What We've Learned

What we learned

- Covered a lot of material this semester
- Went very fast
- Still lots more to learn

Really Three Parts

- C++ and its syntax
- The STL
- Building our own solutions

C++

- Different from Python
  - Strongly typed, everything has a type
  - More complicated syntax
- You’ve seen maybe 50% of the modern C++ syntax
  - You have a long ways to go
  - So do most people

You've seen some rough stuff

- Templates
  - Pretty challenging, but useful
- Pointers
  - Typically only seen in C / C++
  - Much more restricted in other languages
- Dynamic memory
  - Also mostly only in C / C++
  - In other languages, garbage collection handles this for you

STL

- We learned that the STL is your friend
  - Iterators
  - Generic algorithms
    - Copy, find, sort, transform, etc
  - Containers
    - Vectors, deques, maps, sets, strings
- Use if you can

Remember the STL!

- You probably will not get a lot more information on the STL during your stay here at MSU
  - Remember that it is your friend
  - Remember to look at it **first** to solve a problem you might have
  - STL is faster, less error prone, easier

Classes

- We built our own classes
  - struct and class
    - privacy, friends
  - The complicated construction, destruction of objects
  - Overloaded operators

You have a lot of the basics

- From
  - vectors / maps
  - dynamic memory / arrays
  - linked lists
- These are the foundations. Take a look!
- https://en.wikipedia.org/wiki/List_of_data_structures

What We Didn't Teach (Yet)

What's next?

- CSE 331: More complicated data structures and algorithms
- CSE 335: Last course on C++, inheritance, virtual functions, group work
- There are lots of specialty courses (you get to chose)
  - Graphics
  - Database
  - Security
  - Compilers
  - Etc.

