Table of Contents

Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

Chapter 4. External Data Manipulation

IN THIS CHAPTER

On a clean disk you can seek forever.

—Thomas B. Steel, Jr.

Computers are good at computing. This tautology is more profound than it appears. If we only had to sit and chew up the CPU cycles and reference RAM as needed, life would be easy.

A computer that only sits and thinks to itself is of little use to us, however. Sooner or later we have to get information into it and out of it, and that is where life gets harder.

Several things make I/O complicated. First of all, input and output are rather different things, but we naturally lump them together. Second, the varieties of I/O operations (and their usages) are as diverse as species of insects.

History has seen such devices as drums, paper tapes, magnetic tapes, punched cards, and teletypes. Some operated with a mechanical component; others were purely electromagnetic. Some were read-only; others were write-only or read-write. Some writable media were erasable, and others were not. Some devices were inherently sequential; others were random access. Some media were permanent; others were transient or volatile. Some devices depended on human intervention; others did not. Some were character oriented; others were block oriented. Some block devices were fixed length; others were variable length. Some devices were polled; others were interrupt driven. Interrupts could be implemented in hardware or software, or both. We have seen both buffered and nonbuffered I/O. We have seen memory-mapped I/O, channel-oriented I/O, and with the advent of operating systems such as Unix, we have seen I/O devices mapped to files in a file system. We have done I/O in machine language, in assembly language, and in high-level languages. Some languages have the I/O capabilities firmly hardwired in place; others leave it out of the language specification completely. We have done I/O with and without suitable device drivers or layers of abstraction.

Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

If this seems like a confusing mess, that's because it is. Part of the complexity is inherent in the concept of input/output, part of it is the result of design tradeoffs, and part of it is the result of legacies or traditions in computer science and the quirks of various languages and operating systems.

Ruby's I/O is complex because I/O in general is complex. However, we have tried to make it understandable and present a good overview of how and when to use various techniques.

The core of all Ruby I/O is the `IO` class, which defines behavior for every kind of input/output operation. Closely allied with `IO` (and inheriting from it) is the `File` class. There is a nested class within `File` called `Stat`, which is an object that encapsulates various details about a file that we might want to examine (such as its permissions and timestamps). The methods `stat` and `lstat` return objects of type `File::Stat`.

The module `FileTest` also has methods that allow us to test much the same set of properties. This is mixed into the `File` class and can also be used on its own.

Finally, there are I/O methods in the `Kernel` module that are mixed into `Object` (the ancestor of all objects). These are the simple I/O routines we have used all along without worrying about what their receiver was. These naturally default to standard input and standard output.

The beginner may find these classes to be a confused jumble of overlapping functionality. The good news is that you need only use small pieces of this framework at any given time.

On a higher level, Ruby offers features to make object persistence possible. The `Marshal` enables simple serialization of objects, and the more sophisticated `PStore` library is based on `Marshal`. We include the `DBM` library in this section, although it is only string based.

On the highest level of all, data access can be performed by interfacing to a separate database management system such as MySQL or Oracle. This issue is complex enough that one or more books could devoted to it. We will provide only a brief overview to get you started. In some cases, we provide only a pointer to an online archive.

Working with Files and Directories

When we say "file," we usually mean a disk file, although not always. We do use the concept of a file as a meaningful abstraction in Ruby as in other programming languages. When we say "directory," we mean a directory in the normal Windows or Unix sense.

Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

The `File` class is closely related to the `IO` class from which it inherits. The `Dir` class is not so closely related, but we chose to discuss files and directories together because they are still conceptually related.

Opening and Closing Files

The class method `File.new`, which instantiates a `File` object, will also open that file. The first parameter is naturally the filename.

The optional second parameter is called the *mode string,* which tells how to open the file (for reading, writing, or whatever). The mode string has nothing to do with the mode as in permissions. This defaults to `"r"` for reading. Here's an example:

```
file1 = File.new("one")       # Open for reading
file2 = File.new("two", "w")  # Open for writing
```

There is another form for `new` that takes three parameters. In this case, the second parameter specifies the original permissions for the file (usually as an octal constant), and the third is a set of flags `OR`ed together. The flags are constants such as `File::CREAT` (create the file when it is opened if it doesn't already exist) and `File::RDONLY` (open for reading only). This form will rarely be used. Here's an example:

```
file = File.new("three", 0755, File::CREAT|File::WRONLY)
```

As a courtesy to the operating system and the runtime environment, always close a file that you open. In the case of a file open for writing, this is more than mere politeness and can actually prevent lost data. Not surprisingly, the `close` method will serve this purpose:

```
out = File.new("captains.log", "w")
# Process as needed...
out.close
```

There is also an `open` method. In its simplest form, it is merely a synonym for `new`, as shown here:

```
trans = File.open("transactions","w")
```

However, `open` can also take a block; this is the form that is more interesting. When a block is specified, the open file is passed in as a parameter to the block. The file remains open throughout the scope of the block and is closed automatically at the end. Here's an example:

Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

```
File.open("somefile","w") do |file|
  file.puts "Line 1"
  file.puts "Line 2"
  file.puts "Third and final line"
end
# The file is now closed
```

This is obviously an elegant way of ensuring that a file is closed when we've finished with it. In addition, the code that handles the file is grouped visually into a unit.

Updating a File

Suppose we want to open a file for reading and writing. This is done simply by adding a plus sign (+) in the file mode when we open the file (see the section titled "Opening and Closing Files" for more information). Here's an example:

```
f1 = File.new("file1", "r+")
    # Read/write, starting at beginning of file.

    f2 = File.new("file2", "w+")
    # Read/write; truncate existing file or create a new one.
    f3 = File.new("file3", "a+")
    # Read/write; start at end of existing file or create a
    # new one.
```

Appending to a File

Suppose we want to append information onto an existing file. This is done simply by using `"a"` in the file mode when we open the file (see the section titled "Opening and Closing Files" for more information). Here's an example:

```
logfile = File.open("captains_log", "a")
    # Add a line at the end, then close.
    logfile.puts "Stardate 47824.1: Our show has been canceled."
    logfile.close
```

Random Access to Files

If you want to read a file randomly rather than sequentially, you can use the method `seek`, which `File` inherits from `IO`. The simplest usage is to seek to a specific byte position. The position is relative to the beginning of the file, where the first byte is numbered `0`. Here's an example:

```
# myfile contains only: abcdefghi
file = File.new("myfile")
file.seek(5)
str = file.gets    # "fghi"
```


Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

If you took care to ensure that each line is a fixed length, you could seek to a specific line, like so:

```
# Assume 20 bytes per line.
# Line N starts at byte (N-1)*20
file = File.new("fixedlines")
file.seek(5*20)                  # Sixth line!
# Elegance is left as an exercise.
```

If you want to do a relative seek, you can use a second parameter. The constant `IO::SEEK_CUR` will assume the offset is relative to the current position (which may be negative). Here's an example:

```
file = File.new("somefile")
file.seek(55)                  # Position is 55
file.seek(-22, IO::SEEK_CUR)   # Position is 33
file.seek(47, IO::SEEK_CUR)    # Position is 80
```

You can also seek relative to the end of the file. Only a negative offset makes sense here:

```
file.seek(-20, IO::SEEK_END)  # twenty bytes from eof
```

There is also a third constant, `IO::SEEK_SET`, but it is the default value (seek relative to the beginning of file).

