

# C++11 BLWS (Tuesday 1)

## Basic and Advanced String Processing

---

1. Standard Strings (Recap)
  2. Boost: Tokenizer
  3. C++11: String Conversions
  4. Boost: Lexical Cast
  5. C++11: Regular Expressions
  6. Boost: Regular Expressions
  7. Boost: Xpressive
  8. Boost: Spirit
- 

Short breaks will be inserted as convenient.

# Standard Strings (Recap)

The class `std::string` was introduced in its current form with C++98. while C++11 extended it in a number of ways it left the basic design unchanged:

- The `std::string` object itself has a fixed size, typically holding just some pointers.\*
- The string content - its "characters" - is stored on the heap in space that is automatically allocated as required.
- Very often, for efficiency purposes if a string gets extended by adding characters one after the other to its end, there is some unused extra space kept at the end of the payload.

---

\*: E.g. it may use three pointers: to refer to the string's first character, its last, and the upper limit of the extra space available for extending the string at its end.

# Non-modifying `std::string` Functions

There are a few trivial functions like:

- `empty()` to find out if the string contains any characters at all, or
- `size()` or `length()` to determine how many characters the string contains,

Others are at most moderately complex but not heavily complex, like:

- Searching a string for given characters or patterns ...
- ... from either end ...
- ... possibly skipping over already considered parts.
- Accessing parts of a string (single characters or sub-strings).

This is all well documented in many books and online references and usually causes no problems (beyond the level of general C++ programming).

# Modifying `std::string` Functions

The various functions to modify `std::string` range from

- modify a single character, over
- modify a substring, to
- modify the whole string.

If a string is shortened there usually is no behavioral guarantee, in fact implementations vary with respect to when and how many of the used space will be eventually returned (and hence become available for other usages).

## C++ `std::string` vs. C-Style Strings

What has to be understood and taken into account is the link from `std::string` to the string representation used in C.

Strings as used in C boil down to a pointer referring to the string's first character. Furthermore a special termination is expected as part of the string, that is, the `'\0'` character, a byte (or word) with all bits cleared.

Because of C compatibility there exist still a number of functions in the C++ standard library dealing with a C-style string representation which is always at the risk of becoming invalid when a string's content changes.

## C++ template `std::basic_string`

Depending on the character set used and its stored representation there may be different types necessary for storing the single characters of a string.\*

To avoid code duplication C++ implements all string functionality only once for the template `std::basic_string` - via typedef-s - making two instantiations available in C++98 with the names `std::string` and `std::wstring`.

With C++11 more standard string types were added that internally use UTF16 or UTF32, named `std::u16string` and `std::u32string`, with the underlying character type `char16_t` and `char32_t`.

---

\*: Other template arguments to instantiate the `std::basic_string` class are a traits class for the character type used and an allocator

# Boost: Tokenizer

All Boost additions for improved string processing in C++ build on the standard string class.

Boost.Tokenizer makes string tokenization available, for which a very basic usage example looks as follows:

```
#include <boost/tokenizer.hpp>
#include <string>
#include <iostream>

int main() {
    std::string line;
    while (std::getline(std::cin, line)) {
        for (auto w : boost::tokenizer<>(s))
            std::cout << w << std::endl;
    }
}
```

---

\*: To avoid ambiguity it should be noted that Boost actually supplies two variants of string tokenizer implementationss: The "official" component (the larger one to which this slide refers), and a separate, stripped down implementation bundled with its extended string algorithms.

# C++11: String Conversions

A significant addition were conversions between to and from numeric types:

- For the former a number of overloads for the global function `std::to_string` were supplied.
- The latter was done with a set of differently named functions, all taking either a `std::string` or `std::wstring` argument.



# Converting Numeric Values to Strings

The various overloads exist because each type has its own conversion, i.e. there is no dependance 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 compred to sprintf or snprintf is there are no more buffers of fixed size involved that might overflow or cause truncation.

