C++ Concurrency Framework

Chenyu Zheng

CSCI 5828 – Spring 2010
Prof. Kenneth M. Anderson
University of Colorado at Boulder

Content

- Actuality Introduction
  - Concurrency framework in the 2010 new C++ standard
  - History of multi-threading in C++
- Feature demonstration
  - Comparison between previous concurrency solutions and current concurrency support
- Framework Component Analysis
  - Thread Management
  - Data Sharing
  - Synchronization of Concurrent Operations
  - The New C++ Memory Model
  - Operations on Atomic Types

New Standard (C++0x)

- 11 years after the original C++ Standard (published in 1998), the C++ Standards committee is giving the language and its supporting library a major overhaul.

- The C++0x is due to be published at the end of 2010 and will bring with it a whole swathe of changes that will make working with C++ easier and more productive.

- One of the most significant new features: **the support of multi-threaded programs**
  - allow us to write multi-threaded C++ programs without relying on platform-specific extensions

    =>
  - write portable multi-threaded code with guaranteed behavior

History of multi-threading in C++

- The 1998 C++ Standard
  - Does not acknowledge the existence of threads
  - The memory model is not formally defined

  =>
  - Can't write multi-threaded applications without compiler-specific extensions

- Solutions
  - compiler vendors add extensions to the language themselves
  - C APIs led compiler vendors to support multi-threading with various platform specific extensions.
    - POSIX C Standard
    - Microsoft Windows API

History of multi-threading in C++ Cont.

- More Advanced Solutions
  - Accumulate sets of C++ classes that wrap the underlying platform specific APIs to provide higher level facilities for multi-threading that simplify the tasks
    - Application frameworks such as MFC
    - General-purpose C++ libraries such as Boost and ACE
- Common design among these solutions
  - the use of the Resource Acquisition Is Initialization (RAII) idiom with locks to ensure that mutexes are unlocked when the relevant scope is exited

History of multi-threading in C++ Cont.

- Lack of a formal multi-threading-aware memory model and standard library support means
  - Development has to
    - allow the use of the corresponding C API for the platform
    - ensure the C++ runtime library works in the presence of multiple threads
  - Optimization is limited when
    - trying to gain higher performance by using knowledge of the processor hardware
    - writing cross-platform code where the actual behavior of the compilers varies between platforms
- Although not perfect, we survived, WHY?
  - a large number of multi-threaded C++ programs have been written and because of the development of the compilers and processor

Comparison: Functionality

- Concurrency Support in the New Standard
  - Brand new thread-aware memory model
  - Extended standard library support
    - managing threads
    - protecting shared data
    - synchronizing operations between threads
    - low-level atomic operations

    =>

    - Provides both integrated high-level facilities and sufficient low-level facilities

Comparison: Efficiency

- Abstraction Penalty
  - Costs associated with using any high-level facilities compared to using the underlying low-level facilities directly
- Not Really
  - Although sometimes the use of high-level facilities does comes with a performance cost due to the additional code that must be executed
    - in general the cost is no higher than would be incurred by writing equivalent functionality by hand
    - the compiler may well inline much of the additional code anyway
- And Also Design…
  - Even if profiling does demonstrate that the bottleneck is in the C++ Standard Library facilities, it may be due to poor application design rather than a poor library implementation

The Flavor: Hello World

- Before we go to the component analysis

```
83  #include <iostream>
84  #include <thread>
85  void hello()
86  {
87      std::cout<<"Hello Concurrent World\n";
88  }
89  int main()
90  {
91      std::thread t(hello);
92      t.join();
93  }
```

The Flavor: Hello World Cont.

- the functions and classes for managing threads are declared in <thread>
- Initial Function
  - Every C++ program has at least one thread, which is started by the C++ runtime: initial function: main()
  - std::thread object named t has the new function hello() as its initial function

Framework Component Analysis

- Thread Management
- Data Sharing
- Synchronization of Concurrent Operations
- The New C++ Memory Model
- Operations on Atomic Types

Thread Management: Two Ways of Launching a Thread

- Using function
  - void-returning function that takes no parameters
- Using class
  - Pass an instance of a class with a function call operator to the std::thread constructor
- Attention: scope and lifetime
  - Since the callable object supplied to the constructor is copied into the thread, the original object can be destroyed immediately.
  - However, if the object contains any pointers or references, it is important to ensure that those pointers and references remain valid as long as they may be accessed from the new thread

