23

Text Manipulation

"Nothing is so obvious that it's obvious . . .
The use of the word 'obvious' indicates
the absence of a logical argument."

—Errol Morris

This chapter is mostly about extracting information from text. We store lots of our knowledge as words in documents, such as books, email messages, or "printed" tables, just to later have to extract it into some form that is more useful for computation. Here, we review the standard library facilities most used in text processing: **string**s, **iostream**s, and **map**s. Then, we introduce regular expressions (**regex**s) as a way of expressing patterns in text. Finally, we show how to use regular expressions to find and extract specific data elements, such as ZIP codes (postal codes), from text and to verify the format of text files.

23.1 Text

We manipulate text essentially all the time. Our books are full of text, much of what we see on our computer screens is text, and our source code is text. Our communication channels (of all sorts) overflow with words. Everything that is communicated between two humans could be represented as text, but let's not go overboard. Images and sound are usually best represented as images and sound (i.e., just bags of bits), but just about everything else is fair game for program text analysis and transformation.

We have been using **iostream**s and **string**s since Chapter 3, so here, we'll just briefly review those libraries. Maps (§23.4) are particularly useful for text processing, so we present an example of their use for email analysis. After this review, this chapter is concerned with searching for patterns in text using regular expressions (§23.5–10).

23.2 Strings

A **string** contains a sequence of characters and provides a few useful operations, such as adding a character to a **string**, giving the length of the **string**, and concatenating **string**s. Actually, the standard **string** provides quite a few operations, but most are useful only when you have to do fairly complicated text manipulation at a low level. Here, we just mention a few of the more useful. You can look up their details (and the full set of **string** operations) in a manual or expert-level textbook should you need them. They are found in **<string>** (note: not **<string.h>**):

Selected **string** operations	
s1 = s2	Assign **s2** to **s1**; **s2** can be a **string** or a C-style string.
s += x	Add **x** at end; **x** can be a character, a **string**, or a C-style string.
s[i]	Subscripting.
s1+s2	Concatenation; the characters in the resulting **string** will be a copy of those from **s1** followed by a copy of those from **s2**.
s1==s2	Comparison of **string** values; **s1** or **s2**, but not both, can be a C-style string. Also **!=**.
s1<s2	Lexicographical comparison of **string** values; **s1** or **s2**, but not both, can be a C-style string. Also **<=**, **>**, and **>=**.
s.size()	Number of characters in **s**.
s.length()	Number of characters in **s**.
s.c_str()	C-style version of characters in **s**.
s.begin()	Iterator to first character.
s.end()	Iterator to one beyond the end of **s**.
s.insert(pos,x)	Insert **x** before **s[pos]**; **x** can be a **string** or a C-style string. **s** expands to make room for the characters from **x**.
s.append(x)	Insert **x** after the last character of **s**; **x** can be a **string** or a C-style string. **s** expands to make room for the characters from **x**.
s.erase(pos)	Remove trailing characters from **s** starting with **s[pos]**. **s**'s size becomes **pos**.
s.erase(pos,n)	Remove **n** characters from **s** starting at **s[pos]**. **s**'s size becomes **max(pos,size–n)**.
pos = s.find(x)	Find **x** in **s**; **x** can be a character, a **string**, or a C-style string; **pos** is the index of the first character found, or **string::npos** (a position off the end of **s**).
in>>s	Read a whitespace-separated word into **s** from **in**.
getline(in,s)	Read a line into **s** from **in**.
out<<s	Write from **s** to **out**.

The I/O operations are explained in Chapters 10 and 11 and summarized in §23.3. Note that the input operations into a **string** expand the **string** as needed, so that overflow cannot happen.

The **insert()** and **append()** operations may move characters to make room for new characters. The **erase()** operation moves characters "forward" in the **string** to make sure that no gap is left where we erased a character.

The standard library **string** is really a template, called **basic_string**, that supports a variety of character sets, such as Unicode, providing thousands of characters (such as £, Ω, μ, δ, ☺, and 𝅘𝅥𝅮 in addition to "ordinary characters"). For example, if you have a type holding a Unicode character, such as **Unicode**, you can write

```
basic_string<Unicode> a_unicode_string;
```

The standard string, **string**, which we have been using, is simply the **basic_string** of an ordinary **char**:

```
using string = basic_string<char>;     // string means basic_string<char> (§20.5)
```

We do not cover Unicode characters or Unicode strings here, but if you need them you can look them up, and you'll find that they can be handled (by the language, by **string**, by **iostream**s, and by regular expressions) much as ordinary characters and strings. If you need to use Unicode characters, it is best to ask someone experienced for advice; to be useful, your code has to follow not just the language rules but also some system conventions.

In the context of text processing, it is important that just about anything can be represented as a string of characters. For example, here on this page, the number **12.333** is represented as a string of six characters (surrounded by whitespace). If we read this number, we must convert those characters to a floating-point number before we can do arithmetic operations on the number. This leads to a need to convert values to **string**s and **string**s to values. In §11.4, we saw how to turn an integer into a **string** using an **ostringstream**. This technique can be generalized to any type that has a **<<** operator:

```
template<typename T> string to_string(const T& t)
{
        ostringstream os;
        os << t;
        return os.str();
}
```

For example:

```
string s1 = to_string(12.333);
string s2 = to_string(1+5*6–99/7);
```

The value of **s1** is now **"12.333"** and the value of **s2** is **"17"**. In fact, **to_string()** can be used not just for numeric values, but for any class **T** with a **<<** operator.

The opposite conversion, from **string**s to numeric values, is about as easy, and as useful:

```
struct bad_from_string : std::bad_cast {  // class for reporting string cast errors
        const char* what() const override
        {
                return "bad cast from string";
        }
};

template<typename T> T from_string(const string& s)
{
        istringstream is {s};
        T t;
        if (!(is >> t)) throw bad_from_string{};
        return t;
}
```

For example:

```
double d = from_string<double>("12.333");

void do_something(const string& s)
try
{
        int i = from_string<int>(s);
        // . . .
}
catch (bad_from_string e) {
        error("bad input string",s);
}
```

The added complication of **from_string()** compared to **to_string()** comes because a **string** can represent values of many types. This implies that we must say which type of value we want to extract from a **string**. It also implies that the **string** we are looking at may not hold a representation of a value of the type we expect. For example:

```
int d = from_string<int>("Mary had a little lamb");        // oops!
```

So there is a possibility of error, which we have represented by the exception **bad_from_string**. In §23.9, we demonstrate how **from_string()** (or an equivalent function) is essential for serious text processing because we need to extract numeric values from text fields. In §16.4.3, we saw how an equivalent function **get_int()** was used in GUI code.

Note how **to_string()** and **from_string()** are similar in function. In fact, they are roughly inverses of each other; that is (ignoring details of whitespace, rounding, etc.), for every "reasonable type **T**" we have

```
s==to_string(from_string<T>(s))      // for all s
```

and

```
t==from_string<T>(to_string(t))      // for all t
```

Here, "reasonable" means that **T** should have a default constructor, a **>>** operator, and a matching **<<** operator defined.

