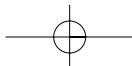
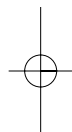
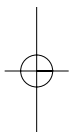
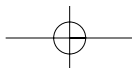


Part I

Basics

The four chapters that follow will provide the fundamental knowledge you need to write networking applications in Perl using Berkeley sockets. They set the stage for later parts of the book that delve more deeply into specific network problems and their solutions.



Chapter 1

Input/Output Basics

This chapter provides you with the background information you'll need to write TCP/IP applications in Perl. We review Perl's input/output (I/O) system using the language's built-in function calls, and then using the object-oriented (OO) extensions of Perl5. This will prepare you to use the object-oriented constructions in later chapters.

Perl and Networking

Why would you want to write networking applications in Perl?

The Internet is based on Transmission Control Protocol/Internet Protocol (TCP/IP), and most networking applications are based on a straightforward application programming interface (API) to the protocol known as Berkeley sockets. The success of TCP/IP is due partly to the ubiquity of the sockets API, which is available for all major languages including C, C++, Java, BASIC, Python, COBOL, Pascal, FORTRAN, and, of course, Perl. The sockets API is similar in all these languages. There may be a lot of work involved porting a networking application from one computer language to another, but porting the part that does the socket communications is usually the least of your problems.

For dedicated Perl programmers, the answer to the question that starts this chapter is clear—because you can! But for those who are not already members of the choir, one can make a convincing argument that not only is networking good for Perl, but Perl is good for networking.

A Language Built for Interprocess Communication

Perl was built from the ground up to make it easy to do interprocess communication (the thing that happens when one program talks to another). As we shall see later in this chapter, in Perl there is very little difference between opening up

4 Chapter 1 Input/Output Basics

a local file for reading and opening up a communications channel to read data from another local program. With only a little more work, you can open up a socket to read data from a program running remotely on another machine somewhere on the Internet. Once the communications channel is open, it matters little whether the thing at the other end is a file, a program running on the same machine, or a program running on a remote machine. Perl's input/output functions work in the same way for all three types of connections.

A Language Built for Text Processing

Another Perl feature that makes it good for network applications is its powerful integrated regular expression-matching and text-processing facilities. Much of the data on the Internet is text based (the Web, for instance), and a good portion of that is unpredictable, line-oriented data. Perl excels at manipulating this type of data, and is not vulnerable to the type of buffer overflow and memory overrun errors that make networking applications difficult to write (and possibly insecure) in languages like C and C++.

An Open Source Project

Perl is an Open Source project, one of the earliest. Examining other people's source code is the best way to figure out how to do something. Not only is the source code for all of Perl's networking modules available, but the whole source tree for the interpreter itself is available for your perusal. Another benefit of Perl's openness is that the project is open to any developer who wishes to contribute to the library modules or to the interpreter source code. This means that Perl adds features very rapidly, yet is stable and relatively bug free.

The universe of third-party Perl modules is available via a distributed Web-based archive called CPAN, for Comprehensive Perl Archive Network. You can search CPAN for modules of interest, download and install them, and contribute your own modules to the archive. The preface to this book describes CPAN and how to reach it.

Object-Oriented Networking Extensions

Perl5 has object-oriented extensions, and although OO purists may express dismay over the fast and loose way in which Perl has implemented these features, it is inarguable that the OO syntax can dramatically increase the readability and maintainability of certain applications. Nowhere is this more evident than in the library modules that provide a high-level interface to networking protocols. Among many others, the `IO::Socket` modules provide a clean and elegant interface to Berkeley sockets; `Mail::Internet` provides cross-platform access to Internet mail; `LWP` gives you everything you need to write Web clients; and

the Net::FTP and Net::Telnet modules let you write interfaces to these important protocols.

Security

Security is an important aspect of network application development, because by definition a network application allows a process running on a remote machine to affect its execution. Perl has some features that increase the security of network applications relative to other languages. Because of its dynamic memory management, Perl avoids the buffer overflows that lead to most of the security holes in C and other compiled languages. Of equal importance, Perl implements a powerful “taint” check system that prevents tainted data obtained from the network from being used in operations such as opening files for writing and executing system commands, which could be dangerous.

Performance

A last issue is performance. As an interpreted language, Perl applications run several times more slowly than C and other compiled languages, and about par with Java and Python. In most networking applications, however, raw performance is not the issue; the I/O bottleneck is. On I/O-bound applications Perl runs just as fast (or as slowly) as a compiled program. In fact, it's possible for the performance of a Perl script to exceed that of a compiled program. Benchmarks of a simple Perl-based Web server that we develop in Chapter 12 are several times better than the C-based Apache Web server.

If execution speed does become an issue, Perl provides a facility for rewriting time-critical portions of your application in C, using the XS extension system. Or you can treat Perl as a prototyping language, and implement the real application in C or C++ after you've worked out the architectural and protocol details.

Networking Made Easy

Before we get into details, let's look at two simple programs.

The *lgetl.pl* script (for “line get local,” Figure 1.1) reads the first line of a local file. Call it with the path to the file you want to read, and it will print out the top line. For example, here's what I see when I run the script on a file that contains a quote from James Hogan's “Giants Star”:

```
% lgetl.pl giants_star.txt
"Reintegration complete," ZORAC advised. "We're back in the universe."
```

This snippet illustrates the typographic conventions this book uses for terminal (command-line interpreter) sessions. The “%” character is the prompt printed out by my command-line interpreter. Bold-faced text is what I (the user) typed. Everything else is regular monospaced font.

6 Chapter 1 Input/Output Basics

Figure 1.1: *lgetl.pl*—Read the first line of a local file

```

0  #!/usr/bin/perl
1  # file: lgetl.pl

2  use IO::File;

3  my $file = shift;
4  my $fh   = IO::File->new($file);
5  my $line = <$fh>;
6  print $line;

```

The script itself is straightforward:

Lines 1–2: Load modules We use() the IO::File module, which wraps an object-oriented interface around Perl file operations.

Line 3: Process the command line argument We shift() the filename off the command line and store it in a variable named \$file.

Line 4: Open the file We call the IO::File->new() method to open the file, returning a filehandle, which we store in \$fh. Don't worry if the OO syntax is unfamiliar to you; we discuss it more later in this chapter.

Lines 5–6: Read a line from the filehandle and print it We use the <> operator to read a line of text from the filehandle into the variable \$line, which we immediately print.

Now we'll look at a very similar script named *lgetr.pl* (for “line get remote,” Figure 1.2). It too fetches and prints a line of text, but instead of reading from a local file, this one reads from a remote server. Its command-line argument is the name of a remote host followed by a colon and the name of the network service you want to access.

To read a line of text from the “daytime” service running on the FTP server *wuarchive.wustl.edu*, we use an argument of “wuarchive.wustl.edu:daytime.” This retrieves the current time of day at the remote site:

```

% lgetr.pl wuarchive.wustl.edu:daytime
Tue Aug 8 06:49:20 2000

```

Figure 1.2: *lgetr.pl*—Read the first line from a remote server

```

0  #!/usr/bin/perl
1  # file: lgetr.pl

2  use IO::Socket;

3  my $server = shift;
4  my $fh     = IO::Socket::INET->new($server);
5  my $line   = <$fh>;
6  print $line;

```

To read the welcome banner from the FTP service at the same site, we ask for “wuarchive.wustl.edu:ftp”:

```
% lgetr.pl wuarchive.wustl.edu:ftp
220 wuarchive.wustl.edu FTP server (Version wu-2.6.1(1) Thu Jul 13
21:24:09 CDT 2000) ready.
```

Or for a change of hosts, we can read the welcome banner from the SMTP (Internet mail) server running at *mail.hotmail.com* like this:

```
% lgetr.pl mail.hotmail.com:smtp
220-HotMail (NO UCE) ESMTP server ready at Tue Aug 08 05:24:40 2000
```

Let’s turn to the code for the *lgetr.pl* script in Figure 1.2.

Lines 1–2: Load modules We use() the IO::Socket module, which provides an object-oriented interface for network socket operations.