Thread Management: Two Ways of Launching a Thread Cont.

```
105  void do_some_work();
106  std::thread my_thread(do_some_work);
```

```
111   class background_task
112   {
113   public:
114       void operator()() const
115       {
116           do_something();
117           do_something_else();
118       }
119   };
120   background_task f;
121   std::thread my_thread(f);
```

Thread Management: Wait for Thread Completion

- join(): block and wait
  - Inserting a call to std::thread instance.join() before the to ensure that the thread was finished before the closing brace of the function body would therefore be sufficient function was exited, and thus before the local variables were destroyed

Thread Management: Wait for Thread Completion Cont.

```
131  struct func;
132  void f()
133  {
134      int some_local_state=0;
135      std::thread t(func(some_local_state));
136      try
137      {
138          do_something_in_current_thread();
139      }
140      catch(...)
141      {
142          t.join();                          #2
143          throw;
144      }
145      t.join();                              #1
146  }
```

- This code ensures that a thread with access to local state is finished before the function exits whether the function exits
  - normally (#1)
  - by an exception (#2)

Thread Management: Ensure Safe Thread Completion

- Standard Resource Acquisition Is Initialization idiom (RAII)

```
156  class thread_guard
157  {
158      std::thread& t;
159
160  public:
161
162      ~thread_guard()
163      {
164          if(t.joinable())
165          {
166              t.join();
167          }
168      }
169      thread_guard(thread_guard const&)=delete;
170      thread_guard& operator=(thread_guard const&)=delete;
171  };
```

Thread Management: Ensure Safe Thread Completion Cont.

- joinable()
  - tests to see if the std::thread object is joinable before calling join().
  - join() can only be called once for a given thread of execution
  - it would therefore be a mistake to do so if the thread had already been joined with
- =delete
  - The copy constructor and copy-assignment operator are marked
  - ensure they are not automatically provided by the compiler
  - copyed or assigned objects probably would outlive the scope of the thread it was joining

Thread Management: Set Thread Completion Point

- detach() member function of the std::thread
  - Destroy std::thread object at the point where you wish to detach the thread
  - After the call completes
    - the std::thread object is no longer associated with the actual thread of execution
    - no longer joinable

```
184  void do_some_work();
185  std::thread t(do_background_work);
186  t.detach();
187  assert(!t.joinable());
```

Thread Management: Passing Arguments to a Thread Function

- Passing additional arguments to the std::thread constructor
  BY
  Passing arguments to the callable object or function

- The arguments are copied into internal storage, where they can be accessed by the newly created thread of execution, even if the corresponding parameter in the function is expecting a reference.

Thread Management: Passing Arguments to a Thread Function Cont.

```
195  //Function
196  void f(int i,std::string const& s);
197  std::thread t(f,3,"hello");
198
199  //Class
200  class X
201  {
202  public:
203      void do_lengthy_work(int i);
204  };
205
206  X my_x;
207  std::thread t(&X::do_lengthy_work,&my_x,2);
```

- This code will invoke my_x.do_lengthy_work() on the new thread
- the third argument to the std::thread constructor will be the first argument to the member function, and so forth

Thread Management: Transferring Ownership of a Thread

```
213  std::move function
214
215  void some_function();
216  void some_other_function();
217  std::thread t1(some_function);          #1
218  std::thread t2=std::move(t1);           #2
219  t1=std::thread(some_other_function); #3
220  std::thread t3;
221  t3=std::move(t2);
222  t1=std::move(t3);
```

- #1: a new thread is started and associated with t1
- #2: ownership of some_function is transferred over to t2 when t2 is constructed, t1 no longer has an associated thread of execution
- #3: a new thread is started, and associated with t1

Thread Management: Transferring Ownership of a Thread Cont.

- Ownership could also be transferred into a function

```
230  std::thread f(std::thread t){return t};
231  void some_function();
232  void g()
233  {
234      // ownership NOT transferred
235      std::thread a(some_function);
236      f(a);
237      // ownership transferred
238      std::thread b(some_function);
239      f(std::move(b));
240  }
```

Thread Management: Number of Threads and Thread ID

- std::thread::hardware_concurrency()
  - Returns an indication of the number of threads that can truly run concurrently for a given execution of a program.
  - On a multi-core system it might be the number of CPU cores.
  - Only a hint: might return 0 if this information is not available
- std::thread:: get_id()
  - Returns the identifier for a thread
  - If the std::thread object doesn't have an associated thread of execution, returns a default-constructed std::thread::id object, which indicates “not any thread”.

