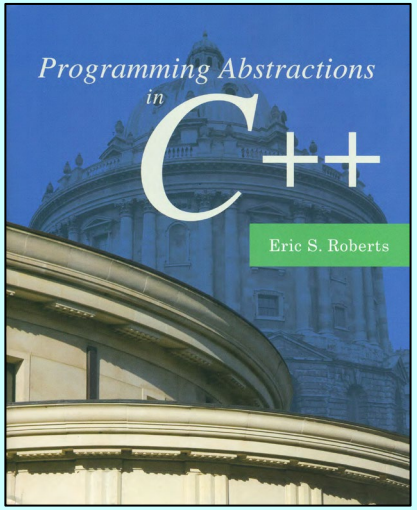
CHAPTER 4

Streams

We will not be satisfied until justice rolls down like waters and righteousness like a mighty stream.

—The Reverend Martin Luther King, Jr.,
I Have a Dream, August 28, 1963
(paraphrasing Amos 5:24)



Introduction to the C++ Standard Libraries

- A collection of ***classes*** and ***functions***, which are written in the core language and part of the C++ ISO Standard itself. Features of the C++ Standard Library are declared within the `std` ***namespace***
  - Containers: vector, queue, stack, map, set, etc.
  - General: algorithm, functional, iterator, memory, etc.
  - Strings
  - Streams and Input/Output: iostream, fstream, sstream, etc.
  - Localization
  - Language support
  - Thread support library
  - Numerics library
  - C standard library: cmath, cctype, cstring, cstdio, cstdlib, etc.

Using <cstdio> (stdio.h) Interface

Things that you only need to know that you don't know. I don't need you to memorize it for the exam.

File access:

fclose	Close file (function)
fflush	Flush stream (function)
fopen	Open file (function)
freopen	Reopen stream with different file or mode (funct
setbuf	Set stream buffer (function)
setvbuf	Change stream buffering (function)

Formatted input/output:

fprintf	Write formatted data to stream (function)
fscanf	Read formatted data from stream (function)
printf	Print formatted data to stdout (function)
scanf	Read formatted data from stdin (function)
snprintf C++11	Write formatted output to sized buffer (function)
sprintf	Write formatted data to string (function)
sscanf	Read formatted data from string (function)
vfprintf	Write formatted data from variable argument list to stream (function)
vfscanf C++11	Read formatted data from stream into variable argument list (function)
vprintf	Print formatted data from variable argument list to stdout (function)
vscanf C++11	Read formatted data into variable argument list (function)
vsnprintf C++11	Write formatted data from variable argument list to sized buffer (function)
vsprintf	Write formatted data from variable argument list to string (function)
vsscanf C++11	Read formatted data from string into variable argument list (function)

Using print

```
#include <c
#include <i
#include <i
using names

int main()
{
    char c =
    int i =
    double d
    // C sty
    printf("
    printf("
    printf("
    // C++ s
    cout << "This is a character: " << c << endl;
    cout << "This is an integer: " << i << endl;
    cout << "This is a double: " << fixed
         << setprecision(2) << d << endl;
    return 0;
}
```

specifier	Output	Example
d *or* i	Signed decimal integer	392
u	Unsigned decimal integer	7235
o	Unsigned octal	610
x	Unsigned hexadecimal integer	7fa
X	Unsigned hexadecimal integer (uppercase)	7FA
f	Decimal floating point, lowercase	392.65
F	Decimal floating point, uppercase	392.65
e	Scientific notation (mantissa/exponent), lowercase	3.9265e+2
E	Scientific notation (mantissa/exponent), uppercase	3.9265E+2
g	Use the shortest representation: %e or %f	392.65
G	Use the shortest representation: %E or %F	392.65
a	Hexadecimal floating point, lowercase	-0xc.90fep-2
A	Hexadecimal floating point, uppercase	-0XC.90FEP-2
c	Character	a
s	String of characters	sample
p	Pointer address	b8000000
n	Nothing printed. The corresponding argument must be a pointer to a signed int. The number of characters written so far is stored in the pointed location.	
%	A % followed by another % character will write a single % to the stream.	%

C++ Streams

- Ever since the very first program in the text, which displayed the message **`"hello, world"`** on the screen, the programs in this text have made use of an important object called a ***stream***, which C++ uses to manage the flow of information to or from some data source.
- In the earlier chapters, you have used the **`<<`** and **`>>`** operators and have already had occasion to use the three standard streams exported by the **`<iostream>`** library: **`cin`**, **`cout`**, and **`cerr`**.
- Later, to implement file-processing applications, we will learn the notion of data files.
- Finally, we explore the structure of the C++ ***stream*** classes as a representative example of inheritance hierarchies in an object-oriented language.

