CS11 Advanced C++

Spring 2018 – Lecture 1

Welcome to CS11 Advanced C++!

- A deeper dive into C++ programming language topics
- Prerequisites:
  - CS11 Intro C++ track is strongly recommended (obvious)
- You should be familiar with:
  - Implementing C++ classes and functions / member-functions
  - Basic use of the C++ standard library - strings, collection templates, stream IO and stream-based file IO
  - Pointers and references (not necessarily rvalue-references)
  - Basic operator overloading
  - Proper use of const keyword
  - Basic exception handling
  - Using tools like make and Doxygen

Assignments and Grading

- Each lecture has a corresponding lab for the material
- Labs are due approximately one week later, at noon
  - e.g. this term labs will be due on Tuesdays at noon
  - Submit on csman
- Labs are given a 0..3 grade, meaning:
  - 3 = excellent (masters all important parts)
  - 2 = good (demonstrates mastery of key idea; a few minor issues)
  - 1 = insufficient (not passing quality; significant bugs must be addressed)
  - 0 = incorrect (worthy of no credit)
- Must receive at least 75% of all possible points to pass track
- Can submit up to 2 reworks of labs to improve grade
- Not uncommon for initial submission to get a 0!
  - Don’t take it personally; it’s really not a big deal in CS11 tracks

Assignments and Grading (2)

- All code submitted is expected to be well documented and written in a clean, uniform coding style
  - Specifics of coding style are not as relevant to me
  - Consistency is the most important thing
- Use Doxygen to generate docs from C++ code
  - **`doxygen -g`** to generate initial **`Doxyfile`** config-file
  - Edit configuration file to generate HTML docs only, into a subdirectory
  - **`doxygen`** should generate docs after that
  - See http://www.stack.nl/~dimitri/doxygen/ for more details
- Initially will use **`make`** for building projects

C++ Compiler

- Can use g++/gdb or clang++/lldb for this track
  - Currently most of the instructions are written for GNU toolset
  - Feel free to contribute LLVM-related info along the way!
- Should be using C++14/17 for your implementations
- Can ask the compiler what version of C++ is its default

  -
    ```
    g++ -dM -E -x c++  /dev/null | grep cplusplus
    #define __cplusplus 201402L
    ```

  -
    ```
    clang++ -dM -E -x c++ /dev/null | grep cplusplus
    #define __cplusplus 199711L
    ```
- Can tell the compiler what version of C++ to use (recommended!)
  - `g++ -std=c++14 ...`
  - `clang++ -std=c++14 ...`
  - Or, in Makefiles (recommended!): `CXXFLAGS = -std=c++14`

Local Variables and Destructors

- Normally, destructors are called when variables go out of scope

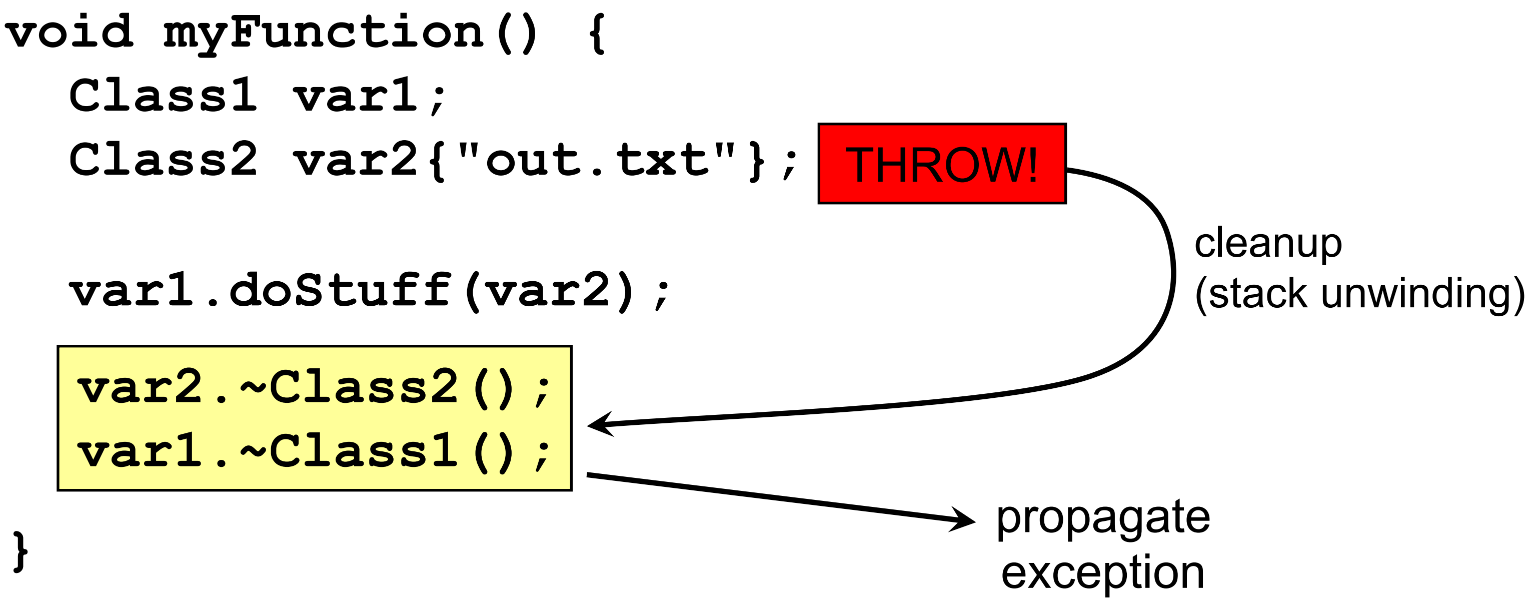
```
void myFunction() {
  Class1 var1;
  Class2 var2{"out.txt"};

  var1.doStuff(var2);
}
```

