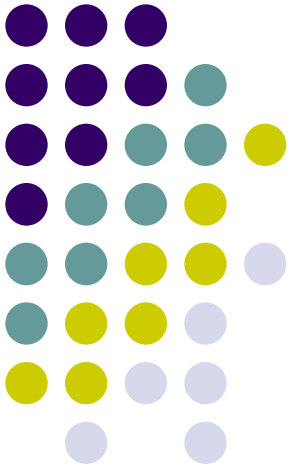
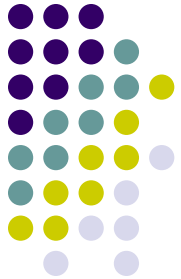
C++

March 25, 2025

Mike Spertus

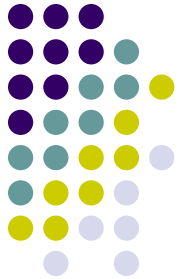
spertus@uchicago.edu





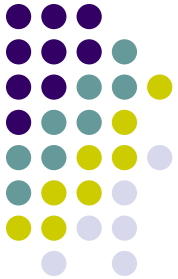
COURSE INFO

What is this course?



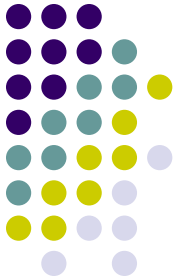
- Advanced C++
- As with last quarter, we really have two goals
  - Learn Advanced C++
  - Use C++ as "an excuse" to learn advanced programming techniques that apply to any languages

Additional resources



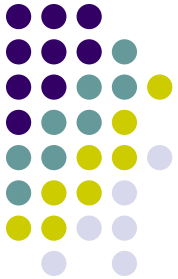
- Office hours
  - 10-11:30 Sundays on Zoom
  - 4-5 Tuesday in JCL298F and Zoom
- TA: Kaitlyn Zhang
  - Thursday 5-7 PM, JCL 207 and Zoom
- Ed discussions
  - Reach directly or from the Canvas tab

The most important rule



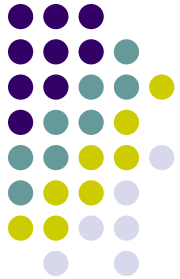
- If you are ever stuck or have questions or comments
- Be sure to contact me
  - spertus@uchicago.edu
  - Or on ed

Homework and Lecture Notes

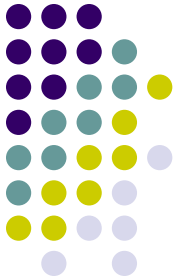


- Homework and lecture notes posted on Canvas
  - Choose MPCS 51045 and then go to “Pages”
- HW due on the following Tuesday before class
- Submit on Gradescope
  - Let me know if you can't access through the Canvas Gradescope tab
- Homework will be graded by the start of the following class
- ☹
  -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

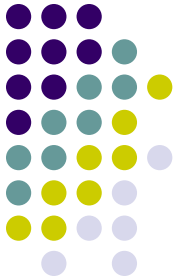
Course grade



- 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



TODAY'S TOPIC
TEXT CONTENT

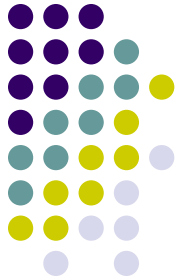


IOSTREAMS

IO Streams are also templates

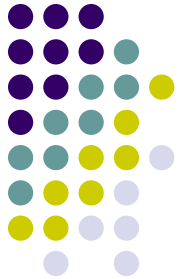
- `ostream` is for 8-bit characters
- That misses a lot of the world
- In reality, ostream is a typedef for basic_ostream, which is templatized by the character type
  - ```
    using ostream=basic_ostream<char>;
    // Use wostream for wide characters
    using wostream=basic_ostream<wchar_t>;
    ```
- `basic_ostream` has another template parameter giving more info about the character set, but this can usually be ignored because it has a good default
  - ```
    template<class CharT,
             class Traits=char_traits<charT>>
    class basic_ostream;
    ```

Sidebar: How wide is a wchar_t?



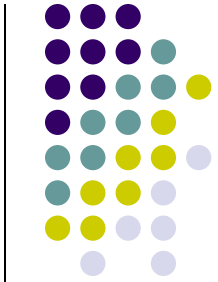
- 32 bits on Linux and 16 bits on Windows
- Ugh!
- It took a while to realize that 16 bits aren't enough
- But backwards-compatibility prevents fixing
- This means that a single wchar_t might not hold a character
- Better to avoid `wchar_t` and stick with `char`, `utf8_t`, or `utf32_t`
  - Even though Unicode support is still lacking…

I/O Manipulators



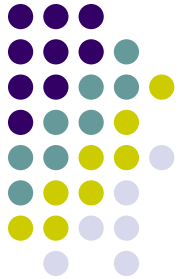
- Recall that `endl` not only inserts a `'\n'` in an `ostream`, it also flushes it.
- An object that actively manipulates a stream when inserted (or extracted) like `endl` is called an I/O manipulator.
- Some iomanipulators
  - `std::endl` of course
  - `std::hex` for hexadecimal i/o
  - `std::setw` to set a fixed width for the next insertion
  - `std::setfill` allows you to set the fill characters if the insertion is smaller than the width

Defining `endl`



- C++ makes it easy to create an I/O manipulator.
  - Simply define it as a function that takes and returns a stream
- If we only care about `ostream` (i.e., ordinary characters), it could be defined as follows
- ```
  ostream &
  endl(ostream &os)
  {
      os << '\n';
      os.flush();
      return os;
  }
  ```
  ```

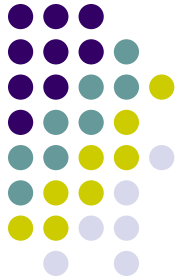
How endl gets called



- The standard library defines an overload of operator<< for inserting a function in an ostream that calls the function with the ostream as argument

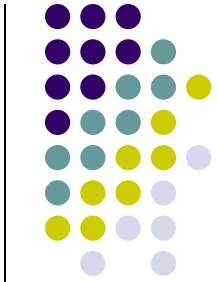
```
ostream &
operator<<
 (ostream&os,
  ostream&(*manip)(ostream &))
{
    return manip(os);
}
```

More advanced



- Really, `endl` should work for any output stream, regardless of character type.
- Recall from a few slides ago, that the stream class are really templates.
- Here's how `endl` is really defined.
- ```
  template<typename charT, typename Traits>
  basic_ostream<charT, Traits> &
  endl(basic_ostream<charT, Traits> &os)
  {
      os << '\n';
      os.flush();
      return os;
  }
  ```
  ```

