



3

Design, Use, and Analysis Extending the Foundation

Strings, Streams, and Operators

9

He was a poet and hated the approximate.

Rainer Maria Rilke
The Journal of My Other Self

Computer programs require precision even when abstraction is required
to make them intelligible.

J.A. Zimmer
Abstraction for Programmers

Abstraction ...is seductive; forming generic abstract types can lead into confusing excess

Marian Petre
Psychology of Programming, 112.

In 1936 Alan Turing, a British mathematician, published a famous paper titled “On Computable Numbers, with an Application to the Entscheidungsproblem.” This paper helped lay the foundation for much of the work done in theoretical computer science, even though computers did not exist when the paper was written.¹ Turing invented a model of a computer, called a **Turing machine**, and he used this model to develop ideas and proofs about what kinds of numbers could be computed. His invention was an abstraction, not a real machine, but it provided a framework for reasoning about computers. The **Church–Turing thesis** says that, from a theoretical standpoint, all computers have the same power. This is commonly accepted; the most powerful computers in the world compute the same things as Turing’s abstract machine could compute. Of course some computers are faster than others, and computers continue to get faster every year,² but the kinds of things that can be computed have not changed.

How can we define abstraction in programming? The American Heritage Dictionary defines it as “the act or process of separating the inherent qualities or properties of something from the actual physical object or concept to which they belong.” The general user’s view of a computer is an abstraction of what really goes on behind the scenes. You do not need to know how to program a pull-down menu or a tracking mouse to use these tools. You do not need to know how numbers are represented in computer memory to write programs that manipulate numeric expressions. In some cases such missing knowledge is useful, because it can free you from worrying unnecessarily about issues that aren’t relevant to programming at a high level.

Abstraction is a cornerstone of all computer science and certainly of our study of programming. The capability that modern programming languages and techniques pro-

¹At least, computers as we know them had not yet been invented. Several kinds of calculating machines had been proposed or manufactured, but no general-purpose computer had been built.

²No matter when you read this sentence, it is likely to be true.

vide us to avoid dealing with details permits more complex and larger programs to be written than could be written with assembly language, for example.

In this chapter we'll discuss characters, strings, files, and streams. These form an abstraction hierarchy with characters at the lowest level and streams at the highest level. A character is a symbol such as 'a'. Strings and files are both constructed from characters. We'll see that streams can be constructed from strings as well as from files. Although a character lies at the lowest level, we'll see that characters are also abstractions. We'll discuss programming tools that help in using and combining these abstractions.

9.1 Characters: Building Blocks for Strings

From the beginning of our study of C++ we have worked with the class `string`. Although we haven't worried about how the `string` class is implemented or about the individual characters from which strings are built, we have used the `string` class extensively in many programs. We have treated strings as abstractions—we understand strings by their use and behavior rather than by their construction or implementation. If we understand the `string` member functions (such as `length`, `substr`, `operator ==`, and `operator <<`), we do not need to understand the details and idiosyncrasies of the implementation. However, some programs manipulate the individual characters used to build strings, so we'll need to expand our understanding of characters.

9.1.1 The Type `char` as an Abstraction

We have discussed strings as sequences of characters but have not included detailed discussions of how a character is implemented in C++. The type `char` is used for characters in C++.³

A `char` variable stores legal character values. The range of legal values depends on the computer system being used and even the country in which the system is used. The range of legal characters that is supported in a computing system is called the **character set**. The most commonly used set is the ASCII set (pronounced “askee,” an acronym for American Standard Code for Information Interchange); all programs in this book are run on a system with this character set. An emerging standard set is called Unicode, which supports international characters, such as ä, that are not part of the ASCII set. Chinese, Japanese, Arabic, and Cyrillic character sets may also be represented using Unicode. You must try to isolate your programs as much as possible from the particular character set being used in the program's development. This will help ensure that the program is **portable**—that is, useful in other computing environments than the one in which it was developed.

The type `char` is the smallest built-in type. A `char` variable uses less (actually, no more) memory than any other type of variable. A `char` literal is identified by using

³Some people pronounce `char` as “care,” short for “character.” Others pronounce it “char” as in “charcoal.” A third common pronunciation is “car” (rhymes with “star”). I don't like the “charcoal” pronunciation and use the pronunciation that has character.

single quotes, as shown in the first two of the following examples:

```
char letter = 'a';
char digit = '9';
string word = "alphabetic";
```

Note that `string` literals use double quotes, which are different from two single quotes.

As an abstraction, a `char` is very different from an `int`. Unfortunately, in almost all cases a `char` can be treated as an `int` in C++ programs. This similarity has the potential to be confusing. From a programmer's view, a `char` is distinguished from an `int` by the way it is printed and, perhaps, by the amount of computer memory it uses. The relationship between `char` and `int` values is determined by the character set being used. For ASCII characters this relationship is given in Table F.3 in Howto F.



Program 9.1 shows how the type `char` is very similar to the type `int` but prints differently. The `char` variable `k` is incremented just as an `int` is incremented, but, as the output shows, characters appear on the screen differently than integers.

The output of Program 9.1 shows that capital letters come before lower-case letters when the ASCII character set is used. Notice that the characters representing the digits '0' through '9' are contiguous and come before any alphabetic character.

Program 9.1 charlist.cpp

```
#include <iostream>
using namespace std;

// illustrates use of char as an integral type

int main()
{
    char first,last;
    cout << "enter first and last characters" << endl;
    cout << "with NO SPACE separating them: ";
    cin >> first >> last;

    cout << first;          // print first char (fencepost problem)
    char k;
    for(k=first+1; k <= last; k++)
    {   cout << " " << k;
        }
    cout << endl;
    return 0;
}
```

charlist.cpp

```

O U T P U T

prompt> charlist
enter first and last characters
with NO SPACE separating them: AZ
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
prompt> charlist
enter first and last characters
with NO SPACE separating them: 2B
2 3 4 5 6 7 8 9 : ; < = > ? @ A B
prompt> charlist
enter first and last characters
with NO SPACE separating them: Zf
Z [ \ ] \^{} _ ` a b c d e f
prompt> charlist
enter first and last characters
with NO SPACE separating them: &3
& ' ( ) * + , - . / 0 1 2 3

```

If we change the output statement to cast the character to an `int`,

```
cout << " " << int(k);
```

the program will display the internal numeric representation of each `char` rather than its symbolic character representation.

```

O U T P U T

prompt> charlist
enter first and last characters
with NO SPACE separating them: AM
65 66 67 68 69 70 71 72 73 74 75 76 77

```

Using the cast makes it more difficult to verify that the output is correct, because the symbolic form of each character isn't used. In general, there isn't any reason to be concerned with what the numerical representation of each character is, because C++ provides many mechanisms that allow programs to use the type `char` abstractly without regard for the underlying character set. You can make the following assumptions about character codes on almost every system you'll use.

1. The digit characters '0' through '9' (ASCII values 48 through 57) are consecutive with no intervening characters.

Table 9.1 Some functions in `<cctype>`

function prototype	returns true when
<code>int isalpha(int c)</code>	c is alphabetic (upper or lower case)
<code>int isalnum(int c)</code>	c is alphabetic or a digit
<code>int islower(int c)</code>	c is a lowercase letter
<code>int isdigit(int c)</code>	c is a digit character '0'-'9'

- The lower-case characters 'a' through 'z' (ASCII 97 through 122) are consecutive, and the upper-case characters 'A' through 'Z' (ASCII 65 through 90) are consecutive.

These assumptions are true for the ASCII character set and the Unicode character set, but not necessarily for all character sets.⁴ In almost all programming environments you'll use either ASCII or Unicode. In the next section we'll study utility functions that help in writing portable programs.

9.1.2 The Library `<cctype>`

To be portable, your code must not rely on a specific character set. Just as the functions in the math library `<cmath>` make writing mathematical and scientific programs easier, a library of character functions helps in writing portable character-manipulating programs. This character library is accessible by using `#include <cctype>` or on some systems using `<ctype.h>`. The prototypes for some of the functions in `<cctype>` are given in Table 9.1; prototypes for all the functions are found in Table F.2 in Howto F.



Although the formal parameter for each function is an `int`, these functions are intended to work with `char` arguments. Thus `isalpha('9')` evaluates to zero (false), because '9' is not an alphabetic character. In an ASCII environment `isdigit(57)` evaluates to nonzero (true), because 57 is the ASCII value for the character '9'. You should avoid using these functions in such a manner; treat characters as symbolic abstractions.

Program Tip 9.1: The functions in `<cctype>` return an `int` value rather than a `bool` value; but treat the value as a `bool`. In particular, there is no guarantee that the return value will be 1 for true (although the return value will always be 0 for false). This means you should write `if (isdigit(ch))` rather than `if (isdigit(ch) == 1)` in your code.

⁴The C++ standard requires that '0' through '9' be consecutive, but in the EBCDIC character set the letters 'a' through 'z' and 'A' through 'Z' are not consecutive.

To write portable programs, use the functions in `<cctype>` rather than writing equivalent functions. For example, if the ASCII character set is used, the following function could serve as an implementation of `tolower`:

```
int tolower(int c)
// postcondition: returns lowercase equivalent of c
//               if c isn't upper case, returns c unchanged
{
    if ('A' <= c && c <= 'Z')    // c is uppercase
    {   return c + 32;
    }
    return c;
}
```

This function works only when the ASCII character set is used, and it relies on two properties of the character set:

- The uppercase letters occur in order with no intervening characters.
- The difference between a lower-case letter and its corresponding upper-case equivalent is always 32.

You can isolate some dependencies on ASCII by subtracting characters:

```
int tolower(int c)
// postcondition: returns lowercase equivalent of c
//               if c isn't upper case, returns c unchanged
{
    if ('A' <= c && c <= 'Z')    // c is uppercase
    {   return c + ('a' - 'A');
    }
    return c;
}
```

The correctness of this code depends only on a character set in which 'a' through 'z' and 'A' through 'Z' are consecutive ranges. Since `char` values can be manipulated as `int` values, you can subtract one character from another, yielding an `int` value. However, although you can multiply 'a' * 'b', the result doesn't make sense; using ASCII, the result is `97*98 == 9506`, which is not a legal character value. Although you can use `char` variables as integers, you should restrict arithmetic operations of characters to the following:

1. Adding an integer to a character—for example, `'0' + 2 == '2'`
2. Subtracting an integer from a character—for example, `'9' - 3 == '6'` and `'C' - 2 == 'A'`
3. Subtracting two characters—for example, `'8' - '0' == 8` and `'Z' - 'A' == 25`

You can use a `char` value in a `switch` statement, because `char` values can be used as integers. You can also compare two `char` values using the relational operators `<`, `<=`, `>`, `>=`. Character comparisons are based on the value of the underlying character set, which will always reflect lexicographic (dictionary) order.

Now that we have covered the lowest level of the character-string-file-stream hierarchy, we'll see how characters are used to build strings and files. We'll investigate strings first.