Compiler adds:

```
var2.~Class2();
var1.~Class1();
```

  - Compiler automatically inserts destructor calls in the appropriate places
  - Your code doesn’t manually call destructors, ever.

Destructors and Exceptions



```
void myFunction() {
  Class1 var1;
  Class2 var2{"out.txt"};

  var1.doStuff(var2);

  var2.~Class2();
  var1.~Class1();
}
```

- What happens if **`var2`** constructor throws?
  - Only **`var1`** was constructed, so only **`var1`** destructor gets called

Destructors and Exceptions (2)

```
void myFunction() {
  Class1 var1;
  Class2 var2("out.txt");

  var1.doStuff(var2);

  var2.~Class2();
  var1.~Class1();

}
```

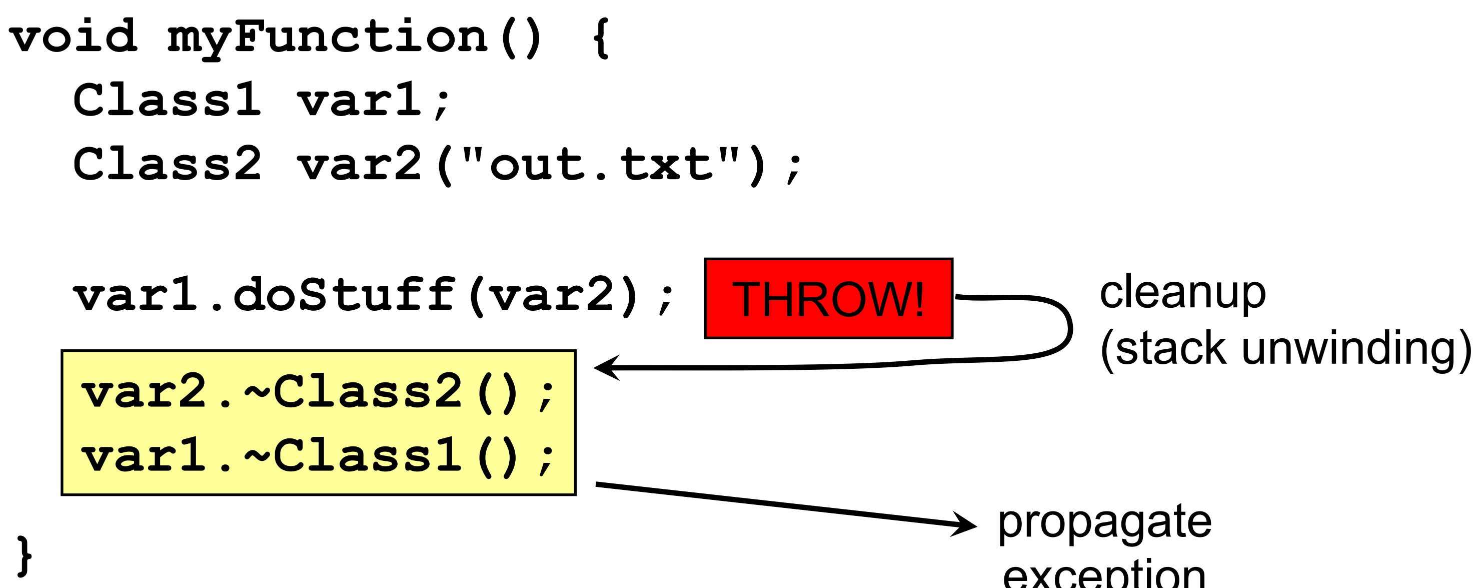
THROW!

no cleanup needed

propagate exception

- What happens if **`var1`** constructor throws?
  - Nothing was constructed, so no destructors get called

