Now let’s see it in C++17

The point is that if you use “if constexpr” it doesn’t matter if there are compile errors in the branch not taken

```
template<typename I1, typename I2>
I2 optimized_copy(I1 first, I1 last, I2 out)
{
    if constexpr (is_trivially_copy_constructible_v<T>()) {
        memcpy(out, first, (last - first) * sizeof(T));
        return out + (last - first);
    }
    else {
        memcpy(out, first, (last - first) * sizeof(T));
    }
    return out + (last - first);
}
```

Summary

- C++17 makes real progress in
  - Making templates work for you rather than against you
  - Doing natural type inference while retaining static type safety
  - Preparing your code for C++20 concepts
- Of course, the big win will be the introduction of Concepts in C++20, but hopefully this is more than enough to tide you over until then

2017 CPP-Summit

Memory in C++: Past, Present, and Future

Mike Spertus

Fellow, Chief Scientist, Cyber Security Services, Symantec

C++ Memory Management

- The first rules that every C++ programmer learns
    - You create with (dynamic lifetime) by calling new
    - When you are done with the object, you must release it by calling delete to avoid a memory leak
- Our first goal today will be to discard these rules
- And then things will get interesting

What's wrong with new & delete?

- Exceptions
  - Exceptions make control flow unpredictable, so it is very difficult to know when to delete it

```
int f() {
  try {

   A *ap = new A;
   g();
   delete ap; // If g() throws exception, ap never deleted
 } catch (...) {

      cout << "Exception " << endl;
 }
 return 0;
}
```

- Threads
  - Which thread is the last to use an object likely will not be known until runtime

Can we ensure the delete is never skipped?

- A Java programmer would wonder why we didn't just use `finally`
  - Actually, they would wonder why we didn' t use GC, but more on that later

  -
    ```
    int f() {
      try {
    ```

-
  ```
        A *ap = new A;
        g();
      } catch (...) { cerr << "Exception\n";}
      } finally { delete ap; } // Whoops! Not C++
      return 0;
    }
    ```
- One big problem
  - It' s not C++!

RAII

- Why doesn't C++ have `finally`?
- Because it has something better
- Destructors of local variables are always called however you leave scope
- Using this to manage resources is called RAII
  - Resource Acquisition Is Initialization

unique_ptr

- `unique_ptr` destructor deletes the object pointed to
- The memory leaked fixed:

```
int f() {
  try {

     unique_ptr<A> ap{new A};
     g();
  } catch (...) {

     cout << "Exception " << endl;
  }
  return 0;
}
```

Variants

- `unique_ptr<A[]>` owns an array
  - Destructor uses `delete []`
  - Replaces C++98’s now deprecated `auto_ptr`
- `shared_ptr<A>` is a reference counted pointer to A
  - The object is deleted when its last `shared_ptr` goes away
  - Interestingly, there is no `shared_ptr<A[]>`
    - This is a mistake and is corrected in the Library Fundamentals Technical Specification
    - Until then, you can use a custom deleter (google it), Boost, your own class, etc.

Best practice

- *Effective C++* item 17
    - Store newed objects in smart pointers in standalone statements
    - Gets rid of `delete`
- An interesting proposal to make this easier
    - Walter Brown, [N3418: A Proposal for the World’ s Dumbest Smart Pointer, v3]
    - `observer_ptr` acts like a raw pointer, but reminds you that it doesn’ t contribute to object ownership
    - Included in Library Fundamentals TS on the way to a future C++ standard

Getting rid of new

- Why get rid of new?
  - Even garbage collected languages like Java have it
- The problem is that new returns an owning raw pointer, in violation of the above best practice, which can get you into trouble:

```
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

make_shared and make_unique

- make_shared<T> and make_unique<T> create an object and return an owning pointer
- The following two lines act the same
  - `auto ap { make_shared<T>(4, 7) };`
  - `shared_ptr<T> ap { new T(4, 7) };`
- make_unique wasn't added until C++14
  - Oops
- Now we can fix our previous example

```
void f()
{
  auto a1 = make_unique<A>(), a2 = make_unique<A>();
  // g is responsible for deleting
  g(a1.release(), a2.release());
}
```

- *Effective Modern C++* Item 21
  - Prefer `std::make_unique` and `std::make_shared` to direct use of new

Some further improvements

- If we can modify `g()`, we should really change it to take `unique_ptr<T>` arguments because otherwise, we would have an owning raw pointer
  - Remember, `g()` takes ownership
  - `g(unique_ptr<T>, unique_ptr<T>);`
- Now we can call `g(make_unique<T>(), make_unique<T>());`
- Interestingly, the following does not work because ownership will no longer be unique

  -
    ```
    auto p1 = g(make_unique<T>();
    auto p2 = g(make_unique<T>();
    g(p1, p2); // Illegal! unique_ptr not copyable
    ```
- To fix, we need to move from p1 and p2
  - `g(move(p1), move(p2)); // OK. unique_ptr is movable`

