Lecture 3 – Atomics and Memory Model in Modern C++

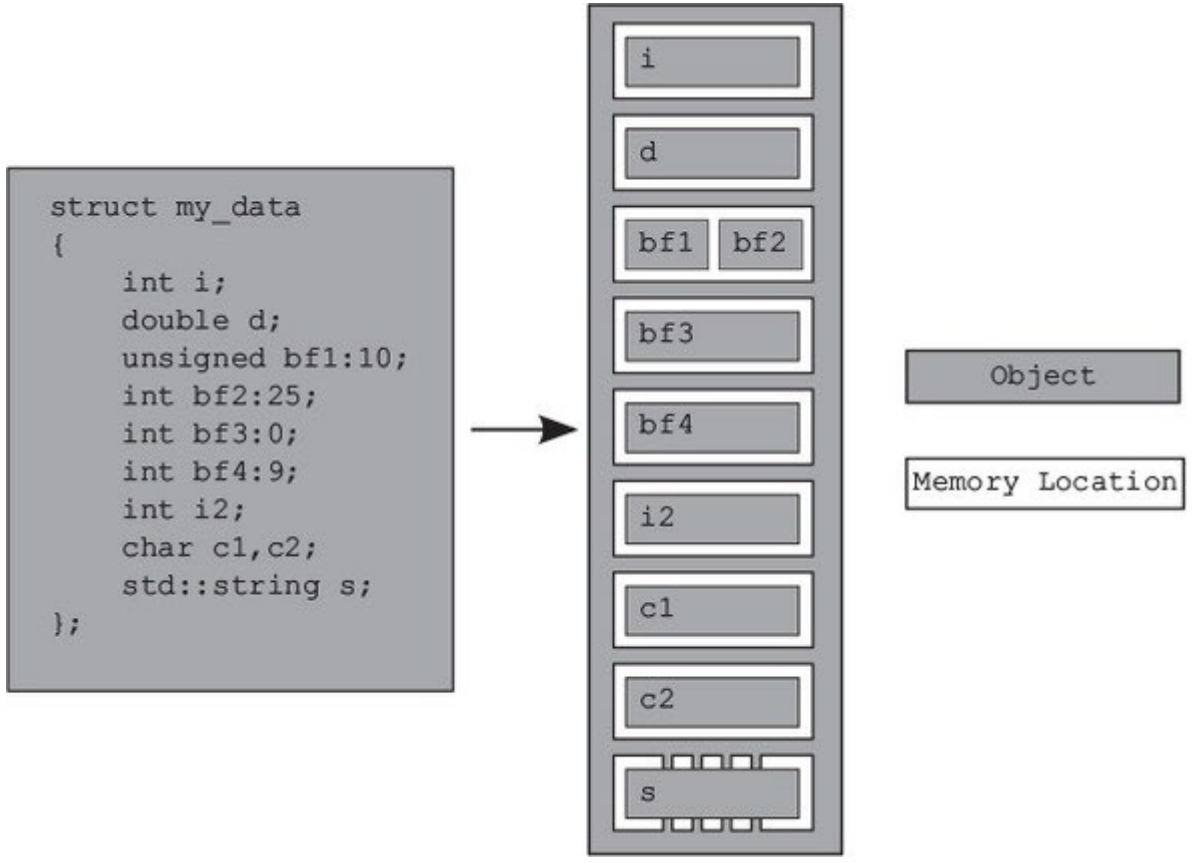
Additional Examples & Questions

Outline

- Questions
- From last week: atomic weapons

Structure

- Every variable is an object, including those that are members of other objects.
- Every object occupies *at least one* memory location.
- Variables of fundamental types such as `int` or `char` occupy *exactly one* memory location, whatever their size, even if they're adjacent or part of an array.
- Adjacent bit fields are part of the same memory location.