Thread Management: Number of Threads and Thread ID Cont.

- std::thread::id a, b; a == b
  - True if: a and b represent the same thread
  - True if: a and b are holding the “not any thread” value
  - False if: a and b represent different threads
  - False if: a or b represents a thread and the corresponding b or a is holding the “not any thread” value

Data Sharing: Avoiding Problematic Race Conditions

- C++ synchronization primitive: Mutexes (named after mutual exclusion)
  - they're not a silver bullet
  - we should protect the right data
  - we should avoid race conditions inherent in your interfaces
  - we should avoid deadlock inherent from the nature of mutexes

Data Sharing: Avoiding Problematic Race Conditions Cont.

```
265  #include <list>
266  #include <mutex>
267  #include <algorithm>
268  std::list<int> some_list;
269  std::mutex some_mutex;
270
271  void add_to_list(int new_value)
272  {
273      std::lock_guard<std::mutex> guard(some_mutex);
274      some_list.push_back(new_value);
275  }
276  bool list_contains(int value_to_find)
277  {
278      std::lock_guard<std::mutex> guard(some_mutex);
279      return std::find(some_list.begin(),some_list.end(),value_to_find)
280          != some_list.end();
281  }
```

- std::lock_guard<std::mutex> guard(some_mutex) in add_to_list and list_contains functions means that the accesses in these functions are mutually exclusive

Data Sharing: more Flexible Locking

- std::unique_lock
  - does not always own the mutex that it is associated with
  - you can pass std::adopt_lock as a second argument to the constructor: have the lock object manage the lock on a mutex
  - you can also pass std::defer_lock as the second argument to indicate that the mutex should remain unlocked on construction, The lock can then be acquired later by
    - calling lock() on the std::unique_lock object
    - passing the std::unique_lock object itself to std::lock()

Data Sharing: more Flexible Locking Cont.

- Usage
  - deferred locking
  - transferring mutex ownership between scopes

```
294  class some_big_object;
295  bool operator<(some_big_object& lhs,some_big_object& rhs);
296  class X
297  {
298  private:
299      some_big_object some_detail;
300      mutable std::mutex m;
301  public:
302      X(some_big_object const& sd):some_detail(sd){}
303      friend bool operator<(X const& lhs, X const& rhs)
304      {
305          if(&lhs==&rhs)
306              return false;
307          std::unique_lock<std::mutex> lock_a(lhs.m,std::defer_lock); #1
308          std::unique_lock<std::mutex> lock_b(rhs.m,std::defer_lock); #1
309          std::lock(lock_a,lock_b);                                   #2
310          return lhs.some_detail<rhs.some_detail;
311      }
312  };
```

Data Sharing: Transferring Mutex Ownership Between Scopes

```
320  std::unique_lock<std::mutex> get_lock()
321  {
322      extern std::mutex some_mutex;
323      std::unique_lock<std::mutex> lk(some_mutex);
324      prepare_data();
325      return lk;
326  }
327  void process_data()
328  {
329    std::unique_lock<std::mutex> lk(get_lock()); #1
330    do_something();
331  }
```

- #1: The function can transfer ownership directly into its own process_data() std::unique_lock instance (#1),
- the call to do_something() can rely on the data being correctly prepared without another thread altering the data in the mean time.

Synchronizing Concurrent Operations: Waiting for an Event

- Sometimes you don't just need to protect the data, but to synchronize actions on separate threads.
- Waiting for an Event: mutex + sleep

Synchronizing Concurrent Operations: Waiting for an Event Cont.

```
 7  bool flag;
 8  std::mutex m;
 9  void wait_for_flag()
10  {
11      std::unique_lock<std::mutex> lk(m);
12      while(!flag)
13      {
14          lk.unlock();                                                    #1
15          std::this_thread::sleep(std::chrono::milliseconds(100));        #2
16          lk.lock();                                                      #3
17      }
18  }
```

- #1 Unlock the mutex whilst we sleep, so another thread can acquire it and set the flag
- #2 Sleep for 100ms
- #3 Lock the mutex again before we loop round to check the flag

Synchronizing Concurrent Operations: Waiting for a Condition with Condition Variables

- two implementations of a condition variable: std::condition_variable, and std::condition_variable_any
  - they need to work with a mutex in order to provide appropriate synchronization
  - the former is limited to working with std::mutex
  - the latter can work with anything that meets minimal criteria for being mutex-like
  - there is the potential for additional costs in terms of size, performance or operating system resources, so the latter std::condition_variable should be preferred unless the additional flexibility is required