Garbage Collection in C++

- C++ can be garbage collected
- There is some minimal support in the standard
  - See Boehm, Spertus, "Garbage Collection in the Next C++ Standard", ISMM 09
- To enable garbage collection, use a 3rd-party library like the Boehm collector

The challenge of C++ garbage collection

- One needs to be careful because C++ types are not available at runtime
  - RTTI is something else entirely
- Even if it were, C++ code often does things like store pointers in integers, xor pointers, etc. that could hide them from the garbage collector's analysis of pointers leading to a premature free
  - Most of these practices were turned into undefined behavior in C++11

Conservative collection

- Treat every word of an object as if it were a pointer
  - "Garbage Collection in an Uncooperative Environment", *Software Practice & Experience*, September 1988, pp. 807-820
- Amazingly, this is performant
  - Computers are built to scan memory, and the vast majority of non-pointers can be rejected with one or two comparisons against the heap bounds
  - Works amazingly well in practice
  - See Why Conservative Collection?
- Try the Boehm collector on some of your code to see
- Precise (i.e., non-conservative) garbage collection approaches can be used with C++ as well

What is Memory in C++?

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
- WE WILL RETURN TO THE INTERACTION OF THREADS AND MEMORY MANAGEMENT AGAIN AND AGAIN

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
  - They are taken from Hans Boehm’ s PLDI paper “Threads cannot be implemented as a library.”
- You do not need to learn them in detail (or at all) as C++11 resolves these problems
- However, we give these slides for several reasons
  - They motivate and clarify the memory model changes in C++11
  - Without seeing such bizarre unexpected behavior, one might be tempted to continue using thread libraries
  - They are very interesting

What can r1 and r2 end up as before C++11? (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 before C++11? (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++.

C++11 Memory Model

- Sequential Consistency in the absence of race conditions
  - This basically means that if data is shared between threads, you must use an atomic or lock
- Herb Sutter atomic<> Weapons
  - http://channel9.msdn.com/Shows/Going+Deep/Cpp-and-Beyond-2012-Herb-Sutter-atomic-Weapons-1-of-2

Memory model best practices

- Here are the takeaways
  - Try to avoid sharing data between threads except when necessary
  - When you share data between threads, always use locks or atomics to ensure both threads have a coherent view of the shared data
- A good reference
  - Boehm, Adve, "*You Don' t Know Jack about Shared Variables of Memory Models: Data Races are Evil*" Communications of the ACM 55, 2 Feb. 2012
  - http://queue.acm.org/detail.cfm?id=2088916

Cache-Conscious Programming

Cache effects

- Accessing main memory can take a processors hundreds of cycles
- Therefore, processors use high-speed caches to maintain local copies of data
  - See http://bucarotechelp.com/computers/anatomy/images/L3_cache.png
  - If another processor needs to read/write that memory, it needs to force other processors to flush or invalidate any cached copies of the memory
    - See http://en.wikipedia.org/wiki/Cache_coherency

Cache lines and false sharing

- When data is moved from main memory to cache, enough data is always moved to fill a ″cache line.”
  - The size of a cache line varies by processor and needs to be looked up in the processor datasheet. A typical size would be 32 bytes, but it varies greatly.
- As a result, if two processors are modifying data within 32 bytes, they are constantly forcing each other to invalidate their cache (″false sharing″)

False sharing example

- Let us look at some code for a distributed method counter
- Looks good. What is the problem?
- Since all of the thread-specific counters are stored in an array, they pretty much all end up in the same cache line, which means updating a counter on one thread means that all of the other threads will have to reload their counters from main memory since they are in the same cache line
    - This is very slow. Maybe 100x slower than accessing cache memory
- This kind of coupling of seemingly independent variables because they reside on the same cache line is known as **false sharing**

Eliminating the false sharing

- Lets see what happens if we add some padding to the counters so they all are far enough apart to fall in distinct cache lines
- On my dual socket workstation with 10 threads per CPU, the program gets ~15 times faster!
- However, on some single-socket laptops with a low number of cores, changing the amount of padding has no effect on performance!
  - Since all of the cores use the same caches
- This is very insidious because code that performs well on dev laptops often performs very badly on multiprocessor servers!
- This is a good illustration of why following best practices like ″putting independent variables on distinct cache lines″ is important, even if you aren′t seeing it in your own benchmarking

Direct-Mapped Caches

- Often a single memory location can only be mapped to one or two possible cache lines
  - See http://www.cs.umd.edu/class/sum2003/cmsc311/Notes/Memory/direct.html
- Not understanding direct-mapped caches can have dire consequences
  - See next slide

Important real-world example of direct-mapped caches going haywire

- The popular postscript rendering program ghostscript was originally written by Peter Deutsch, who wrote a custom memory manager. It is certainly true that malloc()/free() performance is critical in postscript and Peter Deutsch was a memory management expert, having coauthored the first high-performance Smalltalk implementation.
- Peter Deutsch used a custom allocator along the lines we discussed last quarter, with a free pages stored in a linked list
- Tests 10 years later showed that ghostscript's memory manager was actually slowing it down by 30%

What went wrong?

