# Lecture 13
# Access Control

CS211 – Fundamentals of Computer Programming II

Branden Ghena – Fall 2021

Slides adapted from:
Jesse Tov

# Administrivia

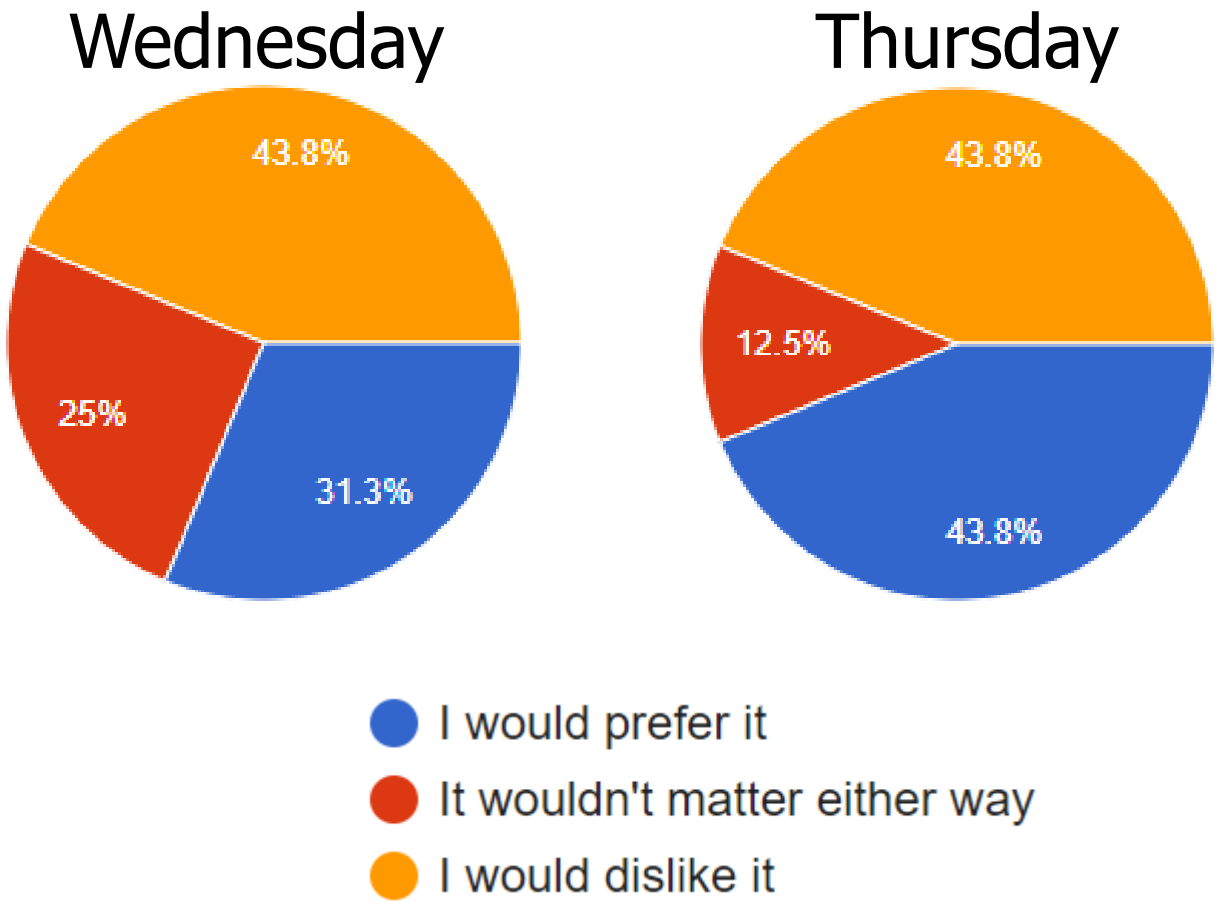
- Homework 5 is underway
  - Remember this is a SOLO ASSIGNMENT
  - Hardest part: getting used to C++ syntax
- Example: calling a function on an object
  - Documentation: `Posn<int>::right_by(…)`
  - Means: `Posn<int>` has a member function called `right_by()`
  - To call it: `pos.right_by(…)`