9.1.3 Strings as `char` Sequences

The class `string`, accessible using the header file `<string>`,⁵ is an abstraction that represents sequences of characters. However, we haven't yet studied any mechanism for extracting individual characters from a `string`. Although the `substr` member function extracts strings and `operator +` concatenates strings, until now we haven't been able to alter the individual characters of a `string`.

Basically, a `string` acts like a vector of characters. The characters are indexed from 0 to `s.length() - 1`. For example, if `str` represents the string "computer", then 'c' has index 0 and 'r' has index 7. Individual characters in a `string` are accessed using the **indexing** operator `[]`, shown in *spreader.cpp*, Program 9.2, to print a string with spaces between each character.

Program 9.2 *spreader.cpp*

```
#include <iostream>
#include <string>
using namespace std;
#include "prompt.h"

// spread a string by inserting spaces between characters

int main()
{
    string s = PromptString("enter a string: ");
    int k, limit = s.length(); // # of chars in s
    if (limit > 0) // at least one character
    {
        cout << s[0]; // first character, fencepost problem
        for(k=1; k < limit; k++) // then loop over the rest
        {
            cout << " " << s[k];
        }
        cout << endl;
    }
    return 0;
}
```

spreader.cpp

⁵The C++ standard `string` class is accessible using the header file `<string>`. You may be using `"tstring.h"` or `"apstring.h"` rather than the standard header file. Each of these implementations work with the programs in this book.

```

O U T P U T

prompt> spreader
enter a string: longwinded
l o n g w i n d e d
prompt> spreader
enter a string: !*#$$%
! * # $ %
```

Because the expression `s[k]` is used for output, and because the compiler can determine that the expression `s[k]` is a `char`, the symbolic form of each character is printed; that is, an `'o'` instead of 111 (the ASCII value of `'o'`). The indexing operator can also be used to change an individual character in a string. For example, the following sequence of statements would cause `taste` to be displayed:

```
string s = "paste";
s[0] = 't';
cout << s << endl;
```

A program that uses the `[]` operator with an index that is **out of range** (i.e., less than 0 or greater than or equal to the number of characters in a string) will cause undefined behavior if the standard string class is used because the standard class does not check for illegal indexes.⁶

Program Tip 9.2: Out-of-range string indexes can cause indeterminate and hard-to-find errors. The errors are indeterminate because the program may behave differently each time it is run, depending on what values are in memory. An out-of-range index will either read from or write to a memory location that is not part of the string. Such memory accesses invariably lead to errors.

⁶The implementations of `string` in `"tstring.h"` or `"apstring.h"` do check for illegal indexes. These implementations will generate an error message when a program indexes a string with an out-of-range value.

John von Neumann (1903–1957)

John von Neumann was a genius in many fields. He founded the field of game theory with his book *Theory of Games and Economic Behavior* (cowritten with Oskar Morgenstern). He helped develop the atomic bomb as part of the Manhattan Project. Almost all computers in use today are based on the von Neumann model of stored programs and use an architecture that he helped develop in the early years of computing.



In 1944, von Neumann was working with the ENIAC (Electronic Numerical Integrator and Computer), a machine whose wires had to be physically rearranged to run a different program. The idea of storing a program in the computer, just as data are stored, is generally credited to von Neumann (although there has been a history of sometimes rancorous dispute; see [Gol93, Mac92]).

Hans Bethe, a Nobel Prize–winning physicist, graded academic seminars on a scale of one to ten:

Grade one was something my mother could understand. Grade two my wife could understand. Grade seven was something I could understand. Grade eight was something only the speaker and Johnny von Neumann could understand. Grade nine was something Johnny could understand, but the speaker didn't. Grade ten was something even Johnny could not yet understand, but there was little of that.

Von Neumann's powers of memory and calculation were prodigious, as were his contributions to so many fields. For a full account of von Neumann's life see [Mac92].

Pause to Reflect



9.1 The following function is intended to return the decimal equivalent of a digit character; for example, for '0' it should return 0, and for '3' it should return 3.

```
int todigit(int c)
// pre: c is a digit character: '0', '1', ..., '9'
// post: returns digit equivalent,
//       e.g., 3 for '3'
{
    if (isdigit(c))
    { return c - '0';
    }
}
```

This function does return the correct values for all digit characters. The function is not robust, because it may cause programs to crash if the precondition isn't true. How would you make it more robust?

- 9.2** The underlying numeric value of a character (in ASCII and other character sets) reflects lexicographic order. For example, 'C' < 'a', since upper-case letters precede lower-case letters in the ASCII ordering. Why does this help to explain why "Zebra" < "aardvark" but "aardvark" < "yak"?
- 9.3** Explain why the statement `cout << 'a' + 3 << endl` generates the integer 100 as output. Why does the statement `cout << char('a' + 3) << endl` generate the character 'd'?
- 9.4** If the ASCII set is used, what are the values of `isctrnl('\t')`, `isspace('\t')`, and `islower('\t')`?
- 9.5** Write a function `isvowel` that returns true when its parameter is a vowel: 'a', 'e', 'i', 'o', or 'u' (or the upper-case equivalent). What is an easy way of writing `isconsonant` (assuming `isvowel` exists)?
- 9.6** Write a boolean-valued function `IsPalindrome` that returns true when its string parameter is a palindrome and false otherwise. A **palindrome** is a word that reads the same backwards as forwards, such as "racecar," "mom," and "amanaplanacanalpanama" (which is "A man, a plan, a canal—Panama!" with no spaces, capitals, or punctuation).
For a challenge, make the function ignore spaces and punctuation so that "A man, a plan, a canal — Panama!!" is recognized as a palindrome.
- 9.7** Write the body of the following function `MakeLower` so that all upper-case letters in `s` are converted to lower case. Why is `s` a reference parameter?

```
void MakeLower(string & s)
// post: all letters in s are lower case
```

- 9.8** There are several functions in the library "strutils.h" (see *strutils.h*, Program G.8 in Howto G.) for converting strings to numbers: `atoi` converts a string to an int and `atof` converts a string to a double. (The "a" is for "alphabetic"; "atoi" is pronounced "a-to-i.")

Write a function with prototype `int atoi(string s)` that converts a string to its decimal equivalent; for example, `atoi("1234")` evaluates to 1234, and `atoi("-52")` evaluates to -52.



9.2 Streams and Files as Lines and Characters

A `string` variable is a sequence of characters, but we manipulate strings abstractly without knowing the details of how the characters are stored or represented. When

information is hidden in this way, and a type is used independently of the underlying representation of the data, the type is sometimes called an **abstract data type**, or **ADT**. The data type is abstract because knowledge of its underlying implementation is not necessary to use it. You probably don't know how individual 0s and 1s are stored to represent `int` and `double` values, but you can still write programs that use these numeric types.

In this section we'll see that a stream is also an abstract data type. Until now we have viewed a stream as a sequence of words or numbers. We extract words or numbers from a stream using `>>` and insert onto a stream using `<<`. We have developed programs using the standard streams `cin` and `cout`, as well as streams bound to files using the classes `ifstream` and `ofstream`. In this section we'll study functions that let us view streams as a sequence of lines rather than words and numbers. Other functions let us view streams as sequences of characters; different views are useful in different settings. We'll see some applications that are most easily implemented when streams are viewed as sequences of lines and others where a sequence of characters is a better choice.

9.2.1 Input Using `getline()`

Input operations on strings using `>>` result in word-at-a-time input, where words are treated as any sequence of non-white space characters. In some applications other methods of input are needed. In particular, an `ifstream` variable bound to a file may require line-oriented input. Consider, for example, processing a file in the following format, where an artist/group name is followed on the next line by the title of a compact disc (CD) by the artist:

```
Spin Doctors
Pocket Full of Kryptonite
The Beatles
Sergeant Pepper's Lonely Hearts Club Band
Strauss
Also Sprach Zarathustra
The Grateful Dead
American Beauty
```

There is no way to read all the words on one line of a file using the stream-processing tools currently at our disposal. Since many text files are arranged as a sequence of lines rather than white space-delimited words, we need a method for reading input other than the extraction operator `>>`. The function `getline` allows an entire line of input to be read at once. When we view a stream as line-oriented rather than word-oriented, we need to be able to include white space as part of the line read from a stream.

If the line `cin >> s` in *spreader.cpp*, Program 9.2, is replaced with `getline(cin, s)`, the user can enter a string with spaces in it:

```
O U T P U T
```

```
prompt> spreader
enter a string: Green Eggs and Ham
G r e e n   E g g s   a n d   H a m
```

In the original program the only word read by the program is *Green*, because the space between “Green” and “Eggs” terminates the extraction operation when >> is used. The characters “Eggs and Ham” will not be processed but will remain on the input stream.

The function `getline` is used in Program 9.3 to count the total number of lines in a file. This gives a better count of the number of characters in a file too, because a line can contain white space characters that would not be read if >> were used.

Program 9.3 `filelines.cpp`

```
#include <iostream>
#include <fstream>
#include <cstdlib>
#include <string>
using namespace std;
#include "prompt.h"

// count # of lines in input file

int main()
{
    ifstream input;
    string s; // line entered by user
    long numLines = 0;
    long numChars = 0;
    string filename = PromptString("enter name of input file: ");
    input.open(filename.c_str());

    if (input.fail() )
    {
        cout << "could not open file " << filename << endl;
        exit(1);
    }

    while (getline(input,s))
    {
        numLines++;
        numChars += s.length();
    }
    cout << "number of lines = " << numLines
         << ", number of characters = " << numChars << endl;
    return 0;
}
```

`filelines.cpp`

The function `getline` extracts a line, stores the line in a `string` variable, and returns the state of the stream. Some programmers prefer to test the stream state explicitly:

```
while (getline(input,s) && ! input.fail())
```

However, it is fine to use `getline` in a loop test, both to extract a line and as a test to see whether the extraction succeeds, just as the expression `infile >> word` can be used as the test of a `while` loop to process all the white space–delimited words in a stream.

OUTPUT

```
prompt> filelines
enter name of input file: macbeth.txt
number of lines = 2849, number of characters = 110901
prompt> lines
enter name of input file: hamlet.txt
number of lines = 4463, number of characters = 187271
prompt> filelines
enter name of input file: filelines.cpp
number of lines = 31, number of characters = 696
```

As used in Program 9.3, `getline` has two parameters: an input stream and a string for storing the line extracted from the stream. The stream can be a predefined stream such as `cin` or an `ifstream` variable such as `input`, as used in Program 9.3. An optional third parameter to `getline` indicates the **line delimiter** or sentinel character that identifies the “end of line”. The `string` function `getline` extracts one line from

the stream passed as the first parameter. The characters composing the line are stored in the `string` parameter `s`. The state of the stream after the extraction is returned as the value of the function. The return value is a reference to the stream, because streams should not be passed or returned by value.

