7. Strings and Regular Expressions

Prefer the standard to the offbeat.

– Strunk & White

• Introduction

• Strings

string Implementation

• Regular Expressions

Searching; Regular Expression Notation; Iterators

• Advice

7.1. INTRODUCTION

Text manipulation is a major part of most programs. The C++ standard library offers a sting type to save most users from C-style manipulation of arrays of characters through pointers. In addition, regular expression matching is offered to help find patterns in text. The regular expressions are provided in a form similar to what is common in most modern languages. Both strings and regex objects can use a variety of character types (e.g., Unicode).

The standard library provides a string type to complement the string literals (§1.3). The string type provides a variety of useful string operations, such as concatenation. For example:

Click here to view code image

```
string compose(const string& name, const string& domain)
{
        return name + '@' + domain;
}


auto addr = compose("dmr","bell–labs.com");
```

Here, addr is initialized to the character sequence dmr@bell–labs.com. "Addition" of strings means concatenation. You can concatenate a string, a string literal, a C-style string, or a character to a string. The standard string has a move constructor so returning even long strings by value is efficient (§4.6.2).

In many applications, the most common form of concatenation is adding something to the end of a string. This is directly supported by the += operation. For example:

Click here to view code image

```
void m2(string& s1, string& s2)
{
        s1 = s1 + '\n';    // append newline
        s2 += '\n';          // append newline
}
```

The two ways of adding to the end of a string are semantically equivalent, but I prefer the latter because it is more explicit about what it does, more concise, and possibly more efficient.

A string is mutable. In addition to = and +=, subscripting (using [ ]), and substring operations are supported. Among other useful features, it provides the ability to manipulate substrings. For example:

Click here to view code image

```
string name = "Niels Stroustrup";

void m3()
{
        string s = name.substr(6,10);          // s = "Stroustrup"
        name.replace(0,5,"nicholas");          // name becomes "nich
        name[0] = toupper(name[0]);            // name becomes "N
}
```

The substr() operation returns a string that is a copy of the substring indicated by its arguments. The first argument is an index into the string (a position), and the second is the length of the desired substring. Since indexing starts from 0, s gets the value Stroustrup.

The replace() operation replaces a substring with a value. In this case, the substring starting at 0 with length 5 is Niels; it is replaced by nicholas. Finally, I replace the initial character with its uppercase equivalent. Thus, the final value of name is Nicholas Stroustrup. Note that the replacement string need not be the same size as the substring that it is replacing.

Naturally, strings can be compared against each other and against string literals. For example:

Click here to view code image

```
string incantation;

void respond(const string& answer)
{
        if (answer == incantation) {
                // perform magic
        }
        else if (answer == "yes") {
                // ...
        }
        // ...
}
```

Among the many useful string operations are assignment (using =), subscripting (using [ ] or at() as for vector; §9.2.2), iteration (using iterators as for **vector**; §10.2), input (§8.3), streaming (§8.8).

If you need a C-style string (a zero-terminated array of **char**), **string** offers read-only access to its contained characters. For example:

Click here to view code image

```
void print(const string& s)
{
        printf("For people who like printf: %s\n",s.c_str());
        cout << "For people who like streams: " << s << '\n';
}
```

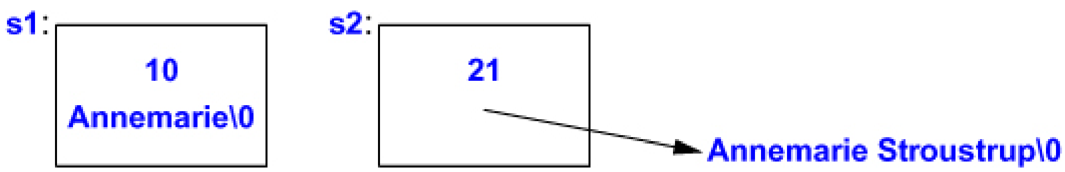
7.2.1. string Implementation

Implementing a string class is a popular and useful exercise. However, for general-purpose use, our carefully crafted first attempts rarely match the standard **string** in convenience or performance. These days, **string** is usually implemented using the *short-string optimization*. That is, short string values are kept in the **string** object itself and only longer strings are placed on free store. Consider:

Click here to view code image

```
string s1 {"Annemarie"};                    //  short string
string s2 {"Annemarie Stroustrup"};    //  long string
```

The memory layout will be something like:



When a **string**'s value changes from a short to a long string (and vice verse) its representation adjusts appropriately.

The actual performance of **string**s can depend critically on the run-time environment. In particular, in multi-threaded implementations, memory allocation can be relatively costly. Also, when lots of strings of differing lengths are used, memory fragmentation can result. These are the main reasons that the short-string optimization has become ubiquitous.

