CS 246 Fall 2013 - Tutorial 10

November 27, 2013

1 Summary

- Exceptions
- Casting
- Resource Acquisition is Initialization (RAII)

2 Exceptions

- Recall that traditional exception handling mechanisms (return codes, status flags (`errno`), fix-up routines) have a fatal flaw
  - They can all be ignored
  - This implies a user does not need to deal with a potential error, which can cause bad things to happen
- 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 overarching Exception base class (like in Java)
- Accordingly, the `catch(...)` syntax is used to catch anything that is thrown.
  - However, we have no way to know what it is.
  - 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 catch higher up the stack
- How do we **rethrow** an exception?

```
try{
  ...
} catch (int e){
  // Do stuff
  throw;
} catch ( ... ){
  // Do stuff
  throw;
}
```

- While C++ does not enforce an exception hierarchy, it does provide an exception hierarchy (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 catch exceptions matters (see `exception-order.cpp`)

```
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 Casting

- There are 4 types of casts in C++, 3 of which are common and 1 of which should be used sparingly
  1. `const_cast`: used to add or remove const from its parameter
  2. `dynamic_cast`: used for pointers and references, ensures result of cast is actually of appropriate type
  3. `static_cast`: convert between related classes (base to derived, and derived to base). Can also perform implicit conversions (int to double).
     - Note: `static_cast` is not checked (and is equivalent to C-style casts).
  4. `reinterpret_cast`: cast anything to anything else (EXTREMELY unsafe - should be used sparingly)
- Typically, `const_cast` this is handy when we want to call C functions from C++
- Recall, when we `dynamic_cast` a pointer, then if it succeds the pointer is not NULL (0) and NULL otherwise
- `dynamic_cast` should be used sparingly because it is expensive (e.g. have to constantly check runtime type)
- Generally, it is bad style to have code that looks like:

```
int foo(Base* bp){
  if(! dynamic_cast<Derived1*>(bp)){
    ...
  } else if (! dynamic_cast<Derived2*>(bp)){
    ...
  } else if (! dynamic_cast<Derived3*>(bp)){
    ...
  } ...
}
```

- It would be smarter to create three separate functions with different parameters (e.g. reference parameters to each type)
- Also, recall from our discussions of exceptions that `bad_cast` can be thrown when `dynamic_cast` fails

```
try{
  Base& br = ...; // Something
  Derived1& dr = dynamic_cast<Derived1&>(br);
}catch (bad_cast& e){
  cerr << "DANGER WILL ROBINSON DANGER" << endl;
}
```

- Why **must** `dynamic_cast` throw an exception here?
- Finally, we have to be careful not to shoot ourselves in the foot with `static_cast`:

```
Base* bp = new Derived2;
Derived1* d1p = static_cast<Derived1*>(bp);
```

- This will compile but it may not work as we intend or expect.

4 Resource Acquisition is Initialization (RAII)

- RAII is vital to writing exception-safe code in C++
- RAII requires that the only code that is executed when an exception is thrown are destructors
- 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 (potentially an 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 it's 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 by:

```
#include <memory>
```

- 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;
}
```