Syntax: `getline`

```
istream &
getline(istream & is,
        string & s,
        char sentinel = '\n');
```

Normally, the end of a line is marked by the newline character `'\n'`. However, it is possible to specify a different value that will serve as the end-of-line character. An optional third argument can be passed to `getline`. This `char` parameter, (`sentinel` in the diagram), is used as the end-of-line character. The end-of-line character is extracted from the stream but is *not* stored in the `string` `s`.

For example, suppose a file is formatted with a CD artist and title on the same line, separated by a colon `':'`, as follows:

```

Jimmy Buffet : Fruitcakes
Paul Simon : The Rhythm Of The Saints
Boyz II Men : Cooleyhighharmony

```

The following loop reads this file storing the artist and title in two strings.

```

string artist,title;
while (getline(input,artist,':') && getline(input,title))
{   cout << artist << "\t" << title << endl;
}

```

Program Tip 9.3: Be very careful when using both `getline` and the extraction operator `>>` with the same stream. Extraction skips white space, but often leaves the white space on the stream. For example, if you type characters and press Enter when `>>` is used, the newline character that was input by pressing the Enter key is still on the `cin` stream. A subsequent `getline` operation reads all characters until the newline, effectively reading nothing. If your programs seem to be skipping input from the user, look for problems mixing these two input operations. It's better to use just `getline` to read strings, and the conversion operators `atof` and `atoi` (see "strutils.h" in Howto G) to convert a string to an `int` or to a `double`, respectively, than to mix the two forms of stream input.

The value returned by `getline` is the same value that would be returned if the stream member function `fail` were called immediately after the call to `getline`. As we've seen, some programmers prefer to make the call to `fail` explicitly rather than to use the value returned by `getline`. A `getline` operation will fail if the stream cannot be read, either because it is bound to a nonexistent file or because no more lines are left on the stream.

A stream variable can be used by itself instead of the function `fail`. For example,

```

input.open(filename.c_str());
if (input.fail())
{   cout << "could not open file " << filename << endl;
    exit(1);
}

```

can be replaced by the statements

```

input.open(filename.c_str());
if (!input)
{   cout << "could not open file " << filename << endl;
    exit(1);
}

```

The use of `!input` in place of `input.fail()` is common in C++ programs. I'll use `fail` most of the time, because it makes it clear how the stream is being tested.

9.2.2 Parsing Line-Oriented Data Using `istringstream`

Data are often line-oriented, because people find it easy to edit and read lines of words, numbers, and other data. Reading data is straightforward when the number of items per line is the same for an entire data set, since a `for` loop can be used to iterate a set number of times for each input line. Another approach is needed when the number of items per line varies. For example, we might want to access the individual words in the titles of the CDs stored in a file:

```
The Beatles
Sergeant Pepper's Lonely Hearts Club Band
Strauss
Also Sprach Zarathustra
...
```

We might need to write a program to average students' grades, where each student has a different number of grades stored in the following format (firstname lastname grades):

```
Dave Reed 55 60 75 67 72 59
Mike Clancy 88 92 91 97
Stuart Reges 99 94 98 91 95
```

In general, parsing input and reading data often make up the hardest part of developing a program. Reading data is not an algorithmically challenging problem, but dealing with badly formed data and different kinds of data can be an unpleasant part of programming.

We already know how to process stream input a word at a time using the extraction operator `>>`. We need a tool that lets us use `>>` on one line of a file. The class `istringstream` (for **input string stream**), accessible by including the file `<sstream>`⁷, is just the tool we need. The `istringstream` class constructs a stream bound to a string as the source of the input, much as the `ifstream` class constructs a stream bound to a disk file as the source of input. Because an `istringstream` object is a stream, it supports the same functions and operators as `ifstream` objects and the standard input stream `cin`.

The code in `readnums.cpp`, Program 9.4, uses an `istringstream` variable to read line-oriented numerical data where the number of integers on each line varies. The average of the numbers on each line of input is calculated and printed.

Program 9.4 `readnums.cpp`

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

// illustrates use of input string streams
```

⁷The name `istringstream` is relatively new; older compilers that don't use this name will use `istrstream`. The header file for `istrstream` is `<strstream.h>`. On some systems this may be shortened to `<strstrea.h>`.

```

int main()
{
    string s;
    cout << "program computes averages of lines of numbers." << endl;
    cout << "to exit, use end-of-file" << endl << endl;

    while (getline(cin,s))
    {
        int total = 0;
        int count = 0;
        int num;

        istringstream input(s);
        while (input >> num)
        {
            count++;
            total += num;
        }
        if (count != 0)
        {
            cout << "average of " << count << " numbers = "
                << double(total)/count << endl;
        }
        else
        {
            cout << "data not parsed as integers" << endl;
        }
    }
    return 0;
}

```

readnums.cpp

OUTPUT

```

prompt> readnums
program computes averages of lines of numbers.
to exit, use end-of-file

10 20 30
average of 3 numbers = 20
1 2 3 4 5 6 7 8
average of 9 numbers = 4.5
1 -1 2 -2 3 -3 4 -4 5 -5
average of 10 numbers = 0
apple orange guava
data not parsed as integers
2 4 apple 8 10
average of 2 numbers = 3
^Z

```

The `getline` function reads one line of input into the string `s`, and the `istringstream` variable `input` is constructed from `s`. Then `input` is used as a stream: integers are extracted using `>>` until the extraction fails. The variable `input` *must* be defined (and hence constructed) inside the `while (getline(cin,s))` loop of `readnums.cpp`. The source of data in an `istringstream` object is the string passed as an argument to the `istringstream` constructor. It is not possible to define `input` before the loop and then rebind `input` to a string entered by the user within the loop. The `istringstream` variable `input` is constructed anew at each iteration of the `while (getline(cin,s))` loop.

An `istringstream` is constructed from a standard string object, but it will work correctly when constructed from a C-style string⁸. Changing the value of the string used to construct an `istringstream` object while the stream is being used can lead to trouble.

9.2.3 Output Using `ostringstream`

It's relatively easy to learn to program with `istringstream` objects because they behave exactly like `cin` or an `ifstream` variable. Sometimes it's useful in programs to be able to form a string by joining different values, such as a string formed from the string "the answer is " and the int 257. The string formed by joining the different values can be passed as a parameter, printed, and in general be treated like any string object. You can use the conversion function `tostring` from `strutils.h` in `Howto G` and string catenation for this.



```
string result = "the answer is " + tostring(257);
```

The value of `result` is the string "the answer is 257". However, it's much easier to combine together different values using an `ostringstream` object (output string stream).

```
ostringstream output;
output << "the answer is " << 257;
string result = output.str();
```

An `ostringstream` (like `istringstream`, accessible from `<sstream>`) behaves like an output stream, that is like `cout` or an `ofstream` variable. Values can be written to the output string stream using standard stream insertion, including formatting operators like `setw` (see `Howto B`). The method `ostringstream::str()` returns a string that contains the characters written to the output string stream.



⁸If a non-standard string class is used, (e.g., from "apstring.h" or "tstring.h"), you'll need to use the `c_str()` string member function when constructing an `istringstream` variable.

Niklaus Wirth (b. 1934)

Niklaus Wirth is perhaps best known as the inventor/developer of the programming language Pascal. He also was an early adherent of a methodology of programming he called “stepwise refinement,” writing a paper in 1971 that called for developing programs in a style I’ve called *iterative enhancement* in this book. Pascal was developed in the early 1970s; it was not, as conventional wisdom would have it, developed solely as a language for educational use. In his 1984 Turing Award lecture Wirth says:

Occasionally, it has been claimed that Pascal was designed as a language for teaching. Although this is correct, its use in teaching was not the only goal. In fact, I do not believe in using tools and formalisms in teaching that are inadequate for any practical task. By today’s standards, Pascal has obvious deficiencies for programming large systems, but 15 years ago it represented a sensible compromise between what was desirable and what was effective.

Wirth continued to develop languages that were successors of Pascal, notably Modula-2 and Oberon. In discussing the difficulties of developing hardware and software, Wirth has this to say about the complexity of these tasks:

It is true that we live in a complex world and strive to solve inherently complex problems, which often do require complex mechanisms. However, this should not diminish our desire for elegant solutions, which convince by their clarity and effectiveness. Simple, elegant solutions are more effective, but they are harder to find than complex ones, and they require more time, which we too often believe to be unaffordable.

When contacted about providing a picture for the second edition of this book, Wirth replied “I must say that I have never been a friend of C++.” Most of this material is taken from [Wir87].

9.2.4 Strings, Streams, and Characters

Sometimes it is useful to regard a file (and its associated stream) as a collection of characters rather than as a collection of lines. Of course, we could read a file a line at a time using `getline` and then access each character of the extracted string, but sometimes character-at-a-time input is more appropriate than line-at-a-time input. The stream member function `get` is used to read one character at a time. White space is *not* skipped when `get` is used. Program 9.5, *filelines2.cpp*, uses `get` to count the characters in a file one at a time. Note that `getline` is not a stream member function but that `get` is.

Program 9.5 filelines2.cpp

```
#include <iostream>
#include <fstream>
#include <cstdlib>    // for exit
#include <string>
using namespace std;

#include "prompt.h"

// count # of lines and chars in input file

int main()
{
    long numChars = 0;
    long numLines = 0;
    char ch;
    string filename = PromptString("enter name of input file: ");
    ifstream input;
    input.open(filename.c_str());

    if (input.fail() )
    {
        cout << "could not open file " << filename << endl;
        exit(1);
    }
    while (input.get(ch))           // reading char succeeds?
    {
        if ('\n' == ch)           // read newline character
        {
            numLines++;
        }
        numChars++;
    }

    cout << "number of lines = " << numLines
         << ", number of characters = " << numChars << endl;
    return 0;
}
```

filelines2.cpp**OUTPUT**

```
prompt> filelines2
enter name of input file: macbeth.txt
number of lines = 2849, number of characters = 113750
prompt> filelines2
enter name of input file: hamlet.txt
number of lines = 4463, number of characters = 191734
```

The number of lines printed by *filelines2.cpp*, Program 9.5, is the same as the number of lines calculated by *filelines.cpp*, Program 9.3, but the number of characters printed is different. If you look carefully at all the numbers printed by both programs, you may be able to determine what the “missing” characters are. In the on-line version of *Hamlet*, both programs calculate the number of lines as 4,463, but Program 9.3 calculates 187,271 characters, compared to the 191,734 calculated by Program 9.5. Not coincidentally, $187,271 + 4,463 = 191,734$. The newline character ‘\n’ is not part of the total number of characters calculated by Program 9.3. This points out some subtle behavior of the `getline` function. `getline` reads a line of text, terminated by the newline character ‘\n’. The newline character is read but is *not* stored in the string parameter to `getline`. You can change Program 9.3 to count newlines by changing the calculation of `numChars` as follows:

```
numChars += s.length() + 1; // +1 for newline
```

The comment is important here; the reason for the addition of + 1 may not be apparent without it.

Pause to Reflect



9.9 Write a small program that prompts for the name of an artist and prints all CDs by the artist. Assume input is in the following format.