# Warning: CLion isn’t always trustworthy

- CLion tries too hard to be useful
  - And can end up changing files you didn’t mean to
  - When it pops up and asks if you want to do something, usually the answer is “No!”
    - Example: static functions
- This can end up changing code in files you didn’t mean to touch
  - Easiest fix is often to check out the project again and move your files over

# Survey results

- Question:
  Should we change some office hours to in-person?
- Mixed responses
- Going to stay as-is for this quarter
- Keeping a homework FAQ post on Campuswire



# Today's Goals

- Continue practice on constructors and objects
- Discuss using exceptions to signal errors
- Introduce concept of encapsulation and access control
  - How technically it's done in C++
  - Why we care about it

# Getting the code for today

- Download code in a zip file from here: https://nu-cs211.github.io/cs211-files/lec/13_access.zip

- Extract code wherever

- Open with CLion
  - Make sure you open the folder with the CMakeLists.txt

# Outline

- **More Constructors**
- Exceptions
- Access Control
- Encapsulation Policy

# Today’s working example

- String_Holder
  - Manages strings using a constant-length array to hold characters
  - Members:
    - `int length`
    - `char characters[80]`
  - Rules (invariants)
    - 0 <= `length` <= 80
    - `length` matches the number of valid characters in `characters`

# Live Coding: constructors for String_Holder

src/string_holder-implemented.cxx
src/string_holder.hxx

- `String_Holder::String_Holder()`
  - Initialize empty
- `String_Holder::String_Holder(const char* str)`
  - Construct from null-terminated string
- `String_Holder::String_Holder(const char* str, int len)`
  - Construct from a length of characters
- `String_Holder::String_Holder(const String_Holder& other)`
  - Copy constructor (from another `String_Holder`)

# Delegating constructors

- One constructor can call another to handle initialization
  - Delegates construction to that other constructor

```
// defined somewhere else
String_Holder::String_Holder(const char* str, int len);


// delegates to other constructor
String_Holder::String_Holder(const String_Holder& other)
  : String_Holder(other.characters, other.length)
{}
```

# Explicit constructors

- The `explicit` keyword before a constructor means that the constructor must be manually called by the developer
  - Rather than automatically called by the compiler

- Reason to have compiler automagic:

  - `String_Holder str = “Test”;`

  - Automatically calls `String_Holder::String_Holder(“Test”);`
    - Kind of nice that it just works…

# Explicit constructors

- The `explicit` keyword before a constructor means that the constructor must be manually called by the developer
  - Rather than automatically called by the compiler

- Reason to use `explicit`:
  - `void do_complicated_string_stuff(String_Holder str);`
  - `do_complicated_string_stuff("Test");`
  - Also automatically calls the constructor
    - But maybe the user just passed in the wrong argument and a compile error would have been better…

# Enforcing invariants with constructors

- What if a user violates the rules?
  - 0 <= `length` <= 80
  - `length` matches the number of valid characters in `characters`

- Possibilities
  - Probably length should be an `unsigned int` to start with
  - Truncate length to 80
  - Only copy over as many characters as will fit

  - But what if there's no obvious choice for what to do?
    - Constructor cannot return a value to say it failed

# Outline

- More Constructors
- **Exceptions**
- Access Control
- Encapsulation Policy

# Exceptions conceptually

- Stop running this code and return a special error to the caller
- Things went wrong, so we can’t just keep executing code like normal
- If the caller doesn’t expect the error and can’t handle it, repeat the process
  - Again stop running the code and return the special error

# Exceptions are “thrown” by the function

- `throw` keyword performs the special “error return”

- Takes an argument of the error to return
  - Example:

```
throw std::invalid_argument(“String is too long”);
```

- Actually, you can throw anything (for historical reasons)

```
throw 6;
```

  - You should almost certainly throw a class based on `std::exception`
    - https://en.cppreference.com/w/cpp/error/exception