The method `tell` will report the file position (`pos` is an alias):

```
file.seek(20)
pos1 = file.tell             # 20
file.seek(50, IO::SEEK_CUR)
pos2 = file.pos              # 70
```

The `rewind` method can also be used to reposition the file pointer at the beginning. This terminology comes from the use of magnetic tapes.

If you are performing random access on a file, you may want to open it for updating (reading and writing). Updating a file is done by specifying a plus sign (+) in the mode string. See the section titled "Updating a File" for more information.

Working with Binary Files

In days gone by, C programmers would use the `"b"` character appended to the mode string in order to open a file as a binary. This character is still allowed for compatibility in most

Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

cases, but nowadays binary files are not as tricky as they used to be. A Ruby string can easily hold binary data, and a file need not be read in any special way.

The exception is the Windows family of operating systems. The chief difference between binary and text files on these platforms is that in binary mode, the end-of-line is not translated into a single linefeed but is kept as a carriage-return/linefeed pair.

The `"b"` character is indeed used in this circumstance:

```
# Input file contains a single line: Line 1.
file = File.open("data")
line = file.readline             # "Line 1.\n"
puts "#{ line.size}  characters."  # 8 characters
file.close

file = File.open("data","rb")
line = file.readline             # "Line 1.\r\n"
puts "#{ line.size}  characters."  # 9 characters
file.close
```

Note that the `binmode` method can switch a stream to binary mode. Once switched, it cannot be switched back. Here's an example:

```
file = File.open("data")
file.binmode
line = file.readline             # "Line 1.\r\n"
puts "#{ line.size}  characters."  # 9 characters
file.close
```

If you really want to do low-level input/output, you can use the `sysread` and `syswrite` methods. The former takes a number of bytes as a parameter; the latter takes a string and returns the actual number of bytes written. (You should *not* use other methods to read from the same stream; the results may be unpredictable.) Here's an example:

```
input = File.new("infile")
output = File.new("outfile")
instr = input.sysread(10);
ytes = output.syswrite("This is a test.")
```

Note that `sysread` raises `EOFError` at end-of-file. Either of these methods will raise `SystemCallError` when an error occurs.

Note that the `Array` method `pack` and the `String` method `unpack` can be very useful in dealing with binary data.

Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

Locking Files

On operating systems where it is supported, the `flock` method of `File` will lock or unlock a file. The second parameter is one of these constants: `File::LOCK_EX`, `File::LOCK_NB`, `File::LOCK_SH`, `File::LOCK_UN`, or a logical `OR` of two or more of these. Note, of course, that many of these combinations will be nonsensical; primarily the nonblocking flag will be `OR`ed in if anything is. Here's an example:

```
file = File.new("somefile")

file.flock(File::LOCK_EX)  # Lock exclusively; no other process
                           # may use this file.
file.flock(File::LOCK_UN)  # Now unlock it.

file.flock(File::LOCK_SH)  # Lock file with a shared lock (other
                           # processes may do the same).
file.flock(File::LOCK_UN)  # Now unlock it.

locked = file.flock(File::LOCK_EX | File::LOCK_NB)
# Try to lock the file, but don't block if we can't; in that case,
# locked will be false.
```

Performing Simple I/O

You are already familiar with some of the I/O routines in the `Kernel` module; these are the ones we have called all along without specifying a receiver for the methods. Calls such as `gets` and `puts` originate here; others are `print`, `printf`, and `p` (which calls the object's `inspect` method to display it in some way readable to humans).

There are some others that we should mention for completeness, though. For example, the `putc` method will output a single character. (The corresponding method `getc` is *not* implemented in `Kernel` for technical reasons; it can be found in any `IO` object, however.) If a `String` is specified, the first character of the string will be taken. Here's an example:

```
putc(?\n)   # Output a newline
putc("X")   # Output the letter X
```

A reasonable question is, where does output go when we use these methods without a receiver? Well, to begin with, three constants are defined in the Ruby environment corresponding to the three standard I/O streams we are accustomed to in Unix. These are `STDIN`, `STDOUT`, and `STDERR`. All are global constants of the type `IO`.

There is also a global variable called `$defout` that is the destination of all the output coming from `Kernel` methods. This is initialized (indirectly) to the value of `STDOUT` so

Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

that this output all gets written to standard output as we expect. The variable `$defout` can be reassigned to refer to some other `IO` object at any time. Here's an example:

```
diskfile = File.new("foofile","w")
puts "Hello..."  # prints to stdout
$defout = diskfile
puts "Goodbye!"  # prints to "foofile"
diskfile.close
```

Besides `gets`, `Kernel` also has the methods `readline` and `readlines` for input. The former is equivalent to `gets` except that it raises `EOFError` at the end of a file instead of just returning a `nil` value. The latter is equivalent to the `IO.readlines` method (that is, it reads an entire file into memory).

Where does input come from? Well, there is also the standard input stream `$stdin`, which defaults to `STDIN`. In the same way, there is a standard error stream (`$stderr` defaulting to `STDERR`).

Also, an interesting global object called `ARGF` represents the concatenation of all the files named on the command line. It is not really a `File` object, although it resembles one. Standard input is connected to this object in the event files are named on the command line.

Performing Buffered and Unbuffered I/O

Ruby does its own internal buffering in some cases. Consider this fragment:

```
print "Hello... "
sleep 10
print "Goodbye!\n"
```

If you run this, you will notice that the hello and goodbye messages both appear at the same time, *after* the sleep. The first output is not terminated by a newline.

This can be fixed by calling `flush` to flush the output buffer. In this case, we use the stream `$defout` (the default stream for all `Kernel` method output) as the receiver. It then behaves as we probably wanted, with the first message appearing earlier than the second one:

```
print "Hello... "
$defout.flush
sleep 10
print "Goodbye!\n"
```


Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

This buffering can be turned off (or on) with the `sync=` method; the `sync` method will let us know the status:

```
buf_flag = $defout.sync    # true
$defout.sync = false
buf_flag = $defout.sync    # false
```

Also, at least one lower level of buffering is going on behind the scenes. Just as the `getc` method returns a character and moves the file or stream pointer, the `ungetc` method pushes a character back onto the stream:

```
ch = mystream.getc    # ?A
mystream.ungetc(?C)
ch = mystream.getc    # ?C
```

You should be aware of three things here. First of all, the buffering we're speaking of is unrelated to the buffering we mentioned earlier in this section; in other words, `sync=false` won't turn it off. Second, only one character can be pushed back; if you attempt more than one, only the last character will actually be pushed back onto the input stream. Finally, the `ungetc` method will not work for inherently unbuffered read operations (such as `sysread`).

