Advanced C++

January 29, 2015

Mike Spertus

mike_spertus@symantec.com

YIM: spertus

Function-static lifetimes

- A static variable in a function is initialized the first time the function runs
  - Even if the function is called from multiple threads, the language is responsible for making sure it gets initialized exactly once.
  - If the function is never called, the object is never initialized
  - As usual, static duration objects are destroyed in the reverse order in which they are created

Singleton implementation

```
struct A {
  static A *instance() {
    static A ins;
    return &ins;
  }
  int i;
private:
  A() : i(7) {} // No one else can construct
  A(A const &) = delete; // or copy
};
```

Dynamic storage duration

- Generally created by expression of the form "`new` *`typename`*" or "`new` *`typename(constructor args)`*"
  - Returns a properly-typed pointer to the memory
  - `int *ip = new int;`
  - `A *ap = new A(7, x);`
  - `A *arr = new A[7]; // Creates an array`
- Destroyed by calling `delete`
  - `delete ip;`
  - `delete ap;`
  - `delete [] arr; // Deletes an array`

Exceptions

- Can throw an exception (any type) with `throw`
- You can catch an exception within a try block with `catch`.
- Exceptions make memory management very difficult because program flow is hard to predict

Example

```
#include <iostream>
using namespace std;
int main () {
  try {
   throw 20;
  } catch (int e) {
    cout << "Exception " << e << endl;
 }
  return 0;
}
```

Pointers

- Pointers to a type contain the address of an object of the given type.

```
A *ap = new A;
```

- Dereference with `*`

```
A a = *ap;
```

- `->` is an abbreviation for `(*_).`

```
ap->foo(); // Same as (*ap).foo()
```

- If a pointer is not pointing to any object, you should make sure it is nullptr (If not yet in C++11, use 0)

```
ap = nullptr; // don't point at anything
if(ap) { ap->foo(); }
```

Memory leak

```
#include <iostream>
using namespace std;
int f() {
  try {
   A *ap = new A;
   throw 20;
   delete ap; // Never called
  } catch (int e) {
    cout << "Exception " << endl;
 }
  return 0;
}
int main() { for(int = 0; i < 1<<20; i++) f(); }b
```

Tear down

- Objects of automatic storage duration are destroyed as you leave the try block
- Exceptions filter upward to calling functions destroying objects of automatic storage duration as each block scope is left
- This explains why there is no “finally” in C++
  - RAII

Memory leak fixed

```
#include <iostream>
using namespace std;
int f() {
  try {
   unique_ptr<A> ap{new A};
   if(/* error occurs */)
     throw 20;
} catch (int e) {
    cout << "Exception " << endl;
 }
  return 0;
}
int main() { for(int = 0; i < 1<<20; i++) f(); }b
```

Potential memory leak

```
void f()
{
  // g is responsible for deleting
  g(new A(), new A());
}
```

- What if the second time A's constructor is called, an exception is thrown?
- The first one will be leaked

Solution by RAII

```
void f()
{
   unique_ptr<A> arg1(new A());
   unique_ptr<A> arg2(new A());
  g(arg1.release(), arg2.release());
}
```

- Best practice, all heap objects should be owned by a smart pointer

References

- Like pointers but different
  - Allow one object to be shared among different variables
  - Can only be set on creation and never changed
    - Reference members must be initialized in initializer lists

```
struct A {
  A(int &i) : j(i) {}
  int &j;
};
```

  - Cannot be null

Understanding function and method arguments

- Function and method signatures are very complicated
  - Arguments can be passed by value or reference
  - Overloading can make it tricky to know which function will be called
  - Template instantiation rules construct signatures on the fly

Passing arguments by value or reference

- Pass by value

```
void v(int i) { i = 7; }
int x = 3;
v(x);   // v gets its own copy of i
cout << x; // Prints 3
v(3); // OK. Doesn't try to change the value of 3
```

