪 b \* c still happens first!

**? Can we modify the + operator to take only 1 parameter 🡪 NO**

**? How many objects are created??**

* **Depends on the compiler**
  + Min 4
  + Max 7 (calling the "=" operator, the (a+b) argument may be passed by value)
* **When using reference** (Complex &x, Complex &y)
  + Min 4
  + Max 5

1. **Overloading operator as a member function of a class 🡪 Recommended**

Graphical user interface, text, application, email

Description automatically generated

* a + b is the same as a.operator+(b)
* Notice how we can use x.real 🡪 **In C++, access control is per-class, not per-object!**

**Note on const**:

* Putting const when declaring a member function guarantees that function will not modify member variables.
* Putting const before a parameter guarantees that the function will not modify the argument passed (when passing by reference)

However, for example if the member function is defined as:

Graphical user interface, text

Description automatically generated

Note that we are using accessors .getReal() and .getImag(). Even though these functions do not modify the values of x, the compiler won't even go into the functions to check whether they are modifying the objects or not 🡪 Still an error (it thinks that we modified const Comples &x)

Therefore, we should state that the accessor methods will not modify the objects

🡪 adding const to the accessor definition

Graphical user interface, text

Description automatically generated

**Assignment Operator**

**The default implementation of "=" operator for objects is doing member-wise copying**

* O1 (type C, stack) and O2 (type C\*, heap). O1 = \*O2 will call copy assign operator and replace all member variables of O1 by a copy of member variables of \*O2. O1 still lives on stack.
* O1 (type C\*, points to heap) and O2 (type C\*, points to heap). \*O1 = \*O2 will call copy assign operator and replace all member variables of \*O1 by a copy of member variables of \*O2. O1 still lives on stack (a pointer)
* If O1 (type C\*, points to heap), and O2 (type C\*, points to heap), doing O1 = O2 will lead to memory leak. (Deallocating pointer)
* O1 (type C\*, points to stack) and O2 (type C\*, heap). O1 = O2 will replace the address stored in O1 with the address stored in O2, and copy assign operator is not called because we are copying a pointer. In this case O1 will point to \*O2. The original C that lives on stack will be destroyed based on its scope 🡪 No memory leak

When a copy assign operator is called, it will copy the data from the memory location of source to the memory location of destination. The destination has it members replaced, but it still live at same memory location.

* The Assignment Operator must only be a **member function**

Text

Description automatically generated with medium confidence

* NOTE: However, this is not sufficient. This supports b = a, but NOT c = b = a, **because there are no return variables**!
* Therefore, to perform c = b = a, or c.operator(b.operator(a)); we have to return:

Text

Description automatically generated

The **this** keyword is the address of the object! 🡪 \*this will directly return the whole object!

**?** Is the line "return \*this" equivalent to "return x"? They are NOT!

For example, c = b, we are returning c by "return \*this", and return b by "return x"

Another example is

|  |
| --- |
| vector& vector::operator+= (const vector\_2d& rhs) {  x += rhs.x;  y += rhs.y;  return \*this;  } |