MTH 4300, Lecture 4a

Some Notes on Assignment; Prefix and Postfix Increment

E Fink

February 9, 2025

1. Some Notes on Assignment

Everything in this chapter is worth understanding, as it will help make some things clear later when we overload operators in classes. However, I don't encourage you to rely on the features described here.

Consider the following lines of Python code:

```
x = 2
print(x + 3)  # Ok
print(x = 4)  # Not ok
```

In Python, `x + 3` is an expression that can be evaluated (to 5), and the second line would print that value out. On the other hand, `x = 4` is NOT an expression that can be evaluated – it is just an assignment statement . . . in Python.

Some Notes on Assignment

In C++, however, an expression like `x = 4` both *performs an action* (changing the value of a variable) AND *returns a value* – the value assigned to `x`, which is 4.

In other words, the lines

```
int x = 2;
cout << (x = 4) << endl;
cout << x;
```

will print out `4` twice: the first `cout` statement both reassigns `x` and returns the value 4, while the second `cout` just reads the updated value of `x`.

(Note that the parentheses are necessary on the middle line for this to work, for operator precendence reasons; otherwise, your compiler will evaluate the << before the =.)

Some Notes on Assignment

The reason for this is that C++ supports *chained assignment*. For example, consider the code

```
int y, z;
y = z = 5;
```

The first thing to note about this code is the associativity of the assignment operator: it is right-to-left. That means that if several ='s appear in one line of code, the rightmost one is evaluated first.

So on the second line, first `z = 5` is evaluated; this causes `z` to keep the value 5, and this part of the expression returns the value 5. So, the line `y = z = 5;` has now become

```
y = 5
```

Then, that assignment completes as normal, with `y` being assigned the value 5. (The value of 5 is also returned from the expression, but since there is no further assignment on this line, this is irrelevant.)

Some Notes on Assignment

You can even throw in +=, -=, etc. in to these expressions; these operators have the same precedence and same associativity.

```
int y = 1, z = 2;
y -= z += 5;
```

`z += 5` will evaluate first; this will update `z` to be 7, and also return 7. Then `y -= 7` will evaluate, and will update `y` to be -6.

L4ax1_assign.cpp

2. Prefix Increment and Postfix Increment

Same warning as in the last chapter: this is worth knowing, but use judgment when relying on the behaviors described here (using increments in a way where pre- vs post- matters).

Adding or subtracting 1 from a variable are so common that they have their own shortcuts in C++. In fact, they have TWO shortcuts each: adding 1 to `x` can be written as `++x` or `x++`, and likewise subtracting 1 from `x` can be written as `--x` or `x--`.

Let's discuss the difference between the *prefix* increment `++x` vs the *postfix* increment `x++` (similar remarks apply to the decrement). If you use one of these expressions on a line by itself, there is essentially no difference between the two versions.

However, if you embed these operations in more complex expressions and statements, there is a discernible difference:
`++x` updates `x` first and then returns the new value;
`x++` returns the old value of `x` first, and only updates `x` after returning.

Prefix Increment and Postfix Increment

For example: in

```
int x = 4;
int y = ++x;
cout << x << " " << y << endl;
```

the middle line will first update `x` to 5, and then assign that to `y`. So `5 5` will print.

But if the middle line had `x++` instead of `++x`, then the value of `x++` that is returned and assigned to `y` would be 4, and only after that would `x` be updated to 5. So `5 4` would print in that case.

L4ax2_incr.cpp

Notes: in some isolated scenarios, the postfix version is slower, because behind the scenes there are separate "old" and "new" values stored – when in doubt, most modern users opt for the prefix version.

Also, don't include more than one increment in a single expression; this is undefined behavior, since the order in which expressions are evaluated is not guaranteed.