```
The Black Crowes
The Southern Harmony and Musical Companion
10,000 Maniacs
Blind Man's Zoo
The Beatles
Rubber Soul
```

For example, if the user enters `The Beatles`, the output might be

```
Sergeant Pepper's Lonely Hearts Club Band
The White Album
Revolver
Rubber Soul
```

depending on what CD titles are stored in the file.

- 9.10** From its use in *filelines2.cpp*, Program 9.5, the `char` parameter to `get` must be a reference parameter. Why is this the case?
- 9.11** Program 9.5, *filelines2.cpp*, can be modified so that it copies a file by writing every character (using `<<`) that is read. What modifications are necessary so that the user is prompted for the name of a new file to be written that will be a copy of the file that is read?
- 9.12** How can the copy program from the previous exercise be modified so that all upper-case letters in the input file are converted to lower-case letters in the output file? (*Hint*: The change is very straightforward.)

9.3 Case Study: Removing Comments with State Machines

With the stream and string functions we have studied, we now have the choice of reading streams in several ways:

- A word at a time, using `>>` and `string` variables
- A line at a time, using `getline` and `string` variables
- A character at a time, using `get` and `char` variables

In this section we'll develop a program to remove all comments from a file. We'll see that character-at-a-time input facilitates this task, and we'll study an approach that extends to other parsing-related problems.⁹ We'll use a new syntactic feature of C++ called an **enum**.

9.3.1 Counting Words

To make the method used to remove comments more familiar, we'll modify *filelines2.cpp* to count words in addition to counting lines and characters. We'll use the same specification for words that the extraction operator `>>` uses: a white space delimited sequence of characters.



Since we're reading one character at a time, we'll need a method to determine when a word starts and when it ends. We'll use the function `isspace` from `<cctype>` (see Table F.2 in Howto F) to determine if a character is white space, but how can we keep track of word boundaries? The key is recognizing that the program is in one of two **states**. It is either reading a word, or not reading a word. When the program reads a space, it is not in a word. When it reads a non-space character it is in a word. The transition from the in-a-word state to the not-in-a-word state marks a word, so whenever this transition occurs we'll update the word count. The transitions are diagrammed in Figure 9.1 and shown in code in Program 9.6, *wc.cpp*.¹⁰

Program 9.6 *wc.cpp*

```
#include <iostream>
#include <fstream>
#include <cstdlib>    // for exit
#include <cctype>    // for isspace
#include <string>
using namespace std;
```

⁹A program is **parsed** by the compiler in the process of converting it into assembly or machine language. "Parse" usually refers to the process of reading input in identifiable chunks such as C++ identifiers, reserved words, etc.

¹⁰The Unix program *wc* counts words, lines, and characters, hence the name.

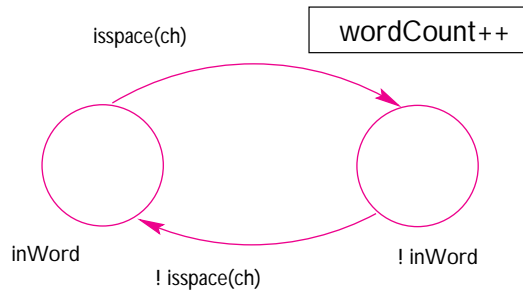


Figure 9.1 State for counting words.

```

#include "prompt.h"

// count # of lines and chars in input file

int main()
{
    long numChars = 0;
    long numLines = 0;
    long numWords = 0;
    char ch;
    bool inWord = false; // initially not reading a word

    string filename = PromptString("enter name of input file: ");
    ifstream input;
    input.open(filename.c_str());

    if (input.fail() )
    {
        cout << "could not open file " << filename << endl;
        exit(1);
    }
    while (input.get(ch)) // reading char succeeds?
    {
        if ('\n' == ch) // read newline character
        {
            numLines++;
        }
        numChars++;
        if (isspace(ch))
        {
            if (inWord) // just finished a word
            {
                inWord = false;
                numWords++;
            }
        }
        else // not a space
        {
            if (! inWord) // just started a word
            {
                inWord = true;
            }
        }
    }
}

```

```

if (inWord) numWords++; // ended in a word

cout << "lines = " << numLines
     << "\tchars = " << numChars
     << "\twords = " << numWords << endl;
return 0;
}

```

wc.cpp

O U T P U T

```

prompt> wc
enter name of input file: melville.txt
lines = 1609      chars = 82140   words = 14353
prompt> wc
enter name of input file: bible10.txt
lines = 228760   chars = 4959549 words = 822899

```

9.3.2 Problem Specification: What Is a Comment?

The first step in writing almost any program is to specify the problem properly. We must decide what a comment is, and we should try to identify potential problems in our definition. We'll write a program that removes comments beginning with `//`. These comments extend to the end of a line and are simpler to remove than `/* ... */` comments, which can extend over several lines. We'll read and echo all characters except those that are part of a comment.

9.3.3 A State Machine Approach to I/O

Our comment-removing program will prompt for the name of a program (actually any text file) and print the program with all the comments removed. Our first program will output using `cout`, but we'll design the program so that output to an `ofstream` object will be a simple change. We must decide whether to read a program a line at a time or a character at a time. Since `//` comments are line-oriented, reading input a line at a time makes sense. We could use the string member function `find` to determine whether each line contains the string `"//"` and, if so, where the `"//"` begins. However, this approach cannot be extended to removing `/* ... */` comments, which can extend over several lines, so we'll use character-at-a-time input instead.

We'll use a state machine approach in reading and removing comments. In a **state machine** program, each input character causes the program to change its behavior depending on the program's state. We'll use a three-state function to remove comments. The function will be in one of three states as it reads each character:

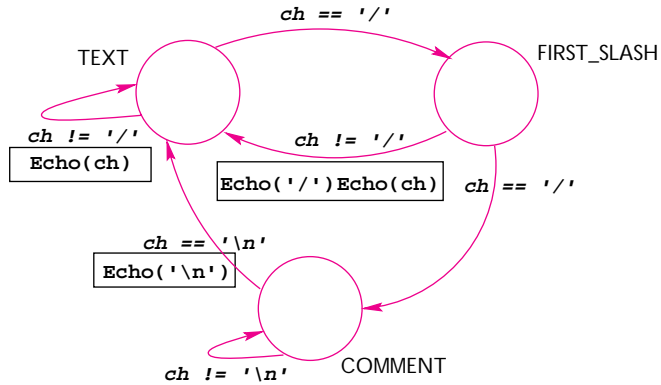


Figure 9.2 State machine diagram for removing // comments.

1. Processing regular, uncommented text
2. A slash '/' has just been read
3. Processing commented text

In Figure 9.2, these states are labeled as *TEXT*, *FIRST_SLASH*, and *COMMENT*. Each state is shown as a circle, and state changes are shown with arrows. The program can change state each time a character is read, although it's possible to stay in the same state. Some state changes (or state transitions) are accompanied by an action, shown in a shaded box. In the text-processing state *TEXT*, nonslash characters are echoed; a slash character is not echoed but causes a state transition to the state labeled *FIRST_SLASH*. In the state *FIRST_SLASH* we don't know yet whether a comment follows or whether the division operator / was just read. The answer depends on the next character read. If a slash character is read, we know a comment follows, so we change state to *COMMENT*; otherwise there was only one slash, so we echo the slash and the character just read and return to *TEXT*, the state of parsing noncommented text. Finally, in the *COMMENT* state, we ignore all characters. However, when a newline character '\n' is read, we know the comment has ended, so the newline is echoed and the state changes back to *TEXT*.

The advantage of the state approach is that we simply read one character at a time and take an action on the character depending on the current state of the program. In a way, the states serve as memory. For example, in the state *FIRST_SLASH* we know that one slash remains unprocessed. If the slash doesn't begin a comment, we'll echo the unprocessed slash and change to reading regular text.

Program 9.7, *decomment.cpp*, implements this state machine approach. The method `Decomment::Transform` actually removes the comments. An enumerated type `Decomment::ReadState` is used so that symbolic values appear in code for each state. The symbolic label *FIRST_SLASH* is more informative than a number like 1 in reading code. We'll cover enumerated types after we discuss the program.

Program 9.7 decomment.cpp

```
#include <iostream>
#include <fstream>
#include <cstdlib>    // for exit
using namespace std;
#include "prompt.h"

// Owen Astrachan 7/4/1996, revised 5/4/99
// state-machine approach for removing all // comments from a file
// (doesn't handle // in a string, e.g., " test // comment ")

class Decoment
{
public:
    Decoment();
    void Transform(istream& input, ostream& output);

private:
    void Echo(char ch, ostream& output);

    const char SLASH;
    const char NEWLINE;

    enum ReadState{TEXT, FIRST_SLASH, COMMENT};
};

Decoment::Decoment()
: SLASH('/'),
  NEWLINE('\n')
{
    // constants initialized
}

void Decoment::Echo(char ch, ostream& output)
{
    output << ch;
}

void Decoment::Transform(istream& input, ostream& output)
{
    char ch;
    ReadState currentState = TEXT;
    while (input.get(ch))        // read one char at a time
    {
        switch(currentState)
        {
            case TEXT:
                if (ch == SLASH)        // potential comment begins
                {
                    currentState = FIRST_SLASH;
                }
                else
                {
                    Echo(ch,output);
                }
            }
        }
    }
}
```

```
    }
    break;

case FIRST_SLASH:
    if (ch == SLASH)
    {   currentState = COMMENT;
    }
    else // one slash not followed by another
    {   Echo(SLASH,output); // print the slash from last time
        Echo(ch,output); // and the current character
        currentState = TEXT; // reading uncommented text
    }
    break;

case COMMENT:
    if (ch == NEWLINE) // end-of-line is end of comment
    {   Echo(NEWLINE,output); // be sure to echo end of line
        currentState = TEXT;
    }
    break;
}
}
}

int main()
{
    string filename = PromptString("enter filename: ");
    ifstream input(filename.c_str());
    if (input.fail())
    {   cout << "could not open " << filename << " for reading" << endl;
        exit(1);
    }
    Decoment dc;
    dc.Transform(input,cout);

    return 0;
}
```

O U T P U T

```

prompt> decomment
enter name of input file: commtest.cpp
#include <iostream>
using namespace std;

int main()
{
    int x = 3;
    cout << x / 3 << endl;
    return 0;
}

```

Enum values are used as the values of the variable `currentState`. Otherwise the logic is precisely illustrated in Figure 9.2. The test input is the following file, named *commtest.cpp*:

```

#include <iostream>
using namespace std;
// this is a sample program for comment removal
int main()
{
    int x = 3;           // meaningful identifier??
    cout << x / 3 << endl; // complex math is fun
    return 0;           // this is a useful comment
}

```

The program *decomment.cpp* does remove all comments properly, but there is a case that causes text to be removed when it shouldn't be. When the two-character sequence `//` is embedded in a string, it is not the beginning of a comment:

```
cout << "Two slashes // not a comment" << endl; // tricky?
```

This situation causes problems with the state machine used in *decomment.cpp*, but it's possible to add more states to fix the problem.

Pause to Reflect



- 9.13** Modify *decomment.cpp*, Program 9.7, so that the output goes to a file specified by the user.
- 9.14** Draw a state transition diagram similar to Figure 9.2 but for removing `/* ... */` comments. Don't worry about `//` comments; just remove the other kind of comment.

- 9.15** It's possible to use two states to remove `//` comments. Instead of using the state `COMMENT` in `decomment.cpp`, use `getline` to gobble up the characters on a line when a slash is read in the state `FIRST_SLASH`. Modify `decomment.cpp` to use this approach.
- 9.16** Add states to either the diagram or the program `decomment.cpp` to avoid removing the `//` sequence when it is embedded in a string.
- 9.17** Write a state transition diagram for word-at-a-time input that you could use to find all `int` variables. Solve a simple version of the problem, assuming that every variable is defined separately—that is, there are no definitions in the form

```
int x, y, z;
```

What other situations can cause problems for your approach?

9.3.4 Enumerated Types

An **enumerated type** allows you to create all the legal values for a new type. For example, a coin type might have the values `heads` and `tails`, and a color spectrum type might have the values `red`, `orange`, `yellow`, `green`, `blue`, `indigo`, and `violet`. Using enumerated types makes programs more readable. The type `ReadState` in the class `Decomment` from `decomment.cpp`, Program 9.7 has three values: `TEXT`, `FIRST_SLASH`, and `COMMENT`. A variable of type `ReadState` can have only these values.

An `enum` introduces a new type whose possible values are defined completely when the `enum` is declared. Each value of an `enum` type has an associated integer value; default values of 0, 1, ... are assigned to each `enum` value in succession. However, `enums` are most often used because they let values be represented symbolically rather than numerically. For example, the declaration

```
enum CardSuit{spade, heart, diamond, club};
```

creates a new type `CardSuit`. The variable definition `CardSuit suit;` creates a variable `suit` whose only possible values are `spade`, `heart`, `diamond`, and `club`. The assignment `suit = spade` is legal; the assignment `suit = 1` is not legal. The integer values associated with `CardSuit` values make `spade` have the value 0 and `club` have the value 3. The statement `cout << suit` outputs an integer, either 0, 1, 2, or 3. `Enums` are *not* printed symbolically except, perhaps, in a debugging environment. It's possible to assign explicit values using

```
enum CardSuit {spades=2, hearts=5, diamonds=7, clubs=9};
```

so that the value associated with `diamonds` is 7, for example, but there are very few good reasons to do this. `Enums` let you use symbolic values in your code, and this can make code easier to read and maintain. Relying on a correspondence between the value 1 and a suit of hearts, which would be necessary if `enums` weren't used, can cause errors since it's easy to forget that 1 means hearts and 0 means spades.

Using enums: Conversion between enum and int. As noted earlier, an `int` value cannot be assigned to an enum variable. It is possible, however, to assign an enum to an `int`.

```
enum CardSuit{spades, hearts, diamonds, clubs};
int k = spades;           // legal
CardSuit c = 3;          // illegal
CardSuit s = CardSuit(3); // legal
```

As this example shows, if an explicit cast is used, an `int` can be converted to an enum. Program 9.8 shows an enum used as an `int` as an argument to `RandGen::RandInt` and as the index of an array.

Program 9.8 `enumdemo.cpp`

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

#include "randgen.h"

int main()
{
    enum spectrum{red, orange, yellow, green, blue, indigo, violet};

    string specstrings[] = {"red", "orange", "yellow", "green",
                           "blue", "indigo", "violet"};

    RandGen gen;
    spectrum color = spectrum(gen.RandInt(red,violet));
    cout << specstrings[color] << endl;

    if (color == red)
    {    cout << "roses are red" << endl;
    }
    else
    {    cout << "that's a pretty color" << endl;
    }
    return 0;
}
```

`enumdemo.cpp`

O U T P U T

```
prompt> enumdemo
indigo
that's a pretty color
prompt> enumdemo
red
roses are red
```

Class enums. By restricting an enum to be used as part of a class, several different classes can share the same enum symbolic names. For example, it is not legal to declare the following two enums in the same program unless they are declared inside separate classes because the value orange cannot be shared among different enumerated types.

```
enum spectrum {red, orange, yellow, green,
              blue, indigo, violet};
enum fruit {orange, apple, cherry, banana};
```

However, if the enumerated types are moved inside classes `Spectrum` and `Fruit`, respectively, then there is no conflict since the values are `Spectrum::orange` and `Fruit::orange`, which are different.

```
class Spectrum
{
public:
    ...
    enum Color{red, orange, yellow, green,
              blue, indigo, violet};
};
class Fruit
{
public:
    ...
    enum Kind{orange, apple, cherry, banana};
};
```

Here the new types introduced by the enum declaration are `Spectrum::Color` and `Fruit::Kind`; the scope-resolution operator `::` is required as part of the enum type name except when the enum is used within a member function.

9.4 Case Study: Overloaded Operators and the `clockTime` Class

Often the hardest part of writing a program is reading the data. This can be difficult because data are often stored in a form that is inconvenient from the point of view of the programmer. In general, this sentiment is aptly stated as

I/O is messy.

In this section we'll develop a program that calculates the total playing time for all the music stored on a compact disc (CD). The program could be extended with more options such as those found on typical CD players: select some specific songs/tracks or play all songs/tracks after randomly shuffling them. You could also use the program to combine tracks from different CDs and compute the total playing time, part of making your own CD of your favorite songs.

We'll use the program to explore the implementation of overloaded operators. We have used overloaded operators in many programs.

9.4 Case Study: Overloaded Operators and the `clockTime` Class

427

Overloaded operators are used when we add `BigInt` values using a plus sign, compare them using a less-than sign, and read and write them using extraction and insertion operators. These operators are defined for built-in types, but C++ allows programmers to define the operators for user-constructed types. Operator overloading is covered in detail in Howto E, but we'll give basic guidelines and details here on how to overload operators with minimal programmer effort without sacrificing performance.



The input to the program is a file in the format shown here. Each line of the file consists of the duration of a track followed by the name of the track. For example, for the compact disc *The Best of Van Morrison* (1990, Mercury Records) the input follows.

```
3:46    Bright Side Of The Road
2:36    Gloria
4:31    Moondance
2:40    Baby Please Don't Go
4:19    Have I Told You Lately
3:04    Brown Eyed Girl
4:21    Sweet Thing
3:22    Warm Love
3:57    Wonderful Remark
2:57    Jackie Wilson Said
3:14    Full Force Gale
4:28    And It Stoned Me
2:46    Here Comes The Night
3:04    Domino
4:05    Did Ye Get Healed
3:32    Wild Night
4:40    Cleaning Windows
4:54    Whenever God Shines His Light
4:54    Queen Of The Slipstream
4:44    Dweller On The Threshold
```

For Handel's *Water Music (Suite in F Major, Suite in D Major)* (Deutsche Grammophon, 1992, Orpheus Chamber Orchestra) the input is

```
3:12    Overture
1:49    Adagio e staccato
2:23    Allegro
2:11    Andante
2:25    da capo
3:22    Presto
3:26    Air.Presto
2:33    Minuet
1:38    Bourree.Presto
2:17    Hornpipe
2:53    (without indication)
1:52    Allegro
2:42    Alla Hornpipe
```

```
1:01   Minuet
1:37   Lentement
1:10   Bourree
```

To determine the total playing time of a CD, the following pseudocode provides a good outline.

```
total = 0;
while (getline(input,line))
{   parse track_time and title from line
    total += track_time;
}
cout << "total playing time = " << total;
```

There are several details that must be handled to translate the pseudocode into a working program. Most of these details involve getting the data from a file into the computer for a program to manipulate. Although algorithmically this is a simple problem, the details make it hard to get right.¹¹ There are enough sticky details in the I/O that developing the program takes patience, even if it seems easy at first.

9.4.1 Throw-Away Code vs. Class Design



We'll be able to read the input using the string and stream functions we've covered in this chapter. In previous programs we used the functions `atoi` and `atof` from `strutils.h`, Program G.8 in *Howto G* to convert strings to ints or doubles. We can use `atoi` here to transform strings into int values for minutes and seconds.

At this point we face a decision as developers of the program. We could develop code specifically for this program that correctly accumulates the total time for a CD. The program would work well for the task at hand, but the code would not be very general. At another extreme, we could develop a class for manipulating time stored in hours, minutes, and seconds with overloaded arithmetic and I/O operators. The class would provide code that could be reused in other contexts. Code reuse is a goal of object-oriented design and programming, but it takes more effort to develop reusable code than to develop program-specific code. The decision as to which approach to take is not always simple; often a "quick and dirty" programming approach is quite appropriate. We might even implement the quick and dirty solution as a first step in developing a working program.

Program Tip 9.4: A quick and dirty solution is sometimes the best approach in getting a working program to solve a problem. Even quick and dirty programs should be elegant and should be carefully commented since today's quick and dirty, use it once and forget it program may be still running ten years from now.

¹¹David Chaiken, a computer scientist trained at MIT, uses the acronym SMOP to refer to this kind of problem—it's a *Simple Matter Of Programming*. Usually those who claim it's simple aren't writing the program.

Using the three-parameter `getline` function, for example, we could write this loop to solve the problem.

```
string minutes, seconds, title;
int secSum = 0, minSum = 0;
while (getline(input, minutes, ':') &&
       getline(input, seconds, ' ') &&
       getline(input, title)) // reading line ok
{
    minSum += atoi(minutes);
    secSum += atoi(seconds);
}
cout << "total time is " << minSum << ": " << secSum << endl;
```

This will yield a total like `65:644`, not quite as readable as `1:15:44`. We'll design a class for manipulating time as stored in the format: hours, minutes, and seconds. We'll name the class `clockTime` and write functions that permit times in this format to be added together, compared using boolean operators, and output to streams.

9.4.2 Implementing the `clockTime` Class

In designing a class, two major decisions influence the development process.

1. What is the class behavior? This helps in determining appropriate public and private member functions. (See Programming Tip 7.1.)
2. What is the class state? This helps in determining what instance variables are needed for the class.

We'll concentrate on behavior first. To make the `clockTime` class minimally useful we'll need to implement the following.

