C++

March 28, 2022

Mike Spertus

spertus@uchicago.edu

COURSE INFO

Who am I?

- Senior Principal Engineer in AWS Dev Tools
  - Design tools for developing applications in the cloud: Both C++ and other languages
- Only started at Amazon a couple of years ago, so maybe my previous job is more relevant
  - Fellow and Chief Scientist for Cyber Security Services at Symantec, where my job was to help turn Symantec into a security as a service company
  - Chief Architect for their Authoritative Data Lake, which contains trillions of lines of security telemetry across dozens of products
    - You can hear learnings about that in MPCS53013 Big Data Application Architecture
- A long-time member of the ANSI/ISO C++ committee
  - Written over 50 C++ standards proposals, including C++11 non-static data member initializers, and C++17' Constructor Template Argument Deduction
  - Consider joining the standards committee. It's the best way to learn C++ ☺
- Helped write one of the first commercial C compilers for the original IBM PC AT

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
  - Me: 3-5PM Mondays JCL398F or by arrangement
    - Come to office hours in person or on zoom
  - Our TA, Shobhit: 10-12AM Thursdays JCL356
- 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 Monday 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 Thursday evening, you will receive a grade and comments back by noon on Saturday, 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

INTROS

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
- Internally, `condition_variable_any` uses the *type-erasure* technique that we learned about last quarter

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 happens if we forget to unlock or use RAII

- Since one reader does not block others, the race would run just fine
- Still, it is safer and more “correct” to release the lock when you no longer need it
- Whether you view this “protection” as a good thing or a bad thing is a matter of opinion :/

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

REGULAR EXPRESSIONS

What are regular expressions?

- What computer scientists means by regular expressions is much different from programmer mean by regular expressions
- Very important to keep in mind
- Nevertheless, even though we are programmers, it is helpful to first understand regular expressions from a computer science point of view
  - The theory slides below freely quote from Grune & Jacobs, Parsing Techniques: A Practical Guide
- Don’t worry, we’ll get practical soon enough
  - Although the theory can provide context and deeper understanding, you should be able to apply the practical sections even if you didn’t understand the theoretical ones

THEORY OF REGULAR EXPRESSIONS

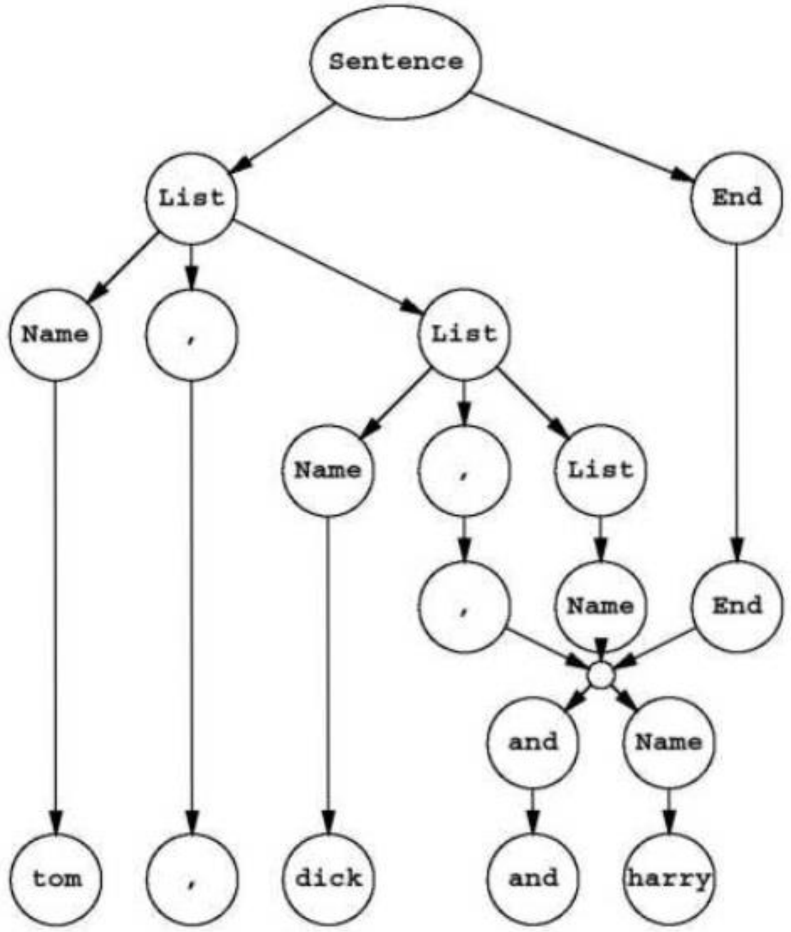
Phrase structure grammars

- The theory of formal languages is based on the idea of a Phrase Structure grammar (PS grammar)
- A PS grammar is just a list of translation rules
  - Name → tom | dick | harry
  - Sentence$_s$ → Name | List End
  - List → Name | Name , List
  - , Name End → and Name
- PS grammars were first developed in India over 2000 years ago!
  - Bhate and Kak, Pāṇini's grammar and computer science. *Annals of the Bhandarkar Oriental Research Institute*, 72:79–92, 1993.
    - In the Astādhyāyī, the Sanskrit scholar Pāṇini (probably c. 520–460 BC) gives a complete account of the Sanskrit morphology in 3,959 phrase structure rules.