Manipulating File Ownership and Permissions

The issue of file ownership and permissions is highly platform dependent. Typically, Unix provides a superset of the functionality; for other platforms many features may be unimplemented.

To determine the owner and group of a file (which are integers), `File::Stat` has a pair of instance methods, `uid` and `gid`, as shown here:

```
data = File.stat("somefile")
owner_id = data.uid
group_id = data.gid
```

Class `File::Stat` has the instance method `mode`, which will return the mode (or permissions) of the file:

```
perms = File.stat.mode("somefile")
```


Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

`File` has class and instance methods named `chown` to change the owner and group IDs of a file. The class method will accept an arbitrary number of filenames. Where an ID is not to be changed, `nil` or `-1` can be used. Here's an example:

```
uid = 201
gid = 10
File.chown(uid, gid, "alpha", "beta")
f1 = File.new("delta")
f1.chown(uid, gid)
f2 = File.new("gamma")
f2.chown(nil, gid)      # Keep original owner id
```

Likewise, the permissions can be changed by `chmod` (also implemented both as class and instance methods). The permissions are traditionally represented in octal format, although they need not be:

```
File.chmod(0644, "epsilon", "theta")
f = File.new("eta")
f.chmod(0444)
```

A process always runs under the identity of some user (possibly root); as such, a user ID is associated with it. (Here we are talking about the *effective* user ID.) We frequently need to know whether that user has permission to read, write, or execute a given file. There are instance methods in `File::Stat` to make this determination, as shown here:

```
info = File.stat("/tmp/secrets")
rflag = info.readable?
wflag = info.writable?
xflag = info.executable?
```

Sometimes we need to distinguish between the effective user ID and the real user ID. The appropriate instance methods are `readable_real?`, `writable_real?`, and `executable_real?`. Here's an example:

```
info = File.stat("/tmp/secrets")
rflag2 = info.readable_real?
wflag2 = info.writable_real?
xflag2 = info.executable_real?
```

We can test the ownership of the file as compared with the effective user ID (and group ID) of the current process. The class `File::Stat` has instance methods `owned?` and `grpowned?` to accomplish this.

Note that many of these methods can also be found in the module `FileTest`:

Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

```
rflag = FileTest::readable?("pentagon_files")
# Other methods are: writable? executable? readable_real? writable_real?
# executable_real? owned? grpowned?
# Not found here: uid gid mode
```

The "umask" associated with a process determines the initial permissions of new files created. The standard mode 0777 is logically ANDed with the negation of the umask so that the bits set in the umask are "masked" or cleared. If you prefer, you can think of this as a simple subtraction (without borrow). Therefore, a umask of 022 will result in files being created with a mode of 0755.

The umask can be retrieved or set with the class method `umask` of class `File`. If there is a parameter specified, the umask will be set to that value (and the previous value will be returned):

```
File.umask(0237)              # Set the umask
current_umask = File.umask    # 0237
```

Some file mode bits (such as the "sticky" bit) are not strictly related to permissions. For a discussion of these, see the section titled "Checking Special File Characteristics."

Retrieving and Setting Timestamp Information

Each disk file has multiple timestamps associated with it (although there are some variations between operating systems). The three timestamps that Ruby understands are the modification time (the last time the file contents were changed), the access time (the last time the file was read), and the change time (the last time the file's directory information was changed).

These three pieces of information can be accessed in three different ways. Each of these fortunately gives the same results.

The `File` class methods `mtime`, `atime`, and `ctime` will return the times without the file being opened or any `File` object being instantiated:

```
t1 = File.mtime("somefile")
# Thu Jan 04 09:03:10 GMT-6:00 2001
t2 = File.atime("somefile")
# Tue Jan 09 10:03:34 GMT-6:00 2001
t3 = File.ctime("somefile")
# Sun Nov 26 23:48:32 GMT-6:00 2000
```

If there happens to be a `File` instance already created, the instance method can be used:

```
myfile = File.new("somefile")
t1 = myfile.mtime
t2 = myfile.atime
t3 = myfile.ctime
```

And if there happens to be a `File::Stat` instance already created, it has instance methods to do the same thing:

```
myfile = File.new("somefile")
info = myfile.stat
t1 = info.mtime
t2 = info.atime
t3 = info.ctime
```

Note that a `File::Stat` is returned by `File`'s class (or instance) method `stat`. The class method `lstat` (or the instance method of the same name) is identical except that it reports on the status of the link itself instead of following the link to the actual file. In the case of links to links, all links are followed but the last one.

File access and modification times may be changed using the `utime` method. It will change the times on one or more files specified. The times may be given either as `Time` objects or as a number of seconds since the epoch. Here's an example:

```
today = Time.now
yesterday = today - 86400
File.utime(today, today, "alpha")
File.utime(today, yesterday, "beta", "gamma")
```

Because both times are changed together, if you want to leave one of them unchanged, you have to save it off first, as shown here:

```
mtime = File.mtime("delta")
File.utime(Time.now, mtime, "delta")
```

Checking File Existence and Size

One fundamental question we sometimes want to know about a file is whether the file of the given name exists. The `exist?` method in the `FileTest` module provides a way to find out:

```
flag = FileTest::exist?("LochNessMonster")
flag = FileTest::exists?("UFO")
# exists? is a synonym for exist?
```


Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

Intuitively, such a method could not be a class instance of `File` because by the time the object is instantiated, the file has been opened; `File` conceivably could have a class method `exist?`, but in fact it does not.

Related to the question of a file's existence is the question of whether it has any contents. After all, a file may exist but have zero length (which is the next best thing to not existing).

If we are only interested in this yes/no question, `File::Stat` has two instance methods that are useful. The method `zero?` will return `true` if the file is zero length; otherwise, it will return `false`:

```
flag = File.new("somefile").stat.zero?
```

Conversely, the method `size?` will return either the size of the file in bytes, if it is nonzero length, or the value `nil`, if it is zero length. It may not be immediately obvious why `nil` is returned rather than `0`. The answer is that the method is primarily intended for use as a predicate, and `0` is "true" in Ruby, whereas `nil` tests as "false." Here's an example:

```
if File.new("myfile").stat.size?
  puts "The file has contents."
else
  puts "The file is empty."
end
```

The methods `zero?` and `size?` also appear in the `FileTest` module:

```
flag1 = FileTest::zero?("file1")
flag2 = FileTest::size?("file2")
```

This leads naturally to the question, how big is this file? We've already seen that in the case of a nonempty file, `size?` will return the length, but if we're not using it as a predicate, the `nil` value would confuse us.

