COP3503

Copy Constructors, Assignment Operators, and Destructors

AKA "The Big Three"

What Are the "Big Three"?

- Three **class member functions** (one we've already seen!)

Copy Constructor	Copy Assignment Operator	Destructor
Used to construct an object from another, existing object	Used to copy one object into another object	Used when an object is destroyed—when it falls out of scope, or when delete is called on a pointer to an object

- Three special functions which, if you do not declare them, get **implicitly created** for you.
  - The implicit versions are not necessarily what you want in your own program.
  - Especially if you are using **dynamic memory**

Free functions! Yay!

Free functions that don't work right! Boo!

Implicitly Created Functions
The compiler writes one for you, if you don't. (Why? Because C++ requires objects to have these functions)

They are written in a standard way, which **might** be what you need. Or… that standard functionality will break your program

Copy Constructor

- A special constructor which is called when, and only when, a class object is **initialized with another instance of the same class**.
- The purpose is to construct a new object, as a **copy** of the other object.

```
LineItem a;       // Constructor
LineItem b(a);    // Copy Constructor
LineItem c = a;   // Also Copy Constructor (just
```

Copying primitives is easy; how exactly do we copy objects? Like any other class operation, with a function!

```
// In a similar sense...
int x = 5; // "Construct" an integer
int y = x; // "Construct" a copy of x
```

Copy Constructor

How do you copy?

```
class LineItem
{
    string _name;
    string _description;
    int _quantity;
    float _price;
public:
    LineItem(const LineItem& otherObject); // Copy constructor
};

LineItem::LineItem(const LineItem& otherObject)
{
    // Copy EVERYTHING from the "other" object into "th
    this->_name          = otherObject._name;
    this->_description   = otherObject._description;
    this->_quantity      = otherObject._quantity;
    this->_price         = otherObject._price;
}
```

Style Reminder:
You don't have to use the **this** keyword and the indirect membership operator to access variables in a class; it's implied if you leave it out.

This is a simple **member-to-member copy**.

The implicitly created copy constructor will do exactly this, if you don't write it.

You don't have to write this **unless** you are dealing with dynamic memory.

How Can We Access Private Variables?

```
class LineItem
{
    string _name;
    string _description;
    int _quantity;
    float _price;
public:
    LineItem(const LineItem& otherObject); // Copy constructor

              t LineItem& otherObject)

    // Copy EVERYTHING from the "other" object into "this"
    _name        = otherObject._name;
    _description = otherObject._description;
    _quantity    = otherObject._quantity;
    _price       = otherObject._price;
}
```

If all of these variables are private…

Quick reminder: `this->` can be omitted if you want; it's optional.

Because this is a member function of the `LineItem` class, it "knows" about private variables.

It's like two members of a club knowing something about the club that isn't made open to the public.

How can we access them here? Wouldn't we need accessor functions?

Copy Constructor Syntax

Given a class:

```
class ExampleClass
{
public:
    ExampleClass(const ExampleClass& otherObject);
};
```

Each copy constructor works differently, but the “signature” of each is the same: a constructor with a const reference to an instance of the class

The copy constructor will **always** have this format:

```
ExampleClass::ExampleClass(const ExampleClass& otherObject)
{

}
```

A **const** reference, so we don’t change the other thing.

One parameter, always: a **constant reference** to some instance of this class

A **reference**, to pass it quickly

Passing by value would create a copy and call this very function—hello infinite recursion!

Copy Assignment Operator

- A function that is called when you **assign an existing object to another existing object**.
- They are **largely similar to copy constructors**, with a few differences.
- They will **overwrite existing values** (a copy constructor has no “old” values to overwrite).
- They **can be invoked more than once** on an object (copy constructors can only be called once).

```
LineItem a, b; // Default constructor

// Copy constructor—assign 'a' to 'c' WHILE CREATING 'c'
LineItem c = a;

c = b; // Copy Assignment operator, ‘c' already exists, overwrite its values
c = a; // Copy Assignment operator
```

Copy Assignment Operator

```
class LineItem
{
    string _name;
    string _description;
    int _quantity;
    float _price;
public:
    LineItem(const LineItem& otherObject); // Co
    LineItem& operator=(const LineItem& otherObj
};

LineItem& LineItem::operator=(const LineItem& ot
{
    // Copy EVERYTHING from the "other" object i
    _name        = otherObject._name;
    _description = otherObject._description;
    _quantity    = otherObject._quantity;
    _price       = otherObject._price;

    return *this; // This is always the last line of this function
}
```

This is a simple **member-to-member copy** (just like the copy constructor!).