Matching tom, dick and harry

Intermediate form	Rule used
Sentence	Always start with Sentence
List End	$\text{Sentence}_s \rightarrow$ List End
Name , List End	List $\rightarrow$ Name , List
Name , Name , List End	List $\rightarrow$ Name , List
Name , Name , Name End	List $\rightarrow$ Name
Name , Name and Name	, Name End $\rightarrow$ and Name
tom, dick and harry	Name $\rightarrow$ tom | dick | harry (3 times)

Sometimes its easier to visualize parsing as a Directed Graph



The Chomsky hierarchy

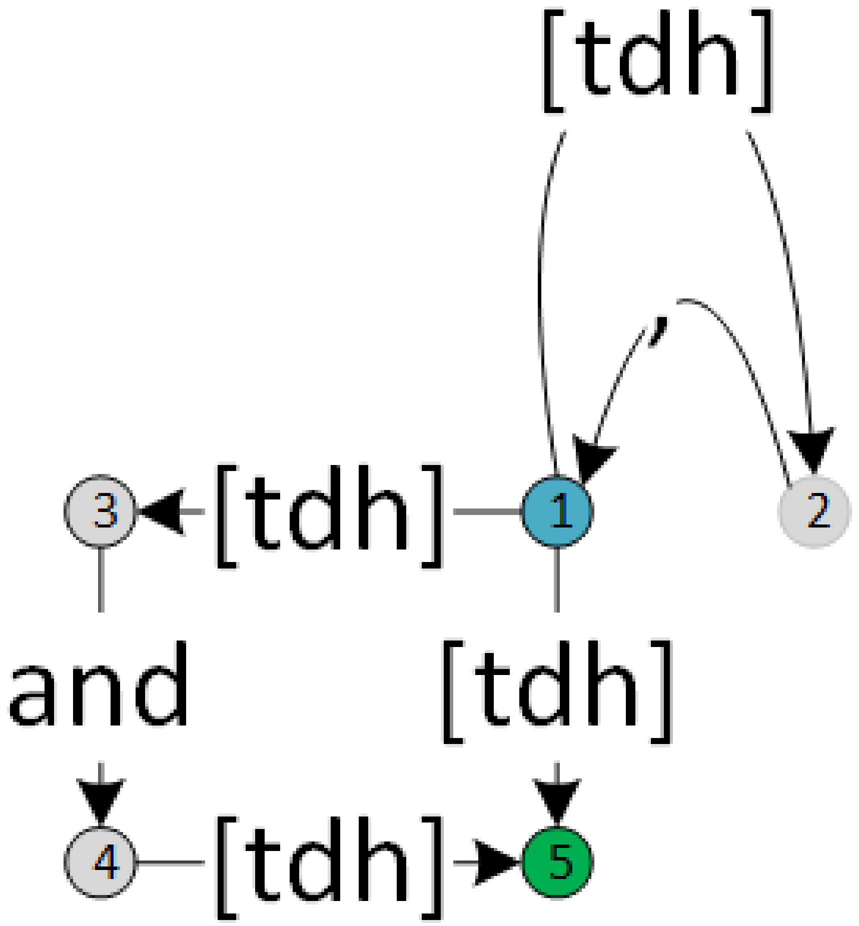
- Matching arbitrary PS grammars is very hard (and inefficient), so Noam Chomsky created a hierarchy of conditions to create more specialized grammars that are easier to match
  - Type 0: All PS grammars
  - Type 1 “Context Sensitive”
    - Only one symbol on the left-hand side gets replaced, while any others (the context) are unaffected. In the following example, only Comma is replaced
    - Name **Comma** Name End → Name **and** Name End
  - Type 2 “Context Free”
    - Rules have only one symbol on the left (i.e., no context)
    - List → Name , List | Name
  - Type 3 “Regular”
    - The only place a rule can have a symbol (“non-terminal”) on the right side is in the first position. All other positions are fixed text (“terminals”)
    - List → ListHead & tom
    - Regular languages are the easiest to match but still useful

Regular expressions

- Regular languages are so simple that you can express them with a single expression, called a Regular Expression.
  - *: “zero or more”
  - +: “one or more”
  - ?: “optional
- Here is the names example as a regular expression
  - (((tom|dick|harry),)*(tom|dick|harry) and)? (tom|dick|harry)
  - Parentheses are just for grouping. They are not part of the text
    - They are not capturing (no such thing as capture groups)
  - | means “or”
  - More simply:
    - (([tdh],)*[tdh] and)? [tdh]
    - [xyz] is a “character class” shorthand for x|y|z meaning any of x or y or z.
    - Common character classes often have standard abbreviations like \s for whitespace characters