The `File` class has a class method (but *not* an instance method) to give us this answer. The instance method of the same name is inherited from the `IO` class, and `File::Stat` has a corresponding instance method:

```
size1 = File.size("file1")
size2 = File.stat("file2").size
```

If we want the file size in blocks rather than bytes, we can use the instance method `blocks` in `File::Stat`. This is certainly dependent on the operating system. (The

Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

method `blksize` will also report on the operating system's idea of how big a block is.) Here's an example:

```
info = File.stat("somefile")
total_bytes = info.blocks * info.blksize
```

Checking Special File Characteristics

There are numerous aspects of a file that we can test. We summarize here the relevant built-in methods that we don't discuss elsewhere. Most, though not all, are predicates.

Bear in mind two facts throughout this section (and most of this chapter). First of all, because `File` mixes in `FileTest`, any test that can be done by invoking the method qualified with the module name may also be called as an instance method of any file object. Second, remember that there is a high degree of overlap between the `FileTest` module and the `File::Stat` object returned by `stat` (or `lstat`). In some cases, there will be three different ways to call what is essentially the same method. We won't necessarily show this every time.

Some operating systems have the concept of block-oriented devices as opposed to character-oriented devices. A file may refer to neither, but not both. The methods `blockdev?` and `chardev?` in the `FileTest` module will test for this:

```
flag1 = FileTest::chardev?("/dev/hdisk0")  # false
flag2 = FileTest::blockdev?("/dev/hdisk0") # true
```

Sometimes we want to know whether the stream is associated with a terminal. The `IO` class method `tty?` tests for this (as will the synonym `isatty`):

```
flag1 = STDIN.tty?                   # true
flag2 = File.new("diskfile").isatty  # false
```

A stream can be a pipe or a socket. There are corresponding `FileTest` methods to test for these cases:

```
flag1 = FileTest::pipe?(myfile)
flag2 = FileTest::socket?(myfile)
```

Recall that a directory is really just a special case of a file. Therefore, we need to be able to distinguish between directories and ordinary files, which a pair of `FileTest` methods enable us to do. Here's an example:

Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

```
file1 = File.new("/tmp")
file2 = File.new("/tmp/myfile")
test1 = file1.directory?          # true
test2 = file1.file?               # false
test3 = file2.directory?          # false
test4 = file2.file?               # true
```

Also, a `File` class method named `ftype` will tell us what kind of thing a stream is; it can also be found as an instance method in the `File::Stat` class. This method returns a string that has one of the following values: `file`, `directory`, `blockSpecial`, `characterSpecial`, `fifo`, `link`, or `socket`. (The string `fifo` refers to a pipe.) Here's an example:

```
this_kind = File.ftype("/dev/hdisk0")     # "blockSpecial"
that_kind = File.new("/tmp").stat.ftype   # "directory"
```

Certain special bits may be set or cleared in the permissions of a file. These are not strictly related to the other bits we discuss in the section "Manipulating File Ownership and Permissions." These are the set-group-id bit, the set-user-id bit, and the sticky bit. There are methods in `FileTest` for each of these, as shown here:

```
file = File.new("somefile")
info = file.stat
sticky_flag = info.sticky?
setgid_flag = info.setgid?
setuid_flag = info.setuid?
```

A disk file may have symbolic or hard links that refer to it (on operating systems supporting these features). To test whether a file is actually a symbolic link to some other file, use the `symlink?` method of `FileTest`. To count the number of hard links associated with a file, use the `nlink` method (found only in `File::Stat`). A hard link is virtually indistinguishable from an ordinary file; in fact, it *is* an ordinary file that happens to have multiple names and directory entries. Here's an example:

```
File.symlink("yourfile","myfile")           # Make a link
is_sym = FileTest::symlink?("myfile")       # true
hard_count = File.new("myfile").stat.nlink  # 0
```

Incidentally, note that in this example we use the `File` class method `symlink` to create a symbolic link.

In rare cases, you may want even lower-level information about a file. The `File::Stat` class has three more instance methods that give you the gory details. The method `dev` will

Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

give you an integer identifying the device on which the file resides; `rdev` will return an integer specifying the kind of device; and for disk files, `ino` will give you the starting "inode" number for the file:

```
file = File.new("diskfile")
info = file.stat
device = info.dev
devtype = info.rdev
inode = info.ino
```

Working with Pipes

Ruby provides various ways of reading and writing pipes. The class method `IO.popen` will open a pipe and hook the process's standard input and standard output into the `IO` object returned. Frequently we will have different threads handling each end of the pipe; here we just show a single thread writing and then reading:

```
check = IO.popen("spell","r+")
check.puts("'T was brillig, and the slithy toves")
check.puts("Did gyre and gimble in the wabe.")
check.close_write
list = check.readlines
list.collect! {  |x| x.chomp }
# list is now %w[brillig gimble gyre slithy toves wabe]
```

Note that the `close_write` call is necessary. If it were not issued, we would not be able to reach end-of-file when we read the pipe.

There is also a block form:

```
File.popen("/usr/games/fortune") do |pipe|
  quote = pipe.gets
  puts quote
  # On a clean disk, you can seek forever. - Thomas Steel
end
```

If the string `"-"` is specified, a new Ruby instance is started. If a block is specified with this, the block is run as two separate processes rather like a fork; the child gets `nil` passed into the block, and the parent gets an `IO` object with the child's standard input and/or output connected to it, as shown here:

```
IO.popen("-") do |mypipe|
  if mypipe
    puts "I'm the parent: pid = #{ Process.pid} "
    listen = mypipe.gets
    puts listen
  else
```


Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

```
    puts "I'm the child: pid = #{ Process.pid} "
  end
end

# Prints:
#   I'm the parent: pid = 10580
#   I'm the child: pid = 10582
```

Also, the `pipe` method returns a pair of pipe ends connected to each other. Here, we create a pair of threads and let one pass a message to the other (the first message that Samuel Morse sent over the telegraph):

```
pipe = IO.pipe
reader = pipe[0]
writer = pipe[1]

str = nil
thread1 = Thread.new(reader,writer) do |reader,writer|
  # writer.close_write
  str = reader.gets
  reader.close
end

thread2 = Thread.new(reader,writer) do |reader,writer|
  # reader.close_read
  writer.puts("What hath God wrought?")
  writer.close
end
thread1.join
thread2.join

puts str         # What hath God wrought?
```

Performing Special I/O Operations

It is possible to do lower-level I/O in Ruby. We will only mention the existence of these methods; if you need to use them, note that some of them will be highly machine specific (varying even between different versions of Unix).

