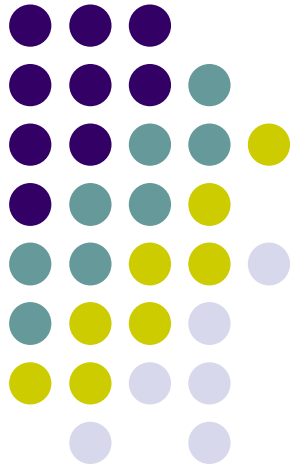
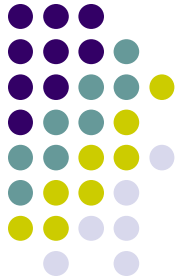
C++

March 22, 2024

Mike Spertus

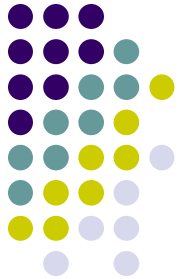
spertus@uchicago.edu





C++ AS A LIVING LANGUAGE

One of the most defining characteristics of C++

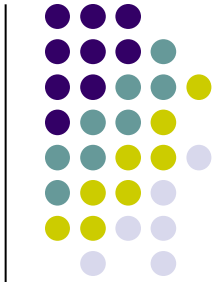


- C++ has been widely used since the 1980s
- But remains a cutting edge language today
- It is not clear if the same can be said about any other language
  - Languages like C would look very familiar to a time traveler from the 1980s
  - Languages like Rust are cutting edge but can't leverage half a century-long codebase
- Indeed, when standards committee members considered creating an academic journal, we were going to name it "The Journal of Language Evolution"

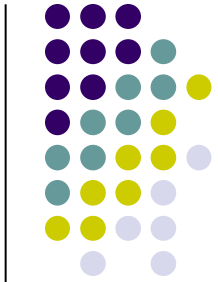
Special Guest: Inbal Levi

- I am currently at the C++ standards meeting
- Inbal Levi has kindly volunteered to explain how this evolution happens and some thing to look forward to in C++26
- Inbal is eminently qualified
  - Chair of the Library Evolution Working Group
  - Director of ISO C++ and Boost Foundations
  - Israeli National Body Chair
  - Core C++ Conference Organizer
- Inbal's slides: https://slides.com/inballevi/deck/fullscreen

Reflection: Ahead of Schedule?

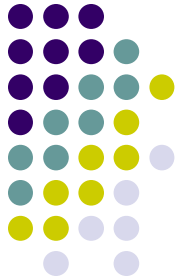


- Reflection is a huge need for C++
  - The ability to examine the structure of classes, functions, etc. from within the language
- Essential for features like serialization and automatic generation of class hierarchies
- No one expected in C++26
- But let's see this presentation from Daveed Vandevoorde to the committee
  - https://docs.google.com/presentation/d/1plTna9qreBPu6G4ZWBUMBq_XmqJ5qAVFswWRGqp_xrw
- Proposal is at https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2023/p2996r0.html



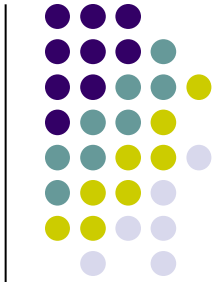
COURSE INFO

What is this course?



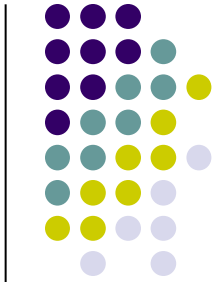
- Advanced C++
- As with last quarter, we really have two goals
  - Learn Advanced C++
  - Use C++ as "an excuse" to learn advanced programming techniques that apply to any languages

Additional resources



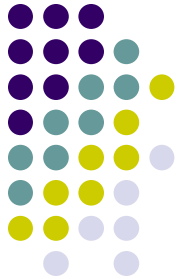
- Office hours
  - 10-11:30 Sundays on Zoom
  - 4-5 Thursdays in JCL298F and Zoom
- TA: Chris Brown
  - Office Hours TBD
- Ed discussions
  - Reach directly or from the Canvas tab

The most important rule



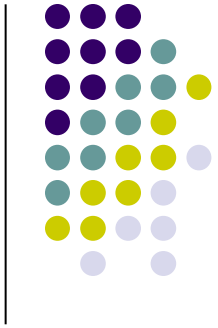
- If you are ever stuck or have questions or comments
- Be sure to contact me
  - spertus@uchicago.edu
  - Or on ed

Homework and Lecture Notes

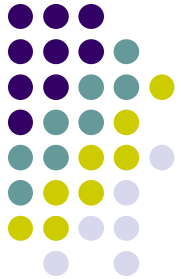


- Homework and lecture notes posted on Canvas
  - Choose MPCS 51045 and then go to “Pages”
- HW due on the following Thursday before class
- Submit on Gradescope
  - Let me know if you can't access through the Canvas Gradescope tab
- Homework will be graded by the start of the following class
- ☹
  - Since I go over the answers in class, **no late homework will be accepted**
- ☺
  - If you submit by Sunday evening, you will receive a grade and comments back by midnight on Tuesday, so you can try submitting again

