

Introduction

It's fair to say that C++ is "Lego language" - it gives you basic tools to do almost everything, and requires craftsmanship and creativity in return to build astonishing things.

Sometimes it's helpful to have a ready-to-use toolset, to spare a valuable time and focus on the more crucial parts of development - like architectural decisions: working on the concepts, being able to combine the reusable (and exchangeable) functional blocks and accelerate development - to start with the proof-of-concept as soon as possible.

For that we have utility classes/methods, to give us the boost - yes, like Boost as third-party library which is used as an incubator for std library: for cooking all those concepts before they arrive into official std library.

This is small set of algorithms that you may use for manipulating with strings: especially if you plan to participate in one of those meaningless coding interviews with algorithms that you will never encounter in praxis.

Algorithms

Splitting the string

It's kind of frustrating that many languages out there have out-of-box available **split method**: for splitting the text into words, based on the given delimiter.

Before C++20 and ranges, the library based solutions was not that straight forward.

One commonly see solution with *istringstream* and iterators works fine - but only with blank space as delimiter

```
// use concepts for >= C++20
template <typename S>
static constexpr bool is_string = std::is_same_v<S, std::string> ||
                                   std::is_same_v<S, std::pmr::string> ||
                                   std::is_convertible_v<S, std::string> ||
                                   std::is_constructible_v<std::string, S>;

/**
 *@brief Counts the number of words delimited by blank space(s)
 */
template <typename S>
// requires is_string<S>
auto total_words_count(const S& in)
{
    std::istringstream ins;
    if constexpr (std::is_same_v<std::string_view, S>) ins = std::istringstream{in.data()};
    else ins = std::istringstream{in};

    return std::distance(std::istream_iterator<std::string>(ins),
                        std::istream_iterator<std::string>());
}

// Splitting with std::istringstream the entry text with predefined blank space as delimiter
template <typename Container, typename S>
// requires is_string<S>
auto split(const S& text) -> std::enable_if_t<is_string<S>, Container>
{
    const std::istringstream is {text};
    const auto wordsCount = total_words_count(text);

    Container container;
    if (0 == wordsCount)
    {
        container.push_back(text);
        return container;
    }

    std::istringstream is {text};
    container.reserve(wordsCount);
    container = Container{std::istream_iterator<S>(is), std::istream_iterator<S>()};

    return container;
}
```

More generic solution, that works with an arbitrary delimiter

```
// One can use std::pmr::vector<std::pmr::string> with preallocated memory
// on the stack for compile-time execution

template <typename Delimiter, typename Container = std::vector<std::string_view>>
constexpr auto split(std::string_view text, Delimiter& delimiter)
{
    Container words;

    for(;;)
    {
        const auto word = delimiter(text);
        if (word.size() == 0)
        {
            words.push_back(text);
            break;
        }
        words.push_back(word);
    }
    return words;
}

struct Delimiter
{
private:
    std::string_view delim_;
public:

    constexpr explicit Delimiter(std::string_view delim) noexcept: delim_(delim) {}

    constexpr std::string_view operator() (std::string_view& text) const
    {
        const auto pos = text.find(delim_, 0);
        if (pos == std::string::npos) return {};

        auto word = text.substr(0, pos);
        text = text.substr(pos + 1);

        return word;
    }
};
```

We can accomplish the same with regular expressions, which are notorious of being inefficient

```
std::size_t count_words_with_reg_expr(const std::string& in, std::string_view regex)
{
    std::regex re {regex.data()};

    const auto first = std::sregex_iterator(in.begin(), in.end(), re);
    const auto last = std::sregex_iterator();

    return std::distance(first, last);
}

// Split the words in a given text using regular expressions
template <typename Container = std::vector<std::string> >
[[nodiscard]] auto split_with_regex(Container& words, const std::string& in, std::string_view regex) ->
std::optional<Container>
{
    try
    {
        std::regex re {regex.data()};
        const auto first = std::sregex_iterator(in.begin(), in.end(), re);
        const auto last = std::sregex_iterator();

        for (auto it = first; it != last; ++it)
        {
            const std::smatch match {*it};
            words.emplace_back(match[0]);
        }

        return words;
    }
    catch (const std::regex_error& e)
    {
        std::cout << __func__ << ", Exception: " << e.what() << '\n';
        return std::nullopt;
    }
}
```

```

template <typename Container = std::vector<std::string>>
[[nodiscard]] auto split_with_regex(const std::string& in, std::string_view regex) -> std::optional<Container>
{
    // Check first the count of delimited words
    const auto count = count_words_with_regex(in, regex);
    if (count == 0) return std::nullopt;

    // Allocate space for desired container
    Container words;
    words.reserve(count);

    return split_with_regex(words, in, regex);
}

```

Fortunately, the C++20 introduced ranges, with views and adaptors - chainable operators, so that we finally have library support for this

```

namespace details
{
    template <typename StrType = std::string_view>
    constexpr auto split_view(StrType text, StrType delimiter)
    {
        using namespace ranges;
        return text | views::split(delimiter);
    }
}

```

@note ranges::to() is part of the v3 library

The entire code: <https://godbolt.org/z/jsd68YzTa>

More on ranges: https://github.com/damirli/modern_cpp_tutorials/blob/main/docs/C%2B%2B20/Ranges.pdf

Benchmarking: <https://quick-bench.com/q/G22VG6ZHPArPWAmOwybhBzgWlg>

Trimming the string

This is also useful algorithm that may find its practical implementation in text processing, removing some characters (usually blank spaces) either at the beginning,

```

struct is_blank
{
    bool operator()(unsigned char ch) const
    {
        return std::isblank(ch);
    }
};

template <typename S>
// requires is_string<S>
auto trim_left(const S& s) -> std::enable_if_t<is_string<S>, S>
{
    using std::cbegin;
    using std::cend;

    const auto firstNonBlankChar = std::find_if_not(cbegin(s), cend(s), is_blank{});
    return s.substr(std::distance(cbegin(s), firstNonBlankChar));
}

// Erase: modify input string

template<typename S>
// requires is_string<S>
auto trim_left_mod(S&& s) -> std::enable_if_t<is_string<S>, S>
{
    auto&& in = std::forward<S>(s);

    s.erase(s.begin(), std::find_if_not(s.cbegin(), s.cend(), is_blank{}));
    return s;
}

```

or at the trailing one at the end

```

template <typename S>
// requires is_string<S>
auto trim_right(const S& s) -> std::enable_if_t<is_string<S>, S>
{

```

```

    const auto lastNonBlankChar = std::find_if_not(s.cbegin(), s.crend(), is_blank{}).base();

    return s.substr(0, std::distance(s.cbegin(), lastNonBlankChar));
}

// Erase: modify input string
//
// @note Calling reverse_iterator<T>::base() converts the reverse iterator into underlying
// forward iterator, that points to the next element
template<typename S>
// requires is_string<S>
auto trim_right_mod(S&& s) -> std::enable_if_t<is_string<S>, S>
{
    auto&& in = std::forward<S>(s);
    in.erase(std::find_if_not(in.rbegin(), in.rend(), is_blank{}).base(), in.end());

    return in;
}

```

Whenever you want to reuse the reverse iterator with algorithms that require the forward iterator type, just call `base()` - a handy method to convert reverse into forward iterator.

Replace the strings

In similar fashion, we can compare the string replacement algorithms, implemented around the existing std library functions.

```

// Replace "what" with "with", without modifying input string
template<typename S>
// requires is_string<S>
[[nodiscard]] auto replace(std::string& out,
                           const S& in,
                           std::string_view what,
                           std::string_view with) -> std::enable_if_t<is_string<S>, std::string>
{
    std::size_t offset = 0;
    for (;;)
    {
        const auto pos = in.find(what, offset);
        if (pos == std::string::npos) break;
        // replace "what" with corresponding "with"
        out.append(in.substr(offset, pos - offset));
        out.append(with);

        offset = pos + what.size();
    }

    if (offset < in.size()) out.append(in.substr(offset));

    return out;
}

```

What is interesting with `substr` method, is its time complexity, which vary, depends on the (input) string type we have chosen. For `std::string`, `substr()` is $O(n)$ complexity, while for `std::string_view` it's constant time complexity - $O(1)$.

This is no surprise, since `std::string_view` is just the non-owning snapshot of the `std::string` - {ptr, length} pair, so determine a subview is nothing but simple pointer arithmetic operation^{1,2}

Another benefit of the light-weight `std::string_view` is that all methods (except `copy`) are `constexpr`

```

template<typename S>
// requires is_string<S>
[[nodiscard]] auto replace(const S& in,
                           std::string_view what,
                           std::string_view with) -> std::enable_if_t<is_string<S>, std::string>
{
    const auto whatAppearance = count_word_appearance(in, what);
    if (0 == whatAppearance) return std::string(in);

    const std::size_t size = in.size() - whatAppearance * what.size() + whatAppearance * with.size();

    // Starting with C++17, once can use std::pmr::string instead
    // For that, stringify output placeholder needs to be specified at client side
    std::string out;
    out.reserve(size);

    return replace(out, in, what, with);
}

```

Starting with C++20, there is a dedicated `std::string::replace` method

```
namespace details
{
    template <typename StrType = std::string_view>
    [[nodiscard]] auto replace(StrType text, StrType what, StrType with)
    {
        auto txt = std::string {text};
        std::size_t pos = 0;
        for(;;)
        {
            pos = txt.find(what, pos);
            if (pos == StrType::npos) break;
            txt.replace(pos, what.size(), with);
            pos += with.size();
        }
        return txt;
    }
}
```

And, once again, using `regular expressions`

```
// Replace with regular expression
[[nodiscard]] auto replace_with_regex(const std::string& in, std::string_view what, std::string_view with) ->
std::optional<std::string>
{
    try
    {
        std::regex re{what.data()};
        return std::regex_replace(in, re, with.data());
    }
    catch(const std::regex_error& e)
    {
        std::cout << __func__ << ", Exception: " << e.what() << '\n';
        return std::nullopt;
    }
}
```

As you guess by now, we have ranges to come in rescue for most of these algorithms.

We reuse the `split_view` helper method in previous example, and `join` on each splitted word on delimiter "what", the replacement word "with"

```
namespace details
{
    template <typename StrType = std::string_view>
    constexpr auto replace_view(StrType text, StrType what, StrType with)
    {
        using namespace ranges;
        return split_view(text, what) | views::join(with);
    }
}
```

Generators

For generating the sequence of chars - like english alphabet, you can use another std function: `iota`

```
// Generate the char sequence of 'length' chars, starting with 'start' char
[[nodiscard]] std::string generate(char start, std::size_t length)
{
    std::string out(length, '\0');
    std::iota(out.begin(), out.end(), start);

    return out;
}

const auto alphabet = utils::strings::generate('A', 26);
```

And yes - there is matching functionality with ranges as well, which is exactly the point: algorithms should be provided by the library, and not reinvented.

Code

C++17 based solutions.

Compiler Explorer: <https://godbolt.org/z/W5YnP1qdE>

Links

1. <https://www.modernescpp.com/index.php/c-17-avoid-copying-with-std-string-view/>
2. <https://www.cppstories.com/2018/07/string-view-perf/>