Synchronizing Concurrent Operations: Waiting for a Condition with Condition Variables Cont.

```
32  std::mutex mut;
33  std::queue<data_chunk> data_queue;
34  std::condition_variable data_cond;
35
36  void data_preparation_thread()
37  {
38      while(more_data_to_prepare())
39      {
40          data_chunk const data=prepare_data();
41          std::lock_guard<std::mutex> lk(mut);
42          data_queue.push(data);
43          data_cond.notify_one();
44      }
45  }
46
47  void data_processing_thread()
48  {
49      while(true)
50      {
51          std::unique_lock<std::mutex> lk(mut);
52          data_cond.wait(lk,[]{return !data_queue.empty();});
53          data_chunk data=data_queue.front();
54          data_queue.pop();
55          lk.unlock();
56          process(data);
57          if(is_last_chunk(data))
58              break;
59      }
60  }
```

Synchronizing Concurrent Operations: Waiting for a Condition with Condition Variables Cont.

- []{return !data_queue.empty();}: checks to see if the data_queue is not empty (lambda function)
- data_cond.wait method: checks the condition (by calling the lambda function), and returns if it is satisfied.
- If the condition is not satisfied, it unlocks the mutex and puts the thread in a “waiting” state

Synchronizing Concurrent Operations: Waiting for One-off Events with Futures

- One-off Events
  - Suppose you're going on holiday abroad by plane, fundamentally you're just waiting for one thing: the signal that it's time to get on the plane. Not only that, but a given flight only goes once
- Future
  - C++ model of the one-off event.
  - A future may have data associated with it.
  - A thread can poll the future to see if the event has occurred.
  - Once an event has happened (and the future has become ready), then the future cannot be reset.

Synchronizing Concurrent Operations: Waiting for One-off Events with Futures Cont.

- Two class templates: std::unique_future<> and std::shared_future<>
  - an instance of std::unique_future is the one and only instance that refers to its associated event
  - multiple instances of std::shared_future may refer to the same event
    - all the instances will of course become ready at the same time,
    - they may all access any data associated with the event

Synchronizing Concurrent Operations: Waiting for One-off Events with Futures Cont.

```
79  //sit and wait
80  void wait_for_flight1(flight_number flight)
81  {
82      std::shared_future<boarding_information>
83          boarding_info=get_boarding_info(flight);
84      board_flight(boarding_info.get());
85  }
86
87  get(): waits for the future to become ready before returning the associated data
88
89  //get on working and periodically check
90  void wait_for_flight2(flight_number flight)
91  {
92      std::shared_future<boarding_information>
93          boarding_info=get_boarding_info(flight);
94      while(!boarding_info.is_ready())
95      {
96          eat_in_cafe();
97          buy_duty_free_goods();
98      }
99      board_flight(boarding_info.get());
100 }
```

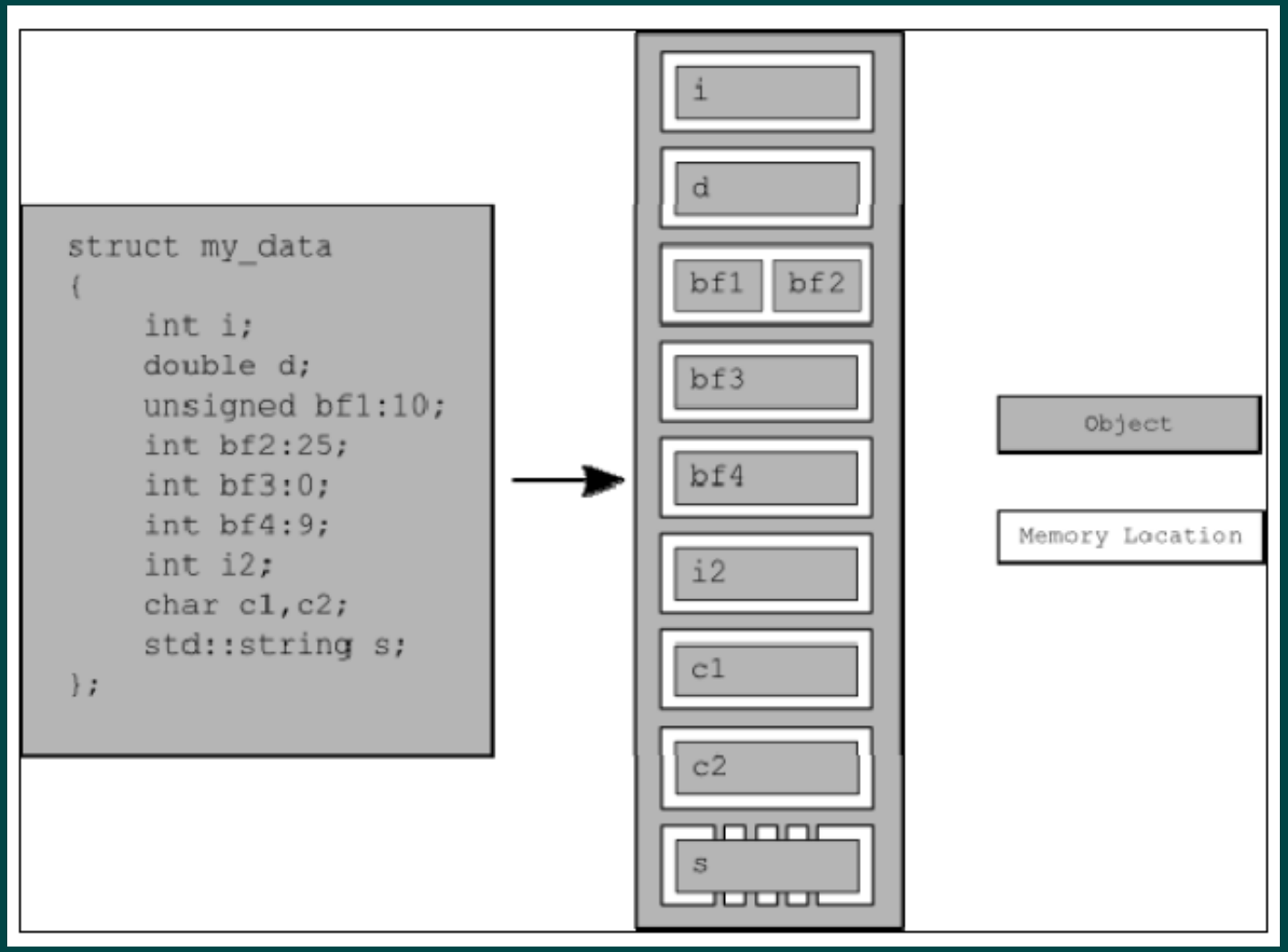
The New C++ Memory Model

- All data in a C++ program is made up of objects
- The C++ Standard defines an object as "a region of storage", though it goes on to assign properties to these objects, such as their type and lifetime
- Whatever its type, an object is stored in one or more memory locations.
- Each such memory location is either an object (or sub-object) of a scalar type, such as
  - unsigned short
  - my_class*
  - a sequence of adjacent bit-fields

The New C++ Memory Model Cont.

- Significance to Concurrency
  - If there is no enforced ordering between two accesses to a single memory location from separate threads, these accesses is not atomic,
  - if one or both accesses is a write, this is a data race, and causes undefined behaviour
- Example
  - The division of a struct into objects and memory locations

The New C++ Memory Model Cont.



Atomic Types

- Goals of the Standards committee is that there shall be no need for a lower-level language than C++
- The Standard atomic types can be found in the <cstdatomic> header.
  - All operations on such types are atomic
  - Only operations on these types are atomic in the sense of the language definition
- As well as the basic atomic types, the C++ Standard Library also provides a set of typedefs for the atomic types corresponding to the various non-atomic Standard Library typedefs

Operations on Atomic Types

- Each of the operations on the atomic types has an optional memory ordering argument, three categories
  - store operations
    - can have memory_order_relaxed, memory_order_release or memory_order_seq_cst ordering;
  - load operations
    - can have memory_order_relaxed, memory_order_consume, memory_order_acquire or memory_order_seq_cst ordering;
  - read-modify-write operations
    - can have memory_order_relaxed, memory_order_consume, memory_order_acquire, memory_order_release, memory_order_acq_rel or memory_order_seq_cst ordering
  - the default ordering for all operations is memory_order_seq_cst.

Review and Conclusion

- Actuality Introduction
  - Concurrency framework in the 2010 new C++ standard
  - History of multi-threading in C++
- Feature demonstration
  - Comparison between previous concurrency solutions and current concurrency support
- Framework Component Analysis
  - Thread Management
  - Data Sharing
  - Synchronization of Concurrent Operations
  - The New C++ Memory Model
  - Operations on Atomic Types

Q & A

- Thank you ☺