C++ Streams

- A ***stream*** is an abstraction (object) that represents an input source or output destination of characters of indefinite length, on which input and output operations can be performed.
- Streams are often associated to a physical source or destination of characters, like a disk file (file streams), the keyboard (standard input stream **`cin`**), or the console (standard output stream **`cout`**), so the characters gotten from or written to our abstraction called stream are physically input from or output to the corresponding physical device.
- For example, file streams are C++ objects to manipulate and interact with files. Once a file stream is used to open a file, any input or output operation performed on that stream is physically reflected in the file.

Standard Output Stream `cout` and the Insertion Operator `<<`

- The standard technique to specify formatted output in C++ uses the ***insertion operator***, which is written as `<<`. This operator takes an output stream on the left and an expression of any type on its right. The effect is to write the value of the expression to the output stream using the current format settings.
- The insertion operator returns the output stream as its result. The advantage of this interpretation is that output operations can be chained together, as in the following statement:

```
cout << "The total is " << total << endl;
```

- C++ allows you to control the output by including items in the output chain called ***manipulators*** (`<iomanip>`), which affect the way how subsequent values are formatted. A list of the most common output manipulators appears on the next slide.

Output Manipulators

`endl`	Moves cursor to the next line.
`setw(`*n*`)`**	Sets the width of the next value to *n* characters.
`setprecision(`*digits*`)`**	Sets how many digits should appear.
`setfill(`*ch*`)`**	Sets the fill character used to pad values.
`left`	Aligns the value at the left edge of the field.
`right`	Aligns the value at the right edge of the field.
`fixed`	Sets fixed-point output (no scientific notation).
`scientific`	Sets scientific-notation output.
`showpoint/noshowpoint`	Controls whether a decimal point must appear.
`showpos/noshowpos`	Controls appearance of a plus sign.
`uppercase/nouppercase`	Controls whether uppercase is used in hex.
`boolalpha/noboolalpha`	Controls whether **`bool`**s appear as **`true`**/**`false`**.

Formatting output

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

const double PI = 314.159265358979323846;

int main() {
   cout << fixed << setprecision(20) << PI << endl;
   cout << fixed << setprecision(6) << PI << endl;
   cout << scientific << setprecision(15) << PI << endl;
   cout << uppercase << scientific << setprecision(6) << PI << endl;
   cout << fixed << setw(16) << setprecision(9) << PI << endl;
   cout << fixed << setfill('0') << setw(10) << setprecision(4) << PI << endl;
   cout << fixed << setfill('0') << setw(10) << setprecision(4) << left << PI << endl;
   return 0;
}
```

```
314.15926535897932581065
314.159265
3.141592653589793e+02
3.141593E+02
   314.159265359
00314.1593
314.159300
```

The closest floating-point number with the specified precision.

You might see `002` *in the exponent in some compilers/systems.*

Precision Example

```
const double PI = 3.14159265358979323846;
const double SPEED_OF_LIGHT = 2.99792458E+8;
const double FINE_STRUCTURE = 7.2573525E-3;

int main() {
   cout << uppercase << right;
   cout << "Default format:" << endl << endl;
   printPrecisionTable();
   cout << endl << "Fixed format:" << fixed << endl << endl;
   printPrecisionTable();
   cout << endl << "Scientific format:" << scientific << endl << endl;
   printPrecisionTable();
   return 0;
}

void printPrecisionTable() {
   cout << " prec |      pi        |  speed of light  | fine structure" << endl;
   cout << "------+----------------+------------------+---------------" << endl;
   for (int prec = 0; prec <= 6; prec += 2) {
      cout << setw(4) << prec << "  |";
      cout << " " << setw(13) << setprecision(prec) << PI << " |";
      cout << " " << setw(16) << setprecision(prec) << SPEED_OF_LIGHT << " |";
      cout << " " << setw(14) << setprecision(prec) << FINE_STRUCTURE << endl;
   }
}
```

Precision Example

PrecisionExample

