ctre

Jun 24, 2019

Contents

A compile-time (almost) PCRE-compatible regular expression matcher for C++.

CHAPTER 1

Overview

Fast compile-time regular expressions with support for matching/searching/capturing at compile-time or runtime.

```
ctre::match<"REGEX">(subject); // C++20
"REGEX"_ctre.match(subject); // C++17 + N3599 extension
```

1.1 API

class ctll::**fixed_string**

A compile-time fixed string.

Example:

```
static constexpr auto pattern = ctll::fixed_string{ "h.*" };

constexpr auto match(std::string_view sv) noexcept {
    return ctre::match<pattern>(sv);
}
```

template<class **Iterator**, class ...**Captures**>
class ctre::**regex_results**

using char_type = **typename** std::iterator_traits<*Iterator*>::value_type

The character type used by the `Iterator`.

template<size_t **Id**>
constexpr *captured_content*<*Id*, void>::storage<*Iterator*> **get**()

template<class **Name**>
constexpr *captured_content*<deduced, *Name*>::storage<*Iterator*> **get**()

template<ctll::*fixed_string* **Name**>

constexpr *captured_content*<deduced, *Name*>::storage<*Iterator*> **get**()
Returns the capture specified by `Id` or `Name`. ID `0` is the full match, ID `1` is the first capture group, ID `2` is the second, etc. Named groups are specified using `(?<name>)`.

Example:

```
if (auto m = ctre::match<"(?<chars>[a-z]+)([0-9]+)">("abc123")) {
  m.get<"chars">(); //abc
  m.get<2>(); //123
}
```

constexpr size_t **size**()
Returns the number of captures in this result object.

constexpr operator bool() **const noexcept**
Returns whether the match was successful.

constexpr operator std::basic_string_view<char_type>() **const noexcept**

constexpr std::basic_string_view<*char_type*> **to_view**() **const noexcept**

constexpr std::basic_string_view<*char_type*> **view**() **const noexcept**
Converts the match to a string view.

explicit constexpr operator std::basic_string<char_type>() **const noexcept**

constexpr std::basic_string<*char_type*> **to_string**() **const noexcept**

constexpr std::basic_string<*char_type*> **str**() **const noexcept**
Converts the match to a string view.

template<size_t **Id**, typename **Name** = void>
class captured_content

template<typename **Iterator**>
class storage

constexpr auto **begin**() **const noexcept**

constexpr auto **end**() **const noexcept**
Returns the begin or end iterator for the captured content.

constexpr operator bool() **const noexcept**
Returns whether the match was successful.

constexpr auto **size**() **const noexcept**
Returns the number of characters in the capture.

constexpr operator std::basic_string_view<char_type>() **const noexcept**

constexpr std::basic_string_view<char_type> **to_view**() **const noexcept**

constexpr std::basic_string_view<char_type> **view**() **const noexcept**
Converts the capture to a string view.

explicit constexpr operator std::basic_string<char_type>() **const noexcept**

constexpr std::basic_string<char_type> **to_string**() **const noexcept**

constexpr std::basic_string<char_type> **str**() **const noexcept**
Converts the capture to a string view.

static constexpr size_t **get_id**() **noexcept**
Returns `Id`

template<auto &**RE**, class ...**Args**>
constexpr ctre::*regex_results*<deduced> **match** (*Args*&&... *args*)

template<ctll::*fixed_string* **RE**, class ...**Args**>
constexpr ctre::*regex_results*<deduced> **match** (*Args*&&... *args*)

Matches `RE` against the whole input. `Args...` must be either a string-like object with `begin` and `end` member functions, or a pair of forward iterators.

template<auto &**RE**, class ...**Args**>
constexpr ctre::*regex_results*<deduced> **search** (*Args*&&... *args*)

template<ctll::*fixed_string* **RE**, class ...**Args**>
constexpr ctre::*regex_results*<deduced> **search** (*Args*&&... *args*)

Searches for a match somewhere within the input. `Args...` must be either a string-like object with `begin` and `end` member functions, or a pair of forward iterators.

