Strings (including Regular Expressions)

- Library Basics: String
- Regular Expressions

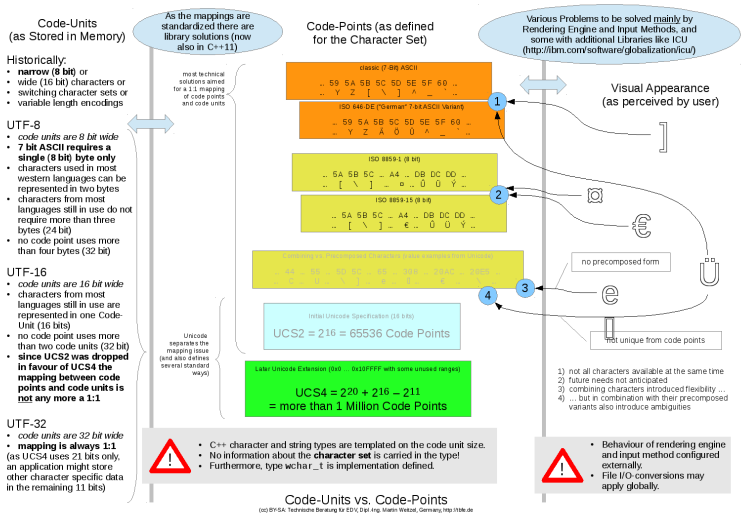
Prelude: Code Units vs. Code Points

The main reason to use strings is to represent "readable" text.

The challenges in the mapping

- of code units (raw storage type)
- via code points (often Unicode)
- to perceived characters

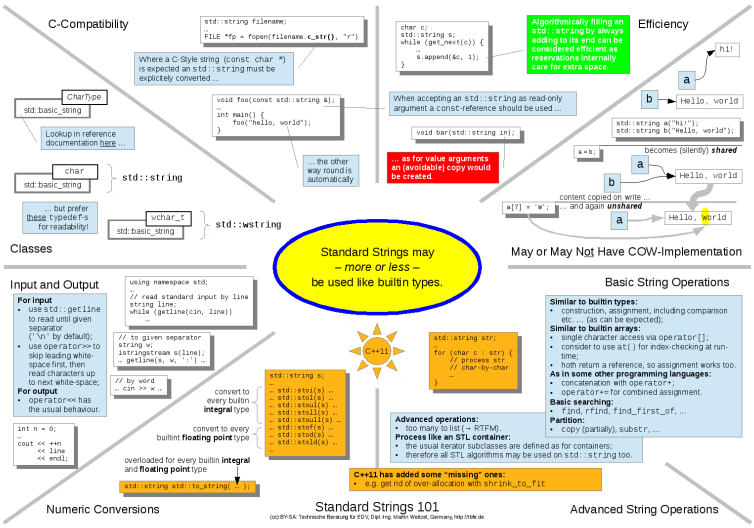
needs to be understood!



Library-Basics – Strings

- Classes (Overview)
- C Compatibility
- Efficiency Considerations
- Optional "Copy On Write"

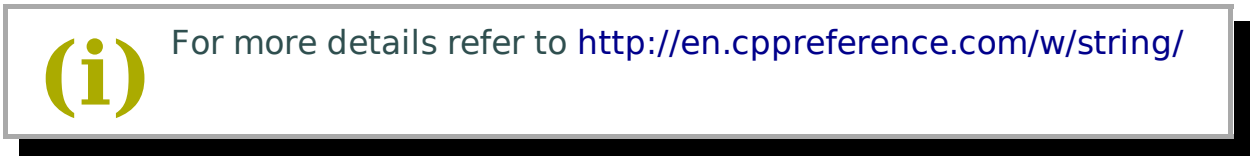
- Basic Operations
- More Operations (Overview)
- Conversions to/from Arithmetic Types
- Input and Output



Classes (Overview)

The classes for character strings `std::string` and `std::wstring` from C++98 were augmented in C++11 with `std::u16string` und `std::u32string`. These are just type definitions like the following:*

```
namespace std {
    typedef basic_string<char> string;      // since C++98
    typedef basic_string<wchar_t> wstring;  // since C++98
    typedef basic_string<char16_t> u16string; // since C++11
    typedef basic_string<char32_t> u32string; // since C++11
}
```



*: The type definitions show only half of the truth: other template arguments are a character traits class and an allocator (memory management policy). Both have been omitted in the example as they do not change the essential point to make.

Compatibility with C

The class `std::_basic_string` stores characters of a string in contiguous* memory, which has at least the required size, but often is allocated with some extra space.

The internal representation frequently is via three pointers:

- The address of the first contained character.
- The address of the last (valid) character.
- The address to mark the end of allocated memory (usually one character beyond)

Also on 64-bit architectures (and for strings of 8-bit wide characters) Small Buffer Optimisation (SBO) is common.

When dealing with the character strings based on the standard string classes there is no restriction to which characters can be contained (other as in C where `'\0'` ends the string).

*: The C++98 standard left it to the library implementors whether to chose contiguous or non-contiguous storage, though that freedom is – at least in effect – removed in C++11.

Using `std::string` as `const char *`

A C-API expecting a C-style string – for example as name of a file – needs an address (of the first character) and `'\0'` termination (to mark the end).

The necessary adaption is done by the `c_str()` member function:

```
std::string filename;
…
… // get file name from user (or elsewhere)
…
// open file for reading, using the C-API
FILE *fp = std::fopen(filename.c_str(), "r");
```

The above code is **correct and bears no risk**, as the pointer returned by `c_str()` (or the memory reachable via it)

- is accessed only **inside** `std::fopen`, this
- does not modify the variable `filename`, and
- therefore any possible heap allocation will not change.

---

*: If the mode to open the file came from an `std::string` too, of course it needs to be converted similarily like that: … `std::fopen( … , openmode.c_str())` …

Risks of Using `c_str()`

It should be understood that when using `c_str()` the memory area accessible via the returned pointer will have a valid content **only** up to the next modifying operation of the underlying string object.

```
std::string s("see me, ");
const char *p = s.c_str();
s += std::string("feel me,");
… // gets s further modified here?
s.append(" touch me, hear me");
```

```
const char *mammamia() {
    std::string local;
    …
    return local.data();  // as
    // of C++11 same as c_str()
}
```

Whether the example on the left is problematic depends on what happens at the section indicated with "…", while the code on the right hand side – **with near to sure probability** – will dereference a pointer to already deallocated heap memory.

[!] The above code has a **high potential risk*** to access invalid memory, allocated for quite a different purpose than to store the characters of the string it once held.

Using `const char *` as `std::string`

Turning a classic C-string into an `std::string` object is always automatic, as it happens as type conversion by a (non-`explicit`) constructor.

> The easiest way to make functions callable with both, `std::string` objects **and** classic C strings is to use an argument of type `const std::string&`.

In terms of the code to be written it will be much more work to provide a number of overloads*, e.g. for

- `const char *`,
- `const std::string &`, and
- (maybe) `std::string &&`.

*: On the other hand, each version could then be optimized for its argument type.

Efficiency Considerations

Typical measures to improve efficiency of `std::string`-s are:

- Allocating some excess memory at the end.
- Proportionally enlarging the allocation (not a constant number of extra characters).
- Especially on 64-Bit hardware: Small String Optimisation

*: Proportional here means that when the current allocation doesn't suffice any more it will be doubled (or made 1.5 or 1.8 times as large). This gives O(1) performance to algorithms that fill a long character string by appending single characters to the end. Increasing the allocation by a constant, fixed amount would yield much worse $O(N^2)$ performance.

Optional "Copy On Write"

This optimisation will especially improve code that hands over `std::string` objects by value (instead of `const std::string&`).

- Copying the string content will not happen right away:*
  - Instead a flag is set to mark the string content as shared, and
  - actually copying the content only if a modification takes place.
- As long as in the shared state only there is only non-modifying access to the string content, nothing more needs to happen.

If a (shared) instance ends its life-time without any modification to the string content, no copying ever needs to take place.

*: Copy On Write (COW) "optimisations" have shown substantial disadvantages in multi-threaded environments: the expected performance improvement often is more then outweighed by necessary locking mechanisms to guarantee exclusive access of only one single thread.

Basic Operations

Most of these are intuitive, like

- Assignment with `=`,
- Comparision with `==`, `!=` etc.
- Concatenation with `+` and
- Element Access with `[…]`.

and hence cause never problems

> Code not extremely performance relevant should consider to use `at()` instead of `operator[]` to avoid undefined behavior in case of out-of-bounds access.

Overview of More Operations

It is fully intended to model `std::string` with a close resemblance to `std::vector<char>`. Especially:

- Also `std::string` has the usual iterator interface,
- making it compatible with all STL algorithms of interest (besides member and free functions dealing with the class itself).

It may be a matter of experience – and to slight degree a matter of taste – what kind of coding style is more comprehensible:

The following fragment reads an `std::string s` from standard input and checks if it contains nothing else but white-space before further processing.

```
// with std::string member function
if (std::getline(std::cin, s)
 && s.find_first_not_of(" \t") == std::string::npos)) …

// with STL algorithm (and predicate specified as lambda)
if (std::getline(std::cin, s)
 && !std::all_of(s.begin(), s.end(),
                 [](char c) { return (c == ' ' || c == '\t'); })) …
```

Converting between `std::string` and Arithmetic Types

Conversions between character sequences and native arithmetic types (`int`, `unsigned`, `long`, … `double`) is a frequent necessity.

Often unnecessary complicated, cumbersome, and hence error prone code is used:

```
std::string tmpfilename; // fixed part, followed by sequence number
…
char *cp = const_cast<char*>(tmpfilename.c_str());
while (*cp && !std::isdigit(*cp))
    ++cp;                          // locate first digit
const int num = std::atoi(cp);  // convert digit sequence to int
std::sprintf(cp, "%d", num+1);  // and store back incremented by 1
```

*: If not obvious from reviewing the code, the fragment above has the following problems:
1. `std::atoi` converts up to the first non-numeric character only (hence a missing numeric part will not be recognized – though in some cases this might be rather a feature than a bug).
2. Some else's memory might be silently overwritten if at `cp` not enough space is available for storing `num` incremented.

Convert `std::string` into Arithmetic Type

C++11 has introduced a new set of functions:

Function Name	Conversion to	(based on)
`std::strtoi`	`int`	`std::strtol`
`std::strtol`	`long`	`std::strtol`
`std::strtoll`	`long long`	`std::strtoll`
`std::strtoul`	`unsigned long`	`std::strtoul`
`std::strtoull`	`unsigned long long`	`std::strtoull`
`std::strtof`	`float`	`std::strtod`
`std::strtod`	`double`	`std::strtod`
`std::strtold`	`long double`	`std::strtold`

Note that the new set provided is a little bit "richer" as the classic (C) functions on which it is based.

For more information see:
http://en.cppreference.com/w/cpp/string/basic_string/stol,
http://en.cppreference.com/w/cpp/string/basic_string/stoul,
and http://en.cppreference.com/w/cpp/string/basic_string/stof.

Converting Strings to Numeric Values

For that purpose individual functions were added carrying the returned type in their name (some `std::` prefixes omitted for brevity):

- `std::stoi`, `…stol`, `…stoll` (returning `int`, `long`, `long long`)
- `std::stoul`, `…stoull` (returning `unsigned long`, `unsigned long long`)
- `std::stod`, `…stod`, `…stold` (returning `float`, `double`, `long double`))

Typical usage fragments (assuming `std::string s` and `std::size_t p`):

```
… = std::stol(s);     // converts from s to long with base 8,
                      // 10 or 16, automatically determined by
                      // prefix 0, no prefix, or prefix 0x / 0X
… = std::stol(s, &p);    // as before with p holding the index
                         // of the character that stopped the
                         // conversion or std::string::npos
… = std::stol(s, &p, 8);    // as before but always uses base 8
… = std::stol(s, &p, 10);   // ... base 10 ...
… = std::stol(s, &p, 16);   // ... base 16 ...
… = std::stol(s, &p, 36);   // ... up to base 36 (0..9-A..Z)
… = std::stol(s, nullptr, 16); // converts from s with base 16,
                               // stop position not of interest
```

Converting Numeric Values to Strings

To convert arithmetic types into `std::string` C++11 introduced a number of overloads for the (free) function `std::to_string`:

The various overloads exist because each type has its own conversion, i.e. there is no dependence on argument type conversions.

Typical usage fragments might look as follows:

```
int i; unsigned u; unsigned long ul; double d;
auto s1 = std::to_string(i);
auto s2 = std::to_string(u);
auto s3 = std::to_string(ul);
auto s4 = std::to_string(d);
```

A big advantage compared to `sprintf` or `snprintf` is there are no more buffers of fixed size involved that might overflow or cause truncation.

Create std::string from Arithmetic Type

The following is a rewrite of a previous example using the new conversion functions:

```
std::string tmpfilename; // fixed part followed by sequence number
…
const auto n1 = tmpfilename.find_first_of("0123456789");
assert(n1 != std::string::npos);
const auto n2 = tmpfilename.find_first_not_of("0123456789", n1+1);
std::size_t nx;
const auto num = std::stou(tmpfilename.substr(n1, n2), &nx);
assert(nx == n2-n1);
tmpfilename = tmpfilename.substr(0, n1)
            + std::to_string(num+1)
            + tmpfilename.substr(n2);
```

Boost: Lexical Cast

With Boost.Lexical_cast there is a particular elegant solution for converting between `std::string`-s and numeric types, spelled `lexical_cast`.*

- Internally `std::stringstream`-s are used, so it can convert
  - **from** every class or basic type that defines and implements `operator<<` as stream insertion,
  - **to** every class or basic type that defines and implements `operator>>` as stream extraction.

The basic usage form looks like this (with `std::string s` and `double d`):

```
s = boost::lexical_cast<std::string>(d);
d = boost::lexical_cast<double>(s);
```

*: Still better: `boost::lexical_cast`s are not limited to conversions which – at one end – have an `std::string` involved, but are capable to cross-convert between anything that adheres to the requirements summarized above.

Input and Output

Output of character strings is usually done with an overload of `operator<<`, sometime even called "output operator":*

```
std::string greet{"hello, world"};
…
std::cout << greet;
```

Note that the result for strings containing embedded `'\0'` characters might not be what is expected (depending on the expectation :-)).

*: Of course, this is not a specific operator for output but an overload to the left-shift operator (as introduced in C), when the left-hand operand is an output stream. Nevertheless, especially when C is used outside the realm of embedded programming, some call `operator<<` now *output operator*.

Reading with operator>>

The overloaded `operator>>` for `std::string`-s reads words by word:

```
std::string word;
while (std::cin >> word) …
```

Words are separated by arbitrary White Space, usually (at least) the characters for:

- Line Feed (`'\n'`)
- Space (`' '`), and
- horizontal / vertical Tabulators (`'\t'` and `'\v'`).

> When reading `std::string`-s with `operator>>` usually no empty lines can be recognized, as any line feed characters are silently skipped as white space.

Reading with `std::getline`

Input into `std::string` can also be read by line …

```
std::string line;
… std::getline(std::cin, line) …
```

… or upto an arbitrary delimiter character:

```
std::string field;
… std::getline(std::cin, field, ':') …
```

> Flexibility of the above is limited as **exactly one** delimiter character may be specified.

It is not possible to specify a set of alternative delimiters – e.g. full stop, comma, and semicolon.*

---

*: While a small helper function accepting a set of delimiters shouldn't be that hard to write, this kind and much more sophisticated parsing of input patterns is possible with Regular Expressions. After having been available through Boost.Regex for a long time – regular expressions became part of the C++11 standard library (see http://en.cppreference.com/w/cpp/regex/basic_regex).

C++11: Regular Expressions

With the adoption of regular expressions in C++11 a powerful library component was made available for:

- Comparing character strings
- Extracting parts from character strings
- Modifying character strings

> Developers experienced with regular expressions usually tend to do nearly every string processing by means of regular expressions.

Compared to an equivalent algorithm using low-level string processing functionality, regular expressions typically

- have much more compact source,
- therefore are better comprehensible and
- hence easier to modify and extend.

Linux <regex> defects warning

[!] For more than a year after major Linux variants had begun the transition to the C++11 library implementation, the header file <regex> was contained with the actual implementation code to 99% broken.

The effect was that a program with regular expression processing compiled but

- either exposed exceptions at run-time (due to the missing parts in the implementation)
- or simply did not do what was expected.

Luckily the regular expressions component from Boost could – *and still can* – be used as compatible replacement.*

*: The difference expresses itself typically only in which header file is included and from which namespace global names are taken, `std::` or `boost::`.

Regular Expressions by Example

A short introduction to regular expressions is best given by example.

To try such examples, a simple small main program is fully sufficient:*

```
#include <iostream>
#include <regex>
#include <string>

int main() {
    using namespace std;
    const string rs{"… (regular expression to try) …"};
    regex rx{rs};
    string line;
    while (getline(std::cin, line)) {
        if (regex_match(line, rx))
            cout << "input matched: " << rs << endl;
        else
            cout << "failed to match: " << rs << endl;
    }
}
```

*: Of course, for any new regular expression to try, the program needs to be recompiled. To avoid this the program could be rewritten to take the regular expression (string) from a command line argument.

Regex Example: Integral Numbers

Recognizing integral numbers with regular expressions is very easy:

- `[0-9]+` says that there should be a digit, optionally repeated but at least one.
- `\d+` is basically the same.
- `-?\d+` allows for an optional sign prefixed to the number.

> **[!]** As a backslash is often part of regular expressions, it must not be forgotten to quote it (by duplication) if the expression is specified as classic literal.

Classic literal:

```
const std::string rs{"-?\\d"};
```

As a Raw String Literal as introduced with C++11:

```
const std::string rs{R"(-?\d)"};
```

Regex Example: Integral Numbers

The following regular expressions describe floating point numbers, starting out simple and adding feature by feature:

> The dot cannot be used directly below and needs to be quoted, as it would represent "any character" otherwise.*

- `\d+[.]\d+` just two dot-separated parts
- `\d*[.]\d+|\d+[.]\d*` digits either before or after the dot optional
- `[-+]?(\d*[.]\d+|\d+[.]\d*)` left hand optional plus or minus sign ...
- `[-+]?(\d*[.]\d+|\d+[.]\d*)([eE][-+]?\d+)?` ... optional exponent ...
- `[-+]?((\d*[.]\d+|\d+[.]\d*)|(\d+[.]?\d*[eE][-+]?\d+))`
  dot optional if there is an exponent

*: Readers already familiar with regular expressions may ask why the dot wasn't quoted with a backslash like in: `\d+(\.\d+){3}`. Of course this would have worked too but the author of this text prefers to quote the dot with a *single character* character class for better readability.

Side Note: Commenting Regular Expressions

Experience tells that simple regular expressions are easily comprehensible after a little training and practicing. With rising complexity explaining the regular expression structure in comments should be considered:

```
… R"([-+]?((\d+[.]\d*|\d*[.]\d+)|(\d+[.]?\d*[eE][-+]?\d+)))" …
//  :|   |^|~~~~~~| ||| |~~~~~~|^|~~~|     |~~~~|  ||~~~|^:
//  :|   |^|      | ||| |      |^|| ||     || sign ^|   |^:
//  :|   |^|right ^^^|| |      |^|| ||     ||  optional |^:
//  :|   |^|... or --^| |      |^|| ||     |^-exponent-^|^:
//  :|   |^|... left -^^^      |^|| |^^^^^^^-- opt. and |^:
//  :|   |||... opt. digits    |||^^^- mandatory digits ||:
//  :|   ||\___ no exponent ___/|\___ with exponent ____/|:
//  :|   |+----<< alternative major components >>--------+:
//  :^^^^^-- optional sign                               :
//  :.............. raw string delimiters .................:
// underlining marks mandatory parts of each major component
// ~~~~~~~~~~~       ~~~~~~~~~ ~~~~~
```

If it gets even more complicated than this a large regular expression might be split into parts which are then explained separately.

Regex Example: IP numbers

An IP number consists of for integral parts separated by dots:

- `\d+[.]\d+[.]\d+[.]\d+` – the straight forward approach
- `\d+([.]\d+){3}` – same with using repetition

Note that the parentheses in the second example will apply the repetition count `{3}` to the whole parenthesized sequence `[.]\d+`.

- Without (necessary) parentheses the meaning would be different: `\d+[.]\d{3}` is a digit sequence of any length, followed by a dot, followed by exactly three more digits.
- On the other hand, parentheses that do not change precedence may be used for more clarity,*
  - e.g. `(\d+{1,3})(([.]\d+){1,3}){3}`
  - similar to `(a*b) + (c/d)` in an arithmetic expression.

---

*: Or rather *in the hope for more clarity*, as too many parentheses also reduce readability ... (but may help to skip over related parts with a parentheses matching editor).

Side Note: More on Parentheses

The parentheses were used in the last examples to changed precedence of regular expression operators, which are from highest to lowest:*

- optionality and repetition (?, +, *, {…}, {…,…})
- concatenation ("invisible" operator)
- alternative (|)

When extracting parts of a string controlled by regular expressions, parentheses may also determine the (sub-) sequences of interest.

> There are also purely grouping parentheses, written as (:? and). But as their use tends to make regular expressions slightly less readable they were avoided in the introductory examples.

*: There are a few more like anchoring at the beginning (^) or at the end ($) for which special rules apply as there use is limited to certain places.

The Trick to Do Group Separators

Frequently numbers are written by separating digits into groups:

- Currencies often group digits by three, except for the fractional part, which typically has exactly two digits (or may be omitted).
- The German way to write telephone numbers (according to the DIN) is to build groups of two digits from the right, separated by blanks.

```
std::regex euro_currency("[1-9]\\d{,2}([.]\\d{3})*(,\\d{2})?");
// first up to three .....|^^^^^^^^^^||         | |        |
// (opt. repeated) groups of three ...|^^^^^^^^^| |        |
// finally (opt.) cents ..........................|^^^^^^^^|

std::regex din_telephone(
//|------- optional ------------------------|
//|        including ONE trailing blank ---| |
//|   v--- surrounded by parentheses ----v V |
 "(?:[(]" "(0|0[1-9])(?: [0-9][0-9])*" "[)] )?" // <-- area code
         "[1-9][0-9]?(?: [0-9][0-9])*"          // <-- local line
);
```

---

*: That second part of the example uses String Literal Concatenation (introduced with C89 but maybe a lesser known feature of the C++ language) and makes the regex c'tor argument more stand-out.

The Trick to Do White Space Sequences

Space characters in regular expressions represent themselves,* i.e. they must match exactly the *same amount and kind* of white space.

Regular expressions may be easily bloated with `\s+` or `[ \t]*` if they are used to match text in which

- sequences of blanks or even any white space (including tab etc.) can be used with the same meaning as a single blank, or
- white space may optionally be interspersed at many locations.

A good solution is to run a normalization step on a string before checking it against a regular expression:

1. Remove all **optional** white space (and sequences thereof).
2. Where white space is **mandatory** turn sequences of any kind of white space into exactly one single blank.

*: Some variants of regular expressions allow to take away any meaning from white space embedded in a regular expression, so that it may be used for structuring. Where a space characters has to occur literally, `\s` can be used (and `\t` for tabs etc.).

The Trick NOT Trying It Too Perfect

You might feel tempted to use sequence counts for direct control of digit sequences, like in the IP number example each components can have no more than three digits:

- `\d+{1,3}([.]\d{1,3}){3}` allows at most three digits in each part

But this is of course not sufficient to exclude any invalid input, as the numeric values between dots must not exceed 255. Which problems arise when overdoing range checks are easily demonstrated in the simple example to recognize a month in a date as a number between 1 and 12:

- `1[0-2]|[1-9]` for 1, 2, 3 ... 12 like in English dates (8/12/2014)
- `1[0-2]|0[1-9]` for 01, 02, ... 12 like in ISO dates (2014-08-12)

So anything* is possible at the cost of readability ... but:

> Often the better approach is to allow any number (maybe with checking for an upper limit of contained digits) and to postpone more validity tests until the numeric value is picked up from the string.

Accessing Sub-Patterns

Once some text matches a given regular expression, it is very easy to access sub-patterns.

At first parentheses need to be placed in the regular expression to mark the sub-patterns of interest:

```
// start c'tor -+       v-----------------------------v
//   arg. list  |       | 1st sub-pattern selection   |
std::regex telno{"(?:[(]((0[1-9]?(?: [0-9][0-9])*)[)] )?"
                      "([1-9][0-9]?(?: [0-9][0-9])*)"};
                 //   | 2nd sub-pattern selection | |   end c'tor
                 //   ^---------------------------^ +-- arg. list
```

Then `std::regex_match` is called with an additional argument:

```
std::string input;
while (std::getline(std::cin, input)) {
    std::smatch select;
    if (std::regex_match(input, select, telno))
        std::cout << "area code:  " << select[1] << '\n'
                  << "local line: " << select[2] << '\n';
}
```

Matching vs. Searching

If in an regular expression there is only a single sub-pattern of interest instead of `std::regex_match` using `std::regex_find` should be considered:

Assuming an `std::string input` the following

```
std::regex eur_amount("EUR \\d+,\\d{2}");
if (std::regex_search(input, smatch, eur_amount)) …
   … smatch[0] … // access whole match
```

is equivalent to:

```
std::regex eur_amount(".*?(EUR \\d+,\\d{2}).*");
if (std::regex_match(input, smatch, eur_amount)) …
   … smatch[1] … // access first (and only) sub-match
```

Here the *non-greedy* variant `*?` of the plain repetition operator `*` has been used, though it only makes a difference if in `input` there are two possible matches starting with "EUR" each*

*: Getting accustomed to take care helps to avoid subtle problems like that of `".*(\\d+).*"` which often come to surprise of regular expression newbies ...

Doing Substitutions

Regular expressions can also be used to describe what should be substituted inside a string:*

```
std::string s;
… // read an input line into s

// remove leading and trailing white space:
s = std::regex_replace(s, std::regex("^[ \t]+"), "");
s = std::regex_replace(s, std::regex("[ \t]+$"), "");
// normalize remaining white space to a single blank:
s = std::regex_replace(s, std::regex("[ \t]+"), " ");
```

The use of ^… and …$ in the regular expressions above will *anchor the patterns* to match only at the beginning or the end respectively.

> Preprocessing to normalize input in which white space sequences are optionally allowed at many places can help to simplify regular expressions used later for input analysis.

*: It is **not** an oversight to use `\t`, not `\\t` despite the classic form of string literal, as the former gets translated to a tab character which is then taken literally in the regular expression.

Reusing Sub-Matches in Substitutions

Substitutions controlled by regular expression become even more powerful as sub-matches can easily be reused in substitutions.

The following code swaps

- currency names (first sub-pattern matched by `EUR|DKK|GBP|SEK|NOK`)
- with amounts (second sub-pattern matched by `\d+,\d{2}`):*

```
std::regex eur_amount("(EUR|DKK|GBP|SEK|NOK) (\\d+,\\d{2})");
…
std::regsub(input, eur_amount, "$2 $1");
```

> The default syntax to refer to matched text is that of ECMA-Script.

Alternatively POSIX Rules can be enabled with the flag `std::regex_constants::format_sed` as fourth argument.

*: Using double backslashes in the regular expression specification is necessary because it is specified here as classic C string literal.

Boost: Regular Expressions

C++11 regular expressions emerged from Boost.Regex.

Therefore there is little difference:

- Some options and features available from Boost have not been standardized.
- Moreover, Boost regular expression* may evolve more rapidly with respect of new, experimental features or dialects supported.

> Unless you have a broken implementation of regular expressions the standard implementation is usually sufficient and there is little need to use Boost regular expressions.

*: To avoid ambiguity it should be noted that Boost actually supplies two variants of regular expressions: The full-blown implementation available as a separate component (to which this page refers), and a separate, much stripped down implementation bundled with Boost.String_algo, the extended string algorithms.

Boost: Xpressive

Processing regular expressions is usually broken into two parts:

- Constructing a state machine (FSM) from what is textually specified.
- Executing that FSM guided by an actual string to compare to an regular expression, access parts, etc.

> The separation makes sense because the first part is typically more time consuming but will need to happen only once, while the second may happen frequently but executes fast.

With C++11 and Boost both parts usually take place at runtime.*

Boost.Xpressive implements regular expressions with Meta Programming techniques, effectively shifting the first part from runtime to compile-time.

*: Of course, any program using regular expressions should be structured to exploit this separation, shifting the FSM construction out of the loops that actually match against or otherwise process strings with the regular expression machinery.

Boost: More String Operations

Two more libraries of Boost which specifically deal with string processing are

- Boost.String_algo and
- Boost.Tokenizer.

The first provides many string processing functions that some might feel "missing" in the standard libraries, the second provides (relatively) easy but nevertheless flexible mechanisms to "chop" a longer string into parts.