CS 246 Fall 2015 - Tutorial 12

December 3, 2015

1 Summary

- Exceptions
- Resource Acquisition is Initialization (RAII)

2 Exceptions

- Recall that traditional exception handling mechanisms (return codes, global status flags (`errno`), fix-up routines) have a fatal flaw
  - They can all be ignored
  - This implies a client does not need to deal with a potential error, which can cause bad things to happen
  - We want exceptional situations to *require* a response or risk facing termination
  - A client should be *proactive* and not *reactive*
- C++ Exceptions provide a mechanism that requires immediate attention or your program will terminate
- Recall the basic format for using exceptions in C++:

```
int main(){
  try{
    ...
    throw 42;
    ...

  }
  catch (int e){ ... }
  catch (char p){ ... }
  catch (bad_alloc e) { ... }
  catch (...) { ... } // Catch anything
};
```

- Recall, that **anything** in C++ can be thrown as an exception
- This implies that there is no *required* overarching Exception base class (like in Java)
- Accordingly, the `catch(...)` syntax is used to catch anything that is thrown while ignoring what was thrown.
  - We have no way to know what was thrown.
  - Typically, this is used to clean up (e.g. delete memory, write log messages, etc)
  - After which we **rethrow** the exception to be handled by some other handler higher up the stack
- There are three guarantees that you can provide with respect to exception safety:
  - **basic** guarantee: if an exception is thrown, data will be in a valid state but may not make sense
    - e.g., if we change variables in an assignment operator before allocating heap memory

  - **strong** guarantee: if an exception is thrown, the data will appear as if nothing happened
    * The copy-and-swap idiom provides the strong guarantee
  - **nothrow** guarantee: an exception is never thrown
    * Swapping two pointers using std::swap is guaranteed not to throw an exception

- While C++ does not enforce an exception hierarchy, it does provide one (which the Standard Library uses)
  - Can be included with

    ```
    #include <exception>
    ```
  - `exception` is the base class of the C++ exception hierarchy (and has a virtual what method, that specifies what the exception is)
  - Common derived exceptions include:
    * `bad_alloc`: is thrown when `new` fails
    * `bad_cast`: is thrown when `dynamic_cast` fails (only when casting to a reference)
    * `out_of_range`: is thrown when a `vector`, `string`, etc, have an element accessed that is out of range

- We can define our own exception classes, either as part of the C++ exception hierarchy or as part of our own

- Accordingly, the order in which handle exceptions matters (see `exception-order.cpp`)

- We might run into some other surprising issues with exceptions. For example:

```
struct myexception{
  virtual string toString(){ return "myexception";}
  virtual ~myexception(){}

};
struct otherexception : public myexception{
  string contents;
  otherexception(string msg) : contents(msg){}
  string toString(){ return contents;}
  ~otherexception(){}
};
int main(){
  try{
    throw otherexception("Foobar");
  } catch (myexception e){
    cout << "Caught: " << e.toString() << endl;
  }
}
```

- Why doesn't this program work the way we expect?

- What's the fix for it?

3 Resource Acquisition is Initialization (RAII)

- RAII is vital to writing exception-safe code in C++

- RAII relies on the guarantee that when an exception is thrown, destructors for stack-allocated objects will be called

- Resources are acquired during initialization (e.g. in a constructor), so that they cannot be used before they are available, and are released when the owning object is destroyed

- However, pointers and dynamic memory pose a problem for us. The pointer is deallocated but the memory (possibly an enormous object) is not.

- `auto_ptr` is a templated type that behaves exactly like a pointer, except that when it is destroyed it calls `delete` on its pointed to memory
- Note that only one `auto_ptr` can ever point to the same memory location (e.g. assignment transfers ownership)
- In addition, `auto_ptr` only calls `delete`, and never `delete []` or `free`
- `auto_ptr` is included in the `<memory>` library
- Let's see an example:

```
#include <memory>
#include <iostream>
using namespace std;
int main(){
  auto_ptr<int> ap(new int);
  *ap = 7;
  cout << *ap << endl;
  // get() returns the pointer being stored
  cout << ap.get() << endl;
  auto_ptr<int> ap2 = ap;
  cout << ap.get() << endl;
  cout << ap2.get() << endl;
}
```

- `auto_ptr` is quite limited in its use as we've seen. We might like something more robust
- In C++11, `auto_ptr` is deprecated and replaced by `shared_ptr`, `unique_ptr`, and `weak_ptr`
  - `shared_pointer` allows many pointers to the same block of memory and only deletes that memory when no other `shared_ptr`s point to it
  - `unique_ptr` is similar to `auto_ptr` but supports more functionality (e.g. operator[])
  - `weak_ptr` is like `shared_ptr` but doesn't count towards the "shared count;" it is used to prevent cyclic ownership in `shared_ptr`s