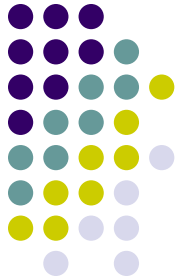
I/O manipulators with arguments



```
struct setw {
  setw(size_t s) : width(s) {}
  size_t width;
}

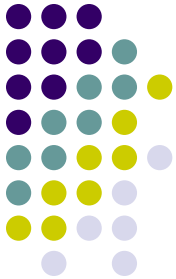
template<typename char_t, typename traits>
basic_ostream<char_t, traits> &
operator<<(basic_ostream<char_t, traits> os,
                    setw sw) { … }
```

Stream buffers



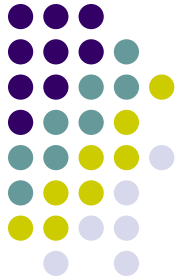
- Stream classes do formatted I/O (i.e., they let you insert and extract arbitrary types from a character stream).
- Once it is time to actually output characters to (or input from) a device, a stream buffer is used to perform the I/O.
- Every stream has a streambuf member that can be set using `basic_ios::rdbuf`. Recall that `basic_ios` is the base class for all streams.

Stream buffers



- Let's look at http://www.angelikalanger.com/IOStreams/Excerpt/excerpt.htm to go deeper into stream buffers

Customizing streams



- Basically, different types of streams are created by customizing a streambuf type
  - Stream types have no virtual functions
  - While stream buffers do
- However, we need to create a new stream type just to attach our stream buffer in the constructor. For example, the implementation of file streams is (logically)

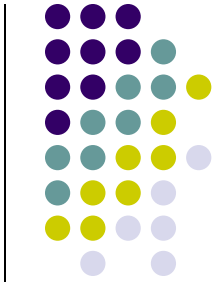
```
struct ofstream : public ostream {
  ofstream() : ostream(mybuf) {…}
  … // Other constructors and methods
private:
  filebuf mybuf;
};
```

Now with templates

- Of course, since ostream is really a typedef for basic_ostream<charT, traits>, the above code could really be:

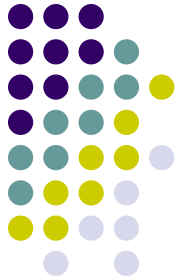
```
template
  <class charT,
   class traits=char_traits<charT> >
struct basic_ofstream
  : public basic_ostream<charT, traits> {
  basic_ofstream() : basic_ostream(mybuf) {…}
  // Other constructors and methods
private:
  filebuf mybuf;
};
```

Customizing stream buffers

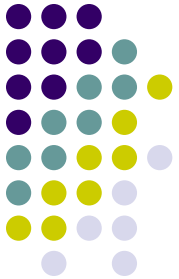


- A stream buffer can have an underlying memory buffer that can buffer characters
  - Set it with `setp(beg, end)` method with the beginning and end of the buffer
  - If you don't call it, you will have unbuffered I/O (`overflow` always called)
  - There is a similar method `setg` for input
- To customize a stream buffer, we override the overflow (for output) or underflow (for input) methods
  - These are called when the buffer (if it exists) cannot buffer any more characters
- Note: The slide in class talked about an `_Init` method, but that was Microsoft-specific. Use the standards-based approach her

Storing state in streams

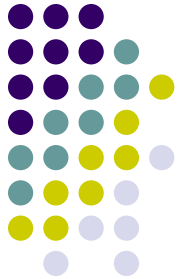


- Inserters/IO manipulators may need this
  - For example, `std::hex` has to store somewhere on a per-stream basis that the radix (base) of a stream is 16
  - Likewise, the inserters for numeric types will need to check the radix of the stream to print them
  - `std::ios_base::xalloc()` can allocate a slot in each stream for you to store your data
- New stream types may need this
  - E.g., fstreams need to store a file handle
  - stringstreams need to store a string
  - This can be done either with (1) xalloc, or (2) by adding a field to the stream, or (3) adding a field to the streambuf
    - Which do you think is best?



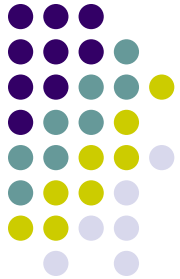
CUSTOMIZING STD::FORMAT

Stream formatting is (slowly) being replaced by `std::format`



- `std::format` improves on both C-style `printf` and traditional C++-style stream formatting
- This was the motivating example for variadics!
- Let's review

Variadics case study: Improved printing

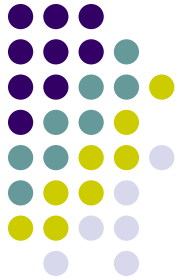


- Nearly every program does formatted output
- Yet the two built-in ways provided by C++, IOStreams and printf have serious problems
- We will review the problems and then show how to fix them with variadics

The problem with IOStreams

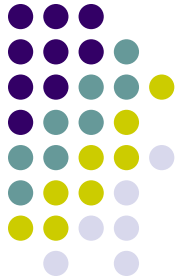
- C++ replaced “printf”-based I/O with I/O streams like you learned
- The problem is that IOStreams are very cumbersome for many simple tasks
- For example, we often see code like:
  - ```
    } catch(MyException &e) {
      ostringstream msg;
      msg << ”Operation ”
          << e.operation
          << ” failed with error code ”
          << e.errorCode;
      Logger.output(msg.str()); // Some logging framework
    }
    ```
  - In fact, what you more often see in this case, is a lame “Error detected” message to avoid the 6 lines above
  - We'd rather write
    ```
    printf(”Operation %s failed with error code %d”, e.operation, e.error_code);
    ```
    ```

The problems with `printf`



- Unsurprisingly, many people continue to use printf with C++
- However, `printf` is not extensible
  - It doesn't know how to print any new types you have created
- Even worse, it ignores argument types and prototypes and crashes and corrupts the stack at runtime

This code compiles without warning but crashes



- ```
  int main()
  {
    printf("%s\n", 1, 2);
  }
  ```
- The problem is that unlike nearly every other function, printf doesn't declare or check the type or number of arguments it was called with because that depends on the format string
- `int printf(char *fmt, ...);`
- The `...` could be anything
  - Don't confuse with C++ variadics
  ```

Solution: C++20 format

- We can use variadic templates to create a typesafe extensible formatting library that is easier to use than printf
- `cout << format("id {} - name {}", 5, "Mike");`
- Automatically uses the right formatter
  - So I don't need to specify the type in the format string like I did with `printf`

Reminder: Variadics

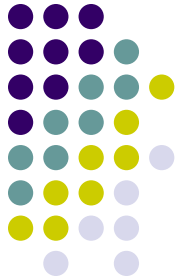
- A variadic template is a template that can take a variable number of parameters called a parameter pack
- Here's how to create a function that adds up all of its arguments

```
template<typename T>
T adder(T v) {
  return v;
}

template<typename T, typename... Args>
auto adder(T first, Args... args) {
  return first + adder(args...);
}
```

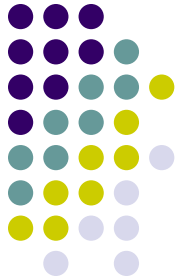
  - For example, `adder(1, 2, 3, 4)` will be `10`, and `adder("foo"s, "bar"s)` returns the string `"foobar"s`
    - Reminder: The `s` after the closing double-quote is a `string` literal
- We will learn much more about variadics later in the course
  - But this will do for now

Implement with variadics



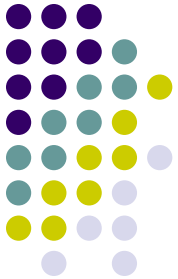
- Let's practice with variadics by implementing a simplified version of format ourselves
- https://godbolt.org/z/bnn7ss33s

First handle the base case



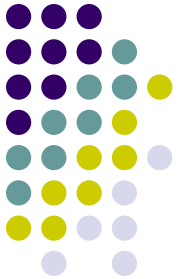
- A typical approach in variadics is to have a non-variadic "base case" overload and then recurse
- This case is easy for format
  - No arguments to format!
- ```
  string simple_format(string_view fmt) {
    return string(fmt);
  }
  ```
  ```

How to print an argument



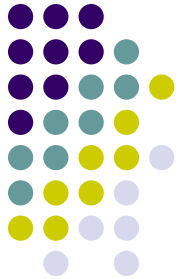
- We'll just use the stream inserter to print a type for the moment
- ```
  template<typename T>
  string make_string(T const &t) {
      ostringstream oss;
      oss << t;
      return oss.str();
  }
  ```
- IRL this would be to slow and limited, but let’s not get distracted from variadics
- Will come back to this in a few slides
  ```

Now recurse and we're done



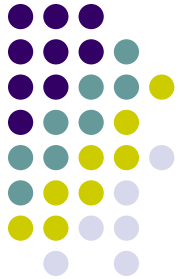
```
template<typename T, typename ...Ts>
string simple_format(string_view fmt, T const&t, Ts const &... ts)
{
    auto braces = fmt.find("{}"); // Omitting error handling :(
    return string(fmt.substr(0, braces)) + make_string(t)
           + simple_format(fmt.substr(braces + 2), ts...);
}
```

Do streams still matter if you use `std::format`?



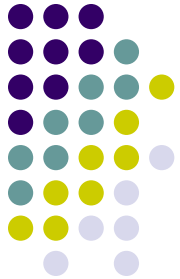
- Yes, they do
- Eventually, you usually write to `cout`, or an `ofstream`, etc.
- Even if you use `format`
- `cout << format("Bye, {}\n", "Mike");`
- Basically, we use streams for their I/O capability, but not for formatting

But how do we create custom formatters?



- We emphasized above that extensibility is preserved by `std::format`
- But how do we create formatters for our own types?
- With streams, one customized formatting for their own types by defining
  `ostream& operator<<(ostream &, myType);`
- `std::format` (somewhat controversially) won't recognize these

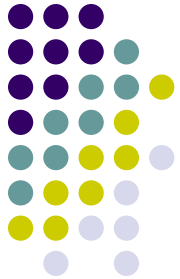
Formatters



- Instead, you define your own specialization of `std::formatter`
- An “easy” way is to inherit from an existing formatter
- `struct Temp { double deg{}; bool farenheit{true}};`

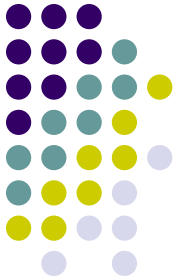
```
template<>
struct formatter<Temp> : formatter<string_view>
{
  template<typename FormatContext>
  auto format(const Temp &t, FormatContext& ctx)
  {
      return formatter<string_view>::format(
        std::format(”{} degrees {}”, t.deg,
                                     t.farenheit? ”F“: ”C”)), ctx);
  }
}
Temp t(25, false);
auto s1 = format(”{}”, t); // 25 degrees C
```

That wasn't very easy
Are there any benefits?



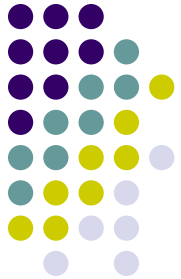
- First, it's a lot more efficient
- Well, you get all of the format specifiers that string_view has for free
- `auto s = std::format("{:~^10}", Temp{12});`
  `// prints ~~~12 F~~~~`

Let's do it without inheriting



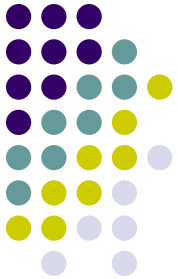
- Can we do it ourselves?
-
```
struct Temp { double deg{}; bool farenheit{true}};
template<>
struct formatter<Temp>
{
  template <typename FormatParseContext>
    auto parse(FormatParseContext& pc) { // What? We'll see
      return pc.begin();
  }
  template<typename FormatContext>
  auto format(const Temp &t, FormatContext& c)
  {
    auto&& out= ctx.out();
    format_to(out, "{} degrees ", t.deg);
    return format_to(out, t.farenheit ? "F" : "C");
  }
}
Temp t(25, false);
auto s1 = format("{}", t); // 25 degrees C
```

What is the purpose of parse?

