CS106L Lecture 13:
Special Member Functions ⭐

Thomas Poimenidis, Rachel Fernandez

Attendance

https://forms.gle/zog29AvfwfSNSBJo7

Today's Agenda

1. Recap
2. Special Member Functions
    - An overview
    - Copy and copy assignment
    - `delete`
    - Move and move assignment

Today's Agenda

1. **Recap**
2. Special Member Functions
   - An overview
   - Copy and copy assignment
   - `delete`
   - Move and move assignment

Non-member overloading

Non-member Operator Overloading

```
bool operator< (const StudentID& lhs, const StudentID& rhs);
```

Member Operator Overloading

```
bool StudentID::operator< (const StudentID& rhs) const {...}
```

Clarification

.cpp file

```
#include StanfordID.h

std::string StanfordID::getIdNumber() {
    return idNumber;
}


bool StanfordID::operator<(const StanfirdID& other) const {
    return idNumber < other.getIdNumber(); ✅
}
```

Clarification

.cpp file

```
#include StanfordID.h

std::string StanfordID::getIdNumber() {
    return idNumber;
}


bool StanfordID::operator<(const StanfirdID& other) const {
    return idNumber < other.idNumber; ✅
}
```

Clarification

.cpp file

```
#include StanfordID.h

// defined within the StanfordID class
std::string StanfordID::getIdNumber() {
    return idNumber;
}
. . .

bool operator<(const StanfirdID& lhs, const StanfirdID& rhs) const
{
    return lhs.idNumber < rhs.idNumber; ❌
}
```

Clarification

.cpp file

```
#include StanfordID.h

// defined within the StanfordID class
std::string StanfordID::getIdNumber() {
    return idNumber;
}
. . .

bool operator<(const StanfirdID& lhs, const StanfirdID& rhs) const
{
    return lhs.getIdNumber() < rhs.getIdNumber(); ✅
}
```

Hello `friend`!

Non-member Operator Overloading

```
bool operator< (const StudentID& lhs, const StudentID& rhs);
```

The **`friend`** keyword allows non-member functions or classes to access private information in another class!

How do you use `friend`?

In the header of the target class you declare the operator overload function as a `friend`

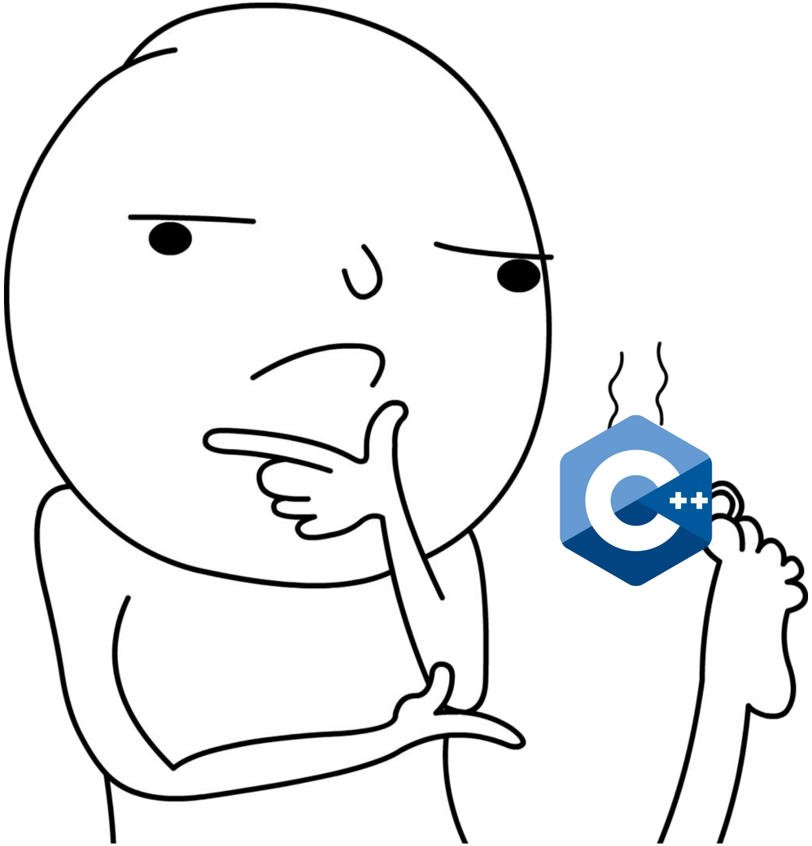
Notice: If **`StanfordID`** didn't have a **`getIdNumber()`** method, you'd have to add **`friend`** to access **`idNumber`** directly

So Many Operators!!!