Line 3: Process the command line argument We shift() the host and service name off the command line and store it in a variable named \$server.

Line 4: Open a socket We call the IO::Socket::INET->new() method to create a “socket” connected to the designated service running on the remote machine. IO::Socket::INET is a filehandle class that is adapted for Internet-based communications. A socket is just a specialized form of filehandle, and can be used interchangeably with other types of filehandles in I/O operations.

Lines 5–6: Read a line from the socket and print it We use the <> operator to read a line of text from the socket into the variable \$line, which we immediately print.

Feel free to try the *lgetr.pl* script on your favorite servers. In addition to the services used in the examples above, other services to try include “nntp,” the Netnews transfer protocol, “chargen,” a test character generator, and “pop3,” a protocol for retrieving mail messages. If the script appears to hang indefinitely, you’ve probably contacted a service that requires the client to send the first line of text, such as an HTTP (Web) server. Just interrupt the script and try a different service name.

Although *lgetr.pl* doesn’t do all that much, it is useful in its own right. You can use it to check the time on a remote machine, or wrap it in a shell script to check the time synchronization of all the servers on your network. You could use it to generate a summary of the machines on your network that are running an SMTP mail server and the software they’re using.

Notice the similarity between the two scripts. Simply by changing IO::File->new() to IO::Socket::INET->new(), we have created a fully functional network client. Such is the power of Perl.

Filehandles

Filehandles are the foundation of networked applications. In this section we review the ins and outs of filehandles. Even if you’re an experienced Perl

8 Chapter 1 Input/Output Basics

programmer, you might want to scan this section to refresh your memory on some of the more obscure aspects of Perl I/O.

Standard Filehandles

A filehandle connects a Perl script to the outside world. Reading from a filehandle brings in outside data, and writing to one exports data. Depending on how it was created, a filehandle may be connected to a disk file, to a hardware device such as a serial port, to a local process such as a command-line window in a windowing system, or to a remote process such as a network server. It's also possible for a filehandle to be connected to a "bit bucket" device that just sucks up data and ignores it.

A filehandle is any valid Perl identifier that consists of uppercase and lowercase letters, digits, and the underscore character. Unlike other variables, a filehandle does not have a distinctive prefix (like "\$"). So to make them distinct, Perl programmers often represent them in all capital letters, or caps.

When a Perl script starts, exactly three filehandles are open by default: `STDOUT`, `STDIN`, and `STDERR`. The `STDOUT` filehandle, for "standard output," is the default filehandle for output. Data sent to this filehandle appears on the user's preferred output device, usually the command-line window from which the script was launched. `STDIN`, for "standard input," is the default input filehandle. Data read from this filehandle is taken from the user's preferred input device, usually the keyboard. `STDERR` ("standard error") is used for error messages, diagnostics, debugging, and other such incidental output. By default `STDERR` uses the same output device as `STDOUT`, but this can be changed at the user's discretion. The reason that there are separate filehandles for normal and abnormal output is so that the user can divert them independently; for example, to send normal output to a file and error output to the screen.

This code fragment will read a line of input from `STDIN`, remove the terminating end-of-line character with the `chomp()` function, and echo it to standard output:

```
$input = <STDIN>;
chomp($input);
print STDOUT "If I heard you correctly, you said: $input\n";
```

By taking advantage of the fact that `STDIN` and `STDOUT` are the defaults for many I/O operations, and by combining `chomp()` with the input operation, the same code could be written more succinctly like this:

```
chomp($input = <>);
print "If I heard you correctly, you said: $input\n";
```

We review the `<>` and `print()` functions in the next section. Similarly, `STDERR` is the default destination for the `warn()` and `die()` functions.

The user can change the attachment of the three standard filehandles before launching the script. On UNIX and Windows systems, this is done using the redirect metacharacters “<” and “>”. For example, given a script named *muncher.pl* this command will change the script’s standard input so that it comes from the file *data.txt*, and its standard output so that processed data ends up in *crunched.txt*:

```
% muncher.pl <data.txt >crunched.txt
```

Standard error isn’t changed, so diagnostic messages (e.g., from the built-in `warn()` and `die()` functions) appear on the screen.

On Macintosh systems, users can change the source of the three standard filehandles by selecting filenames from a dialog box within the MacPerl development environment.

Input and Output Operations

Perl gives you the option of reading from a filehandle one line at a time, suitable for text files, or reading from it in chunks of arbitrary length, suitable for binary byte streams like image files.

For input, the `<>` operator is used to read from a filehandle in a line-oriented fashion, and `read()` or `sysread()` to read in a byte-stream fashion. For output, `print()` and `syswrite()` are used for both text and binary data (you decide whether to make the output line-oriented by printing newlines).

```
$line = <FILEHANDLE>
@lines = <FILEHANDLE>
$line = <>
@lines = <>
```

The `<>` (“angle bracket”) operator is sensitive to the context in which it is called. If it is used to assign to a scalar variable, a so-called scalar context, it reads a line of text from the indicated filehandle, returning the data along with its terminating end-of-line character. After reading the last line of the filehandle, `<>` will return `undef`, signaling the end-of-file (EOF) condition.

When `<>` is assigned to an array or used in another place where Perl ordinarily expects a list, it reads all lines from the filehandle through to EOF, returning them as one (potentially gigantic) list. This is called a list context.

If called in a “void context” (i.e., without being assigned to a variable), `<>` copies a line into the `$_` global variable. This is commonly seen in `while()` loops, and often combined with pattern matches and other operations that use `$_` implicitly:

```
while (<>) {
    print "Found a gnu\n" if /GNU/i;
}
```

The `<FILEHANDLE>` form of this function explicitly gives the filehandle to read from. However, the `<>` form is “magical.” If the script was called with a set of file names as

10 Chapter 1 Input/Output Basics

command-line arguments, `<>` will attempt to `open()` each argument in turn and will then return lines from them as if they were concatenated into one large pseudofile.

If no files are given on the command line, or if a single file named “-” is given, then `<>` reads from standard input and is equivalent to `<STDIN>`. See the `perlfunc` POD documentation for an explanation of how this works (`pod perlfunc`, as explained in the Preface).

`$bytes = read (FILEHANDLE,$buffer,$length [,$offset])`

`$bytes = sysread (FILEHANDLE,$buffer,$length [,$offset])`

The `read()` and `sysread()` functions read data of arbitrary length from the indicated filehandle. Up to `$length` bytes of data will be read, and placed in the `$buffer` scalar variable. Both functions return the number of bytes actually read, numeric 0 on the end of file, or `undef` on an error.

This code fragment will attempt to read 50 bytes of data from `STDIN`, placing the information in `$buffer`, and assigning the number of bytes read to `$bytes`:

```
my $buffer;
$bytes = read (STDIN,$buffer,50);
```

By default, the read data will be placed at the beginning of `$buffer`, overwriting whatever was already there. You can change this behavior by providing the optional numeric `$offset` argument, to specify that read data should be written into the variable starting at the specified position.

The main difference between `read()` and `sysread()` is that `read()` uses standard I/O buffering, and `sysread()` does not. This means that `read()` will not return until either it can fetch the exact number of bytes requested or it hits the end of file. The `sysread()` function, in contrast, can return partial reads. It is guaranteed to return at least 1 byte, but if it cannot immediately read the number of bytes requested from the filehandle, it will return what it can. This behavior is discussed in more detail later in the Buffering and Blocking section.

`$result = print FILEHANDLE $data1,$data2,$data3...`

`$result = print $data1,$data2,$data3...`

The `print()` function prints a list of data items to a filehandle. In the first form, the filehandle is given explicitly. Notice that there is no comma between the filehandle name and the first data item. In the second form, `print()` uses the current default filehandle, usually `STDOUT`. The default filehandle can be changed using the one-argument form of `select()` (discussed below). If no data arguments are provided, then `print()` prints the contents of `$_`.

If output was successful, `print()` returns a true value. Otherwise it returns false and leaves an error message in the variable named `$!`.

