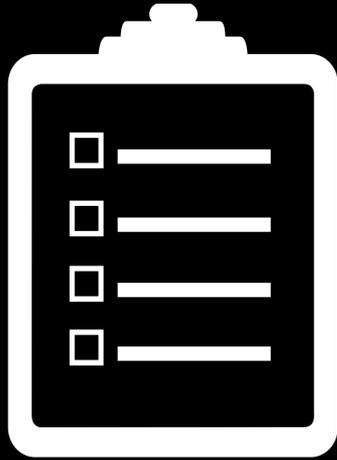


Constructors and Assignment

Ali Malik

malikali@stanford.edu

Game Plan



Recap

Const Correct Vector

Copy Constructor

Assignment Constructor

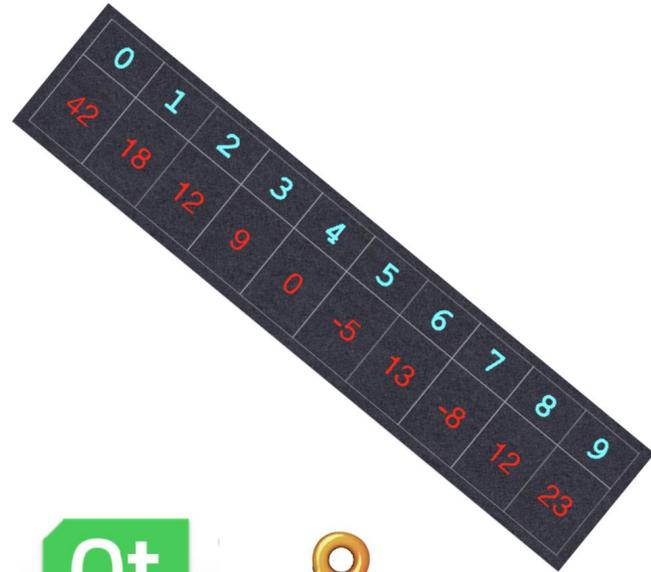
Rule of Three

Recap

Designing Vector

The Vector<int> Class: Implementation

- In order to demonstrate how useful (and necessary) dynamic memory is, let's implement a Vector that has the following properties:
- It can hold **ints** (unfortunately, it is beyond the scope of this class to create a Vector that can hold *any* type)
- It has useful Vector functions: **add()**, **insert()**, **get()**, **remove()**, **isEmpty()**, **size()**, << **overload**
- We can add as many elements as we would like
- It cleans up its own memory



Class Templates

The idea with class templates is the same.

A few more annoying nuances to watch out for.

```
template <typename ValueType>
class StrVector {

public:
    void push_back(const ValueType& elem);
    // rest of implementation
}
```



Tell compiler
ValueType is a
generic type.

Class Templates - Details

When we define a class template, we **only** use a .h file, and **do not** define member functions in a .cpp file.

Member functions are defined differently.

There's a bit of weird syntax for accessing **nested types**.

Class Templates - Details

Must announce that every method is templated

```
template <typename ValueType>
class Vector {

public:
    void push_back(const ValueType& elem);
    // rest of implementation
}

void Vector::push_back(const ValueType& val) {

}
```

Class Templates - Details

Must announce that every method is templated

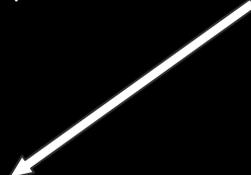
```
template <typename ValueType>
class Vector {

public:
    void push_back(const ValueType& elem);
    // rest of implementation
}

void Vector::push_back(const ValueType& val) {

}
```

Compiler error



Class Templates - Details

Must announce that every method is templated

```
template <typename ValueType>
class Vector {

public:
    void push_back(const ValueType& elem);
    // rest of implementation
}

void Vector::push_back(const ValueType& val) {

}
```

Class Templates - Details

Must announce that every method is templated

```
template <typename ValueType>
class Vector {

public:
    void push_back(const ValueType& elem);
    // rest of implementation
}

void Vector::push_back(const ValueType& val) {

}
```

Class Templates - Details

Must announce that every method is templated

```
template <typename ValueType>
class Vector {

public:
    void push_back(const ValueType& elem);
    // rest of implementation
}

void Vector<ValueType>::push_back(const ValueType& val) {

}
```

Class Templates - Details

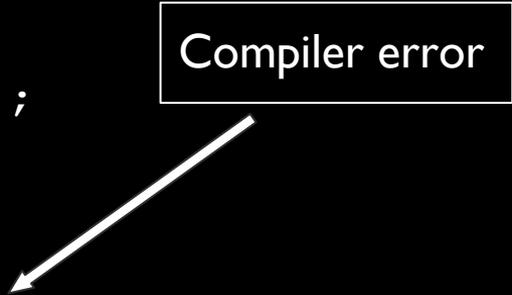
Must announce that every method is templated

```
template <typename ValueType>
class Vector {

public:
    void push_back(const ValueType& elem);
    // rest of implementation
}

void Vector<ValueType>::push_back(const ValueType& val) {

}
```



Compiler error

Class Templates - Details

Must announce that every method is templated

```
template <typename ValueType>
class Vector {

public:
    void push_back(const ValueType& elem);
    // rest of implementation
}

void Vector<ValueType>::push_back(const ValueType& val) {

}
```

Class Templates - Details

Must announce that every method is templated

```
template <typename ValueType>
class Vector {

public:
    void push_back(const ValueType& elem);
    // rest of implementation
}

template <typename ValueType>
void Vector<ValueType>::push_back(const ValueType& val) {

}
```

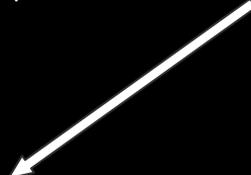
Class Templates - Details

Must announce that every method is templated

```
template <typename ValueType>  
class Vector {
```

```
public:  
    void push_back(const ValueType& elem);  
    // rest of implementation  
}
```

All good!



```
template <typename ValueType>  
void Vector<ValueType>::push_back(const ValueType& val) {  
  
}
```


Class Templates - Details II

Must use `typename` keyword for nested types:

```
template <typename ValueType>
Vector<ValueType>::iterator Vector<ValueType>::push_back(const
                                                               ValueType& val) {

}
}
```

```
❗ missing 'typename' prior to dependent type name 'Vector<ValueType>::iterator'
Vector<ValueType>::iterator Vector<ValueType>::begin() {
~~~~~
typename
/Users/alimalik/Desktop/Programs/C++/StringVector/strvector.h
```


Class Templates - Details II

Must use `typename` keyword for nested types:

```
template <typename ValueType>
typename Vector<ValueType>::iterator
    Vector<ValueType>::push_back(const ValueType& val) {

}
}
```

Const Correct Vector

Const Correctness

We need to write both `const` and `non-const` versions for some methods.

The method called depends on the `const`-ness of the object it is called on.

Examples:

- `operator[]`
- iterator `begin()` and `end()`

Refining Abstractions

Why Constructors?

Set up **initial state** of object:

- Make sure everything has a **sensible** starting value.
- Take information from user to **set up** object appropriately.

Initialisation vs Assignment

Initialisation vs Assignment

Initialisation:

Transforms an object's **initial junk** data into **valid** data.

Assignment:

Replaces **existing valid** data with other **valid** data.

Initialisation vs Assignment

Initialisation:

Defined by the **constructor** for a type.

Assignment:

Defined by the **assignment operator** for a type.

Initialisation vs Assignment

```
Vector<string> defV;  
// initialisation
```

```
Vector<string> fillV(10, "hello");  
// initialisation
```

```
Vector<string> copyV(defC);  
// initialisation
```

```
Vector<string> v = defV;  
// initialisation
```

```
v = fillV;  
// assignment
```

Constructors

Normal Constructor:

- What you are used to!

Copy Constructor

- **Initialise** an instance of a type to be a **copy** of another instance

Copy Assignment

- **Not** a constructor
- **Assign** an instance of a type to be a **copy** of another instance

Constructors