- There are many operators that you can define in C++ like we saw

```
+ - * / % ^ & | ~ ! , = < > <= >=
++ -- << >> == != && || += -= *=
/= %= ^= &= |= <<= >>= [] () ->
->* new new[] delete delete[]
```

What questions do we have?



Today's Agenda

1. Recap
2. **Special Member Functions**
   - **An overview**
   - Copy and copy assignment
   - `delete`
   - Move and move assignment

You may remember

Classes have

1. Constructor
2. Destructor
3.
4.

🤯 Surprise 🤯, these are called **Special Member Functions (SMFs)**

A **constructor** is called every time a new instance of the class is created, and the **destructor** is called when it goes out of scope

The Special 6 SMFs

These functions are generated only when they're called (and before any are explicitly defined by you):

- Default constructor: `T()`
- Destructor: `~T()`
- Copy constructor: `T(const T&)`
- Copy assignment operator: `T& operator=(const T&)`
- Move constructor: `T(T&&)`
- Move assignment operator: `T& operator=(T&&)`

Let's look at Widget :)

```
class Widget {
 public:
  Widget();                                   // default constructor
  Widget (const Widget& w);                   // copy constructor
  Widget& operator = (const Widget& w);       // copy assignment operator
  ~Widget();                                  // destructor
  Widget (Widget&& rhs);                      // move constructor
  Widget& operator = (Widget&& rhs);          // move assignment operator
}
```

There are 6 special member functions!

```
class Widget {
  public:
    Widget();                                   // default constructor
    Widget (const Widget& w);                   // copy constructor
    Widget& operator = (const Widget& w);       // copy assignment operator
    ~Widget();                                  // destructor
    Widget (Widget&& rhs);                      // move constructor
    Widget& operator = (Widget&& rhs);          // mov
}
```

Takes no parameters and creates a new object

There are 6 special member functions!

```
class Widget {
  public:
    Widget();                                 // default constructor
    Widget (const Widget& w);                 // copy constructor
    Widget& operator = (const Widget& w);    // copy assignment operator
    ~Widget();                                // destructor
    Widget (Widget&& rhs);                    // move constructor
    Widget& operator = (Widget&& rhs);        // mov
}
```

Creates a new object as a member-wise copy of another

When is the copy constructor invoked?

```
Widget widgetOne;
Widget widgetTwo = widgetOne;  // Copy constructor is called
```

There are 6 special member functions!

```
class Widget {
  public:
    Widget();                                 // default constructor
    Widget (const Widget& w);                 // copy constructor
    Widget& operator = (const Widget& w);     // copy assignment operator
    ~Widget();                                // destructor
    Widget (Widget&& rhs);                    // move constructor
    Widget& operator = (Widget&& rhs);        // mov
}
```

Assigns an already existing object to another

When is the copy assignment operator invoked?

```
Widget widgetOne;
Widget widgetTwo;
widgetOne = widgetTwo
```

Note that here both objects are constructed before the use of the = operator

Copy Constructor vs Assignment Operator

Copy Constructor Invocation

```
Widget widgetOne;
Widget widgetTwo = widgetOne;
```

Copy Assignment Operator Invocation

```
Widget widgetOne;
Widget widgetTwo;
widgetOne = widgetTwo
```

There are 6 special member functions!

```
class Widget {
  public:
    Widget();                                    // default constructor
    Widget (const Widget& w);                    // copy constructor
    Widget& operator = (const Widget& w);  // copy assignment operator
    ~Widget();                                   // destructor
    Widget (Widget&& rhs);                       // move constructor
    Widget& operator = (Widget&& rhs);     // mov
}
```