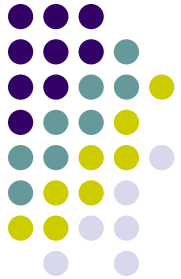


- Really, we should decide celsius vs farenheit when we print
- `cout << format("{C}", t); // Print as Celsius`

Code like this

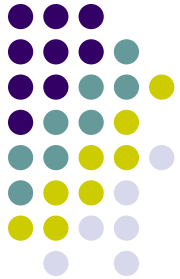


```
struct Temp { double deg{}; // Always store a celsius
template<>
struct formatter<Temp>
{
  template <typename FormatParseContext> // h/t https://www.cppstories.com/2022/custom-stdformat-cpp20/
  auto parse(FormatParseContext& pc) {
      auto pos = ctx.begin();
      while (pos != ctx.end() && *pos != '}') {
         if (*pos == 'F')
            isCelsius = false;
         ++pos;
       }
       return pos;  // expect `}` at this position, otherwise,
                    // it's error! exception!
  }
  template<typename FormatContext>
  auto format(const Temp &t, FormatContext& c)
  {
    auto&& out= ctx.out();
    format_to(out, "{} degrees ", calculateUnit(t.deg, isCelsius);
    return format_to(out, isCelsius ? "C" : "F");
  }
  bool isCelsius{true};
}
Temp t(25);
auto s1 = format("{C}", t); // 25 degrees C
```

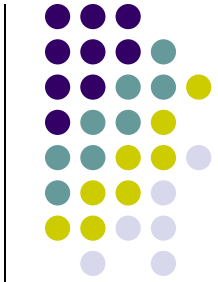


REGULAR EXPRESSIONS

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

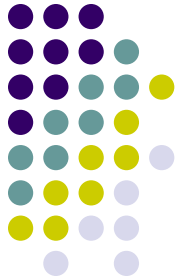


THEORY OF REGULAR EXPRESSIONS

Phrase structure grammars

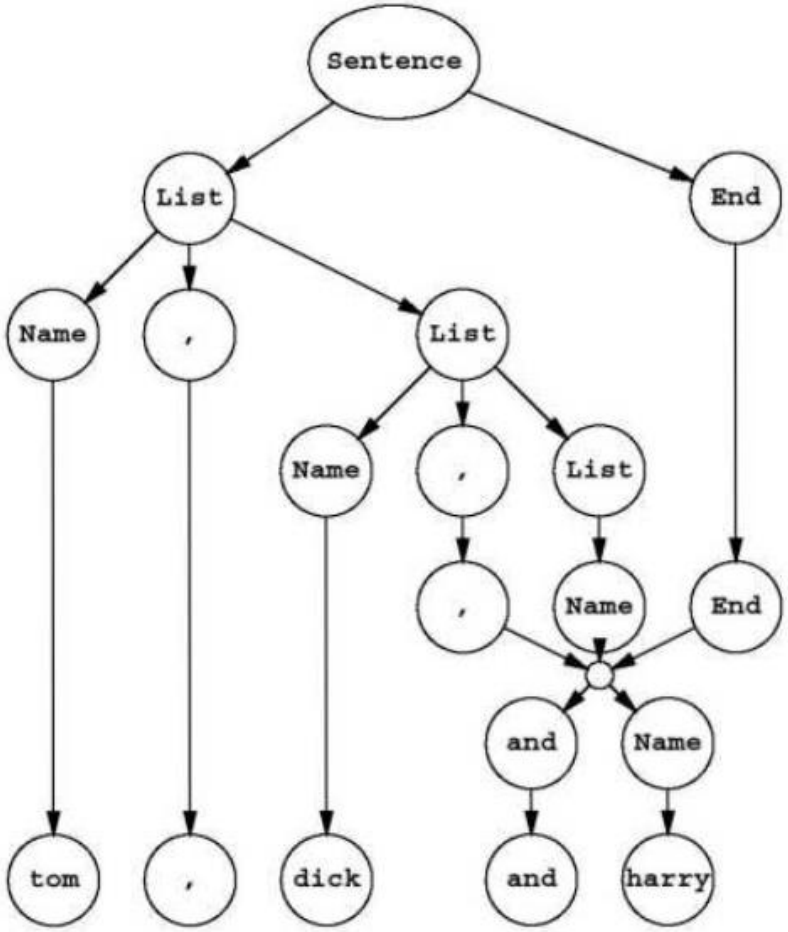
- The theory of formal languages is based on the idea of a Phrase Structure grammar (PS grammar)
- A PS grammar is just a list of translation rules
  - Name → tom | dick | harry
  - Sentence_s → Name | List End
  - List → Name | Name , List
  - , Name End → and Name
- PS grammars were first developed 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

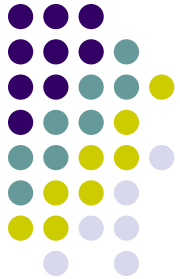


Intermediate form	Rule used
Sentence	Always start with Sentence
List End	$\text{Sentence}_s \rightarrow$ List End
Name , List End	List → Name , List
Name , Name , List End	List → Name , List
Name , Name , Name End	List → Name
Name , Name and Name	, Name End → and Name
tom, dick and harry	Name → tom |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

