Lecture 13: Special Member Functions

CS 106L, Winter ‘21
CS 106B covers the **barebones** of C++ classes

we’ll be covering the rest

template classes • const correctness • operator overloading
special member functions • move semantics • RAII
CS 106B covers the **barebones** of C++ classes

we’ll be covering the rest

**template classes** • **const correctness** • **operator overloading**

**special member functions** • **move semantics** • **RAII**
Key questions we will answer today

- What are special member functions? When are they called?
- When should we declare a special member function?
- When should we not declare a special member function?
Agenda

- More info about mycollection::vector implementation
- Intro to special member functions
- Member initializer lists
- Why aren’t the default functions always sufficient?
- Copy assignment and construction
- Delete
- Rule of three/zero
Live Code Demo:
mycollection::vector implementation
Intro to special member functions
Special member functions are (usually) automatically generated by the compiler.
Special Member Functions

- Default construction: object created with no parameters.
- Copy construction: object is created as a copy of an existing object.
- Copy assignment: existing object replaced as a copy of another existing object.
- Destruction: object destroyed when it is out of scope.
🤔 Questions? 🤔
What Special Member Function is Called on Each Line?

vector<int> function(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{vec3 + vec4};
  vector<int> vec8 = vec4;
  vec8 = vec2;
  return vec8;
}

- Default construction: object created with no parameters.
- Copy construction: object is created as a copy of an existing object.
- Copy assignment: existing object replaced as a copy of another existing object.
- Destruction: object destroyed when it is out of scope.
Copy constructor (passing by value)

```cpp
vector<int> function(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{vec3 + vec4};
    vector<int> vec8 = vec4;
    vec8 = vec2;
    return vec8;
}
```

- Default construction: object created with no parameters.
- Copy construction: object is created as a copy of an existing object.
- Copy assignment: existing object replaced as a copy of another existing object.
- Destruction: object destroyed when it is out of scope.
Default constructor creates empty vector

vector<int> function(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{vec3 + vec4};
    vector<int> vec8 = vec4;
    vec8 = vec2;
    return vec8;
}
Not a special member function - creates a vector \{0, 0, 0\}

```cpp
vector<int> function(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{vec3 + vec4};
    vector<int> vec8 = vec4;
    vec8 = vec2;
    return vec8;
}
```

- Default construction: object created with no parameters.
- Copy construction: object is created as a copy of an existing object.
- Copy assignment: existing object replaced as a copy of another existing object.
- Destruction: object destroyed when it is out of scope.
vector<int> function(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{vec3 + vec4};
    vector<int> vec8 = vec4;
    vec8 = vec2;
    return vec8;
}
vector<int> function(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{vec3 + vec4};
    vector<int> vec8 = vec4;
    vec8 = vec2;
    return vec8;
}
Copy constructor - vec created as a copy of another one

```cpp
vector<int> function(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{vec3 + vec4};
    vector<int> vec8 = vec4;
    vec8 = vec2;
    return vec8;
}
```

- Default construction: object created with no parameters.
- Copy construction: object is created as a copy of an existing object.
- Copy assignment: existing object replaced as a copy of another existing object.
- Destruction: object destroyed when it is out of scope.
Also the default constructor!

```cpp
vector<int> function(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{vec3 + vec4};
    vector<int> vec8 = vec4;
    vec8 = vec2;
    return vec8;
}
```

- Default construction: object created with no parameters.
- Copy construction: object is created as a copy of an existing object.
- Copy assignment: existing object replaced as a copy of another existing object.
- Destruction: object destroyed when it is out of scope.
vector<int> function(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{vec3 + vec4};
    vector<int> vec8 = vec4;
    vec8 = vec2;
    return vec8;
}
Copy constructor - vec8 is newly constructed

vector<int> function(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{vec3 + vec4};
    vector<int> vec8 = vec4;
    vec8 = vec2;
    return vec8;
}
Copy assignment - vec8 is an existing object