- **Constructor(s).** We should probably include a default constructor so that vectors of `clockTime` objects can be defined, and other constructors that seem useful as we develop scenarios of how the class will be used.
- **Printing.** This means overloading the stream insertion operator `<<` so that `clockTime` objects can be written to `cout` and other streams. We'll use the same method in this class we use in other classes developed for this book; we'll implement a converter function `clockTime::toString()` that converts a `clockTime` object to a string. This makes it easy to insert onto a stream since we can already output string objects. The `clockTime::toString()` function is useful in other contexts. For example, we use it in the implementation of the class `CircleStatusBar` (from `statusbar.h`) used in Program 6.16, `maxword3.cpp`.
- **Relational operators.** We'll need at least operator `==`, but as we'll see the implementation of these operators is relatively straightforward.
- **Arithmetic.** We need to be able to add `clockTime` objects since that's the reason we're exploring the class. We'll implement addition only, but make it possible to implement other operations later.

These functions lead to the interface given in *clockt.h*, Program 9.9. As we'll see when discussing overloaded operators, the functions `Less` and `Equal` are helper functions for implementing the relational operators. We'll discuss the prototype for the arithmetic operator `+=` in Section 9.4.8, and the function `Normalize` when we discuss constructors below.

Program 9.9 *clockt.h*

```
#ifndef _CLOCKTIME_H
#define _CLOCKTIME_H

#include <iostream>
#include <string>
using namespace std;

// class for manipulating "clock time", time given in hours, minutes, seconds
// class supports only construction, addition, Print() and output <<
//
// Owen Astrachan: written May 25, 1994
//                 modified Aug 4, 1994, July 5, 1996, April 29, 1999
//
// ClockTime(int secs, int mins, int hours)
//   - normalized to <= 60 secs, <= 60 mins
//
//   access functions
//
//   Hours()   - returns # of hours in ClockTime object
//   Minutes() - returns # of minutes in ClockTime object
//   Seconds() - returns # of seconds in ClockTime object
//   toString() - time in format h:m:s
//                (with :, no space, zero padding)
//
//   operators (for addition and output)
//
//   ClockTime & operator +=(const ClockTime & ct)
//   ClockTime operator +(const ClockTime & a, const ClockTime & b)
//
//   ostream & operator <<(ostream & os, const ClockTime & ct)
//   inserts ct into os, returns os, uses Print()

class ClockTime
{
public:
    ClockTime();
    ClockTime(int secs, int mins, int hours);

    int    Hours()      const;    // returns # hours
    int    Minutes()    const;    // returns # minutes
    int    Seconds()    const;    // returns # seconds
    string toString()   const;    // converts to string

    bool   Equals(const ClockTime& ct) const; // true if == ct
};
```

```

bool    Less (const ClockTime& ct) const; // true if < ct

const ClockTime & operator +=(const ClockTime & ct);

private:

    void Normalize();           // < 60 secs, < 60 min

    int mySeconds;             // constrained: 0-59
    int myMinutes;            // constrained: 0-59
    int myHours;

};
// free functions, not member functions

ostream & operator << (ostream & os, const ClockTime & ct);
ClockTime operator + (const ClockTime & lhs, const ClockTime & rhs);

bool operator == (const ClockTime& lhs, const ClockTime& rhs);
bool operator != (const ClockTime& lhs, const ClockTime& rhs);
bool operator <  (const ClockTime& lhs, const ClockTime& rhs);
bool operator >  (const ClockTime& lhs, const ClockTime& rhs);
bool operator <= (const ClockTime& lhs, const ClockTime& rhs);
bool operator >= (const ClockTime& lhs, const ClockTime& rhs);

#endif

```

clockt.h



The `ClockTime` Constructors. An instance of the class `ClockTime` might be constructed by specifying just the seconds or just the hours. For the preliminary development of the class we'll provide a default constructor, which will initialize a time to 0 hours, 0 minutes, and 0 seconds, and a three-parameter constructor, which specifies all three quantities. In our final design, the default constructor will construct an object representing the current time when the object is constructed, just as the default `Date` class constructor yields the current day (see `date.h`, Program G.2, Howto G.)

The first step in implementing the class requires implementing a constructor and some mechanism for determining the value of a `ClockTime` object. For example, we could implement accessor functions for obtaining hours, minutes, or seconds. We could also implement a function to print a `ClockTime` object. We can't develop other operators or member functions until we can define objects and determine what the objects look like.

Program Tip 9.5: The first step in implementing a class should include constructors and some method for determining what an object looks like.

The state of an object can be examined by accessor functions or by using a `toString` method and then printing the object.

We can't implement a constructor without deciding about the state of the class. For the `ClockTime` class the state instance variables are straightforward: hours, minutes,

and seconds. These are each integer fields, although the minutes and seconds fields are constrained to have values in the range 0 through 59. There are alternatives. Rather than store three values, we could store just seconds, and convert to other formats for printing. This would make it very easy to add 1:02:15 and 2:17:24 since these values would be represented as 3,735 and 8,244 seconds, respectively. The sum is simple to compute in C++, but conversion to hours, minutes, and seconds is needed for printing.

9.4.3 Class or Data Invariants

What if the user constructs objects as follows (hours first, seconds last)?

```
ClockTime a;
ClockTime b(2,27,31);
ClockTime c(3,77,91);
```

Object `a` is 0:0:0, object `b` is 2:27:31, and object `c` is 4:18:31, although this isn't immediately obvious from the arguments to the constructor of `c`. We could ignore values that aren't between 0 and 59 for minutes and seconds, but we'd like our class to be robust in the face of errors so we'll try to do something that makes sense, but leaves an object in a good state (i.e., minutes and seconds between 0 and 59.)

Just as the constructor may involve overflow values, the code for operator `+=` will need to check for overflow of minutes and seconds and adjust the other fields accordingly. Alternatively, a **normalizing** function could be written to ensure that all minutes and seconds were within proper range, as a **data invariant** of the class `ClockTime`. Just as a loop invariant is a statement that is true on every pass through a loop, a class data invariant is a property of class state that is true after each method has executed. In this case the data invariant would be something like:

Internal representations of minutes and seconds in a `ClockTime` object are always between 0 and 59.

All objects must maintain this invariant, but some methods like operator `+=` may invalidate the invariant, so the code must ensure that the invariant is re-established after each method executes. An object that maintains the invariant is said to be in a **normal form**. We'll include a private, helper member function `Normalize`, that ensures that the invariant is maintained. This normalizing function will be called after adding two times and after construction in case a time is constructed with 79 seconds. At this point, we'll implement two constructors and accessors for hours, minutes, and seconds. The constructors are shown below with the accessor function for hours.

```
ClockTime::ClockTime()
    : mySeconds(0), myMinutes(0), myHours(0)
    // postcondition: time is 0:0:0
    {
}

ClockTime::ClockTime(int secs, int mins, int hours)
    : mySeconds(secs), myMinutes(mins), myHours(hours)
```

```
// precondition: all data fields initialized
{
    Normalize();
}

int ClockTime::Hours() const
// precondition: return # of hours
{
    return myHours;
}
```

With the header file *clockt.h*, constructors, and accessors we're ready to test the preliminary implementation. Once we're sure we can construct valid `ClockTime` objects, we'll turn to implementing the overloaded operators. We'll test the class first, so that we know its minimal implementation works correctly before developing new code.

9.4.4 Overloaded Operators



Only member functions have access to an object's private data. This makes it difficult to overload the stream insertion operator `<<`, which for technical reasons cannot be a member function of the `ClockTime` class (see *Howto E*.) Instead, we implement a member function `toString()` that can be used to print a `ClockTime` object. We can then overload the insertion operator `<<` using `toString()`.

9.4.5 Friend Classes

It is not so much our friends' help that helps us as the confident knowledge that they will help us.
Epicurus

Sometimes, however, it is useful for nonmember functions to have access to private data fields. You can design functions (and even other classes) that have access to private data by declaring the functions as **friend** functions (or friend classes). However, granting nonmember functions access to private data violates the principles of encapsulation and information hiding that we've upheld in our programs. You should be very careful if you decide you need to implement a friend function. We'll discuss the syntax for declaring friends in Section 12.3.3. The only use of friends in the code used in this book is to couple a class with its associated iterator, for example, the class `StringSetIterator` is a friend of the class `StringSet` (see Section 6.5.)

In the `ClockTime` class, implementing `toString` and an overloaded operator `+=` makes it possible to implement operator `<<` and operator `+` without making them friend functions. We'll also implement member functions `Less` and `Equal` and use these to implement overloaded relational operators. An in-depth discussion of why we overload operators this way instead of using friend functions is found in *Howto E*.



9.4.6 Overloaded operator <<

If we assume, for the moment, that the `ClockTime::toString` function is implemented, we can easily overload the stream insertion operator as follows.

```
ostream& operator << (ostream & os, const ClockTime & ct)
// precondition: inserts ct onto os, returns os
{
    os << ct.toString();
    return os;
}
```

The `ClockTime` object `ct` is inserted onto the stream `os` and the stream is returned. Returning the stream allows insertion operations to be chained together since the insertion operator is left-associative (see Table A.4 in Howto A.) Using a `toString` member function to overload insertion has two benefits.

- The same method for overloading insertion can be used for any class, and the `toString` function may be useful in other contexts, such as in a debugging environment.
- Using `toString` avoids making the insertion operator a friend function.

The statement below first inserts `ct` onto the stream `cout`, then returns the stream so that the string literal "is the time for fun" can be inserted next.

```
ClockTime ct(1,30,59);
cout << ct << " is the time for fun" << endl;
```

Careful coding in the implementation of `ClockTime::toString()` ensures that 5 seconds is printed as 05 and that 1:02:03 is printed for one hour, two minutes, and three seconds. Two digits are always printed for each number, and a leading zero is added when necessary. The stream manipulator `setw` specifies a field width of 2, and a fill character '0' is specified using the stream function `fill`.

```
string ClockTime::toString() const
{
    ostringstream os;
    os.fill('0');
    os << Hours() << ":" << setw(2) << Minutes() << ":"
    << setw(2) << Seconds();
    return os.str();
}
```

Because we use an `ostringstream` variable it's fine to set the fill character to '0'. If we were using `cout`, for example, we couldn't set the fill character to '0' and leave it that way since users won't expect the fill character to change (e.g., from the default fill character space) just by printing a `ClockTime` object. Details on setting and resetting the fill character can be found in Howto B.



