C++

February 19, 2015

Mike Spertus

mike_spertus@symantec.com

Lambdas

- We have used C++11 lambdas in class
- `[](int x, int y) { return x < y; }` is an anonymous boolean valued "function" returning true if `x < y`
- "Function" is in quotes because it is actually an unspecified type. You can only assign to a function pointer if there is no capture list (see following slides), but you can always store as follows
  - `auto f = [](int x, int y) { return x < y; }`

Lambda return values

- If your lambda just consists of a return statement, the compiler infers the type.
- If not, you give it using the “unified function syntax”

```
[](int x, int y) -> double {
  int z = x + y; return z + x;
}
```

- This syntax avoids parsing problems

Capture lists

- To capture local variables by reference, use [&]
- To capture local variables by value, use [=]
- For finer-grained results, specify exactly what you want to capture

  -
    ```
    int i=0;
    vector<int> v;
    /* ... */
    for_each(v.begin(), v.end(),
      [&i](int x) { i += x; });
    ```

Lambdas and algorithms

- Now all of the standard library algorithms can be used as easy as for loops

-
  ```
  std::vector<int> someList; // Wikipedia
  int total = 0;
  std::for_each
   (someList.begin(), someList.end(),
    [&](int x) { total += x; });
  ```

C++14: Polymorphic lambdas

- In C++14, you can give a lambda an `auto` argument
- In the following code, the compiler figures out that `x` is an `int`.
-
  ```
  std::vector<int> someList; // Wikipedia
  int total = 0;
  std::for_each
    (someList.begin(), someList.end(),
     [&](auto x) { total += x; });
  ```

Behind the scenes: How polymorphic lambdas work

- Advanced
- When the compiler sees
  - `[](auto x) { cout << x; }`
- It generates a functor

  -
    ```
    struct lambdaFunctor_42393626243 {
      template<typename T>
      void operator()(T t) {
        cout << t;
      }
    };
    ```

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

Why is this a big deal?

- Perhaps the biggest secret in computer progress is that computer cores have not gotten any faster in 10 years
  - 2005’s Pentium 4 HT 571 ran at 3.8GHz, which is better than many high-end CPUs today
  - The problem with increasing clock speeds is heat
    - A high end CPU dissipates over 100 watts in about 1 cubic centimeter
    - A light bulb dissipates 100 watts in about 75 cubic centimeters

Why doesn’t anyone know about this?

- Even though cores have not gotten faster, the continued progression of Moore’s law means that computers today have many cores to run computations in parallel
  - Even cell phones can have 4 cores
  - 12 to 24 cores are not unusual on high-end workstations and servers
    - 24 to 48 if you count hyperthreading

Back to C++

- Unfortunately, C++ did not have any notion of multithreading until C++11 came out
- C++ programmers used os-provided multithreading libraries like pthreads and win32 threads
- But this is not acceptable
  - Using these libraries are clunky, not well integrated with other language constructs, and not C++ like
  - Even worse, Threads Cannot be Implemented as a Library (Hans Boehm, PLDI 2005)
    - http://www.hpl.hp.com/techreports/2004/HPL-2004-209.pdf

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

WARNING!

- The next several slides are very confusing
- You do not need to learn them in detail (or at all) as C++11 resolves these problems
- However, we give these slides for two reasons
    - Without seeing such bizarre unexpected behavior, one would be tempted to continue using thread libraries
    - They are very interesting

What can r1 and r2 end up as? (Boehm)

Initially x = y = 0;

Thread 1

```
x = 1;
r1 = y;
```

Thread 2

```
y = 1;
r2 = x;
```

Answer: Any combinations of 0 and 1!

- Intuitively r1 == r2 == 0 impossible
- Practically, the compiler (or the hardware) may reorder the statements because it doesn’t matter within a given thread which order the assignments take place
- However, it does matter if the variables are used by another thread at the same time and we could end up with both r1 and r2 being 0
    - Note: Under pthreads rules this is simply illegal

If q = 0, what can another thread see count as? (Boehm)

```
[count is global]
for (p = q; p!= 0; p = p->next) {
    count++;
}
```

- Other threads may see `count == 1`!
- Compiler may rewrite code by speculatively incrementing `count` before the loop, and decrementing if necessary at the end!
- Even gcc –O2 does this.

Is this code correct?

```
class A {
public:
    virtual void f();
};
A *a; // Global variable
```

Thread 1

```
a = new A;
```

Thread 2

```
if (a) a->f();
```

Not on modern multicore computers!

- Writes made on one processor may not be seen in the same order on another processor!
  - Allows microprocessor designers to use write buffers, instruction execution overlap, out-of-order memory accesses, lockup-free caches, etc.
- Thread 2 may see the assignment to `a` before it sees the vtable of the new `A` object!
- If that happens, the `a->f()` call will crash!
- Modern processors use *Weak Consistency*

Weak memory consistency

In a multiprocessor system, storage accesses are weakly ordered if (1) accesses to global synchronizing variables are strongly ordered, (2) no access to a synchronizing variable is issued by a processor before all previous global data accesses have been globally performed, and if (3) no access to global data is issued by a processor before a previous access to a synchronizing variable has been performed.

—Dubois, Scheurich, Briggs (1986)

If the compiler does not have a notion of synchronizing variables, the above says nothing! Prior to C++11, this is addressed non-portably by vendor-specific synchronization extensions to C++.

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`.
- How can we make sure we release the mutex when we are done no matter what?
- RAII!
- 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 looks 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);
...
```

- In fact, thread constructors use "variadic argument lists," which we haven't (yet) covered

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 separate 0-arg, 1-arg, etc. constructors:

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

- We'll learn about these later

Lambdas can help

- As a result of the above complexity, some people recommend using lambdas instead of functions or functors with threads

-
  ```
  int main() {
      int i = 2;
      std::thread t([&i]() {i=5;});
      t.join();
      cout << i << endl;
      return 0;
  }
  ```
- Correctly prints 5

Thread local storage

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

Async functions: Running functions in another 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

Parallel accumulate

- It would be really nice to have an implementation of std::accumulate that breaks up its input into pieces, adds up each piece in parallel and then adds up the results from each of the pieces
- Let’s do this with futures
- async_accumulate_function.cpp

Homework

HW 7-1

- 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 threads
- 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 7-2

- Write a thread-safe stack using locks
  - The only required operations are push and pop
- E.g., multiple threads can concurrently do things like the following without corrupting the stack

  ```
  mpcs50144::stack<int> s;
  s.push(7);
  s.push(5);
  cout << s.pop();
  ```
- For extra credit, add additional useful functionality (e.g., initializer list constructor, etc.)

HW 7-3

- Modify the matrix program to compute the determinant in parallel
- How does the performance depend on the size of the matrix?
- For extra credit, come up with other improvements to the performance of determinant

HW 7-4

- Use a lambda with a capture list to give another solution to the `for_each` homework from lecture 4 that is different from all of the solutions I gave in class during the week 5 homework review
  - I have uploaded them to this week's chalk
- Compare and contrast it to the other solutions