Course grade

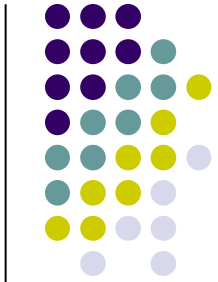


- 2/3 HW
  - Many extra credit opportunities
  - Extra credit can get your HW total for the quarter to 100% (but no higher) to cancel out any problems you miss
- 1/3 Final Project
  - Practice the techniques from the class on a C++ project on your choice
  - If you did a final project last quarter, you can build on that or start from scratch
- **Notes:**
  - Even though HW is 2/3 of grade, the final will have more impact because the variance is bigger
  - I do not use a 70/80/90 scale
  - Feel free to ask me if you have questions



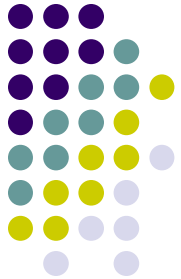
FUNDAMENTAL TYPES

Fundamental Types



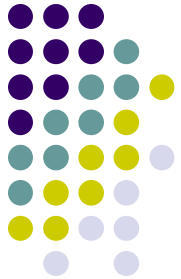
- The topic of this section is the fundamental types of C++
  - Integral types, floating point types, character types
- This is important for two reasons
  - All types are built up from fundamental types
    - To use a chemistry analogy, think of them as the “atoms” of your datatypes
  - They are where the “rubber meets the road” on interacting with native hardware, so they are a great place to look at the tradeoffs of trying to maximize efficiency on real hardware instead of virtual machines

C++ fundamental types



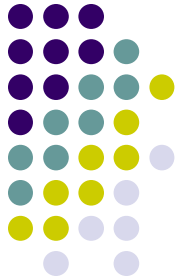
- As we mentioned, C++ fundamental (built-in) types are not fully specified in order to best match the hardware architecture they are running on
  - At least in theory
- In this lecture, we will dig into the basic types that C++ comes with
  - After all, they form the building blocks for all other types
- But first, let's try to understand what this "Schrödinger's standardization" means

How C++ deals with behavior that is not fully specified



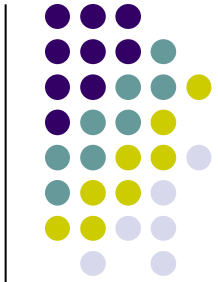
- Just because the behavior of code is not fully specified by the standard, it doesn't mean that the standard has nothing to say about it
- There are three kind of behaviors that can be ascribed to code that is not fully specified
  - Undefined behavior
  - Unspecified behavior
  - Implementation-defined behavior

Undefined Behavior



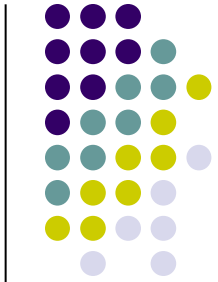
- The least specified is “undefined behavior,” which means the computer can do anything, up to and include “Halt and Catch Fire”
- Reading a random memory address is a good example of undefined behavior, as the program is likely to crash
- Newer language features tend to have less undefined behavior as more powerful hardware and more complex programs have increased the emphasis on safety over “that last bit of performance”
  - For example, a C++ variant is safer but slower than a C union

Unspecified behavior



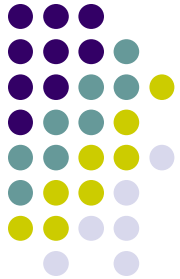
- It's not "anything goes" like undefined behavior, but the implementation has several choices
- For example, if you write 0.1+0.2, it is unspecified whether this is calculated at compile-time or run-time
- Since floating point arithmetic behaves differently depending on hardware, this means that it is unspecified whether the following is true
  - 0.1 + 0.2 == 0.1 + 0.2 // May not be true!
- Still, the computer can't halt and catch fire!

Implementation-defined behavior

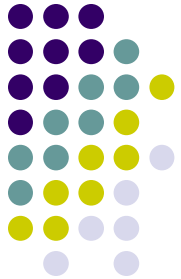


- Implementation-defined behavior is similar to unspecified behavior except that the implementation needs to specify what it does
- For example, C++ doesn't say whether `char` is signed or unsigned, but any particular compiler needs to document which it is
- We'll see some ways to find this out later in the course

The better-defined, the better



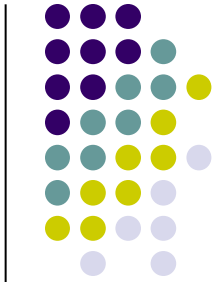
- When you can, rely on specified behavior
  - E.g., prefer C++ variants over unions unless there is a strong reason not to
- If you can’t, prefer implementation-defined behavior over unspecified behavior
- Avoid undefined behavior where you can
- At least your computer won’t halt and catch fire!