Perl is a parentheses-optional language. Although I prefer using parentheses around function arguments, most Perl scripts drop them with `print()`, and this book follows that convention as well.

`$result = printf $format,$data1,$data2,$data3...`

The `printf()` function is a formatted print. The indicated data items are formatted and printed according to the `$format` format string. The formatting language is quite

rich, and is explained in detail in Perl's POD documentation for the related `sprintf()` (string formatting) function.

\$bytes = syswrite (FILEHANDLE,\$data [,\$length [,\$offset]])

The `syswrite()` function is an alternative way to write to a filehandle that gives you more control over the process. Its arguments are a filehandle and a scalar value (a variable or string literal). It writes the data to the filehandle, and returns the number of bytes successfully written.

By default, `syswrite()` attempts to write the entire contents of `$data`, beginning at the start of the string. You can alter this behavior by providing an optional `$length` and `$offset`, in which case `syswrite()` will write `$length` bytes beginning at the position specified by `$offset`.

Aside from familiarity, the main difference between `print()` and `syswrite()` is that the former uses standard I/O buffering, while the latter does not. We discuss this later in the Buffering and Blocking section.

Don't confuse `syswrite()` with Perl's unfortunately named `write()` function. The latter is part of Perl's report formatting package, which we won't discuss further.

\$previous = select(FILEHANDLE)

The `select()` function changes the default output filehandle used by `print()`. It takes the name of the filehandle to set as the default, and returns the name of the previous default. There is also a version of `select()` that takes four arguments, which is used for I/O multiplexing. We introduce the four-argument version in Chapter 8.

When reading data as a byte stream with `read()` or `sysread()`, a common idiom is to pass `length($buffer)` as the offset into the buffer. This will make `read()` append the new data to the end of data that was already in the buffer. For example:

```
my $buffer;
while (1) {
    $bytes = read (STDIN,$buffer,50,length($buffer));
    last unless $bytes > 0;
}
```

Detecting the End of File

The end-of-file condition occurs when there's no more data to be read from a file or device. When reading from files this happens at the literal end of the file, but the EOF condition applies as well when reading from other devices. When reading from the terminal (command-line window), for example, EOF occurs when the user presses a special key: control-D on UNIX, control-Z on Windows/DOS, and command-. on Macintosh. When reading from a network-attached socket, EOF occurs when the remote machine closes its end of the connection.

The EOF condition is signaled differently depending on whether you are reading from the filehandle one line at a time or as a byte stream. For byte-stream

12 Chapter 1 Input/Output Basics

operations with `read()` or `sysread()`, EOF is indicated when the function returns numeric 0. Other I/O errors return `undef` and set `$!` to the appropriate error message. To distinguish between an error and a normal end of file, you can test the return value with `defined()`:

```
while (1) {
    my $bytes = read(STDIN,$buffer,100);
    die "read error" unless defined ($bytes);
    last unless $bytes > 0;
}
```

In contrast, the `<>` operator doesn't distinguish between EOF and abnormal conditions, and returns `undef` in either case. To distinguish them, you can set `$!` to `undef` before performing a series of reads, and check whether it is defined afterward:

```
undef $!;
while (defined(my $line = <STDIN>)) {
    $data .= $line;
}
die "Abnormal read error: $!" if defined ($!);
```

When you are using `<>` inside the conditional of a `while()` loop, as shown in the most recent code fragment, you can dispense with the explicit `defined()` test. This makes the loop easier on the eyes:

```
while (my $line = <STDIN>) {
    $data .= $line;
}
```

This will work even if the line consists of a single 0 or an empty string, which Perl would ordinarily treat as false. Outside `while()` loops, be careful to use `defined()` to test the returned value for EOF.

Finally, there is the `eof()` function, which explicitly tests a filehandle for the EOF condition:

\$eof = eof(FILEHANDLE)

The `eof()` function returns true if the next read on `FILEHANDLE` will return an EOF. Called without arguments or parentheses, as in `eof`, the function tests the last filehandle read from.

When using `while(<>)` to read from the command-line arguments as a single pseudofile, `eof()` has “magical”—or at least confusing—properties. Called with empty parentheses, as in `eof()`, the function returns true at the end of the very last file. Called without parentheses or arguments, as in `eof`, the function returns true at the end of each of the individual files on the command line. See the Perl POD documentation for examples of the circumstances in which this behavior is useful.

In practice, you do not have to use `eof()` except in very special circumstances, and a reliance on it is often a signal that something is amiss in the structure of your program.

Anarchy at the End of the Line

When performing line-oriented I/O, you have to watch for different interpretations of the end-of-line character. No two operating system designers can seem to agree on how lines should end in text files. On UNIX systems, lines end with the linefeed character (LF, octal `\012` in the ASCII table); on Macintosh systems, they end with the carriage return character (CR, octal `\015`); and the Windows/DOS designers decided to end each line of text with two characters, a carriage return/linefeed pair (CRLF, or octal `\015\012`). Most line-oriented network servers also use CRLF to terminate lines.

This leads to endless confusion when moving text files between machines. Fortunately, Perl provides a way to examine and change the end-of-line character. The global variable `$/` contains the current character, or sequence of characters, used to signal the end of line. By default, it is set to `\012` on Unix systems, `\015` on Macintoshes, and `\015\012` on Windows and DOS systems.

The line-oriented `<>` input function will read from the specified handle until it encounters the end-of-line character(s) contained in `$/`, and return the line of text with the end-of-line sequence still attached. The `chomp()` function looks for the end-of-line sequence at the end of a text string and removes it, respecting the current value of `$/`.

The string escape `\n` is the *logical* newline character, and means different things on different platforms. For example, `\n` is equivalent to `\012` on UNIX systems, and to `\015` on Macintoshes. (On Windows systems, `\n` is usually `\012`, but see the later discussion of DOS text mode.) In a similar vein, `\r` is the logical carriage return character, which also varies from system to system.

When communicating with a line-oriented network server that uses CRLF to terminate lines, it won't be portable to set `$/` to `\r\n`. Use the explicit string `\015\012` instead. To make this less obscure, the `Socket` and `IO::Socket` modules, which we discuss in great detail later, have an option to export globals named `$CRLF` and `CRLF()` that return the correct values.

There is an additional complication when performing line-oriented I/O on Microsoft Windows and DOS machines. For historical reasons, Windows/DOS distinguishes between filehandles in "text mode" and those in "binary mode." In binary mode, what you see is exactly what you get. When you print to a binary filehandle, the data is output exactly as you specified. Similarly, read operations return the data exactly as it was stored in the file.

In text mode, however, the standard I/O library automatically translates LF into CRLF pairs on the way out, and CRLF pairs into LF on the way in. The virtue of this is that it makes text operations on Windows and UNIX Perls look the same—from the programmer's point of view, the DOS text files end in a single `\n` character, just as they do in UNIX. The problem one runs into is when reading or writing binary files—such as images or indexed databases—and the files become mysteriously corrupted on input or output. This is due to

14 Chapter 1 Input/Output Basics

the default line-end translation. Should this happen to you, you should turn off character translation by calling `binmode()` on the filehandle.

binmode (FILEHANDLE [,\$discipline])

The `binmode()` function turns on binary mode for a filehandle, disabling character translation. It should be called after the filehandle is opened, but before doing any I/O with it. The single-argument form turns on binary mode. The two-argument form, available only with Perl 5.6 or higher, allows you to turn binary mode on by providing `:raw` as the value of `$discipline`, or restore the default text mode using `:crlf` as the value.

`binmode()` only has an effect on systems like Windows and VMS, where the end-of-line sequence is more than one character. On UNIX and Macintosh systems, it has no effect.

Another way to avoid confusion over text and binary mode is to use the `sysread()` and `syswrite()` functions, which bypass the character translation routines in the standard I/O library.

A whole bevy of special global variables control other aspects of line-oriented I/O, such as whether to append an end-of-line character automatically to data output with the `print()` statement, and whether multiple data values should be separated by a delimiter. See Appendix C for a brief summary.