- The custom allocator maintained a pool of free-pages in a linked list, with the first word of each free page as a pointer to the next free page. As this code was developed on a machine without a direct-mapped cache, it ran fine. However, on machines with direct-mapped caches, all of the freelist pointers mapped to the same cache line causing a cache miss on each step of walking through the freelist. Ghostscript was spending about a third of its time in cache misses from walking through the page freelist.
- Note: You don’t need to understand this as long as you understand the moral

Cache-conscious programming
(Adapted from Herlihy&Shavit p. 477)

- Objects or fields that are accessed independently should be aligned and padded so they end up on different cache lines.
- Keep read-only data separate from data that is modified frequently.
- When possible, split an object into thread-local pieces. For example, a counter used for statistics could be split into an array of counters, one per thread, each one residing on a different cache line. While a shared counter would cause invalidation traffic, the split counter allows each thread to update its own replica without causing cache coherence traffic.
- If a lock protects data that is frequently modified, then keep the lock and the data on distinct cache lines, so that threads trying to acquire the lock do not interfere with the lock-holder's access to the data.
- If a lock protects data that is frequently uncontended, then try to keep the lock and the data on the same cache lines, so that acquiring the lock will also load some of the data into the cache.
- If a class or struct contains a large chunk of data whose size is divisible by a high power of two, consider separating it out of the class and holding it with an `auto_ptr` to avoid the Ghostscript problem from lecture 5.
- Use a profiling tool like VTune to identify where your cache bottlenecks are

C++ support for cache-conscious programming

- C++17 adds `std::hardware_destructive_interference_size` and `std::hardware_constructive_interference_size` giving the maximum and minimum expected cache line sizes
- hardware_destructive_interference_size tells you how far you need to keep objects used by different threads apart from each other
- hardware_constructive_interference_size tells you how close you need to keep objects for a single thread to get both of them with a single read of main memory
- By using these constructs, you can portably follow all of the cache-conscious programming Best Practices with ease

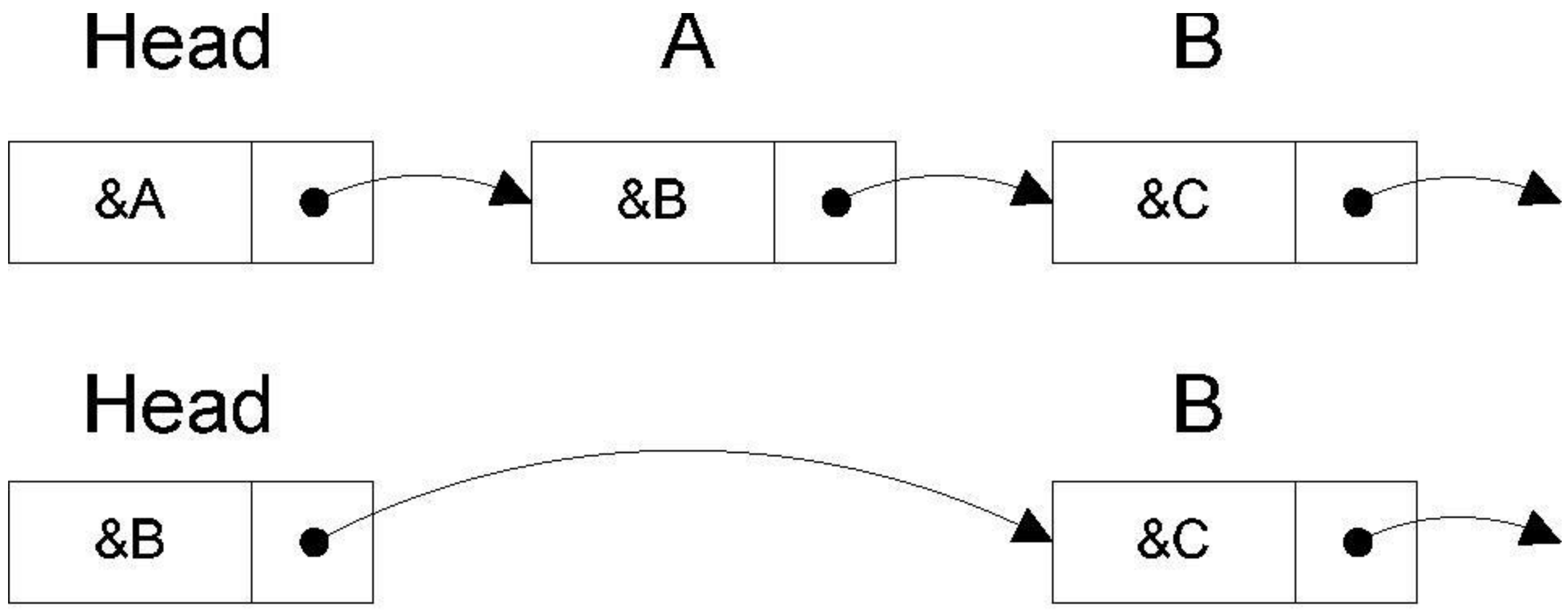
Case Study on the Risks And Rewards of Trying to (Over?) Optimize Multithreaded Code

Background: How to quickly allocate objects of a fixed size?