# Converting Strings to Numeric Values

For that purpose individual functions were added carrying the returned type in their name:

- `std::stoi` (returning `int`)
- `std::stol` (returning `long`)
- `std::stoll` (returning `long long`)
- `std::stod` (returning `double`)
- ...

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

```
... = std::stoi(s);           // converts from s with base 10

... = std::stol(s, &p);      // as before with p holding index of
                             // character that stopped the
                             // conversion

... = std::stol(s, &p, 8);    // as before but converts with base 8

... = std::stol(s, nullptr, 0); // converts from s with the base
                                // determined automatically (e.g.
                                // base 16 for prefix 0x or 0X)
```

# Boost: Lexical Cast

The Boost library offers a particular elegant solution for converting between strings and numeric types, called `lexical_cast`.<sup>\*</sup>

- 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);
```

---

<sup>\*</sup>: Still better: `lexical_cast`s are not even 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.

# 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 usually tend to do nearly every string processing by means of regular expressions.

It is much more compact source, therefore better comprehensible and easier to modify and extend like an equivalent algorithm using low-level string processing features.

## Linux <regex> defects warning



For more that a year after major Linux variants had begun the transition to the C++11 library implementation, the header file `regex` was contained but the actually 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 programm 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 it could be considered to take the regular expression 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"};
```

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: `\d+(\. \d+){3}` Of course this would have worked too but the author of this text prefers to quote with "single character character classes" wherever possible for better readability.



## Sidenote: 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 four integral parts separated by dots:

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

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

- Without (necessary) parentheses the meaning would change:  
`\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 written for more clarity,<sup>\*</sup>
  - e.g. `(\d+{1,3})(([\.]\d+{1,3})){3}`
  - similar to  $(a*b) + (c/d)$  in an arithmetic expression.

---

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

## Sidenote: More on Parentheses

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

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

When extracting parts of a string controlled by regular expressions, round 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 for the moment.

---

\*: 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 - if it exists - typically has exactly two digits.
- German way to write telephone numbers (according to the DIN) is to build groups of two digits from the right, separated by blanks.

Here is the trick to do both:

```
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
);
```

## The Trick to do White Space Sequences

Space characters in regular expressions represent themselves,<sup>\*</sup> 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 where

- 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](#) before checking against a regular expressions:

1. Remove **all optional** white space (and sequences thereof).
2. Where white space is **mandatory** turn sequences into a single blank.

---

<sup>\*</sup>: Some variants of regular expressions allow to take away any meaning from white space embedded in a regular expression, so 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 numbers between dots must not exceed 255. Which problems arise when overdoing range checks are easily demonstrate in the simple example to recognizing 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<sup>\*</sup> is possible at the cost of readability ... but:

Usually the better approach is just to allow any number, maybe with checking only for an upper limit of contained digits, and to postpone more value checking until the numbers are 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{"(?:([0-9]{1,3})?(?:[0-9]{2,3})?)?"
                "([0-9]{2,3})?(?:[0-9]{2,3})?"};
                //   | 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. Finding

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_find(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 there are two possible matches starting with "EUR" each in input.\*

\*: 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.

## 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::regexsub(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.

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 impenentation** 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 expression implementation: The full-blown implementation available as a separate component (to which this page refers), and a separate stripped down implementation bundled with the extended string algorithms available through Boost.

# 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: Spirit

Though regular expressions are quite powerful, they have a limit - then more powerful tools like [Boost.Spirit](#) enter the scene.

The abstract grammar fragments shown below - they parse a comma separated sequence of floating point values inside curly braces and store them in a sequential container data - can very well be just a part of more complex overall syntax.

```
... '{' << ( float_[push_back(ref(data), _1)]
           << *( ',' << float_[push_back(ref(data), _1)]) ) << '}' ...
```

Or yet simpler:

```
... '{' << (float_[push_back(Phoenix::ref(data), _1)] % ',') << '}' ...
```

---

\*: To a certain degree this may be overcome by combining regular expressions with other string processing facilities, but on the long run switching to superior tools and techniques is advisable.