Called when the object goes out of scope

There are 6 special member functions!

```
class Widget {
  public:
    Widget();                                  // default constructor
    Widget (const Widget& w);                  // copy constructor
    Widget& operator = (const Widget& w);      // copy assignment operator
    ~Widget();                                 // destructor
    Widget (Widget&& rhs);                     // move constructor
    Widget& operator = (Widget&& rhs);         // move assignment operator
}
```

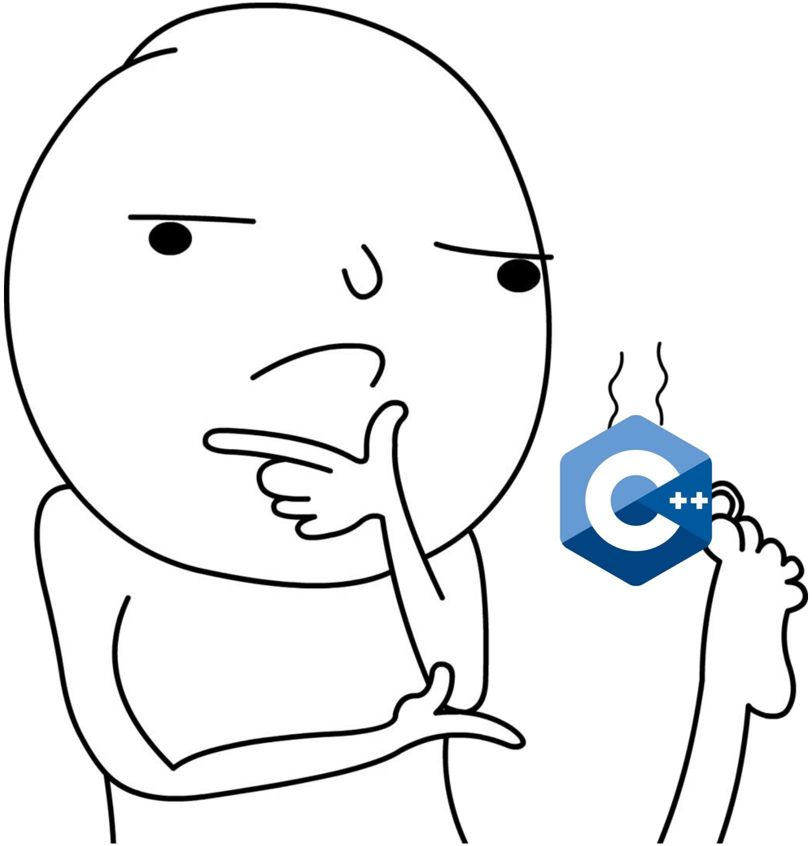
We have an entire lecture on these – not the focus of today

There are 6 special member functions!

```
class Widget {
  public:
    Widget();
    Widget (const Widget& w);
    Widget& operator = (const Widget& w);
    ~Widget();
    Widget (Widget&& rhs);                    // move constructor
    Widget& operator = (Widget&& rhs);        // move assignment operator
}
```

We don’t have to write out any of these! They all have default versions that are generated automatically!

What questions do we have?



Today's Agenda

1. Recap
2. **Special Member Functions**
   - An overview
   - **Copy and copy assignment**
   - `delete`
   - Move and move assignment

Review: initialization

Remember our Vector from lecture 8?

```
template <typename T>
Vector<T>::Vector()
{
   _size = 0;
   _capacity = 4;
   _data = new T[_capacity];
}
```

When we create a constructor, we need to initialize all of our member variables.

Review: initialization

```
template <typename T>
Vector<T>::Vector()
{
   _size = 0;
   _capacity = 4;
   _data = new T[_capacity];
}
```

However, initializing them to be the default value and then reassigning is inefficient!

```
template <typename T>
Vector<T>::Vector()
{
  _size = 0;
  _capacity = 4;
  _data = new T[_capacity];
}
```

There are two steps happening here

Step 1

```
template <typename T>
Vector<T>::Vector()
{
  _size = 0;
  _capacity = 4;
  _data = new T[_capacity
}
```

There are two steps happening here: the first is that `_size`, `_capacity`, and `_data` may have been default initialized

Step 2

```
template <typename T>
Vector<T>::Vector()
{
   _size = 0;
   _capacity = 4;
   _data = new T[_capacity];
}
```

Then the assignment to the variables, which effectively doubles the work.

