Lecture 12
Access Control

CS211 – Fundamentals of Computer Programming II

Branden Ghena – Spring 2023

Slides adapted from:
Jesse Tov (Northwestern), Hal Perkins (University of Washington)

Northwestern

Administrivia

- Exercise 5 due today!

- Office hours with me after class today (3:30-5:30)
    - Moved to Annenberg G32 for today only!

Homework 4: C++ syntax

- Homework 4 is underway
  - 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(…)
    - (use whatever your object name is for pos)

Homework 4 tips

- Read the hxx files!
  - `ball.hxx` and `model.hxx` have explanations about what the functions should do and how they should work

- Read the GE211 documentation
  - To see how various classes work
  - https://tov.github.io/ge211/namespacege211.html

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

Today's Goals

- Continue practice on constructors and objects
  - Discuss operator overloading
  - 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/12_access.zip

- Extract code wherever

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

Outline

- **Constructors**
- Operator Overloading
- Exceptions
- Access Control
- Encapsulation Policy

Contructors initialize newly-created objects

- Written with the class name as the method name, no return value!

```
Position(double x, double y);
```

- Allow us to define how data is initialized
  - Might use inputs as values for some data members
  - Might give default values to some data members
  - Might do some computation to decide what data members should be

  - Any and all of the above

Default constructor

- If you do not create a constructor, C++ will attempt a default
  - Leave all basic types uninitialized
  - Call the default constructor on all data members that are objects

- This is how we’ve been using Position so far

- C++ notation
  - Basic data types: plain old data (POD)
  - Object data types: non-POD

Writing our own constructor

src/position.hxx
src/position.cxx

```
struct Position {
  double x;
  double y;
  Position(double in_x, double in_y);
}
```

Note: doesn't return `void`
Has no return at all!

```
Position::Position(double in_x, double in_y) {
  x = in_x;
  y = in_y;
}
```

Initialization lists

- C++ lets you optionally declare an initialization list as part of your constructor definition
  - Lists fields and initializes them, one-by-one
  - **MUST** be in same order as the data members are in the struct

```
Position::Position(double in_x, double in_y)
    : x(in_x),
      y(in_y)
{ } // must have function body, even if empty
```

Initialization lists

- **Always** write initializer lists for constructors
  - *Nearly* identical to doing it manually
  - But the word nearly hides a lot of pain there

- Examples:
  - Data members that don't have a default constructor need to be created in the initializer list

  - Data members that are references can never be NULL, so they don't have a default! But the initializer list can still set them

Must use exclusively default constructors or defined ones

- Once you create a single constructor, C++ will no longer allow default ones
  - So if you want more options, you'll need to make them!

- Remember: C++ allows multiple functions with the same name, as long as their input arguments are different
  - We can create multiple constructors!

Multiple constructors make objects easier to use

src/position.hxx
src/position.cxx

- Default constructor

```
Position::Position()
    : x(0),
      y(0)
{ }
```

- Constructor with arguments

```
Position::Position(double in_x, double in_y)
    : x(in_x),
      y(in_y)
{ }
```

Copy constructor

src/position.hxx
src/position.cxx

- Makes a copy of an existing object

```
Position::Position(const Position& orig)
    : x(orig.x),
      y(orig.y)
{ }
```

- Can be called automatically or used via assignment

```
Position x;

Position y(x);

Position z = x;
```

When do copies happen?

- The copy constructor is invoked if:

1. You *initialize* an object from another object of the same type

```
Position x;      // default constructor
Position y(x);  // copy constructor
Position z = y;// copy constructor
```

2. You pass a non-reference object as a value parameter to a function

```
void foo(Position x) { ... }

Position y; // default constructor
foo(y);       // copy constructor
```

3. You return a non-reference object value from a function

```
Position foo() {
  Position y; // default constructor
  return y;   // copy constructor
}
```

Destructors

src/position.hxx
src/position.cxx

- Same concept as constructors: used to clean up an object
  - Automatically called when the object goes out of scope
  - Note: you **never** call the destructor yourself!

- Handles any cleanup, including freeing necessary resources

```
Position::~Position() {
  // nothing to clean here since we don’t use
  // dynamic memory
}
```

Break + Question

- Why make a constructor instead of having users set individual fields?

Break + Question

- Why make a constructor instead of having users set individual fields?

  - Constructor can ensure that everything is initialized

  - Constructor knows what the rules are!
    - Can check that the inputs are valid

  - Generally: harder to make mistakes when using someone else's code

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.cxx

- `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…

Outline

- Constructors
- **Operator Overloading**
- Exceptions
- Access Control
- Encapsulation Policy

Defining operators for our objects

- One strength of C++ is that we can define how normal operators work on our objects
  - +, -, +=, ==, <<, etc.

- Most of these are not defined for you
  - How would the compiler know what they mean for a `String_Holder`?
  - An exception is assignment (=), which is defined as a copy of all fields

  - We can implement the operators ourselves though!
  - Can be implemented as standalone functions or member functions

Example overloaded operator

src/string_holder.cxx

Standalone (normal) function

Note: lhs - left-hand side, rhs - right-hand side

```
bool operator==(String_Holder const& lhs, String_Holder const& rhs){
    …
}
```

Member function (assumes the first argument is `*this`)

```
bool String_Holder::operator==(String_Holder const& rhs) const{
    …
}
```

Either is fine, but can’t do both! That would be a duplicate function

What might we want to do with our strings?

src/string_holder.cxx

(substitute `String_Holder` for `T`)

- Compare them
  - `bool operator==(T const& lhs, T const& rhs)`

- Concatenate them
  - `T operator+(T const& lhs, T const& rhs)`
  - `T& operator+=(T& lhs, T const& rhs)`