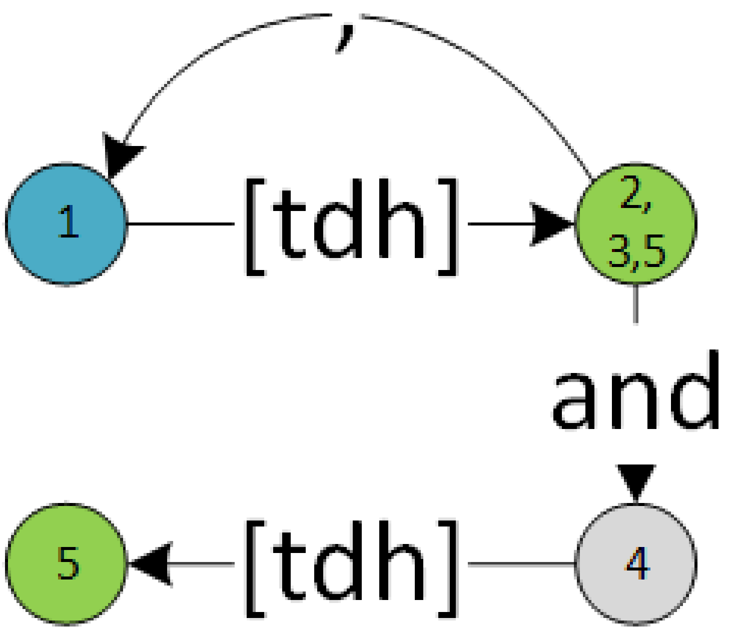
Regular Expressions and Finite State Automata

- Regular expressions are languages that can be recognized by machines with a finite amount of memory
  - In other words, machines that must be in one of a finite set of states
- Start at blue
- Done at green
- This machine is a "nondeterministic finite automaton" (NFA) because we don't know whether to go from 1 to 2, 3, or 5 when we get a "t" (or "d" or "h").
- We have to "backtrack" and try them all



Deterministic finite automata (DFA)

- For maximum efficiency, we would like to avoid backtracking to try out each option one at a time
- The idea is to run all the options in parallel*
  - In the NFA, we went to state 2 after reading a "t" and then later backtracking and trying states 3 and 5 if our first guess doesn't work out
  - By contrast, a DFA just says we are in the state "2 or 3 or 5" when we get the t, so we can keep track of all possibilities at once
- While a DFA can be much more efficient in time, it is much less efficient in space since it generally has more states



*Don't confuse this with multithreading, the parallelism is in the algorithm

Limitations of regular expressions

- Regular expressions are easy to write and efficient to run, but the tradeoff is that the language that a regular expression can match is very restricted
  - For example, we can’t write a regular expression for “a bunch of a’s followed by the same number of b’s” (e.g., aaaabbbb)
  - There are infinite possibilities for how many a’s were seen but regular expressions only have finite state
  - This is usually described as “regular expressions can’t count”

Context-free languages

- Context-free languages are the next level in the Chomsky hierarchy
- They are the languages that can be matched by a finite state automaton and a stack
- The stack lets it build a tree by keeping track of where we are in the top levels while building the bottom levels
  - CF matchers are usually called “parsers” because they build parse trees, which regular expressions are called “lexers”
- We can recognize the “a^nb^n” language with the context-free rule
  $S_s \rightarrow ab \mid aSb$
- However, a CF language cannot recognize “$a^nb^nc^n$”
  - Context-free languages cannot count two things at a time
- A CF language cannot recognize “the same thing repeated twice”
  - NithinNithin
- A good tool for generating CF-parsers in C++ is Boost Spirit
  - ANTLR, and Bison++ are also good

REGULAR EXPRESSIONS IN PRACTICE

Real-life regular expression engines have a lot more than *, +, and ?

- Capture groups: (f.*r)b*
  - Captures foobarbar in foobarbarbaz
- Back references: ^(.*)\1$
  - Matches the same thing repeated twice
  - This shows that regular expression tools can recognize some languages that aren't even context free
- Lazy matching: (f.*?r)b*
  - Captures foobar in foobarbarbaz
- Lookbehind: (?<=foo)\w+
  - Matches bar in foobar
  - There is also lookahead
- A fantastic book on all the features of practical regular expression engines is
  - Jeffrey Friedl. *Mastering Regular Expressions, 3rd Edition*. O'Reilly, 2006.

How do regular expression libraries work?