- Say we're allocating 32-byte objects from 4096-byte pages
- Divide each page in our memory pool into 128 objects in a linked list
- Now, allocate and deallocate 32-byte objects from the list by pushing and popping
  - Fewer than a dozen instructions vs hundreds in a conventional allocator
  - Make sure you lock for thread-safety
- You will implement such a lock-based stack as an exercise

Allocating an object

- Pop the first object off the list



A True Story with a Twist—The Bad Beginning

- A programmer released an application using a linked list allocator like in the previous slide
  - It appeared to speed up his program considerably
- His customers reported that the application become slow as the number of threads increased into the hundreds
- Even though the lock only protects a few instructions, if a thread holding the lock loses its quantum, the list is unavailable until that thread gets another timeslice (perhaps hundreds of quanta later)
- Not acceptable

C++11 atomics

- Sometimes you just want a variable that you can read and update from multiple threads
- Using locks seems a little too complicated for that
- Fortunately, C++11 has a library of atomic types that can be shared between threads

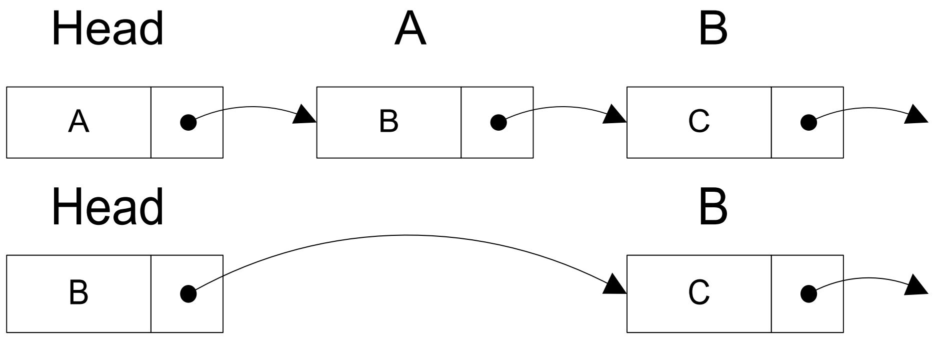
An atomic counter

- You can read an atomic with its `load()` method, write it with its `store()` method and (usually) increment or decrement it with ++ or –-
- Here's how you'd allow a bunch of threads to increment a global task counter

```
atomic<unsigned> tasksCompleted;
void doTask() {
  /* ... */
  // Next line gives right result even if
  // called from multiple threads simultaneously
  tasksCompleted++;
}
void reportsTasksCompleted() {
  cout << tasksCompleted.load();
}
```

Can we make a thread-safe list without locks?

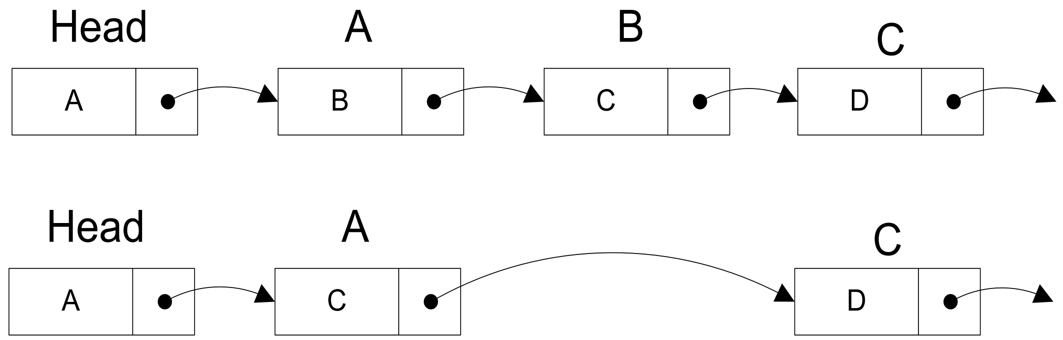
- To remove an element



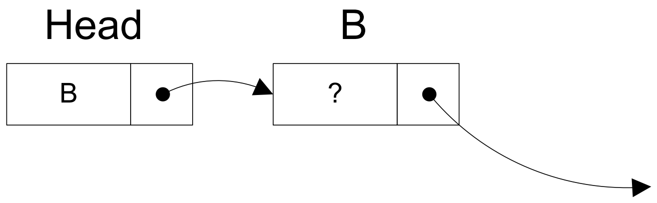
- We need a lock because we need to both return A and update the head to point to B (i.e., A's link) atomically
- Or do we?
- C++11 has an atomic `compare_and_exchange_weak` primitive that does a swap, but only if the target location has the value that we expect
  - Then our update would fail if someone messed with the list in the critical section
  - If so, just loop back and try again

Oops! Doesn't quite work

- Some other thread could do two pops and one push during the critical section, leaving the head unchanged