```
Vector<string> defV;  
// initialisation
```

```
Vector<string> fillV(10, "hello");  
// initialisation
```

```
Vector<string> copyV(defC);  
// initialisation
```

```
Vector<string> v = defV;  
// initialisation
```

```
v = fillV;  
// assignment
```

Constructors

```
Vector<string> defV;           // normal constructor  
// initialisation
```

```
Vector<string> fillV(10, "hello");  
// initialisation
```

```
Vector<string> copyV(defC);  
// initialisation
```

```
Vector<string> v = defV;  
// initialisation
```

```
v = fillV;  
// assignment
```

Constructors

```
Vector<string> defV;           // normal constructor  
// initialisation
```

```
Vector<string> fillV(10, "hello"); // normal constructor  
// initialisation
```

```
Vector<string> copyV(defC);  
// initialisation
```

```
Vector<string> v = defV;  
// initialisation
```

```
v = fillV;  
// assignment
```

Constructors

```
Vector<string> defV;           // normal constructor  
// initialisation
```

```
Vector<string> fillV(10, "hello"); // normal constructor  
// initialisation
```

```
Vector<string> copyV(defC);     // copy constructor  
// initialisation
```

```
Vector<string> v = defV;  
// initialisation
```

```
v = fillV;  
// assignment
```

Constructors

```
Vector<string> defV;           // normal constructor  
// initialisation
```

```
Vector<string> fillV(10, "hello"); // normal constructor  
// initialisation
```

```
Vector<string> copyV(defC);    // copy constructor  
// initialisation
```

```
Vector<string> v = defV;      // copy constructor  
// initialisation
```

```
v = fillV;  
// assignment
```


Constructors - Quick Note

If you don't define some of these constructors, the compiler will create default versions for you.

Constructors - Quick Note

If you don't define some of these constructors, the compiler will create default versions for you.



This might not always do what you want

Default Constructor

Default Constructor

Takes no arguments.

Used to initialise members to sensible starting values.

```
class MyClass {  
  
public:  
    MyClass() { // default constructor  
        privInt = 3;  
    }  
  
private:  
    int privInt;  
}
```

Default Constructor

Used as follows:

```
MyClass defC;           // calls default constructor
```

```
MyClass buggy ();     // DOESNT WORK!
```

Default Constructor

Used as follows:

```
MyClass defC; // calls default constructor
```

```
MyClass buggy(); // DOESNT WORK!
```

Default Constructor

Used as follows:

```
MyClass defC;           // calls default constructor  
  
// Treated as function declaration  
MyClass buggy();      // DOESNT WORK!
```



This is called C++'s Most Vexing Parse

Copy Constructor

Copy Constructor

Used to **initialise** an instance of a class from another existing instance.

Two ways it can be called:

```
// vector<string> v created earlier

// copy constructor called
vector<string> copyV(v);
vector<string> copyV2 = v;
```

Copy Constructor

Syntax is that of a constructor that takes a class object as its argument:

```
MyClass::MyClass (const MyClass& rhs) {  
  
    // implementation  
  
}
```

Copy Constructor

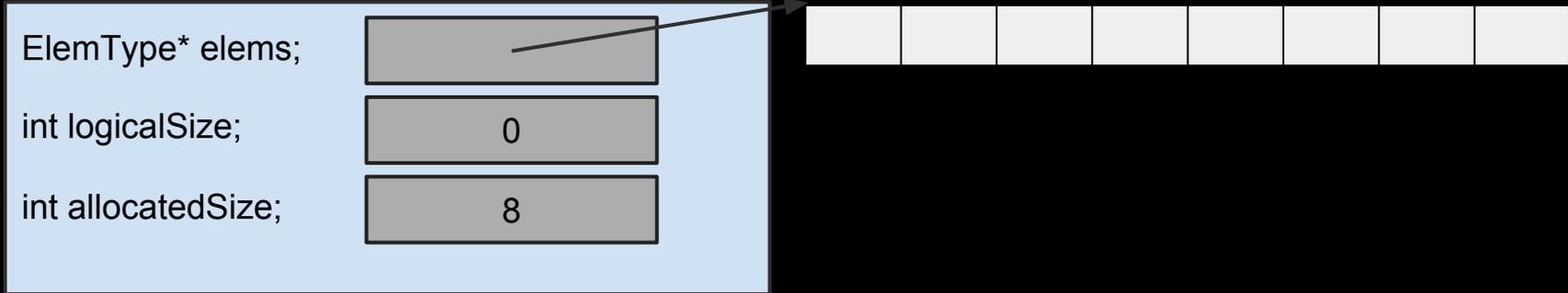
Let's write a copy constructor for our Vector class!

First idea:

- Just copy all the member variables over.
- We'll have the correct size and element pointer, so this works?
- This is what the default copy constructor does if you don't write one.

Copy Constructor

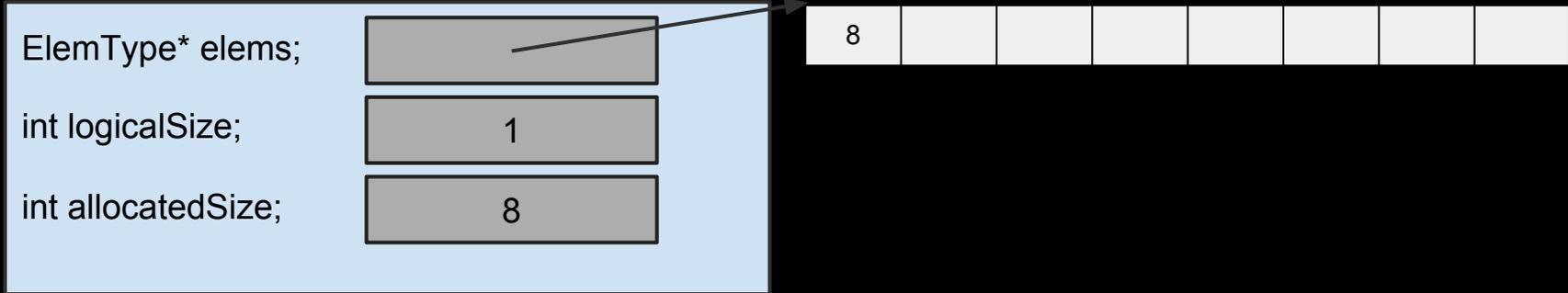
```
vector<int> a:
```



```
Vector<int> a;
```

Copy Constructor

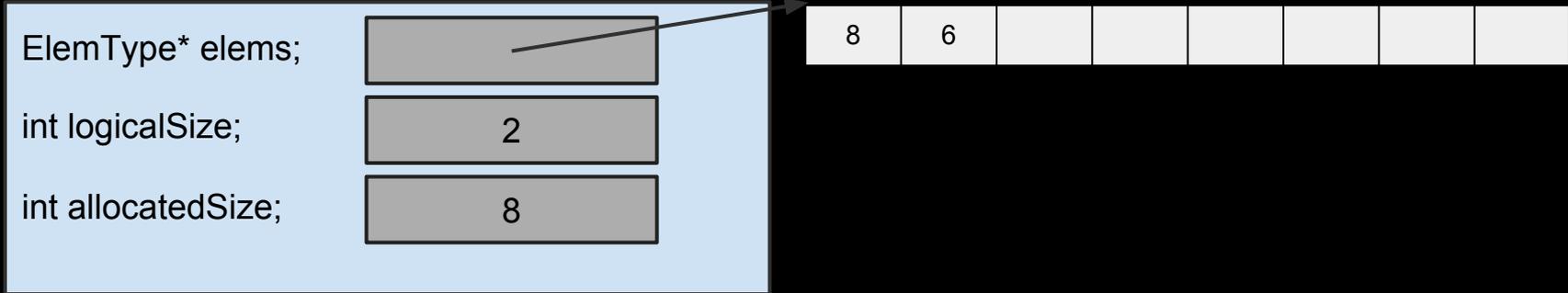
```
vector<int> a:
```



