Container Classes

Dynamic Memory

The Big Three

C++ provides automatic versions of several important methods

- copy constructor
- assignment operator
- destructor

The copy constructor creates a new object by copying another:

```
MyClass y(x);  // create a copy of x
MyClass y = x; // alternate syntax
```

It also gets called during other common operations:

- when the object is a pass-by-value argument
- when an object is returned from a function

The Big Three

C++ provides automatic versions of several important methods

- copy constructor
- assignment operator
- destructor

The assignment operator is used to assign one object to another:

```
MyClass x, y;
y = x; // assignment
```

The Big Three

C++ provides automatic versions of several important methods

- copy constructor
- assignment operator
- destructor

The destructor is called at the end of an object's life:

```
MyClass* x = new MyClass;
delete x; // destroys the object via its destructor
```

The Big Three

C++ provides automatic versions of several important methods

- copy constructor
- assignment operator
- destructor

The destructor is called at the end of an object's life:

```
int create_local_variable() {
    MyClass y; // gets destroyed when the function exits
}
```

Variables are also automatically destroyed when leaving their scope

- the variable ceases to exist after the function exits, because it gets destroyed!

The Big Three

C++ provides automatic versions of several important methods

- copy constructor
- assignment operator
- destructor

Default behavior:

- the copy constructor and assignment operator both simply copy the value of each member variable from the old object to the new
- the default destructor simply calls the destructors of each of the object's members

The Big Three

C++ provides automatic versions of several important methods

- copy constructor
- assignment operator
- destructor

This default behavior is generally exactly what you want

- that's why it's the default!

However, when a class uses dynamic memory...

- the automatic implementations are no longer adequate

A Class with Dynamic Memory

Here's a simple class that uses dynamic memory:

```
class Number {
    public:
        Number(int n = 0) { ptr = new int(n); }

    private:
        int* ptr;
};
```

A Class with Dynamic Memory

It has one private data member (a pointer!):

```
// a pointer to an int
int* ptr;
```

The constructor simply allocates a single integer:

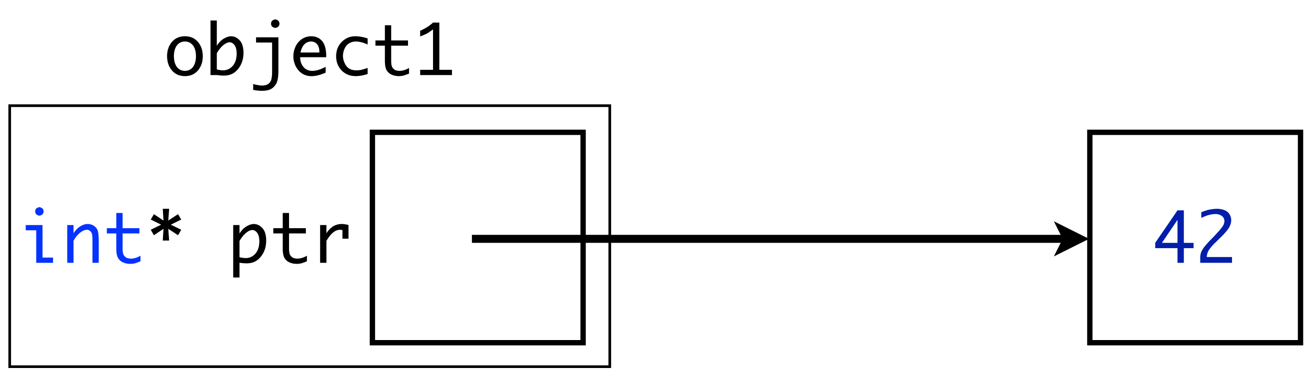
```
// dynamically allocates a single integer
Number(int n = 0) { ptr = new int(n); }
```

A Class with Dynamic Memory

Creating an instance of the class:

```
Number object1(42);
```

We can visualize the object like this:



