MTH 4300, Lecture 14

`Complex` (and Default Arguments and Initializer Lists);
Polymorphism and Overloading Operators;
Overloading Operators as Non-Members, and Friends
Const and Const References

E Fink

March 23, 2025

1. `Complex` (and Default Arguments and Initializer Lists)

Recall that *complex numbers* are numbers of the form $a + bi$, where $i = \sqrt{-1}$ and a and b are real numbers. So $5.1 - 2.3i$ is a complex number, as is $4 = 4 + 0i$ and $i = 0 + 1i$.

One can add two Complex numbers by simply treating them as binomials, e.g.

$$(2 + 4i) + (5 + 18i) = (7 + 22i)$$

One can also multiply two Complex numbers by treating them as binomials, except remembering that $i^2 = -1$, e.g.

$$(1 + 2i)(1 + 3i) = 1 + 5i + 6i^2 = 1 + 5i - 6 = -5 + 5i$$

In **L14x1_complex.cpp** I've provided a very basic implementation of a `Complex` class.

`Complex` (and Default Arguments and Initializer Lists)

In the class declaration, I've provided *default arguments* for one of the constructors. The declaration looks like:

```
YourClass(type1 p1, type2 p2 = 0);
```

(You don't need to rewrite the default argument when you implement your constructor.) Then, if you were to construct a `YourClass` object with a declaration that looked like
`YourClass x(100);`
with only one argument instead of two, then the value `p2` would get assigned the default value `0`, in this case. (But be aware that, if there are multiple default values provided, C++ assumes that the omitted ones are the last parameters – so be careful if you only omit some parameters.)

`Complex` (and Default Arguments and Initializer Lists)

When I implement the constructors, I use the *initializer list* style:

```
YourClass::YourClass(type1 p1, type2 p2): mem1{p1},
    mem2{p2} {
}
```

To explain: `mem1` and `mem2` are two member variables. They are assigned the values `p1` and `p2` respectively. The empty curly braces at the end designates that no further code is required by the constructor.

(This style is necessary if you wish to have member variables that are `const` – i.e., there are attributes of objects that you wish to never change values after they are initially set.)

2. Polymorphism and Overloading Operators

Polymorphism refers to being able to use the same interface to interact with different types of objects.

One example of this: using the same function name for several different types of inputs. Fact: you can create several different functions with the same name, as long as their *signatures* are different – the number and types of parameters. The compiler will determine which version of the function to use for each call, by matching the types of the arguments to the parameter lists for each version.

L14x2_quad.cpp

Polymorphism and Overloading Operators

Another example: there are many situations where you want to take familiar operators like + and ∗ and = and redefine them so that they work with new data types, in the natural ways that we might expect to use them. The idea is to make it as easy to use our classes as possible. (Remember, if you are working on a large project, you might have many different data types – the less amount of time you have to spend remembering what the member functions are, the better!)

Teaching C++ how to use old operators with new types (like `Complex`) is called operator overloading.

Before learning how to do this, first we need to understand what, say, the + operator is. It's a function! Maybe not a function which is called in the typical way, but a function nonetheless: typically, it has 2 arguments, and returns a sum.

Polymorphism and Overloading Operators

We will overload the + operator for the `Complex` class, and after that, we will be able to use that operator in client code. Here is how this works:

- First, we write a method with the very specific name of `operator+()`. To be more precise, the signature line of the function would be

  ```
  Complex Complex::operator+(Complex rhs)
  ```

- After you've written that function, go to client code, and write something like `z1 + z2`, where `z1` and `z2` are `Complex` variables. Your compiler will then, essentially, convert

  ```
  z1 + z2      to      z1.operator+(z2)
  ```

  The latter function is then executed.

Let's implement this function now. Keep in mind: if `z1.operator+(z2)` is called, then within the function, `z1` will be `this`, and `z2` will be passed to the outside argument `rhs`.

After implementing +, implement ∗.

3. Overloading Operators as Non-Members, and Friends

Generally, operators like `*`, `==`, `<<`, etc. can all be overloaded, by implementing functions with the names `operator*`, `operator==`, `operator<<`, etc.