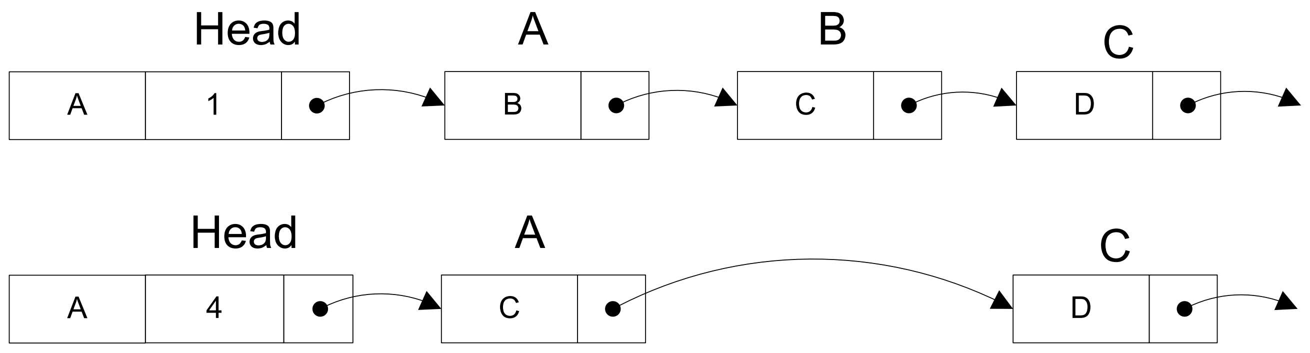


- After the `compare_and_exchange_weak`, B is erroneously back on the list



We can fix this

- Add a "list operation counter" to the head
- Update with 64-bit compare and exchange (on a 32-bit program), which C++ conveniently provides (and maps onto a single x86 instruction provided for just this reason)
- Now the compare and swap fails if intervening list ops happened



What's the point?

- This is much better
- No need for memory barrier
- Only one atomic operation instead of two
- If thread loses its quantum while doing the list operation, other threads are free to manipulate the list
    - This is the big one
- Works on x86-32, x86-64, and Sparc

How is this implemented in C++?

- See lockFreeStack.h in chalk
- Let’s look at it now

What about PPC and Itanium?

- Even better, PPC and Itanium have Linked Load and Store Conditional (LLSC)
- lwarx instruction loads from a memory address and ″reserves″ that address
- stwcx instruction only does a store if no intervening writes have been made to that address since the reservation
- Exactly what we want

What about push?

- The same techniques work for pushing onto the list
    - Exercise to see if you understand
- Not just restricted to lists
    - Many other lock-free data structures are known
    - See the references

A True Story with a Twist—A Happy Ending?

- The programmer switched to using Compare and Exchange-based atomic lists on Sparc
- The customers were happy with the performance
- But wait…

No happy ending?

- The customers started to experience extremely intermittent list corruption
- Virtually impossible to debug
  - He ran 100 threads doing only list operations for hours between failures
- The problem was that Solaris interrupt handlers only saved the bottom 32-bits of some registers
  - Timer interrupts in the critical section corrupted the compare and exchange
  - Fix: Restrict list pointers to specific registers
- Moral: The first rule of optimization is "Don't!"
  - These techniques are powerful but only used where justified

But wait, there's more

- Later, the program started being used on massively SMP systems, and it started to exhibit performance problems
  - The Compare and Exchange locked the bus to be thread-safe but that is expensive as the number of processors went up (this results in a surprising implementation of the Windows Interlocked exchange primitive).
- Since they no longer needed many more threads than processors, they went back to a lock-based

So should you do a class-specific allocator?

- Do you really want to pollute your class with deep assumptions about the HW and OS?
- Do you want to update it everytime there is a new OS rev?
  - Early version of this before threading inadvertently made classes thread unsafe
- The answer is almost always, "No," but…

No way! Except…

- My friend's product wouldn't have been usable without a custom memory manager
- He wouldn't have sold his company for a large sum of money without usable products
- Use it when necessary, but only if you can justify the costs of maintaining your code over every present and future OS/hardware revision
- This story illustrates the real power and danger of using C++
  - Know the difference between "use" and "abuse"

Transactional Memory

- Can we generalize the lock-free stack above to create a new simple and general paradigm for lock-free coding?
- Since locks don′t work well with templates (C++) or modular coding (all languages), people have been looking at other paradigms for concurrent programming that are ″compositional″ (meaning that combining components, either through templates or modular design will not cause deadlocks)

Transactional Memory

- "Transactional memory" is an alternative to locks that is compositional
  - Basically, a thread works lock-free with its own view of shared memory and then "commits" its work as a transaction when it is done. If another conflicting transaction was made by another thread, there is an exception so you can try again.
- See the following references
  - Gottschlich, Boehm, "Generic Programming Needs Transactional Memory"
    - http://transact2013.cse.lehigh.edu/gottschlich.pdf
  - Wong, "What did C++ do for transactional memory?"
    - https://github.com/CppCon/CppCon2014/blob/master/Presentations/What%20did%20C%2B%2B%20do%20for%20Transactional%20Memory/What%20did%20C%2B%2B%20do%20for%20Transactional%20Memory%20-%20Michael%20Wong%20-%20CppCon%202014.pdf
    - Also on chalk
- The C++ committee has released a Preliminary Draft Technical Specification to add transactional memory to C++
  - http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4302.pdf