The `ioctl` method ("I/O control") will accept two arguments. The first is an integer specifying the operation to be done. The second is either an integer or a string representing a binary number.

The `fcntl` method is also for low-level control of file-oriented streams in a system-dependent manner. It takes the same kinds of parameters as `ioctl`.

The `select` method (in the `Kernel` module) will accept up to four parameters; the first is the read-array, and the last three are optional (write-array, error-array, and the timeout value. When input is available from one or more devices in the read-array, or one or more

Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

devices in the write-array are ready, the call will return an array of three elements representing the respective arrays of devices that are ready for I/O.

The `Kernel` method `syscall` takes at least one integer parameter (and up to nine string or integer parameters in all). The first parameter specifies the I/O operation to be done.

The `fileno` method returns an old-fashioned file descriptor associated with an I/O stream. This is the least system dependent of all the methods mentioned. Here's an example:

```
desc = $stderr.fileno      # 2
```

Manipulating Pathnames

When manipulating pathnames, the first things to be aware of are the class methods `File.dirname` and `File.basename`; these work like the Unix commands of the same names and return the directory name and the filename, respectively. If an extension is specified as a second parameter to `basename`, that extension will be removed:

```
str = "/home/dave/podbay.rb"
dir = File.dirname(str)           # "/home/dave"
file1 = File.basename(str)        # "podbay.rb"
file2 = File.basename(str,".rb")  # "podbay"
```

Note that although these are methods of `File`, they are really simply doing string manipulation.

A comparable method is `File.split`, which returns these two components (directory and filename) in a two-element array:

```
info = File.split(str)        # ["/home/dave","podbay.rb"]
```

The `expand_path` class method will expand a relative pathname, converting it to an absolute path. If the operating system understands such idioms as `~` and `~user`, these will be expanded also. Here's an example:

```
Dir.chdir("/home/poole/personal/docs")
abs = File.expand_path("../../misc")    # "/home/poole/misc"
```

Given an open file, the `path` instance method will return the pathname used to open the file:

Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

```
file = File.new("../../foobar")
name = file.path                 # "../../foobar"
```

The constant `File::Separator` gives the character used to separate pathname components (typically a backslash for Windows and a slash for Unix). An alias is `File::SEPARATOR`.

The class method `join` uses this separator to produce a path from a list of directory components:

```
path = File.join("usr","local","bin","someprog")
# path is "usr/local/bin/someprog"
# Note that it doesn't put a separator on the front!
```

Don't fall into the trap of thinking that `File.join` and `File.split` are somehow inverse operations. They're not.

Command-Level File Manipulation

Very often we need to manipulate files in a manner similar to the way we would at a command line. That is, we need to copy, delete, rename, and so on. Many of these capabilities are built-in methods; a few are added by the `ftools` library.

To delete a file, we can use `File.delete` or its synonym `File.unlink`, like so:

```
File.delete("history")
File.unlink("toast")
```

To rename a file, we can use `File.rename` as follows:

```
File.rename("Ceylon","SriLanka")
```

File links (hard and symbolic) can be created using `File.link` and `File.symlink`, respectively:

```
File.link("/etc/hosts","/etc/hostfile")   # hard link
File.symlink("/etc/hosts","/tmp/hosts")   # symbolic link
```

We can truncate a file to zero bytes (or any other specified number) by using the `truncate` instance method:

Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

```
File.truncate("myfile",1000)    # Now at most 1000 bytes
```

Two files may be compared by means of the `compare` method (`cmp` is the alias):

```
require "ftools"

same = File.compare("alpha","beta")  # true
```

The method `syscopy` will efficiently copy a file to a new name or location. A similar method is `copy`, which has an optional flag parameter to write error messages to standard error (`cp` is the alias):

```
require "ftools"

File.syscopy("gamma","delta")      # Copies gamma to delta
File.syscopy("gamma","/tmp")       # Creates /tmp/gamma
# Copy epsilon to theta and log any errors.
File.copy("epsilon","theta", true)
```

A file may be moved with the `move` method (the alias is `mv`). Like `copy`, it also has an optional verbose-flag:

```
require "ftools"

File.move("/tmp/names","/etc")     # Move to new directory
File.move("colours","colors")      # Just a rename
```

The `safe_unlink` method will delete the specified file or files, first trying to make the files writable so as to avoid errors. If the last parameter is `true` or `false`, that value will be taken as the verbose-flag. Here's an example:

```
require "ftools"

File.safe_unlink("alpha","beta","gamma")
# Log errors on the next two files
File.safe_unlink("delta","epsilon",true)
```

Finally, the `install` method basically does a `syscopy`, except that it first checks that the file either does not exist or has different content:

```
require "ftools"

File.install("foo.so","/usr/lib")
```


Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

```
# Existing foo.so will not be overwritten
# if it is the same as the new one.
```

Grabbing Characters from the Keyboard

We use the term "grabbing" here because we sometimes want to process a character as soon as it is pressed rather than buffer it and wait for a newline to be entered.

This can be done in both Unix variants and Windows variants. Unfortunately, the two methods are completely unrelated to each other.

The Unix version is straightforward. We use the well-known technique of putting the terminal in raw mode (and we usually turn off echoing at the same time):

```
system("stty raw -echo")  # Raw mode, no echo
char = STDIN.getc
system("stty -raw echo")  # Reset terminal mode
```

In the Windows world, we need to write a C extension for this. An alternative for now is to use a small feature of the `Win32API` library, shown here:

```
require 'Win32API'
char = Win32API.new("crtdll", "_getch", [], 'L').Call
```

This is obviously not pretty, but it works.

Reading an Entire File into Memory

To read an entire file into an array, you need not even open the file. The method `IO.readlines` will do this, opening and closing the file on its own:

```
arr = IO.readlines("myfile")
lines = arr.size
puts "myfile has #{ lines}  lines in it."

longest = arr.collect { |x| x.length} .max
puts "The longest line in it has #{ longest}  characters."
```

Iterating over a File by Line

To iterate over a file a line at a time, we can use the class method `IO.foreach` or the instance method `each`. In the former case, the file need not be opened in our code. Here's an example:

```
# Print all lines containing the word "target"
```


Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

```
IO.foreach("somefile") do |line|
  puts line if line =~ /target/
end
# Another way...
file = File.new("somefile")
file.each do |line|
  puts line if line =~ /target/
end
```

Note that `each_line` is an alias for `each`.

Iterating over a File by Byte

To iterate a byte at a time, use the `each_byte` instance method. Remember that it feeds a character (that is, an integer) into the block; use the `chr` method if you need to convert to a "real" character. Here's an example:

```
file = File.new("myfile")
e_count = 0
file.each_byte do |byte|
  e_count += 1 if byte == ?e
end
```

Treating a String As a File

Sometimes people want to know how to treat a string as though it were a file. The answer depends on the exact meaning of the question.