A DEEP DIVE INTO INTEGERS

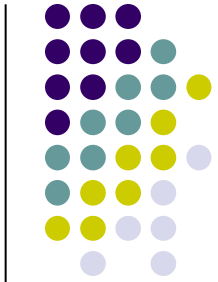
There’s more to them than you think!

What could be simpler than integral types?



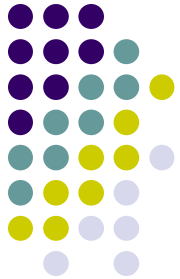
- After all, Java only has
  - byte (8 bits)
  - short (16 bits)
  - int (32 bits)
  - long (64 bits)
- Two’s complement is guaranteed
- No unsigned types!

Well, not so fast



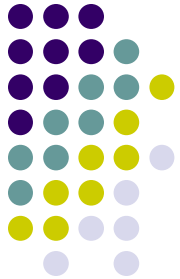
- As we will see, C++ has some simple integral types, but it also needs to work natively with low-level and specialized hardware
- Let's see how it handles it
  - Don't be scared by all the options, they're rarely used
  - I just want to give you an idea of how C++ can support specialized domains without running into a wall, in case you ever have the need
    - E.g, if you're programming DSPs

C++ has some basic types



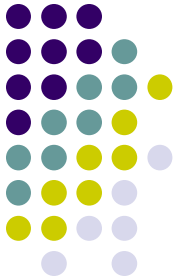
- Short types: short, unsigned short
  - At least 16 bits
- Int types: int, unsigned int
  - At least 16 bits
- Long types: long, unsigned long
  - At least 32 bits
- Long long types: long long, unsigned long long
  - At least 64 bits
- You can be more specific without changing the type
  - short is a synonym for signed short int

How do you know what can fit in an integral type?



- Unlike Java, these do not have fixed size
- This is implementation-defined (beyond the minimum size)
- You can check with `std::numeric_limits`
- `numeric_limits<int>::max()` is the biggest `int`

What is encountered in practice?

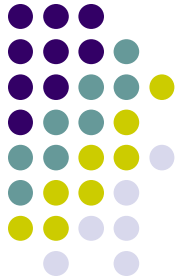


- In practice, there are four common data models (today)

	LP32	ILP32	LLP32	LLP64
short	16	16	16	16
int	16	32	32	32
long	32	32	32	64
long long	64	64	64	64

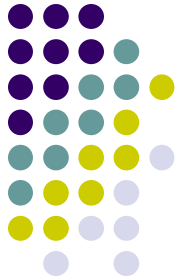
- Note that a long may not be any bigger than an int!

This is confusing, what if I just need a fixed size?



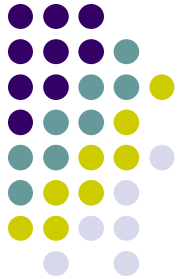
- If you're really concerned about the size, it's best to specify it directly
- There are fixed length integers
  - int8_t, uint8_t, int16_t, uint16_t, int32_t, uint32_t, int64_t, uint64_t
  - These also guarantee 2's complement
- However, it is implementation-defined whether these are available
- Most general purpose platforms support them
  - The only still extant systems that don't use two's complement are some Unisys systems
- Specialized platforms like SHARC DSP's, which only support 32 bit integers will not have them all

What if I need at least a certain size and portability?



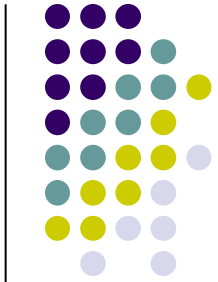
- C++ gives you two choices
- The fastest integers with at least a given width
  - int_fast8_t, uint_fast8_t, int_fast16_t,…
- The smallest integers with at least a given width
  - int_least8_t, uint_least8_t, int_least16_t,…
- An implementation can also define other widths if it makes sense for the target hardware

Specialized integer types

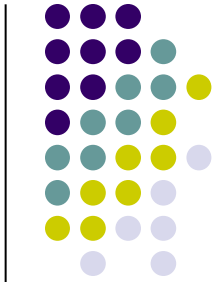


- intmax_t, uintmax_t
  - The maximum-width types
- intptr_t, uintptr_t
  - Integral types capable of holding memory addresses
- size_t is an unsigned integral type able to represent sizes of objects
  - Returned by operators like sizeof
  - How wide typically depends on the size of the address space

What to make of all this?