9.4.7 Overloaded Relational Operators



The relational operators `<`, `<=`, `==`, `!=`, `>`, `>=` should be overloaded for a class as free functions, not as member functions (see [Howto E](#) for details.) As free functions, these operators do not have access to class data which are private. Instead of making the operators friend functions, we'll use member functions `Less` and `Equal` in implementing the relational operators. In fact, only `operator ==` and `operator <` are implemented in terms of class member functions. The other relational operators are implemented in terms of `==` and `<` as follows.

```
bool operator != (const ClockTime& lhs, const ClockTime& rhs)
{   return ! (lhs == rhs);
}
```

```
bool operator > (const ClockTime& lhs, const ClockTime& rhs)
{   return rhs < lhs;
}
```

```
bool operator <= (const ClockTime& lhs, const ClockTime& rhs)
{   return ! (lhs > rhs);
}
```

```
bool operator >= (const ClockTime& lhs, const ClockTime& rhs)
{   return ! (lhs < rhs);
}
```

Using this method to overload operators means we only implement `operator ==` and `operator <` and these implementations are also the same for any class with member functions `Less` and `Equal` (see, for example, `BigInt` and `Date`.)

```
bool operator == (const ClockTime& lhs, const ClockTime& rhs)
// post: returns true iff lhs == rhs
{   return lhs.Equals(rhs);
}
```

```
bool operator < (const ClockTime& lhs, const ClockTime& rhs)
// post: returns true iff lhs < rhs
{   return lhs.Less(rhs);
}
```

9.4.8 Overloaded operator `+` and `+=`

When implementing arithmetic operators, it is much simpler to implement `operator +=` first and then call `+=` when implementing `operator +`. Just as using `toString` made it simple to overload `operator <<`, using `operator +=` makes it simple to overload `operator +` for any class that has both operators.

```

ClockTime operator + (const ClockTime & lhs,
                    const ClockTime & rhs)
// postcondition: return lhs + rhs
{
    ClockTime result(lhs);
    result += rhs;
    return result;
}

```

To execute the statement `lhs + rhs` using this implementation a copy of `lhs` is made, the value of `rhs` added to the copy, and the result returned. Compare this implementation, for example, to `operator +` for the `Date` class in `date.cpp` (see Howto G) – the bodies of the functions are identical.

The implementation of `operator +=` is straightforward; we add values and normalize.

```

ClockTime & ClockTime::operator += (const ClockTime & ct)
// postcondition: add ct, return result (normalized)
{
    mySeconds += ct.mySeconds;
    myMinutes += ct.myMinutes;
    myHours   += ct.myHours;
    Normalize();

    return *this;
}

```



For now, we'll ignore the return type of `ClockTime&` and the last statement `return *this`. These are explained in detail in Howto E. If you overload

Syntax: operator +=

```

const ClassName &
operator += (const ClassName& rhs)
{
    implementation
    return *this;
}

```

any of the arithmetic assignment operators you should have the same statement to return a value: `return *this;`. The return type should be a const reference to the class, such as `const ClockTime&`. The same syntax is used for any of the arithmetic assignment operators, such as `*=`, `-=`, `/=`, and `%=`. The implementation changes, but the format of the overloaded function does not.

9.4.9 Testing the ClockTime Class

Before proceeding with the development of the program to manipulate CDs, we must test the `ClockTime` class. In testing the program we'll look for cases that might cause problems such as adding 59 seconds and 1 second. It may seem like too much work to develop a program just to test a class, but this kind of work pays dividends in the

long run. By constructing a simple test program it's possible to debug a class rather than debug a larger application program. This will make the development of the client program easier as well, because (we hope) the class will be correct.

In the sample run following this program, a complete set of test data is not used. You should think about developing a set of test data that would test important boundary cases.

Program 9.10 useclock.cpp

```
#include <iostream>
using namespace std;

#include "clockt.h"

// test program for ClockTime class

int main()
{
    int h,m,s;
    cout << "enter two sets of 'h m s' data " << endl
         << "Enter non integers to terminate program." << endl << endl;

    while (cin >> h >> m >> s)
    {
        ClockTime a(s,m,h);
        cin >> h >> m >> s;
        ClockTime b(s,m,h);
        ClockTime c = a + b;

        cout << a << " + " << b << " = " << c << endl;
    }
    return 0;
}
```

useclock.cpp

OUTPUT

```
prompt> useclock
enter two sets of 'h m s' data
Enter nonintegers to terminate program.

1 40 20 1 15 40
1:40:20 + 1:15:40 = 2:56:00
0 59 59 0 0 1
0:59:59 + 0:00:01 = 1:00:00
0 0 89 0 0 91
0:01:29 + 0:01:31 = 0:03:00
done done done
```

9.4.10 The Final Program

Each track for a CD is stored in the following format:

```
4:19    Have I Told You Lately
2:46    Here Comes The Night
```

Because white space is used to delimit strings when reading input using the extraction operator `>>`, we'll need to use `getline` to read the title of a CD track, since the number of words is different for each track. We'll also use the optional third parameter of `getline` to signal a sentinel other than newline when we read the minutes and seconds that make up the time of a CD track. We'll read all the characters up to the `':'` as the minutes, then all the characters up to a space as the seconds. The remaining characters on a line are the track's title. Since `getline` reads strings, we'll convert the strings for minutes and seconds to integers using the function `atoi` from `strutils.h`.¹²

The third parameter for `getline` has a default value of `'\n'`. This means that if no value is specified for the third parameter, a **default value** of `'\n'` is used.

Program 9.11 cdsum.cpp

```
#include <iostream>
#include <fstream>           // for ifstream
#include <cstdlib>           // for exit
#include <string>
using namespace std;

#include "strutils.h"        // for atoi
#include "clockt.h"
#include "prompt.h"

// reads file containing data for a cd in format below (one line/track)
//   min:sec title
// and sums all track times

int main()
{
    ifstream input;
    string filename = PromptString("enter name of data file: ");
    input.open(filename.c_str());

    if (input.fail())
    {
        cerr << "could not open file " << filename << endl;
        exit(0);
    }
    string minutes,          // # of minutes of track
           seconds,         // # of seconds of track
           title;           // title of track
    ClockTime total(0,0,0); // total of all times
```

¹²`atoi`, read as “a two i,” stands for “alphabetic to integer.”

```

while (getline(input,minutes,':') &&
      getline(input,seconds,' ') &&
      getline(input,title))          // reading line ok
{
    ClockTime track(atoi(seconds.c_str()),atoi(minutes.c_str()),0);
    cout << track << " " << title << endl;
    total += track;
}
cout << "-----" << endl;
cout << "total = " << total << endl;
return 0;
}

```

cdsum.cpp

O U T P U T

```

prompt> cdsum
enter name of data file vanmor.dat
0:04:31    Moondance
0:02:40    Baby Please Don't Go
0:04:19    Have I Told You Lately
0:03:04    Brown Eyed Girl
0:04:21    Sweet Thing
0:03:22    Warm Love
0:03:57    Wonderful Remark
0:02:57    Jackie Wilson Said
0:03:14    Full Force gail
0:04:28    And It Stoned Me
0:02:46    Here Comes The Night
0:03:04    Domino
0:04:05    Did Ye Get Healed
0:03:32    Wild Night
0:04:40    Cleaning Windows
0:04:54    Whenever God Shines His Light
0:04:54    Queen Of The Slipstream
0:04:44    Dweller On The Threshold
-----
total = 1:15:54

```

If you review the specification for `getline`, you'll see that the sentinel is read but is not stored as part of the string `minutes`. The second `getline` uses a space to delimit the number of seconds from the title. Finally, the third use of `getline` relies on the default value of the second parameter: a newline `'\n'`.

The function `atoi` converts a string to the corresponding integer. If the string parameter does not represent a valid integer, then zero is returned.

Pause to Reflect



- 9.18** In *cdsum.cpp*, Program 9.11, the title read includes leading white space if there is more than one space between the track duration and the title. Explain why this is and describe a method for removing the leading white space from the title.
- 9.19** Provide three sets of data that could be used with *useclock.cpp*, Program 9.10, to test the `ClockTime` implementation.
- 9.20** Explain why the `ClockTime` parameters for operators `<<`, `+`, and `+=` are declared as `const` reference parameters.
- 9.21** What is output by the statement `cout << ct << endl` after each of the following definitions?
- `ClockTime ct(71,16,1);`
 - `ClockTime ct(5,62,1);`
 - `ClockTime ct(12);`
 - `ClockTime ct(21,5);`
 - `ClockTime ct;`
- 9.22** If operators `--` and `-` are implemented for subtracting clock times, which one is easiest to implement? Write an implementation for operator `--`.
- 9.23** After reading the number of minutes using `getline(input, minutes)`, is it possible to replace the expression `getline(input, seconds)` with `input >> seconds`? What if `seconds` is defined as an `int` rather than as a `string`?

9.5 Chapter Review

In this chapter we discussed details of streams and characters and how these abstractions are implemented in C++. We also discussed operator overloading and how it makes it simpler to use some classes by mirroring how the built-in types work. We saw that a low-level understanding of how strings and streams are implemented is not necessary in order to use them in programs.

The following are some of the important topics covered:

- The type `char` represents characters and is used to construct strings and streams. Most systems use ASCII as a way of encoding characters, but you should try to write code that is independent of any particular character set.
- The library `<cctype>` has prototypes for several functions that can be used to write programs that do not depend on a particular character set such as ASCII.
- Except for output and use in strings, `char` variables can be thought of as `int` variables. In particular, it's possible to add 3 to `'a'` and subtract `'a'` from `'z'`.

- String variables are composed of `char` values. Individual characters of a string are accessible using `[]`, the indexing operator. The standard string class does no range-checking for illegal index values.
- The function `getline` is used to read an entire line of text and doesn't use white space to delimit one word from another. The sentinel indicating end of line is an optional third parameter.
- The function `get` reads one character at a time from a stream.
- String streams, variables of type `istream` or `ostream`, are useful in reading line-oriented data and in writing to a string, respectively.
- State machines can be useful when parsing data one character at a time or even one word at a time.
- Enumerated types, or *enums*, are useful as symbolic labels. When possible, enums should be declared within a class.
- A friend class (or function) has access to another class's private data members. Friendship must be granted by the class whose private data members will be accessed.
- A class data invariant is a property of class state that is always true after each member function has executed. Using class invariants helps develop and reason about a class's implementation and use.
- It is possible to overload relational, arithmetic, and I/O operators for classes that you write. A set of guidelines for implementing overloaded operators helps makes coding them a straightforward process.
- Class development and testing should be done together, with testing helping the development process and increasing confidence in the correctness of the class implementation (whether the class works as it should).

9.6 Exercises

- 9.1** Modify *decomment.cpp*, Program 9.7, so that removed comments are output to a separate file. Use string functions so that the name of the output file has a `.ncm` (for no comments) suffix with the same prefix as the input file. For example, if the comments are removed from *frogwalk.cpp*, the removed comments will be stored in *frogwalk.ncm*. Each comment should be preceded by the line number from which it was removed. For example:

```
3 // author: Naomi Smith
4 // written 4/5/93
10 // update the counter here, watch out for overflow
37 // avoid iterating too many times
```

9.2 Add two new operators to the `ClockTime` class and develop a test program to ensure that the operators work correctly.

- `operator -` for subtraction of two times. Here it's clear that

```
03:02:05 - 02:01:03 == 01:01:02
```

but you'll have to make a decision about what `0:01:03 - 0:02:05` means.

- `operator >>` to read from a stream. It's probably easiest to read first into a string, and then convert the string to a `ClockTime` value.