A free version of this function will be created for you automatically.

If you aren't using dynamic memory, you **do not** need to write this function.

We have to understand how this function works so we **can** write it when it's needed.

Copy Assignment Operator Syntax

```
class ExampleClass
{
public:
    ExampleClass& operator=(const ExampleClass& otherObject);
};
```

This is an example of **operator overloading** – a topic for another time!

Return type: **Always** a **non-const reference** to an instance of this class

Not a typo, the name of this function is `operator=`.

One parameter, always: a **constant reference** to some instance of this class

A **reference**, to pass it quickly

A **const** reference, so we don't change the other thing.

Hmmm, this sounds familiar…

Copy Constructor Syntax

```
ExampleClass& ExampleClass::operator=(const ExampleClass& otherObject)
{
    // Assume some super-sweet copying code here

    return *this; // What's the point of this line?
}
```

If we didn't pass references, we'd be creating unnecessary copies (instances) of object—not good.

- **this** is a **pointer** to an object, **not** an object.
- This function returns a **reference** to an object.
- By **dereferencing this**, we get an object, to which we can bind a reference.
- Why do all that? The reason for all of this, is so we can do **this**:

We have 4 objects already, we just want to copy the **values of the data** inside those objects.

What about passing by pointer? Then we'd have to worry about addresses and dereferencing.

```
LineItem a, b, c, d;
// Use objects here, program sets/changes values, and then...
b = c = a = d; // Chaining assignment operations together
```

`a = d` returns a reference to the newly changed `a`, which is now passed to `c = a`…

…which returns a reference to the newly changed `c`, which is now sent to `b = c`…

What About Duplicating Code?

```
LineItem::LineItem(const LineItem &otherObject)
{
    _name        = otherObject._name;
    _description = otherObject._description;
    _quantity    = otherObject._quantity;
    _price       = otherObject._price;
}

LineItem& LineItem::operator=(const LineItem& otherObject)
{
    _name        = otherObject._name;
    _description = otherObject._description;
    _quantity    = otherObject._quantity;
    _price       = otherObject._price;
    return *this;
}
```

Duplicating code can lead to bugs later (also, it can be tedious to repeat yourself)

Put it in a function!

```
SomeFunction(otherObject);
```

Copy Assignment Operator

```
// Purpose: Help copy constructor and copy assignment
// operator assign "that" into "this"
void LineItem::Set(const LineItem& otherObject)
{
    _name        = otherObject._name;
    _description = otherObject._description;
    _quantity    = otherObject._quantity;
    _price       = otherObject._price;
}

LineItem::LineItem(const LineItem &otherObject)
{
    Set(otherObject);
}

LineItem& LineItem::operator=(const LineItem& otherObject)
{
    Set(otherObject);
    return *this;
}
```

Now if your class changes (add, remove, rename variables), you can just update this one **Set** function instead of multiple locations.

Fewer changes to make, fewer opportunities for mistakes!

Code reuse where you can

Unique functionality where you have to

The Destructor

- A function which is called when an object is **destroyed**, either:

- When it **falls out of scope** (like a temporary variable in a function), or…

- When `delete` is called on a **pointer to an object**

- The purpose of a destructor is to clean up or “shut down” an object, which could involve:
  - Deleting any **dynamically allocated memory**.
  - Possibly notifying another object/function that destruction has occurred.
  - Possibly printing something out to the screen as a result of this object finishing its task.

This is going to be the one we worry about in this course.

Destructor Syntax

```
class ExampleClass
{
public:
    ~ExampleClass(); // Prototype
};
```

The name of the function is like a constructor (the name of the class), but with a tilde (**~**) in front of the function.