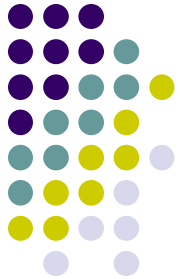


- It’s easy to understand why people say C++ is complicated
- But most of these are rarely used, so maybe not harder than other languages in the usual case
- Consistent with “Make the simple cases easy and the hard things possible”

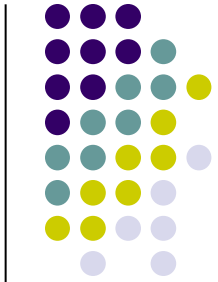


FLOATING POINT TYPES

Floating point types

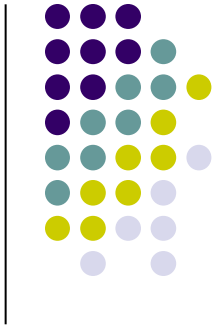


- float
  - IEEE-754 32 bit floating point type (rarely used)
- double
  - IEEE-754 64 bit floating point type
  - Floating point literals like 4.5 are doubles (you can get a float with 4.5f)
- long double
  - Implementation-defined but usually 80 bit x87 floating point types on x86 and x86-64
- For a variety of reasons (mainly the inability of floats to exactly represent decimals like 0.3), you should never compare if two floating point numbers are equal
  - We saw an example of this above
- Even shorter floating point types (e.g., float16) are becoming popular for compressing neural networks, but at present only exist as compiler-extensions

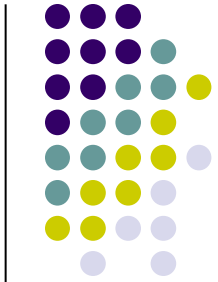


BOOLEANS

bool



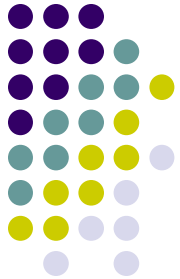
- A bool an be either true or false
- If you assign them to an integer, false is 0 and true is an implementation-defined non-zero integer
  - Usually 1
- Comparisons are a good way to get Booleans
  - 5 == 3 is false



CHARACTER TYPES

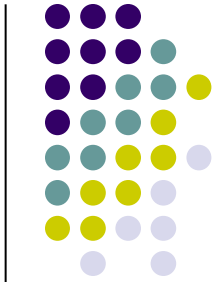
Like integers, these are complicated

char



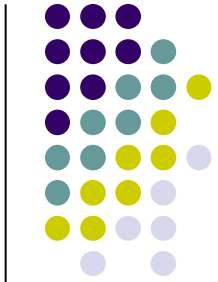
- The char type is the smallest type (i.e., C++' byte)
  - sizeof(char) is guaranteed to be 1, which usually (implementation-defined!) is 8 bits
- This means that the memory behind any object can be considered as an array of chars
- Because it serves as this basic unit, its "signedness" is implementation-defined and typically depends on the processor
  - std::numeric_limits<char>::is_signed
  - Usually unsigned on ARM and signed on x86
- Use signed char or unsigned char if you need a specific signedness

wchar_t: Tempting but usually avoid



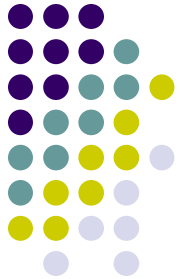
- wchar_t is supposed to represent any character (e.g., ओ)
- It is (appropriately) 32 bits on most platform, but 16 bits on Windows, which isn't wide enough to represent every Unicode code point in a single wchar_t, so it takes more space than a char but doesn't solve the problem of representing every character
  - This is because Windows OS standardized on 16-bit characters before Unicode grew too big
  - Java also has this problem

Unicode character types

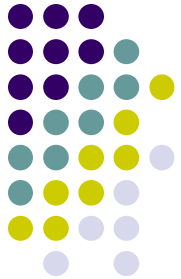


- `char8_t`, `char16_t`, `char32_t` are meant to be character units for UTF-8, UTF-16, and UTF-32
- When you need wide characters, `char32_t` is better than `wchar_t`
- For now at least, the standard library's Unicode support is not great, so expect rough edges
  - E.g., `strings` are made of `chars` and not `char8_ts`, don't understand UTF-8 concepts like surrogate pairs, etc.
  - There is a Unicode Study Group to improve, but for now, generally stick with `char` over `char8_t`

Additional fundamental types



- `void`
  - A type with no possible values
  - You cannot create a variable of type `void`
  - Often used as a return type for functions that don't return anything
- `nullptr_t`
  - A type representing a "null pointer"
  - We will discuss this when we get to pointers

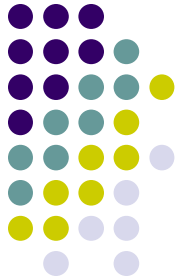


ADVANCED NOTIFICATIONS

From winter quarter `condition_variable`

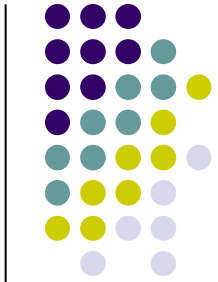
- Last quarter, we discussed the use of the powerful class `condition_variable` to signal events between threads
- Unfortunately, they have limitations that can sometimes be a problem
- Let's start with a quick refresher on `condition_variable`