```
Default format:

 prec |      pi
------+------------
    0 |           3
    2 |         3.1
    4 |       3.142
    6 |     3.14159

Fixed format:

 prec |      pi      |  speed of light   | fine structure
------+--------------+-------------------+---------------
    0 |           3  |         299792458 |             0
    2 |        3.14  |      299792458.00 |          0.01
    4 |      3.1416  |    299792458.0000 |        0.0073
    6 |    3.141593  | 299792458.000000  |      0.007257

Scientific format:

 prec |      pi      |  speed of light   | fine structure
------+--------------+-------------------+---------------
    0 |        3E+00 |           3E+08   |          7E-03
    2 |     3.14E+00 |        3.00E+08   |       7.26E-03
    4 |   3.1416E+00 |      2.9979E+08   |     7.2574E-03
    6 | 3.141593E+00 |    2.997925E+08   |   7.257352E-03
```

Default format is the more compact representation between **`fixed`** *and* **`scientific`**, *chosen by C++.* **`setprecision(`***digits***`)`** *indicates the number of significant digits for default format and specifies the number of digits after the decimal point otherwise.*

Standard Input Stream `cin` and the Extraction Operator `>>`

- For input, C++ includes the `>>` operator, which is called the ***extraction operator***. The `>>` operator is symmetrical to the `<<` operator and reads formatted data from the stream on the left into the variables that appear on the right.
- Up to now, you have used the `>>` operator to request input values from the console (`PowerOfTwo`):

```
int limit;
cout << "Enter exponent limit: ";
cin >> limit;
```

- Similar to the `<<` operator, you can use ***manipulators*** to affect the way how subsequent values are formatted:

```
char ch;
cout << "Enter a single character: ";
cin >> noskipws >> ch;
```

Input Manipulators

TABLE 4-2 **Input manipulators**

skipws **noskipws**	These manipulators control whether the extraction operator skips over whitespace characters before reading a value. If you specify **noskipws**, the extraction operator treats all characters (including whitespace characters) as part of the input field. You can later use **skipws** to restore the default behavior. This property is persistent.
ws	Reads characters from the input stream until some character appears that is not in the whitespace category. The effect of this manipulator is therefore to skip over any spaces, tabs, and newlines in the input. Unlike **skipws** and **noskipws**, which change the behavior of the stream for subsequent input operations, the **ws** manipulator takes effect immediately.

- Exercise: type in `' '' ''A'' ''B'`, and what is the output?