- To support these additional features, regular expression libraries adapt the NFA approach
  - They step through the text while following along with the regex, and they backtrack to another alternative when something doesn’t match
- If your regular expression is truly “regular,” you can use a DFA-based library like Google re2
  - Just +, *, and ?. No capture groups, lookahead/lookbehind, lazy quantifiers, etc.
  - re2 supports some of those “fancy” features, but it generally falls back to NFA’s for those

Regex performance

- Main point is to minimize backtracking
  - http://www.regular-expressions.info/catastrophic.html gives the example of (x+x+)+y which backtracks thousands of times on xxxxxxxxxx
  - Try to avoid multiple +'s and *'s that can be combined in different combinations.
  - In this case, rewrite as xx+y
- Use RegexBuddy to detect backtracking
  - Make sure you look at things that don't match as well as those that do
- Some rules from http://www.javaworld.com/article/2077757/core-java/optimizing-regular-expressions-in-java.html
  - Limit alternation
    - (abcd|abef) is slower than ab(cd|ef) since it doesn't need to check for abef if the string doesn't begin with ab
  - Use non-capturing parentheses (?:) if you don't need the capture
  - Experiment with both greedy and lazy matching. Sometimes one is a lot better than the other

Regular expressions in C++

- C++ supports regular expressions
- Just like string is really a typedef for basic_string<char>, regex is a typedef for basic_regex<char>, so different character types can be handled.

regex_match

-
  ```
  string text("How now, brown cow");
  cout << std::boolalpha;
  regex ow("ow");
  regex Hstarw("H.*?w");
  // False
  cout << regex_match(text, ow);
  // True
  cout << regex_match(text, Hstarw);
  ```

regex_search

- Can just check for matching substring

```
// True
cout << regex_search(text, ow);
```

- Or can find all matches

```
cmatch results;
regex_search(text, results, Hstarw);
// Prints "How"
copy
  (results.begin(),
   results.end(),
   ostream_iterator<string>(cout, "\n"));
```

Capture groups (helpful in HW)

- Find the input text matched by a specific part of your regex
- Any parentheses in your regex create a capture group
- This example is from http://softwareramblings.com/2008/07/regular-expressions-in-c.html,

-
  ```
  string seq = "foo@helloworld.com";
  regex rgx("(.*)@(.*)");
  smatch result;
  regex_search(seq, result, rgx);
  for(size_t i=0; i<result.size(); ++i) {
      cout << result[i] << endl;
  }
  ```
- `result[0]` is always the entire match: `foo@helloworld.com`
- `result[1]` is the first capture group: `foo`
- `result[2]` is the second capture group: `helloworld.com`
- If you want to just use parentheses for grouping in your regex but don't need to capture what they matched, use "non-capturing parentheses: `(?:x|y)` will match x or y but not save the result in a capture group

Repeat counts Helpful in HW

- `*` matches any number of occurrences and `+` matches at least one occurrence, but what if you want a specific number of occurrences?
- You can also include “repeat counts” in regexes
- `x{7}` will match 7 `x`’s in a row
- `x{3,5}` will match at least 3 but at most 5 `x`’s in a row

Some common character classes (Useful in HW)

- `\d` matches any digit
  - `"\\d{4}"` is a regex matching four digits
    - I needed to double the backslash to “escape” it because backslash is an escape character in C++ string literals
- `\s` matches any whitespace
- `\w` matches any alphanumeric (“word”) character

String searching

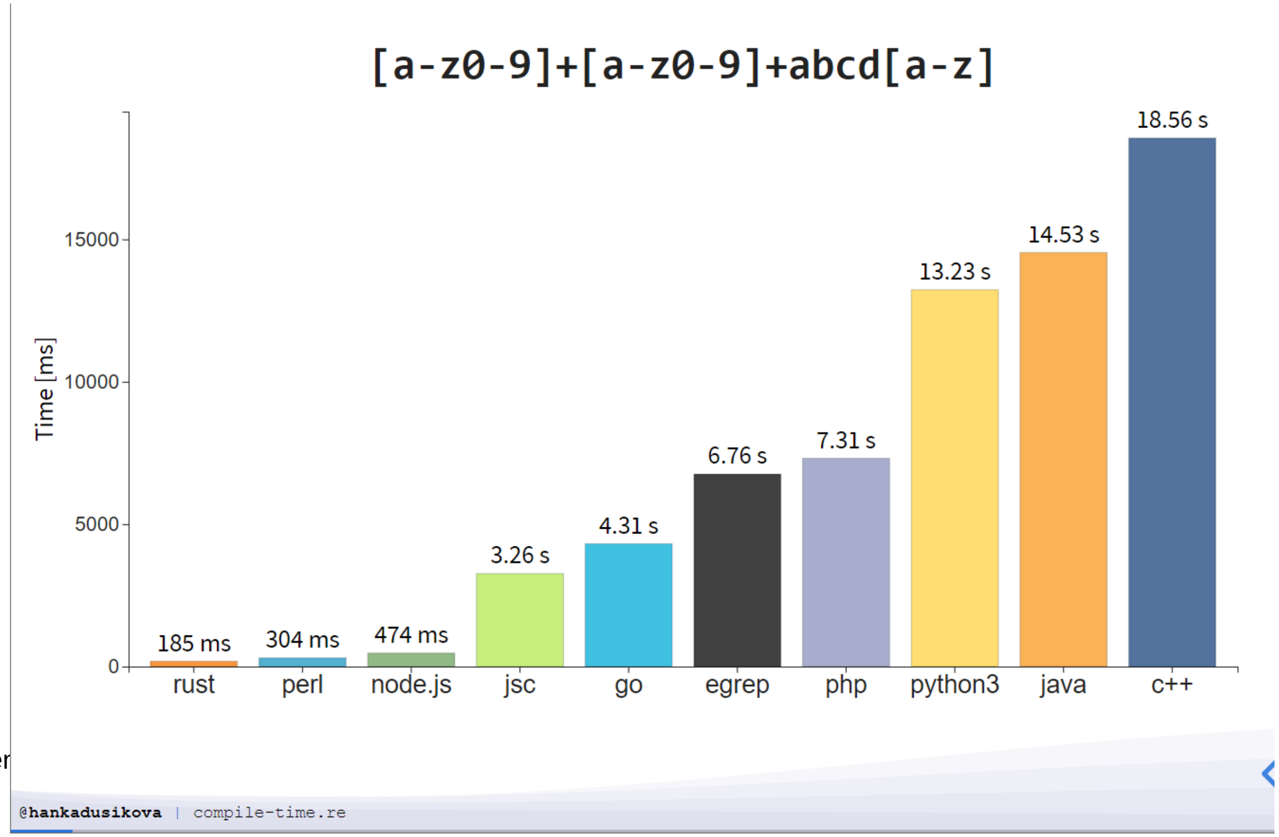
- What if you want to just search for a substring and not a pattern
- C++17 provides
  - std::default_searcher
    - More or less the same as std::search
  - std::boyer_moore_searcher
    - Uses Boyer-Moore
  - std::boyer_moore_horspool_searcher
    - A refined version of Boyer-Moore