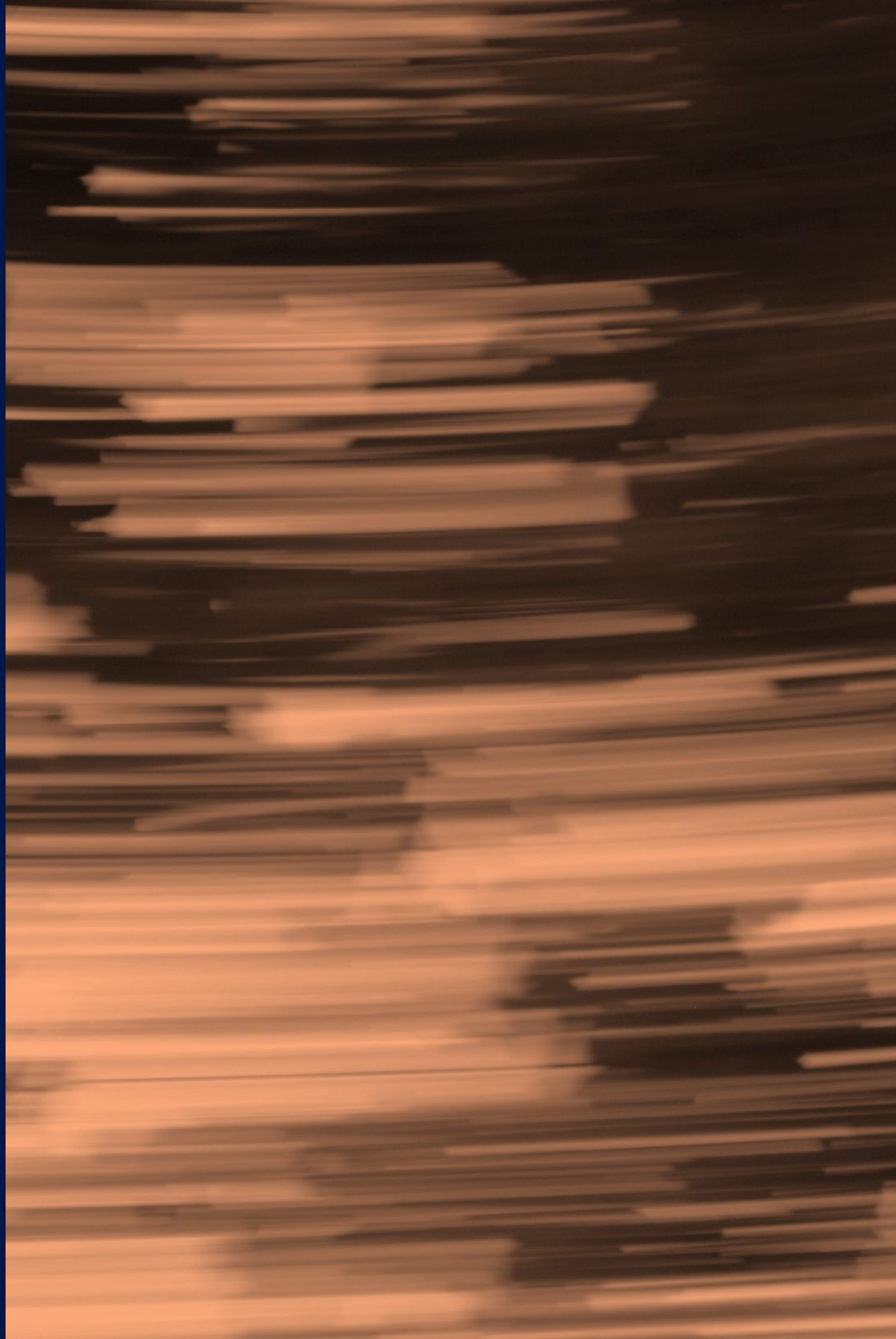
```
ExampleClass::~ExampleClass() // Definition
{

}
```

The implicitly declared version of any destructor does… absolutely nothing.

The assumption is, you have nothing to clean up (i.e. no **`new`** memory you have to **`delete`**).

Why Do You Need the Big Three?

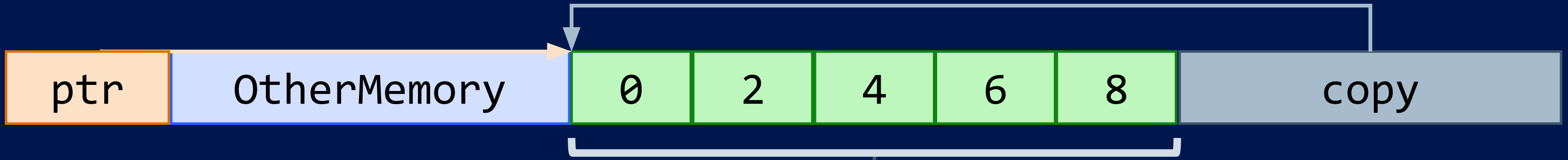
- Programs copy lots of objects and the data they contain.
- Dynamically allocated data must be copied differently, because of pointers.
- Pointers only store memory addresses, the location of the real data.
- Copying pointers (locations) is a **shallow copy**.
- To copy the data itself, we want to perform a **deep copy**.
- Deep copy means **allocating more space** for a copy of the data.
- Also, we have to **clean it all up** (i.e. use destructors!).