```
Vector<int> a;  
a.push_back(8);
```

Copy Constructor

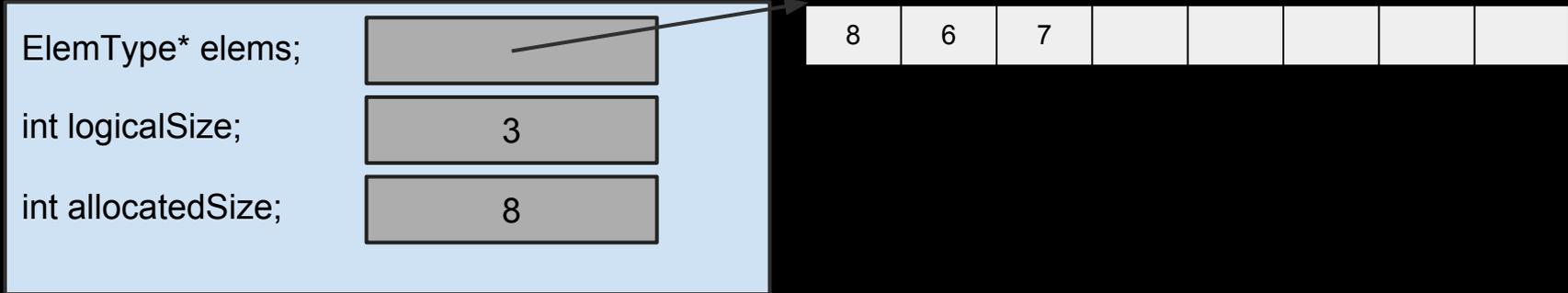
```
vector<int> a:
```



```
Vector<int> a;  
a.push_back(8);  
a.push_back(6);
```

Copy Constructor

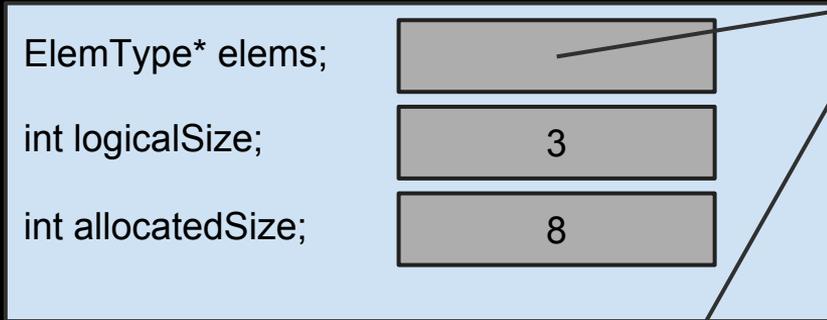
```
vector<int> a:
```



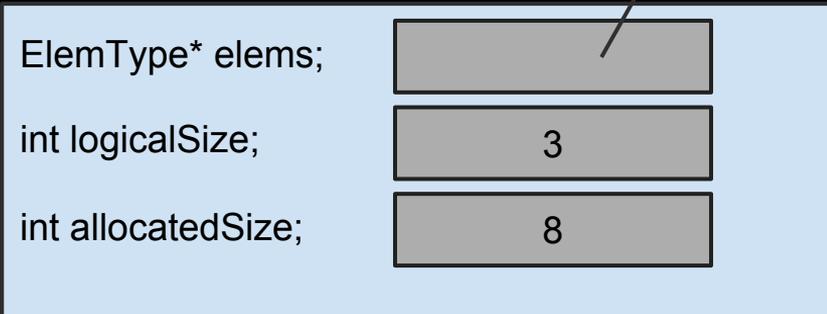
```
Vector<int> a;  
a.push_back(8);  
a.push_back(6);  
a.push_back(7);
```

Copy Constructor

`vector<int> a:`



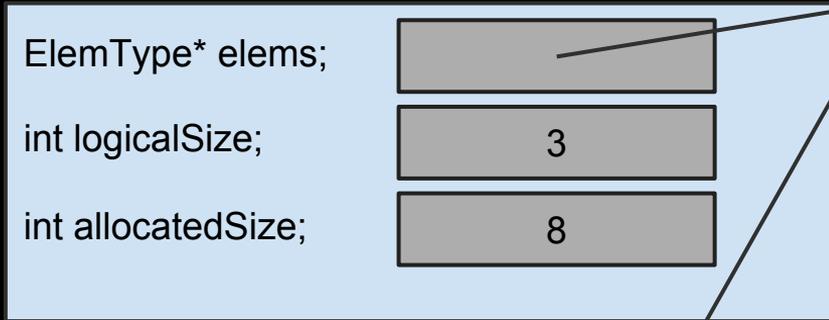
`vector<int> b:`