To handle multipe character sets, string is really an alias for a general template basic_string with the character type char:

Click here to view code image

```
template<typename Char>
class basic_string {
        // ... string of Char ...
};

using string = basic_string<char>
```

A user can define strings of arbitrary character types. For example, assuming we have a Japanese character type Jchar, we can write:

Click here to view code image

```
using Jstring = basic_string<Jchar>;
```

Now we can do all the usual string operations on Jstring, a string of Japanese characters. Similarly, we can handle Unicode strings.

7.3. REGULAR EXPRESSIONS

Regular expressions are a powerful tool for text processing. They provide a way to simply and tersely describe patterns in text (e.g., a U.S. postal code such as TX 77845, or an ISO-style date, such as 2009–06–07) and to efficiently find such patterns in text. In <regex>, the standard library provides support for regular expressions in the form of the std::regex class and its supporting functions. To give a taste of the style of the regex library, let us define and print a pattern:

Click here to view code image

```
regex pat {R"(\w{2}\s *\d{5}(–\d{4})?)"}; // US postal code patter
```

People who have used regular expressions in just about any language will find \w{2}\s *\d{5}(–\d{4})? familiar. It specifies a pattern starting with two letters \w{2} optionally followed by some space \s * followed by five digits \d{5} and optionally followed by a dash and four digits –\d{4}. If you are not familiar with regular expressions, this may be a good time to learn about them ([Stroustrup,2009], [Maddock,2009], [Friedl,1997]).

To express the pattern, I use a *raw string literal* starting with R"(and terminated by)". This allows backslashes and quotes to be used directly in the string. Raw strings are particularly suitable for regular expressions because they tend to contain a lot of backslashes. Had I used a conventional string, the pattern definition would have been:

Click here to view code image

```
regex pat {"\\w{2}\\s*\\d{5}(–\\d{4})?"};         // U.S. postal code
```

In <regex>, the standard library provides support for regular expressions:

• regex_match(): Match a regular expression against a string (of known size) (§7.3.2).

• regex_search(): Search for a string that matches a regular expression in an (arbitrarily long) stream of data (§7.3.1).

• regex_replace(): Search for strings that match a regular expression in an (arbitrarily long) stream of data and replace them.

• regex_iterator: Iterate over matches and submatches (§7.3.3).

• regex_token_iterator: Iterate over non-matches.

7.3.1. Searching

The simplest way of using a pattern is to search for it in a stream:

Click here to view code image

```
int lineno = 0;
for (string line; getline(cin,line);) {          // read into line buffer
        ++lineno;
        smatch matches;                                 // match
        if (regex_search(line,matches,pat))        // search for pa
                cout << lineno << ": " << matches[0] << '\n';
}
```

The regex_search(line,matches,pat) searches the line for anything that matches the regular expression stored in pat and if it finds any matches, it stores them in matches. If no match was found, regex_search(line,matches,pat) returns false. The matches variable is of type smatch. The “s” stands for “sub” or “string,” and an smatch is a vector of sub-matches of type string. The first element, here matches[0], is the complete match. The result of a regex_search() is a collection of matches, typically represented as an smatch:

Click here to view code image

```
void use()
{
        ifstream in("file.txt");       // input file
        if (!in)                              // check that the file was op
                cerr << "no file\n";

        regex pat {R"(\w{2}\s*\d{5}(–\d{4})?)"};       // U.S. postal c

        int lineno = 0;
        for (string line; getline(in,line);) {
                ++lineno;
                smatch matches;          // matched strings go here
                if (regex_search(line, matches, pat)) {
                        cout << lineno << ": " << matches[0] << '\n';
                        if (1<matches.size() && matches[1].matched)
                                cout << "\t: " << matches[1] << '\n';
                }
        }
}
```

This function reads a file looking for U.S. postal codes, such as TX77845 and DC 20500–0001. An smatch type is a container of regex results. Here, matches[0] is the whole pattern and matches[1] is the optional four-digit subpattern.

The regular expression syntax and semantics are designed so that regular expressions can be compiled into state machines for efficient execution [Cox,2007]. The regex type performs this compilation at run time.

7.3.2. Regular Expression Notation

The regex library can recognize several variants of the notation for regular expressions. Here, I use the default notation used, a variant of the ECMA standard used for ECMAScript (more commonly known as JavaScript).