Sometimes locks aren't what you want



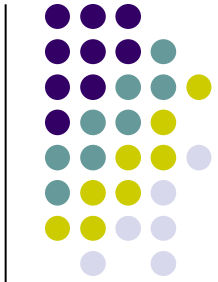
- Suppose we are trying to implement a "producer/consumer" design pattern
  - Think of this as a supply chain
  - Some threads produce work items that are consumed by other threads
  - Incredibly common in multi-threaded programs
- Typically the producers put work onto a queue, and the consumers take them off
- Locks can allow thread-safe access to the queue
- But what happens if there is no work at the moment?
  - The consumer thread needs to go to sleep and wake up when there is work to do
  - Rather than a lock, you'd like to wait for an "event" stating that the queue has become non-empty

Producer-consumer implementation



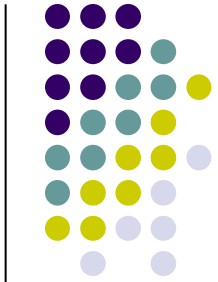
- We will use a couple of new library features
  - `unique_lock`,
    - a richer version of `lock_guard`
  - `condition_variable`
    - “wakes up” waiting threads

Condition variables



- The C++ version of an event
  `condition_variable cv;`
- You can wait for a condition variable to signal
```
mutex m;
boolean test();
unique_lock<mutex> lck(m);
cv.wait(lck, test);
```
- If the test succeeds, the wait returns immediately, otherwise it unlocks m (that's why we used a `unique_lock` instead of `lock_guard`)
- Once the `condition_variable` signals the waiting thread (we'll see how in a moment)
  - The lock is reacquired
  - The test is rerun (if it fails, we wait again)
    - Protects against spurious wakeups
  - Once the test succeeds, the program continues

Signaling an event is simple

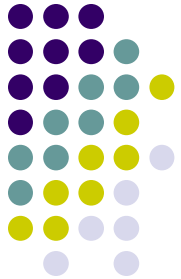


- `cv.notify_one();`
  - Wakes one waiter
    - No guarantees which one
- `cv.notify_all();`
  - Wakes all waiters

Producer-consumer from Williams' *C++ Concurrency in Action*

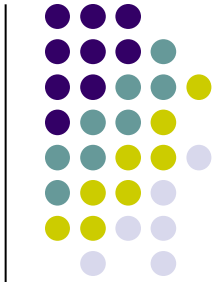
```
mutex m;
queue<data_chunk> data_queue;
condition_variable cond;

void data_preparation_thread()
{
  while(more_data_to_prepare()) {
    data_chunk const data=prepare_data();
    lock_guard lk(m);
    data_queue.push(data);
    cond.notify_one();
  }
}

void data_processing_thread()
{
  while(true) {
    unique_lock<mutex> lk(m);
    cond.wait(lk,[]{return !data_queue.empty();});
    data_chunk data=move(data_queue.front());
    data_queue.pop();
    lk.unlock();
    process(data);
    if(is_last_chunk(data))
      break;
  }
}
```

condition_variable is a very useful class



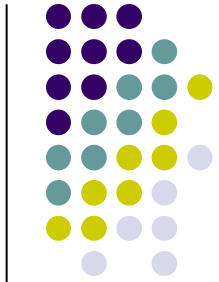
- In addition to implementing producer-consumer queues
- We also used `condition_variable` to create our own implementation of futures and promises
- See last quarter's notes for details

So what's wrong?



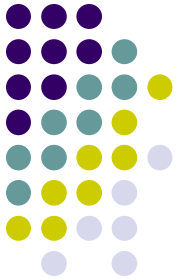
- Sometimes there are situations where it seems like you want a `condition_variable`
- But they are either error-prone or not fit for purpose at all
- Let's see what can go wrong

Waking a set of threads



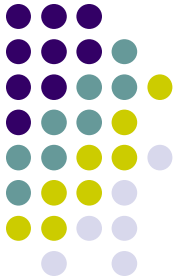
- It is common to want to wake up a set of threads when an event occurs
- In the final exam last quarter, you were asked to run a virtual car race, where each virtual car ran in its own thread
- For a fair race, all threads need to wait for the start of the race to be signaled and then start together
- Let’s look at one possible (acceptable but not optimal) solution

Starting the race



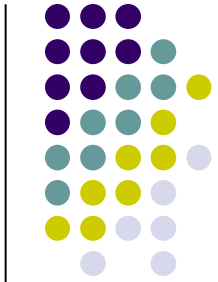
```
bool race_started{};
mutex race_mutex;
condition_variable cv;
// Racing thread function
void run_race(/* some args */)
{
  unique_lock lck(race_mutex);
  cv.wait(lck, [&] { return race_started; });
  lck.unlock();
  /* ... */      // Racing code
}
// Main thread
int main() {
  /* ... */      // Create threads
  unique_lock lck(race_mutex);
  race_started = true;
  lck.unlock(); // Would be better/safer to use the RAII approach above
  cv.notify_all();
  /* ... */
}
```

A common mistake



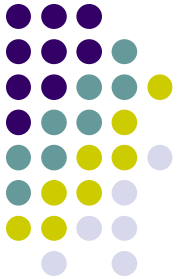
- The most common mistake is to forget to release the lock in the thread function

Incorrect solution
No parallelism!