Note also how the implementations of **to_string()** and **from_string()** both use a **stringstream** to do all the hard work. This observation has been used to define a general conversion operation between any two types with matching **<<** and **>>** operations:

```
template<typename Target, typename Source>
Target to(Source arg)
{
        stringstream interpreter;
        Target result;

        if (!(interpreter << arg)                          // write arg into stream
                || !(interpreter >> result)                // read result from stream
                || !(interpreter >> std::ws).eof())        // stuff left in stream?
                        throw runtime_error{"to<>() failed"};

        return result;
}
```

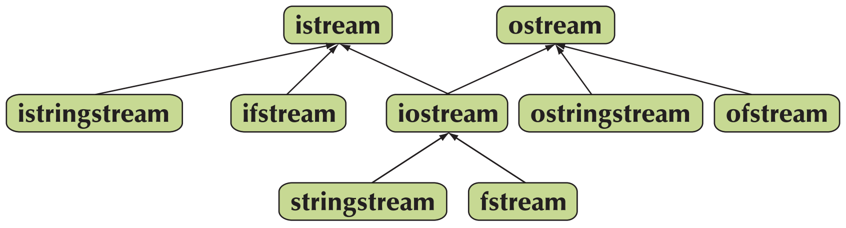
The curious and clever **!(interpreter>>std::ws).eof()** reads any whitespace that might be left in the **stringstream** after we have extracted the result. Whitespace is allowed, but there should be no more characters in the input and we can check that by seeing if we are at "end of file." So if we try to read an **int** from a **string**, both **to<int>("123")** and **to<int>("123 ")** will succeed, but **to<int>("123.5")** will not because of that last **.5**.

23.3 I/O streams

Considering the connection between strings and other types, we get to I/O streams. The I/O stream library doesn't just do input and output; it also performs conversions between string formats and types in memory. The standard library I/O streams provide facilities for reading, writing, and formatting strings of characters. The iostream library is described in Chapters 10 and 11, so here we'll just summarize:

Stream I/O	
in >> x	Read from **in** into **x** according to **x**'s type.
out << x	Write **x** to **out** according to **x**'s type.
in.get(c)	Read a character from **in** into **c**.
getline(in,s)	Read a line from **in** into the string **s**.

The standard streams are organized into a class hierarchy (§14.3):



Together, these classes supply us with the ability to do I/O to and from files and strings (and anything that can be made to look like a file or a string, such as a keyboard and a screen; see Chapter 10). As described in Chapters 10 and 11, the **iostreams** provide fairly elaborate formatting facilities. The arrows indicate inheritance (see §14.3), so that, for example, a **stringstream** can be used as an **iostream** or as an **istream** or as an **ostream**.

Like **string**, **iostreams** can be used with larger character sets such as Unicode, much like ordinary characters. Please again note that if you need to use Unicode I/O, it is best to ask someone experienced for advice; to be useful, your code has to follow not just the language rules but also some system conventions.

23.4 Maps

Associative arrays (maps, hash tables) are key (pun intended) to a lot of text processing. The reason is simply that when we process text, we collect information, and that information is often associated with text strings, such as names, addresses, postal codes, Social Security numbers, job titles, etc. Even if some of those text strings could be converted into numeric values, it is often more convenient and simpler to treat them as text and use that text for identification. The word-counting example (§21.6) is a good simple example. If you don't feel comfortable using **maps**, please reread §21.6 before proceeding.

Consider email. We often search and analyze email messages and email logs – usually with the help of some program (e.g., Thunderbird or Outlook). Mostly, those programs save us from seeing the complete source of the messages, but all the information about who sent, who received, where the message went along the way, and much more is presented to the programs as text in a message header. That's a complete message. There are thousands of tools for analyzing the headers. Most use regular expressions (as described in §23.5–9) to extract information and some form of associative arrays to associate related messages. For example, we often search a mail file to collect all messages with the same sender, the same subject, or containing information on a particular topic.

Here, we will use a very simplified mail file to illustrate some of the techniques for extracting data from text files. The headers are real RFC2822 headers from www.faqs.org/rfcs/rfc2822.html. Consider:

```
xxx
xxx
----
From: John Doe <jdoe@machine.example>
To: Mary Smith <mary@example.net>
Subject: Saying Hello
Date: Fri, 21 Nov 1997 09:55:06 -0600
Message-ID: <1234@local.machine.example>

This is a message just to say hello.
So, "Hello".
----
From: Joe Q. Public <john.q.public@example.com>
To: Mary Smith <@machine.tld:mary@example.net>, , jdoe@test   .example
Date: Tue, 1 Jul 2003 10:52:37 +0200
Message-ID: <5678.21-Nov-1997@example.com>

Hi everyone.
----
To: "Mary Smith: Personal Account" <smith@home.example>
From: John Doe <jdoe@machine.example>
Subject: Re: Saying Hello
```

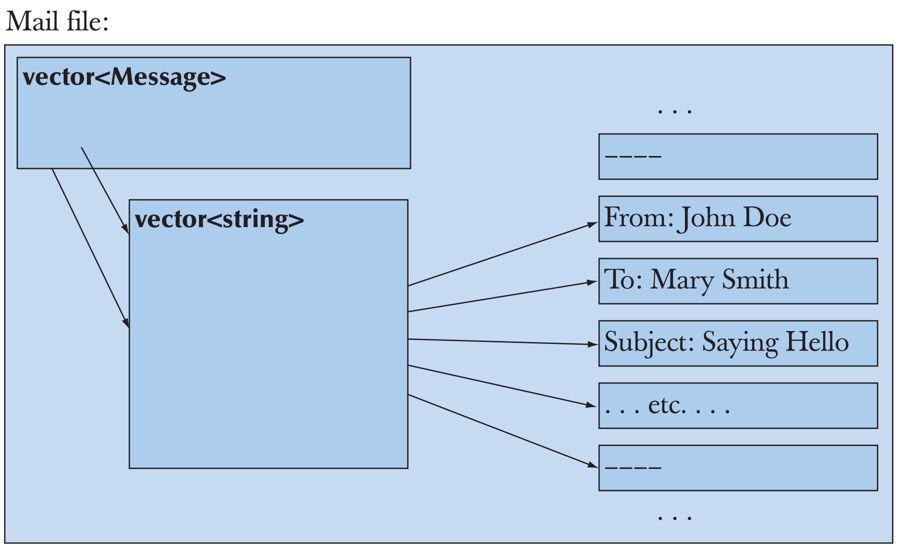
```
Date: Fri, 21 Nov 1997 11:00:00 –0600
Message–ID: <abcd.1234@local.machine.tld>
In–Reply–To: <3456@example.net>
References: <1234@local.machine.example> <3456@example.net>

This is a reply to your reply.
––––
––––
```

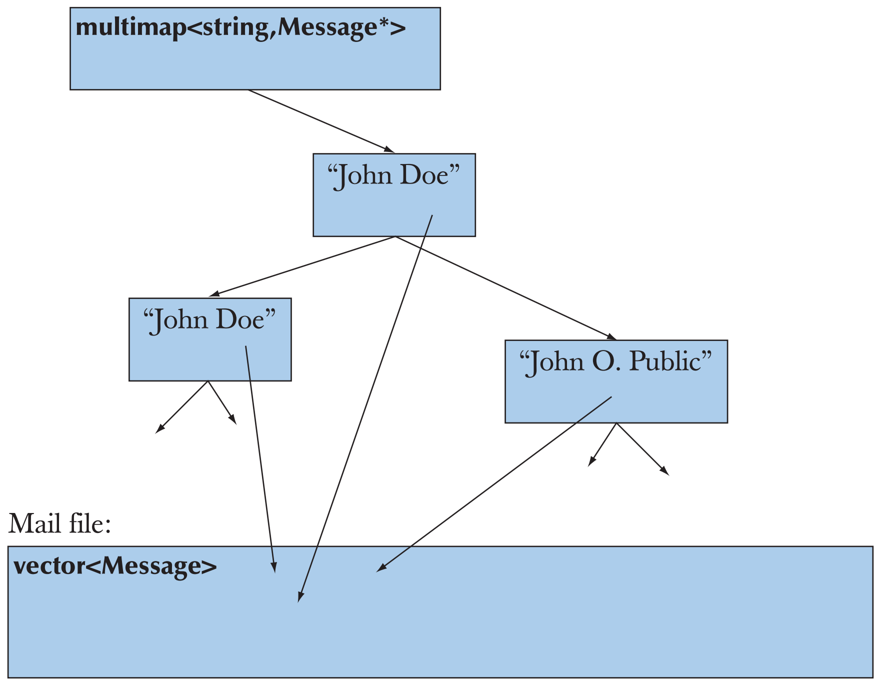
Basically, we have abbreviated the file by throwing most of the information away and eased the analysis by terminating each message by a line containing just **––––** (four dashes). We will write a small "toy application" that finds all messages sent by "John Doe" and write out their "Subject." If we can do that, we can do many interesting things.