How to think about transactional memory

- The linked list allocator above was too low level
  - The programmer needs to keep retrying until the `compare_and_swap` succeeds
  - On a PPC, which doesn’t have hardware `compare_and_swap`, the code would need to be rewritten to avoid locks
- Transactional memory allows you to state the high-level requirements and the compiler can figure out the best way to meet those requirements

Transactional Memory in C++ proposal

- `synchronized { /* code */ }`
  - All such synchronized regions are executed as if they are protected by a single global lock
- `atomic_cancel { /* code */ }`
  - The code appears to all other threads as if it runs atomically
  - If the code throws an exception, the transaction is canceled and all the modifications to memory are rolled back
- Sounds inefficient, but modern compilers and processors have many techniques, like speculative execution, hardware and software transactional memory, etc.

`shared_ptr`

- Since `shared_ptrs` delete their target whenever the reference count goes to zero, it is very difficult to know what locks will be held when the target classes destructor is called.
- Great care (or even handle/proxy classes that schedule destruction in a different thread) may be necessary to avoid violating lock ordering.
- When possible, avoid this complexity by not locking in destructors of class that may be managed by `shared_ptrs`.

Questions?

Crafting Embedded Systems in C++

Dan & Ben Saks
Saks & Associates
www.dansaks.com

Overview

- Programming embedded systems is a very large topic.
- This presentation is intended to illustrate some of the challenges that embedded developers face through a few key examples.
- It will show how you can use the features of C++ to overcome those challenges.
- The goal is to help you think about how to craft objects that accurately model hardware.

- ***craft*** (verb):
  - "to make or manufacture (an object, objects, product, etc.) with skill and careful attention to detail."

Overview

- In many ways, embedded systems programming is just plain programming.
- However, embedded systems programming is a bit different.
  - Embedded systems often have stricter resource limitations:
    - run time
    - memory space
    - communication bandwidth
    - power consumption
  - Embedded systems often control hardware directly.

Your Choice

- You can write very simple declarations to model devices.
  - This is common practice.
  - It’s less work up front, but then…
  - Code that accesses devices will be tedious and error-prone.
- You can write more detailed and accurate declarations to model devices.
  - This is what I’m about to show you to how to do.
  - It’s more work up front, but then…
  - Code that accesses devices will be easier to write and more robust.

Your Choice

- You define the representation of each device at most once.
- However, you probably access each device many times.

- As Scott Meyers advises:

✓ ***Make interfaces:***
  - ✓ ***easy to use correctly***
  - ✓ ***hard to use incorrectly.***

Sample Hardware

- This talk uses examples based on ARM E7T (Evaluator-7T) single board computer.
- Programs running on the E7T communicate with devices via memory-mapped locations known as the ***device registers***:
  - a 64KB (16K word) region in the memory address space,
  - beginning at address 0x03FF0000.
- The device registers are grouped.
- Each group communicates with a single device or small set of devices.

Device Registers

- For example, the UART 0 group consists of six device registers:

Offset	Register	Description
0xD000	ULCON	line control register
0xD004	UCON	control register
0xD008	USTAT	status register
0xD00C	UTXBUF	transmit buffer register
0xD010	URXBUF	receive buffer register
0xD014	UBRDIV	baud rate divisor register

- The UART 1 group consists of another set of these same registers, but beginning at offset 0xE000.
- By the way, UART stands for "***U***niversal ***A***synchronous ***R***eceiver/***T***ransmitter".

Choosing the Right Integer Type

- You can declare a device register using the appropriately sized and signed integer type.
- For example:
  - A one-byte data register might be a plain `char`.
  - A two-byte status register might be an unsigned `short`.
- Each device register in these examples occupies a four-byte word.
  - Declaring each device register as `uint32_t` seems to work well.
  - Using a meaningful typedef is even better:

```
typedef uint32_t device_register;
```

Placing Memory-Mapped Objects

- Normally, you don’t choose the memory locations where program objects reside.
- The compiler does, often with substantial help from the linker.
- For an object declaration at global scope, the compiler sets aside memory within a particular code segment.
- For an object declaration at local scope, the compiler sets aside memory within the stack frame of the function containing the declaration.

Placing Memory-Mapped Objects

- For an object representing memory-mapped device registers, the compiler doesn't get to choose where the object resides.
- The hardware has already chosen.
- We need to craft declarations for objects that let us access the hardware that fit with the memory-mapped locations chosen by hardware designer.