- Pass by reference

```
void r(int &i) { i = 7; }
int x = 3;
r(x); // r "binds" i to the existing x
cout << x; // Prints 7
r(3); // Error! Can't change 3
void c(int const &i); // Won't modify i
c(3); // OK. Doesn't modify 3
```

Function overloading

- The basics
  - Create list of candidate functions
  - Choose a fit that is best on each argument

Example

- Suppose we have

```
struct A { A(char c) {} };
void f(int, double) { cout << "fid"; }
void f(int, int) { cout << "fii"; }
void f(A, double) { cout << "fad"; }
```

- What do we get for each of these?

```
f(7, 7);          // fii (Nothing is better on any arg)
f(7.1, 7.1);      // fid (Nothing is better on any arg)
f(7.1f, 7.1f);    // Error. Ambiguous: fii, fid equally bad
f('a', 7.1);      // fid (built-in char->int beats char->A
```

Template candidate functions

- What template candidate functions are chosen?
  - Each argument is used to infer the template parameters
  - No automatic type conversions are allowed

```
T const &min(T const &x, T const &y)
  { return x < y ? x : y; }
```

- `min(3, 4)` infers `T` is `int`
- For `min(3, 4.5)`, the first argument suggests that `T` is `int`, but the second argument implies `T` is `double`. Ambiguous!

Explicit function template arguments

- We can specify by giving the template arguments explicitly: `min<double>(3, 4.5)`
- This is also useful for places where functions aren't so clear. For example, to take the min of all the elements of a vector, you can use:

```
accumulate
  (v.begin(),
   v.end(),
   numeric_limits<double>::max(),
   min<double>)
```

More on template overload resolution

- Sometimes surprising results:
- What does the following output?

```
double *dp = { 0.1, 0.2, 0.3 }
cout << accumulate(dp, dp + 3, 0);
```

- Answer: 0!

```
template<class _InIt, class _Ty> inline
  _Ty _Accumulate(_InIt _First, _InIt _Last, _Ty _Val)
```

- This implies that `_Ty` is `int`.
- Correct: `accumulate(dp, dp + 3, 0.0);`

Order of argument evaluation

```
int f(int x, int y)
  { return x * y * y; }
int i = 3;
```

- What is `f(i++, i++)`?
- Answer: Undefined!

Undefined vs. Implementation-defined

- Implementation-defined behavior is defined (Section 1.3.5) as "behavior, for a well-formed program construct and correct data, that depends on the implementation and that each implementation shall document.“
- By contrast for undefined behavior (1.3.12), the "...standard imposes no requirement." This is scary because it means your program might work during testing and not fail until you have a million copies in the field when some small C++ run-time patch is pushed out by your compiler vendor and the order gets changed.

Rule of three

- A class should define all or neither of the following
  - Destructor
  - Copy constructor
  - Assignment operator
- http://en.wikipedia.org/wiki/Rule_of_three_%28C%2B%2B_programming%29
- http://www.drdobbs.com/c-made-easier-the-rule-of-three/184401400

MOVE SEMANTICS

Rvalue references

- A reference with “&&” instead of just “&” can bind to a temporary and move it elsewhere.
- Objects are often much cheaper to “move” than copy
- ```
  template<class T>
  void swap(T& a, T& b)// "perfect swap“(almost)
  {
    T tmp = move(a); // could invalidate a
    a = move(b); // could invalidate b
    b = move(tmp); // could invalidate tmp
  }
  ```
  ```

Move semantics example: putting threads into an array

- Recall that std::unique_ptrs are not copyable
- Since they are movable, we can construct a temporary unique_ptr and move it into a vector
- ```
  template<typename T> class vector {
    ...
    push_back(T const &t);
    push_back(T &&t);
    ...
  };
  ```
- ```
  vector<unique_ptr<int>> vt;
  for(int i = 0; i < 10; i++) {
    vt.push_back(unique_ptr<int>(new int(i)));
  }
  ```

“Rvalue reference references”

- Here are some useful references on rvalue references
- http://thbecker.net/articles/rvalue_references/section_01.html
  - What I lectured from in class
- http://blogs.msdn.com/b/vcblog/archive/2009/02/03/rvalue-references-c-0x-features-in-vc10-part-2.aspx
- http://www2.research.att.com/~bs/C++0xFAQ.html#rval

How do I make a type movable?