Sometimes its easier to visualize parsing as a Directed Graph

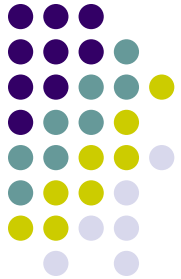


The Chomsky hierarchy



- 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

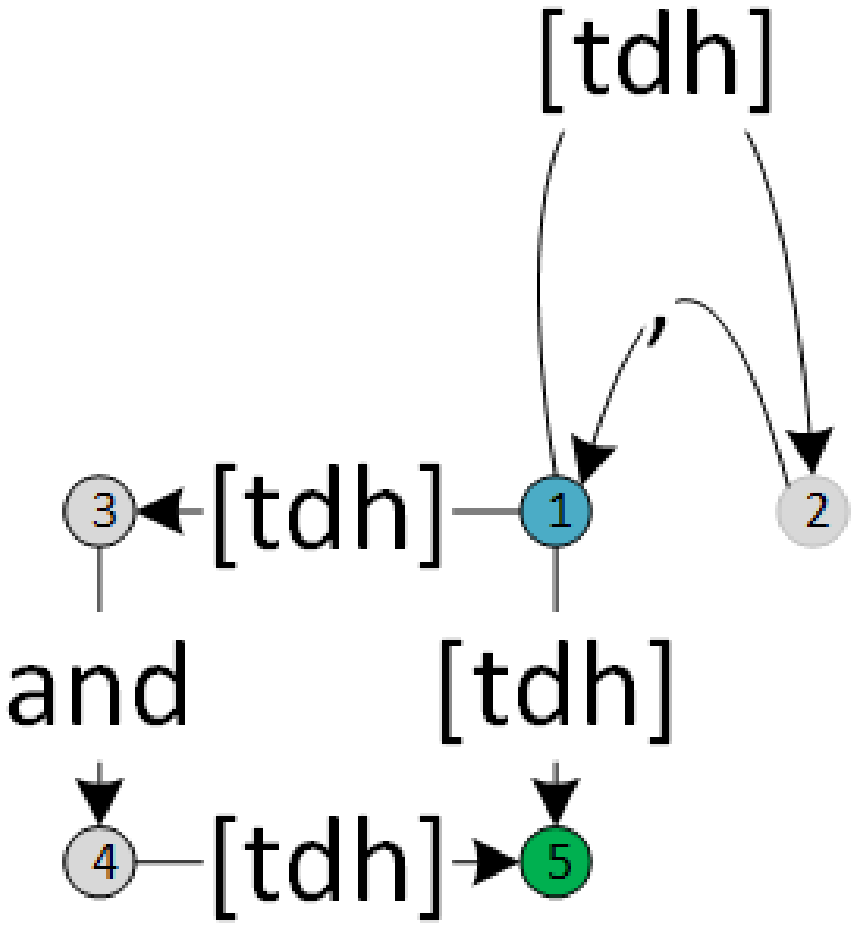
Regular expressions



- 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

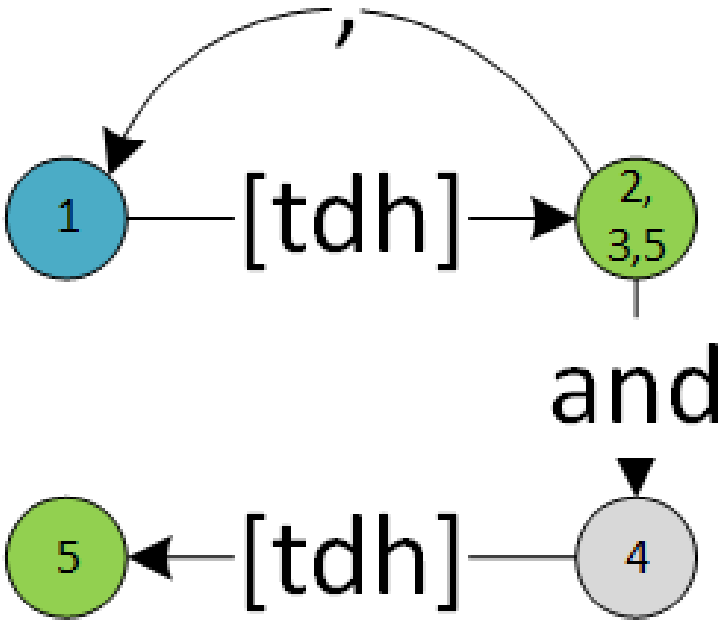
Regular Expressions and Finite State Automata

- 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



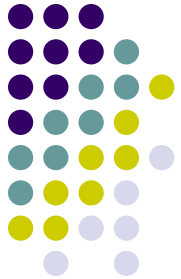
Deterministic finite automata (DFA)

- 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



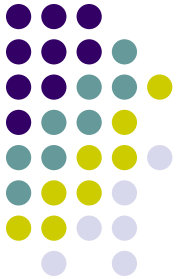
*Don’t confuse this with multithreading, the parallelism is in the algorithm

Limitations of regular expressions

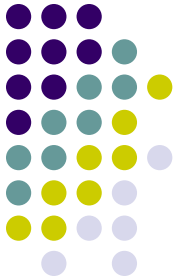


- Regular expressions are easy to write and efficient to run, but the tradeoff is that the language that a regular expression can match is very restricted
  - For example, we can't write a regular expression for "a bunch of a's followed by the same number of b's" (e.g., aaaabbbb)
  - There are infinite 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

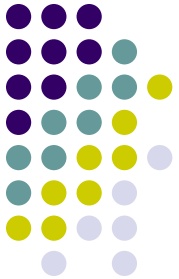


- Context-free languages are the next level in the Chomsky hierarchy
- They are the languages that can be matched by a finite state automaton and a stack
- The stack lets it build a tree by keeping track of where we are in the top levels while building the bottom levels
  - CF matchers are usually called “parsers” because they build parse trees, which regular expressions are called “lexers”
- We can recognize the “$a^n b^n$” language with the context-free rule
  $S_s \rightarrow ab \mid aSb$
- However, a CF language cannot recognize “$a^n b^n c^n$”
  - 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



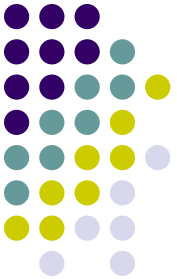
REGULAR EXPRESSIONS IN PRACTICE

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



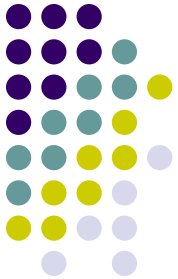
- Capture groups: (f.*r)b*
  - Captures foobarbar in foobarbarbaz
- Back references: ^(.*)\1$
  - Matches the same thing repeated twice
  - This shows that regular expression tools can recognize some languages that aren't even context free
- Lazy matching: (f.*?r)b*
  - Captures foobar in foobarbarbaz
- 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?



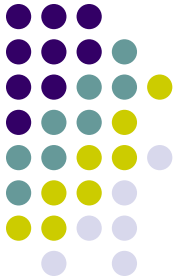
- To support these additional features, regular expression libraries adapt the NFA approach
  - They step through the text while following along with the regex, and they backtrack to another alternative when something doesn't match
- If your regular expression is truly "regular," you can use a DFA-based 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



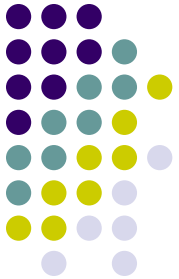
- Main point is to minimize backtracking
  - http://www.regular-expressions.info/catastrophic.html gives the example of (x+x+)+y which backtracks thousands of times on xxxxxxxxxx
  - Try to avoid multiple +'s and *'s that can be combined in different combinations.
  - In this case, rewrite as xx+y
- 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
- Look at the failure case ((x+x+)+y is efficient when it matches

Regular expressions in C++



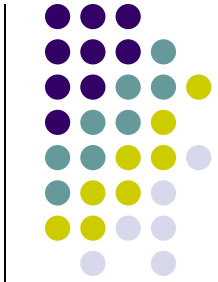
- C++ supports regular expressions
- Just like string is really a typedef for basic_string<char>, regex is a typedef for basic_regex<char>, so different character types can be handled.

