Inheritance

Bad Dad Joke of the Day:
- What do you call a cow in a suit of armor?
- Sirloin!

Creds: NS
Game Plan

- Recap
- Announcements
- Namespaces
- Inheritance
T-minus four lectures left!

(No class during Week 10 - enjoy your true Dead Week!)
Review from Last Time

- Operators
Review from Last Time

- Operators
  - Operators tend to be written in terms of other operators!
  - Use the reference to remember what assumptions your operator overload should satisfy
Review from Last Time

- Operators
  - Operators tend to be written in terms of other operators!
  - Use the reference to remember what assumptions your operator overload should satisfy

- Better explanation for: Why learn about move?
Review from Last Time

● Operators
  ○ Operators tend to be written in terms of other operators!
  ○ Use the reference to remember what assumptions your operator overload should satisfy

● Better explanation for: Why learn about `move`?
  ○ There was a lot of hype for C++11
Challenge Mode:

```c
const int* const myClassMethod(const int* const & param) const;
```
const int* const myClassMethod(const int* const & param) const;
This function takes in a const pointer...

```c
const int* const myClassMethod(const int* const & param) const;
```
Challenge Mode:

This function takes in a const pointer...

```c
const int* const myClassMethod(const int* const & param) const;
```

...to a const int.
Challenge Mode:

This function takes in a const pointer...

const int* const myClassMethod(const int* const & param) const;

This function returns a const pointer...

...to a const int.
Challenge Mode:

```
const int* const myClassMethod(const int* const & param) const;
```

This function takes in a const pointer...

...to a const int.

This function returns a const pointer...

...to a const int.
Challenge Mode:

This function takes in a const pointer...

const int* const myClassMethod(const int* const & param) const;

This function returns a const pointer...

...to a const int.

...to a const int.

And this is a const member function, i.e. this function can’t modify any variables of the this instance.
Challenge Mode:

This function takes in a const pointer...

`const int* const myClassMethod(const int* const & param) const;`

- This function returns a const pointer...
- ...to a const int.
- And this is a const member function, i.e. this function can’t modify any variables of the this instance

When to use each?

https://stackoverflow.com/questions/15999123/const-before-parameter-vs-const-after-function-name-c
Announcements
Announcements

● Assignment 2 is due this Thursday!
  ○ Wednesday (11/13), 10 am - 12 pm, in Lathrop Tech Lounge
  ○ Wednesday (11/13), 3:30 - 4:20 pm, in Huang basement*
  ○ Wednesday (11/13), 9:30 - 11:30 pm, in Huang basement*
  ○ Thursday (11/14), 7 - 7:45 pm, in Huang basement*
  ○ *for Huang basement, meet by the Google server under the big stairs
  ○ Check the Piazza for a full list of office hours.

● Remember you have four late days to use throughout the quarter.

● Assignment 3 will be released Thursday! Due after Thanksgiving Break
Namespaces
There was a lot of std:: and StringVector:: in our code
There was a lot of `std::` and `StringVector::` in our code

```cpp
//using namespace std;

using std::cout;       using std::endl;
using std::string;     using std::vector;
using std::priority_queue; using std::ostream;
using std::unordered_set;
```
There was a lot of `std::` and `StringVector::` in our code.

```cpp
#include "stringvector.h"

// default constructor
StringVector::StringVector() :
    logicalSize(0), allocatedSize(kInitialSize) { //
    elems = new std::string[allocatedSize];
}

// fill constructor
StringVector::StringVector(size_type n, const std::string &val) :
    logicalSize(n), allocatedSize(2*n) {
    elems = new std::string[allocatedSize];
    std::fill(begin(), end(), val);
}

// destructor
StringVector::~StringVector() {
    delete[] elems;
}
There was a lot of `std::` and `StringVector::` in our code

Why?
The standard library uses common names

- string
- max
- count

It is easy for libraries to conflict in their names
Example
Namespace Clash
Most modern languages use namespaces to fix this
Most modern languages use namespaces to fix this

```python
# Generate a random number in Python
import random

print random.random()
```
Most modern languages use namespaces to fix this

# Read file in JavaScript

```javascript
const fs = require('fs');

const data = fs.readFileSync('file.txt');
```

Most modern languages use namespaces to fix this

```cpp
#include <algorithm>

std::count(v.begin(), v.end(), 1);
```
Aside: Scope Resolution

```cpp
#include "stringvector.h"

// default constructor
StringVector::StringVector() :
    logicalSize(0), allocatedSize(kInitialSize) { //
    elems = new std::string[allocatedSize];
}

// fill constructor
StringVector::StringVector(size_type n, const std::string &val) :
    logicalSize(n), allocatedSize(2*n) {
    elems = new std::string[allocatedSize];
    std::fill(begin(), end(), val);
}

// destructor
StringVector::~StringVector() {
    delete[] elems;
} 
```
Why do we need to write `StringVector::` in front of all of our class member functions?
Why do we need to write `StringVector::` in front of all of our class member functions?

→ So that the compiler knows which class you’re defining a function for!

Aside: Scope Resolution

```
#include "stringvector.h"

// default constructor
StringVector::StringVector() :
    logicalSize(0), allocatedSize(kInitialSize) { //
    elems = new std::string[allocatedSize];
}