```
char a, b, c;
cout << "Enter at least 3 character: ";
cin >> noskipws >> a >> ws >> b >> skipws >> c;
cout << a << b << c << endl;
```

Using <cstdio> (stdio.h) Interface

File access:

fclose	Close file (function)
fflush	Flush stream (function)
fopen	Open file (function)
freopen	Reopen stream with different file or mode (function)
setbuf	Set stream buffer (function)
setvbuf	Change stream buffering (function)

Formatted input/output:

fprintf	
fscanf	
printf	
scanf	
snprintf C++11	Write formatted output to sized buffer (function)
sprintf	Write formatted data to string (function)
sscanf	Read formatted data from string (function)
vfprintf	Write formatted data from variable argument list to stream (function)
vfscanf C++11	Read formatted data from stream into variable argument list (function)
vprintf	Print formatted data from variable argument list to stdout (function)
vscanf C++11	Read formatted data into variable argument list (function)
vsnprintf C++11	Write formatted data from variable argument list to sized buffer (function)
vsprintf	Write formatted data from variable argument list to string (function)
vsscanf C++11	Read formatted data from string into variable argument list (function)

Like the relationship between **cout** *and* **<<** *and* **printf**, *you are encouraged to use* **cin** *and* **>>** *in C++, but should know about* **scanf** *in C.*

Data Files

- A ***file*** is the generic name for any named collection of data maintained on the various types of ***permanent*** storage media attached to a computer. In most cases, a file is stored on a hard disk, but it can also be stored on removable medium, such as a CD or flash memory drive.
- Files can contain information of many different types. When you compile a C++ program, for example, the compiler stores its output in an ***object file*** containing the binary representation of the program. The most common type of file, however, is probably a ***text file***, which contains character data of the sort you find in a string.
- Files can be both a source associated to an input stream and a destination associated to an output stream.

Using Text Files

- When you want to read data from a text file as part of a C++ program, you need to take the following steps:
  1. Construct a new **`ifstream`** (i.e., input file stream) object that is tied to the data in the file by declaring a stream variable to refer to the file.
  2. Call the **`open`** method for the stream. This phase of the process is called ***opening the file***. For historical reasons, the argument to **`open`** is a C string literal rather than a C++ string object.
  3. Call the methods provided by the **`ifstream`** class to read data from the file in sequential order. The text of the file can be read in several ways, including character by character or line by line.
  4. Break the association between the reader and the file by calling the stream's **`close`** method, which is called ***closing the file***. This is important because before you close the file, no one else can use the file.

Using Text Files

- Question: Why does it seem so complicated to process a file?

  Files live in the external storage, and are exposed to all programs. You have to open (associate a stream living in your program to) a file before you can use it. While you are accessing a file through a stream, other programs might not be allowed to access the file to avoid conflicts. You have to close (untie the stream to) a file after you use it so that it can be used by others.

- Question: How to write data to a text file?

Using Text Files

Methods supported by all streams

stream`.fail()`	Returns `true` if the stream is in a failure state. This condition usually occurs when you try to read data past the end of the file, but may also indicate an integrity error in the data.
stream`.eof()`	Returns `true` if the stream is positioned at the end of the file. Given the semantics of the C++ stream library, the `eof` method is useful *only* after a call to `fail`. At that point, calling `eof` allows you to test whether the failure indication was caused by the end of file or some other data error.
stream`.clear()`	Resets the status bits associated with the stream. You need to call this method whenever you need to reuse a stream after a failure has occurred.
`if (`*stream*`) . . .`	If you use a stream in a conditional context, C++ interprets it as a test of whether the stream is valid. For the most part, this test has the same effect as calling `if (!`*stream*`.fail())`.

Methods supported by all file streams

stream`.open(`*filename*`)`	Attempts to open the named file and attach it to the stream. The direction is determined by the stream type: input streams are opened for input, output streams are opened for output. The *filename* parameter is a C-style string, which means that you will need to call `c_str` on any C++ string. You can check whether the `open` method fails by calling `fail`.
stream`.close()`	Closes the file attached to the stream.

Using Text Files

Methods supported by all input streams

stream **>>** *variable*	Reads formatted data into a variable. The data format is controlled by the variable type and whatever input manipulators are in effect.
stream`.get(`*var*`)`	Reads the next character into the character variable *var,* which is passed by reference. The return value is the stream itself, with the **`fail`** flag set if there are no more characters.
stream`.get()`	Returns the next character in the stream. The return value is an integer, which makes it possible to identify the end-of-file character, which is represented by the constant **`EOF`**.
stream`.unget()`	Backs up the internal pointer of the stream so that the last character read will be read again by the next call to **`get`**.
`getline(`*stream*`,` *str*`)`	Reads the next line of input from *stream* into the string *str*. The **`getline`** function returns the stream, which simplifies the end-of-file test.

a free function belonging to <string>

Methods supported by all output streams

stream **<<** *expression*	Writes formatted data to an output stream. The data format is controlled by the expression type and whatever output manipulators are in effect.
stream`.put(`*ch*`)`	Writes the character *ch* to the output stream.

Using Text Files (dangerously)

- What does the following program do? And why?