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);
```

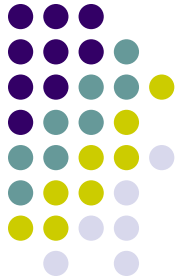
- Or can find all matches

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

Capture groups (helpful in HW)

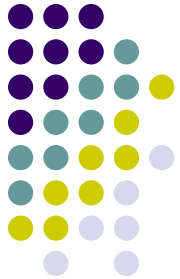
- Find the input text matched by a specific part of your regex
- Any parentheses in your regex create a capture group
- This example is from http://softwareramblings.com/2008/07/regular-expressions-in-c.html,
- ```
  string seq = "foo@helloworld.com";
  regex rgx("(.*)@(.*)");
  smatch result;
  regex_search(seq, 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



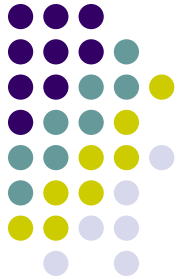
- `*` matches any number of occurrences and `+` matches at least one occurrence, but what if you want a specific number of occurrences?
- You can also include “repeat counts” in regexes
- `x{7}` will match 7 `x`’s in a row
- `x{3,5}` will match at least 3 but at most 5 `x`’s in a row

Some common character classes (Useful in HW)



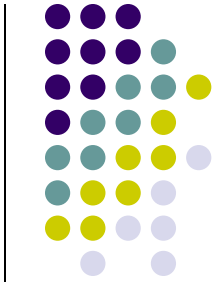
- `\d` matches any digit
  - `"\\d{4}"` is a regex matching four digits
    - I needed to double the backslash to “escape” it because backslash is an escape character in C++ string literals
- `\s` matches any whitespace
- `\w` matches any alphanumeric (“word”) character

Raw strings

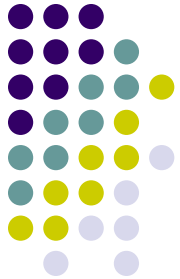


- That double backslash was ugly
- Can’t we create a string literal that is exactly what I say?
- That is what a raw string literal is
- Now I only need a single backslash
- R"(\d)"
- See https://en.cppreference.com/w/cpp/language/string_literal for examples and variations

String searching



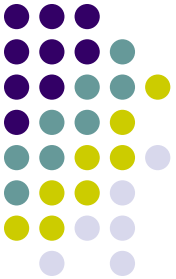
- What if you want to just search for a substring and not a pattern
- C++17 provides
  - `std::default_searcher`
    - More or less the same as std::search
  - `std::boyer_moore_searcher`
    - Uses Boyer-Moore
  - `std::boyer_moore_horspool_searcher`
    - A refined version of Boyer-Moore



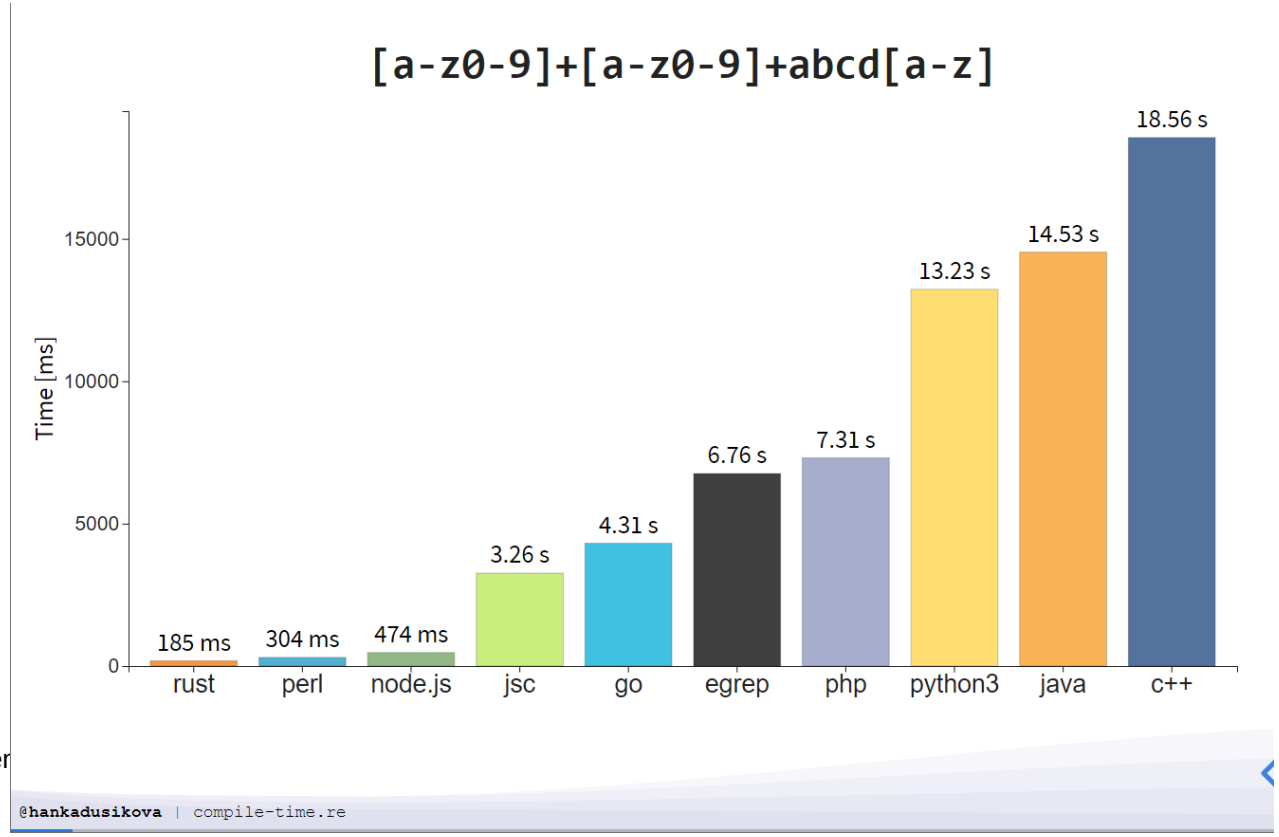
The following slides adapted from Hana Dusikova's CTRE slides
https://compile-time.re/cppcon2019/slides/#

CTRE
COMPILE-TIME REG EXPR

C++ Regular expressions are really slow

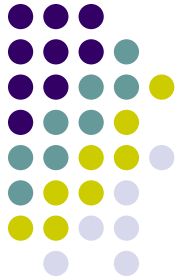


- C++ regular expressions are powerful and ubiquitous but they are really slow



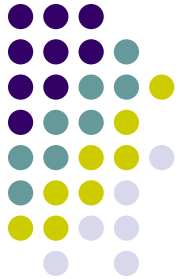
Preser

This is really bad



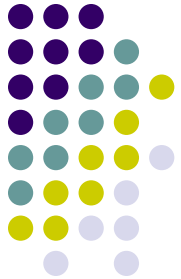
- Programmers use regular expressions constantly
- One of C++' main selling points is high performance
- But C++ regular expressions' performance is at the back of the pack
- 100x slower than Rust

Why are C++ regular expressions so slow?