- Default construction: object created with no parameters.
- Copy construction: object is created as a copy of an existing object.
- Copy assignment: existing object replaced as a copy of another existing object.
- Destruction: object destroyed when it is out of scope.

```cpp
copy_assignment - vec8 is an existing object

vector<int> function(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{vec3 + vec4};
    vector<int> vec8 = vec4;
    vec8 = vec2;
    return vec8;
}
```
vector<int> function(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{vec3 + vec4};
    vector<int> vec8 = vec4;
    vec8 = vec2;
    return vec8;
}
Destructors on all values (except return value) are called

```cpp
vector<int> function(vector<int> vec0) {
    vector<int> vec1;
    vector<int> vec2(3);
    vector<int> vec3{3};
    vector<int> vec4();
    vector<int> vec5(vec2);
    vector<int> vec6{vec2 + vec3};
    vector<int> vec7{vec3 + vec4};
    vector<int> vec8 = vec4;
    vec8 = vec2;
    return vec8;
}
```

- Default construction: object created with no parameters.
- Copy construction: object is created as a copy of an existing object.
- Copy assignment: existing object replaced as a copy of another existing object.
- Destruction: object destroyed when it is out of scope.
🤔 Questions? 🤔
Member initializer lists
How we’re used to writing constructors

```cpp
template <typename T>
vector<T>::vector<T>() {
    _size = 0;
    _capacity = kInitialSize;
    _elems = new T[kInitialSize];
}
```

Members are first default constructed

Then each member is reassigned
Member initializer lists!

```cpp
template <typename T>
vector<T>::vector<T>() {
    _size = 0;
    _capacity = kInitialSize;
    _elems = new T[kInitialSize];
}

template <typename T>
vector<T>::vector<T>() :
    _size(0), _capacity(kInitialSize),
    _elems(new T[kInitialSize]) { }
```

Directly construct each member with a starting value
Quick summary

● Prefer to use member initializer lists, which directly constructs each member with a given value.
  ○ Faster. Why construct, then immediately reassign?
  ○ Members might be a non-assignable type (we’ll see by the end of lecture how this can be possible!)

● Important clarification: you can use member initializer lists for ANY constructor, not just a special member function. This means you can use it even if your constructor has parameters.
Why aren’t the default member functions always sufficient?
The default compiler-generated copy constructor and copy assignment functions work by copying each member variable.
template <typename T>
vector::vector<T>(const vector::vector<T>& other) :
    _size(other._size),
    _capacity(other._capacity),
    _elems(other._elems) { }
Consider the following code

This code, sadly, does not work!

```cpp
vector<int> operator+(const vector<int>& vec, int elem) {
    vector<int> copy = vec; // uses default copy constructor
    copy += element; // assumes we've defined += operator
    return copy;
}
```
Both copy and vec will point to the same underlying array!

```cpp
vector<int> operator+(const vector<int>& vec, int elem) {
    vector<int> copy = vec;
    copy += element;
    return copy;
}
```
The culprit? This line in the default copy constructor!

Remember, _elems is a pointer, so this line makes a copy of a pointer, which isn’t the same as copying the underlying array!

```cpp
template <typename T>
vector::vector<T>(const vector::vector<T>& other) :
    _size(other._size,
    _capacity(other._capacity),
    _elems(other._elems) { }
```
The culprit? This line in the default copy assignment!

This is because when you copy a pointer, you copy the address saved in the pointer, not what’s being pointed to!

template <typename T> 
vector::vector<T>(const vector::vector<T>& other) : 
  _size(other._size, 
  _capacity(other._capacity), 
  _elems(other._elems) {}
Moral of the story: in many cases, copying is not as simple as copying each member variable.
This is one example of when you might want to **overwrite** the default special member functions with your own implementation!
Copy operations: fixing the issues we just saw
Recap about definitions

- Default construction: object created with no parameters.
- Copy construction: object is created as a copy of an existing object.
- Copy assignment: existing object replaced as a copy of another existing object.
- Destruction: object destroyed when it is out of scope.
How do we fix the default copy constructor? (chat)

```cpp
template <typename T>
vector::vector(const vector<T>& other) :
    _size(other._size,
    _capacity(other._capacity),
    _elems(other._elems) {

```
Here's a fix: we can create a new array!

```cpp
template <typename T>
vector::vector(const vector<T>& other) :
    _size(other._size, 
    _capacity(other._capacity), 
    _elems(other._elems) {
        _elems = new T[other._capacity];
        std::copy(other._elems, 
                   other._elems + other._size, _elems);
    }
```
Even better: let’s move things to the initializer list!