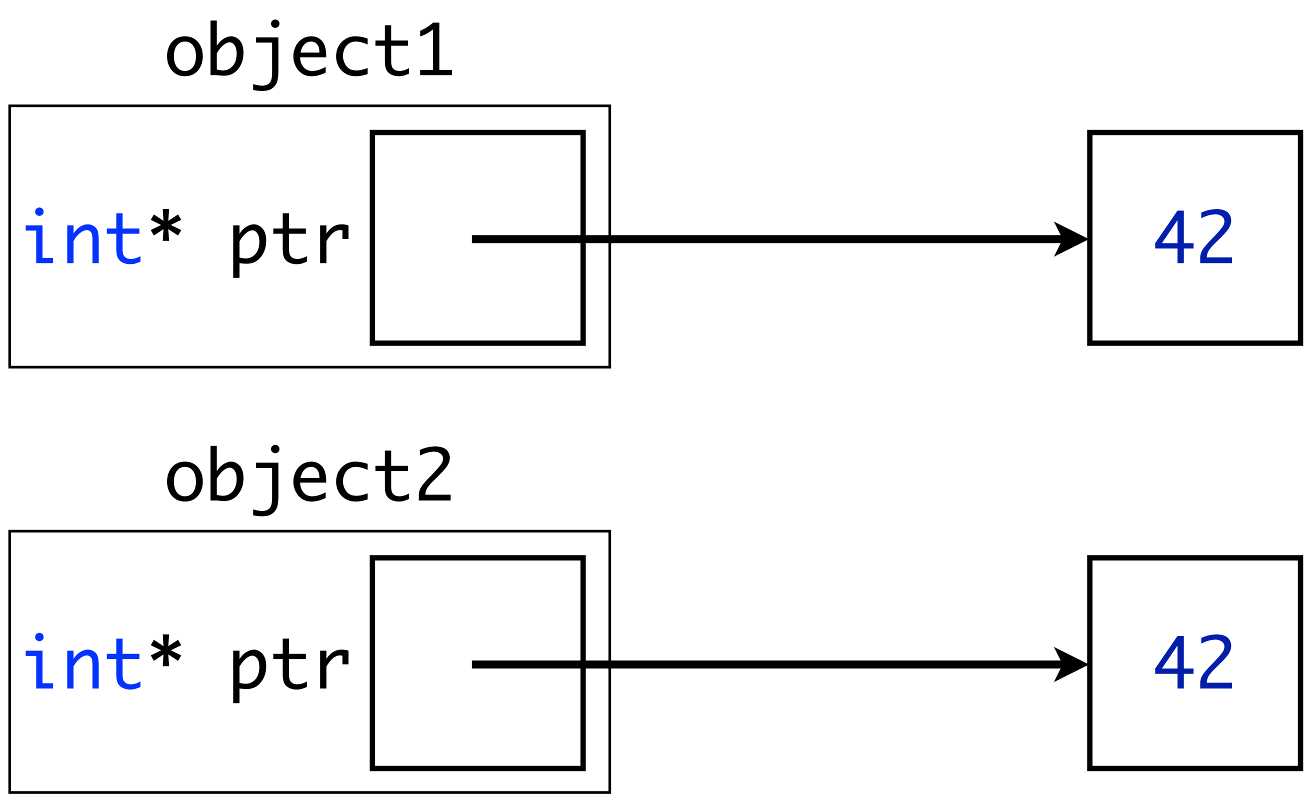
The Copy Constructor

Creating a second instance via the copy constructor:

```
Number object1(42);

Number object2(object1); // copy constructor
```

What we WANT to happen:



`object2` should be an independent copy of `object1`...