First, we must consider whether we want random access to the data or just to analyze it as it streams by in an input stream. We choose the former because in a real program, we would probably be interested in several senders or in several pieces of information from a given sender. Also, it's actually the harder of the two tasks, so it will allow us to examine more techniques. In particular, we get to use iterators again.

Our basic idea is to read a complete mail file into a structure (which we call a **Mail_file**). This structure will hold all the lines of the mail file (in a **vector<string>**) and indicators of where each individual message starts and ends (in a **vector<Message>**):



To this, we will add iterators and **begin()** and **end()** functions, so that we can iterate through the lines and through the messages in the usual way. This "boiler-plate" will allow us convenient access to the messages. Given that, we will write our "toy application" to gather all the messages from each sender so that they are easy to access together:



Finally, we will write out all the subject headers of messages from "John Doe" to illustrate a use of the access structures we have created.

We use many of the basic standard library facilities:

```
#include<string>
#include<vector>
#include<map>
#include<fstream>
#include<iostream>
using namespace std;
```

We define a **Message** as a pair of iterators into a **vector<string>** (our vector of lines):

```
typedef vector<string>::const_iterator Line_iter;

class Message {   // a Message points to the first and the last lines of a message
        Line_iter first;
        Line_iter last;
public:
        Message(Line_iter p1, Line_iter p2) :first{p1}, last{p2} { }
        Line_iter begin() const { return first; }
        Line_iter end() const { return last; }
        // . . .
};
```

We define a **Mail_file** as a structure holding lines of text and messages:

```
using Mess_iter = vector<Message>::const_iterator;

struct Mail_file {                  // a Mail_file holds all the lines from a file
                                    // and simplifies access to messages
        string name;                // file name
        vector<string> lines;       // the lines in order
        vector<Message> m;          // Messages in order

        Mail_file(const string& n); // read file n into lines

        Mess_iter begin() const { return m.begin(); }
        Mess_iter end() const { return m.end(); }
};
```

Note how we added iterators to the data structures to make it easy to systematically traverse them. We are not actually going to use standard library algorithms here, but if we wanted to, the iterators are there to allow it.

To find information in a message and extract it, we need two helper functions:

```
// find the name of the sender in a Message;
// return true if found
// if found, place the sender's name in s:
bool find_from_addr(const Message* m, string& s);

// return the subject of the Message, if any, otherwise "":
string find_subject(const Message* m);
```

Finally, we can write some code to extract information from a file:

```
int main()
{
        Mail_file mfile {"my–mail–file.txt"};         // initialize mfile from a file

        // first gather messages from each sender together in a multimap:

        multimap<string, const Message*> sender;

        for (const auto& m : mfile) {
                string s;
                if (find_from_addr(&m,s))
                        sender.insert(make_pair(s,&m));
        }

        // now iterate through the multimap
        // and extract the subjects of John Doe's messages:
        auto pp = sender.equal_range("John Doe <jdoe@machine.example>");
        for(auto p = pp.first; p!=pp.second; ++p)
                cout << find_subject(p–>second) << '\n';
}
```

Let us examine the use of maps in detail. We used a **multimap** (§20.10, §B.4) because we wanted to gather many messages from the same address together in one place. The standard library **multimap** does that (makes it easy to access elements with the same key). Obviously (and typically), we have two parts to our task:

- Build the map.
- Use the map.

We build the **multimap** by traversing all the messages and inserting them into the **multimap** using **insert()**:

```
for (const auto& m : mfile) {
        string s;
        if (find_from_addr(&m,s))
                sender.insert(make_pair(s,&m));
}
```

What goes into a map is a (key,value) pair, which we make with **make_pair()**. We use our "homemade" **find_from_addr()** to find the name of the sender.

Why did we first put the **Message**s in a **vector** and then later build a **multimap**? Why didn't we just put the **Message**s into a **map** immediately? The reason is simple and fundamental:

- First, we build a general structure that we can use for many things.
- Then, we use that for a particular application.

That way, we build up a collection of more or less reusable components. Had we immediately built a **map** in the **Mail_file**, we would have had to redefine it whenever we wanted to do some different task. In particular, our **multimap** (significantly called **sender**) is sorted based on the Address field of a message. Most other applications would not find that order particularly useful: they might be looking at Return fields, Recipients, Copy-to fields, Subject fields, time stamps, etc.

This way of building applications in stages (or *layers*, as the parts are sometimes called) can dramatically simplify the design, implementation, documentation, and maintenance of programs. The point is that each part does only one thing and does it in a straightforward way. On the other hand, doing everything at once would require cleverness. Obviously, our "extracting information from an email header" program was just a tiny example of an application. The value of keeping separate things separate, modularization, and gradually building an application increases with size.

To extract information, we simply find all the entries with the key **"John Doe"** using the **equal_range()** function (§B.4.10). Then we iterate through all the elements in the sequence [first,second) returned by **equal_range()**, extracting the subject by using **find_subject()**:

```
auto pp = sender.equal_range("John Doe <jdoe@machine.example>");

for (auto p = pp.first; p!=pp.second; ++p)
        cout << find_subject(p–>second) << '\n';
```

When we iterate over the elements of a **map**, we get a sequence of (key,value) pairs, and as with all **pair**s, the first element (here, the **string** key) is called **first** and the second (here, the **Message** value) is called **second** (§21.6).

23.4.1 Implementation details

Obviously, we need to implement the functions we use. It was tempting to save a tree by leaving this as an exercise, but we decided to make this example complete. The **Mail_file** constructor opens the file and constructs the **lines** and **m** vectors:

```
Mail_file::Mail_file(const string& n)
        // open file named n
        // read the lines from n into lines
        // find the messages in the lines and compose them in m
        // for simplicity assume every message is ended by a –––– line
{
        ifstream in {n};                    // open the file
        if (!in) {
                cerr << "no " << n << '\n';
                exit(1);                    // terminate the program
        }

        for (string s; getline(in,s); )     // build the vector of lines
                lines.push_back(s);

        auto first = lines.begin();         // build the vector of Messages
        for (auto p = lines.begin(); p!=lines.end(); ++p) {
                if (*p == "––––") {         // end of message
                        m.push_back(Message(first,p));
                        first = p+1;        // –––– not part of message
                }
        }
}
```

The error handling is rudimentary. If this were a program we planned to give to friends to use, we'd have to do better.

TRY THIS

We really mean it: do run this example and make sure you understand the result. What would be "better error handling"? Modify **Mail_file**'s constructor to handle likely formatting errors related to the use of **––––**.