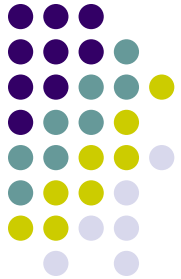


- First, C++ locales classify every character as it is read
  - While potentially valuable for internationalization (I18N), it adds a slow processing step for each character
  - (In fact, C++ locales have not even proven very useful for I18N, so this is really a lose-lose
- The translation of regular 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

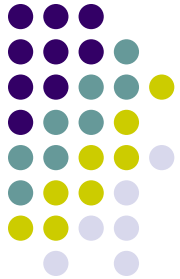


- What if we used `constexpr` and templates to build the NFAs at compile-time?
- Then the compiler could translate them into optimized assembler



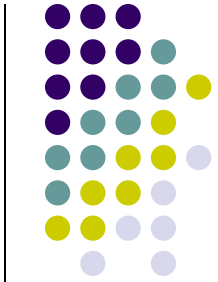
REVIEW OF CONSTEXPR

`constexpr` allows you to do things at compile-time



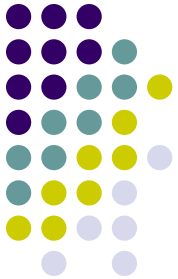
- Normally we think of programming as a way to write code that runs at runtime
- It is surprisingly common that you need "something" to take place at compile-time instead of runtime
- What do I mean by "something"?
- We will look at several examples over the next few slides

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?

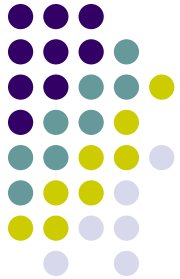


- Consider the following variation
- ```
  auto square(int x) { return x*x; }
  Matrix<square(3), square(3)> m;
  ```
  ```

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



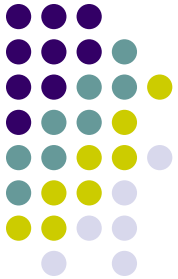
- Not only do you need compile-time values for template arguments
- But they benefit performance as well
- Why compute the following every time the program is run?

```
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

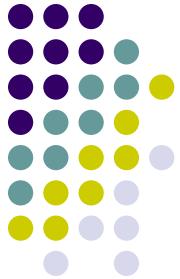


- We've already see constexpr variables

```
int constexpr seven = 7; // immutable
```

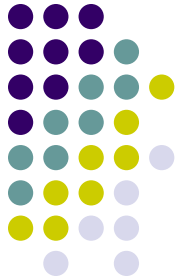
- This means that wherever the compiler sees `seven`, it can substitute the constant expression 7

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?

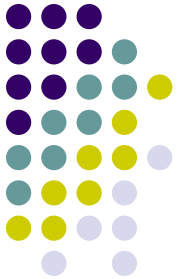


- If the arguments are known at compile-time, the function may be run either at compile-time or at run time at the compiler's discretion. E.g., gcd(34, 55)
- If the value is needed at compile-time, it is calculated by the compiler. E.g., as a non-type template parameter: Matrix<gcd(34, 55)>
- If its 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

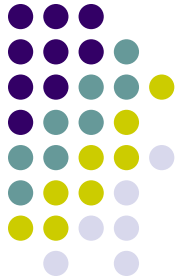
Fixing the square example



- 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

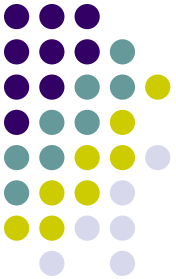
Compile-time classes

- 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, not likely to be standardized
  - 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



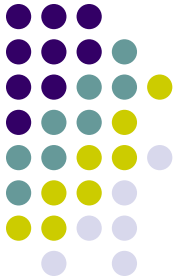
- A regular expression is a class template whose template arguments represent the regex
- Now the compiler can build a method for running the NFA

EXAMPLE: A REGEX TYPE

`(ct)?re`

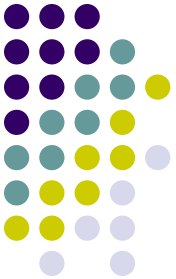
```
using RE = concat<
  opt<concat<
    ch<'c'>,
    ch<'t'>
  >>,
  ch<'r'>,
  ch<'e'>
>;
```

Finite automatons are now `constexpr` instantiation of class templates



```
1 static constexpr auto empty = finite_automaton<0,0>{};
2
3 static constexpr auto epsilon = finite_automaton<0,1>{{}, {0}};
4
5 template <char32_t C> static constexpr auto one_char = finite_automaton<1,1>{
6   {transition(0, 1, C)},
7   {1}
8 };
```

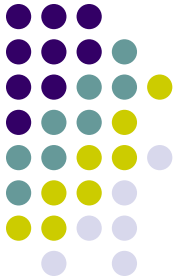
Easy to use



COMPILE TIME REGULAR EXPRESSIONS

```
1  struct date {
2    std::string year;
3    std::string month;
4    std::string day;
5  };
6
7  std::optional<date> extract_date(std::string_view input) noexcept {
8    auto [match, y, m, d] = ctre::match<"([0-9]{4})/([0-9]{2})/([0-9]{2})">(input);
9
10   if (!match) {
11     return std::nullopt;
12   }
13
14   return date{y.str(), m.str(), d.str()};
15 }
```