9.3 Modify Program 9.4, `readnums.cpp`, so that all integers on a line are parsed and added to `total` but nonintegers are ignored. You'll need to change the type of the variable `num` to `string`. If you use the function `atoi`, it will be difficult to determine when an integer is read and when a noninteger string such as "apple" is read since `atoi("apple")` returns zero. However, all valid integers in C++ begin with either a +, a -, or a digit 0–9.

9.4 Write a program that acts as a spell-checker. The program should prompt the user for a filename and check each word in the file. Possible misspellings should be reported for each line with a misspelled word, where the first line in a file is line number one. Print the line number and the entire line, and use the caret symbol to "underline" the word as shown below. Each line should appear only once in the output, with each misspelled word in the line underlined.

```
20: This is a basic spell chekc program.
      ^^^^^
31: There are more thngs in heven and earth,
      ^^^^^ ^^^^^
```

To tell if a word is misspelled, read a file of words from an on-line list of words (see `words.dat` that comes with the files for this book.) This won't be perfect because of plurals and other endings that typically aren't recorded in word lists, but the program will be a start towards a functioning spell checker. Store the list of words in a `StringSet` object and use the method `StringSet::find()` to search for a match.

For extra credit, when a word ends with 's' and is judged as misspelled, look up the word without the 's' to see if it's a possible plural.

9.5 Write a program to generate junk mail (or spam, the electronic equivalent of junk mail). The program should read two files.

- A template file for the junk mail letter; see `spam.dat` below.
- A data file of names, addresses, and other information used to fill in the template.

For each line of the data file a spam message should be generated. In each message, one line of the template file should generate one line of output, with any entry `<n>` of the template file filled in by the `n`th item in the line of the data file (where the first item in a data file has number zero.)

At first you should write the junk letters to `cout`. However, the user should have the option of creating an output file for each entry in the data file. The output files should

be named *0.spm*, *1.spm*, and so on. Each output file has a *.spm* suffix, and the name of the file is the number corresponding to the line in the data file that generated the spam output.

A template file looks like *spam.dat* below.

```
Dear <0> <1> ,
```

```
<0> , as you know, the <1> family name is one
of the few names to be treasured by family name experts.
The branch of the family in <4> is certainly one of
the best of the <1> families. <0> , your own family
in <3> is in the unique position to lead the world
of <1> families as we approach the year 2000.
```

```
For only $100.00, <0> , you can be be one of the
top <1>-family leaders.
```

A corresponding data file for this template follows.

```
John:Doe:26 Main St:Los Alamos:NM:jdoe@aol.com
Susan:Smith:103 Oak St:St. Louis:MO:sues@hotmail.com
Fred:O'Hare:22 Post Rd, #3:Apex:NC:froh@mindspring.com
```

The second line from this data file generates the following message, the linebreaks can change depending on how you write the program.

```
Dear Susan Smith,
```

```
Susan, as you know, the Smith family name is one
of the few names to be treasured by family name experts.
The branch of the family in MO is certainly one of
the best of the Smith families. Susan, your own family
in St. Louis is in the unique position to lead the world
of Smith families as we approach the year 2000.
```

```
For only $100.00, Susan, you can be be one of the
top Smith-family leaders.
```

9.6 Write a program that reads a text file and creates a pig-latin version of the file. The output (piglatin) file should have the same name as the input file, but with a *.pig* suffix added so that *poe.txt* would be written as *poe.txt.pig* (or, for extra credit, replace any suffix with *.pig*). You can write lines to the output file in one of two ways:

- Each output line corresponds to an input line, but each word on the output line is the pig-latin form of the corresponding word on the input line.
- Write at most 80 characters to each output line (or some other number of characters). Put as many words on a line as possible, without exceeding the 80-character limit, and then start a new line.

The first method is easier, but the lines will be long because each word grows a suffix

in its pig-latin form. The lines below could be translated as shown.

```
It was the best of times, it was the worst of times,
it was the age of wisdom, it was the age of foolishness
```

Here's the translation, with end-of-word punctuation preserved, although the line breaks are not preserved.

```
Itway asway ethay estbay ofway imestay, itway asway ethay
orstway ofway imestay, itway asway ethay ageway ofway
isdomyway, itway asway ethay ageway ofway oolishnesssfay
```

You'll need to be careful if you want to preserve punctuation. As a first step, don't worry about punctuation at all. See the exercises in Chapter 4 for the definition of pig-latin used in the example above.

- 9.7** Design, implement, and test a class `Rational` for fractions, or rational numbers, like $2/3$ and $7/11$. Overload operators so that fractions can be added, subtracted, multiplied, and divided. Use `BigInt` values for the (private state) numerator and denominator. You should also write functions so that rational numbers can be printed. You'll need to write a normalizing function to reduce fractions to lowest terms (i.e., so that $\frac{1}{4} + \frac{1}{4} = \frac{1}{2}$). See Euclid's algorithm for finding greatest common divisors in the exercises of Chapter 5 for help with the normalizing function.

You'll want to include at least three different constructors.

```
Rational r(2,3); // r represents 2/3
Rational t(3); // t represents 3/1
Rational s; // s represents 1/1
```

You'll also need to decide what to do if the user uses 0 for the denominator since this isn't a valid fraction.

- 9.8** Design, implement, and test a class for complex numbers. A complex number has the form $a + b \times i$, where $i = \sqrt{-1}$. Implement constructors from `double` values and from complex values. Overload arithmetic and output operators for complex numbers.
- 9.9** Design, implement, and test a class `Poly` for polynomials. A polynomial has the form $a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0$. For example:

$$(3x^4 + 2x^2 + x - 5) + (2x^4 + 3x^3 - 2x^2 + 5) = (5x^4 + 3x^3 + x)$$

You'll need to use a vector to store exponents and coefficients. You should implement a constructor that takes a coefficient and an exponent as arguments so that you can write

```
Poly c = Poly(3,4) + Poly(2,2) + Poly(7,1) + Poly(-5,0);
```

To get the polynomial $3x^4 + 2x^2 + 7x - 5$. You should overload arithmetic operators `+=`, `-=` and `+`, `-` for addition and subtraction. You should overload `*` to multiply a polynomial by a constant: $3 \times (2x^3 - 3x) = 6x^3 - 9x$.

Finally, you should include a member function `at`, that evaluates a polynomial at a specific value for `x`. For example

```
Poly c = Poly(4,2)+Poly(3,1)+Poly(5,0); // 4x^2 + 3x + 5
```

```
double d = c.at(7); // d = 232 = 4*7^2 + 3*7 + 5
```

- 9.10** Write a program that reads a file and generates an output file with the same words as the input file, but with a maximum of n characters per line, where n is entered by the user. The first version of the program should read words (white space delimited characters) and put as many words on a line as possible, without exceeding n chars per line. In the output file, each word on a line is separated from other words by one space. The file transforms input as follows.

```
'Well, I'll eat it,' said Alice, 'and if it makes me
grow larger, I can reach the key; and if it makes me
grow smaller, I can creep under the door; so either way
I'll get into the garden, and I don't
care which happens!'
```

This is transformed as shown below for $n = 30$.

```
'Well, I'll eat it,' said
Alice, 'and if it makes me
grow larger, I can reach the
key; and if it makes me grow
smaller, I can creep under the
door; so either way I'll get
into the garden, and I don't
care which happens!'
```

Once this version works, the user should have the option of right-justifying each line. Here the lines are padded with extra white space so that each line contains exactly n characters. Extra spaces should be inserted between words, starting at the left of the line and inserting spaces between each pair of words until the line is justified. If adding one space between each word isn't enough to justify the line, continue adding spaces until the line is justified.

```
'Well, I'll eat it,' said
Alice, 'and if it makes me
grow larger, I can reach the
key; and if it makes me grow
smaller, I can creep under the
door; so either way I'll get
into the garden, and I don't
care      which      happens!'
```

- 9.11** Write a program to play hangman. In hangman one player thinks of a word and the other tries to guess the word by guessing one letter at a time. The guesser is allowed a fixed number of missed letters, such as 6; if the word is not guessed before 6 misses, the guesser loses. Traditionally each missed letter results in one more part being added to the figure of a person being hanged, as shown in Figure 9.3. When the figure is complete, the guesser loses. Sample output is shown after Figure 9.3.

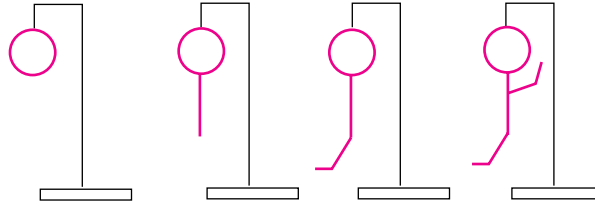


Figure 9.3 Slowly losing at hangman.

```

                                O U T P U T
prompt> hangman
# misses left = 6  word = * * * * *
enter a letter: e
# misses left = 6  word = * * * E * * * * E *
enter a letter: a
# misses left = 5  word = * * * E * * * * E *
enter a letter: i
# misses left = 4  word = * * * E * * * * E *
enter a letter: r
# misses left = 4  word = * * R E * * * * E *
enter a letter: o
# misses left = 3  word = * * R E * * * * E *
enter a letter: n
# misses left = 3  word = * * R E N * * * E N
enter a letter: t
# misses left = 3  word = * T R E N * T * E N
enter a letter: l
# misses left = 2  word = * T R E N * T * E N
enter a letter: u
# misses left = 1  word = * T R E N * T * E N
enter a letter: p
YOU LOSE!!! The word is STRENGTHEN

```

Rather than use graphics (although if you have access to a graphics library, you should try to use it), the program should tell the user how many misses are left and should print a schematic representation of what letters have been guessed correctly. You should try to design and implement a program that uses several classes. Some are suggested here, but you're free to develop scenarios, list nouns for classes and verbs for methods, and develop your own classes.

- class `WordSource` is the source of the secret word the user tries to guess. This class at first could return the same word every time, but eventually it should read

a file (like a file of good hangman words or an on-line dictionary) and return one of the words at random. The same word should not be chosen twice during one run of the program.

- class `Letters` represents the letters the user has guessed (and the unguessed letters). The user might be shown a list of unguessed letters before each guess, or might request such a list as an option. Guessing an already-guessed letter should not count against the user. The case of a letter should not matter so that 'e' and 'E' are treated as the same letter.
- class `Word` represents the word the user is trying to guess (it's initialized from a `WordSource`.) Instead of using `string`, this class encapsulates the word being guessed. The class `Word` might have the following methods (and others).
 - `Display` writes the word with spaces or asterisks (or something else) for unguessed letters. See the sample output for an example. This function could write to `cout` or return a `string`.
 - `ProcessChar` process a character the user guesses. If the character is in the word, it will be displayed by the next call of `Display`. Perhaps this function should return a boolean value indicating if the character is in the word.
- class `Painting` (or `Gallows` for the macabre) is responsible for showing progress in some format as the game progresses. This class might draw a picture, simply display the number of misses and how many remain, or it might use a completely different approach that's not quite as gruesome as hanging someone (be creative).