Opening and Closing Files

In addition to the three standard filehandles, Perl allows you to open any number of additional filehandles. To open a file for reading or writing, use the built-in Perl function `open()`. If successful, `open()` gives you a filehandle to use for the read and/or write operations themselves. Once you are finished with the filehandle, call `close()` to close it. This code fragment illustrates how to open the file *message.txt* for writing, write two lines of text to it, and close it:

```
open (FH,">message.txt") or die "Can't open file: $!";
print FH "This is the first line.\n";
print FH "And this is the second.\n";
close (FH) or die "Can't close file: $!";
```

We call `open()` with two arguments: a filehandle name and the name of the file we wish to open. The filehandle name is any valid Perl identifier consisting of any combination of uppercase and lowercase letters, digits, and the underscore character. To make them distinct, most Perl programmers choose all uppercase letters for filehandles. The ">" symbol in front of the filename tells Perl to overwrite the file's contents if it already exists, or to create the file if it doesn't. The file will then be opened for writing.

If `open()` succeeds, it returns a true value. Otherwise, it returns false, causing Perl to evaluate the expression to the right of the `or` operator. This

expression simply dies with an error message, using Perl's `$!` global variable to retrieve the last system error message encountered.

We call `print()` twice to write some text to the filehandle. The first argument to `print()` is the filehandle, and the second and subsequent arguments are strings to write to the filehandle. Again, notice that there is no comma between the filehandle and the strings to print. Whatever is printed to a filehandle shows up in its corresponding file. If the filehandle argument to `print()` is omitted, it defaults to `STDOUT`.

After we have finished printing, we call `close()` to close the filehandle. `close()` returns a true value if the filehandle was closed uneventfully, or false if some untoward event, such as a disk filling up, occurred. We check this result code using the same type of `or` test we used earlier.

Let's look at `open()` and `close()` in more detail.

`$success = open(FILEHANDLE,$path)`

`$success = open(FILEHANDLE,$mode,$path)`

The `open()` call opens the file given in `$path`, associating it with a designated `FILEHANDLE`. There are both two- and three-argument versions of `open()`. In the three-argument version, which is available in Perl versions 5.6 and higher, a `$mode` argument specifies how the file is to be opened. `$mode` is a one- or two-character string chosen to be reminiscent of the I/O redirection operators in the UNIX and DOS shells. Choices are shown here.

Mode	Description
<	Open file for reading
>	Truncate file to zero length and open for writing
>>	Open file for appending, do not truncate
+>	Truncate file and then open for read/write
<+	Open file for read/write, do not truncate

We can open the file named *darkstar.txt* for reading and associate it with the filehandle `DARKFH` like this:

```
open(DARKFH, '<', 'darkstar.txt');
```

In the two-argument form of `open()`, the mode is appended directly to the filename, as in:

```
open(DARKFH, '<darkstar.txt');
```

For readability, you can put any amount of whitespace between the mode symbol and the filename; it will be ignored. If you leave out the mode symbol,

16 Chapter 1 Input/Output Basics

the file will be opened for reading. Hence the above examples are all equivalent to this:

```
open(DARKFH, 'darkstar.txt');
```

If successful, `open ()` will return a true value. Otherwise it returns false. In the latter case, the `$!` global will contain a human-readable message indicating the cause of the error.

`$success = close(FH);`

The `close ()` function closes a previously opened file, returning true if successful, or false otherwise. In the case of an error, the error message can again be found in `$!`.

When your program exits, any filehandles that are still open will be closed automatically.

The three-argument form of `open ()` is used only rarely. However, it has the virtue of not scanning the filename for special characters the way that the two-argument form does. This lets you open files whose names contain leading or trailing whitespace, ">" characters, and other weird and arbitrary data. The filename "-" is special. When opened for reading, it tells Perl to open standard input. When opened for writing, it tells Perl to open standard output.

If you call `open ()` on a filehandle that is already open, it will be automatically closed and then reopened on the file that you specify. Among other things, this call can be used to reopen one of the three standard filehandles on the file of your choice, changing the default source or destination of the `<>`, `print ()`, and `warn ()` functions. We will see an example of this shortly.

As with the `print ()` function, many programmers drop the parentheses around `open ()` and `close ()`. For example, this is the most common idiom for opening a file:

```
open DARKSTAR, "darkstar.txt" or die "Couldn't open darkstar.txt: $!"
```

I don't like this style much because it leads to visual ambiguity (does the `or` associate with the string "darkstar.txt" or with the `open ()` function?). However, I do use this style with `close ()`, `print ()`, and `return ()` because of their ubiquity.

The two-argument form of `open ()` has a lot of magic associated with it (too much magic, some would say). The full list of magic behavior can be found in the *perlfunc* and *perlopentut* POD documentation. However, one trick is worth noting because we use it in later chapters. You can *duplicate* a filehandle by using it as the second argument to `open ()` with the sequence `>&` or `<&` prepended to the beginning. `>&` duplicates filehandles used for writing, and `<&` duplicates those used for reading:

```
open (OUTCOPY, ">&STDOUT");
open (INCOPY, "<&STDOUT");
```


This example creates a new filehandle named `OUTCOPY` that is attached to the same device as `STDOUT`. You can now write to `OUTCOPY` and it will have the same effect as writing to `STDOUT`. This is useful when you want to replace one or more of the three standard filehandles temporarily, and restore them later. For example, this code fragment will temporarily reopen `STDOUT` onto a file, invoke the system *date* command (using the `system()` function, which we discuss in more detail in Chapter 2), and then restore the previous value of `STDOUT`. When *date* runs, its standard output is opened on the file, and its output appears there rather than in the command window:

```
#!/usr/bin/perl
# file: redirect.pl

print "Redirecting STDOUT\n";
open (SAVEOUT,">&STDOUT");
open (STDOUT,">test.txt") or die "Can't open test.txt: $!";

print "STDOUT is redirected\n";
system "date";

open (STDOUT,">&SAVEOUT");
print "STDOUT restored\n";
```

When this script runs, its output looks like this:

```
% redirect.pl
Redirecting STDOUT
STDOUT restored
```

and the file *test.txt* contains these lines:

```
STDOUT is redirected
Thu Aug 10 09:19:24 EDT 2000
```

Notice how the second `print()` statement and the output of the *date* system command went to the file rather than to the screen because we had reopened `STDOUT` at that point. When we restored `STDOUT` from the copy saved in `SAVEOUT`, our ability to print to the terminal was restored.

Perl also provides an alternative API for opening files that avoids the magic and obscure syntax of `open()` altogether. The `sysopen()` function allows you to open files using the same syntax as the C library's `open()` function.

`$result = sysopen (FILEHANDLE,$filename,$mode [$perms])`

The `sysopen()` function opens the file indicated by `$filename`, using the I/O mode indicated by `$mode`. If the file doesn't exist, and `$mode` indicates that the file should be created, then the optional `$perms` value specifies the permission bits for the newly created file. We discuss I/O modes and permissions in more detail below.

If successful, `sysopen()` returns a true result and associates the opened file with `FILEHANDLE`. Otherwise it returns a false result and leaves the error message in `$!`.

18 Chapter 1 Input/Output Basics

The `$mode` argument used in `sysopen()` is different from the mode used in ordinary `open()`. Instead of being a set of characters, it is a numeric bitmask formed by ORing together one or more constants using the bitwise OR operator `|`. For example, the following snippet opens up a file for writing using a mode that causes it to be created if it doesn't exist, and truncated to length zero if it does (equivalent to `open()`'s `>` mode):

```
sysopen(FH,"darkstar.txt",O_WRONLY|O_CREAT|O_TRUNC)
    or die "Can't open: $!"
```

The standard `Fcntl` module exports the constants recognized by `sysopen()`, all of which begin with the prefix `O_`. Just use `Fcntl` at the top of your script to gain access to them.

The mode constants useful for `sysopen()` are listed in Table 1.1. Each call to `sysopen()` must have one (and only one) of `O_RDONLY`, `O_WRONLY`, and `O_RDWR`. The `O_WRONLY` and `O_RDWR` constants may be ORed with one or more of `O_CREAT`, `O_EXCL`, `O_TRUNC`, or `O_APPEND`.