An object is defined mostly in terms of its methods. The following code shows an iterator applied to an object called `source`; with each iteration, a line of output is produced. Can you tell the type of `source` by reading this fragment?

```
source.each do |line|
  puts line
end
```

Actually, `source` could be a file, or it could be a string containing embedded newlines. Therefore, in cases like these, a string can trivially be treated as a file.

This leads naturally to the idea of writing something like an `IOString` class. We could do that here, but the exact design of such a class would depend on what you want to do with it. Should we inherit from `String` or from `IO`, for instance? A third possibility would be to create a library that adds methods to the `String` class (just as `ftools` extends `File`).

Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

We won't attempt any full implementation here. We'll only make a skeleton to show one approach (see Listing 4.1). Fleshing it out would involve an exhaustive set of methods and rigorous error checking; the example here is neither complete nor robust.

Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

Listing 4.1. Outline of an *IOString* Class

```
class IOString < String

  def initialize(str="")
    @fptr = 0
    self.replace(str)
  end

  def open
    @fptr = 0
  end

  def truncate
    self.replace("")
    @fptr = 0
  end

  def seek(n)
    @fptr = [n, self.size].min
  end

  def tell
    @fptr
  end

  def getc
    @fptr += 1
    self[@fptr-1].to_i
  end

  def ungetc(c)
    self[@fptr -= 1] = c.chr
  end

  def putc(c)
    self[@fptr] = c.chr
    @fptr += 1
  end

  def gets
    s = ""
    n = self.index("\n",@fptr)
    s = self[@fptr..n].dup
    @fptr += s.length
    s
  end
  def puts(s)
    self[@fptr..@fptr+s.length-1] = s
    @fptr += s.length
  end

end



ios = IOString.new("abcdefghijkl\nABC\n123")

ios.seek(5)
ios.puts("xyz")

puts ios.tell      # 8

puts ios.dump      # "abcdexyzijkl\nABC\n123"

c = ios.getc
puts "c = #{ c} "    # c = 105
```


Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

```
ios.ungetc(?w)

puts ios.dump      # "abcdexyzwjkl\nABC\n123"

puts "Ptr = #{ ios.tell} "

s1 = ios.gets      # "wjkl"
s2 = ios.gets      # "ABC"
```

Reading Data Embedded in a Program

When you were 12 years old and you learned BASIC by copying programs out of magazines, you may have used a `DATA` statement for convenience. The information was embedded in the program, but it could be read as if it originated outside.

Should you ever want to, you can do much the same thing in Ruby. The directive `__END__` at the end of a Ruby program signals that embedded data follows. This can be read using the global constant `DATA`, which is an `IO` object like any other. (Note that the `__END__` marker must be at the beginning of the line on which it appears.) Here's an example:

```
# Print each line backwards...
DATA.each_line do |line|
  puts line.reverse
end
__END__
A man, a plan, a canal... Panama!
Madam, I'm Adam.
,siht daer nac uoy fI
.drah oot gnikrow neeb ev'uoy
```

Reading Program Source

Suppose you want to access the source of your own program. This can be done using a variation on a trick we used elsewhere (see the section titled "Reading Data Embedded in a Program").

The global constant `DATA` is an `IO` object that refers to the data following the `__END__` directive. However, if you perform a `rewind` operation, it will reset the file pointer to the beginning of the program source.

The following program will produce a listing of itself with line numbers (it is not particularly useful, but maybe you can find some other good use for this capability):

```
DATA.rewind
num = 1
DATA.each_line do |line|
```


Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

```
  puts "#{ '%03d' % num}   #{ line} "
  num += 1
end
__END__
```

Note that the `__END__` directive *is* necessary; without it, `DATA` cannot be accessed at all.

Performing Newline Conversion

One of the annoyances of dealing with different operating systems is that they may have different concepts of what an end-of-line character is. The "common sense" newline is a carriage return (CR) followed by a linefeed (LF); but in the earliest days of Unix, the decision was made to store only the linefeed, thus saving an entire byte per line of text. (This was when 512KB was a *lot* of memory.) Today we might expect that a text file is a text file, but in moving files between, say, a Unix machine and a Windows machine, we are bitten over and over by the newline problem.

Therefore, we offer a little solution here. The `gets` method will honor either kind of newline, and the `puts` method will always write in the native format. That means this same code will convert to native format on either kind of operating system. We show it here as a simple filter that reads standard input and writes to standard output:

```
while line = gets
  puts line
end
```

Another case might result from receiving an entire file in an unknown OS format. Whereas Unix variants use LF for their newline, and Windows versions use CR-LF, we have yet another possibility with Mac OS, which uses just CR. This very situation arises on the Web when a `TEXTAREA` is processed, among other times.

Here is one way to handle this situation, in which we wish to save the contents of a text area into a file on our Linux Web server:

```
tmp=cgi.params["mytextarea"].to_s
File.open("newfile","w") do |f|
  newstring = tmp.gsub!(/\r\n/m,"\n") or
              tmp.gsub!(/\r/m,"\n") or tmp
  newstring.each {  |line| f.puts line }
end
```

The first `gsub!` looks for the CR-LF pair from a PC. If it finds none, it returns `nil`, meaning the `or` will allow the next `gsub!` to execute, which works on Mac OS–based files. If no carriage returns are found, the originally captured string is used. After conversion, the string is written line by line to a file.

Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

Working with Temporary Files

In many circumstances we need to work with files that are all but anonymous. We don't want the trouble of naming them or making sure there is no name conflict, and we don't want to bother with deleting them.

All these issues are addressed in the `Tempfile` library. The `new` method (alias `open`) will take a base name as a "seed string" and will concatenate onto it the process ID and a unique sequence number. The optional second parameter is the directory to be used; it defaults to the value of environment variable `TMPDIR`, `TMP`, or `TEMP`, and finally the value `"/tmp"`.

The resulting `IO` object may be opened and closed many times during the execution of the program. Upon termination of the program, the temporary file will be deleted.

The `close` method has an optional flag; if set to `true`, the file will be deleted immediately after it is closed (instead of waiting until program termination). The `path` method will return the actual pathname of the file, should you need it. Here's an example:

```
require "tempfile"

temp = Tempfile.new("stuff")
name = temp.path              # "/tmp/stuff17060.0"
temp.puts "Kilroy was here"
temp.close

# Later...
temp.open
str = temp.gets               # "Kilroy was here"
temp.close(true)              # Delete it NOW
```

Changing and Setting the Current Directory

The current directory may be determined by the use of `Dir.pwd` or its alias `Dir.getwd`; these abbreviations historically stand for *print working directory* and *get working directory,* respectively. In a Windows environment, the backslashes will probably show up as normal (forward) slashes.