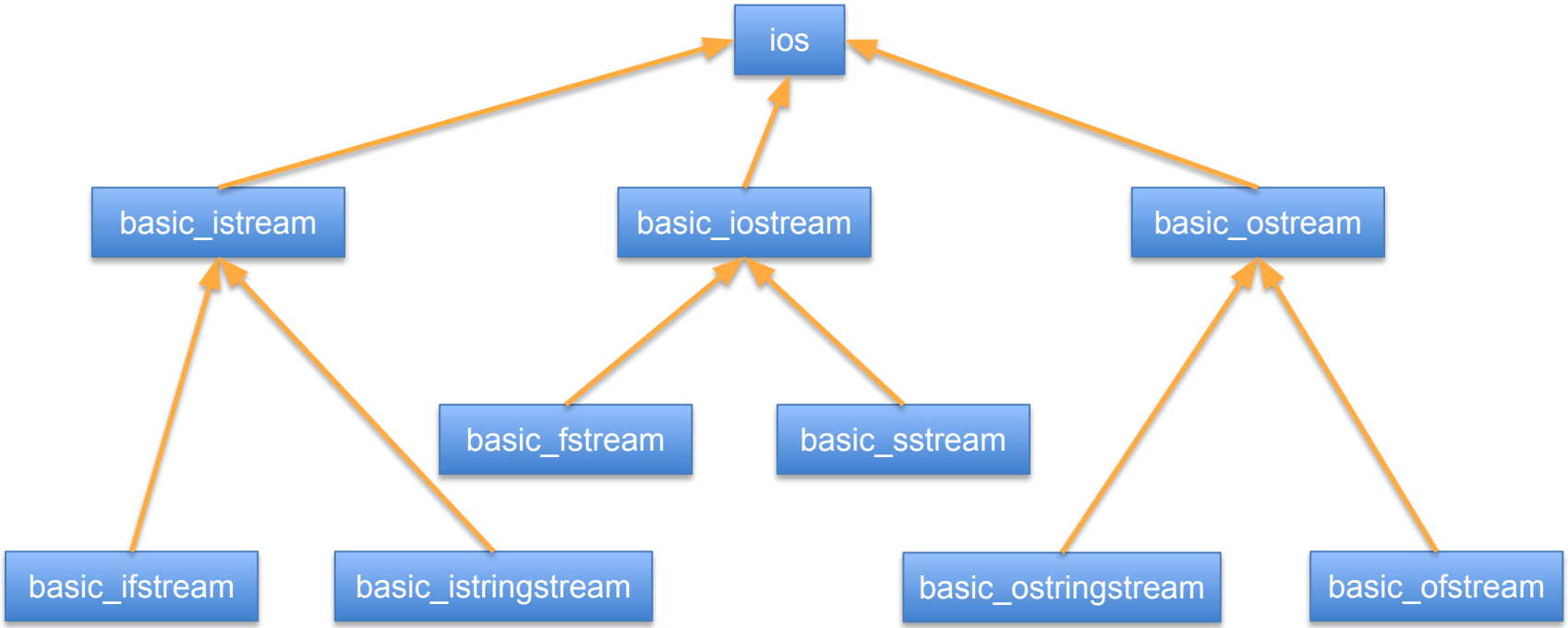
What we didn't cover yet

- Inheritance
- Smart pointers
- Move semantics
- Multithreaded programs
- Static and global variables

Inheritance

- We can inherit behavior from a common ancestor and specialize those aspects that are unique to the class

C++ Stream Hierarchy



Sharing is good!

- Share the functionality in a parent class, specialize in the child
  - Update a parent, all children updated
  - Common functions implemented once
  - Shared element for multiple uses

Virtual functions

- One aspect of OO programming is being able to decide at runtime what method to call
- Runtime polymorphism

Parent point-to / reference a Child instance

- You have a Parent class and a Child class
- A Parent *ptr can point to a Child instance

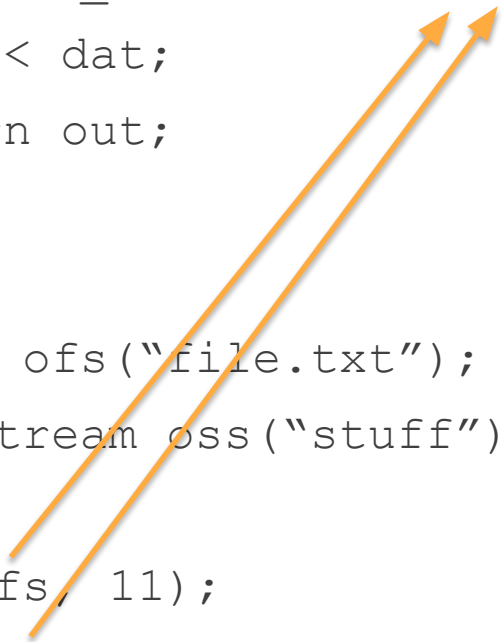
```
Parent *ptr = new Child();
Parent &ref = *ptr;
```

We have seen this in streams

```
ostream& my_fun(ostream& out, long dat) {
    out << dat;
    return out;
}

ofstream ofs("file.txt");
ostringstream oss("stuff");

my_fun(ofs, 11);
my_fun(oss, 11);
```



Either works, each a child of the ostream parent

So which method is called?

```
Parent *ptr = new Child();
ofstream ofs(“file.txt”);
ptr->print(ofs);
```

Is it the Parent (which is the pointer type) or the Child (which is pointed to)?

Virtual Function

- If the method is declared virtual, then the decision as to which to call (what is the pointed to type, not the pointer type) is delayed until runtime!
- The compile-time type does not determine the action performed!
- More details in CSE 335

Maps / sets

- The ordered map and set are typically implemented as trees with algorithms to keep the trees full
  - You get O(log n) lookup on the key elements
  - More details in CSE 331
- There is also an unordered map / set
  - Uses a different underlying approach called a hash table
  - Much faster to access elements, O(~1)
  - There are complications
  - More details in CSE 331

C++14/17 Stuff

- There are lots of features, especially in C++14 and C++17 that we just didn’t have time to get to
- These might be things you want to look at in the future

“Smart” pointers

- We did not work with any of the smart pointers
- These pointers remember to deallocate themselves when finished
  - shared_pointer
    - Remembers who is using it
    - Removing the last reference automatically deallocates
  - unique_pointer
    - Can only be used by one referent
  - More in CSE 335

Prevents Leaks

- Smart pointers help avoid memory leaks
  - Some caveats
- Don’t delete a pointer
  - They handle that for you
  - Do still need to do stuff with a destructor

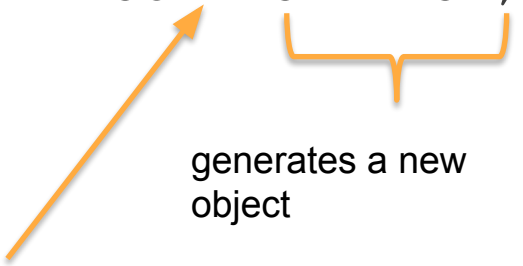
Move semantics

- Where is the efficiency loss below?

```
SomeObj o1, o2, o3;
o3 = o1 + o2;
```

Move semantics

```
SomeObj o1, o2, o3;
o3 = o1 + o2;
```



generates a new object

Copies the new obj to o3

What happens to the object
returned from the op+?