`O_CREAT` causes the file to be created if it doesn't already exist. If it isn't specified, and the file doesn't exist when you try to open it for writing, then `sysopen()` will fail.

Combining `O_CREAT` with `O_EXCL` leads to the useful behavior of creating the file if it doesn't already exist, but failing if it does. This can be used to avoid accidentally clobbering an existing file.

If `O_TRUNC` is specified, then the file is truncated to zero length before the first write, effectively overwriting the previous contents. `O_APPEND` has the opposite effect, positioning the write pointer to the end of the file so that everything written to the file is appended to its existing contents.

Table 1.1: `sysopen()` Mode Constants

<i>Constant</i>	<i>Description</i>
<code>O_RDONLY</code>	Open read only.
<code>O_WRONLY</code>	Open write only.
<code>O_RDWR</code>	Open read/write.
<code>O_CREAT</code>	Create file if it doesn't exist.
<code>O_EXCL</code>	When combined with <code>O_CREAT</code> , create file if it doesn't exist and fail if it does.
<code>O_TRUNC</code>	If file already exists, truncate it to zero.
<code>O_APPEND</code>	Open file in append mode (equivalent to <code>open()</code> 's <code>>></code>).
<code>O_NOCTTY</code>	If the file is a terminal device, open it without allowing it to become the process's controlling terminal.
<code>O_NONBLOCK</code>	Open file in nonblocking mode.
<code>O_SYNC</code>	Open file for synchronous mode, so that all writes block until the data is physically written.

The `O_NOCTTY`, `O_NONBLOCK`, and `O_SYNC` modes all have specialized uses that are discussed in later chapters.

If `sysopen()` needs to create a file, the `$perm` argument specifies the permissions mode of the resulting file. File permissions is a UNIX concept that maps imperfectly onto the Windows/DOS world, and not at all onto the Macintosh world. It is an octal value, such as `0644` (which happens to specify read/write permissions for the owner of the file, and read-only permissions for others).

If `$perm` is not provided, `sysopen()` defaults to `0666`, which grants read/write access to everyone. However, whether you specify the permissions or accept the default, the actual permissions of the created file are determined by performing the bitwise AND between the `$perm` argument and the current contents of the user's `umask` (another UNIX concept). This is often set, at the user's discretion, to forbid access to the file from outside the user's account or group.

In most circumstances, it is best to omit the permissions argument and let the user adjust the `umask`. This also increases the portability of the program. See the `umask()` entry in the *perlfunc* POD documentation for information on how you can examine and set the `umask` programatically.

Buffering and Blocking

When you `print()` or `syswrite()` to a filehandle, the actual output operation does not occur immediately. If you are writing to a file, the system has to wait for the write head to reach the proper location on the disk drive, and for the spinning platter to bring the proper location under the head. This is usually an insignificant length of time (although it may be quite noticeable on a laptop that intermittently spins down its disk to save battery power), but other output operations can take much more time. In particular, network operations may take a considerable length of time to complete. The same applies to input.

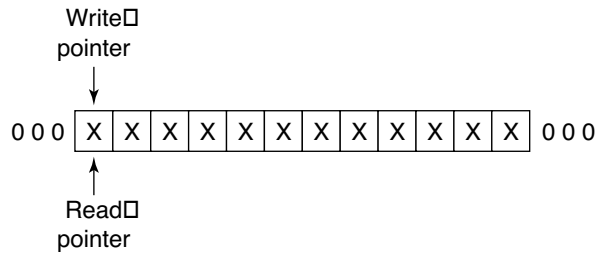
There is a fundamental mismatch between computational speed and I/O speed. A program can execute the contents of a tight loop a million times a second, but a single I/O operation may take several seconds to complete. To overcome this mismatch, modern operating systems use the techniques of buffering and blocking.

The idea behind buffering is shown in Figure 1.3. Buffers decouple the I/O calls invoked by the program from the actual I/O operations that occur at the hardware level. A call to `print()`, for example, doesn't send data directly to the terminal, network card, or disk drive, but instead it results in data being written to a memory area. This occurs quickly, because writes to memory are fast. Meanwhile, in an asynchronous fashion, the operating system reads from data previously written to the buffer and performs the actions necessary to write the information to the hardware device.

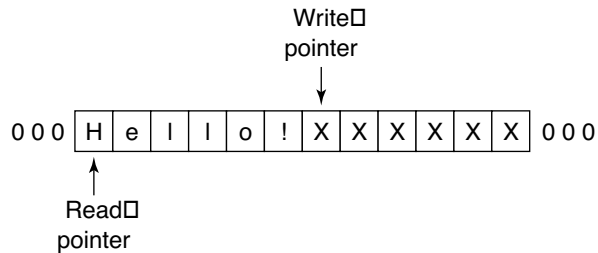
20 Chapter 1 Input/Output Basics

Figure 1.3: Buffers help solve the mismatch between computation speed and I/O speed

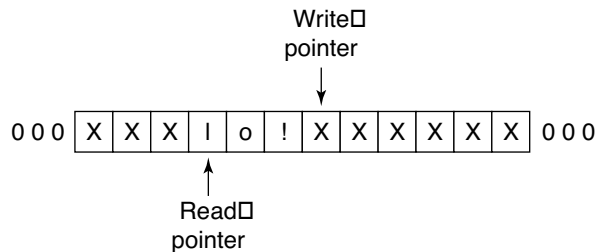
1. I/O buffer starts out empty



2. Program prints "Hello!," advancing write pointer

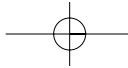


3. Operating system slowly reads from read pointer, advancing it



Similarly, for input operations, the operating system receives data from active input devices (the keyboard, disk drive, network card) and writes that data to an input buffer somewhere in memory. The data remains in the input buffer until your program calls `<>` or `read()`, at which point the data is copied from the operating system's buffer into the memory space corresponding to a variable in your program.

The advantage of buffering is significant, particularly if your program performs I/O in a "bursty" way; that is, it performs numerous reads and writes of unpredictable size and timing. Instead of waiting for each operation to complete at the hardware level, the data is safely buffered in the operating system



and “flushed”—passed on to the output device—whenever the downstream hardware can accept it.

The buffers in Figure 1.3 are conceptually circular FIFO (first in first out) data structures. When data is written beyond the end of the buffer memory area, the operating system merely begins writing new data at the beginning. The operating system maintains two pointers in each of its I/O buffers. The write pointer is the place where new data enters the buffer. The read pointer is the position from which stored data is moved out of the buffer to its next destination. For example, on write operations, each `print()` you perform adds some data to the output buffer and advances the write pointer forward. The operating system reads older data starting at the read pointer and copies it to the low-level hardware device.

The size of the I/O buffer minus the distance between the write pointer and the read pointer is the amount of free space remaining. If your program is writing faster than the output hardware can receive it, then the buffer will eventually fill up and the buffer’s free space will be zero. What happens then?

Because there is no room for new data in the buffer, the output operation cannot succeed immediately. As a result, the write operation blocks. Your program will be suspended at the blocked `print()` or `syswrite()` for an indefinite period of time. When the backlog in the output buffer clears and there is again room to receive new data, the output operation will unblock and `print()` or `syswrite()` will return.

In a similar fashion, reads will block when the input buffer is empty; that is, it blocks when the amount of free space is equal to the size of the buffer. In this case, calls to `read()` or `sysread()` will block until some new data has entered the buffer and there is something to be read.

Blocking is often the behavior you want, but sometimes you need more control over I/O. There are several techniques to manage blocking. One technique, discussed in Chapter 2 under Timing Out System Calls, uses signals to abort an I/O operation prematurely if it takes too long. Another technique, discussed in Chapter 12, uses the four-argument `select()` system call to test a filehandle for its readiness to perform I/O before actually making the read or write call. A third technique, discussed in Chapter 13, is to mark the filehandle as nonblocking, which causes the read or write operation to return immediately with an error code if the operation would block.