Member initialization Lists

```
template <typename T>
Vector<T>::Vector() : _size(0), _capacity(4), _data(new
T[_capacity]) { }
```

We can use **initializer lists** to declare and initialize them with desired values at once!

Initializer Lists

- It’s quicker and more efficient to directly construct member variables with intended values
- What if the variable is a non-assignable type?
- Can be used for any constructor, even non-default ones with parameters!

```
template <typename T>
Vector<T>::Vector() : _size(0), _capacity(4), _data(new
T[_capacity]) { }
```

What if the variable is a non-assignable type?

```
template <typename T>
class MyClass {
    const int _constant;
    int& _reference;

public:
    // Only way to initialize const and reference members
    MyClass(int value, int& ref) : _constant(value),
_reference(ref) { }
};
```

- This code ***only*** works with initializer lists
- Why? 🤔

Why should we override SMFs?

The compiler gives them to us for free…..?

a. By default, the copy constructor will create copies of each member variable

This is **member-wise** copying!

Is this always good enough?

Consider Pointers

If your variable is a pointer, a memberwise copy will point to the same allocated data, not a fresh copy!

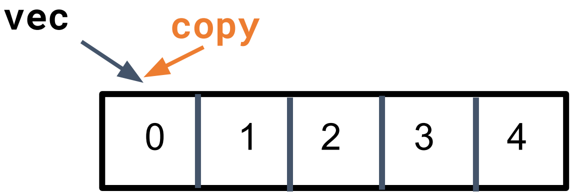
```
template <typename T>
Vector<T>::Vector<T>(const Vector::Vector<T>& other) :
_size(other._size), _capacity(other._capacity),
_data(other._data) { }
```

These pointers will point at the same underlying array!

Consider Pointers

If your variable is a pointer, a memberwise copy will point to the same allocated data, not a fresh copy!

```
template <typename T>
Vector<T>::Vector<T>(const Vector::Vector<T>& other) :
_size(other._size), _capacity(other._capacity),
_data(other._data) { }
```

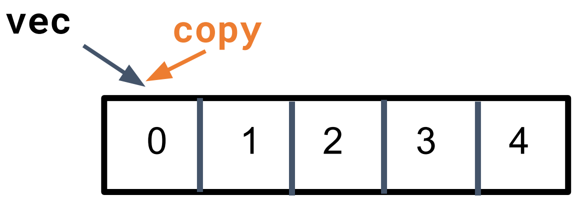


These pointers will point at the same underlying array!

Consider Pointers

If your variable is a pointer, a memberwise copy will point to the same allocated data, not a fresh copy!

```
template <typename T>
Vector<T>::Vector<T>(const Vector::Vector<T>& other) :
_size(other._size), _capacity(other._capacity),
_data(other._data) { }
```



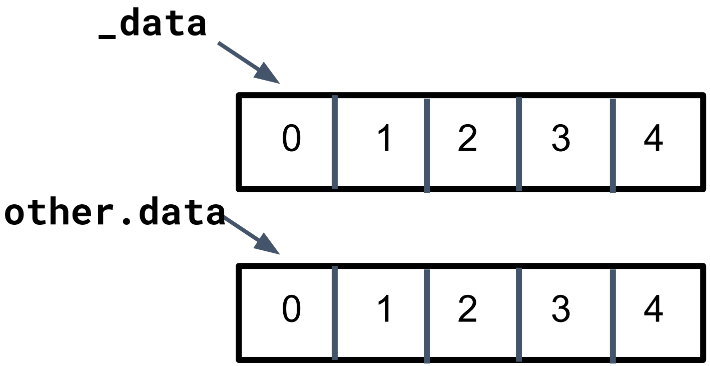
This is problematic because anything done to one pointer affects the other

Copying isn't always so simple!

- Many times, you will want to create a copy that does more than just copies the member variables.
- Deep copy: an object that is a complete, **independent** copy of the original
- In these cases, you'd want to override the default special member functions with your own implementation!
- Declare them in the header and write their implementation in the `.cpp`, like any function!

