C++

January 29, 2013

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

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;
}
```

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

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

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 0

```
ap = 0; // don't point at anything
if(ap != 0) { ap->foo(); }
```

Pointers to members

- ```
  struct A {
    int i;
    int j;
   void foo(double);
   void bar(double);
  };
  ```
- We would like to be able to point to a particular member of A
  - Not an address because we haven't specified an A object
  - More like an offset into A objects
- ```
  int A::*aip = &A::i;
  void (A::*afp)(double) = &A::foo;
  A *ap = new A;
  A a;
  ap->*aip = 3; // Set ap->i to 3
  (a.*afp)(3.141592); // Calls a.foo(3.141592)
  ```

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

Definitions and declarations

- In C++, it's important to understand the difference between definitions and declarations. A declaration just tells how something is used, where a definition defines it (allocates storage for variables, gives the code for functions, lists the members for classes). In general (there are a few important exceptions), a declaration needs to be seen before any use of a symbol and is generally given in a .h file, while a definition is only provided once per program, generally in a .cpp file that is part of your program (Exception: class definitions are given in header files without worrying about the once per program rule).

Definitions and declarations

- For example (note that the rules for what is declaration vs. definition are not that consistent),

```
extern int i; // declaration (an int i will be defined somewhere in the
program)
int i = 5; // definition
int j;  // definition (this really creates j)

class A;  // A will be a class but we don't know anything about it
A *ap; // Legal since we know A is a class
class A { ... }; // The definition of A

class A {
public:
  int foo(); // declaration of foo method
  int i; // definition of i because every A object contains storage for i
  static double d; // declaration because static storage
                   //can only be defined once in a program.
};
```

In a separate .cpp file, you' provide definitions

```
int A::foo() { return 5; }
double A::d;
```

Definitions and declarations

- Some exceptions
  - Defining a static member with a constant integral type expression in the class body
    ```
    struct X { static const int a = 76; }; //OK
    ```
  - Templates should be defined in .h files. Linker must merge
  - Inline methods in .h files
  - non-global statics in .h file (because they are not shared between translation units)
    ```
    extern int i; // Only declare global in .h
    static int j; // Each translation unit
                  // has their own j
    ```

C++11 THREADS

Overview

- Perhaps the biggest addition to C++11 is support for standardized concurrency
  - Multithreading to run tasks in a process in parallel with each other
  - Synchronization primitives and memory model to allow different threads to safely work with the same data

Status

- If you use recent compiler versions, like g++ 4.6 or newer, of Visual Studio 2012, compiler support should at least be good enough for this course
- If you need to use an older compiler, Anthony William's just::thread library provides C++11 thread emulation for older compilers
  - We have negotiated a reduced rate for students of this course. Contact me for info.

References

- C++ Concurrency in Action Book
  - http://www.manning.com/williams/
    - If you buy from Manning rather than Amazon, you can download a preprint right now without waiting for the official publication
  - The author Anthony Williams is one of the lead architects of C++11 threads, the maintainer of Boost::Thread, and the author of just::thread
- Anthony's Multithreading in C++0x blog
  - http://www.justsoftwaresolutions.co.uk/threading/multithreading-in-c++0x-part-1-starting-threads.html
  - Free with concise coverage of all the main constructs
- The standard, of course
    - Also look at the papers on the WG21 site

THE BASICS

Hello, threads

```
#include <iostream>
#include <thread>

void hello_threads() {
    std::cout<<"Hello Concurrent World\n";
}

int
main(){
    // Print in a different thread
    std::thread t(hello_threads);
    t.join(); // Wait for that thread to complete
}
```

What happened?

- Constructing an object of type std::thread immediately launches a new thread, running the function given as a constructor argument (in this case, hello_threads).
  - We’ll talk about passing arguments to the thread function in a bit.
- Joining on the thread, waits until the thread completes
  - Be sure to join all of your threads before ending the program
  - Exception: Later we will discuss detached threads, which don’t need to be joined

Locks

- The simplest way to protect shared data is with a `std::mutex`.
- Because we want to make sure we release the mutex when we are done no matter what, we should use RAII rather than manually releasing the lock
- C++11 includes a handy RAII class `std::lock_guard` for just this purpose.

Locks

```
#include <list>
#include <mutex>
#include <algorithm>

std::list<int> some_list; // A data structure accessed by multiple threads
std::mutex some_mutex;  // This lock will prevent concurrent access to the shared data structure

void
add_to_list(int new_value)
{
    std::lock_guard<std::mutex> guard(some_mutex); // Since I am going to access the shared data struct, acquire the lock
    some_list.push_back(new_value); // Now it is safe to use some_list. RAII automatically releases lock at end of function
}

bool
list_contains(int value_to_find)
{
    std::lock_guard<std::mutex> guard(some_mutex); // Must get  lock every time I access some_list
    return
        std::find
          (some_list.begin(),some_list.end(),value_to_find)
          != some_list.end();
}
```

Not so basic: Thread arguments

- You can add arguments to be passed to the new thread when you construct the `std::thread` object as in the next slide
- But there are some surprising and important gotchas that make passing arguments to thread function different from passing arguments to ordinary functions, so read on

Passing arguments to a thread

```
#include <iostream>
#include <thread>
#include <string>
#include <vector>
#include <mutex>
using namespace std;
mutex io_mutex;

void hello(string name) {
    lock_guard<mutex> guard(io_mutex);
    cout <<"Hello, " << name << endl;
}

int
main(){ // No parens after thread function name:
    vector<string> names = { "John", "Paul"};
    vector<thread> threads;
    for(auto it = names.begin(), it != names.end(); it++) {
        threads.push_back(thread(hello, *it));
    }
    for(auto it = threads.begin(), it != threads.end(); it++) {
        it->join();
    }
}
```

Deceptively simple

- A different notation is used from arbitrary function calls, but otherwise fairly straightforward looking:
  - ```
    void f(int i);
    f(7); // Ordinary call
    thread(f, 7);// f used as a thread function
    ```
    ```