Standard I/O Buffering

Although we have spoken of a single buffer for I/O operations on a filehandle, there may in fact be several buffers at different layers of the operating system. For example, when writing to a disk file, there is a very low-level buffer on the disk hardware itself, another one in the SCSI or IDE driver that controls the disk, a third in the driver for the filesystem, and a fourth in the standard C library used by Perl. I may have missed a few.

22 Chapter 1 Input/Output Basics

You cannot control or even access most of these buffers directly, but there is one class of buffer that you should be aware of. Many of Perl's I/O operations flow through "stdio," a standard C-language library which maintains its own I/O buffers independent of the operating system's.¹

Perl's `<>` operator, `read()`, and `print()` all use stdio. When you call `print()`, the data is transferred to an output buffer in the stdio layer before being sent to the operating system itself. Likewise, `<>` and `read()` both read data from an stdio buffer rather than directly from the OS. Each filehandle has its own set of buffers for input and output. For efficiency reasons, stdio waits until its output buffers reach a certain size before flushing their contents to the OS.

Normally, the presence of the stdio buffering is not a problem, but it can run you into trouble when doing more sophisticated types of I/O such as network operations. Consider the common case of an application that requires you to write a small amount of data to a remote server, wait for a response, and then send more data. You may think that the data has been sent across the network, but in fact it may not have been. The output data may still be sitting in its local stdio buffer, waiting for more data to come before flushing the buffer. The remote server will never receive the data, and so will never return a response. Your program will never receive a response and so will never send additional data. Deadlock ensues.

In contrast, the lower-level buffering performed by the operating system doesn't have this property. The OS will always attempt to deliver whatever data is in its output buffers as soon as the hardware can accept it.

There are two ways to work around stdio buffering. One is to turn on "autoflush" mode for the filehandle. Autoflush mode applies only to output operations. When active, Perl tells stdio to flush the filehandle's buffer every time `print()` is called.

To turn on autoflush mode, set the special variable `$|` to a true value. Autoflush mode affects the currently selected filehandle, so to change autoflush mode for a specific filehandle, one must first `select()` it, set `$|` to true, and then `select()` the previously selected filehandle. For example, to turn on autoflush mode for filehandle `FH`:

```
my $previous = select(FH);
$| = 1;
select($previous);
```

You will sometimes see this motif abbreviated using the following mind-blowing idiom:

```
select((select(FH), $|=1)[0]);
```

¹You can think of the stdio library as a layer sitting on top of the OS, making the OS look more C-like to programs; similarly, a Pascal standard I/O library makes the OS look as if it were written in Pascal.

However, it is much cleaner to bring in the `IO::Handle` module, which adds an `autoflush()` method to filehandles. With `IO::Handle` loaded, `FH` can be put into autoflush mode like this:

```
use IO::Handle;
FH->autoflush(1);
```

If the OO syntax confuses you, see the Objects and References section later in this chapter.

The other way to avoid stdio buffering problems is to use the `sysread()` and `syswrite()` calls. These calls bypass the stdio library and go directly to the operating system I/O calls. An important advantage of these calls is that they interoperate well with other low-level I/O calls, such as the four-argument `select()` call, and with advanced techniques such as nonblocking I/O.

Another ramification of the fact that the `sys*` functions bypass stdio is the difference in behavior between `read()` and `sysread()` when asked to fetch a larger chunk of data than is available. In the case of `read()`, the function will block indefinitely until it can fetch exactly the amount of data requested. The only exception to this is when the filehandle encounters the end of file before the full request has been satisfied, in which case `read()` will return everything to the end of the file. In contrast, `sysread()` can and will return partial reads. If it can't immediately read the full amount of data requested, it will return the data that is available. If no data is available, `sysread()` will block until it can return at least 1 byte. This behavior makes `sysread()` invaluable for use in network communications, where data frequently arrives in chunks of unpredictable size.

For these reasons, `sysread()` and `syswrite()` are preferred for many network applications.

Passing and Storing Filehandles

Network applications frequently must open multiple filehandles simultaneously, pass them to subroutines, and keep track of them in hashes and other data structures. Perl allows you to treat filehandles as strings, store them into variables, and pass them around to subroutines. For example, this functional but flawed code fragment stores the `MY_FH` filehandle into a variable named `$fh`, and then passes it to a subroutine named `hello()` to use in printing a friendly message:

```
# Flawed technique
$fh = MY_FH;
hello($fh);
sub hello {
    my $handle = shift;
    print $handle "Howdy folks!\n";
}
```

24 Chapter 1 Input/Output Basics

This technique often works; however, it will run you into problems as soon as you try to pass filehandles to subroutines in other packages, such as functions exported by modules. The reason is that passing filehandles as strings loses the filehandle package information. If we pass the filehandle `MY_FH` from the main script (package `main`) to a subroutine defined in the `MyUtils` module, the subroutine will try to access a filehandle named `MyUtils::MY_FH` rather than the true filehandle, which is `main::MY_FH`. The same problem also occurs, of course, when a subroutine from one package tries to return a filehandle to a caller from another package.

The correct way to move filehandles around is as a *typeglob* or a *typeglob reference*. Typeglobs are symbol table entries, but you don't need to know much more about them than that in order to use them (see the *perlref* POD documentation for the full details). To turn a filehandle into a glob put an asterisk ("`*`") in front of its name:

```
$fh = *MY_FH;
```

To turn a filehandle into a typeglob reference, put "`*`" in front of its name:

```
$fh = \*MY_FH;
```

In either case, `$fh` can now be used to pass the filehandle back and forth between subroutines and to store filehandles in data structures. Of the two forms, the glob reference (`*HANDLE`) is the safer, because there's less risk of accidentally writing to the variable and altering the symbol table. This is the form we use throughout this book, and the one used by Perl's I/O-related modules, such as `IO::Socket`.

Typeglob references can be passed directly to subroutines:

```
hello(\*MY_FH):
```

They can also be returned directly by subroutines:

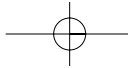
```
my $fh = get_fh();
sub get_fh {
    open (FOO,"foo.txt") or die "foo: $!";
    return \*FOO;
}
```

Typeglob refs can be used anywhere a bare filehandle is accepted, including as the first argument to `print()`, `read()`, `sysread()`, `syswrite()`, or any of the socket-related calls that we discuss in later chapters.

Sometimes you will need to examine a scalar variable to determine whether it contains a valid filehandle. The `fileno()` function makes this possible:

\$integer = fileno (FILEHANDLE)

The `fileno()` function accepts a filehandle in the form of a bare string, a typeglob, or a typeglob reference. If the filehandle is valid, `fileno()` returns the file descriptor for the filehandle. This is a small integer that uniquely identifies the filehandle to the



operating system. `STDIN`, `STDOUT`, and `STDERR` generally correspond to descriptors 0, 1, and 2, respectively (but this can change if you close and reopen them). Other filehandles have descriptors greater than 3.

If the argument passed to `fileno()` does not correspond to a valid filehandle (including former filehandles that have been closed), `fileno()` returns `undef`. Here is the idiom for checking whether a scalar variable contains a filehandle:

```
die "not a filehandle" unless defined fileno($fh);
```

Detecting Errors

Because of the vicissitudes of Internet communications, I/O errors are common in network applications. As a rule, each of the Perl functions that performs I/O returns `undef`, a false value, on failure. More specific information can be found by examining the special global variable `$!`.

`$!` has an interesting dual nature. Treated as a string, it will return a human-readable error message such as `Permission denied`. Treated as a number, however, it will return the numeric constant for the error, as defined by the operating system (e.g., `EACCES`). It is generally more reliable to use these numeric error constants to distinguish particular errors, because they are standardized across operating systems.