The following slides adapted from Hana Dusikova's CTRE slides
https://compile-time.re/cppcon2019/slides/#

CTRE
COMPILE-TIME REG EXPR

C++ Regular expressions are really slow

- C++ regular expressions are powerful and ubiquitous but they are really slow



This is really bad

- Programmers use regular expressions constantly
- One of C++' main selling points is high performance
- But C++ regular expressions' performance is at the back of the pack
- 100x slower than Rust

Why are C++ regular expressions so slow?

- First, C++ locales classify every character as it is read
  - While potentially valuable for internationalization (I18N), it adds a slow processing step for each character
  - (In fact, C++ locales have not even proven very useful for I18N, so this is really a lose-lose
- The translation of regular expressions to NFA's takes place at runtime
  - The translation is slow, esp. if the regex is only run a few times
  - But even more importantly, the NFA's cannot be compiled to machine code since they don't exist until after the regex constructor runs
  - So the NFA are interpreted instead
  - Which is much slower than compiled code

Idea: `constexpr` and templates

- What if we used `constexpr` and templates to build the NFAs at compile-time?
- Then the compiler could translate them into optimized assembler

REVIEW OF CONSTEXPR

constexpr allows you to do things at compile-time

- Normally we think of programming as a way to write code that runs at runtime
- It is surprisingly common that you need "something" to take place at compile-time instead of runtime
- What do I mean by "something"?
- We will look at several examples over the next few slides

Template arguments need to be known at compile time

- The following code doesn’t compile

  -
    ```
    int n;
    cout << "How big a matrix? ";
    cin >> n;
    Matrix<n, n> m; // Ill-formed!
    ```
- The point: Since we don’t know `n` at compile-time, the compiler can’t compile the class Matrix<n, n>

Does this compile?

- Consider the following variation
- auto square(int x) { return x*x; }
  Matrix<square(3), square(3)> m;

Perhaps, surprisingly, it does not?

- While we can see from looking at the code that square(3) is going to be 9
- The compiler can't make that assumption
- Imagine what would go wrong if it did
  - Whether the code is legal would depend on whether the optimizer ran square at compile time or run-time
    - Which would just be weird
  - Also, if someone changed the body of square, it might no longer be computable at compile-time
- Therefore, the compiler has to reject square(3) as a template argument just like it did n in the previous example

Doing things at compile-time can also help performance

- Not only do you need compile-time values for template arguments
- But they benefit performance as well
- Why compute the following every time the program is run?

```
double pi = 4 * atan(1);
```

Constant expressions

- A *constant expression* is an expression that can be evaluated at compile-time
- Built-in values like `3`, `2*7` and `false` are constant expressions
- But since templates and performance programming are so important in C++, we would like to be able to create our own constant expressions
- That is what `constexpr` does
- Let's look at some examples

`constexpr` variables

- We’ve already see constexpr variables

  int constexpr seven = 7; // immutable

- This means that wherever the compiler sees seven, it can substitute the constant expression 7

`constexpr` functions

- Consider
  `double const pi = 4*atan(1);`
- Wouldn’t it be nice if that could be calculated at compile-time?
- To say that a function is computable at compile-time, label it as constexpr
- Example: Calculate greatest common divisor by Euclidean Algorithm

  -
    ```
    int constexpr gcd(int a, int b) {
      return b == 0 ? a : gcd(b, a%b);
    }
    ```

When do `constexpr` functions run?