Shallow Copy Example

A Simple Array

```
int* ptr = new int[5];
for (int i = 0; i < 5; i++)
    ptr[i] = i * 2; // Set values to 0, 2, 4, 6, 8
```

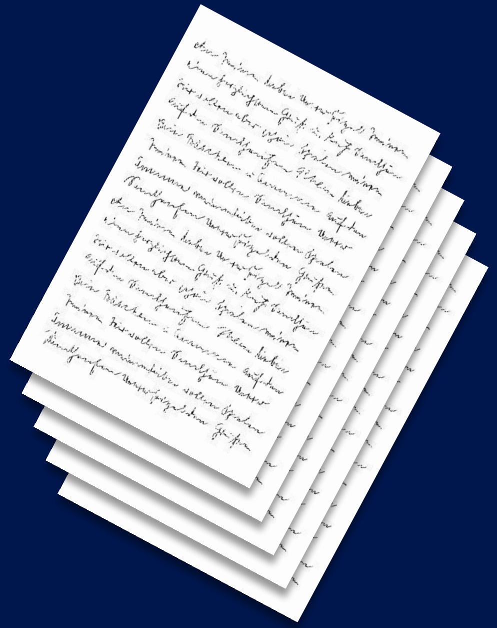



```
int* copy = ptr; // "Copy" the array
```

Now, how many integers do we have?

To copy the data, we need to do a **deep copy**.

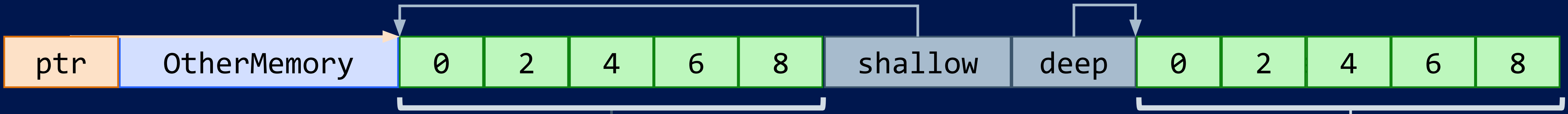
Still five! We haven't made any copies of the **data**, just **access** to the data.



Deep Copy

Duplicate the data

```
int* ptr = new int[5];
for (int i = 0; i < 5; i++)
    ptr[i] = i * 2; // Set values to 0, 2, 4, 6, 8
```




```
int* shallow = ptr; // Shallow copy the array
```

```
// Deep copy takes two steps:
// 1. Allocate space for the duplicate data
int* deep = new int[5];

// 2. Copy the data values from the original location
for (int i = 0; i < 5; i++)
    deep[i] = ptr[i];
```

Shallow Copy in a Class

```
int main()
{
    MemoryExample A, B;
    A.AllocateMemory(3);

    B = A;

    /* assume some awesome code here */
    return 0;
}
```

What's going to happen here?

The (implicitly created) **copy assignment operator** will be called.

```
class MemoryExample
{
    int      _count;
    float*   _someFloats;
    string   _label;
public:
    MemoryExample()
    {
        _count = 0;
        _someFloats = nullptr;
        _label = "Batman";
    }
    void AllocateMemory(int count)
    {
        _count = count;
        _someFloats = new
float[_count];
    }
    ~MemoryExample()
    {
        delete[] _someFloats;
    }
};
```

```
class MemoryExample
{
    int     _count;
    float*  _someFloats;
    string  _label;
public:
    MemoryExample()
    {
        _count = 0;
        _someFloats = nullptr;
        _label = "Batman";
    }
    void AllocateMemory(int count)
    {
        _count = count;
        _someFloats = new
float[_count];
    }
    ~MemoryExample()
    {
        delete[] _someFloats;
    }
};
```

With no **copy assignment operator** explicitly written, one is provided for you.