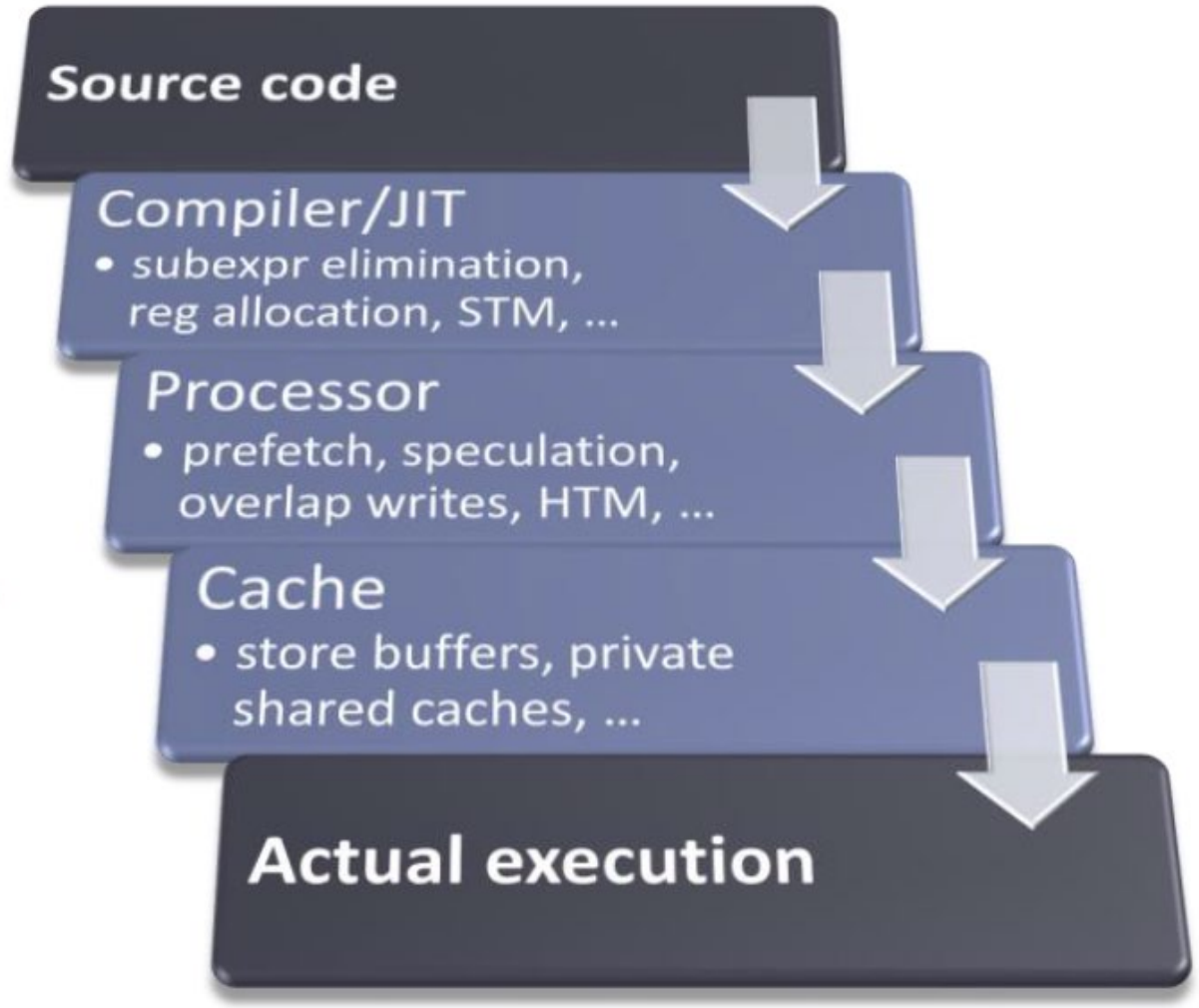


Language level memory models

- Modern (C++11) and not-so-modern (Java 5) languages guarantee **sequential consistency for data-race-free programs ("SC for DRF")**
  - Compilers will insert the necessary synchronization to cope with the hardware memory model
- No gua Only when using memory_order_seq_cst on all our atomics!
  - The intuition is that most programmers would consider a racy program to be buggy
- Use synchronization!
  - L2: Mutex & condition variable
  - L3: Atomics

Language-level vs. Hardware-level Memory Consistency Models

- Do single threaded apps benefit from having a memory model?
- When do we use sequential, rel-acq, relaxed models?