The **find_from_addr()** and **find_subject()** functions are simple placeholders until we can do a better job of identifying information in a file (using regular expressions; see §23.6–10):

```
int is_prefix(const string& s, const string& p)
        // is p the first part of s?
{
        int n = p.size();
        if (string(s,0,n)==p) return n;
        return 0;
}
```

```
bool find_from_addr(const Message* m, string& s)
{
        for (const auto& x : *m)
                if (int n = is_prefix(x, "From: ")) {
                        s = string(x,n);
                        return true;
                }
        return false;
}

string find_subject(const Message* m)
{
        for (const auto& x : *m)
                if (int n = is_prefix(x, "Subject: ")) return string(x,n);
        return "";
}
```

Note the way we use substrings: **string(s,n)** constructs a string consisting of the tail of **s** from **s[n]** onward (**s[n]..s[s.size()–1]**), whereas **string(s,0,n)** constructs a string consisting of the characters **s[0]..s[n–1]**. Since these operations actually construct new strings and copy characters, they should be used with care where performance matters.

Why are the **find_from_addr()** and **find_subject()** functions so different? For example, one returns a **bool** and the other a **string**. They are different because we wanted to make a point:

- **find_from_addr()** distinguishes between finding an address line with an empty address (**""**) and finding no address line. In the first case, **find_from_addr()** returns **true** (because it found an address) and sets **s** to **""** (because the address just happens to be empty). In the second case, it returns **false** (because there was no address line).
- **find_subject()** returns **""** if there was an empty subject or if there was no subject line.

Is the distinction made by **find_from_addr()** useful? Necessary? We think that the distinction can be useful and that we definitely should be aware of it. It is a distinction that comes up again and again when looking for information in a data file: did we find the field we were looking for and was there something useful in it? In a real program, both the **find_from_addr()** and **find_subject()** functions would have been written in the style of **find_from_addr()** to allow users to make that distinction.

This program is not tuned for performance, but it is probably fast enough for most uses. In particular, it reads its input file only once, and it does not keep multiple copies of the text from that file. For large files, it may be a good idea to replace the **multimap** with an **unordered_multimap**, but unless you measure, you'll never know.

See §21.6 for an introduction to the standard library associative containers (**map**, **multimap**, **set**, **unordered_map**, and **unordered_multimap**).

23.5 A problem

I/O streams and **string** help us read and write sequences of characters, help us store them, and help with basic manipulation. However, it is very common to do operations on text where we need to consider the context of a string or involve many similar strings. Consider a trivial example. Take an email message (a sequence of words) and see if it contains a U.S. state abbreviation and ZIP code (two letters followed by five digits):

```
for (string s; cin>>s; ) {
        if (s.size()==7
        && isalpha(s[0]) && isalpha(s[1])
        && isdigit(s[2]) && isdigit(s[3]) && isdigit(s[4])
        && isdigit(s[5]) && isdigit(s[6]))
                cout << "found " << s << '\n';
}
```

Here, **isalpha(x)** is **true** if **x** is a letter and **isdigit(x)** is **true** if **x** is a digit (see §11.6).

There are several problems with this simple (too simple) solution:

- It's verbose (four lines, eight function calls).
- We miss (intentionally?) every postal code not separated from its context by whitespace (such as **"TX77845"**, **TX77845–1234**, and **ATX77845**).
- We miss (intentionally?) every postal code with a space between the letters and the digits (such as **TX 77845**).
- We accept (intentionally?) every postal code with the letters in lower case (such as **tx77845**).
- If we decide to look for a postal code in a different format (such as **CB3 0FD**), we have to completely rewrite the code.

There has to be a better way! Before revealing that way, let's just consider the problems we would encounter if we decided to stay with the "good old simple way" of writing more code to handle more cases.

- If we want to deal with more than one format, we'd have to start adding **if**-statements or **switch**-statements.
- If we want to deal with upper and lower case, we'd explicitly have to convert (usually to lower case) or add yet another **if**-statement.
- We need to somehow (how?) describe the context of what we want to find. That implies that we must deal with individual characters rather than with strings, and that implies that we lose many of the advantages provided by **iostream**s (§7.8.2).

If you like, you can try to write the code for that, but it is obvious that on this track we are headed for a mess of **if**-statements dealing with a mess of special cases. Even for this simple example, we need to deal with alternatives (e.g., both five- and nine-digit ZIP codes). For many other examples, we need to deal with repetition (e.g., any number of digits followed by an exclamation mark, such as **123!** and **123456!**). Eventually, we would also have to deal with both prefixes and suffixes. As we observed (§11.1–2), people's tastes in output formats are not limited by a programmer's desire for regularity and simplicity. Just think of the bewildering variety of ways people write dates:

```
2007–06–05
June 5, 2007
jun 5, 2007
5 June 2007
6/5/2007
5/6/07
. . .
```

At this point – if not earlier – the experienced programmer declares, "There has to be a better way!" (than writing more ordinary code) and proceeds to look for it. The simplest and most popular solution is using what are called *regular expressions*. Regular expressions are the backbone of much text processing, the basis for the Unix grep command (see exercise 8), and an essential part of languages heavily used for such processing (such as AWK, PERL, and PHP).

The regular expressions we will use are part of the C++ standard library. They are compatible with the regular expressions in PERL. This makes many explanations, tutorials, and manuals available. For example, see the C++ standard committee's working paper (look for "WG21" on the web), John Maddock's **boost::regex** documentation, and most PERL tutorials. Here, we will describe the fundamental concepts and some of the most basic and useful ways of using regular expressions.

TRY THIS

The last two paragraphs "carelessly" used several names and acronyms without explanation. Do a bit of web browsing to see what we are referring to.

23.6 The idea of regular expressions

The basic idea of a regular expression is that it defines a pattern that we can look for in a text. Consider how we might concisely describe the pattern for a simple U.S. postal code, such as **TX77845**. Here is a first attempt:

```
wwddddd
```

Here, **w** represents "any letter" and **d** represents "any digit." We use **w** (for "word") because **l** (for "letter") is too easily confused with the digit 1. This notation works for this simple example, but let's try it for the nine-digit ZIP code format (such as **TX77845–5629**). How about

```
wwddddd–dddd
```

That looks OK, but how come that **d** means "any digit" but **–** means "plain" dash? Somehow, we ought to indicate that **w** and **d** are special: they represent character classes rather than themselves (**w** means "an **a** or a **b** or a **c** or . . ." and **d** means "a **1** or a **2** or a **3** or . . ."). That's too subtle. Let's prefix a letter that is a name of a class of characters with a backslash in the way special characters have always been indicated in C++ (e.g., **\n** is newline in a string literal). This way we get

```
\w\w\d\d\d\d\d–\d\d\d\d
```

This is a bit ugly, but at least it is unambiguous, and the backslashes make it obvious that "something unusual is going on." Here, we represent repetition of a character by simply repeating. That can be a bit tedious and is potentially error-prone. Quick: Did we really get the five digits before the dash and four after it right? We did, but nowhere did we actually *say* **5** and **4**, so you had to count to make sure. We could add a count after a character to indicate repetition. For example:

```
\w2\d5–\d4
```

However, we really ought to have some syntax to show that the **2**, **5**, and **4** in that pattern are counts, rather than just the alphanumeric characters **2**, **5**, and **4**. Let's indicate counts by putting them in curly braces:

```
\w{2}\d{5}–\d{4}
```

That makes **{** special in the same way as \ (backslash) is special, but that can't be helped and we can deal with that.