Destroyed

Why not just move it?

- By move we mean that we
  - **Reassign** the resources from the op+ return to o3, no copy required
    - Like copy-swap
  - Skip the destruction
- Very efficient!
- Compilers can do this automatically, called **copy elision**
  - Now under C++ control

The && type

- Can designate an r-reference (&&)
- This is a move element.
- Rule of 3 becomes 5
  - Move constructor
  - Move assignment

Moves can “pass object” around

- If you move an object into a function (pass it as an && value) then the value indeed moves into the function and the value from where the pass came from is now invalid
- Sounds weird, but it has its advantages, especially for “stateful” objects

The emplace methods

- Why is the following inefficient?

```
vector<string> v;
v.push_back(“some chars”);
```

copies and a destructor call

- We know that “some chars” is not a `string`, but a `char*`, in fact `char[10]`
  - Implicitly convert `char*` to `string`, constructing a `temp` string var
  - `push_back` is a copy operation (as are all STL ops by default), so `temp` is copied into the `vector`
  - The original `temp` is now destroyed (destructor call)

emplace_back

- `vector` also has `emplace_back` method

```
vector<string> v;
v.emplace_back(“some chars”);
```

- The temp string var is still constructed but now directly in the vector
  - No copy
  - No call to destructor

Thread Models

- Threads are small units of execution that, potentially anyway, can be run independently on multiple cores
- C++11 has a complete thread model with some pretty fancy high-level interface elements

Static variables, static functions, static instances, global variables, ...

Basically we only taught you about local variables (which are enough for 95% of problems).

Where To Go From Here

So how hard is C++ really?

- Honestly, it is pretty hard
  - Syntax is pretty unforgiving
  - Error messages are cryptic
  - Templates are tough to work with
- And there is a lot more to learn. C++ is one of the hardest languages to work with.

So why C++ again?

- You got to look under the hood
  - Strong types
  - Dynamic memory
  - Pointers
- These are parts of most programming languages, just not always programmer accessible
- If you never use them again, you now know more about how a language really works

So why C++

- Efficiency is important and C++ allows you to have it all in your hands if you want it
- You get to pick the level you work at
  - STL
  - Pointers
  - Dynamic memory
  - Do what you want / need

It takes time

- C++ is hard
- You get better with practice
- Pick a language and get **good** at it. Other concepts come easier

What to do to get better

- You are now developed programmers
  - Multiple languages
  - Multiple problems
  - Data structures
  - **Algorithms**
- What to do to get better?

Moving from language to algorithm

- After you've seen (at least) two programming languages, you start to abstract away how to "write a program" and can focus on "algorithm"
- Syntax gets in the way less and less
- You become a programmer, not a C++/Python programmer

Four things

- Planning / experimentation
- Deliberate programming
- Testing
- Lifelong learner

Plan / experiment

- When you have a problem, better to plan and/or experiment before coding
  - Try some ideas out
    - In code or otherwise
  - Draw some pictures
- Can’t get anywhere until you know roughly where you are going

Deliberate Programming

- Remember this workflow
  - Write a line
  - Compile a line
  - Write a line
  - Compile a line
- Make sure each line does what you think.
- Stop writing junk and calling it code
- Be deliberate!

Testing

- **Test everything, all the time!**
- One of the best signs of a mature programmer is their ability to write small pieces of code and test it
- I do it all the time. Do you?

Lifelong learner

- In no other industry do things change as fast as the one you are thinking about
- Plenty of jobs for people who learned to do X (PHP, MySQL, C#, etc) did it for a living, never learned anything new, and were dead-enders in their 30s

Keep learning

- Make the choice to learn. Learning will keep you flexible, help you respond to change, make your life better.

- “The capacity to learn is a gift; the ability to learn is a skill; the willingness to learn is a choice.”
  - Brian Herbert

Thanks!

Please let me know about any feedback you have on the class via the Online SIRS.

If you are interested in being an Undergraduate Learning Assistant (ULA), where you would lead lab sections, hold help room, or answer questions on Piazza for future offerings of CSE 232, reach out to me!