Gotcha: Passing pointers and references

- Be very careful about passing pointers or references to local variables into thread functions unless you are sure the local variables won't go away during thread execution
- Example (based on Boehm)

```
void f() {
  int i;
  thread t(h, &i);
  bar(); // What if bar throws an exception?
  t.join(); // This join is skipped
} // h keeps running with a pointer
  // to a variable that no longer exists
  // Undefined (but certainly bad) behavior
```

- Use try/catch or better yet, a RAII class that joins like the `thread_guard` class in *Concurrency In Action* book

Gotcha: Signatures of thread functions silently “change”

- What does the following print?

```
void f(int &i) { i = 5; }
int main() {
  int i = 2;
  std::thread t(f, i);
  t.join();
  cout << i << endl;
  return 0;
}
```

A compile error (if you're lucky), 2 if your not!

- Of course, 5 was intended
- Unfortunately, thread arguments are not interpreted exactly the same way as just calling the thread function with the same arguments
- This means that even an application programmer using threads needs to understand something subtle about templates

What went wrong, continued

- Imagine std::thread's constructor like the following

```
struct thread { ...
  // 0 arg thrfunc constructor
  template<typename func>
  thread(func f);
  // 1 arg thrfunc constructor
  template<typename func, typename arg>
  thread(func f, arg a);
  ...
};
...
  // Deduces thread::thread<void(*)(int), int)
  std::thread t(f, i);
...
```

IOW, Templates don’t know f takes its argument by reference

- To do this, we will use the “ref” wrapper in <functional>
-
```
void f(int &i) { i = 5; }
int main() {
  int i = 2;
  std::thread t(f, std::ref(i));
  t.join();
  cout << i << endl;
  return 0;
}
```

Does thread's constructor really look like that?

- No, C++11 has "variadic templates" that can take any number of arguments, so we don't need to do separate 0-arg, 1-arg, etc. constructors:

```
struct thread {
    template
     <typename F, typename… argtypes>
    thread(F f, argtypes... a);
    ...};
```

- We'll learn about these next quarter

Thread local storage

- A new storage duration.
- Each thread gets its own copy
- `thread_local int i;`

Futures: Getting values back from a thread

- It's nice that we can pass arguments to a thread (like we do to functions), but how can we get the thread to return a value back?
- Basically, we want to be able to use threads as "asynchronous functions"
- C++11 defines a std::future class that lets a thread return a value when it's done
- Create a future with std::async
  - As soon as you create it, it starts running the function you passed it in a new thread
  - Call get() when you want to get the value produced by the function
  - get() will wait for the thread function to finish, then return the value
  - See example below

std::future example

- From Multithreading in C++0x Part 8

```
#include <future>
#include <iostream>

int calculate_the_answer_to_LtUaE();
void do_stuff();

int main()
{
  std::future<int> the_answer
            = std::async(calculate_the_answer_to_LtUaE);
  do_stuff();
  std::cout <<"The answer to life, the universe and everything is "
            << the_answer.get()
            << std::endl;
 }
```

Can I check if the future has a value yet?

- Yep, std::future has an is_ready() method that tells you if the thread function has completed.

What if the asynchronous function throws an exception?

- If the thread function in a future throws an exception instead of returning a value, then calling get() will throw the exception, just like the asynchronous function was a real function

Homework

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?

Homework 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;
}
```

Homework 4.3

- This problem consists of a series of types. Write a program that defines variables of each type set to some meaningful value (You are highly encouraged to check with a compiler). Googling "c++ declarators" may help. Each one you get is worth 2 points.
- Example problem 1: `int *`
  - One possible answer:
    ```
    int *ip = new int;
    ```
  - Another possible answer
    ```
    int i = 5;
    int *ip = &i;
    ```
- Example problem 2: `int &`
  - One possible answer:
    ```
    int i = 5;
    int &ir(i);
    ```

HW 4.3 (cont)

- `int *`
- `int &`
- `double`
- `A *` (A is any appropriate class).
- `const char *`
- `const char &`
- `long[7]`
- `int **`
- `int *&`
- `float &`
- `int (*)()` (See http://www.newty.de/fpt/index.html)
- `int (*&)()`
- `char *(*)(char *, char *)`

HW 4.3 Extra credit

- See http://www.informit.com/guides/content.aspx?g=cplusplus&seqNum=142 or the standard
- `int A::*`
- `int (A::*)(int *)`
- `int (A::**)(int *)`
- `int (A::*&)(int *)`
- `int (A::*)(double (*)(float &))`
- `void (*p[10]) (void (*)() );`

HW 4-4

- The purpose of this problem is to ensure that you can write basic multithreaded code on your system. Since threading is not portable, please send a transcript. Use the C++11 compliant just::thread library on the cluster or get your own for half price
- Write a program that creates 3 threads that each count up to 100 and output lines like:
  `Thread 3 has been called 4 times`
- To get a thread number, use `std::this_thread.get_id()`
- Make sure you use synchronization to keep different threads from garbling lines like the above.
- Submit the output from your program. What does it tell you about how threads are actually scheduled on your system?

HW 4.5

- Since this lecture is on low-level systems programming and memory, it is a good chance to remind ourselves that computer memory stores numbers in binary
- Learn to count in binary on your fingers
  - See http://en.wikipedia.org/wiki/Finger_binary
  - We'll test this in class
- How high can you count on both hands?
- Extra credit: Count to 31 in 15 seconds or less