So far, so good, but we have to deal with two more messy details: the final four digits in a ZIP code are optional. We somehow have to be able to say that we will accept both **TX77845** and **TX77845–5629**. There are two fundamental ways of expressing that:

```
\w{2}\d{5} or \w{2}\d{5}–\d{4}
```

and

```
\w{2}\d{5} and optionally –\d{4}
```

To say that concisely and precisely, we first have to express the idea of grouping (or sub-pattern) to be able to speak about the **\w{2}\d{5}** and **–\d{4}** parts of **\w{2}\d{5}–\d{4}**. Conventionally, we use parentheses to express grouping:

```
(\w{2}\d{5})(–\d{4})
```

Now we have split the pattern into two sub-patterns, so we just have to say what we want to do with them. As usual, the cost of introducing a new facility is to introduce another special character: **(** is now "special" just like \ and **{**. Conventionally **|** is used to express "or" (alternatives) and **?** is used to express something conditional (optional), so we might write

```
(\w{2}\d{5})|(\w{2}\d{5}–\d{4})
```

and

```
(\w{2}\d{5})(–\d{4})?
```

As with the curly braces in the count notation (e.g., **\w{2}**), we use the question mark (**?**) as a suffix. For example, **(–\d{4})?** means "optionally **–\d{4}**"; that is, we accept four digits preceded by a dash as a suffix. Actually, we are not using the parentheses around the pattern for the five-digit ZIP code (\w{2}\d{5}) for anything, so we could leave them out:

```
\w{2}\d{5}(–\d{4})?
```

To complete our solution to the problem stated in §23.5, we could add an optional space after the two letters:

```
\w{2} ?\d{5}(–\d{4})?
```

That " ?" looks a bit odd, but of course it's a space character followed by the ?, indicating that the space character is optional. If we wanted to avoid a space being so unobtrusive that it looks like a bug, we could put it in parentheses:

```
\w{2}( )?\d{5}((–\d{4})?
```

If someone considered that still too obscure, we could invent a notation for a whitespace character, such as \s (s for "space"). That way we could write

```
\w{2}\s?\d{5}(–\d{4})?
```

But what if someone wrote two spaces after the letters? As defined so far, the pattern would accept TX77845 and TX 77845 but not TX 77845. That's a bit subtle. We need to be able to say "zero or more whitespace characters," so we introduce the suffix * to mean "zero or more" and get

```
\w{2}\s*\d{5}(–\d{4})?
```

This makes sense if you followed every step of the logical progression. This notation for patterns is logical and extremely terse. Also, we didn't pick our design choices at random: this particular notation is extremely common and popular. For many text-processing tasks, you need to read and write this notation. Yes, it looks a bit as if a cat walked over the keyboard, and yes, typing a single character wrong (even a space) completely changes the meaning, but please just get used to it. We can't suggest anything dramatically better, and this style of notation has already been wildly popular for more than 30 years since it was first introduced for the Unix grep command – and it wasn't completely new even then.

23.6.1 Raw string literals

Note all of those backslashes in the regular expression patterns. To get a backslash (\) into a C++ string literal we have to precede it with a backslash. Consider our postal code pattern:

```
\w{2}\s*\d{5}(–\d{4})?
```

To represent that pattern as a string literal, we have to write

```
"\\w{2}\\s*\\d{5}(–\\d{4})?"
```

Thinking a bit ahead, we realize that many of the patterns we would like to match contain double quotes ("). To get a double quote into a string literal we have to precede it with a backslash. This can quickly become unmanageable. In fact, in real use this “special character problem” gets so annoying that C++ and other languages have introduced the notion of raw string literals to be able to cope with realistic regular expression patterns. In a raw string literal a backslash is simply a backslash character (rather than an escape character) and a double quote is simply a double quote character (rather than an end of string). As a raw string literal our postal code pattern becomes

```
R"(\w{2}\s*\d{5}(–\d{4})?)"
```

The R"(starts the string and)" terminates it, so the 22 characters of the string are

```
\w{2}\s*\d{5}(–\d{4})?
```

not counting the terminating zero.

23.7 Searching with regular expressions

Now, we will use the postal code pattern from the previous section to find postal codes in a file. The program defines the pattern and then reads a file line by line, searching for the pattern. If the program finds an occurrence of the pattern in a line, it writes out the line number and what it found:

```
#include <regex>
#include <iostream>
#include <string>
#include <fstream>
using namespace std;

int main()
{
        ifstream in {"file.txt"};                        // input file
        if (!in) cerr << "no file\n";

        regex pat {R"(\w{2}\s*\d{5}(–\d{4})?)"};         // postal code pattern
        int lineno = 0;
        for (string line; getline(in,line); ) {    // read input line into input buffer
                ++lineno;
                smatch matches;                    // matched strings go here
                if (regex_search(line, matches, pat))
                        cout << lineno << ": " << matches[0] << '\n';
        }
}
```

This requires a bit of a detailed explanation. We find the standard library regular expressions in **<regex>**. Given that, we can define a pattern **pat**:

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

A **regex** pattern is a kind of **string**, so we can initialize it with a string. Here, we used a raw string literal. However, a **regex** is not *just* a **string**, but the somewhat sophisticated mechanism for pattern matching that is created when you initialize a **regex** (or assign to one) is hidden and beyond the scope of this book. However, once we have initialized a **regex** with our pattern for postal codes, we can apply it to each line of our file:

```
smatch matches;
if (regex_search(line, matches, pat))
        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**. Naturally, 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 for "string." Basically, an **smatch** is a vector of sub-matches of type **string**. The first element, here **matches[0]**, is the complete match. We can treat **matches[i]** as a string if **i<matches.size()**. So if – for a given regular expression – the maximum number of sub-patterns is **N**, we find **matches.size()==N+1**.

So, what is a sub-pattern? A good first answer is "Anything in parentheses in the pattern." Looking at **\w{2}\s*\d{5}(–\d{4})?**, we see the parentheses around the four-digit extension of the ZIP code. That's the only sub-pattern we see, so we guess (correctly) that **matches.size()==2**. We also guess that we can easily access those last four digits. For example:

```
for (string line; getline(in,line); ) {
        smatch matches;
        if (regex_search(line, matches, pat)) {
                cout << lineno << ": " << matches[0] << '\n';        // whole match
                if (1<matches.size() && matches[1].matched)
                        cout  << "\t: " << matches[1] << '\n';        // sub-match
        }
}
```

Strictly speaking, we didn't have to test **1<matches.size()** because we already had a good look at the pattern, but we felt like being paranoid (because we have been experimenting with a variety of patterns in **pat** and they didn't all have just one sub-pattern). We can ask if a sub-match succeeded by looking at its **matched** member, here **matches[1].matched**. In case you wonder: when **matches[i].matched** is **false**, the unmatched sub-pattern **matches[i]** prints as the empty string. Similarly, a sub-pattern that doesn't exist, such as **matches[17]** for the pattern above, is treated as an unmatched sub-pattern.

We tried this program with a file containing

```
address TX77845
ffff tx 77843 asasasaa
ggg TX3456–23456
howdy
zzz TX23456–3456sss ggg TX33456–1234
cvzcv TX77845–1234 sdsas
xxxTx77845xxx
TX12345–123456
```

and got the output