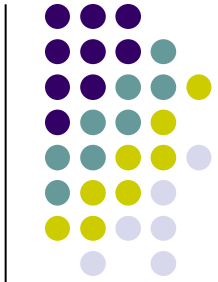
```
bool race_started{};
mutex race_mutex;
condition_variable cv;

// Racing thread function
void run_race(/* some args */)
{
  unique_lock lck(race_mutex);
  cv.wait(lck, [&] { return race_started; });
  lck.unlock();
  /* ... */      // Racing code
}

// Main thread
int main() {
  /* ... */      // Create threads
  unique_lock lck(race_mutex);
  race_started = true;
  lck.unlock();  // This one sometimes missed as well
  cv.notify_all();
  /* ... */
}
```

Incorrect solution



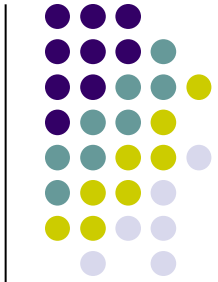
```
bool race_started{};
mutex race_mutex;
condition_variable cv;

// Racing thread function
void run_race(/* some args */)
{
  unique_lock lck(race_mutex);
  cv.wait(lck, [&] { return race_started; });

  /* ... */     // Racing code
}

// Main thread
int main() {
  /* ... */     // Create threads
  unique_lock lck(race_mutex);
  race_started = true;

  cv.notify_all();
  /* ... */
}
```

A real mistake



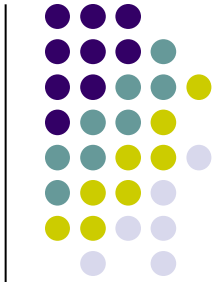
- Why is this wrong?
- Since the thread function doesn’t release the lock until it’s done driving
- No other thread can drive until the current one is done
- The cars drive one after the other
- They don’t race against each other
- No concurrency!

A natural mistake



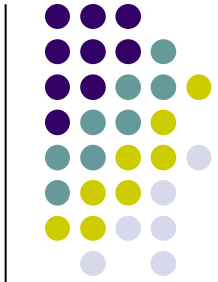
- Why did this error occur so often?
- The thread function never accesses `race_started` outside of the `wait` call
- So it is easy to forget that you need to release it outside of the `wait` call

A dangerous mistake



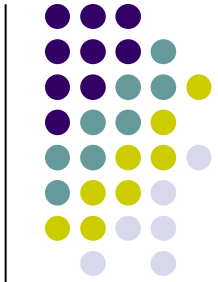
- While it is a serious mistake
- The program will likely behave functionally correct
- Just have unacceptable performance characteristics
- It may well not get caught during testing
- Manual unlocking feels very "un-C++-like"

What to do about it



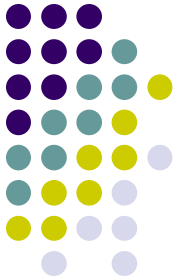
- “Be more careful”?
- While not necessarily bad advice, it’s better to follow best practices that don’t require extreme alertness

Using RAII



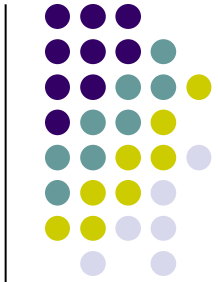
- As mentioned earlier, manual unlocking is very “un-C++-like”
- Also, it feels like it might be exception-unsafe
  - While that’s not a problem in this particular example, it’s easy to construct examples where it is
  - We gave examples of that when initially explaining RAII
- How can we modify the code to use RAII?

RAII version



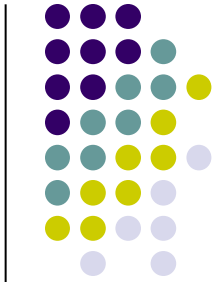
```
bool race_started{};
mutex race_mutex;
condition_variable cv;

// Racing thread function
void run_race(/* some args */)
{
  { // Scope ensure lock is released
    unique_lock lck(race_mutex);
    cv.wait(lck, [&] { return race_started; });
  }
  /* ... */      // Racing code
}

// Main thread
int main() {
  /* ... */      // Create threads
  {              // Scope helps here, too
    lock_guard lck(race_mutex);
    race_started = true;
  }
  cv.notify_all();
  /* ... */
}
```

This is a better solution



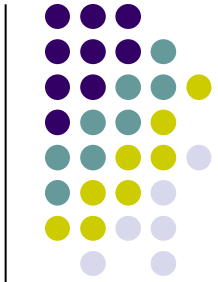
- Plenty of perfectly good code looks a lot like this
- However,…