Locating Device Registers

- You can access a device register through a pointer whose value is the specified address.
- You can use ***pointer-placement*** to cast an integer value representing the address into a pointer value.
- You can encapsulate the cast-expression in a macro, as in:

```
#define UTXBUF0 ((device_register *)0x03FFD00C)
```

- Alternatively, you can use the cast-expression to initialize a pointer, as in:

```
device_register *UTXBUF0
    = (device_register *)(0x03FFD00C);
```

Placing Memory-Mapped Objects

- Once you've got the pointer initialized, you can manipulate the device register via the pointer.
- For example:

```
*UTXBUF0 = c;   // OK: send the value of c out the port
```

  writes the value of character `c` to the UART 0's transmit buffer, sending the character value out the port.

Placing Memory-Mapped Objects

- Device registers typically have fixed locations.
- `UTXBUF0`'s pointer value shouldn't change during run time.
- The compiler should reject any attempt to modify the value of `UTXBUF0`, as in:

```
UTXBUF0 = UTXBUF1;  // should not compile
++UTXBUF0;          // should not compile
```

Locating Device Registers

- If you declare `UTXBUF0` as a pointer object, then you can modify the pointer's value.
- You should declare `UTXBUF0` as a `const` pointer:

```
device_register *const UTXBUF0
    = ((device_register *)0x03FFD00C);
```

- Using a `const` pointer allows things that should work to still work, but rejects things that shouldn't work:

```
*UTXBUF0 = c;         // still OK
UTXBUF0 = UTXBUF1;    // now a compile error - good
++UTXBUF0;            // now a compile error - good
```

Reference-Placement

- In C++, you can use ***reference-placement*** as an alternative to pointer-placement:

```
device_register &UTXBUF0
    = *(device_register *)0x03FFD00C;
```

- A reference refers to an object.
- Here, you must initialize `UTXBUF0` to refer to a `device_register`.
- However, the cast above yields a "pointer to `device_register`", not a `device_register`.
- You must dereference the result of the cast to obtain an object to which the reference can bind.

Reference-Placement

- Using reference-placement, you can treat `UTXBUF0` as the register itself, not a pointer to the register, as in:

```
UTXBUF0 = c;    // OK: send the value of c out the port
```

- A reference acts like a const pointer in that:
  - You must bind the reference to an object at initialization.
  - After that, you can't rebind it to another object.

New-Style Casts

- C++ provides an alternative notation that distinguishes portable casts from potentially non-portable ones:

```
                             // behavior:
static_cast<T>(e)            //     explicitly specified
reinterpret_cast<T>(e)       //     implementation-defined
const_cast<T>(e)             //     const only
```

- These “new-style” casts don’t provide any additional functionality beyond the “old-style” casts.
- They just offer a chance to write code that reveals its intent more clearly.

New-Style Casts

- In general, you should avoid using casts.

✓ *If you must use a cast in C++, use a new-style cast.*

- These new-style casts are easier to spot in source code:
  - for humans and
  - for search tools such as `grep`.
- This is good, because casts are hazardous.
- A hazard that's easier to spot is usually less of a hazard.

New-Style Casts

- Using a new-style cast, the definition for `UART0` now looks like:

```
UART *const UART0 =
    reinterpret_cast<UART *>(0x03FFD000);
```

- As a reference, it looks like:

```
UART &UART0 =
    *reinterpret_cast<UART *>(0x03FFD000);
```

Traditional Register Representation

- C programs often define symbols for device register addresses as clusters of related macros:

```
// timer registers
#define TMOD   ((unsigned volatile *)0x3FF6000)
#define TDATA  ((unsigned volatile *)0x3FF6004)
~~~

// UART registers
#define ULCON0 ((unsigned volatile *)0x3FFD000)
#define UCON0  ((unsigned volatile *)0x3FFD004)
~~~
```

- This approach has a couple of weaknesses…

Traditional Register Representation

- It leads to awkward and inconvenient interfaces.
  - Many device operations involve more than one device register.
  - You either have to pass more than one register address, as in:

```
UART_put(USTAT0, UTXBUF0, c);
```

  - Or worse, you must treat all device registers as global objects.
- It leads to error-prone interfaces.
  - All device register addresses have the same pointer type.
  - Type checking can't catch accidents such as:

```
UART_put(USTAT0, TDATA, c);  // put c to a timer?
```

Using Structures and Classes

✓ ***Use structures in C and classes in C++ to implement abstract types.***

- A structure or class provides a more accurate declaration for the collection of registers for a given device:

```
struct UART {
    device_register ULCON;
    device_register UCON;
    device_register USTAT;
    ~~~
};

void UART_put(UART *u, int c);
```

Using Structures and Classes

- Each structure or class is a distinct type.
- You can't convert a pointer to one into a pointer to another without using a cast.
- Type checking can now catch accidents such as:

```
UART *const com0 = reinterpret_cast<UART *>(0x3FFD000);
timer *const timer0 =
    reinterpret_cast<timer *>(0x3FF6000);

UART_put(timer0, c);    // put c to a timer?  no
```

- Let's apply this approach to the UART in greater detail…

Modeling Devices

- As mentioned earlier, many UART operations involve accessing more than one UART register.
- For example:
  - The TBE bit (Transmit Buffer Empty) in the USTAT register indicates whether the UTXBUF register is ready for use.
  - You shouldn't store a character into UTXBUF until the TBE bit is set to 1.
  - Storing a character into UTXBUF initiates output to the port and clears the TBE bit.
  - The TBE bit goes back to 1 when the output operation completes.