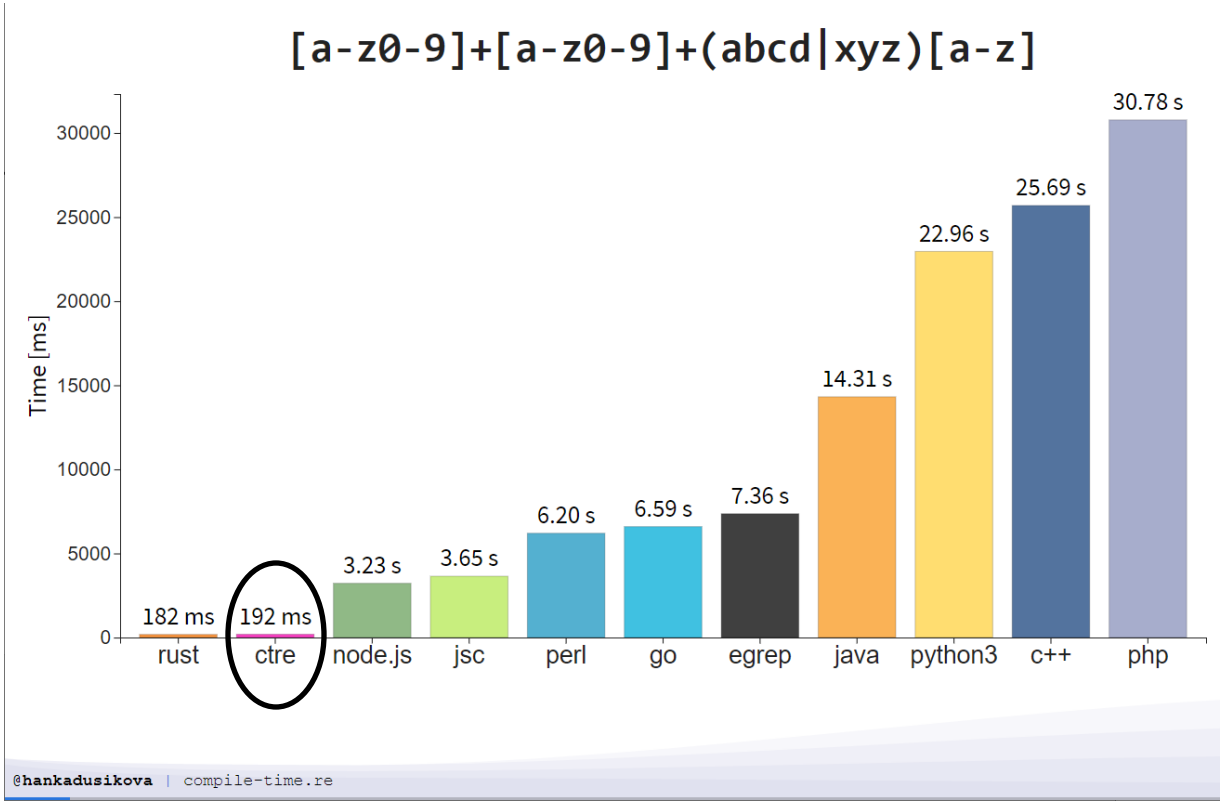
Let's look at it on godbolt



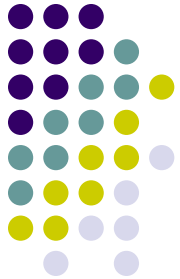
- https://godbolt.org/z/e4Tdx8vPc
- Notes:
  - While CTRE is widely used, it is not part of the standard, so I selected it in the "libraries" section of Godbolt
  - Similarly, you will need to download CTRE to use it yourself
    - https://github.com/hanickadot/compile-time-regular-expressions

So is it faster?

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

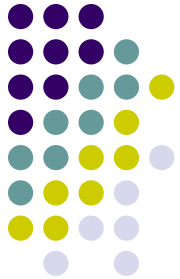


Obviously, there is much more than that



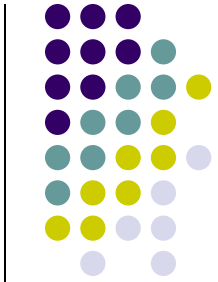
- See https://compile-time.re for fantastic walkthroughs on how the parser, etc. is built at compile-time
  - Source code https://github.com/hanickadot/compile-time-regular-expressions
- If you want to really see what production-quality `constexpr` and metaprogramming look like, there's no better place

HW 10-1



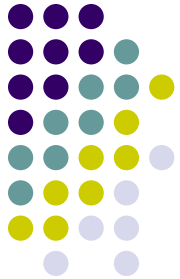
- Use a regex to extract all of numbers with a decimal point from
  - `"Here are some numbers: 1.23, 4, 5.6, 7.89"`
- Use capture groups to generate the following output
  - ```
    1 is before the decimal and 23 is after the decimal
    5 is before the decimal and 6 is after the decimal
    7 is before the decimal and 89 is after the decimal
    ```
- For full credit, implement with both `std::regex` and CTRE
  - 5 points each
    ```

Exercise 10-2



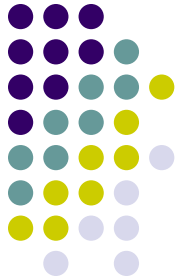
- This problem, originally given by Stuart Kurtz, will expose you to some real-world I/O challenges. Use `getline()` on your input stream to process a line at a time.
- Use either `std::regex` or CTRE
  - For extra credit, try both and compare (see ex. 11-3)
- You will likely want to look up `ifstream` 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)

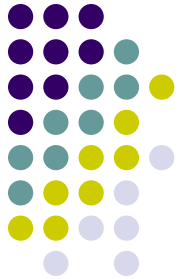


- We will use the following easily computed aggregate measure
- The familiar scale of hurricane strength giving Hurricane category based on its windspeed is called the Saffir-Simpson scale
  - The windspeed range for each category is available on Wikipedia
- I want you to measure the aggregate storm 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)

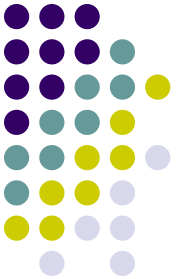
- Historical data on hurricane strength is given in the Canvas file hurdat_atlantic_1851-2011.txt
  - An explanation of the format of the data can be found in the Hurdat format.pdf file on Canvas
  - This data set is pretty typical for scientific datasets accumulated over many years -- it is an ASCII 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: Extra Credit



- Create a new stream `IndentStream` and I/O manipulators `indent` and `unindent` with the following properties.
  - Each line output to an `IndentStream` is indented by a given amount.
  - Inserting `indent` or `unindent` into a stream increases or decreases the indent level by 4.

HW 10-3: Extra Credit (Continued)



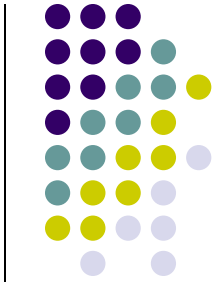
- For example,

```
IndentStream ins(cout);
ins << "int" << endl;
ins << fib(int n) {" << indent << endl;
ins << "if (n == 0) return 0;" << endl;
ins << "if (n == 1) return 1;" << endl;
ins << "return fib(n-2) + fib(n-1);" << unindent << endl;
ins << "}" << endl;
```

will output

```
int
fib(int n) {
    if (n == 0) return 0;
    if (n == 1) return 1;
    return fib(n-2) + fib(n-1);
}
```

HW 10-3: Extra Credit (continued)



- As described in class, almost all the work will be done by creating an `IndentStreamBuf` class and redefining its `overflow` method. Just about all `IndentStream` will do is set its stream buffer to an appropriate `IndentStreamBuf` in the constructor.
  - You will not want to have an actual memory buffer in your `IndentStreamBuf`, so your `overflow` method will get called on each character
- Useful link
  - http://www.angelikalanger.com/IOStreams/Excerpt/excerpt.htm