The syntax of regular expressions is based on characters with special meaning:

Regular Expression Special Characters			
.	Any single character (a "wildcard")	\	Next character has a special meaning
[	Begin character class	*	Zero or more (suffix operation)
]	End character class	+	One or more (suffix operation)
{	Begin count	?	Optional (zero or one) (suffix operation)
}	End count	|	Alternative (or)
(	Begin grouping	^	Start of line; negation
)	End grouping	$	End of line

For example, we can specify a line starting with zero or more As followed by one or more Bs followed by an optional C like this:

```
^A *B+C?$
```

Examples that match:

```
AAAAAAAAAAAABBBBBBBBBC
BC
B
```

Examples that do not match:

```
AAAAA              // no B
   AAAABC          // initial space
AABBCC             // too many Cs
```

A part of a pattern is considered a subpattern (which can be extracted separately from an smatch) if it is enclosed in parentheses. For example:

```
\d+-\d+            // no subpatterns
\d+(-\d+)          // one subpattern
(\d+)(-\d+)        // two subpatterns
```

A pattern can be optional or repeated (the default is exactly once) by adding a suffix:

Repetition	
{n}	Exactly n times
{n,}	n or more times
{n,m}	At least n and at most m times
*	Zero or more, that is, {0,}
+	One or more, that is, {1,}
?	Optional (zero or one), that is {0,1}

For example:

```
A{3}B{2,4}C *
```

Examples that match:

```
AAABBC
AAABBB
```

Example that do not match:

```
AABBC          // too few As
AAABC          // too few Bs
AAABBBBBCCC     // too many Bs
```

A suffix ? after any of the repetition notations (?, *, ?, and { }) makes the pattern matcher "lazy" or "non-greedy." That is, when looking for a pattern, it will look for the shortest match rather than the longest. By default, the pattern matcher always looks for the longest match; this is known as the *Max Munch rule*. Consider:

```
ababab
```

The pattern (ab) * matches all of ababab. However, (ab) *? matches only the first ab.

The most common character classifications have names:

Character Classes	
alnum	Any alphanumeric character
alpha	Any alphabetic character
blank	Any whitespace character that is not a line separator
cntrl	Any control character
d	Any decimal digit
digit	Any decimal digit
graph	Any graphical character
lower	Any lowercase character
print	Any printable character
punct	Any punctuation character
s	Any whitespace character
space	Any whitespace character
upper	Any uppercase character
w	Any word character (alphanumeric characters plus the underscore)
xdigit	Any hexadecimal digit character

In a regular expression, a character class name must be bracketed by [: :]. For example, [:digit:] matches a decimal digit. Furthermore, they must be used within a [ ] pair defining a character class.

Several character classes are supported by shorthand notation:

Character Class Abbreviations		
\d	A decimal digit	[[:digit:]]
\s	A space (space, tab, etc.)	[[:space:]]
\w	A letter (a-z) or digit (0-9) or underscore (_)	[_[:alnum:]]
\D	Not \d	[^[:digit:]]
\S	Not \s	[^[:space:]]
\W	Not \w	[^_[:alnum:]]

In addition, languages supporting regular expressions often provide:

Nonstandard (but Common) Character Class Abbreviations		
\l	A lowercase character	[[:lower:]]
\u	An uppercase character	[[:upper:]]
\L	Not \l	[^[:lower:]]
\U	Not \u	[^[:upper:]]

For full portability, use the character class names rather than these abbreviations.

As an example, consider writing a pattern that describes C++ identifiers: an underscore or a letter followed by a possibly empty sequence of letters, digits, or underscores. To illustrate the subtleties involved, I include a few false attempts:

Click here to view code image

```
[:alpha:][:alnum:]*                   //  wrong: characters from the se
[[:alpha:]][[:alnum:]]*               //  wrong: doesn't accept undersco
([[:alpha:]]|_)[[:alnum:]]*           //  wrong: underscore is not part of a

([[:alpha:]]|_)([[:alnum:]]|_)*        //  OK, but clumsy
[[:alpha:]_][[:alnum:]_]*               //  OK: include the undersco
[_[:alpha:]][_[:alnum:]]*               //  also OK
[_[:alpha:]]\w*                           //  \w is equivalent to [
```

Finally, here is a function that uses the simplest version of **regex_match()** (§7.3.1) to test whether a string is an identifier:

Click here to view code image

```
bool is_identifier(const string& s)
{
        regex pat {"[_[:alpha:]]\\w*"}; //  underscore or letter
                                                    //  followed by zero
        return regex_match(s,pat);
}
```

Note the doubling of the backslash to include a backslash in an ordinary string literal. Use raw string literals to alleviate problems with special characters. For example:

Click here to view code image

```
bool is_identifier(const string& s)
{
        regex pat {R"([_[:alpha:]]\w*)"};
        return regex_match(s,pat);
}
```

Here are some examples of patterns:

Click here to view code image

```
Ax*                  // A, Ax, Axxxx
Ax+                  // Ax, Axxx       Not A
\d–?\d               // 1–2, 12        Not 1––2
\w{2}–\d{4,5}        // Ab–1234, XX–54321, 22–5432          Digits a
(\d*:)?(\d+)         // 12:3, 1:23, 123, :123    Not 123:
(bs|BS)              // bs, BS          Not bS
[aeiouy]             // a, o, u         An English vowel, not x
[^aeiouy]            // x, k            Not an English vowel, not e
[a^eiouy]            // a, ^, o, u      An English vowel or^
```

A **group** (a subpattern) potentially to be represented by a **sub_match** is delimited by parentheses. If you need parentheses that should not define a subpattern, use **(?** rather than plain **(**. For example:

Click here to view code image

```
(\s|:|,)*(\d*)        // spaces, colons, and/or commas followed by a
```

Assuming that we were not interested in the characters before the number (presumably separators), we could write:

Click here to view code image

```
(?\s|:|,)*(\d*)       // spaces, colons, and/or commas followed by a n
```

This would save the regular expression engine from having to store the first characters: the **(?** variant has only one subpattern.

Regular Expression Grouping Examples	
\d*\s\w+	No groups (subpatterns)
(\d*)\s(\w+)	Two groups
(\d*)(\s(\w+))+	Two groups (groups do not nest)
(\s*\w*)+	One group, but one or more subpatterns; only the last subpattern is saved as a **sub_match**
<(.*?)>(.*?)</\1>	Three groups; the \1 means "same as group 1"

That last pattern is useful for parsing XML. It finds tag/end-of-tag markers. Note that I used a non-greedy match (a *lazy match*), **.*?**, for the subpattern between the tag and the end tag. Had I used plain **.***, this input would have caused a problem:

Click here to view code image

```
Always look for the <b>bright</b> side of <b>life</b>.
```

A *greedy match* for the first subpattern would match the first < with the last >. A greedy match on the second subpattern would match the first with the last . Both would be correct behavior, but unlikely what the programmer wanted.

For a more exhaustive presentation of regular expressions, see [Friedl,1997].

7.3.3. Iterators

We can define a regex_iterator for iterating over a stream finding matches for a pattern. For example, we can output all whitespace-separated words in a string:

Click here to view code image

```
void test()
{
        string input = "aa as; asd ++e^asdf asdfg";
        regex pat {R"(\s+(\w+))"};
        for (sregex_iterator p(input.begin(),input.end(),pat); p!=srege
                cout << (*p)[1] << '\n';
}
```

This outputs:

```
as
asd
asdfg
```

Note that we are missing the first word, aa, because it has no preceding whitespace. If we simplify the pattern to R"((\ew+))", we get

```
aa
as
asd
```

```
e
asdf
asdfg
```

A regex_iterator is a bidirectional iterator, so we cannot directly iterate over an istream. Also, we cannot write through a regex_iterator, and the default regex_iterator (regex_iterator{}) is the only possible end-of-sequence.

7.4. ADVICE

[1] The material in this chapter roughly corresponds to what is described in much greater detail in Chapters 36-37 of [Stroustrup,2013].

[2] Prefer string operations to C-style string functions; §7.1.

[3] Use string to declare variables and members rather than as a base class; §7.2.

[4] Return strings by value (rely on move semantics); §7.2, §7.2.1.

[5] Directly or indirectly, use substr() to read substrings and replace() to write substrings; §7.2.

[6] A string can grow and shrink, as needed; §7.2.

[7] Use at() rather than iterators or [ ] when you want range checking; §7.2.

[8] Use iterators and [ ] rather than at() when you want to optimize speed; §7.2.

[9] string input doesn't overflow; §7.2, §8.3.

[10] Use c_str() to produce a C-style string representation of a string (only) when you have to; §7.2.

[11] Use a string_stream or a generic value extraction function (such as to<X>) for numeric conversion of strings; §8.8.

[12] A basic_string can be used to make strings of characters on any type; §7.2.1.

[13] Use regex for most conventional uses of regular expressions; §7.3.

[14] Prefer raw string literals for expressing all but the simplest patterns; §7.3.

[15] Use regex_match() to match a complete input; §7.3, §7.3.2.

[16] Use regex_search() to search for a pattern in an input stream; §7.3.1.

[17] The regular expression notation can be adjusted to match various standards; §7.3.2.

[18] The default regular expression notation is that of ECMAScript; §7.3.2.

[19] Be restrained; regular expressions can easily become a write-only language; §7.3.2.

[20] Note that \i allows you to express a subpattern in terms of a previous subpattern; §7.3.2.

[21] Use ? to make patterns "lazy"; §7.3.2.

[22] Use regex_iterators for iterating over a stream looking for a pattern; §7.3.3