Destructors and Exceptions (3)



```
void myFunction() {
  Class1 var1;
  Class2 var2("out.txt");

  var1.doStuff(var2);
   var2.~Class2();
   var1.~Class1();
}
```

- What happens if **`var1.doStuff(var2)`** throws?
  - Both **`var1`** and **`var2`** were constructed, so both destructors get called (in reverse order of construction)

Classes and Exceptions

- Similar model used when constructors throw

```
class Logger {
  LogConfig config;
  RotatingFile outputFile;
public:
  Logger(const string &configFile) {
    ...  // initialize logger   THROW!
  }
  ...
};
```

- What happens if the constructor body throws?
  - The new **`Logger`** instance failed to be constructed
  - **`config`** and **`outputFile`** have already been initialized, so their destructors are automatically called

Classes and Exceptions (2)

- Member initialization might also throw

```
class Logger {
  LogConfig config;
  RotatingFile outputFile;
public:
  Logger(const string &configFile) :
    config{configFile}, outputFile{config} {
    ...  // initialize logger
  }
  ...
};
```

THROW!

- What happens if `outputFile` constructor throws?
  - The new `Logger` instance failed to be constructed (again)
  - `config` was already initialized, so its destructor gets called

Classes and Exceptions (3)

- Another member constructor throws

```
class Logger {
  LogConfig config;
  RotatingFile outputFile;
public:
  Logger(const string &configFile) :
    config{configFile}, outputFile{config} {
THROW!  ...  // initialize logger
  }
  ...
};
```

- What happens if **config** constructor throws?
  - The new **Logger** instance failed to be constructed (yet again)
  - Nothing was initialized, so no member destructors are called

Heap-Allocation and Exceptions

- What if members are heap-allocated?

```
Simulator::Simulator(SimConfig *conf) {
  // Initialize my simulator members.
  entityData = new Entity[conf->maxEntities];
  playerData = new Player[conf->maxPlayers];
}
```

  - If an allocation fails, **`new`** will throw **`bad_alloc`**
- What happens if second allocation throws **`bad_alloc`**?
  - Simple: **`entityData`** doesn't get cleaned up

A Safer Constructor

- Can fix the problem by doing this:

```
Simulator::Simulator(SimConfig *conf) :
  entityData{nullptr}, playerData{nullptr} {
  try {
    entityData = new Entity[conf->maxEntities];
    playerData = new Player[conf->maxPlayers];
  }
  catch (bad_alloc &ba) {
    delete[] entityData;
    delete[] playerData;
    throw;  // Don't forget to propagate this!
  }
}
```

  - Not the prettiest code, but at least it’s safe.

Again and Again!

- This pattern gets old fast:

```
void initSimulation() {
  SimConfig *conf = new SimConfig{"sim.conf"};
  Simulator *sim = new Simulator{conf};
  ...
}
```

  - What if **`Simulator`** constructor throws?
    - (sigh)

- This approach to leak-free, exception-safe code is a giant pain!

Safe Resource Management

- Problem:
  - Dynamic allocation of resources, plus exception-handling, is a potentially dangerous mix!
  - Memory allocated with **`new`**
  - Opening files, pipes, etc.
  - Threads, mutexes, condition variables, semaphores, …

- Just catching exceptions isn’t enough!
- Also need to release any resources that were allocated before the exception was thrown.

Typical Resource Allocation Model

- General form of the problem:

```
void doStuff() {
  // acquire resource 1
  // ...
  // acquire resource N

  // use the resources

  // release resource N
  // ...
  // release resource 1
}
```

  - Resources usually released in opposite order of allocation
- Hey, C++ constructors and destructors do this!
  - Local variables are created and destroyed this way

Simpler Leak-Proofing Approach

- Make a wrapper class for managing a dynamic resource
  - Constructor allocates the dynamic resource, or constructor assumes ownership of the resource
  - Destructor frees the resource
  - Use the wrapper class for <u>local variables</u>
    - i.e. <u>don't</u> do "**`new Wrapper(resource)`**" !!!
- "Clean up" handlers become unnecessary
  - When exception is thrown, C++ calls wrapper-class destructor automatically, since it's local.

Resource Acquisition Is Initialization

- Pattern called: Resource Acquisition Is Initialization (RAII for short)
  - A local variable’s constructor immediately assumes ownership of the dynamic resource
  - C++ will call the destructor at the Right Time.
- Typically realized as “smart pointers”
  - They follow this model for heap-allocated memory
- This can be applied to *any* dynamic resource
  - Files! Semaphores! Mutexes! …

C++ Smart Pointers

- C++11 Standard Library introduced two smart-pointer classes (originally part of Boost library)
- **`std::shared_ptr<T>`** is a *multiple-ownership* smart pointer for when dynamically-allocated memory is shared by multiple parts of a program
  - Manages a **`T*`**, allocated with "**`new T(…)`**"
- **`std::unique_ptr<T>`** is a *single-ownership* smart pointer for when dynamically-allocated memory is used by only one part of a program
  - Also manages a **`T*`**, allocated with "**`new T(…)`**"
- **`#include <memory>`** to use these smart-pointers

C++ Smart Pointers (2)

- **`shared_ptr<T>`** uses reference-counting to track how many parts of the program are using an object
    - Reference-counting imposes a (slight) time and space overhead
- **`unique_ptr<T>`** assumes it is sole owner of the memory
    - No reference-counting overhead
    - Only one **`unique_ptr`** may point to a given **`T*`**
- Use **`unique_ptr<T>`** when only one part of your program needs to access/manage a heap-allocated chunk of memory
    - e.g. local variable within a function, or owned by a single object
- Use **`shared_ptr<T>`** when several parts of your program need to access/manage a heap-allocated chunk of memory
    - e.g. if multiple objects need to manage a single chunk of memory
    - (e.g. if you were implementing a tree or graph data structure)

Smart Pointer Operator Overloads

- Smart pointer classes overload these operators:
  - Dereference (member-access) operator `->`
  - Dereference operator `*` (not multiplication…)
  - Allows the smart-pointer to act like a pointer
- Smart pointers don't support other common pointer operations, e.g. pointer-arithmetic
  - For example: no `++` and `--` operator overloads!
- Cannot use `=` to assign raw pointer to smart pointer
  - Forces thoughtful consideration of how objects are managed
  - e.g. `reset(T*)` to assign a raw pointer to a smart pointer, or `reset()` to set a smart pointer to `nullptr`
  - Can assign a smart pointer value to another smart pointer

Usage Examples

- Example:

```
#include <memory>

shared_ptr<Widget> spw{new Widget{}};
unique_ptr<Widget> upw{new Widget{}};
```

- Can create an alias for the shared-pointer type:

```
using sp_widget_t = shared_ptr<Widget>;
using up_widget_t = unique_ptr<Widget>;

sp_widget_t spw{new Widget{}};
up_widget_t upw{new Widget{}};
```

Usage Examples (2)

- C++ also provides two helper functions to wrap initialization

```
#include <memory>

sp_widget_t spw = make_shared<Widget>{};
up_widget_t upw = make_unique<Widget>{}
```

- These are actually variadic function templates that call the specified class' constructor
  - Can pass arguments to the class' constructor as well

Usage Examples (3)

- Example:

```
class Widget {
    ...
public:
    Widget();
    Widget(const string &type, float weight);
    ...
};

spw1 = make_shared<Widget>{};
spw2 = make_shared<Widget>{"frob", 4.3};
```

- Heap-allocates two **`Widget`** objects, and manages them with smart pointers
  - Second **`Widget`** initialized with two-arg constructor

Smart Pointers and Destructors

- With proper use of smart pointers and **`make_shared`**/**`make_unique`**, should never need to directly use **`new`** or **`delete`** in your code
- Destructors also become unnecessary (!!!)
  - If class wraps all allocations with smart-pointer members, they will automatically delete allocations at the right time
- This affects the Rule of Three / Rule of Five
  - If your class defines any of the following:
    - A destructor, a copy-constructor, a copy-assignment operator
    - [Rule of Five: A move-constructor, a move-assignment operator]
  - It probably needs to define all three [all five].
  - Destructor will become unnecessary, but other important member functions are likely still essential

Smart Pointers and Destructors (2)

- Can either leave out the destructor (the compiler will generate a default for you)
- Or, can specify you are using the default destructor

```
class MyClass {
    MyClass() { ... }
    ~MyClass() = default;
    ...
};
```

- Second option will explicitly document that your class doesn’t require a custom destructor

This Week's Assignment

- This week:
  - Build a **`TreeSet`**, an ordered set implementation, to store **`int`** values
- Use a binary search tree for internal representation
  - Each node stores a value V
  - A node's left subtree holds values less than V
  - A node's right subtree holds values greater than V
  - Tree <u>does not</u> need to be kept balanced
- Encapsulate all implementation details!
  - Class should not expose that it uses a tree internally
- Use smart pointers everywhere in implementation

This Week’s Assignment (2)

- When writing data structures in C++, must frequently decide between using smart pointers and raw pointers
- For particularly complicated structures (e.g. graphs), raw pointers may in fact be better than smart pointers
  - Smart pointers can’t deal with cycles in reference graph due to using reference-counting internally
  - Updating reference-counts in shared pointers is a cost that will add up over time
- A binary search tree is simple enough to use shared pointers without increasing complexity too much
  - A good opportunity to practice using smart pointers ☺
- Moral: Always pick the best tool for the job!