- If the arguments are known at compile-time, the function may be run either at compile-time or at run time at the compiler's discretion. E.g., gcd(34, 55)
- If the value is needed at compile-time, it is calculated by the compiler. E.g., as a non-type template parameter: Matrix<gcd(34, 55)>
- If its arguments are not know at compile-time, it won't be run until run-time

  -
    ```
    void f(int i) {
      return gcd(34, i); // i is unknown
    }
    ```

`constexpr` functions

- A constexpr function cannot contain just any code.
  - For example, if it contained a `thread_local` variable, what would that mean at compile-time?
- What is not allowed:
  - Uninitialized declarations
  - `static` or `thread_local` declarations
  - Modification of objects that were not created in the function
    - Or potential modifications by, say, calling a non-constexpr function.
  - virtual methods
  - Non-literal return types or parameters
    - A type is literal if its constructor is trivial or constexpr

Fixing the square example

- Now our code runs
- auto square(int x) constexpr
  { return x*x; }

  Matrix<square(3), square(3)> m;

What types can I use at compile-time?

- We’ve only discussed using built-in types for constexpr programming
- Still useful
  - auto constexpr pi = 4*atan(1);
- However, much less powerful than the runtime language

Compile-time classes

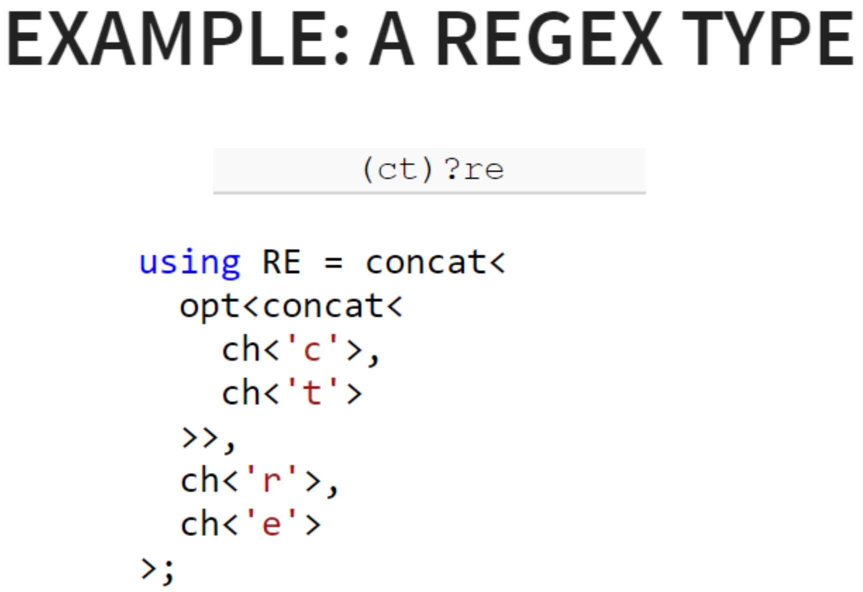
- A class can be created at compile-time if its constructor and destructors are constexpr
  - The default destructor is considered constexpr if all non-trivial member and base class constructors are as well
- C++20 makes vector, string, etc. usable at compile-time, which will be awesome. However
  - Most compilers have not yet implemented this big change
  - There are some restrictions

The CTRE project

- Hana Dusikova has done this
- Compiler-time regular expressions
  - http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1433r0.pdf
  - Great performance, errors checked at compile-time
  - Very positively received, so good candidate for C++23
  - No need to wait until then
    - https://github.com/hanickadot/compile-time-regular-expressions

Builds (a lot) on the kinds of `constexpr` and metaprogramming techniques we have seen

- A regular expression is a class template whose template arguments represent the regex
- Now the compiler can build a method for running the NFA



EXAMPLE: A REGEX TYPE