You can obtain the values of specific error message constants by importing them from the `Errno` module. In the `use` statement, you can import individual constants by name, or all of them at once. To bring in individual constants, list them in the `use ()` statement, as shown here:

```
use Errno qw(EACCES ENOENT);
my $result = open (FH,">/etc/passwd");
if (!$result) { # oops, something went wrong
    if ($! == EACCES) {
        warn "You do not have permission to open this file.";
    } elsif ($! == ENOENT) {
        warn "File or directory not found.";
    } else {
        warn "Some other error occurred: $!";
    }
}
```

The `qw()` operator is used to split a text string into a list of words. The first line above is equivalent to:

```
use Errno ('EACCESS','ENOENT');
```

and brings in the `EACCESS` and `ENOENT` constants. Notice that we use the numeric comparison operator `"=="` when comparing `$!` to numeric constants.

To bring in all the common error constants, import the tag: `POSIX`. This brings in the error constants that are defined by the POSIX standard,

26 Chapter 1 Input/Output Basics

a cross-platform API that UNIX, Windows NT/2000, and many other operating systems are largely compliant with. For example:

```
use Errno qw(:POSIX);
```

Do not get into the habit of testing `$!` to see if an error occurred during the last operation. `$!` is set when an operation fails, but is not unset when an operation succeeds. The value of `$!` should be relied on only immediately after a function has indicated failure.

Using Object-Oriented Syntax with the IO::Handle and IO::File Modules

We use Perl5's object-oriented facilities extensively later in this book. Although you won't need to know much about creating object-oriented modules, you will need a basic understanding of how to use OO modules and their syntax. This section illustrates the basics of Perl's OO syntax with reference to the IO::Handle and IO::File module, which together form the basis of Perl's object-oriented I/O facilities.

Objects and References

In Perl, references are pointers to data structures. You can create a reference to an existing data structure using the backslash operator. For example:

```
$a = 'hi there';
$a_ref = \$a; # reference to a scalar
@b = ('this', 'is', 'an', 'array');
$b_ref = \@b; # reference to an array
%c = ( first_name => 'Fred', last_name => 'Jones' );
$c_ref = \%c; # reference to a hash
```

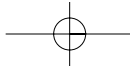
Once a reference has been created, you can make copies of it, as you would any regular scalar, or stuff it into arrays or hashes. When you want to get to the data contained inside a reference, you dereference it using the prefix appropriate for its contents:

```
$a = $$a_ref;
@b = @$b_ref;
%c = %$c_ref;
```

You can index into array and hash references without dereferencing the whole thing by using the `->` syntax:

```
$b_ref->[2];           # yields "an"
$c_ref->{last_name};    # yields "Jones"
```

It is also possible to create references to anonymous, unnamed arrays and hashes, using the following syntax:



```
$anonymous_array = ['this','is','an','anonymous','array'];
$anonymous_hash = { first_name => 'Jane', last_name => 'Doe' };
```

If you try to print out a reference, you'll get a string like `HASH(0x82ab0e0)`, which indicates the type of reference and the memory location where it can be found (which is just short of useless).

An *object* is a reference with just a little bit extra. It is “blessed” into a particular module’s package in such a way that it carries information about what module created it.² The blessed reference will continue to work just like other references. For example, if the object named `$object` is a blessed hash reference, you can index into it like this:

```
$object->{last_name};
```

What makes objects different from plain references is that they have *methods*. A method call uses the `->` notation, but followed by a subroutine name and optional subroutine-style arguments:

```
$object->print_record(); # invoke the print_record() method
```

You may sometimes see a method called with arguments, like this:

```
$object->print_record(encoding => 'EBCDIC');
```

The “`=>`” symbol is accepted by Perl as a synonym for `,`. It makes the relationship between the two arguments more distinct, and has the added virtue of automatically quoting the argument on the left. This allows us to write *encoding* rather than “*encoding*”. If a method takes no arguments, it’s often written with the parentheses omitted, as in:

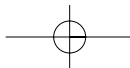
```
$object->print_record;
```

In many cases, `print_record()` will be a subroutine defined in the object’s package. Assuming that the object was created by a module named `BigDatabase`, the above is just a fancy way of saying this:

```
BigDatabase::print_record($object);
```

However, Perl is more subtle than this, and the `print_record()` method definition might actually reside in another module, which the current module *inherits* from. How this works is beyond the scope of this introduction, and can be found in the *perltoot*, *perlobj*, and *perlref* POD pages, as well as in [Wall 2000] and the other general Perl reference works listed in Appendix D.

²The function responsible for turning ordinary references into blessed ones is, naturally enough, called `bless()`.



28 Chapter 1 Input/Output Basics

To create an object, you must invoke one of its *constructors*. A constructor is a method call that is invoked from the module's name. For example, to create a new `BigDatabase` object:

```
$object = BigDatabase->new(); # call the new() constructor
```

Constructors, which are a special case of a *class method*, are frequently named `new()`. However, any subroutine name is possible. Again, this syntax is part trickery. In most cases an equivalent call would be:

```
$object = BigDatabase::new('BigDatabase');
```

This is not quite the same thing, however, because class methods can also be inherited.

The `IO::Handle` and `IO::File` Modules

The `IO::Handle` and `IO::File` modules, standard components of Perl, together provide object-oriented interface to filehandles. `IO::Handle` provides generic methods that are shared by all filehandle objects, including pipes and sockets. The more specialized class, `IO::File`, adds functionality for opening and manipulating files. Together, these classes smooth out some of the bumps and irregularities in Perl's built-in filehandles, and make larger programs easier to understand and maintain.

`IO::File`'s elegance does not by itself provide any very compelling reason to choose the object-oriented syntax over native filehandles. Its main significance is that `IO::Socket`, `IO::Pipe`, and other I/O-related modules also inherit their behavior from `IO::Handle`. This means that programs that read and write from local files and those that read and write to remote network servers share a common, easy-to-use interface.

We'll get a feel for the module by looking at a tiny example of a program that opens a file, counts the number of lines, and reports its findings (Figure 1.4).

Lines 1–3: Load modules We turn on strict syntax checking, and load the `IO::File` module.

Lines 4–5: Initialize variables We recover from the command line the name of the file to perform the line count on, and initialize the `$counter` variable to zero.

Line 6: Create a new `IO::File` object We call the `IO::File::new()` method, using the syntax `IO::File->new()`. The argument is the name of the file to open. If successful, `new()` returns a new `IO::File` object that we can use for I/O. Otherwise it returns `undef`, and we die with an error message.

Lines 7–9: Main loop We call the `IO::File` object's `getline()` method in the test portion of a `while()` loop. This method returns the next line of text, or `undef` on end of file—just like `<>`.

Figure 1.4: The *count_lines.pl* program

```

0  #!/usr/bin/perl
1  # file: count_lines.pl

2  use strict;
3  use IO::File;

4  my $file    = shift;
5  my $counter = 0;
6  my $fh = IO::File->new($file) or die "Can't open $file: $!\n";
7  while ( defined (my $line = $fh->getline) ) {
8      $counter++;

9  }
10  STDOUT->print("Counted $counter lines\n");

```

Each time through the loop we bump up `$counter`. The loop continues until `getline()` returns `undef`.

Line 10: Print results We print out our results by calling `STDOUT->print()`. We'll discuss why this surprising syntax works in a moment.

When I ran *count_lines.pl* on the unfinished manuscript for this chapter, I got the following result:

```
% count_lines.pl ch1.pod
Counted 2428 lines
```

IO::File objects are actually blessed typeglob references (see the Passing and Storing Filehandles section earlier in this chapter). This means that you can use them in an object-oriented fashion, as in:

```
$fh->print("Function calls are for the birds.\n");
```

or with the familiar built-in function calls:

```
print $fh "Object methods are too effete.\n";
```

Many of IO::File's methods are simple wrappers around Perl's built-in functions. In addition to `print()` and `getline()` methods, there are `read()`, `syswrite()`, and `close()` methods, among others. We discuss the pros and cons of using object-oriented method calls and function calls in Chapter 5, where we introduce IO::Socket.