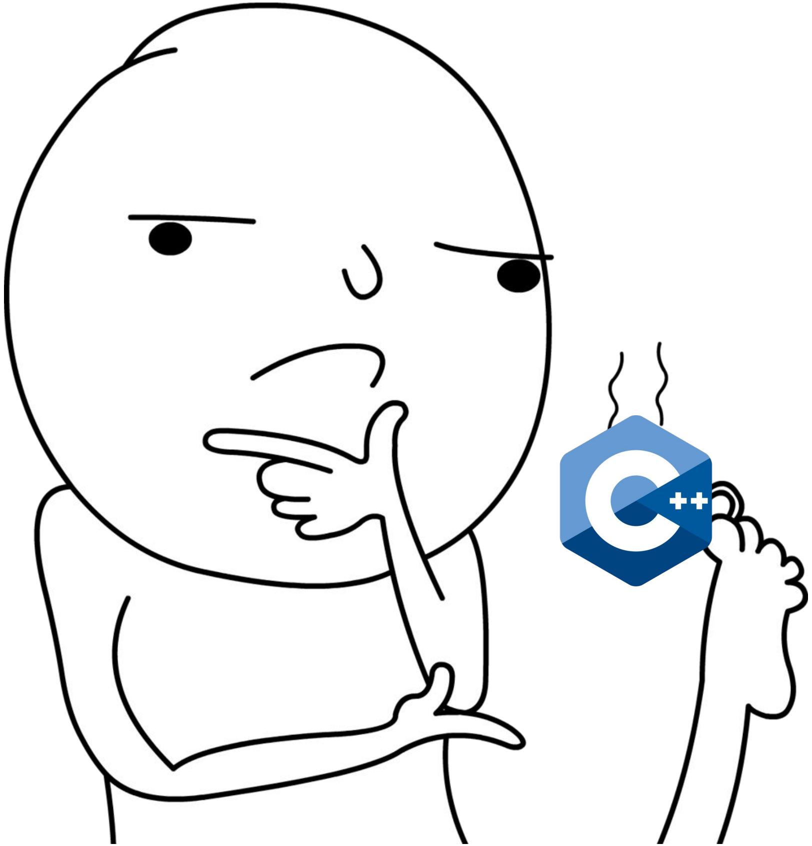
Fixing the pointer issue

```
Vector<T>::Vector(const Vector<T>& other)
    : _size(other._size), _capacity(other._capacity), _data(new
T[other._capacity]) {
    for (size_t i = 0; i < _size; ++i) {
        _data[i] = other._data[i];
    }
}
```



Now we have a “deep” copy of the data in our Vector.

What questions do we have?



Today's Agenda

1. Recap
2. **Special Member Functions**
    - An overview
    - Copy and copy assignment
    - **`delete`**
    - Move and move assignment

How do you prevent copies?

Let’s say you have a class that handles all of your passwords:

```
class PasswordManager {
  public:
    PasswordManager();
    ~PasswordManager();
    // other methods ...
    PasswordManager(const PasswordManager& rhs);
    PasswordManager& operator = (const PasswordManager& rhs);

  private:
    // other important members ...
}
```

We can delete special member functions

Setting a special member function to **delete** removes its functionality!

```
class PasswordManager {
  public:
    PasswordManager();
    PasswordManager(const PasswordManager& pm);
    ~PasswordManager();
    // other methods ...
    PasswordManager(const PasswordManager& rhs) = delete;
    PasswordManager& operator = (const PasswordManager& rhs) = delete;

  private:
    // other important members ...
}
```

We can delete special member functions

Setting a special member function to **delete** removes its functionality!

```
class PasswordManager {
  public:
    PasswordManager();
    PasswordManager(const PasswordManager& pm);
    ~PasswordManager();
    // other methods ...
    PasswordManager(const PasswordManager& rhs) = delete;
    PasswordManager& operator = (const PasswordManager& rhs) = delete;

  private:
    // other important members ...
}
```

Now copying isn't a possible operation!

Why?

We can selectively allow functionality of special member functions!

- This has lots of uses – what if we only want one copy of an instance to be allowed?
- This is how classes like `std::unique_ptr` wor

You may see this in `cppreference` which specifies this!

The class satisfies the requirements of *MoveConstructible* and *MoveAssignable*, but of neither *CopyConstructible* nor *CopyAssignable*.