```
(ct)?re
```

```
using RE = concat<
  opt<concat<
    ch<'c'>,
    ch<'t'>
  >>,
  ch<'r'>,
  ch<'e'>
>;
```

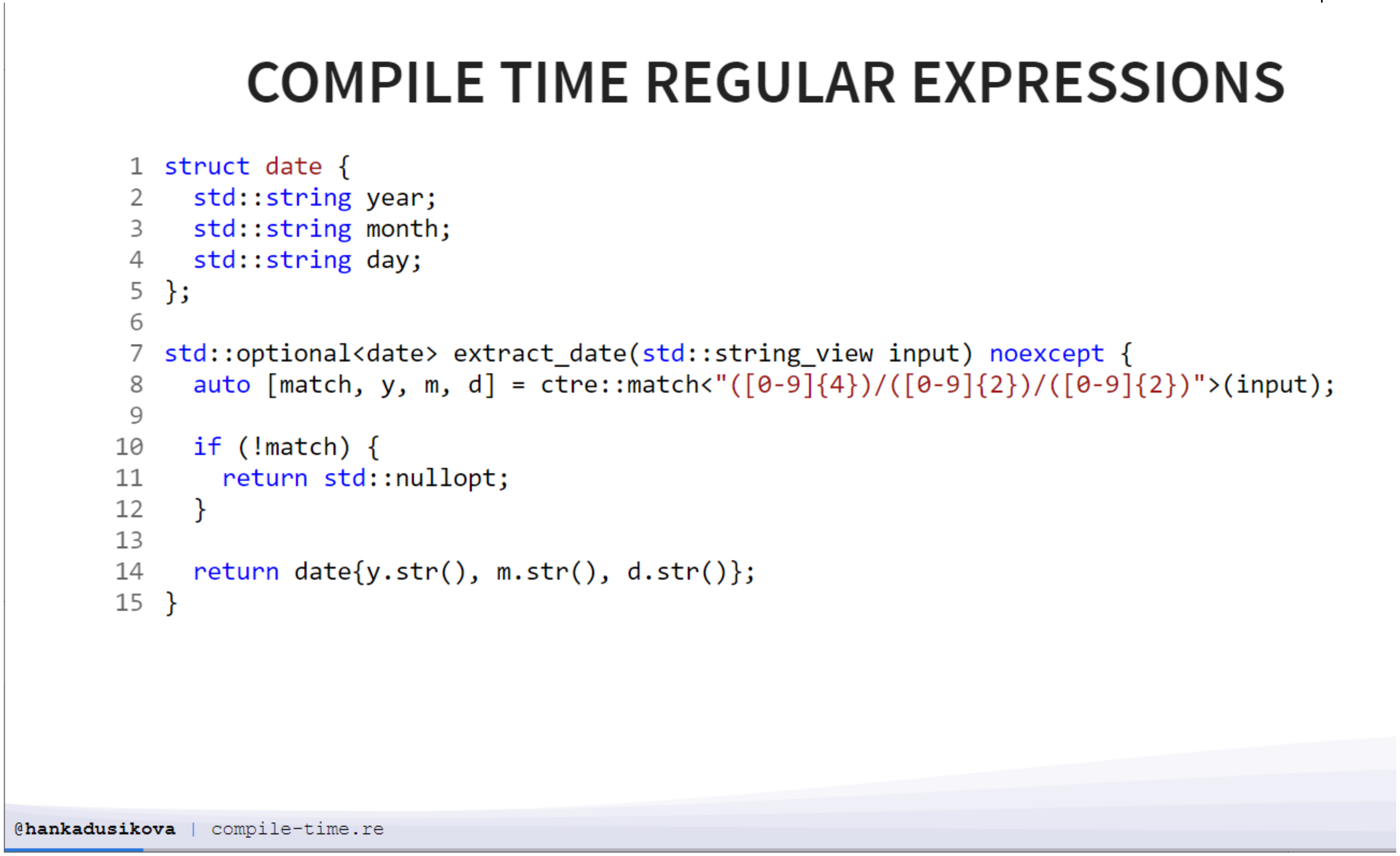
Finite automatons are now `constexpr` instantiation of class templates

```cpp
static constexpr auto empty = finite_automaton<0,0>{};

static constexpr auto epsilon = finite_automaton<0,1>{{}, {0}};

template <char32_t C> static constexpr auto one_char = finite_automaton<1,1>{
  {transition(0, 1, C)},
  {1}
};
```

Easy to use



COMPILE TIME REGULAR EXPRESSIONS

```cpp
struct date {
  std::string year;
  std::string month;
  std::string day;
};

std::optional<date> extract_date(std::string_view input) noexcept {
  auto [match, y, m, d] = ctre::match<"([0-9]{4})/([0-9]{2})/([0-9]{2})">(input);