```
Vector<int> a;  
a.push_back(8);  
a.push_back(6);  
a.push_back(7);  
Vector<int> b = a;
```

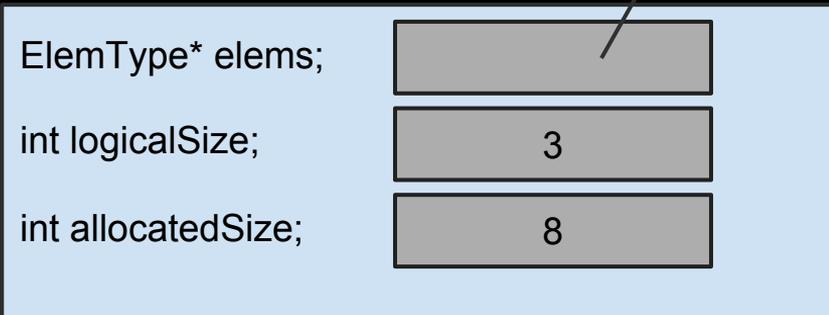
Copy Constructor

`vector<int> a:`



Changing the value of `b` also changed the value of `a`!

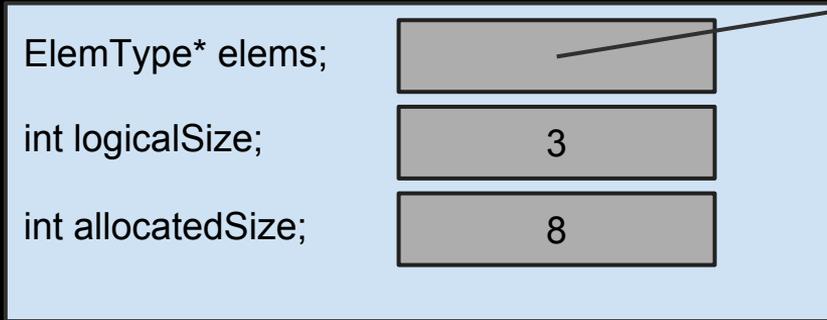
`vector<int> b:`



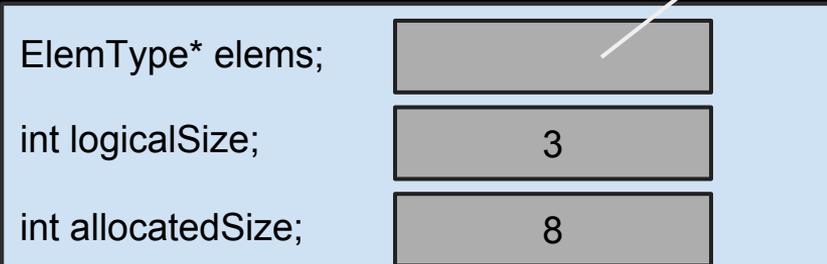
```
Vector<int> a;  
a.push_back(8);  
a.push_back(6);  
a.push_back(7);  
Vector<int> b = a;  
b[0] = 9;
```

Copy Constructor

`vector<int> a:`



`vector<int> b:`



```
Vector<int> a;  
a.push_back(8);  
a.push_back(6);  
a.push_back(7);  
Vector<int> b = a;  
b[0] = 9;
```

Copy Constructor - Deep copy

Lesson:

If you have **pointer** variables, you should always define a copy constructor.

Copy Constructor

Let's add this to our vector class:

```
MyVector.pro
```

Copy Assignment

Copy Assignment

Takes **already initialised** object and gives it new values.

```
// vector<string> v, v2 created earlier

// copy constructor called
vector<string> copyV = v;

// copy assignment
copyV = v2;
```

Copy Assignment

Works by overloading the = operator.

Syntax is exact same as any other operator overload.

```
class MyClass {  
  
public:  
    MyClass& operator=(const MyClass& rhs) {  
  
    }  
private:  
    int privInt;  
}
```

Copy Assignment

Slightly more involved than copy constructor because object already contains valid data!

We need to watch out for:

- Catching memory leaks
- Handling self assignment
- Understanding the return value

The Rule of Three

The Rule of Three

If you implement a copy constructor, assignment operator, or destructor, you should implement the others, as well

Next Time

RAII