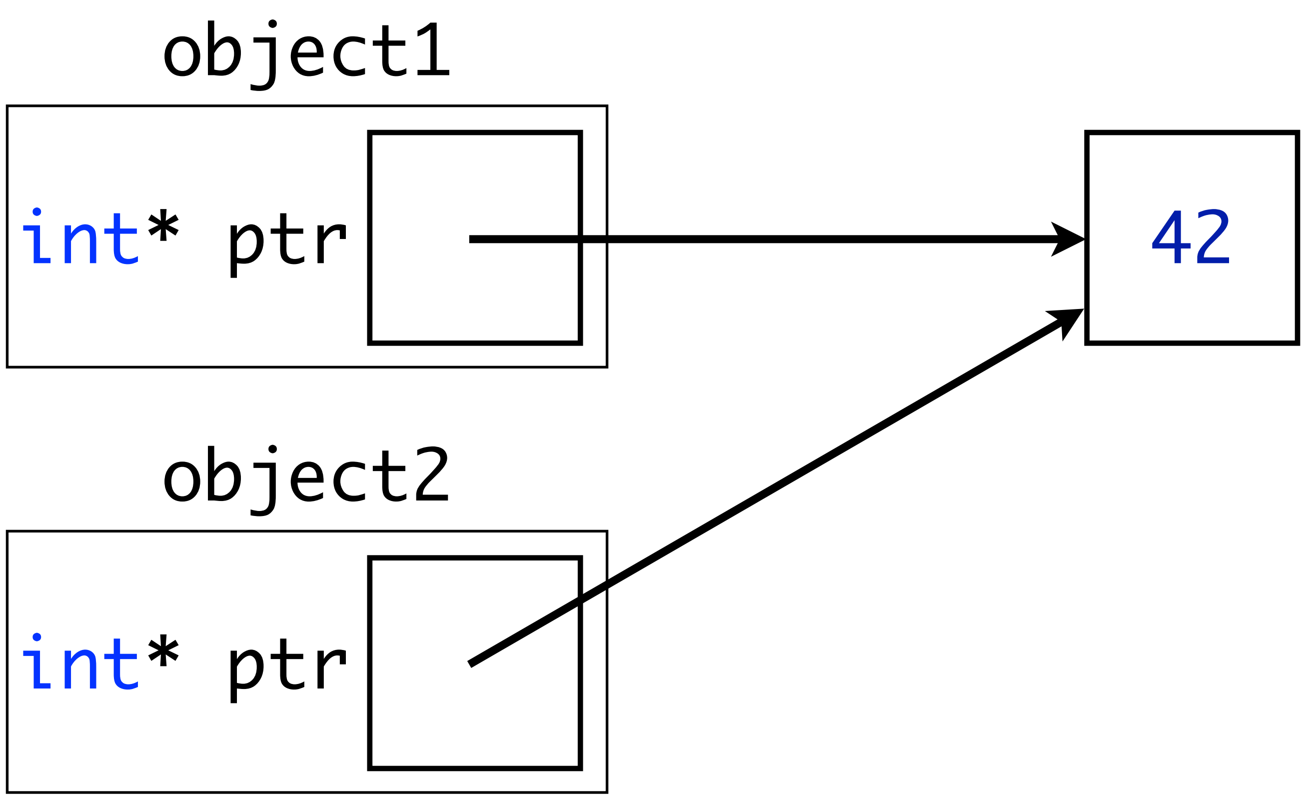
The Copy Constructor

Creating a second instance via the copy constructor:

```
Number object1(42);

Number object2(object1); // copy constructor
```

What ACTUALLY happens:



But the copy points at the original's value, not its own!

The Copy Constructor

Creating a second instance via the copy constructor:

```
Number object1(42);

Number object2(object1); // copy constructor
```

What ACTUALLY happens:

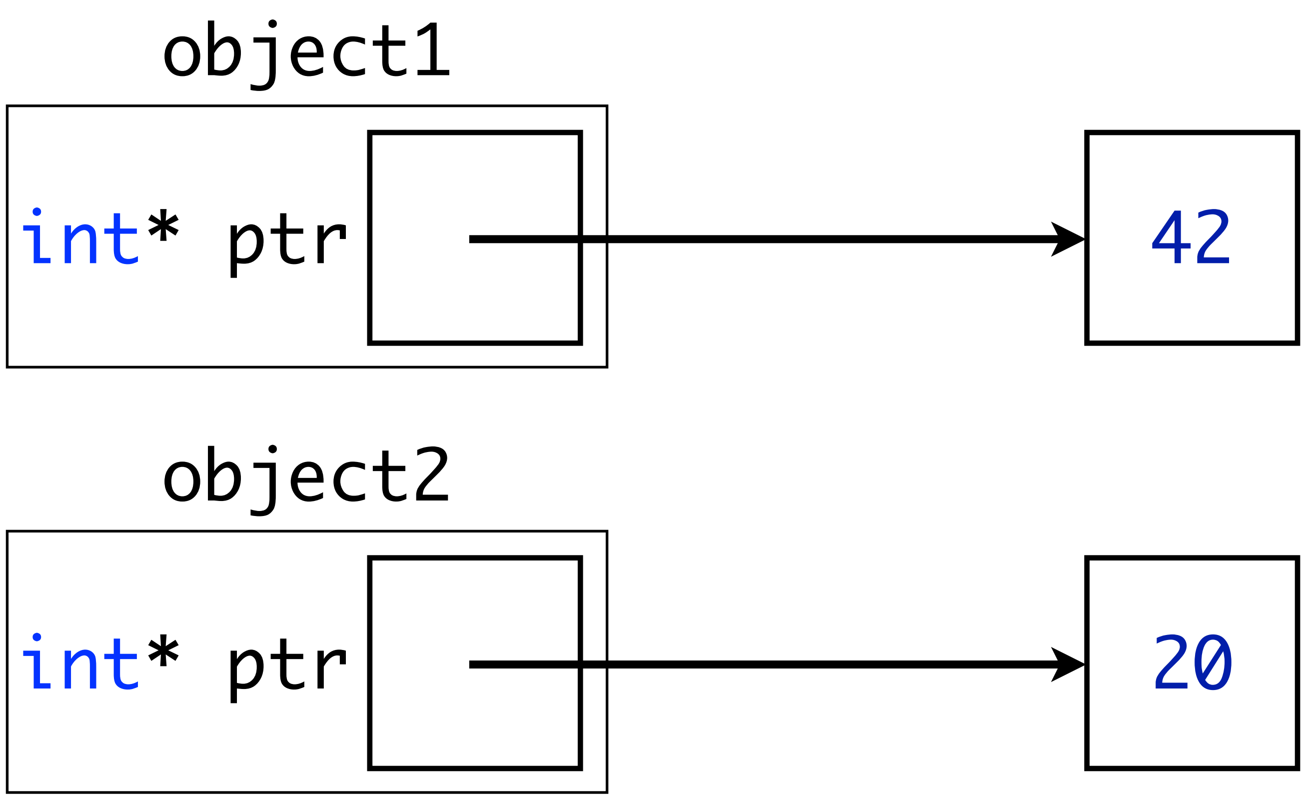
- the default copy constructor simply copies the data members from the old object to the new one
- this is great for statically allocated variables (perfect, actually)...
- however, when the data member is a pointer, the pointer—*not* what it points at—gets copied
- this means that there is now a second copy of the pointer, but its value (a memory address) is still the exact same
- the copy of the pointer still points at the original's value!

The Assignment Operator

Same problem for the default assignment operator...

```
Number object1(42), object2(20);
```

The objects start like this:



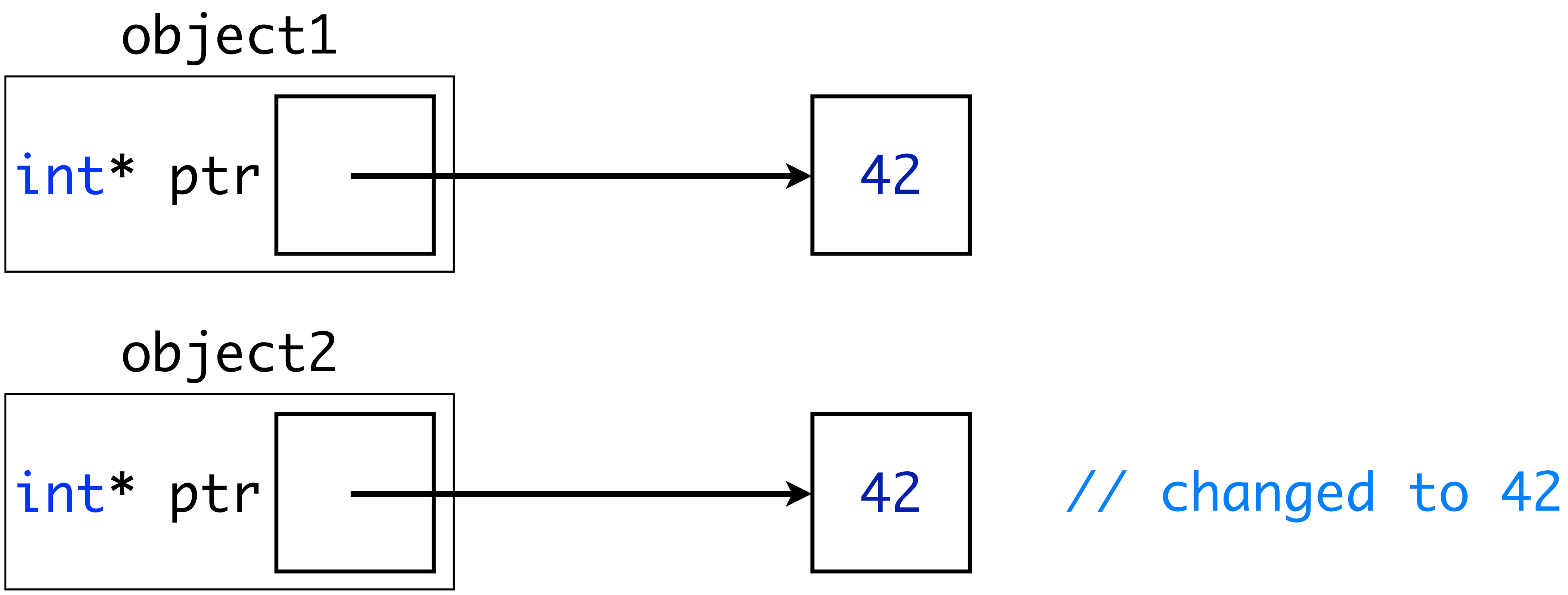
The Assignment Operator

Same problem for the default assignment operator...

```
Number object1(42), object2(20);

object2 = object1; // assignment
```

What we WANT to happen:



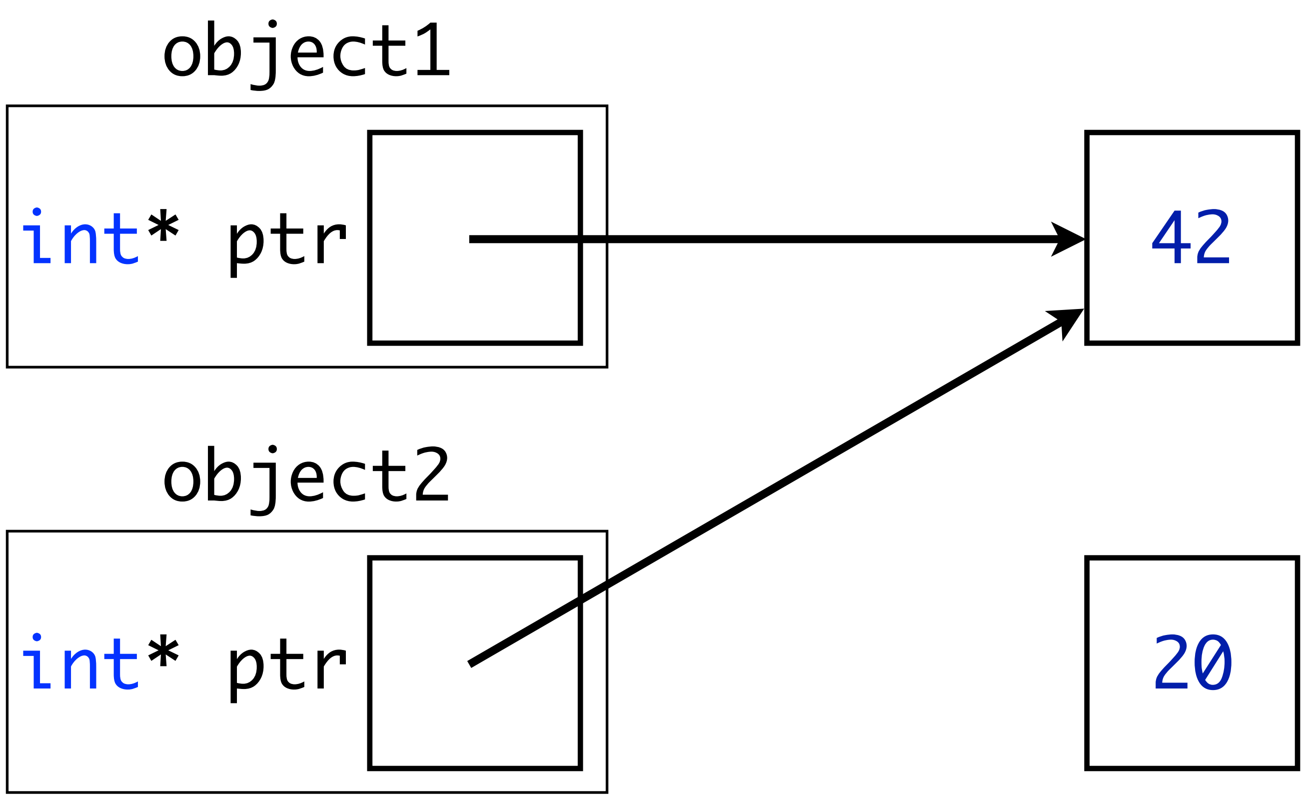
The Assignment Operator

Same problem for the default assignment operator...

```
Number object1(42), object2(20);

object2 = object1; // assignment
```

What ACTUALLY happens:



Same problem as before,
but now also a memory leak!