That assignment operator performs a simple member-to-member copy.

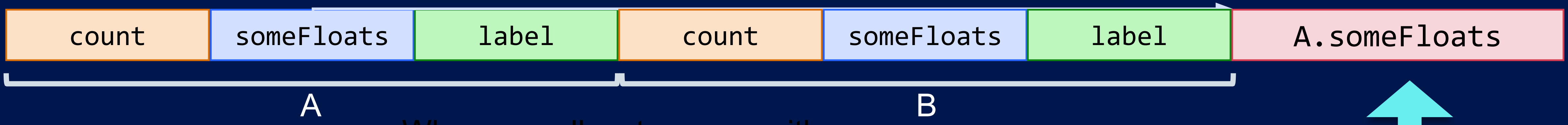
```
// Default behavior, simple member-to-member copy
MemoryExample& operator=(const MemoryExample&
rhs)
{
    this->_count        = rhs._count;
    this->_someFloats   = rhs._someFloats;
    this->_label        = rhs._label;

    return *this;
}
```

1 of the 3 variables needs a deep copy.

Shallow Copy

```
class MemoryExample
{
    int      _count;
    float*   _someFloats;
    string   _label;
public:
     void AllocateMemory(int count);
}
```

A.someFloats points to some other memory address where the data can be found.



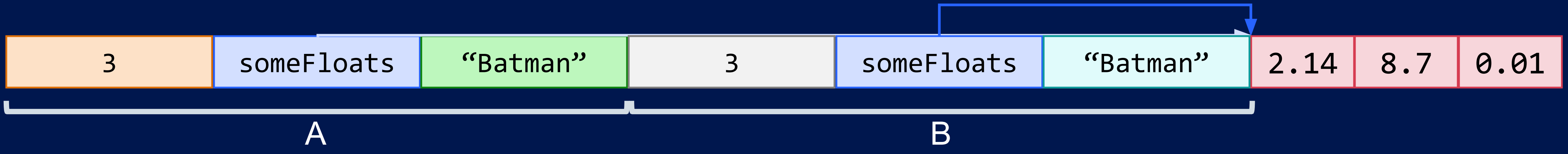
When you allocate space with new, that memory is located elsewhere

```
MemoryExample A, B;

A.AllocateMemory(3);
```

Shallow Copy

```
MemoryExample A, B;
A.AllocateMemory(3);

B = A; // What will happen here?
```

```
// Member-to-member copy...
B._count        = A._count;
B._someFloats   = A._someFloats;
B._label        = A._label;
```



A and B are now equal… sort of.

What if we change the array? Or delete it?

A shallow copy will cause your code to **break somewhere else**.

Shallow Copy

```
MemoryExample A;
A.AllocateMemory(3);
Foo(A);

void Foo(const MemoryExample& A)
{
    MemoryExample B;
    B = A;

    // do something with B
} // B.~MemoryExample();
```

What happens when **B** falls out of scope, and its destructor is called?

Clean up after ourselves, like a responsible class!

```
class MemoryExample
{
    int      _count;
    float*   _someFloats;
    string   _label;
public:
    MemoryExample()
    {
        _count = 0;
        _someFloats = nullptr;
        _label = "Batman";
    }
    void AllocateMemory(int count)
    {
        _count = count;
        _someFloats = new
float[_count];
    }
    ~MemoryExample()
        delete[] _someFloats;
    }
};
```

Deleting Shared Memory

```
MemoryExample A;
A.AllocateMemory(3);
Foo(A);

void Foo(const MemoryExample& A)
{
    MemoryExample B;
    B = A;

    // do something with B
} // B.~MemoryExample();
```

```
~MemoryExample()
{
    delete[] someFloats;
}
```

When the function finishes, the temporary object falls out of scope, invoking the destructor.

Dangling pointer: A pointer that does not point to valid memory

`A.someFloats` is now what is called a **dangling pointer**.

It points to **something**… but isn't really usable.

