C++11 BLWS (Tuesday 1)

Basic and Advanced String Processing

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 automatically allocated as required.
- For efficiency (if a string gets extended by adding characters one after the other to its end) there is typically some unused extra space after 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 member 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 basic 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).

*: These functions are also members of the `std::string` class, though for some of them that design has sometimes been criticised with the argument that only operations closely related to accessing or modifying the private state of an object should be members, while for others, especially those that manufacture and return a new instance – like `std::string::substr` – global functions should have been preferred.

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 exists 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 representation of Code-Points as Code-Units 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 two instantiations are made available in C++98, named

- `std::string` and
- `std::wstring`.

With C++11 more standard string types were added that internally use UTF16 or UTF32, named

- `std::u16string` (underlying character type `char16_t`) and
- `std::u32string` (underlying character type `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<>(line))
            std::cout << w << std::endl;
    }
}
```

*: To avoid ambiguity it should be noted that Boost actually supplies two variants of string tokenizers: Shown above is the larger one, and then there is a tiny, stripped down implementation bundled with Boost.String_algorithms.

C++11: String Conversions

A significant addition were conversions 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 an `std::string` or an `std::wstring` argument.

Converting Numeric Values to Strings

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.

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
```

Boost: Lexical Cast

Quick Link to Introduction

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.

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 a long time even after major Linux variants had begun the transition to C++11 the header file <regex> supplied an actual implementation that was to 99% broken.

The effect is that a program with regular expressions compiles flawlessly **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 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::`. By using a namespace alias the switch between `std::regex` and Boost.Regex can be made with a changing to some few letters in a single line.

Regular Expressions by Example

A short introduction to regular expressions is best given by example using a small program demonstrating pattern matching with `std::regex_match`:*

```
#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: Floating Point 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+))` (and this?)

---

*: 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 have been 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{1,2}([.]\\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[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';
}
```

* 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.

???