There is still a subtle performance problem



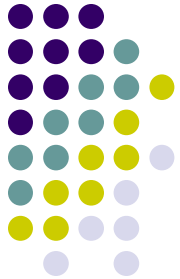
- Even though the threads are no longer serialized in their entirety
- The waking up process is
  - Only the thread that holds the mutex can run the test
- While the test is fast, switching threads is slow
- So this could still lead to thrashing
- Giving one car a head start ☹

Would like to fix with a `shared_mutex`



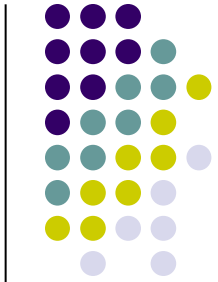
- Since the test only involves reading `race_started`
- There is no reason all of the racing threads can't run the test and wake concurrently
- This is exactly what reader-writer locks are for
- In C++, recall that a reader-writer lock is called a `shared_mutex`

Oops! A `condition_variable` only waits on a `unique_lock<mutex>`



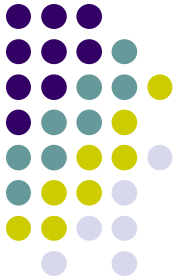
- For ultimate efficiency, the type of the lock is baked into `condition_variable`
- But in this case, it forces us to write inefficient code
- And won't work whenever we use any kind of mutex other than `std::mutex`

condition_variable_any



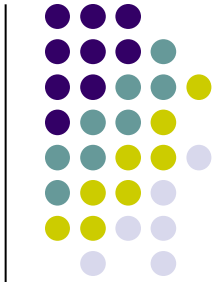
- condition_variable_any is just like condition_variable
- Only it can wait on anything with a lock() and unlock() method

shared_mutex version



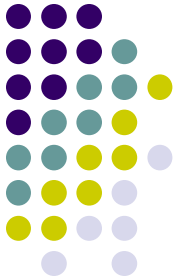
```
bool race_started{};
shared_mutex race_mutex;    // Changed to shared_mutex
condition_variable_any cv;  // Changed to condition_variable_any

// Racing thread function
void run_race(/* some args */)
{
  { // Scope ensure lock is released
    shared_lock lck(race_mutex);  // Now we can use a shared_lock ☺
    cv.wait(lck, [&] { return race_started; });
  }
  /* ... */      // Racing code
}

// Main thread
int main() {
  /* ... */      // Create threads
  {              // Scope helps here, too
    lock_guard lck(race_mutex);
    race_started = true;
  }
  cv.notify_all();
  /* ... */
}
```

What if we don’t release the lock



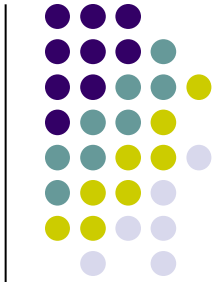
- Since one reader does not block others, the race would run just fine
- Whether you view this “protection” as a good thing or a bad thing is a matter of opinion :/

shared_mutex works in our race without releasing



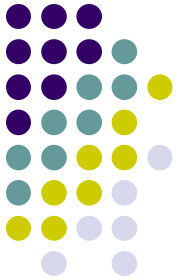
```
bool race_started{};
shared_mutex race_mutex;    // Changed to shared_mutex
condition_variable_any cv;  // Changed to condition_variable_any

// Racing thread function
void run_race(/* some args */)
{
    shared_lock lck(race_mutex);  // Now we can use a shared_lock ☺
    cv.wait(lck, [&] { return race_started; });
   /* ... */      // Racing code
}

// Main thread
int main() {
  /* ... */      // Create threads
  {              // Still need to release this one because it is a unique lock
    lock_guard lck(race_mutex);
    race_started = true;
  }
  cv.notify_all();
  /* ... */
}
```

Going lock-free



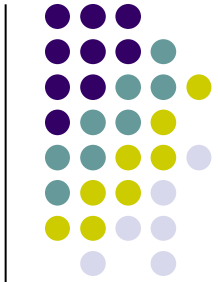
- Last quarter we discussed using atomics as an alternative to locks
- Can that help here?
- If we read and write the `race_started` flag atomically
- We don’t really need a lock on it
- Here it is with a dummy lock

“Dummy lock” version



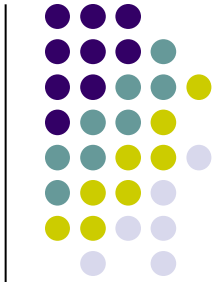
```
atomic<bool> race_started{}; // Atomic now
struct {
  void lock() {}    // Locking does nothing
  void unlock() {}  // Unlocking too
} dummy_lock;
condition_variable_any cv;

// Racing thread function
void run_race(/* some args */)
{
  cv.wait(dummy_lock, [&] { return race_started.load(); }); // Atomic load
  /* ... */      // Racing code
}

// Main thread
int main() {
  /* ... */      // Create threads
  race_started.store(true);
  cv.notify_all();
  /* ... */
}
```

That was much shorter. Is it better?