The method `Dir.chdir` may be used to change the current directory. On Windows, the logged drive may appear at the front of the string. Here's an example:

```
Dir.chdir("/var/tmp")
puts Dir.pwd           # "/var/tmp"
puts Dir.getwd         # "/var/tmp"
```


Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

Changing the Current Root

On most Unix variants, it is possible to change the current process's idea of where the root or "slash" is. This is typically done for security reasons (for example, when running unsafe or untested code). The `chroot` method will set the new root to the specified directory:

```
Dir.chdir("/home/guy/sandbox/tmp")
Dir.chroot("/home/guy/sandbox")
puts Dir.pwd                      # "/tmp"
```

Iterating over Directory Entries

The class method `foreach` is an iterator that will successively pass each directory entry into the block. The instance method `each` behaves the same way. Here's an example:

```
Dir.foreach("/tmp") {  |entry| puts entry }

dir = Dir.new("/tmp")
dir.each  {  |entry| puts entry }
```

Both of the preceding code fragments will print the same output (the names of all files and subdirectories in `/tmp`).

Getting a List of Directory Entries

The class method `Dir.entries` will return an array of all the entries in the specified directory:

```
list = Dir.entries("/tmp")  # %w[. .. alpha.txt beta.doc]
```

Creating a Chain of Directories

Sometimes we want to create a chain of directories where the intermediate directories themselves don't necessarily exist yet. At the Unix command line, we would use `mkdir -p` for this.

In Ruby code, we can do this by using the `makedirs` method, which the `ftools` library adds to `File`:

```
require "ftools"

File.makedirs("/tmp/these/dirs/need/not/exist")
```


Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

Deleting a Directory Recursively

In the Unix world, we can type `rm -rf dir` at the command line and the entire subtree starting with `dir` will be deleted. Obviously, we should exercise caution in doing this.

If you need a piece of code to accomplish this, here it is:

```
def delete_all(dir)
  Dir.foreach(dir) do |e|
    # Don't bother with . and ..
    next if [".",".."].include? e
    fullname = dir + File::Separator + e
    if FileTest::directory?(fullname)
      delete_all(fullname)
    else
      File.delete(fullname)
    end
  end
  Dir.delete(dir)
end


delete_all("dir1")   # Remove dir1 and everything under it!
```

Finding Files and Directories

Here, we make use of the standard library `find.rb` to create a method that will find one or more files and return the list of files as an array. The first parameter is the starting directory; the second is either a filename (that is, a string) or a regular expression:

```
require "find"

def findfiles(dir, name)
  list = []
  Find.find(dir) do |path|
    Find.prune if [".",".."].include? path
    case name
      when String
        list << path if File.basename(path) == name
      when Regexp
        list << path if File.basename(path) =~ name
    else
      raise ArgumentError
    end
  end
  list
end

findfiles "/home/hal", "toc.txt"
# ["/home/hal/docs/toc.txt", "/home/hal/misc/toc.txt"]

findfiles "/home", /^[a-z]+.doc/
# ["/home/hal/docs/alpha.doc", "/home/guy/guide.doc",
#  "/home/bill/help/readme.doc"]
```


Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

Performing Higher-Level Data Access

Frequently we want to store and retrieve data in a more transparent manner. The `Marshal` module offers simple object persistence, and the `PStore` library builds on that functionality. Finally, the `dbm` library is used like a hash stored permanently on disk. It does not truly belong in this section, but it is rather too simple to put in the database section.

Simple Marshaling

In many cases, we would like to create an object and simply save it for use later. Ruby provides rudimentary support for such object persistence or *marshaling*. The `Marshal` module enables programs to *serialize* and *unserialize* Ruby objects in this way:

```
# array of elements [composer, work, minutes]
works = [["Leonard Bernstein","Overture to Candide",11],
         ["Aaron Copland","Symphony No. 3",45],
         ["Jean Sibelius","Finlandia",20]]
# We want to keep this for later...
File.open("store","w") do |file|
  Marshal.dump(works,file)
end

# Much later...
File.open("store") do |file|
  works = Marshal.load(file)
end
```

This technique does have the shortcoming that not all objects can be dumped. If an object includes an object of a fairly low-level class, it cannot be marshaled; these include `IO`, `Proc`, and a few others. Singleton objects also cannot be serialized.

More Complex Marshaling

Sometimes we want to customize our marshaling to some extent. Creating `_load` and `_dump` methods will enable you to do this. These hooks are called when marshaling is done so that you are handling your own conversion to and from a string.

In this example, a person has been earning five percent interest on his beginning balance since he was born. We don't store the age and the current balance since they are a function of time:

```
class Person

  def initialize(name,birthdate,beginning)
    @name = name
    @birthdate = birthdate
    @beginning = beginning
```


Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

```
    @age = (Time.now - @birthdate)/(365*86400)
    @balance = @beginning*(1.05**age)
  end

  def _dump(depth)
    # (We ignore depth here)
    @name + ":" + @birthdate + ":" + @beginning
  end

  def _load(str)
    a, b, c = str.split(":")
    Person.new(a,b,c)
  end
  # Other methods...

end
```

When an object of this type is saved, the age and current balance will not be stored; when the object is "reconstituted," they will be computed.

Performing Limited "Deep Copying" Using `Marshal`

Ruby has no "deep copy" operation. The methods `dup` and `clone` may not always work as you would initially expect. An object may contain nested object references that turn a copy operation into a game of Pick Up Sticks.

We offer here a way to handle a restricted deep copy (it is restricted because it is still based on `Marshal` and has the same inherent limitations):

```
def deep_copy(obj)
   Marshal.load(Marshal.dump(obj))
end

a = deep_copy(b)
```

Better Object Persistence with `PStore`

The `PStore` library provides file-based persistent storage of Ruby objects. A `PStore` object can hold a number of Ruby object hierarchies. Each hierarchy has a *root* identified by a key. Hierarchies are read from a disk file at the start of a transaction and written back at the end. Here's an example:

```
require "pstore"

# save
db = PStore.new("employee.dat")
db.transaction do
    db["params"] = { "name" => "Fred", "age" => 32,
                     "salary" => 48000 }
end
```


Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

```
# retrieve
require "pstore"
db = PStore.new("employee.dat")
emp = nil
db.transaction {  emp = db["params"] }
```

Typically, within a transaction block we use the `PStore` object passed in. We can also use the receiver directly, however.

This technique is transaction oriented; at the start of the block, data is retrieved from the disk file to be manipulated. Afterward, it is transparently written back out to disk.

In the middle of a transaction, we can interrupt with either `commit` or `abort`; the former will keep the changes we have made, whereas the latter will throw them away. Refer to the longer example in Listing 4.2.