3	someFloats	"Batman"	Unusable values	Deallocated memory

A

What if A tries to use that memory now? It has no idea this data was just deleted…

Solution: Deep Copy

```
MemoryExample& operator=(const MemoryExample& otherObject)
{
    _count = otherObject._count; // Simple assignment
    _label = otherObject._label; // std::string has its own copy assignment operator

    // Delete any old memory previously allocated
    delete[] _someFloats;

    /*===== Deep copy requires two steps =====*/
    // Step 1, allocate space
    _someFloats = new float[_count];

    // Step 2, copy data
    for (int i = 0; i < _count; i++)
        _someFloats[i] = otherObject._someFloats[i];
     return *this;
}
```

All of this work "behind the scenes" is to ensure that one "simple" operation works:

```
MemoryExample one, two;
one = two; // Simple... sort of!
```

The Big Three Are Vital!

- Objects get copied time and again in our code.
- Our code has to use memory properly, classes especially.
- We can't always (easily) tell where these functions will get used in our programs.

```
MemoryExample obj;
obj.AllocateMemory(5);    // No copying or assignment?
obj.Print();              // No need for the Big Three?
// Maybe just write the Big One (the destructor)?

// What about this code?
MemoryExample x;
vector<MemoryExample> objects;
objects.push_back(x);
objects.push_back(x);
```

How does a vector handle its data internally? Will the copy constructor of `MemoryExample` be called?

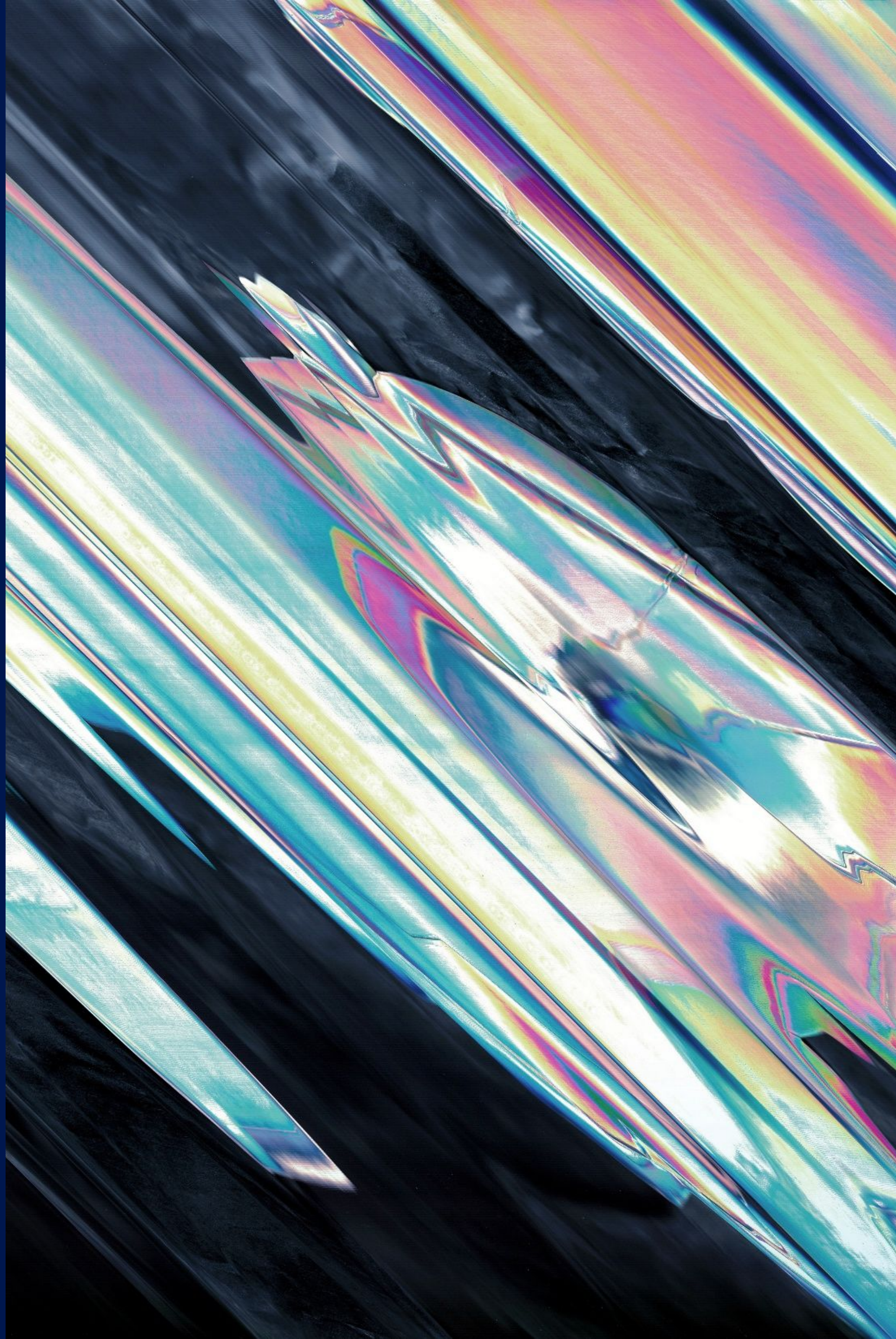
What about copy assignment operators?

