P1433R0: Compile Time Regular Expressions

ISO/IEC JTC1 SC22/WG21 - Programming Languages - C++

Author: Hana Dusíková <hana.dusikova@avast.com>

Audience:
Unicode Study Group (SG16)
Library Evolution Working Group Incubator (LEWGI)
Intended Shipping Vehicle: C++23

TL;DR

Manipulating regular expressions (RE) in C++ is verbose and hard. This proposal discuss possibility how to solve it with a library solution which can check the pattern during the compilation and generate very efficient assembly. This library (CTRE, Compile Time Regular Expressions) is already implemented and very warmly welcomed.

Motivation

The current std::regex design and implementation are slow, mostly because the RE pattern is parsed and compiled at runtime. Users often don't need a runtime RE parser engine as the pattern is known during compilation in many common use cases. I think this breaks C++'s promise of *"don't pay for what you don't use."*

If the RE is known at compile time, the pattern should be checked during the compilation. The design of std::regex doesn't allow for this as the RE input is a runtime string and syntax errors are reported as exceptions.

Because of all these problems and limitations users instead turn to non-standard libraries.

Proposed API

In the CTRE library, the pattern is passed as a class non-type template parameter (P0732):

// match whole input and return captures
template <fixed_string Pattern> constexpr *impl-specified* **match**(auto && subject) noexcept;

// return iterable range where each elements is matched part of subject
template <fixed_string Pattern> constexpr *impl-specified* **multi_match**(auto && subject) noexcept;

// search for pattern match somewhere in subject (not bound to whole subject)
template <fixed_string Pattern> constexpr *impl-specified* **search**(auto && subject) noexcept;

// search and replace subject, can throw as it can allocate (currently not implemented in CTRE)
template <fixed_string Pattern, fixed_string Replacement> constexpr *impl-specified* **replace**(auto && subject) noexcept(false);

Return type

The return type is tuple-like implementation specific struct which is implicitly convertible to bool, appropriate std::basic_string_view (depending on char type of subject), and explicitly to std::string.

It also provides std::get support and thereby structured bindings.

```
If (auto [re, c1, c2] = match<"([0-9]+),([0-9]+)">(subject); re) {
        return {std::string_view(c1), std::string_view(c2)};
}
```

Using of Template Arguments

There are proposals for the language (Parametric Expressions, P1221) which enables the compile-time pattern to be passed as a function argument instead of a template argument. This would allow for a more natural syntax:

```
auto m = match("([0-9]+),([0-9]+)", subject);
```

Implementation & Usage Experience

This proposal is based on the CTRE library, which is implemented in both C++17 and C++20 with class non-type template parameters. It was presented at CppCon 2018.

The library generates very nice assembly and in most benchmarks is able to beat every other common RE library (more in Runtime Benchmark section).

Currently, the library supports PCRE derived syntax. However, we should use only Unicode or ECMAScript RE syntax with proper Unicode support, as both specifications are well standardized. C++11's std::regex is using multiple dialects: ECMA, Posix, awk, ...

Compile Time

For backtracking implementation, the compilation time of a RE pattern linearly depends on the length of a string (high-level complexity, not taking into account internal implementations of constexpr functionality in a compiler.)

For DFA based implementation the compilation time is exponential with RE complexity (non-determinism in the initial FA).

However, it should not be a problem, as most REs are short. Also for most usages (with only a few RE in a translation unit) compiling with CTRE library is actually quicker than with std::regex.

Runtime Benchmarks

Current CTRE implementation is using a backtracking matching so that the performance can be strongly influenced by poorly designed RE pattern. In typical scenarios, the performance is similar or better than existing high-performance RE implementations.

The following table shows grep-like benchmark scenario of a CSV file (with long lines and file length appr. 1.3GB). Tested pattern is: `([0-9]{4,16})?[aA]`

Table 1: GNU based system (gcc)

Implementation	**Time [sec]**	**Assembly generated [B]**
GNU egrep (baseline)	4.927	N/A
CTRE	3.172	19896
PCRE2	17.403	19872
std::regex (**libstdc++**)	33.585	213280
RE2	3.759	19704
boost::regex	22.692	183048

Table 2: MacOS (clang)

Implementation	**Time [sec]**	**Assembly generated [B]**
BSD egrep (baseline)	21.92	N/A
CTRE	11.11	29976
PCRE2	16.92	25580
std::regex (**libc++**)	1655 (28 minutes)	113284
RE2	10.35	25512
boost::regex	20.37	108900

Possible Approaches

- Introduce CTRE-like library functionality into the standard library.
- Make std::regex constexpr (P1149)
- Don't do anything (and continue recommending RE implementation outside of the standard if users want performance)

References

[CTRE] Hana Dusíková. Compile Time Regular Expressions library
https://github.com/hanickadot/compile-time-regular-expressions

[CTRE-talk] Hana Dusíková. Compile Time Regular Expressions library (CppCon 2018 talk)
https://www.youtube.com/watch?v=QM3W36COnE4

[P0732] Jeff Snyder, Louis Dionne: Class Types in Non-Type Template Parameters
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0732r2.pdf

[P1221] Jason Rice: Parametric Expressions
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p1221r0.html

[P1149] Antony Polukhin: Constexpr regex
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p1149r0.html

[UnicodeRe] Unicode® Technical Standard #18: Unicode Regular Expressions
http://unicode.org/reports/tr18/

[ecmaRE] ECMAScript® 2018 Language Specification
https://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf