CS162 Lecture 16

Error and Exception handling
Housekeeping

- Assignment 3 Design Document due tonight at 11:59!
  - Turn in via canvas, follow recitation rubric, same as design doc 1
- Assignment 3 Due Sunday, Nov. 11!
  - Next Sunday!

- **Amnesty Law**
  - You are able to call 911 for emergency response when in the need of medical support.
  - It protects you **and** the person in need from being charged with Minor in Possession (MIP)
- SafeRide Number: 541-737-5000
- Survivor Advocacy and Resource Center (SARC) number: 541-737-2030
Big 3 with Inheritance

Constructors go outside in: call parent constructor in child constructor
- Will use parent ctr to allocate and fill inherited data members
- Only need to deal with the data members unique to this child in its ctr

Destructors go inside out: call child destructor in parent destructor
- Will use child dtor to clean up memory
- Will avoid memory leak where child exists without parent object
Arrays for assignment 3

Recall:

>> int* A = new int[size]

A[0] now points to first element in A

What if size = 0?

A[0] points to a garbage value!
Arrays for assignment 3

Recall:

\[
\text{int* } A = \text{new int[size]}
\]

A[0] now points to first element in A

What if size = 0?

A[0] points to a garbage value!

>> int* A = null

A is still a ptr of type int*! Looks like an empty array, not a garbage value.
Interface Classes

An interface class is a class that has no member variables, and where all of the functions are pure virtual!

- Can have a class with only a .h file!

Interfaces are useful when you want to define the functionality that derived classes must implement, but leave the details of how the derived class implements that functionality entirely up to the derived class.

- Often named beginning with an I
Example Interface Class

class IErrorLog
{
public:
    virtual bool openLog(const char *filename) = 0;
    virtual bool closeLog() = 0;

    virtual bool writeError(const char *errorMessage) = 0;

    virtual ~IErrorLog() {}; // make a virtual destructor in case we delete an
                             // IErrorLog pointer, so the proper derived destructor is
                             // called
};
Why would we make that Interface Class?

- Any class inheriting from IErrorLog must provide implementations for all three functions in order to be instantiated.
- You could derive a class named FileErrorLog, where:
  - `openLog()` opens a file on disk
  - `closeLog()` closes the file
  - `writeError()` writes the message to the file
- You could derive another class called ScreenErrorLog, where:
  - `openLog()` and `closeLog()` do nothing
  - `writeError()` prints the message to the console
Example garbage way of error handling

```cpp
#include <cmath> // for sqrt()

double mySqrt(double value, FileErrorLog &log)
{
    if (value < 0.0)
    {
        log.writeError("Tried to take square root of value less than 0");
        return 0.0;
    }
    else
    
        return sqrt(value);

}
```

If you write your code so it includes FileErrorLog or ScreenErrorLog directly, then you’re effectively stuck using that kind of error log (at least without recoding your program). This method forces callers of mySqrt() to use a FileErrorLog.
Example of error handling with IErrorLog Interface

```cpp
#include <cmath> // for sqrt()

double mySqrt(double value, IErrorLog &log)
{
    if (value < 0.0)
    {
        log.writeError("Tried to take square root of value less than 0");
        return 0.0;
    }
    else
        return sqrt(value);
}
```

Now the caller can pass in any class that conforms to the IErrorLog interface. If they want the error to go to a file, they can pass in an instance of FileErrorLog. If they want it to go to the screen, they can pass in an instance of ScreenErrorLog.
Exception Handling

- Deals with unusual circumstances that may require different reactions
- Not your typical error handling
  - Handling bad input that you can predict will be common; should not use Exception Handling
- Save Exception Handling for edge cases
try-throw-catch

tytry {
  // the code which may encounter an error, but we hope won’t
  throw Exception_parameter; // throw user_input; -> int
  // maybe more code
}

catch (type_of_exception_param e) { // <type_of_param>:<int>
  // code to handle the exception
}
Details

- If no exception is thrown, catch is ignored
- Can catch multiple exceptions, just have more catch blocks
  - Get executed in order of appearance
  - catch(...) catches any type → good default!
- Common to define specialized exception class

class NegativeNumber {
  public:
    NegativeNumber();
    NegativeNumber(string theMessage):message(theMessage);
    string getMessage();
  private:
    string message;
};
Throwing Exceptions in Functions

- Can throw in one function and catch in a different one
- Need to have Exception Specification List
  - Should appear in function declaration and definition
  - If a function has more than one declaration, needs to appear in all of them
  - If more than one exception may be thrown, separate via comma
  - If an exception is thrown in the function but not listed, then unexpected() is called
    - Terminates the program by default

Examples:

```cpp
// treat specified normally, all others unexpected()
void someFunctionA() throw (NegativeNumber, DivideByZero);
// list empty, treat all unexpected();
void someFunctionB() throw();
// none, treat all normally
void someFunctionC();
```
Void functionA() throw (MyException) {
    ...
    Throw MyException(<Maybe an argument>)
    ...
}

Void functionB() {
    Try {
        functionA();
    }
    Catch (MyException e) {
        <Handle exception>
    }
}
```cpp
#include <iostream>
#include <string>
#include <exception> // this is the base class for all other exceptions
#include <stdexcept> // this is where out_of_range is
#include <new> // this is where bad_alloc is

using namespace std;

int main() {
    string s;
    char *str;

    try {
        cout << s[2] << endl; // a segfault or lack of isn't an exception
        cout << s.at(2) << endl; // throws out_of_range exception
        str = new char[10000000000000000];
    }

    catch(bad_alloc &e) {
        cout << "You ran out of memory!" << endl;
    }
    catch(out_of_range &e) {
        cout << "You entered a bad index!" << endl;
    }
    catch(exception &e) {
        cout << e.what() << endl;
    }

    cout << "Program still running!" << endl;

    return 0;
}
```
Next week: Templates

- Assignment 3 Design Document due tonight at 11:59!
  - Follow Recitation instructions; just like design doc #1

- Assignment 3 Due Next Sunday, 11/2!
  - Allowed to use PolyMorphism!
  - Please use polymorphism!
  - You can’t always use it, but you should know when you can’t!
    - Can’t use pure virtuals for classes you need to instantiate
    - Can’t write virtual functions for classes that don’t inherit from the same parent