Example 1

Initially: x = 0; y = 0; z = 0;
Should this code ever print 0?

```
x = 0;
y = 0;
z = 0;
```

Thread 1:

```
x.store(1, memory_order_relaxed);

y.store(2, memory_order_release);
```

Thread 2:

```
while (y.load(memory_order_acquire)!=2);

y.store(3, memory_order_relaxed);

z.store(4, memory_order_release);

x.store(5, memory_order_relaxed);
```

Thread 3:

```
while (z.load(memory_order_acquire)!=4);

cout << x.load(memory_order_relaxed);
```

Example 1

Initially: x = 0; y = 0; z = 0;
Should this code ever print 0?

```
x = 0;
y = 0;
z = 0;
```

Thread 1:

```
x.store(1, memory_order_relaxed);

y.store(2, memory_order_release);
```

Thread 2:

```
while (y.load(memory_order_acquire)!=2);

y.store(3, memory_order_relaxed);

z.store(4, memory_order_release);

x.store(5, memory_order_relaxed);
```

Thread 3:

```
while (z.load(memory_order_acquire)!=4);

cout << x.load(memory_order_relaxed);
```

http://svr-pes20-cppmem.cl.cam.ac.uk/cppmem/

```
int main() {
  atomic_int x=0;
  atomic_int y=0;
  atomic_int z=0;
  {{{ { x.store(1, relaxed); y.store(2, release); }
  ||| { y.load(acquire).readsvalue(2); y.store(3, relaxed); z.store(4, release);
x.store(5, relaxed); }
  ||| { z.load(acquire).readsvalue(4); r1 = x.load(relaxed); }
  }}};
  return 0;
}
```

Example 2

Initially: x = 0; y = 0; z = 0;
Should this code ever print 0?

```
x = 0;
y = 0;
z = 0;
```

Thread 1:

```
x.store(1, memory_order_relaxed);

y.store(2, memory_order_release);
```

Thread 2:

```
r1 = y.load(memory_order_acquire);

y.store(3, memory_order_relaxed);

z.store(4, memory_order_release);

x.store(5, memory_order_relaxed);
```

Thread 3:

```
while (z.load(memory_order_acquire)!=4);

cout << x.load(memory_order_relaxed);
```

Example 3

Initially: x = 0; y = 0; z = 0;
x is not atomic
Should this code ever print 0?

```
x = 0;
y = 0;
z = 0;
```

Thread 1:

```
x = 1

y.store(2, memory_order_release);
```

Thread 2:

```
while (y.load(memory_order_acquire)!=2);

y.store(3, memory_order_relaxed);

z.store(4, memory_order_release);

x = 3;
```

Thread 3:

```
while (z.load(memory_order_acquire)!=4);

cout << x;
```

Fences

- Operations that enforce memory-ordering constraints without modifying any data and are typically combined with atomic operations that use the `memory_order_relaxed` ordering constraints
- *Memory barriers*
  - Put a line in the code that certain operations can't cross
- An `atomic_thread_fence` with `memory_order_release` ordering prevents all preceding reads and writes from moving past all subsequent stores
  - Remember: an atomic store-release operation prevents all preceding reads and writes from moving past the store-release

Atomic_thread_fence

- Line 28: assert never fires
- Line 9 synchronizes-with 15
- Line 8 happens-before 16

- Swap 8 and 9?
  - Assert can fire

```
1  #include <atomic>
2  #include <thread>
3  #include <assert.h>
4  std::atomic<bool> x,y;
5  std::atomic<int> z;
6  void write_x_then_y()
7  {
8      x.store(true,std::memory_order_relaxed);
9      std::atomic_thread_fence(std::memory_order_release);
10     y.store(true,std::memory_order_relaxed);
11 }
12 void read_y_then_x()
13 {
14     while(!y.load(std::memory_order_relaxed));
15     std::atomic_thread_fence(std::memory_order_acquire);
16     if(x.load(std::memory_order_relaxed))
17         ++z;
18 }
19 int main()
20 {
21     x=false;
22     y=false;
23     z=0;
24     std::thread a{write_x_then_y};
25     std::thread b{read_y_then_x};
26     a.join();
27     b.join();
28     assert(z.load()!=0);
29 }
```

Terminology

- In lecture 3
  - We preferred to say “operation seen by a thread…”
- In general, memory model is explained by mentioning
  - What ordering is allowed and what ordering is not allowed
  - Visible side effects
  - https://en.cppreference.com/w/cpp/atomic/memory_order