When you load IO::File (technically, when IO::File loads IO::Handle, which it inherits from), it adds methods to ordinary filehandles. This means that any of the methods in IO::File can also be used with STDIN, STDOUT,

30 Chapter 1 Input/Output Basics

STDERR, or even with any conventional filehandles that you happen to create. This is why line 10 of Figure 1.4 allows us to print to standard output by calling `STDOUT->print()`.

Of the method listings that follow, only the `new()` and `new_tmpfile()` methods are actually defined by `IO::File`. The rest are inherited from `IO::Handle` and can be used with other descendents of `IO::Handle`, such as `IO::Socket`. This list is not complete. I've omitted some of the more obscure methods, including those that allow you to move around inside a file in a record-oriented fashion, because we won't need them for network communications.

`$fh = IO::File->new($filename [, $mode [, $perms]])`

The `new()` method is the main constructor for `IO::File`. It is a unified replacement for both `open()` and `sysopen()`. Called with a single argument, `new()` acts like the two-argument form of `open()`, taking a filename optionally preceded by a mode string. For example, this will open the indicated file for appending:

```
$fh = IO::File->new(">darkstar.txt");
```

If called with two or three arguments, `IO::File` treats the second argument as the open mode, and the third argument as the file creation permissions. `$mode` may be a Perl-style mode string, such as `"+<"`, or an octal numeric mode, such as those used by `sysopen()`. As a convenience, `IO::File` automatically imports the `Fcntl` `O_*` constants when it loads. In addition, `open()` allows for an alternative type of symbolic mode string that is used in the C `fopen()` call; for example, it allows `"w"` to open the file for writing. We won't discuss those modes further here, because they do not add functionality.

The permission agreement given by `$perms` is an octal number, and has the same significance as the corresponding parameter passed to `sysopen()`.

If `new()` cannot open the indicated file, it will return `undef` and set `$!` to the appropriate system error message.

`$fh = IO::File->new_tmpfile`

The `new_tmpfile()` constructor, which is called without arguments, creates a temporary file opened for reading and writing. On UNIX systems, this file is anonymous, meaning that it cannot be seen on the file system. When the `IO::File` object is destroyed, the file and all its contents will be deleted automatically.

This constructor is useful for storing large amounts of temporary data.

`$result = $fh->close`

The `close()` method closes the `IO::File` object, returning a true result if successful. If you do not call `close()` explicitly, it will be called automatically when the object is destroyed. This happens when the script exits, if you happen to `undef()` the object, or if the object goes out of scope such as when a `my` variable reaches the end of its enclosing block.

`$result = $fh->open($filename [, $mode [, $perms]])`

You can reopen a filehandle object on the indicated file by using its `open()` method. The input arguments are identical to `new()`. The method result indicates whether the open was successful.

This is chiefly used for reopening the standard filehandles STDOUT, STDIN, and STDERR. For example:

```
STDOUT->open(">log.txt") or die "Can't reopen STDOUT: $!";
```

Calls to `print()` will now write to the file `log.txt`.

`$result = $fh->print(@args)`

`$result = $fh->printf($fmt,@args)`

`$bytes = $fh->write($data [,$length [,$offset]])`

`$bytes = $fh->syswrite($data [,$length [,$offset]])`

The `print()`, `printf()`, and `syswrite()` methods work exactly like their built-in counterparts. For example, `print()` takes a list of data items, writes them to the filehandle object, and returns true if successful.

The `write()` method is the opposite of `read()`, writing a stream of bytes to the filehandle object and returning the number successfully written. It is similar to `syswrite()`, except that it uses stdio buffering. This method corrects the inconsistent naming of the built-in `write()` function, which creates formatted reports. The IO::File object method that corresponds to the built-in `write()` goes by the name of `format_write()`.

`$line = $fh->getline`

`@lines = $fh->getlines`

`$bytes = $fh->read($buffer,$length [,$offset])`

`$bytes = $fh->sysread($buffer,$length [,$offset])`

The `getline()` and `getlines()` methods together replace the `<>` operator. `getline()` reads one line from the filehandle object and returns it, behaving in the same way in both scalar and list contexts. The `getlines()` method acts like `<>` in a list context, returning a list of all the available lines. `getline()` will return undef at the end of file.

The `read()` and `sysread()` methods act just like their built-in function counterparts.

`$previous = $fh->autoflush([$boolean])`

The `autoflush()` method gets or sets the `autoflush()` mode for the filehandle object. Called without arguments, it turns on autoflush. Called with a single boolean argument, it sets autoflush to the indicated status. In either case, `autoflush()` returns the previous value of the autoflush state.

`$boolean = $fh->opened`

The `opened()` method returns true if the filehandle object is currently valid. It is equivalent to:

```
defined fileno($fh);
```

`$boolean = $fh->eof`

Returns true if the next read on the filehandle object will return EOF.

32 Chapter 1 Input/Output Basics

`$fh->flush`

The `flush()` method immediately flushes any data that is buffered in the filehandle object. If the filehandle is being used for writing, then its buffered data is written to disk (or to the pipe, or network, as we'll see when we get to `IO::Socket` objects). If the filehandle is being used for reading, any data in the buffer is discarded, forcing the next read to come from disk.

`$boolean = $fh->blocking([$boolean])`

The `blocking()` method turns on and off blocking mode for the filehandle. We discuss how to use this at length in Chapter 13.

`$fh->clearerr`

`$boolean = $fh->error`

These two methods are handy if you wish to perform a series of I/O operations and check the error status only after you're finished. The `error()` method will return true if any errors have occurred on the filehandle since it was created, or since the last call to `clearerr()`. The `clearerr()` method clears this flag.

In addition to the methods listed here, `IO::File` has a constructor named `new_from_fd()`, and a method named `fdopen()`, both inherited from `IO::Handle`. These methods can be used to save and restore objects in much the way that the `>&FILEHANDLE` does with standard filehandles.

`$fh = IO::File->new_from_fd($fd,$mode)`

The `new_from_fd()` method opens up a copy of the filehandle object indicated by `$fd` using the read/write mode given by `$mode`. The object may be an `IO::Handle` object, an `IO::File` object, a regular filehandle, or a numeric file descriptor returned by `fileno()`. `$mode` must match the mode with which `$fd` was originally opened. For example:

```
$saveout = IO::File->new_from_fd(STDOUT,">");
```

`$result = $fh->fdopen($fd,$mode)`

The `fdopen()` method is used to reopen an existing filehandle object, making it a copy of another one. The `$fd` argument may be an `IO::Handle` object or a regular filehandle, or a numeric file descriptor `$mode` must match the mode with which `$fd` was originally opened.

This is typically used in conjunction with `new_from_fd()` to restore a saved filehandle:

```
$saveout = IO::File->new_from_fd(STDOUT,">"); # save STDOUT
STDOUT->open('>log.txt');                      # reopen on a file
STDOUT->print("Yippy yie yay!\n");              # print something
STDOUT->fdopen($saveout,">");                  # reopen on saved value
```

See the POD documentation for `IO::Handle` and `IO::File` for information about the more obscure features that these modules provide.

Summary

Perl and network programming were made for each other. Perl's strong text-processing abilities combine with a flexible I/O subsystem to create an environment that is ideal for interprocess communication. This, combined with its native support for the Berkeley Sockets protocol, make Perl an excellent choice for network applications.

In this chapter we reviewed the essential components of Perl's I/O API. Filehandles are the fundamental object used for Perl input/output operations, and offer both line-oriented and byte-stream-oriented modes.

The `STDIN`, `STDOUT`, and `STDERR` filehandles are available when a program is started, and correspond to the standard input, output, and error devices. A script may open up additional filehandles, or reopen the standard ones on different files.

The standard I/O library, used by the `<>`, `read()`, and `print()` functions, improves I/O efficiency by adding a layer of buffering to input and output operations. However, this buffering can sometimes get in the way. One way to avoid buffering problems is to put the filehandle into autoflush mode. Another way is to use the lower-level `syswrite()` and `sysread()` functions.

The `IO::File` and `IO::Handle` modules add object-oriented methods to filehandles. They smooth out some of the inconsistencies in Perl's original design, and pave the way to a smooth transition to `IO::Socket`.

