# Lecture 10—C++ atomics; Compilers and You

January 26, 2015

# Roadmap

Last Time: Synchronization Mechanisms
This Time: C++ atomics; C Compilers; Dependencies

# Part I

## C++ atomics

# About C++ atomics

You can use the default `std::memory_order`.
(= sequential consistency)

Don't use relaxed atomics unless you're an expert!

`http://stackoverflow.com/questions/9553591/c-stdatomic-what-is-stdmemory-order-and-how-to-use-them`

# Really, don't use C++ relaxed atomics!

# C++ atomics: Key Idea

An *atomic operation* is indivisible.

Other threads see state before or after the operation, nothing in between.

# Simplest: atomic_flag

Represents a boolean flag.

```
#include <atomic>

atomic_flag f = ATOMIC_FLAG_INIT;
```

# Operations on `atomic_flag`

Can clear, and can test-and-set:

```
#include <atomic>

atomic_flag f = ATOMIC_FLAG_INIT;
int foo() {
  f.clear();
  if (f.test_and_set()) {
    // was true
  }
}
```

`test_and_set`: atomically sets to true,
returns previous value.

No assignment (=) operator.

# Using more general C++ atomics

Declaring them:

```
#include <atomic>

atomic<int> x;
```

Libary's implementation:

- on small types, lock-free operations;
- on large types, mutexes.

# What to do with Atomics

Kinds of operations:

- reads
- writes
- read-modify-write (RMW)

# Reads and Writes

C++ has syntax to make these all transparent:

```
#include <atomic>
#include <iostream>

std::atomic<int> ai;
int i;

int main() {
    ai = 4;
    i = ai;
    ai = i;
    std::cout << i;
}
```

Can also use `i = ai.load()` and `ai.store(i)`.

# Read-Modify-Write (RMW)

Consider `ai++`.

This is really

```
tmp = ai.read(); tmp++; ai.write(tmp);
```

Hardware can do that atomically.

Other RMWs: `+-`, `&=`, `etc`, `compare-and-swap`

more info:
`http://preshing.com/20130618/atomic-vs-non-atomic-operations/`

# Part II

# Making C Compilers Work For You

# Three Address Code

- An intermediate code used by compilers for analysis and optimization.
- Statements represent one fundamental operation—we can consider each operation atomic.
- Statements have the form:
  $$result := operand_1\ operator\ operand_2$$
- Useful for reasoning about data races, and easier to read than assembly.
  (separates out memory reads/writes).

# GIMPLE

- GIMPLE is the three address code used by `gcc`.
- To see the GIMPLE representation of your code use the `-fdump-tree-gimple` flag.
- To see all of the three address code generated by the compiler use `-fdump-tree-all`. You'll probably just be interested in the optimized version.
- Use GIMPLE to reason about your code at a low level without having to read assembly.

# Live Coding Demo: GIMPLE

# Branch Prediction Hints

As seen earlier in class, `gcc` allows you to give branch prediction hints by calling this builtin function:

```
long __builtin_expect (long exp, long c)
```

The expected result is that `exp` equals `c`.

Compiler reorders code & tells CPU the prediction.

# The `restrict` Keyword

A new feature of C99: "The restrict type qualifier allows programs to be written so that translators can produce significantly faster executables."

- To request C99 in `gcc`, use the `-std=c99` flag.

`restrict` means: you are promising the compiler that the pointer will never alias (another pointer will not point to the same data) for the lifetime of the pointer.

# Example of `restrict` (1)

Pointers declared with `restrict` must never point to the same data.

From Wikipedia:

```
void updatePtrs(int* ptrA, int* ptrB, int* val) {
    *ptrA += *val;
    *ptrB += *val;
}
```

Would declaring all these pointers as `restrict` generate better code?

# Example of `restrict` (2)

Let's look at the GIMPLE:

```
1  void updatePtrs(int* ptrA, int* ptrB, int* val) {
2   D.1609 = *ptrA;
3   D.1610 = *val;
4   D.1611 = D.1609 + D.1610;
5   *ptrA = D.1611;
6   D.1612 = *ptrB;
7   D.1610 = *val;
8   D.1613 = D.1612 + D.1610;
9   *ptrB = D.1613;
10 }
```

- Could any operation be left out if all the pointers didn't overlap?

# Example of `restrict` (3)

```
1  void updatePtrs(int* ptrA, int* ptrB, int* val) {
2   D.1609 = *ptrA;
3   D.1610 = *val;
4   D.1611 = D.1609 + D.1610;
5   *ptrA = D.1611;
6   D.1612 = *ptrB;
7   D.1610 = *val;
8   D.1613 = D.1612 + D.1610;
9   *ptrB = D.1613;
10 }
```

- If `ptrA` and `val` are not equal, you don't have to reload the data on **line 7**.
- Otherwise, you would: there might be a call

  `updatePtrs(&x, &y, &x);`

# Example of `restrict` (4)

Hence, this markup allows optimization:

```
void updatePtrs(int* restrict ptrA,
                int* restrict ptrB,
                int* restrict val)
```

Note: you can get the optimization by just declaring `ptrA` and `val` as `restrict`; `ptrB` isn't needed for this optimization

# Summary of `restrict`

- Use `restrict` whenever you know the pointer will not alias another pointer (also declared `restrict`)

It's hard for the compiler to infer pointer aliasing information; it's easier for you to specify it.

$\Rightarrow$ compiler can better optimize your code (more perf!)

Caveat: don't lie to the compiler, or you will get undefined behaviour.

Aside: `restrict` is not the same as `const`. `const` data can still be changed through an alias.