```
1: TX77845
2: tx 77843
5: TX23456–3456
        : –3456
6: TX77845–1234
        : –1234
7: Tx77845
8: TX12345–1234
        : –1234
```

Note that we

- Did not get fooled by the ill-formatted "postal code" on the line that starts with **ggg** (what's wrong with that one?)
- Only found the first postal code from the line with **zzz** (we only asked for one per line)

- Found the correct suffixes on lines 5 and 6
- Found the postal code "hidden" among the **xxx**s on line 7
- Found (unfortunately?) the postal code "hidden" in **TX12345–123456**

23.8 Regular expression syntax

We have seen a rather basic example of regular expression matching. Now is the time to consider regular expressions (in the form they are used in the **regex** library) a bit more systematically and completely.

Regular expressions ("regexps" or "regexs") is basically a little language for expressing patterns of characters. It is a powerful (expressive) and terse language, and as such it can be quite cryptic. After decades of use, there are many subtle features and several dialects. Here, we will just describe a (large and useful) subset of what appears to be the currently most widely used dialect (the PERL one). Should you need more to express what you need to say or to understand the regular expressions of others, go look on the web. Tutorials (of wildly differing quality) and specifications abound.

The library also supports the ECMAScript, POSIX, awk, grep, and egrep notations and a host of search options. This can be extremely useful, especially if you need to match some pattern specified in another language. You can look up those options if you feel the need to go beyond the basic facilities described here. However, remember that "using the most features" is not an aim of good programming. Whenever you can, take pity on the poor maintenance programmer (maybe yourself in a couple of months) who has to read and understand your code: write code that is not unnecessarily clever and avoid obscure features whenever you can.

23.8.1 Characters and special characters

A regular expression specifies a pattern that can be used to match characters from a string. By default, a character in a pattern matches itself in a string. For example, the regular expression (pattern) **"abc"** will match the **abc** in **Is there an abc here?**

The real power of regular expressions comes from "special characters" and character combinations that have special meanings in a pattern:

Characters with special meaning	
.	any single character (a "wildcard")
[	character class
{	count

Characters with special meaning (*continued*)	
(	begin grouping
)	end grouping
\	next character has a special meaning
*****	zero or more
+	one or more
?	optional (zero or one)
|	alternative (or)
^	start of line; negation
$	end of line

For example,

```
x.y
```

matches any three-character string starting with an **x** and ending with a **y**, such as **xxy**, **x3y**, and **xay**, but not **yxy**, **3xy**, and **xy**.

Note that **{ . . . }**, *****, **+**, and **?** are suffix operators. For example, **\d+** means "one or more decimal digits."

If you want to use one of the special characters in a pattern, you have to "escape it" using a backslash; for example, in a pattern **+** is the one-or-more operator, but **\+** is a plus sign.

23.8.2 Character classes

The most common combinations of characters are represented in a terse form as "special characters":

Special characters for character classes		
\d	a decimal digit	**[[:digit:]]**
\l	a lowercase character	**[[:lower:]]**
\s	a space (space, tab, etc.)	**[[:space:]]**
\u	an uppercase character	**[[:upper:]]**
\w	a letter (a–z or A–Z) or digit (0–9) or an underscore (_)	**[[:alnum:]]**
\D	not **\d**	**[^[:digit:]]**

Special characters for character classes (*continued*)		
\L	not \l	[^[:lower:]]
\S	not \s	[^[:space:]]
\U	not \u	[^[:upper:]]
\W	not \w	[^[:alnum:]]

Note that an uppercase special character means "not the lowercase version of that special character." In particular, \W means "not a letter" rather than "an uppercase letter."

The entries in the third column (e.g., [[:digit:]]) give an alternative syntax using a longer name.

Like the string and iostream libraries, the regex library can handle large character sets, such as Unicode. As with string and iostream, we just mention this so that you can look for help and more information should you need it. Dealing with Unicode text manipulation is beyond the scope of this book.

23.8.3 Repeats

Repeating patterns are specified by the suffix operators:

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,

```
Ax*
```

matches an A followed by zero or more xs, such as

```
A
Ax
Axx
Axxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

If you want at least one occurrence, use **+** rather than *****. For example,

```
Ax+
```

matches an **A** followed by one or more **x**s, such as

```
Ax
Axx
Axxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

but not

```
A
```

The common case of zero or one occurrence ("optional") is represented by a question mark. For example,

```
\d–?\d
```

matches the two digits with an optional dash between them, such as

```
1–2
12
```

but not

```
1––2
```

To specify a specific number of occurrences or a specific range of occurrences, use curly braces. For example,

```
\w{2}–\d{4,5}
```

matches exactly two letters and a dash (**–**) followed by four or five digits, such as

```
Ab–1234
XX–54321
22–54321
```

but not

```
Ab–123
?b–1234
```

Yes, digits are `\w` characters.

23.8.4 Grouping

To specify a regular expression as a sub-pattern, you group it using parentheses. For example:

```
(\d*:)
```

This defines a sub-pattern of zero or more digits followed by a colon. A group can be used as part of a more elaborate pattern. For example:

```
(\d*:)?(\d+)
```

This specifies an optional and possibly empty sequence of digits followed by a colon followed by a sequence of one or more digits. No wonder people invented a terse and precise way of saying such things!

23.8.5 Alternation

The "or" character (`|`) specifies an alternative. For example:

```
Subject: (FW:|Re:)?(.*)
```

This recognizes an email subject line with an optional `FW:` or `Re:` followed by zero or more characters. For example:

```
Subject: FW: Hello, world!
Subject: Re:
Subject: Norwegian Blue
```

but not

```
SUBJECT: Re: Parrots
Subject  FW: No subject!
```

An empty alternative is not allowed:

```
(|def)        // error
```

However, we can specify several alternatives at once:

```
(bs|Bs|bS|BS)
```

23.8.6 Character sets and ranges

The special characters provide a shorthand for the most common classes of characters: digits (\d); letters, digits, and underscore (\w); etc. (§23.7.2). However, it is easy and often useful to define our own. For example:

[\w @]	a word character, a space, or an @
[a–z]	the lowercase characters from a to z
[a–zA–Z]	upper- or lowercase characters from a to z
[Pp]	an upper- or lowercase P
[\w\–]	a word character or a dash (plain – means "range")
[asdfghjkl;']	the characters on the middle line of a U.S. QWERTY keyboard
[.]	a dot
[.[{(*+?^$]	a character with special meaning in a regular expression

In a character class specification, a – (dash) is used to specify a range, such as [1–3] (1, 2, or 3) and [w–z] (w, x, y, or z). Please use such ranges carefully: not every language has the same letters and not every letter encoding has the same ordering. If you feel the need for any range that isn't a sub-range of the most common letters and digits of the English alphabet, consult the documentation.

Note that we can use the special characters, such as \w (meaning "any word character"), within a character class specification. So, how do we get a backslash (\) into a character class? As usual, we "escape it" with a backslash: \\.

When the first character of a character class specification is ^, that ^ means "negation." For example:

[^aeiouy]	not an English vowel
[^\d]	not a digit
[^aeiouy]	a space, a ^, or an English vowel

In the last regular expression, the ^ wasn't the first character after the [, so it was just a character, not a negation operator. Regular expressions can be subtle.

An implementation of **regex** also supplies a set of named character classes for use in matching. For example, if you want to match any alphanumeric character (that is, a letter or a digit: a–z or A–Z or 0–9), you can do it by the regular expression [[:alnum:]]. Here, alnum is the name of a set a characters (the set of alphanumeric characters). A pattern for a nonempty quoted string of alphanumeric characters would be "[[:alnum:]]+". To put that regular expression into an ordinary string literal, we have to escape the quotes:

```
string s {"\" [[:alnum:]]+\""};
```

Furthermore, to put that string literal into a **regex**, we must escape the backslashes:

```
regex s {"\\\" [[:alnum:]]+\\\""};
```

Using a raw string literal is simpler:

```
regex s2 {R"(" [[:alnum:]]+")"};
```

Prefer raw string literals for patterns containing backslashes or double quotes. That turns out to be most patterns in many applications.

Using regular expressions leads to a lot of notational conventions. Anyway, here is a list of the standard character classes:

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

An implementation of **regex** may provide more character classes, but if you decide to use a named class not listed here, be sure to check if it is portable enough for your intended use.

23.8.7 Regular expression errors

What happens if we specify an illegal regular expression? Consider:

```
regex pat1 {"(|ghi)"};        // missing alternative
regex pat2 {"[c–a]"};         // not a range
```

When we assign a pattern to a regex, the pattern is checked, and if the regular expression matcher can't use it for matching because it's illegal or too complicated, a **bad_expression** exception is thrown.

Here is a little program that's useful for getting a feel for regular expression matching:

```
#include <regex>
#include <iostream>
#include <string>
#include <fstream>
#include<sstream>
using namespace std;

// accept a pattern and a set of lines from input
// check the pattern and search for lines with that pattern

int main()
{
        regex pattern;

        string pat;
        cout << "enter pattern: ";
        getline(cin,pat);          // read pattern

        try {
                pattern = pat;     // this checks pat
                cout << "pattern: " << pat << '\n';
        }
        catch (bad_expression) {
                cout << pat << " is not a valid regular expression\n";
                exit(1);
        }

        cout << "now enter lines:\n";
        int lineno = 0;

        for (string line; getline(cin,line); ) {
                ++lineno;
                smatch matches;
                if (regex_search(line, matches, pattern)) {
                        cout << "line " << lineno << ": " << line << '\n';
                        for (int i = 0; i<matches.size(); ++i)
                                cout << "\tmatches[" << i << "]: "
                                        << matches[i] << '\n';
                }
                else
                        cout << "didn't match\n";
        }
}
```

TRY THIS

Get the program to run and use it to try out some patterns, such as **abc**, **x.*x**, **(.*)**, **\([^)]*\)**, and **\w+ \w+(Jr\.)?**.

23.9 Matching with regular expressions

There are two basic uses of regular expressions:

- *Searching* for a string that matches a regular expression in an (arbitrarily long) stream of data – **regex_search()** looks for its pattern as a substring in the stream.
- *Matching* a regular expression against a string (of known size) – **regex_match()** looks for a complete match of its pattern and the string.

The search for ZIP codes in §23.6 was an example of searching. Here, we will examine an example of matching. Consider extracting data from a table like this:

KLASSE	ANTAL DRENGE	ANTAL PIGER	ELEVER IALT
0A	12	11	23
1A	7	8	15
1B	4	11	15
2A	10	13	23
3A	10	12	22
4A	7	7	14
4B	10	5	15
5A	19	8	27

KLASSE	ANTAL DRENGE	ANTAL PIGER	ELEVER IALT
6A	10	9	19
6B	9	10	19
7A	7	19	26
7G	3	5	8
7I	7	3	10
8A	10	16	26
9A	12	15	27
0MO	3	2	5
0P1	1	1	2
0P2	0	5	5
10B	4	4	8
10CE	0	1	1
1MO	8	5	13
2CE	8	5	13
3DCE	3	3	6
4MO	4	1	5
6CE	3	4	7
8CE	4	4	8
9CE	4	9	13
REST	5	6	11
Alle klasser	184	202	386

This table (of the number of students in Bjarne Stroustrup's old primary school in 2007) was extracted from a context (a web page) where it looks nice and is fairly typical of the kind of data we need to analyze:

- It has numeric data fields.
- It has character fields with strings meaningful only to people who understand the context of the table. (Here, that point is emphasized by the use of Danish.)
- The character strings include spaces.
- The "fields" of this data are separated by a "separation indicator," which in this case is a tab character.

We chose this table to be "fairly typical" and "not too difficult," but note one subtlety we must face: we can't actually see the difference between spaces and tab characters; we have to leave that problem to our code.

We will illustrate the use of regular expressions to

- Verify that this table is properly laid out (i.e., every row has the right number of fields)
- Verify that the numbers add up (the last line claims to be the sum of the columns above)

If we can do that, we can do just about anything! For example, we could make a new table where the rows with the same initial digit (indicating the year: first grades start with 1) are merged or see if the number of students is increasing or decreasing over the years in question (see exercises 10–11).

To analyze the table, we need two patterns: one for the header line and one for the rest of the lines:

```
regex header {R"(^[\w ]+(        [\w ]+)*$)"};
regex row {R"(^[\w ]+(           \d+)(  \d+)(   \d+)$)"};
```

Please remember that we praised the regular expression syntax for terseness and utility; we did not praise it for ease of comprehension by novices. In fact, regular expressions have a well-earned reputation for being a "write-only language." Let us start with the header. Since it does not contain any numeric data, we could just have thrown away that first line, but – to get some practice – let us parse it. It consists of four "word fields" ("alphanumeric fields") separated by tabs. These fields can contain spaces, so we cannot simply use plain **\w** to specify its characters. Instead, we use **[\w]**, that is, a word character (letter, digit, or underscore) or a space. One or more of those is written **[\w]+**. We want the first of those at the start of a line, so we get **^[\w]+**. The "hat" (**^**) means "start of line." Each of the rest of the fields can be expressed as a tab followed by some words: **([\w]+)**. Now we take an arbitrary number of those followed by an end of line: **([\w]+)*$**. The dollar sign (**$**) means "end of line."

Note how we can't see that the tab characters are really tabs, but in this case they expand in the typesetting to reveal themselves.

Now for the more interesting part of the exercise: the pattern for the lines from which we want to extract the numeric data. The first field is as before: **^[\w]+**. It is followed by exactly three numeric fields, each preceded by a tab, **(\d+)**, so that we get

```
^[\w ]+(        \d+)(        \d+)(        \d+)$
```

which, after putting it into a raw string literal, is

```
R"(^[\w ]+(       \d+)(       \d+)(       \d+)$)"
```

Now all we have to do is to use those patterns. First we will just validate the table layout:

```
int main()
{
        ifstream in {"table.txt"};      // input file
        if (!in) error("no input file\n");

        string line;                    // input buffer
        int lineno = 0;

        regex header {R"(^[\w ]+(       [\w ]+)*$)"};               // header line
        regex row {R"(^[\w ]+(       \d+)(  \d+)(   \d+)$)"};       // data line

        if (getline(in,line)) {         // check header line
                smatch matches;
                if (!regex_match(line, matches, header))
                        error("no header");
        }
        while (getline(in,line)) {      // check data line
                ++lineno;
                smatch matches;
                if (!regex_match(line, matches, row))
                        error("bad line",to_string(lineno));
        }
}
```

For brevity, we left out the **#include**s. We are checking all the characters on each line, so we use **regex_match()** rather than **regex_search()**. The difference between those two is exactly that **regex_match()** must match every character of its input to succeed, whereas **regex_search()** looks at the input trying to find a substring that matches. Mistakenly typing **regex_match()** when you meant **regex_search()** (or vice versa) can be a most frustrating bug to find. However, both of those functions use their "matches" argument identically.

We can now proceed to verify the data in that table. We keep a sum of the number of pupils in the boys ("drenge") and girls ("piger") columns. For each row, we check that last field ("ELEVER IALT") really is the sum of the first two fields. The last row ("Alle klasser") purports to be the sum of the columns above. To check that, we modify **row** to make the text field a sub-match so that we can recognize "Alle klasser":