For each of these operators, there is a similar pattern: e.g., if `x == y` appears in client code, it essentially gets converted to one of the following two forms:

`x.operator==(y)` (MEMBER function)

or

`operator==(x,y)` (NON-MEMBER function)

whichever is defined (if both are defined, there will be a compiler error). In the latter case, we are not implementing the operator as a member function!

For each operator, you get to choose how the operator works for each data type – ideally, your choices align with the ways most humans think about these operators.

Overloading Operators as Non-Members, and Friends

Next, let's try and overload ==. Just to see how this works, we'll overload this one as a NON-member: that means that `x == y` will be converted to

```
operator==(x, y)
```

Then, the signature line of the function ought to be:

```
bool operator==(Complex lhs, Complex rhs)
```

Notice: since we're not calling this with the dot, it seems like we should NOT put this in the class definition! Okay, let's implement it like a normal function now, and see how well it works out . . .

Overloading Operators as Non-Members, and Friends

. . . Almost everything we would naturally write should make perfect sense, except we have run into the issue of trying to access private members variables of `Complex` in a NON-MEMBER function. At this point, it's tempting to cast away the private members. But it's not like any of the motivations we've had for privacy have gone away!

Instead, we DO declare the function we just wrote in the class definition – but we declare it as a `friend` function. This means:

- the function is NOT a member function – that is, it is NOT called with the dot notation;
- but despite not being a member function, it is such "good friends" with the class that it gets to access private member variables/functions.

With the addition of

```
friend bool operator==(Complex, Complex);
```

to the class definition, everything works. (And the word `friend` is not necessary when you write the function.)

4. Const and Const References

When we declare a variable, we may add the qualifier `const` in front of it. This qualifier means that any attempt to modify this variable after declaration will result in a compilation error. Because of that, `const` variables need to be initialized at declaration. E.g.

```
const double PI = 3.141592653;
```

The benefits of this: it prevents against accidental modifications to the variable, and communicates the fact that this variable isn't really meant to vary – for example, if we wrote

```
if(PI = x) { // WHOOPS, SINGLE = CAUSES ASSIGNMENT!
    ...
```

we'd get a compilation error.

Const and Const References

One can also declare const reference variables: e.g.

```
int x = 15;
const int &y = x;
```

The `const` here means only that you can't use the identifier `y` to change the value of the shared object – one can still have code which changes `x`.

Note however that the following would be illegal:

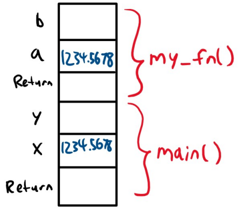
```
const int a = 15;
int &b = a;
```

We've asked the compiler to promise that the original variable a will not be changed – it can't keep that promise unless references to it are also constant.

L14x3_ref.cpp

Const and Const References

For a more compelling application of `const`, first, let's first examine the code in **L14x4_fn.cpp** which exists to remind us about the stack, and allows me to make a point. Recall that when `my_fn(x)` is called, then right before the body of `my_fn` starts to be executed, the stack frame looks like this:



Crucial: **notice how** `1234.5678` **appears TWICE in the picture – it has been COPIED**. If you have bigger pieces of data (e.g. very long strings, or complex class objects), the operation of copying is similarly repetitive in hardware, and can be time-consuming, and often unnecessary . . .

Const and Const References

. . . which is why big values are usually passed by reference. However, variables passed-by-reference are vulnerable to side-effects, whereby their value gets changed – perhaps unintentionally.

Therefore, it is very common to pass variables by const reference: e.g.

```
ret_type fn(const type &param) { ...
```

The reference ensures that `param` will refer to the original passed-in variable, rather than a copy of its value; and the `const` means that the name `param` will not be used to modify the value of the passed-in variable in the body of the function.

L14x5_param.cpp

Note that if you try to pass a `const` variable to a function by reference, then the parameter better be a `const` reference parameter, or you will receive a compilation error. Your compiler doesn't like passing a object that is supposed to be constant into a function that could change it, and non-const references can cause changes.