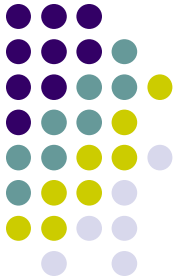
- Hmm, the `dummy_lock` is a bit of a hack
- Having a fake lock feels like it is asking for trouble…
- Also, it isn't even lock free because `condition_variable` uses a lock internally
  - Food for thought: What does this mean for our earlier `shared_lock` version? You can look at the `condition_variable` source code in the standard library to find out

In C++20, we can now do this fully atomically



- In C++20, atomic variables have `wait` and `notify_...` methods, just like condition variables
- Their behavior is similar but not identical
- Instead of a predicate, you give `wait()` the “old” value, and you wake up when the value becomes different
- Because there is no lock, you’re not protected against the value changing back quickly
  - And you will fail to wake because it will have the old value when checked
  - This is called the ABA problem
- In our case, we never reset `race_started`, so not a problem

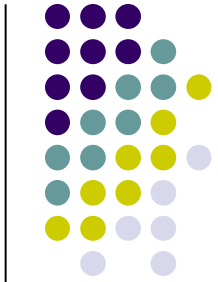
atomic<bool> version



```
atomic<bool> race_started{};

// Racing thread function
void run_race(/* some args */)
{
  race_started.wait(false); // Wait until no longer false
  /* ... */      // Racing code
}


// Main thread
int main() {
  /* ... */      // Create threads
  race_started.store(true);
  race_started.notify_all();
  /* ... */
}
```

Clear and concise

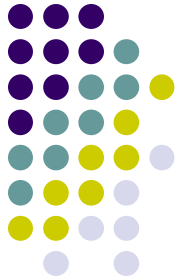
- This is starting to look good
- Are we done?



Is it lock free?

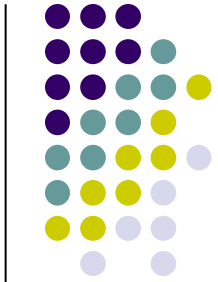
- Not necessarily!
- C++ has a rich atomic capability
- But it can’t do the impossible
- If the computer hardware isn’t capable of supporting a particular atomic operation directly
- The C++ standard library will simulate it with locks
- Since this is implementation-defined, you can check at runtime with `race_started.is_lock_free()`
- On most hardware, `atomic<bool>` is lock free, but in principle at least, it might be run on weaker hardware

`atomic_flag`



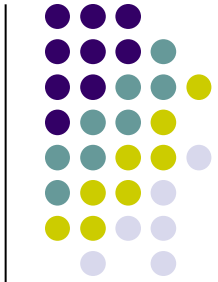
- C++ has one atomic type that is always required to be lock free
- It is a simple flag bit
  - Pretty much any hardware can handle
- Similar (but not identical) interface and behavior to `atomic<bool>`
- Let’s take a look

atomic_flag version



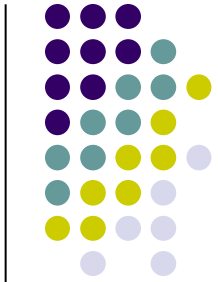
```
atomic_flag race_started; // Use atomic_flag now

// Racing thread function
void run_race(/* some args */)
{
  race_started.wait(false);
  /* ... */      // Racing code
}


// Main thread
int main() {
  /* ... */      // Create threads
  race_started.test_and_set(); // Only change to our use
  race_started.notify_all();
  /* ... */
}
```

Are we there yet?



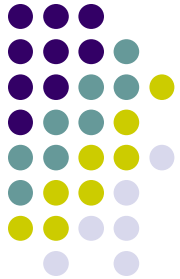
- Yes, we are there ☺
- This version is lock free
- Small gotcha: Prior to C++20, `atomic_flag` may be initialized with garbage
  - Learn about and use the (now deprecated) `ATOMIC_FLAG_INIT` in that case
  - Of course, prior to C++20, atomics didn't have `wait` or `notify_...` methods, so we couldn’t use it anyway

Conclusion



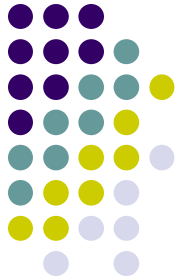
- Using a variety of techniques, we were able to repeatedly make our code
  - Clearer, Shorter, Faster, Safer
- If these techniques show up even in such a simple example
- They may be useful for your code as well
- Since a number of these weren't added until C++20, it can also be thought of as an example of how architects improve abstractions over time

HW 10-1



- We are implementing a rock-paper scissors game
  - https://en.wikipedia.org/wiki/Rock_paper_scissors
- The director thread signals the start of a round
  - “Rock-Paper-Scissors Go!”
- The two player threads produce answers as simultaneously as possible
- The director thread updates a score
- Ten rounds total and then reports the overall results

HW 10-2: Extra Credit



- What is something you would like to see added to C++?
- It could be something big like reflection or small like a new integer type
- Make a good case for why it would be a good feature and explain what it would look like