```
int main()
{
        ifstream in {"table.txt"};          // input file
        if (!in) error("no input file");

        string line;                        // input buffer
        int lineno = 0;

        regex header {R"(^[\w ]+(        [\w ]+)*$)"};          // header line
        regex row {R"(^[\w ]+(        \d+)(  \d+)(   \d+)$)"};  // data line

        if (getline(in,line)) {             // check header line
                smatch matches;
                if (regex_match(line, matches, header)) {
                        error("no header");
                }
        }

        // column totals:
        int boys = 0;
        int girls = 0;

        while (getline(in,line)) {
                ++lineno;
                smatch matches;
                if (!regex_match(line, matches, row))
                        cerr << "bad line: " << lineno << '\n';

                if (in.eof()) cout << "at eof\n";

                // check row:
                int curr_boy = from_string<int>(matches[2]);
                int curr_girl = from_string<int>(matches[3]);
                int curr_total = from_string<int>(matches[4]);
                if (curr_boy+curr_girl != curr_total)  error("bad row sum \n");

                if (matches[1]=="Alle klasser") {          // last line
                        if (curr_boy != boys) error("boys don't add up\n");
                        if (curr_girl != girls) error("girls don't add up\n");
```

```
                    if (!(in>>ws).eof()) error("characters after total line");
                    return 0;
               }

               // update totals:
               boys += curr_boy;
               girls += curr_girl;
          }

          error("didn't find total line");
     }
```

The last row is semantically different from the other rows – it is their sum. We recognize it by its label ("Alle klasser"). We decided to accept no more non-whitespace characters after that last one (using the technique from **to<>()**; §23.2) and to give an error if we did not find it.

We used **from_string()** from §23.2 to extract an integer value from the data fields. We had already checked that those fields consisted exclusively of digits so we did not have to check that the **string**-to-**int** conversion succeeded.

23.10 References

Regular expressions are a popular and useful tool. They are available in many programming languages and in many formats. They are supported by an elegant theory based on formal languages and by an efficient implementation technique based on state machines. The full generality of regular expressions, their theory, their implementation, and the use of state machines in general are beyond the scope of this book. However, because these topics are rather standard in computer science curricula and because regular expressions are so popular, it is not hard to find more information (should you need it or just be interested).

For more information, see:

Aho, Alfred V., Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools, Second Edition* (usually called "The Dragon Book"). Addison-Wesley, 2007. ISBN 0321547985.

Cox, Russ. "Regular Expression Matching Can Be Simple and Fast (but Is Slow in Java, Perl, PHP, Python, Ruby, . . .)." http://swtch.com/~rsc/regexp/regexp1.html.

Maddock, J. boost::regex documentation. www.boost.org/.

Schwartz, Randal L., Tom Phoenix, and Brian D. Foy. *Learning Perl, Fourth Edition*. O'Reilly, 2005. ISBN 0596101058.

Drill

1. Find out if **regex** is shipped as part of your standard library. Hint: Try **std::regex** and **tr1::regex**.
2. Get the little program from §23.7 to work; that may involve figuring out how to set the project and/or command-line options to link to the **regex** library and use the **regex** headers.
3. Use the program from drill 2 to test the patterns from §23.7.

Review

1. Where do we find "text"?
2. What are the standard library facilities most frequently useful for text analysis?
3. Does **insert()** add before or after its position (or iterator)?
4. What is Unicode?
5. How do you convert to and from a **string** representation (to and from some other type)?
6. What is the difference between **cin>>s** and **getline(cin,s)** assuming **s** is a **string**?
7. List the standard streams.
8. What is the key of a **map**? Give examples of useful key types.
9. How do you iterate over the elements of a **map**?
10. What is the difference between a **map** and a **multimap**? Which useful **map** operation is missing for **multimap**, and why?
11. What operations are required for a forward iterator?
12. What is the difference between an empty field and a nonexistent field? Give two examples.
13. Why do we need an escape character to express regular expressions?
14. How do you get a regular expression into a **regex** variable?
15. What does **\w+\s\d{4}** match? Give three examples. What string literal would you use to initialize a **regex** variable with that pattern?
16. How (in a program) do you find out if a string is a valid regular expression?
17. What does **regex_search()** do?
18. What does **regex_match()** do?
19. How do you represent the character dot (**.**) in a regular expression?
20. How do you represent the notion of "at least three" in a regular expression?
21. Is **7** a **\w** character? Is _ (underscore)?
22. What is the notation for an uppercase character?
23. How do you specify your own character set?
24. How do you extract the value of an integer field?

25. How do you represent a floating-point number as a regular expression?
26. How do you extract a floating-point value from a match?
27. What is a sub-match? How do you access one?

Terms

match	**regex_match()**	search
multimap	**regex_search()**	**smatch**
pattern	regular expression	sub-pattern

Exercises

1. Get the email file example to run; test it using a larger file of your own creation. Be sure to include messages that are likely to trigger errors, such as messages with two address lines, several messages with the same address and/or same subject, and empty messages. Also test the program with something that simply isn't a message according to that program's specification, such as a large file containing no **----** lines.
2. Add a **multimap** and have it hold subjects. Let the program take an input string from the keyboard and print out every message with that string as its subject.
3. Modify the email example from §23.4 to use regular expressions to find the subject and sender.
4. Find a real email message file (containing real email messages) and modify the email example to extract subject lines from sender names taken as input from the user.
5. Find a large email message file (thousands of messages) and then time it as written with a **multimap** and with that **multimap** replaced by an **unordered_multimap**. Note that our application does not take advantage of the ordering of the **multimap**.
6. Write a program that finds dates in a text file. Write out each line containing at least one date in the format **line–number: line**. Start with a regular expression for a simple format, e.g., 12/24/2000, and test the program with that. Then, add more formats.
7. Write a program (similar to the one in the previous exercise) that finds credit card numbers in a file. Do a bit of research to find out what credit card formats are really used.
8. Modify the program from §23.8.7 so that it takes as inputs a pattern and a file name. Its output should be the numbered lines (**line–number: line**) that contain a match of the pattern. If no matches are found, no output should be produced.

9. Using **eof()** (§B.7.2), it is possible to determine which line of a table is the last. Use that to (try to) simplify the table-checking program from §23.9. Be sure to test your program with files that end with empty lines after the table and with files that don't end with a newline at all.
10. Modify the table-checking program from §23.9 to write a new table where the rows with the same initial digit (indicating the year: first grades start with 1) are merged.
11. Modify the table-checking program from §23.9 to see if the number of students is increasing or decreasing over the years in question.
12. Write a program, based on the program that finds lines containing dates (exercise 6), that finds all dates and reformats them to the ISO yyyy-mm-dd format. The program should take an input file and produce an output file that is identical to the input file except for the changed date formatting.
13. Does dot (**.**) match **'\n'**? Write a program to find out.
14. Write a program that, like the one in §23.8.7, can be used to experiment with pattern matching by typing in a pattern. However, have it read a file into memory (representing a line break with the newline character, **'\n'**), so that you can experiment with patterns spanning line breaks. Test it and document a dozen test patterns.
15. Describe a pattern that cannot be expressed as a regular expression.
16. For experts only: Prove that the pattern found in the previous exercise really isn't a regular expression.

Postscript

It is easy to get trapped into the view that computers and computation are all about numbers, that computing is a form of math. Obviously, it is not. Just look at your computer screen; it is full of text and pictures. Maybe it's busy playing music. For every application, it is important to use proper tools. In the context of C++, that means using appropriate libraries. For text manipulation, the regular expression library is often a key tool – and don't forget the **map**s and the standard algorithms.