  if (!match) {
    return std::nullopt;
  }

  return date{y.str(), m.str(), d.str()};
}
```

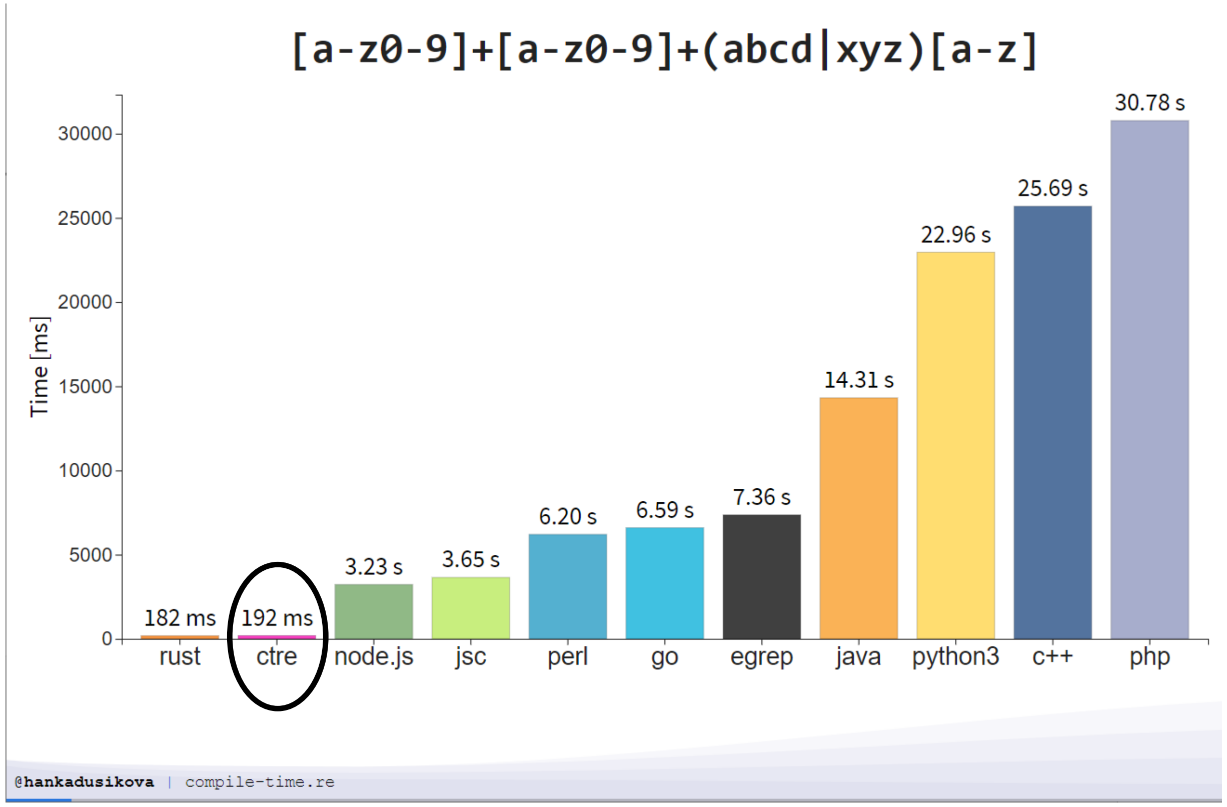
@hankadusikova | compile-time.re

Let's look at it on godbolt

- https://godbolt.org/z/e4Tdx8vPc
- Notes:
  - While CTRE is widely used, it is not part of the standard, so I selected it in the "libraries" section of Godbolt
  - Similarly, you will need to download CTRE to use it yourself
    - https://github.com/hanickadot/compile-time-regular-expressions

So is it faster?

- Is all the excitement about `constexpr` and metaprogramming really worth it?
- You tell me



Obviously, there is much more than that

- See https://compile-time.re for fantastic walkthroughs on how the parser, etc. is built at compile-time
  - Source code https://github.com/hanickadot/compile-time-regular-expressions
- If you want to really see what production-quality `constexpr` and metaprogramming look like, there's no better place

HW 10-1

- Use a regex to extract all of numbers with a decimal point from
  - `"Here are some numbers: 1.23, 4, 5.6, 7.89"`
- Use capture groups to generate the following output

```
1 is before the decimal and 23 is after the decimal
5 is before the decimal and 6 is after the decimal
7 is before the decimal and 89 is after the decimal
```

- For full credit, implement with both std::regex and CTRE
  - 5 points each

Exercise 10-2

- This problem, originally given by Stuart Kurtz, will expose you to some real-world I/O challenges. Use `getline`() on your input stream to process a line at a time.
- Use either std::regex or CTRE
  - For extra credit, try both and compare (see ex. 11-3)
- You will likely want to look up `ifstream` for doing file I/O if you aren't familiar with it already.

Exercise 10-2 (Continued)

- While the political debate over the role of anthropogenic forcing factors has on global warming rages, the hurricanes in the Atlantic rage on. The destructiveness of recent year's storms has been given as evidence for global warming. A serious problem in evaluating this is the extreme variability in the number and strength of hurricanes over time. The NOAA has a good (not perfect) data set that records hurricanes since 1851. What I want you to do is write a program that will attempt to analyze the strength of each hurricane season based on NOAA data. (Continued on next slide)

Exercise 10-2 (Continued)

- We will use the following easily computed aggregate measure
- The familiar scale of hurricane strength giving Hurricane category based on its windspeed is called the Saffir-Simpson scale
  - The windspeed range for each category is available on Wikipedia
- I want you to measure the aggregate storm activity in Saffir-Simpson days as follows
  - The NOAA data has four daily sustained wind speed readings for each storm, given in knots
  - Map each wind speed to the Saffir-Simpson scale
  - Divide each such entry by 4.0, and add it to the year's total (each entry is taken as a surrogate for 1/4 of a day's total activity)
- (Continued on next slide)

Exercise 10-2 (Continued)

- Historical data on hurricane strength is given in the Canvas file [hurdat_atlantic_1851-2011.txt]
  - An explanation of the format of the data can be found in the Hurdat format.pdf file on Canvas
  - This data set is pretty typical for scientific datasets accumulated over many years -- it is an ASCII version of a punch-card set.
- Your assignment is to write a program that computes and prints a table of the annual Saffir-Simpson day totals for each year from the above data
- You can use either `std::regex` or CTRE

HW 10-3

- In this lecture, we signaled a condition without using locks to start a race
- Let's consider when else would atomic signaling makes sense
- Last quarter, we used condition variables and a mutex to implement promises
  - Also on this week's canvas for convenience
- Would it be a good idea to similarly get rid of the mutex here? Why?
- If so, how would you go about doing it? (If not, you don't need to implement anything)
- Hint: Consider whether the First Rule of Optimization applies