Modeling Devices

- Here's code that waits for UART 0's transmit buffer to empty before sending it a character:

```
while ((USTAT0 & TBE) == 0)
    ;
UTXBUF0 = c;
```

- The sample hardware has two UARTs — any UART operation that works on one UART works on the other.
- Again, you shouldn't pass UART registers (such as `USTAT` and `UTXBUF`) as separate function arguments.
- Rather, collect all the UART registers into a single structure and pass a pointer or reference to that structure…

Modeling Devices

- Here’s a structure representing the UART registers:

```
struct UART {
    device_register ULCON;
    device_register UCON;
    device_register USTAT;
    device_register UTXBUF;
    device_register URXBUF;
    device_register UBRDIV;
};

enum { TBE = 0x40 };  // mask for TBE bit in USTAT
```

Modeling Devices

- Now you can define the two UARTs as:

```
UART *const UART0 = reinterpret_cast<UART *>(0x03FFD000);
UART *const UART1 = reinterpret_cast<UART *>(0x03FFE000);
```

- You can pass either `UART0` or `UART1` to a function that sends characters from a null-terminated character sequence to a UART:

```
UART_put(UART0, "hello, world\n");
UART_put(UART1, "goodnight, moon\n");
```

Modeling Devices

- Here’s that function:

```
void put(UART *u, char const *s) {
    for (; *s != '\0'; ++s) {
        while ((u->USTAT & TBE) == 0)
            ;
        u->UTXBUF = *s;
    }
}
```

Modeling Devices

- Here’s it is again using a reference parameter in C++:

```
void put(UART &u, char const *s) {
    for (; *s != '\0'; ++s) {
        while ((u.USTAT & TBE) == 0)
            ;
        u.UTXBUF = *s;
    }
}
```

- These functions might work.
- Then again, they might not…

Overly Aggressive Optimization

- It might not work properly if the compiler's optimizer is too aggressive.
- Ordinarily, an object's value doesn't change unless the program changes it.
  - The compiler uses this knowledge when optimizing code.
- Hardware registers aren't ordinary objects.
  - The value of `u.USTAT` can change due to external hardware events.
  - The compiler doesn't know this.
  - How can we tell it?

The Volatile Qualifier

- Declaring an object `volatile` informs the compiler that the object may change state even though the program didn't change it.
- Specifically, declaring an object `volatile`:
  - tells the compiler to treat each access (read or write) to that object literally
  - prevents the compiler from "optimizing away" accesses to that object, even when it seems safe to do so

The Right Dose of Volatility

- This declares UART0 as a "const pointer to a volatile UART":

```
UART volatile *const UART0 =
    reinterpret_cast<UART *>(0x03FFD000);
```

- This declares UART1 as a "reference to a volatile UART".

```
UART volatile &UART1 =
    *reinterpret_cast<UART *>(0x03FFE000);
```

- In these declarations, volatile isn't part of the UART type.
- This is appropriate only if some UARTs aren't volatile.
- If volatility is inherent in every UART, volatile should be part of the UART type…

The Right Dose of Volatility

- You could try declaring the entire UART type volatile, as in:

```
volatile struct UART {
    ~~~
};  // error: missing declarator
```

- It won't compile because the compiler wants to apply the volatile keyword to a UART object, as in:

```
volatile struct UART {  // type UART isn't volatile...
    ~~~
} u;                    // but object u is
```

The Right Dose of Volatility

- The compiler would also be happy applying `volatile` to a typedef name, as in either:

```
typedef volatile struct { // no struct tag
    ~~~
} UART;

typedef struct {          // no struct tag
    ~~~
} volatile UART;
```

- Either way, `UART` is now a volatile type, which could be what we want.
- But maybe not…

The Right Dose of Volatility

- Typically, all device registers (not just those in UARTs) are volatile.
- In that case, you should define `device_register` as a volatile type:

```
typedef uint32_t volatile device_register;
```

The Right Dose of Volatility

- If `device_register` is a volatile type, then you can revert to the original definition for `UART`:

```
struct UART {
    device_register ULCON;
    device_register UCON;
    device_register USTAT;
    ~~~
};
```

- This `UART` isn't a volatile type; however…

The Right Dose of Volatility

- Although `UART` isn't volatile, every non-static `UART` data member is volatile, which is really what we need.
- This actually simplifies using the `UART` type.
  - For example, you'll never need to declare a pointer or reference to a `UART` as a pointer or reference to a `volatile UART`.
- This form works fine in C:

```
UART *const UART0 = (UART *)0x03FFD000;
```

- This form is also just fine in C++:

```
UART &UART0 = *reinterpret_cast<UART *>(0x03FFD000);
```

Controlling a UART

- You can use a C++ class to package the UARTs as a abstract type.
- Our basic UART supports the following operations:
  - enable the UART
  - disable the UART
  - receive data from the UART
  - ask if the UART is ready to send data
  - send data to the UART