- ```
  template<class T> class vector {
    // ...
    vector(vector<T> const &); // copy constructor
    vector(vector<T> &&); // move constructor
    vector& operator=(const vector<T>&); // copy
  assignment
    vector& operator=(vector<T>&&); // move assignment };
  // note: move constructor and move assignment takes
  // non-const && they can, and usually do, write to
  // their argument
  ```
- In, C++11 all containers have move constructors, and versions of insert, push_back, etc. taking rvalue references, improving performance because they copy less
- Move constructors also allow a “non-broken auto_ptr” called unique_ptr that can be stored in an STL container and more efficient return managed objects
  ```

How is std::move implemented? (Very advanced)

- First, we need to understand the rules for collapsing rvalue references to template parameters only
- T& & ≅T&
- T& && ≅T&
- T&& & ≅T&
- T&& && ≅T&&

std::move code

- ```
  template <class T>
  typename remove_reference<T>::type&&
  move(T&& a)
  { return a; }
  ```
- What happens in the code
  ```
  A a;
  f(move(a)); // calls f(A &&)
  ```
- For what `T` is `T&&` an `A` or `A&`?
- By the collapsing rules, we see that the only option is that `T ≅ A&. (T && ≅ A& && ≅ A&)`
- Now, we return a `remove_reference<A&>::type&& ≅ A&&`
- If you are interested, you can check that all the other cases work
  ```

Rule of five?

- There is a lot of discussion about whether the rule of 3 should be extended to a “rule of 5,”
  - If you define any of
    - The destructor
    - The copy constructor
    - Copy assignment operator
    - Move constructor
    - Move assignment operator
  - You should probably assign them all
- C++11 deprecated some features to better mesh with the rule of 5
  - A proposal (with history) to remove the deprecated features was rejected for C++14. Even though it was rejected it makes interesting and illuminating reading for aspiring language lawyers
    - http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3578.pdf
    - Warning: This paper is hard-core. Only recommended if you have substantial C++ experience

HW 4.1

The following function tries to ensure cout is flushed before leaving:

```
int f() {
  cout << "Some text";
  g(); // g and h are functions whose
  cout << h(); // definitions are unknown
  cout.flush();
  return 0;
}
```

Is this code correct (i.e., is it guaranteed that cout will be flushed)? If not, how would you fix it?

Extra credit: When I originally posted this slide, I inadvertently gave the third line of `f()` as "`cout << f()`", which seems to result in an infinite recursion where `f` calls itself indefinitely (until a stack overflow occurs). In the original version, is it possible that `f()` will ever complete or is it guaranteed to recur forever?

HW 4.2

- Are the following delete statements correct? If not, tell why not and fix the code

```
.
int main()
{
    int i;
    int *ip = new int[10];
    delete &i;
    delete ip;
}
```

HW 4-3

- Combining functors with the standard library is very powerful, but sometimes gives unexpected results.
- The following code (next slide) to find the maximum length of a collection of strings unexpectedly always returns 0. Why doesn’t it work? How can you fix it?
  - Looking up the documentation of for_each may suggest possible solutions

HW 4.3 (Code)

```
#include<algorithm>
#include<iostream>
#include<string>
#include<vector>
using namespace std;

struct maxlenftn {
   maxlenftn() { maxlen = 0; }
    void operator()(string s) {
         maxlen = max(maxlen,s.size());
   }
   string::size_type maxlen;
};

int main() {
   vector<string> names{"Spertus", "Lemon", "Golden", "Melhus"};
   maxlenftn maxf;
   for_each(names.begin(),names.end(),maxf);
   cout << maxf.maxlen << endl;
    return 0;
}
```

HW 4-4

- Modify the binary tree class at http://www.cprogramming.com/tutorial/lesson18.html to be movable
- For extra credit, improve the class in other ways