template <typename T>
vector::vector<T>(const vector<T>& other) :
    _size(other._size,
    _capacity(other._capacity),
    _elems(new T[other._capacity]) {
    std::copy(other._elems,
               other._elems + other._size, _elems);
}
How do we fix the default copy assignment operator?

template <typename T>
vector<T>& vector<T>::operator=(const vector<T>& other) {
    _size = other._size;
    _capacity = other._capacity;
    _elems = other._elems;

    return *this;
}
template <typename T>
vector<T>& vector<T>::operator=(const vector<T>& other) {
    _size = other._size;
    _capacity = other._capacity;
    _elems = new T[other._capacity];

    std::copy(other._elems,
              other._elems + other._size, _elems);
}
There’s a problem here with memory leaks!

```cpp
template <typename T>
vector<T>& vector<T>::operator=(const vector<T>& other) {
    _size = other._size;
    _capacity = other._capacity;
    _elems = new T[other._capacity];

    std::copy(other._elems,
              other._elems + other._size, _elems);
}
```

Remember, we’re changing the members of an existing object. What about the old array of elements that _elems pointed to?
template <typename T>
vector<T>& vector<T>::operator=(const vector<T>& other) {
    _size = other._size;
    _capacity = other._capacity;
    delete [] _elems;
    _elems = new T[other._capacity];

    std::copy(other._elems,
               other._elems + other._size,
               _elems);
}
template<typename T>
vector<T>& vector<T>::operator=(const vector<T>& other) {
    _size = other._size;
    _capacity = other._capacity;
    delete [] _elems;
    _elems = new T[other._capacity];

    std::copy(other._elems,
              other._elems + other._size, _elems);

    return *this;
}
template <typename T>
vector<T>& vector<T>::operator=(const vector<T>& other) {
    if (&other == this) return *this;
    _size = other._size;
    _capacity = other._capacity;
    delete [] _elems;
    _elems = new T[other._capacity];

    std::copy(other._elems,
              other._elems + other._size, _elems);
    return *this;
}
Summary: Steps to follow for an assignment operator

1. Check for self-assignment.
2. Make sure to free existing members if applicable.
3. Copy assign each automatically assignable member.
4. Manually copy all other members.
5. Return a reference to *this (that was just reassigned).
The copy operations must perform the following tasks

<table>
<thead>
<tr>
<th>Copy constructor</th>
<th>Copy assignment</th>
</tr>
</thead>
</table>
| Use initializer list to copy members where simple copying does the correct thing.  
  • int, other objects, etc.  
| Clean up any resources in the existing object about to be overwritten.  
| Manually copy all members where assignment does not work.  
  • pointers to heap memory  
  • non-copyable things  
| Copy members using direct assignment when assignment works.  
| Manually copy members where assignment does not work.  
| (Not necessarily in this order)  

🤔 Questions? 🤔
Delete
How could you prevent copies from being created?

Explicitly delete the copy member functions!

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

    private:
        // other stuff
};
How could you prevent copies from being created?

Explicitly delete the copy member functions!

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

    private:
    // other stuff
};

This was why we couldn’t capture the WikiScraper by value in assignment 1!
Rule of three
When to define your own special member functions?

- When the default ones generated by the compiler won’t work
- Most common reason: there’s a resource that our class uses that’s not stored inside of our class
  - E.g. dynamically allocated memory
    - Our class only stores the pointers, not the memory itself
If you explicitly define a copy constructor, copy assignment, or destructor, you should define all three.

What’s the rationale?
Rule of three

- If you explicitly define a copy constructor, copy assignment, or destructor, you should define all three.
- What’s the rationale?
  - The fact that you defined one of these means one of your members has ownership issues that need to be resolved.
Rule of 0

- If the default operations work, then don’t define your own custom ones
Summary

● The C++ compiler is powerful enough to generate special member functions for us
● In some cases, we may need to redefine these functions if the default behavior does not match our desired behavior
● We can delete special member functions to make certain behavior impossible (e.g., make it impossible to copy an object of our class)
Questions?
Next time

Move semantics