- Print them through `std::cout` (which is type `std::ostream`)
  - `std::ostream&`
    `operator<<(std::ostream& os, T const& value)`
  - Note: cannot be a member function because `String_Holder` is not the lhs

[List of operator functions: https://gist.github.com/beached/38a4ae52fcadfab68cb6de05403fa393](https://gist.github.com/beached/38a4ae52fcadfab68cb6de05403fa393)

Break + Question

- If we wanted to write operator+ as a *member function*, what would its signature be?
  - `T operator+(T const& lhs, T const& rhs)`
    (substitute `String_Holder` for `T`)

```
struct String_Holder {
    …
    ???
}
```

Break + Question

- If we wanted to write operator+ as a *member function*, what would its signature be?
  - `T operator+(T const& lhs, T const& rhs)`
    (substitute `String_Holder` for `T`)

```
struct String_Holder {
    …
    String_Holder operator+(String_Holder const& rhs) const;
}
```

Outline

- Constructors
- Operator Overloading
- **Exceptions**
- Access Control
- Encapsulation Policy

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

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

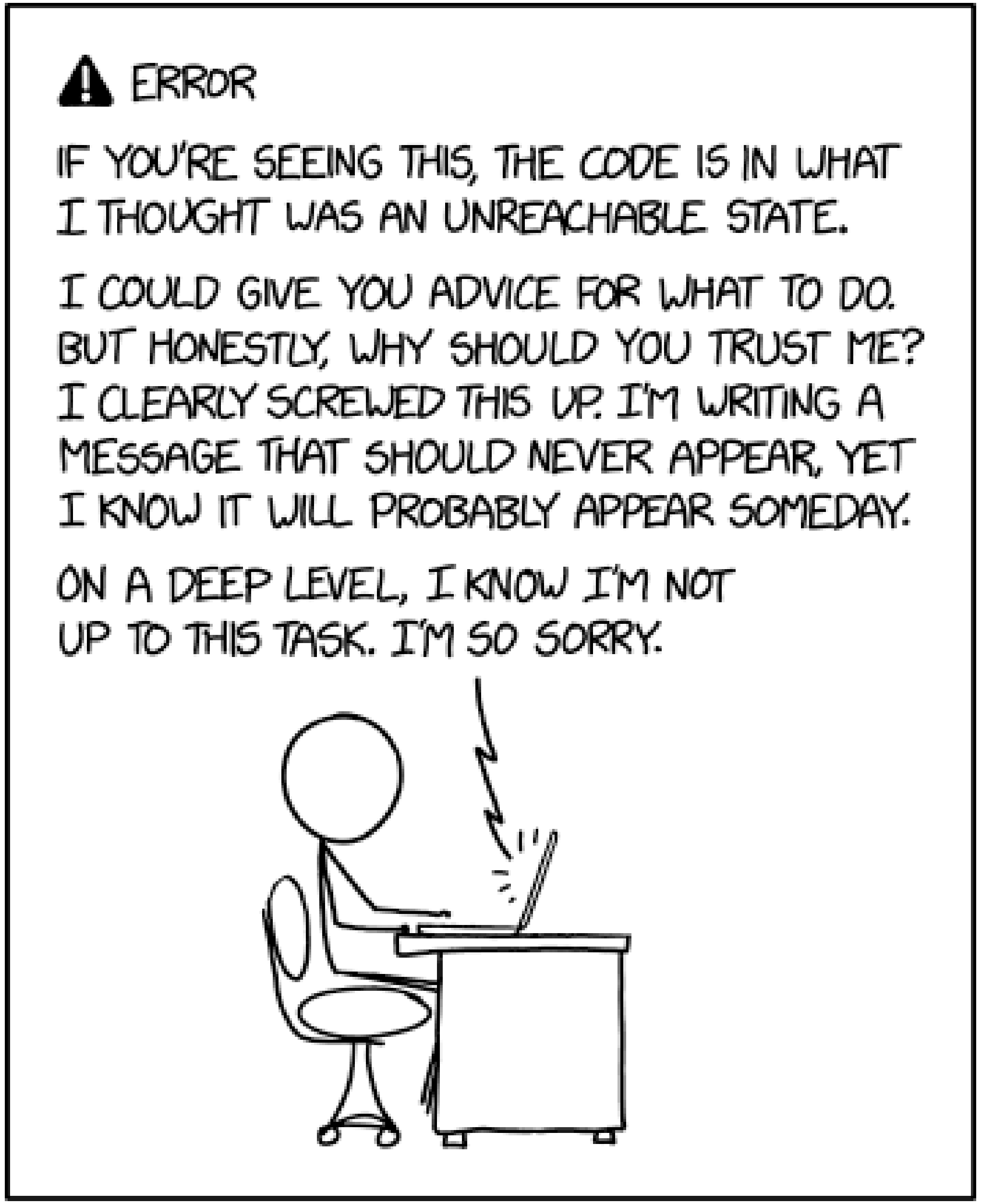
```
try {
  // code that could throw exceptions
} catch ( some specific exception ) {
  // handler code
} catch ( another specific exception ) {
  // handler code
} catch (...) {
  // general case matches all exceptions
  // actually includes the ... in the C++ code
}
```

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



NEVER WRITE ERROR MESSAGES TIRED.

https://xkcd.com/2200/

Outline

- Constructors
- Operator Overloading
- 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
  - But 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

- Constructors
- Operator Overloading
- Exceptions
- Access Control
- **Encapsulation Policy**

Encapsulation

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

- Policy:
  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
  - Don’t need to bother double-checking those assumptions

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

- Warning: does NOT improve security
  - Data can still be accessed, just not by accident

Outline

- Constructors
- Operator Overloading
- Exceptions
- Access Control
- Encapsulation Policy