```
#include <iostream>
#include <fstream>
#include <string>
#include <cctype>
using namespace std;
int main() {
   string filename;
   fstream file;
   char c;
   cout << "Input file name: ";
   cin >> filename;
   file.open(filename.c_str());
   file.get(c); // or c = file.get();
   c = toupper(c);
   file.put(c);
   file.close();
   return 0;
}
```

Does this program uppercase the first character in a file?

- Data in files are usually read and written sequentially.

Reading Characters

- You can read characters from an input stream by calling the **get** method, which comes in two forms:
  - If you supply no arguments, **get()** reads and returns the next character value as an **int**, which is **EOF** at the end of the file.
  - If you instead pass a character variable by reference, **get(ch)** reads the next character into that variable. This form of **get** returns a value that acts like **false** at the end of the file.

```
int ch;                                          // int get();
while (          ch          != EOF) {
    . . . Perform the necessary operations using the character . . .
}
```

```
char ch;                                         // istream& get(char& c);
while ( !infile.fail() ) {
    . . . Perform the necessary operations using the character . . .
}
```

Type casting a stream to bool

- The second is less conventional but typically more convenient.

Reading a File Character by Character

```
/*
 * File: ShowFileContents.cpp
 * --------------------------
 * This program displays the contents of a file chosen by the user.
 */

#include <iostream>
#include <fstream>
#include <string>
#include "filelib.h"
using namespace std;

int main() {
   ifstream infile;
   promptUserForFile(infile, "Input file: ");
   int ch;
   while ((ch = infile.get()) != EOF) {
      cout.put(ch);
   }
   infile.close();
   return 0;
}
```

comes from the Stanford C++ Library

Reading a File Character by Character

```
/*
 * File: ShowFileContents.cpp
 * --------------------------
 * This program displays the contents of a file chosen by the user.
 */

#include <iostream>
#include <fstream>
#include <string>
#include "filelib.h"
using namespace std;

int main() {
   ifstream infile;
   promptUserForFile(infile, "Input file: ");
   char ch;
   while (infile.get(ch)) {
      cout.put(ch);
   }
   infile.close();
   return 0;
}
```

Reading Lines from a File

- You can also read lines from a text file by calling the ***free function*** (unlike a ***method***, a free function is not bound to a particular class) **`getline`**, which takes an **`ifstream`** and a **`string`** as reference parameters.
- The effect of **`getline`** is to store the next line of data from the file into the string variable after discarding the end-of-line character.
- If you try to read past the end of the data, **`getline`** sets the ***fail indicator*** for the stream, which is then interpreted as **`false`**.
- The following code fragment uses the **`getline`** method to…?

```
...
int max = 0;
string line;
while (getline(infile, line)) {
   if (line.length() > max) max = line.length();
}
```

determine the length of the longest line in the stream **`infile`**.

Reading a File Line by Line

```
/*
 * File: ShowFileContents.cpp
 * --------------------------
 * This program displays the contents of a file chosen by the user.
 */

#include <iostream>
#include <fstream>
#include <string>
#include "filelib.h"
using namespace std;

int main() {
   ifstream infile;
   promptUserForFile(infile, "Input file: ");
   string line;
   while (getline(infile, line)) {
      cout << line << endl;
   }
   infile.close();
   return 0;
}
```

The Hello Name Program

Flashback

FIGURE 3-1 An interactive version of the "Hello World" program

```
/*
 * File: HelloName.cpp
 * -------------------
 * This program extends the classic "Hello world" program by asking
 * the user for a name, which is then used as part of the greeting.
 * This version of the program reads a complete line into name and
 * not just the first word.
 */

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

int main() {
   string name;
   cout << "Enter your full name: ";
   cin >> name;
   cout << "Hello, " << name << "!" << endl;
   return 0;
}
```

getline

- The ***free function*** **`std::getline`** is defined in **`<string>`**, and not only works on input file stream but other input streams:

```
istream& getline(istream& is, string& str, char delim);
istream& getline(istream& is, string& str);
```

- There are actually another overloaded version of **`getline`** in C++. It is the ***member method*** of the **`istream`** class, i.e., **`std::istream::getline`**

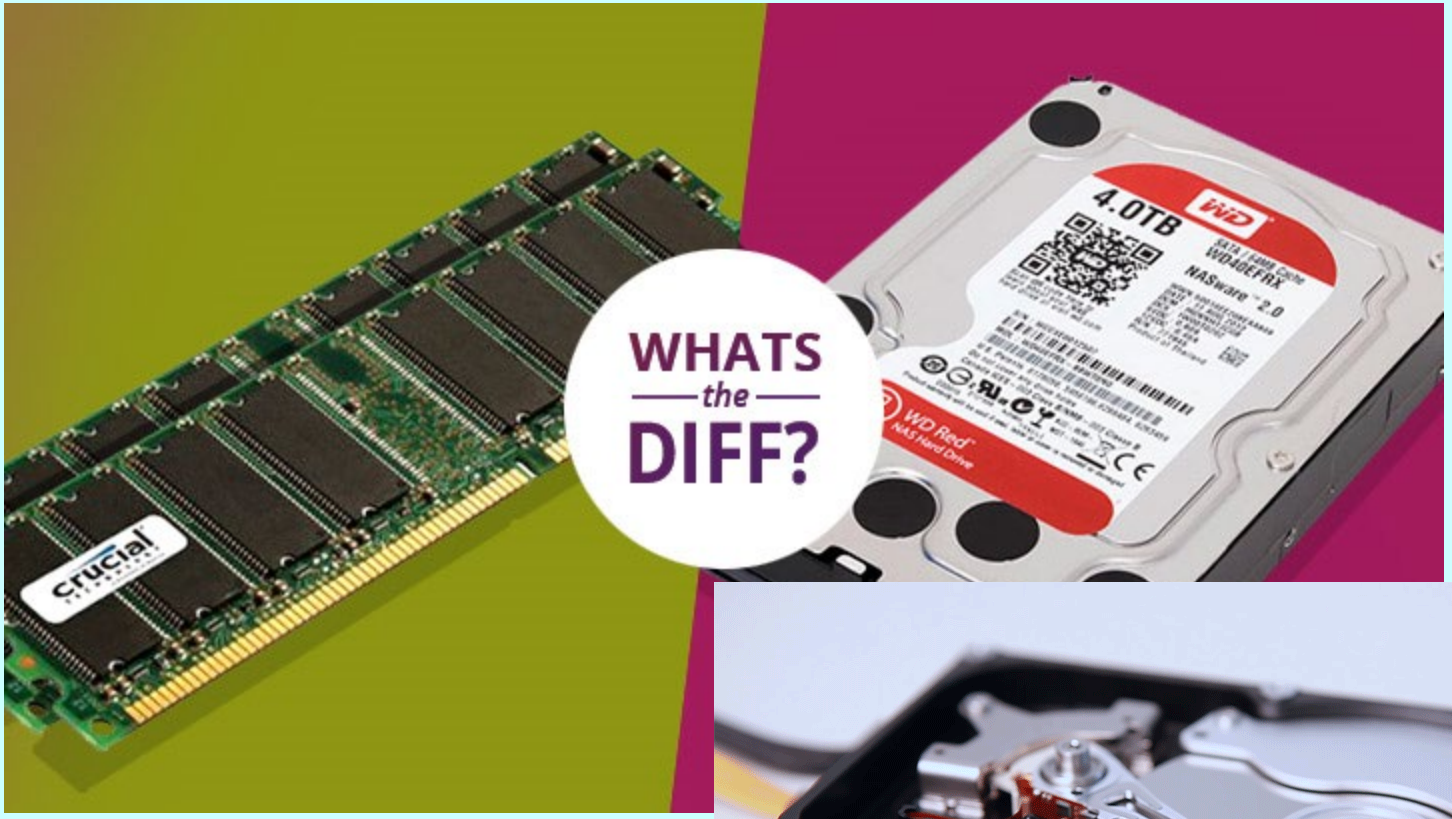
```
istream& getline(char* s, streamsize n);
istream& getline(char* s, streamsize n, char delim);
```

- Question: Why is there **`istream`** in the parameter list of the first version of **`getline`** but not in the second version?

Text Files *vs.* Strings

- Although text files and strings both contain character data, it is important to keep in mind the following important differences between text files and strings:
  - *The information stored in a file is permanent.* Information stored in a file exists in the external storage until the file is deleted. Strings only live in a certain program while files are exposed to all programs. The value of a string variable persists only as long as the variable does. Local variables disappear when the method returns, and instance variables disappear when the object goes away, which typically does not occur until the program exits.
  - *Data in files are usually accessed sequentially.* When you read data from a file, you usually start at the beginning and read the characters in order, either individually or in groups that are most commonly individual lines. Once you have read one set of characters, you then move on to the next set of characters until you reach the end of the file.

RAM *vs.* Storage



String Streams

- Given that files and strings are both sequences of characters, C++ allows you to treat them symmetrically.
- C++ provides that capability through the **`<sstream>`** library, which exports several classes that allow you to associate a stream with a string value in much the same way that the **`<fstream>`** library allows you to associate a stream with a file.
- The **`istringstream`** class is the counterpart of **`ifstream`** and makes it possible to use stream operators to read data from a string.
- For output, the **`ostringstream`** class works very much like **`ofstream`** except that the output is directed to a string rather than a file.
- A **string** is a collection of characters, a **stream** is an object to manipulate a flow of data (e.g., characters), and a **string stream** is a special stream object that lets you use a string as the source and destination of the flow of data.

String Streams

FIGURE 4-4 Function to read an integer from the console

```
/*
 * Function: getInteger
 * Usage: int n = getInteger(prompt);
 * ----------------------------------
 * Requests an integer value from the user.  The function begins by
 * printing the prompt string on the console and then waits for the
 * user to enter a line of input data.  If that line contains a
 * single integer, the function returns the corresponding integer
 * value.  If the input is not a legal integer or if extraneous
 * characters (other than whitespace) appear on the input line,
 * the implementation gives the user a chance to reenter the value.
 */

int getInteger(string prompt) {
   int value;
   string line;
   while (true) {
      cout << prompt;
      getline(cin, line);
      istringstream stream(line);
      stream >> value >> ws;
      if (!stream.fail() && stream.eof()) break;
      cout << "Illegal integer format. Try again." << endl;
   }
   return value;
}
```

String Streams

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

int main()
{
    int i;
    cin >> i;
    cout << "My favorite number is: " << i << endl;
    string s = "My favorite number is: " + to_string(i);
    cout << s << endl;
    stringstream ss;
    ss << "My favorite number is: " << i;
    s = ss.str();
    cout << s << endl;
}
```

Only available since C++ 11.

Using <cstdio> (stdio.h) Interface

File access:

fclose	Close file (function)
fflush	Flush stream (function)
fopen	Open file (function)
freopen	Reopen stream with different file or mode (function)
setbuf	Set stream buffer (function)
setvbuf	Change stream buffering (function)

Formatted input/output:

fprintf	Write formatted data to stream (function)
fscanf	Read formatted data from stream (function)
printf	Print formatted data to stdout (function)
scanf	Read formatted data from stdin (function)
snprintf C++11	Write formatted output to sized buffer (function)
sprintf	Write formatted data to string (function)
sscanf	Read formatted data from string (function)
vfprintf	Write formatted data from variable argument list to stream (function)
vfscanf C++11	Read formatted data from stream into variable argument list (function)
vprintf	Print formatted data from variable argument list to stdout (function)
vscanf C++11	Read formatted data into variable argument list (function)
vsnprintf C++11	Write formatted data from variable argument list to sized buffer (function)
vsprintf	Write formatted data from variable argument list to string (function)
vsscanf C++11	Read formatted data from string into variable argument list (function)

The `simpio.h` Interface

https://web.stanford.edu/dept/cs_edu/cppdoc/simpio.html

simpio.h

simpio.h

This interface exports a set of functions that simplify input/output operations in C++ and provide some error-checking on console input.

Functions

`getInteger(prompt)`	Reads a complete line from **`cin`** and tries to scan it as an integer.
`getReal(prompt)`	Reads a complete line from **`cin`** and tries to scan it as a floating-point number.
`getLine(prompt)`	Reads a line of text from **`cin`** and returns that line as a string.

Function detail

```
int getInteger(string prompt = "");
```

Reads a complete line from **`cin`** and scans it as an integer. If the scan succeeds, the integer value is returned. If the argument is not a legal integer or if extraneous characters (other than whitespace) appear in the string, the user is given a chance to reenter the value. If supplied, the optional **`prompt`** string is printed before reading the value.

Usage:

```
int n = get
```

Compare this to the standard C++ function:

```
istream& getline(istream& is, string& str);
```

```
double getReal(string prompt = "");
```

The `filelib.h` Interface

https://web.stanford.edu/dept/cs_edu/cppdoc/filelib.html

filelib.h

`filelib.h`

This file exports a standardized set of tools for working with files. The library offers at least some portability across the file systems used in the three supported platforms: Mac OSX, Windows, and Linux. Directory and search paths are allowed to contain separators in any of the supported styles, which usually makes it possible to use the same code on different platforms.

Functions

`createDirectory(path)`	Creates a new directory for the specified path.
`createDirectoryPath(path)`	Creates a new directory for the specified path.
`defaultExtension(filename, ext)`	Adds an extension to a file name if none already exists.
`deleteFile(filename)`	Deletes the specified file.
`expandPathname(filename)`	Expands a filename into a canonical name for the platform.
`fileExists(filename)`	Returns **`true`** if the specified file exists.
`findOnPath(path, filename)`	Returns the canonical name of a file found using a search path.
`getCurrentDirectory()`	Returns an absolute filename for the current directory.
`getDirectoryPathSeparator()`	Returns the standard directory path separator used on this platform.
`getExtension(filename)`	Returns the extension of **`filename`**.

Web Documentation for `filelib.h`

The `StanfordCPPLib` package

`filelib.h`

Function Detail

```
bool openFile(ifstream & stream, string filename);
bool openFile(ofstream & stream, string filename);
```

Opens the filestream **`stream`** using the specified **`filename`**. This function is similar to the **`open`** method of the stream classes, but uses a C++ **`string`** object instead of the older C-style string. If the operation succeeds, **`openFile`** returns **`true`**; if it fails, **`openFile`** sets the failure flag in the stream and returns **`false`**.

Usage:

```
if (openFile(stream, filename)) ...
```

Compare this to the standard C++ function:

```
void open(const char* filename);
```

```
string promptUserForFile(ifstream & stream, string prompt = "");
string promptUserForFile(ofstream & stream, string prompt = "");
```

Asks the user for the name of a file. The file is opened using the reference parameter **`stream`**, and the function returns the name of the file. If the requested file cannot be opened, the user is given additional chances to enter a valid file. The optional **`prompt`** argument provides an input prompt for the user.

Usage:

```
string filename = promptUserForFile(stream, prompt);
```

Opening an Input File

```
/*
 * File: filelib.h
 * ---------------
 * This file exports a standardized set of tools for working with
 * files . . .
 */

#ifndef _filelib_h
#define _filelib_h

/*
 * Function: promptUserForFile
 * Usage: string filename = promptUserForFile(stream, prompt);
 * -----------------------------------------------------------
 * Asks the user for the name of a file.  The file is opened
 * using the reference parameter stream, and the function
 * returns the name of the file.  If the requested file cannot
 * be opened, the user is given additional chances to enter a
 * valid file.  The optional prompt argument provides an input
 * prompt for the user.
 */
```

Opening an Input File

```
string promptUserForFile(ifstream & stream, string prompt) {
   while (true) {
      string filename;
      filename = getLine(prompt);
      openFile(stream, filename);
      if (!stream.fail()) return filename;
      stream.clear();
      cout << "Unable to open that file.  Try again." << endl;
      if (prompt == "") prompt = "Input file: ";
   }
}
```

Instead of using the more advanced **`getLine`** and **`openFile`** from Stanford **`filelib.h`**, you can simply use the standard **`getline`** for strings and **`open`** for file streams.

```
cout << prompt;
// cin >> filename; // usually only get a word
getline(cin, filename); // get a whole line of input
stream.open(filename.c_str());
```

The End