When are objects deleted (invoking the destructor)?

Write your classes **properly**, not with the least amount of effort possible.

The Rule of Three

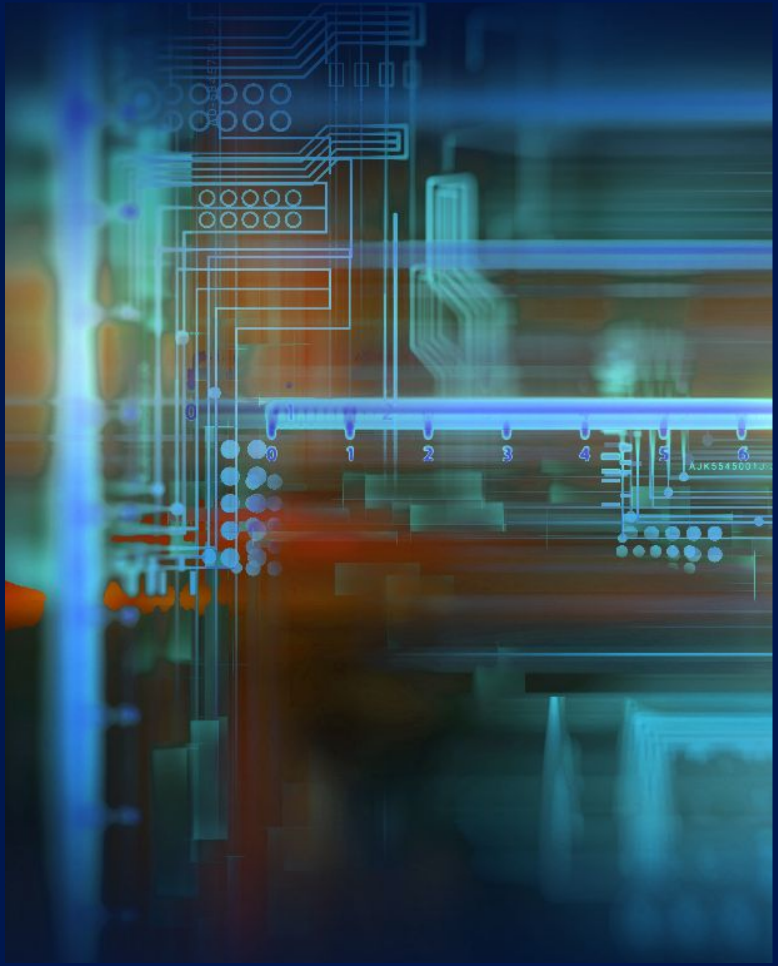
- If you **write one** of the Big Three, you should **write the other two**. For example…
- If you write a destructor to clean up dynamic memory…
- You need deep copy support (the other “Big Two”).
- **In reverse**: if you have deep copy support for dynamic memory (in one, maybe two places!)…
- You need to delete that dynamic memory!
- Modern C++ extends this to the Big Five, but we won’t worry about the other two functions (they’re for optimization and not strictly necessary).



Recap

- The Big Three are special **class member functions** to handle dynamic memory properly.
  - Copy Constructor, Copy Assignment Operator, and the Destructor
- We have to write them **for classes that use dynamic memory**.
- Implicitly-defined versions work without dynamic memory
  - Copy constructor: **member-to-member copy**
  - Copy assignment operator: **member-to-member copy**
  - Destructor: **empty, does nothing**!
- To copy dynamic memory properly, we use a **deep copy**.
- **The Rule of Three** says if we write one, we write all three.

Conclusion



Placeholder for the instructor’s welcome message. Video team, please insert the instructor’s video here.

Thank you for watching.