# Properly handling exceptions

- If no caller in the "call stack" handles the exception, the program will exit

- Handle exceptions with a try-catch block

```
try {
  // code that could throw an exception goes here
} catch (const std::invalid_argument& ex) {
  // code to handle the exception goes here
}
```

  - This example only catches `std::invalid_argument` exceptions

# General try-catch form

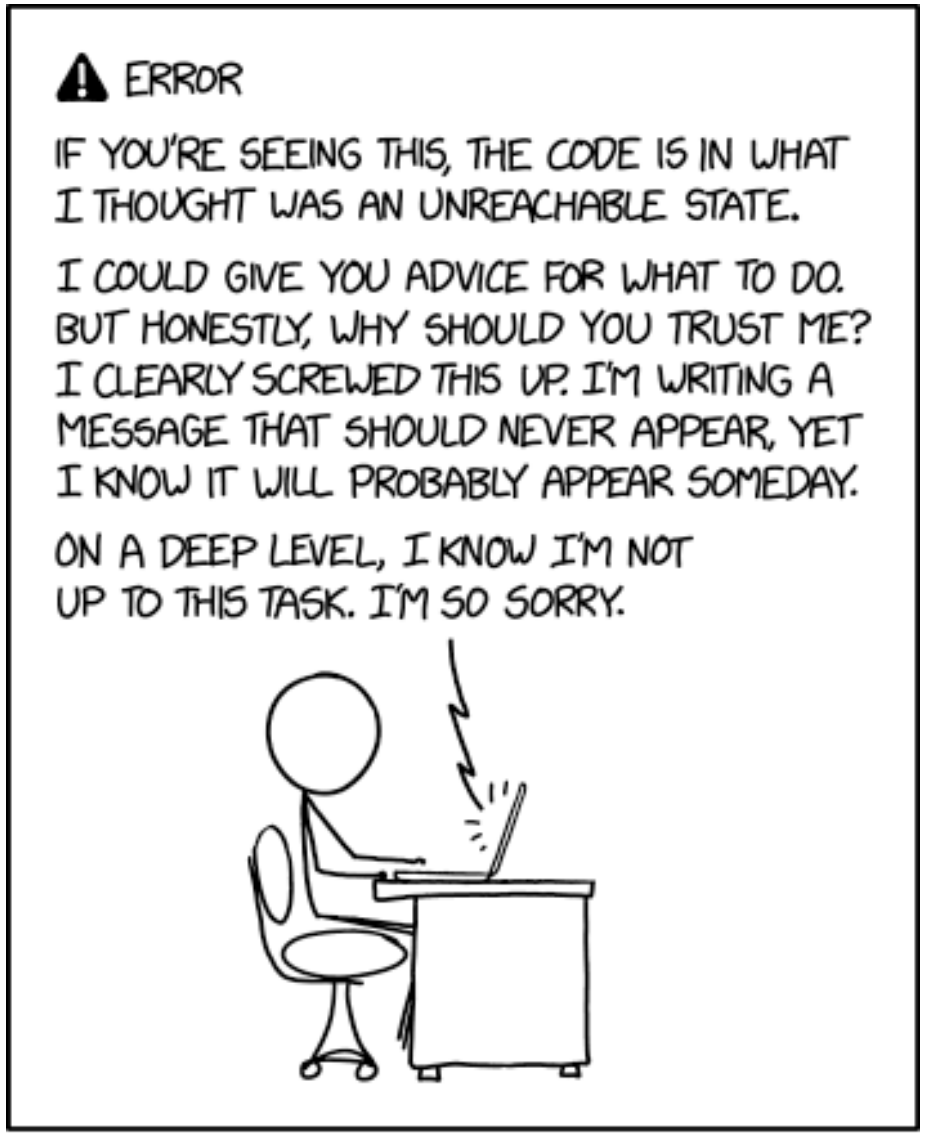
```
try {
   // code that could throw exceptions
} catch ( some specific exception ) {
   // handler code
} catch ( another specific exception ) {
   // handler code
} catch (...) {
   // general case matches all exceptions
   // actually includes the ... in the C++ code
}
```

