Digression: Call-by-reference

- In C, we know function arguments are copies
  - But copying a pointer means you still point to the same (uncopied) thing
- Same also works in C++; but can also use a "reference parameter" (`&` character before var name)
- Function definition: `void f(int& x) {x = x+1;}`
- Caller writes: `f(y)`
- But it's as though the caller wrote `f(&y)` and every occurrence of `x` in the function really said `*x`.
- So that little `&` has a big meaning.

Class declaration/definition

- split class into declaration (specification) and definition
  - header contains

```
class C {
      public:
        int foo();
        void print();
 }
```

  - .cc contains

```
C::foo() {
    // implementation…
}
C::print() {
    // implementation…
}
```

Copy Constructors

- In C, we know `x=y` or `f(y)` copies `y` (if a struct, then member-wise copy)
- Same in C++, unless a copy-constructor is defined, then do whatever the copy-constructor says
- A copy-constructor by definition takes a reference parameter (else we'd need to copy the parameter, but that's what we're defining!) of the same type
- Copy constructor vs. assignment
  - Copy constructor *initializes* a new bag of bits (new variable or parameter)
  - Assignment (=) *replaces* an existing value with a new one – may need to clean up old state (free heap data?)

const

- **const** can appear in many places in C++ code
  - Basically means “won’t change”, but there are subtleties
- Examples:

```
const int default_length = 125; // cannot be reassigned
void examine (const thing &t);  // won’t change t
void examine() const;   // won’t change this
```

- **const** is important in real C++ code for reducing the chance of errors
- lack of const means the value may change but is not required to
- it is perfectly okay to pass a non-const object as **this** to a const method or as const parameter