Philosophy time

Rule of Zero

If the default SMFs work, **don’t define your own**!

We should only define new ones when the default ones generated by the compiler won't work.

- This usually happens when we work with dynamically allocated memory, like pointers to things on the heap!

Rule of Zero

If you don't need a constructor or a destructor or copy assignment etc. Then simply don't use it!

If your class relies on objects/classes that already have these SMFs implemented, then there's no need to reimplement this logic!

```
class a_string_with_an_id() {
    public:
        /// getter and setter methods for our private variables
    private:
        int id;
        std::string str;
}
a_string_with_an_id object;
```

Our class **a_string_with_an_id** has self managing variables.

Rule of Zero

If you don't need a constructor or a destructor or copy assignment etc. Then simply don't use it!

If your class relies on objects/classes that already have these SMFs implemented, then there's no need to reimplement this logic!

```
class a_string_with_an_id() {
    public:
        /// getter and setter methods for our private variables
    private:
        int id;
        std::string str;
}
a_string_with_an_id object;
```

`std::string` ***already*** has copy constructor, copy assignment, move constructor, and move assignment!

Rule of Three

If you need a custom destructor, then you also probably ***<u>need</u>*** to define a copy constructor and a copy assignment operator for your class

Why is this the case?

If you use a destructor, that often means that you are manually dealing with dynamic memory allocation/are generally just handling your own memory.

If this is the case:

The compiler will not be able to automatically generate these for you, because of the manual memory management.

Recap

The four special member functions discussed so far:

- **Default Constructor**
  - Object created with no parameters, no member variables instantiated
- **Copy Constructor**
  - Object created as a copy of existing object (member variable-wise)
- **Copy Assignment Operator**
  - Existing object replaced as a copy of another existing object.
- **Destructor**
  - Object destroyed when it is out of scope.

Pop Quiz

```
vector<int> func(vector<int> vec0) {
    vector<int> vec1;
    vector<int> vec2(3);
    vector<int> vec3{3};
    vector<int> vec4();
    vector<int> vec5(vec2);
    vector<int> vec6{};
    vector<int> vec7{static_cast<int>(vec2.size() + vec6.size())};
    vector<int> vec8 = vec2;
    vec8 = vec2;
    return vec8;
}
```

What type of operation or function is each of these lines?

Pop Quiz

```
vector<int> func(vector<int> vec0) {
    vector<int> vec1;
    vector<int> vec2(3);
    vector<int> vec3{3};
    vector<int> vec4();
    vector<int> vec5(vec2);
    vector<int> vec6{};
    vector<int> vec7{static_cast<int>(vec2.size() + vec6.size())};
    vector<int> vec8 = vec2;
    vec8 = vec2;
    return vec8;
}
```

Default Constructor

Pop Quiz

```
vector<int> func(vector<int> vec0) {
    vector<int> vec1;
    vector<int> vec2(3);
    vector<int> vec3{3};
    vector<int> vec4();
    vector<int> vec5(vec2);
    vector<int> vec6{};
    vector<int> vec7{static_cast<int>(vec2.size() + vec6.size())};
    vector<int> vec8 = vec2;
    vec8 = vec2;
    return vec8;
}
```

Custom constructor,
not SMF

Pop Quiz

```
vector<int> func(vector<int> vec0) {
    vector<int> vec1;
    vector<int> vec2(3);
    vector<int> vec3{3};
    vector<int> vec4();
    vector<int> vec5(vec2);
    vector<int> vec6{};
    vector<int> vec7{static_cast<int>(vec2.size() + vec6.size())};
    vector<int> vec8 = vec2;
    vec8 = vec2;
    return vec8;
}
```

Uniform initialization, not an SMF

Pop Quiz

```
vector<int> func(vector<int> vec0) {
    vector<int> vec1;
    vector<int> vec2(3);
    vector<int> vec3{3};
    vector<int> vec4();
    vector<int> vec5(vec2);
    vector<int> vec6{};
    vector<int> vec7{static_cast<int>(vec2.size() + vec6.size())};
    vector<int> vec8 = vec2;
    vec8 = vec2;
    return vec8;
}
```