# Live coding: exceptions

src/string_holder-exception.cxx

- Functions to add to:
  - String_Holder::String_Holder(const char*, int)
    - Ensure that int values are:
      - >= 0
      - < MAX_STRING_LENGTH
  - String_Holder::char_at(int)
    - Ensure that int values are:
      - >= 0
      - < length

# Break + Relevant XKCD




https://xkcd.com/2200/

# Outline

- More Constructors
- Exceptions
- **Access Control**
- Encapsulation Policy

# The problem of public access

- Constructors (and other member functions) that enforce rules are insufficient
  - Anyone could access the data member directly

```
String_Holder str("Test String");

str->length = 5000;

std::cout << str; // oops, UNDEFINED BEHAVIOR
```

# Access modifiers

By default, all data and functions are “public”

```
struct My_struct {

  // accessible to all parts of the program

}
```

# Access modifiers

Can choose to make data/functions “private”

```
struct My_struct {

private:
   // accessible only to member functions

}
```

# Access modifiers

Can choose exactly which data / functions are publicly accessibly versus privately accessible!

```
struct My_struct {

public:
 // accessible to all parts of the program

private:
   // accessible only to member functions

}
```

# Access modifiers

Can choose exactly which data / functions are publicly accessibly versus privately accessible!

```
struct My_struct {

public:
 // accessible to all parts of the program


private:
  // accessible only to member functions


public:
 // accessible to all parts of the program
}
```

# Structs versus Classes

- Struct and Class are interchangeable
  - The difference is the default behavior
  - Both can use `private:` and `public:` access modifiers

```
struct Test {
   // accessible to all parts of the program
}

class Test {
   // accessible only to member functions
}
```

# Style convention

- Use classes for abstractions (smart data)
  - Example: String_Holder, Ball
- Use structs for “plain old data”
  - Example: Position, Dimension
- We intentionally violated this in homework 5 to keep things simple
  - And to make transition from C simpler: “structs with functions”

# Additional specifier: protected

- Like private, but accessible to classes that inherit from this one
  - i.e., other classes that are based on this one

  - Will talk about more next week
  - If you see it around before then, consider it the same as private

# Outline

- More Constructors
- Exceptions
- Access Control
- **Encapsulation Policy**

# Encapsulation

- Goal: protect the rules of your data so it remains consistent

- Method:
  1. Make the data private

  2. Add public member functions to let clients do useful things

  3. Don’t add public member functions that let clients do bad things (like break the rules of the data)

# Step back: why do we care about consistency?

- Helps us avoid **UNDEFINED BEHAVIOR**
  - Keep track of sizes of arrays, for instance
- Avoids errors
  - Maybe you expect your data to always be sorted
- Improves efficiency
  - Make assumptions about the data that you know MUST be true

# Live coding: update String_Holder access control

- Data members should be private
  - Convention: private members end with "_"

- Functions should be public
  - And functions should never allow the rules to be broken

# Encapsulation cuts off direct access to data members

- Problem: functions outside of the class can never access data members, even to just read from them

- Options:
  1. Include as a member function
  2. Add "getters" for data variables
     `String_Holder::size()`
  3. Declare function as a `friend`

# Allowing specific things access to private members

- friend keyword declares another thing that can access private members from this class

- Example overloaded operator! `operator<<()`
  - Needs to access the private members of String_Holder
  - Inside the String_Holder class definition, add:

```
friend std::ostream& operator<<(std::ostream&, const String_Holder&);
```

# Welcome to Encapsulation

- Software engineering principle:
  1. Bundle your data and operations together
  2. Don’t let non-bundled operations mess with your bundled data

- Benefits
  - Correctness
    - Data will never become inconsistent
  - Flexibility
    - Implementation details can change without modifying the API

# Outline

- More Constructors
- Exceptions
- Access Control
- Encapsulation Policy