MTH 4300, Lecture 15

Const and Const References; Overloading << and >>; Separate Compilation

E Fink

March 23, 2025

1. 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.

L15x1_ref.cpp

Const and Const References

For a more compelling application of `const`, first, let's first examine the code in **L15x2_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.

L15x3_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.

2. Overloading << and >>

L15x4_complex.cpp

Overloading the stream extraction and insertion operators is somewhat subtle. Consider a line of code like

```
cout << x << y;
```

where `x` and `y` are ints. In many ways, this should be thought of like `3 + 4 + 5`, where first `3 + 4` is evaluated, and the sum has `5` added to it.

Here is what is happening:

- `cout` is an output stream variable which holds characters to be printed – these characters periodically get "flushed" to the screen, usually almost immediately.
- `cout << x` converts `x` to characters (e.g., the `int` 14 would get converted to characters `'1'` and `'4'`), and inserts those characters into the stream ...
- ... **and then this expression returns** `cout` ...
- ... so that then `cout << y` is left to evaluate, as before.

Overloading << and >>

Generally, streams which can be used like `cout` are all variables of type `std::ostream`.

Therefore, the code to overload << should be (maybe?)

```
// WARNING: RIGHT-ISH, BUT NOT 100% CORRECT
std::ostream operator<<(std::ostream s, Complex z) {
   s << z.re << " " << z.im;
   return s;
}
```

So, when you write `cout << z2 << z3` where `z2` and `z3` are `Complex`, the following will happen, kind of:

- `operator<<(cout, z2)` is called;
- `cout << z2.re << " " << z2.im` is executed, adding the `doubles` and `string` to the output stream (which probably gets flushed to the screen immediately) – C++ already knows how to `cout` these types;
- the function returns `cout`, so that `cout << z3` will be executed.

One issue: `re` **and** `im` **are private!** Not a problem – just add a `friend` declaration to the class. (You can't overload << and >> as members.)

Overloading << and >>

There's a second issue: as we saw before, non-reference parameter arguments get copied when a function is called.

Here's a fact which hopefully seems reasonable: most streams **should not** and **cannot** be copied. For instance, there should only be a SINGLE `cout` variable running around, in charge of all the printing to the screen! So, we pass by *reference*.

And: we don't to create a new stream to return – we want to return the exact same stream that came in. So we RETURN a reference as well.

While we're at it: there's no need to make a copy of `z`. So we'll make `z` a reference variable:

```
std::ostream& operator<<(std::ostream &s,const Complex &z)
```

Overloading << and >>

Stream EXTRACTION is similar. The biggest difference is that the types in question are `istreams`, not `ostreams`; and the second argument **definitely** needs to be a reference variable, and a non-constant one at that.

After all, `cin >> z` becomes `operator>>(cin, z)`, and the primary purpose of this isn't to produce a value – it is supposed to put a value into `z`.

So the signature (again, this is a friend) is

```
istream& operator>>(istream&, Complex&)
```

How should it work? I guess it should assume the $a + bi$ format for complex numbers – a stronger implementation might account for other formats.

3. Separate Compilation

For a large project in any language, it is customary to separate code into various files. This makes it easier to organize projects, locate code, and build incrementally. It also allows individual files, representing incomplete programs, to be compiled separately – so that if only one file needs to be changed, only one file needs to be rebuilt.

A very common strategy is to separate class definitions from client code. In fact, we can go a little further, and put

- class DECLARATION in one *header* file, typically with a `.h` extension;
- class IMPLEMENTATION (e.g. method definitions) in another, `.cpp` file;
- and client code, including `main()`, in a different `.cpp` file.

Separate Compilation

L15x5_complex_folder

How does one actually compile and run this? Here's a VS Code-oriented route.

1. Open up VS Code, and select "Open Folder", choosing the folder that contains all your files. Make sure everything is in the same folder.
2. Also, make sure that that the two `.cpp` files `#include "complex.h"`, and that `complex.h` itself has *include guards*:
   `#ifndef COMPLEX_H`
   `#define COMPLEX_H`
   and `#endif` at the end.
3. Open up a terminal in the directory containing your code, by selection "New Terminal" in VS Code.
4. For the GCC compiler, type `g++ -c complex.cpp` and press Enter. Then type `g++ -c client.cpp` and press Enter.
5. Next, type `g++ complex.o client.o -o finalprogramname`
6. And then run, with `./finalprogramname`

Separate Compilation

Notes:

- If a constructor or other method has **default parameter values**, these default values should appear in the *header* file and NOT in the implementation file.
- If a constructor has an **initializer list**, that list should appear in the *implementation* file and NOT in the header file.
- Just like other methods, the **body of the constructor** (which is sometimes just {}) should appear in the *implementation* file, not the header.
- Make sure your method signature lines in the implementation match their declarations exactly (aside from parameter names), including `consts` and `&`s!