// fill constructor
StringVector::StringVector(size_type n, const std::string &val) :
    logicalSize(n), allocatedSize(2*n) {
    elems = new std::string[allocatedSize];
    std::fill(begin(), end(), val);
}

// destructor
StringVector::~StringVector() {
    delete[] elems;
}
```
Why do we need to write `StringVector::` in front of all of our class member functions?

→ So that the compiler knows which class you’re defining a function for!
Inheritance
Warning: This is a quick overview of inheritance.

We’ll cover C++-specific details of inheritance, but won’t be spending time on when to actually use it.

Take CS 108 to learn more!
Inheritance

Motivation:

```cpp
ifstream& ifstream::operator<<(int i) {
    // Implementation
}
istringstream& istringstream::operator<<(int i) {
    // Implementation
}
```
void print(ifstream &stream, int i) {
    // do some stuff
    stream << i;
}

void print(istringstream &stream, int i) {
    // do some stuff
    stream << i;
}
Motivation:

```c++
void print(ifstream &stream, int i) {
    // do some stuff
    stream << i;
}
void print(istringstream &stream, int i) {
    // do some stuff
    stream << i;
}
```

Would much rather have just one!
Try #1:

template <typename StreamType>
void print(StreamType& stream, int i) {
    // do some stuff
    stream << i;
}

This works because templates use the concept of implicit interface.

Note that there isn’t a list of what operators/functions are required.
Inheritance

Try #1:

```cpp
template <typename StreamType>
void print(StreamType& stream, int i) {
    // do some stuff
    stream << i;
}
```

This works because templates use the concept of implicit interface.

Note that there isn’t a list of what operators/functions are required.

Next lecture: Concepts (C++20)!
If there’s an implicit interface, there must be an explicit one

Usually just called an interface, the simplest form of inheritance
In Java:

```java
interface Drink {
    public void make();
}

class Tea implements Drink {
    public void make() {
        // implementation
    }
}
```
Interface

In Java:

```java
interface Drink {
    public void make();
}

class Tea implements Drink {
    public void make() {
        // implementation
    }
}
```

In C++:

```cpp
class Drink {
public:
    virtual void make() = 0;
};

class Tea : public Drink {
public:
    void make() {
        // implementation
    }
};
```
In Java:

```java
interface Drink {
    public void make();
}

class Tea implements Drink {
    public void make() {
        // implementation
    }
}
```

In C++:

```cpp
class Drink {
public:
    virtual void make() = 0;
};
```

Called a **pure virtual** function, denoted by the `= 0`.

Means that the inheriting class **must** define that function.
There is no interface keyword in C++!

- To **be** an interface, a class must consist only of pure virtual functions

- To **implement** an interface, a class must define **all** of those virtual functions
Interfaces

There is no interface keyword in C++!

- To **be** an interface, a class must consist only of **pure virtual functions**
  
  ➔ What if we do want to define some functions in our class?

- To **implement** an interface, a class must define **all** of those virtual functions
Abstract Classes

If a class has at least one pure virtual function, then it’s called an **abstract class**. (Interfaces are a subset of abstract classes.)

Abstract classes cannot be instantiated.

```cpp
class Base {
public:
    virtual void foo() = 0; // pure virtual function
    virtual void foo2();    // non-pure virtual function
    void bar() = { return 42; }; // regular function
};
```
Try #2:

```c
void print(istream &stream, int i) {
    // do some stuff
    stream << i;
}
```

As long as istream implements print (as a non-virtual function), and all types of streams inherit from istream, you only need to write one function!
Aside: Inherited Members

No “virtual” members - instead, if a member has the same name as an inherited member, it hides it

```cpp
struct A {
    int a;
};

struct B : public A {
    double a; // Hides A::a
};
```
**Terminology**

**Base class**: the class inherited from

- aka a *superclass* or *parent* class

**Derived class**: the class that inherits from the base class

- aka a *subclass* or *child* class
Always call the superclass constructor.

class Derived : public Base {

    Derived() : Base(args), /*others*/ {
        // rest of constructor
    }

};
Only inherit from a class that has a virtual destructor!

(And if you want your class to be inheritable, make sure you make the destructor virtual!)

```cpp
virtual ~Base() {}
```

Otherwise will almost definitely have memory leaks.
class Base {
    ~Base() {}
};

class Derived : public Base {
    ~Derived() {}
}

Base *b = new Derived();
delete b; // Never calls the destructor for Derived!
Terminology: Access Specifiers

Fancy name for three words you’ve seen a lot:

- private
- protected
- public
Terminology: Access Specifiers

Fancy name for three words you’ve seen a lot:

- **private**
  - Can only be accessed by this class
- **protected**
  - Can only be accessed by this class or derived classes
- **public**
  - Can be accessed by anyone
class Base {
public:
    void foo();
protected:
    void bar();
private:
    void baz();
};

Derived classes can access foo and bar. Unrelated classes can only access foo.
Example

Simple Inheritance (if we have time)
What We Didn’t Cover

- Polymorphism
- All the tricky details of using references and pointers with inheritance
  - These are details with inheritance generally, not C++, so we don’t cover them - we encourage you to read up on inheritance if you plan to write your own class though!
- Next time: brief look at casting
Next time

Template Classes and Concepts