Last day

Some cleaning up still going on, but my intent is to send out a final report (missing only the final exam grade and project 11) on Friday.
- I'll update next week with project 11 before the exam (I have plenty of time unfortunately 😊)
- I'm working through latest complaints!

Final exam

200 points, comprehensive
- focus on the last third of the class
  - class templates, arrays, dynamic memory, bigO, recursion, linked lists,
  - Friday, December 16th, 7:45am, here
- no makeup, don’t be late!!!
- setup.
  - cheat sheet, 40 questions,

What I learned in CSE 232

and where to go from here
So what did I learn again?

At this point it may be a little hard to remember, but we did a lot this semester

What we were supposed to do

"Continuation of object-centered design and implementation in C++. Building programs from modules. Data abstraction and classes to implement abstract data types. Static and dynamic memory allocation. Data structure implementation and algorithm efficiency. Lists, tables, stacks, and queues. Templates and generic programming."

Really three parts

• C++ and its syntax
• The STL
• Building our own

C++

Different from Python
• strongly typed, everything has a type
• multi paradigm.
• more complicated syntax

You've maybe seen 50% of the modern C++ syntax. You have a ways to go.
• so do most people.
But you've seen some rough stuff

- templates
  - pretty hard, but useful
- pointers
  - typically only seen in C/C++
- dynamic memory
  - also mostly only in C/C++

We learned that the STL is your friend

- iterators
- generic algorithms
  - copy, transform, sort, find, etc.
- containers
  - vectors, deques, maps, sets, strings

Use if you can. We even have a good idea how they really work!

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!

We built our own classes

- struct and class
  - privacy, friends
- the complicated construction, destruction of objects
- overloaded operators
You’ve worked with a lot:

### Data Structures

<table>
<thead>
<tr>
<th>vectors</th>
<th>string (STL/dyn-mem)</th>
</tr>
</thead>
<tbody>
<tr>
<td>deque</td>
<td>queue</td>
</tr>
<tr>
<td>stack (dyn-mem, link-list, vector)</td>
<td>circular buffer</td>
</tr>
<tr>
<td>BiStack</td>
<td>2D vector (hole cnt)</td>
</tr>
<tr>
<td>linked list (Painter’s)</td>
<td>sorted list</td>
</tr>
<tr>
<td>map (STL, greedy Network)</td>
<td>arrays</td>
</tr>
<tr>
<td>vector (STL/dyn-mem)</td>
<td>strings (Pangram cipher)</td>
</tr>
</tbody>
</table>

We did a little work with algorithms too!

- memoized fibonacci
- hole counting
- greedy alg.
- ciphers
- Finite State Automata
- ngrams

What's next?

- 331, trees, more complicated data structures and algorithms
- 335, last course on C++, inheritance, virtual functions, group work

Then there are a host of specialty courses: graphics, database, security, compilers, etc. You get to choose.

You have a lot of the basics

From

- vectors/maps
- dynamic memory/arrays
- linked lists
- trees

These are the foundations. Take a look!

**C++, what we missed!**

We've seen this, but haven't written our own yet.

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

**Inheritance**

**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 function/method to call
Runtime polymorphism!

Parent point-to/reference a Child instance
If you have a Parent class and a Child class.
A Parent *ptr can (in C++) 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");
ostreamstream oos("stuff");

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

Which method is called?
Is it the Parent (which is the pointer type) or the Child (which is what is pointed to)
virtual function (335)
If the method is declared `virtual`, then the decision as to which to call (what is pointed to, not the pointer type) is delayed until runtime!

More details than that, but the basic idea is that for once the compile-time type does not determine the action performed!

maps/sets
The ordered map and set are typically implemented as trees, with algorithms to keep the trees full
• thus you get $O(\log n)$ lookup on the key elements

But there is another

unordered map/set
Uses a different underlying approach called a `hash table`

A hash table can be much faster to access elements, around $O(1)$, but there are always complications

hash table (331)
A hash table can be thought of initially as a vector, usually a very large vector, where the index indicates the value stored for a key.

But how to map the key to the index value?
A hash function is a translation of a map’s key into an index value. For example, if the key was a string, you could add up the integer values of each character, modulo the vector/table size.

**Example**

- Table 200 big.
- Key is "abc", value is 123
- "abc" = 97 + 98 + 99 = 294 % 200 = 94
- Store 123 at index 94

**Wins/Losses**

- Lookup is O(1), just some math
  - For large data much faster than map/set
- How to handle collisions, two different keys to same index:
  - Linked list at that index
  - Very large table and a good hash function

**POD vs user defined class**

To use unordered map/set you must have a hash function defined:

- For POD (plain old data), hash already defined
- For your own key class/struct, you must define one
  - Can be hard to do
There are a lot of features, especially in C++11 and C++14 we just didn't have time to get to.

These might be things you take a look at in the future.

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 last reference automatically deallocates
- `unique_pointer`, can only be used by on referent
- should see in 335

Syntax is a little challenging, and doesn't save you from all those seg faults.

However, does avoid memory leaks! Don't have to delete a pointer, though still stuff to do with a destructor.

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

Where is the efficiency loss below

move semantics
SomeObj o1, o2, o3;

o3 = o1 + o2

generates a new object

copies the new obj to o3

What happens to the object return from the op+ overload?

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 then.

Very efficient!

Compilers can do this automatically, called **copy elision**

- now under C++ control

the && type

Can now 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 a && 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?

```cpp
vector<string> v;
v.push_back("some chars");
```

2 copies and a destructor call

We know that "some chars" is not a string, but a char*, in fact char[10]
1. implicitly convert char* to string, 
   constructing a temp string var
2. push_back is a copy operation (as are all STL ops by default), so temp is
   copied into the vector
3. the original temp is now destroyed (destructor call).

emplace_back

vector also has emplace_back method

```cpp
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

others

If your container supports
- push_front
- there is also emplace_front

- insert
- emplace
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.

Moving on

So, how hard is C++ really

Honestly, it is pretty hard.
• syntax is pretty unforgiving
• error messages (holy s*@t)
• templates tough to work with
And there is a lot more to learn. C++ in its entirety is one of the hardest languages to work with. Few really know it all

So why C++ again

You got to look under the hood:
• strong types
• dynamic memory
• pointers
These are all part of 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++ (2)

- 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 what want/need

Every time I teach this class I learn something new. Still. And I’ve been at it awhile. C++ is just that hard.

You get better by practice. If you can, pick a language and get good at it. Other concepts come easier.

It takes time

What to do to get better

You are all 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 more 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

---

**Remember this guy**

- Me: "Why is this in your program?"
- You: "It wouldn't compile/run if I didn't"

Program like you know what you're doing. Stop guessing, know! If you don't know, test!

---

**deliberate programming**

Remember this from the first day:

- write a line
- compile a line
- write a line
- compile a line
  - ...

Make sure each line does what you think. Stop writing crap and calling it code. Be deliberate!
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, you've seen me do it. Do you?

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#, blah, blah, blah), did it for a living, never learned anything new and were dead-enders in their 30's

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

Let me tell you the secret that has led to my goal. My strength lies solely in my tenacity.
-- L. Pasteur

It's not that I'm so smart, it's just that I stay with problems longer.
-- A. Einstein

It does not matter how slowly you go as long as you do not stop.
-- Confucius

It is only through labor and painful effort, by grim energy and resolute courage, that we move on to better things.
-- T.R. Roosevelt

Age wrinkles the body, quitting wrinkles the soul
-- D. Macarthur

Dripping water hollows out stone, not through force but through persistence.
-- Ovid