1.2 Examples

1.2.1 Extracting a number from input

```
std::optional<std::string_view> extract_number(std::string_view s) noexcept {
      if (auto m = ctre::match<"[a-z]+([0-9]+)">(s)) {
        return m.get<1>().to_view();
    } else {
        return std::nullopt;
    }
}
```

link to compiler explorer

1.2.2 Extracting values from date

```
struct date { std::string_view year; std::string_view month; std::string_view day; };
std::optional<date> extract_date(std::string_view s) noexcept {
    using namespace ctre::literals;
    if (auto [whole, year, month, day] = ctre::match<"(\\d{4})/(\\d{1,2})/(\\d{1,2})">
↪(s); whole) {
        return date{year, month, day};
    } else {
        return std::nullopt;
    }
}

//static_assert(extract_date("2018/08/27"sv).has_value());
//static_assert((*extract_date("2018/08/27"sv)).year == "2018"sv);
//static_assert((*extract_date("2018/08/27"sv)).month == "08"sv);
//static_assert((*extract_date("2018/08/27"sv)).day == "27"sv);
```

link to compiler explorer

1.2.3 Lexer

```
enum class type {
    unknown, identifier, number
};

struct lex_item {
    type t;
    std::string_view c;
};

std::optional<lex_item> lexer(std::string_view v) noexcept {
    if (auto [m,id,num] = ctre::match<"([a-z]+)|([0-9]+)">(v); m) {
        if (id) {
            return lex_item{type::identifier, id};
        } else if (num) {
            return lex_item{type::number, num};
        }
    }
    return std::nullopt;
}
```

link to compiler explorer

1.2.4 Range over input

This support is preliminary and probably the API will be changed.

```
auto input = "123,456,768"sv;

for (auto match: ctre::range<"([0-9]+),?">(input)) {
      std::cout << std::string_view{match.get<0>()} << "\n";
}
```

1.3 Regex Syntax

The library supports most of the PCRE syntax with a few exceptions:

- atomic groups
- boundaries other than ^$
- callouts
- character properties
- comments
- conditional patterns
- control characters (\cX)
- horizontal / vertical character classes (\h\H\v\V)
- match point reset (\K)
- named characters

- octal numbers
- options / modes
- subroutines
- unicode grapheme cluster (\X)

TODO more detailed regex information

CHAPTER 2

Supported compilers

- clang 6.0+ (template UDL, C++17 syntax)
- xcode clang 10.0+ (template UDL, C++17 syntax)
- gcc 7.4+ (template UDL, C++17 syntax)
- gcc 9.0+ (C++17 & C++20 cNTTP syntax)
- MSVC 15.8.8+ (C++17 syntax only)

CHAPTER 3

Basic Usage

3.1 Template UDL syntax

Compiler must support N3599 extension, as GNU extension in gcc (not in GCC 9.1+) and clang.

```
constexpr auto match(std::string_view sv) noexcept {
      using namespace ctre::literals;
      return "h.*"_ctre.match(sv);
}
```

If you need N3599 extension in GCC 9.1+ you can't use -pedantic mode and define the macro `CTRE_ENABLE_LITERALS`.

3.2 C++17 syntax

You can provide pattern as a constexpr `ctll::fixed_string variable`.

```
static constexpr auto pattern = ctll::fixed_string{ "h.*" };

constexpr auto match(std::string_view sv) noexcept {
      return ctre::match<pattern>(sv);
}
```

(this is tested in MSVC 15.8.8)

3.3 C++20 syntax

Currently only compiler which supports cNTTP syntax `ctre::match<PATTERN>(subject)` is GCC 9+.

```
constexpr auto match(std::string_view sv) noexcept {
      return ctre::match<"h.*">(sv);
}
```

Index