Tricky, this is a function definition

Pop Quiz

```
vector<int> func(vector<int> vec0) {
    vector<int> vec1;
    vector<int> vec2(3);
    vector<int> vec3{3};
    vector<int> vec4();
    vector<int> vec5(vec2);
    vector<int> vec6{};
    vector<int> vec7{static_cast<int>(vec2.size() + vec6.size())};
    vector<int> vec8 = vec2;
    vec8 = vec2;
    return vec8;
}
```

Copy Constructor

Pop Quiz

```
vector<int> func(vector<int> vec0) {
    vector<int> vec1;
    vector<int> vec2(3);
    vector<int> vec3{3};
    vector<int> vec4();
    vector<int> vec5(vec2);
    vector<int> vec6{};
    vector<int> vec7{static_cast<int>(vec2.size() + vec6.size())};
    vector<int> vec8 = vec2;
    vec8 = vec2;
    return vec8;
}
```

Initializer list is empty – empty vector via list initialization

Pop Quiz

```
vector<int> func(vector<int> vec0) {
    vector<int> vec1;
    vector<int> vec2(3);
    vector<int> vec3{3};
    vector<int> vec4();
    vector<int> vec5(vec2);
    vector<int> vec6{};
    vector<int> vec7{static_cast<int>(vec2.size() + vec6.size())};
    vector<int> vec8 = vec2;
    vec8 = vec2;
    return vec8;
}
```

List initialization

Pop Quiz

```
vector<int> func(vector<int> vec0) {
    vector<int> vec1;
    vector<int> vec2(3);
    vector<int> vec3{3};
    vector<int> vec4();
    vector<int> vec5(vec2);
    vector<int> vec6{};
    vector<int> vec7{static_cast<int>(vec2.size() + vec6.size())};
    vector<int> vec8 = vec2;
    vec8 = vec2;
    return vec8;
}
```

Copy constructor

Pop Quiz

```
vector<int> func(vector<int> vec0) {
    vector<int> vec1;
    vector<int> vec2(3);
    vector<int> vec3{3};
    vector<int> vec4();
    vector<int> vec5(vec2);
    vector<int> vec6{};
    vector<int> vec7{static_cast<int>(vec2.size() + vec6.size())};
    vector<int> vec8 = vec2;
    vec8 = vec2;
    return vec8;
}
```

Copy assignment operator

Pop Quiz

```
vector<int> func(vector<int> vec0) {
    vector<int> vec1;
    vector<int> vec2(3);
    vector<int> vec3{3};
    vector<int> vec4();
    vector<int> vec5(vec2);
    vector<int> vec6{};
    vector<int> vec7{static_cast<int>(vec2.size() + vec6.size())};
    vector<int> vec8 = vec2;
    vec8 = vec2;
    return vec8;
}
```

Copy constructor

Pop Quiz

```
vector<int> func(vector<int> vec0) {
    vector<int> vec1;
    vector<int> vec2(3);
    vector<int> vec3{3};
    vector<int> vec4();
    vector<int> vec5(vec2);
    vector<int> vec6{};
    vector<int> vec7{static_cast<int>(vec2.size() + vec6.size())};
    vector<int> vec8 = vec2;
    vec8 = vec2;
    return vec8;
}
```

Tricky bonus one:

Copy constructor

Today's Agenda

1. Recap
2. **Special Member Functions**
   - An overview
   - Copy and copy assignment
   - `delete`
   - **Move and move assignment**

Is copying enough?

We’ve learned about the default constructor, destructor, and the copy constructor and assignment operator.

- We can create an object, get rid of it, and copy its values to another object!
- Is this ever insufficient?

This can be wasteful

These functions are generated only when they're called

(and before any are explicitly defined by you):

```
class Widget {
  public:
    Widget();                                   //
    Widget (const Widget& w);                   //
    Widget& operator = (const Widget& w);       //                      r
    ~Widget();                                  // destructor
    Widget (Widget&& rhs);                      // move constructor
    Widget& operator = (Widget&& rhs);          // move assignment operator
}
```