Value Semantics

Value semantics refer to how a class behaves when making copies

For a class with dynamic memory:

- you <u>must</u> implement your own copy constructor and assignment operator!
- otherwise, copying will not behave like you (or the users of your class) expect

Uh oh...

RANDOM COMIC!!!

Destructors

Normally there is no need for a destructor...

- the default behavior automatically calls the destructor of any data members that are objects and takes care of primitives
- however, it does NOT free memory that your class allocated on its own!

You must implement a destructor if your class allocates memory!

- this destructor should delete all memory that was allocated; nothing else is required

Implementing the Big Three

Here's the same simple class as before:

```
class Number {
    public:
        Number(int n = 0) { ptr = new int(n); }

    private:
        int* ptr;
};
```

Implementing the Big Three

And here it is with additional prototypes for the big three:

```
class Number {
    public:
        Number(int n = 0) { ptr = new int(n); }

        Number(const Number&);
        ~Number();
        void operator =(const Number&);

    private:
        int* ptr;
};
```

The Copy Constructor

The copy construct must allocate memory for the new object

- then copy the <u>values</u> from the old object (not the pointer itself)

The copy constructor implementation:

```
// creates a new object by copying another
Number::Number(const Number& source) {
    // allocate a new int for this object
    ptr = new int;

    // and copy the value from the original object
    *ptr = *(source.ptr);
}
```

The Copy Constructor

The argument to the copy constructor must be passed by reference

- why?

An incorrect copy constructor implementation:

```
// creates a new object by copying another
Number::Number(Number source) {  // bad!!!
    // code
}
```

Passing by value creates a COPY of the argument

- and what creates copies of an object?
- the copy constructor!

The Assignment Operator

The assignment operator is similar to the copy constructor...

However, there are some differences:

- the copy constructor creates an object from scratch and must always allocate memory as appropriate
- the assignment operator operates on an existing object and thus already has some memory allocated, though it may still need to allocate or deallocate memory
- the implementation of the assignment operator must account for the possibility of self-assignment, where an object is assigned to itself (`b = b`)

The Assignment Operator

Self-assignment may seem pointless...

- however, your implementation should still correctly handle it
- this is accomplished by immediately returning from the function if self-assignment is detected

How to detect self-assignment:

```
void Number::operator =(const Number& source) {
    if (this == &source) {  // self-assignment?
        return;
    }

    // no self assignment... carry on!
}
```

The Assignment Operator

Pay close attention to this line:

```
if (this == &origin) { ... }
```

In member functions, this is a pointer to the calling object

- the value of this changes depending on which object was used to call the function

Also, remember the address operator (&):

- it returns the memory address of whatever variable it precedes
- this address is basically a pointer to the variable

The Assignment Operator

Check for self-assignment by comparing the two pointers:

```
this     // a pointer to the calling object
&origin  // a pointer to the argument
```

If the two pointers are equal, then they point to the same object

```
if (this == &origin) {
    // self-assignment! nothing to do, so just return
    return;
}
```

The Assignment Operator

The assignment operator implementation:

```
// assigns this object the same values as @origin
void Number::operator =(const Number& origin) {
    // check for self-assignment
    if (this == &origin) {
        return;
    }

    // allocate memory and copy values as appropriate
    *ptr = *(origin.ptr);
}
```

The Assignment Operator

Ideally, the assignment operator should support assignment chaining:

by reference

```
Number& Number::operator =(const Number& origin) {
    // returns the calling object
    return *this;
}
```

the calling object

You can do this by returning the calling object <u>by reference</u>

Example of assignment chaining:

```
Number n1, n2, n3(42);
n1 = n2 = n3;  // assignment chaining
```

The Destructor

The destructor must deallocate any dynamically allocated memory

The function itself must follow a few rules, though:

- it must be named the same as the class, but prefixed with a tilde (~)
- it must accept no arguments and must not have a return type declared
- destructors are never called explicitly; C++ does this for us

The prototype for our destructor will look like this:

```
~Number();
```

The Destructor

The destructor implementation:

```
// frees the memory allocated by this object
Number::~Number() {
    delete ptr;
}
```

Easy!

If your class uses dynamic memory...

You need to implement the big 3!

(copy constructor, assignment operator, and destructor)