Managing Object Lifetimes

Marshall Clow
Qualcomm
marshall@idio.com
mclow@boost.org
http://cplusplusmusings.wordpress.com
(intermittent)

C++Now 2014

May 2014

Object Lifetimes

- One of the least appreciated features of C++
- Scope based
- Deterministic
  - Even in the case of exceptions (even from constructors)
- Temporary values, too

Object Lifetimes (2)

- Gaby alluded to this in his keynote on Tuesday. (Sean and Eric as well)
- A constructor turns memory into an object
- A destructor turns an object into memory
- An object's lifetime starts when the constructor completes, and ends when the destructor begins.

Bad code

```
class Fancy;
Fancy* deserialize ( void *ptr, size_t size );

Fancy* read_from_disk( const char *filename ) {
    Fancy *ret_val = NULL;
    FILE *f = fopen ( filename, “rb” );
    if (f) {
        size_t sz = file_size(f);
        void *p = malloc(sz);
        if (p) {
            fread(p, 1, sz, f);
            ret_val = deserialize(p, sz);
            free(p);
            }
        }
    return ret_val;
    }
```

We can fix this

```
class Fancy;
Fancy* deserialize ( void *ptr, size_t size );

Fancy* read_from_disk( const char *filename ) {
    Fancy *ret_val = NULL;
    FILE *f = fopen ( filename, “rb” );
    if (f) {
        size_t sz = file_size(f);
        void *p = malloc(sz);
        if (p) {
            fread(p, 1, sz, f);
            ret_val = deserialize(p, sz);
            free(p);
            }
        fclose(f);
        }
    return ret_val;
    }
```

C++ gives us the tools to do better

- Constructors and destructors are run automatically
- Even in the case of exceptions

RAII

- The second-worst acronym in C++
- It stands for “Resource Acquisition is Initialization”

Examples in the standard library

- all the smart pointers (auto, unique, shared, weak)
- lock a mutex (unique_lock, shared_lock, etc)
- many others

Better (safer) code

```
typedef unique_ptr<FILE,int(*)(FILE*)> upfile_t;

Fancy* read_from_disk1( const char *filename ) {
    upfile_t fp(fopen(filename, "rb"), fclose);
    if (fp) {
        size_t sz = file_size(fp.get());
        unique_ptr<char[]>
                    p(new (nothrow) char[sz]);
        if (p) {
            fread(p.get(), 1, sz, fp.get());
            return deserialize(p.get(), sz);
            }
        }
    return NULL;
    }
```

Different code

```
typedef unique_ptr<FILE,int(*)(FILE*)> upfile_t;
upfile_t F_OPEN (const char *fn, const char *mode) {
   FILE *f = fopen ( fn, mode );
   if (!f)
       throw runtime_error("Can't open file");
   return upfile_t (f, fclose);
  }

Fancy* read_from_disk4( const char *filename ) {
    auto f = F_OPEN(filename, "rb");
    size_t sz = file_size(f.get());
    unique_ptr<char[]> p(new char[sz]);
    fread(p.get(), 1, sz, f.get());
    return deserialize(p.get(), sz);
    }
```

A different approach

```
Fancy* read_from_disk2( const char *filename ) {
    ifstream ifs(filename, ios::binary);
    if (ifs) {
      std::vector<char> v;
      copy(istream_iterator<char>(ifs),
          istream_iterator<char>(),
          back_inserter(v));
      return deserialize(v.data(), v.size());
      }
    return NULL;
    }
```

Other advantages

- Exception safety
- Easy to reason about
- Easy to review

“Error handling is left as an exercise for the reader”

- Error detection should be automatic
- Error handling should be easy.
- Error recovery should be automatic (in many cases)
- Boost.Exception makes layered error handling possible/easy.

Incremental use of RAII

```
Fancy * fancy_factory ( int ct, const char *xx ) {
    unique_ptr<Fancy> ret (new Fancy(ct));
    // ... a bunch of code
    ret->method(xx);
    // .. more code
    if (some_error)
        return NULL;
    // .. maybe more code
    return ret.release();
    }
```

Examples of RAII

Boost.ScopeExit

- Written by Alexander Nasonov
- In boost since 1.38
- uses RAII technique to run arbitrary code at scope exit.
- http://www.boost.org/doc/libs/1_55_0/libs/scope_exit/doc/html/index.html

ScopeExit Example

```
void world::add_person(person const& a_person) {
    bool commit = false;

    persons_.push_back(a_person);

//  Following block is executed when the enclosing scope exits.
    BOOST_SCOPE_EXIT(&commit, &persons_) {
        if(!commit) persons_.pop_back();  // rollback action
    } BOOST_SCOPE_EXIT_END

//  ....
//  other operations

    commit = true;  //disable rollback actions
} // scope_exit code runs here...
```

Nitrogen

- A library for Mac OS Carbon by Lisa Lippincott.
- Wrapped all of the Carbon calls
  - Threw exceptions on errors
  - All resources were returned in “owned” objects.
- Writing code using Nitrogen was *wonderful*

Now for something different: Passing parameters

Parameter passing and smart pointers

- There's a lot of advice around about passing smart pointers around.
- Some of this really strange.
  - Passing shared_ptr<Foo> by const &.

Eric and Sean stole my thunder

- Guideline: Don’t pass parameters as smart pointers.
  - That decreases generality - adds coupling
- There are obvious exceptions to this
  - Routines that consume the smart ptr
  - Routines that keep a copy of the smart ptr.

Passing pointers vs. references

- Guideline: Pass optional parameters by pointer, all others by value or reference.
  - See Wednesday's talks for advice on non-pointer parameters

Questions?