Let's motivate move semantics

This can be wasteful

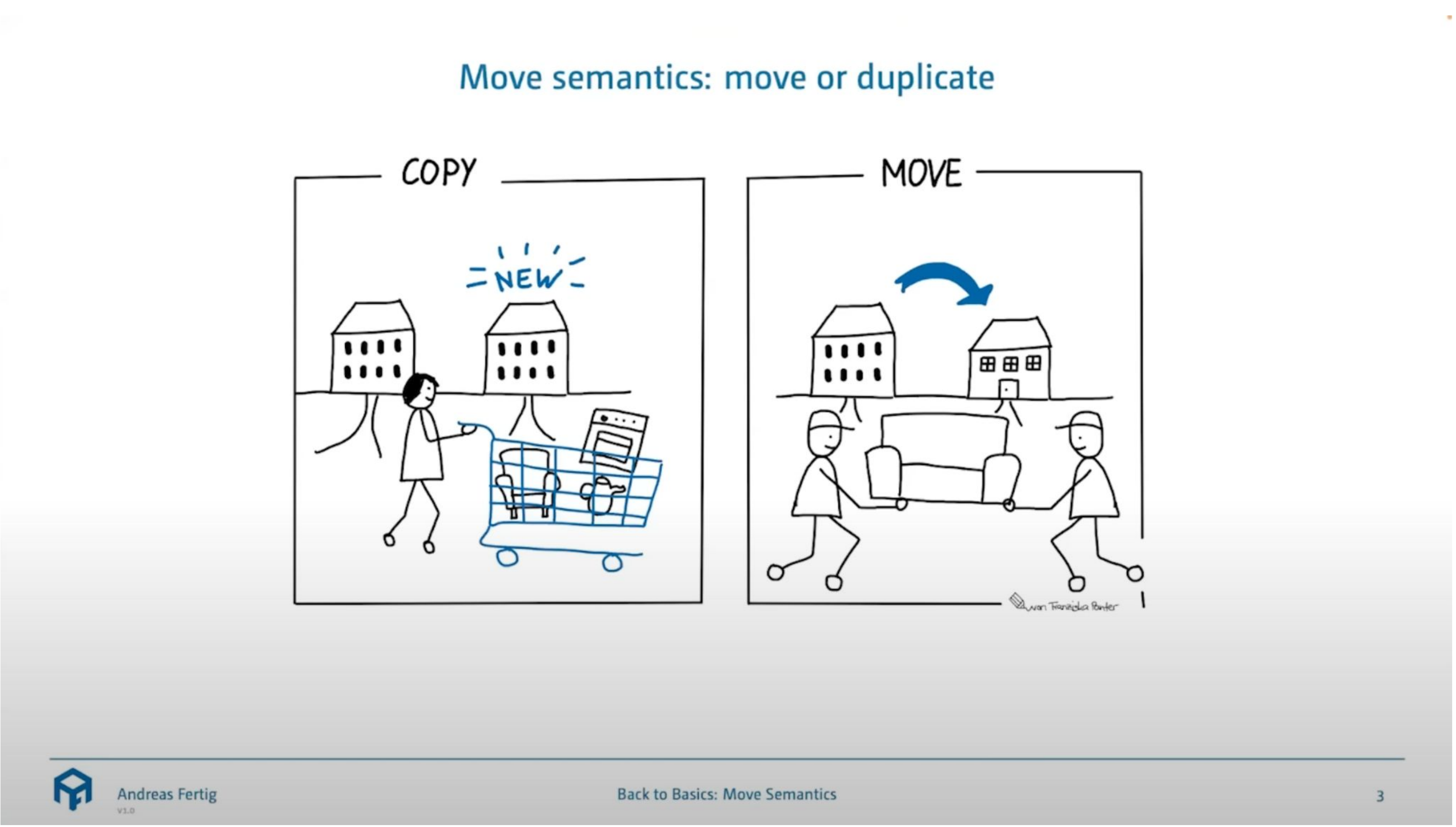
Let's say we had to copy our current StringTable into another, whose reference is given to us, and we have no use for our StringTable afterwards.

```
class StringTable {
 public:
   StringTable() {}
   StringTable(const StringTable& st) {}
   // functions for insertion, erasure, lookup, etc.
   // but no move/dtor functionality
   // ...

 private:
   std::map<int, std::string> values;
}
```

The copy constructor will copy every value in the values map one by one! Very slowly!

A good way to prime move semantics



I really like this way of thinking about move semantics:

Watch the full video [here]()