Listing 4.2. Using *`PStore`*

```
require "pstore"

store = PStore.new("objects")
store.transaction do |s|

  a = s["my_array"]
  h = s["my_hash"]

  # Imaginary code omitted, manipulating
  # a, h, etc.

  # Assume a variable named "condition" having
  # the value 1, 2, or 3...

  case condition
    when 1
      puts "Oops... aborting."
      s.abort   # Changes will be lost.
    when 2
      puts "Committing and jumping out."
      s.commit  # Changes will be saved.
    when 3
      # Do nothing...
  end

  puts "We finished the transaction to the end."
  # Changes will be saved.

end
```

Within a transaction, you can also use the method `roots` to return an array of roots (or `root?` to test membership). Also, the `delete` method is available to remove a root. Here' an example:

Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

```
store.transaction do |s|
  list = s.roots          # ["my_array","my_hash"]
  if s.root?("my_tree")
    puts "Found my_tree."
  else
    puts "Didn't find # my_tree."
  end
  s.delete("my_hash")
  list2 = s.roots         # ["my_array"]
end
```

Using the dbm Library

The `dbm` library is a simple platform-independent, string-based hash, file-storage mechanism. It stores a key and some associated data, both of which must be strings. Ruby's `dbm` interface is built in to the standard installation.

To use this class, create a `dbm` object associated with a filename and work with the string-based hash however you want. When you have finished, you should close the file. Here's an example:

```
require 'dbm'

d = DBM.new("data")
d["123"] = "toodle-oo!"
puts d["123"]        # "toodle-oo!"
d.close

puts d["123"]        # RuntimeError: closed DBM file

e = DBM.open("data")
e["123"]                # "toodle-oo!"
w=e.to_hash                # { "123"=>"toodle-oo!"}
e.close

e["123"]                # RuntimeError: closed DBM file
w["123"]                # "toodle-oo!
```

Here, `dbm` is implemented as a single class that mixes in `Enumerable`. The two (aliased) class methods, `new` and `open`, are singletons, which means you may only have one `dbm` object per data file open at any given time:

```
q=DBM.new("data.dbm")    #
f=DBM.open("data.dbm")   # Errno::EWOULDBLOCK:
                         #   Try again - "data.dbm"
```


Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

There are 34 instance methods, many of which are aliases or similar to the hash methods. Basically, if you are used to manipulating a real hash in a certain way, there is a good chance you can apply the same operation to a `dbm` object.

The method `to_hash` will make a copy of the hash file object in memory, and `close` will permanently close the link to the hash file. Most of the rest of the methods are analogous to hash methods, but there are no `rehash`, `sort`, `default`, and `default=` methods. The `to_s` method just returns a string representation of the object ID.

Connecting to External Databases

Ruby can interface to various databases, thanks to the development work of many different people. These range from monolithic systems such as Oracle down to the more petite MySQL. We have included the CSV (comma-separated values) format here for some measure of completeness.

The level of functionality provided by these packages will continually be changing. Be sure to refer to an online reference for the latest information. The Ruby Application Archive (RAA) is always a good starting point.

Interfacing to MySQL

Ruby's MySQL interface is the most stable and fully functional of its database interfaces. It is an extension and must be installed after both Ruby and MySQL are installed and running. If you upgrade Ruby, you will need to reinstall it. Installation itself is simple, using Ruby's `make` process.

There are three steps to using this module once you have it installed. First, load the module in your script; then connect to the database. Finally, work with your tables. Connecting requires the usual parameters for host, username, password, database, and so on, as shown here:

```
require 'mysql'

m = Mysql.new("localhost","ruby","secret","maillist")
r = m.query("SELECT * FROM people ORDER BY name")
r.each_hash do |f|
  print "#{ f['name']}  - #{ f['email']} "
end
```

Partial output is shown here:

```
John Doe - jdoe@rubynewbie.com
Fred Smith - smithf@rubyexpert.com
```


Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

```
Don Jackson - don@perl2.com
Jenny Williams - jwill27@miss-code.com
```

The class methods `Mysql.new` and `MysqlRes.each_hash` are very useful, along with the instance method `query`.

The module is composed of four classes: `Mysql`, `MysqlRes`, `MysqlField`, and `MysqlError`, as described in the README file. We summarize some useful methods here, but you can always find more information in the actual documentation.

The class method `Mysql.new` takes several string parameters, all defaulting to `nil`, and it returns a connection object. The parameters are `host`, `user`, `passwd`, `db`, `port`, `sock`, and `flag`. Aliases for `new` are `real_connect` and `connect`.

The methods `create_db`, `select_db`, and `drop_db` all take a database name as a parameter; they are used as shown here (note that the method `close` will close the connection to the server):

```
m=Mysql.new("localhost","ruby","secret")
m.create_db("rtest")     # Create a new database
m.select_db("rtest2")    # Select a different database
m.drop_db("rtest")       # Delete a database
m.close                  # Close the connection
```

The method `list_dbs` will return a list of available database names in an array:

```
dbs = m2.list_dbs        # ["people","places","things"]
```

The `query` takes a string parameter and returns a `MysqlRes` object by default. Depending on how `query_with_result` is set, it may return a `Mysql` object.

In the event of an error, the error number can be retrieved by `errno`; `error`, on the other hand, will return the actual error message. Here's an example:

```
begin
  r=m.query("create table rtable
   (
     id int not null auto_increment,
     name varchar(35) not null,
     desc varchar(128) not null,
     unique id(id)
   )")

# exception happens...

rescue
```


Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

```
  puts m.error
  # Prints: You have an error in your SQL syntax
  # near 'desc varchar(128) not null ,
  #   unique id(id)
  # )' at line 5"
  puts m.errno
  # Prints 1064
  # ('desc' is reserved for descending order)
end
```

A few useful instance methods of `MysqlRes` are summarized in the following list:

- `fetch_fields`— Returns an array of `MysqlField` objects from the next row.
- `fetch_row`— Returns an array of field values from the next row.
- `fetch_hash(with_table=false)`— Returns a hash containing the next row's field names and values.
- `num_rows`— Returns the number of rows in the result set.
- `each`— An iterator that sequentially returns an array of field values.
- `each_hash(with_table=false)`— An iterator that sequentially returns a hash of `{fieldname => fieldvalue}`. (Use `x['field name']` to get the field value.)

Here are some instance methods of `MysqlField`:

- `name`— Returns the name of the designated field
- `table`— Returns the name of table to which the designated field belongs
- `length`— Returns the defined length of the field
- `max_length`— Returns the length of the longest field from the result set
-

Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

`hash`— Returns a hash with a name and values for name, table, def, type, length, max_length, flags, and decimals

The material here is always superseded by online documentation. For more information, see the MySQL Web site (www.mysql.com) and the Ruby Application Archive.

Interfacing to PostgreSQL

An extension is available from the RAA that provides access to PostgreSQL (it works with PostgreSQL 6.5/7.0).

Assuming you already have PostgreSQL installed and set up (and you have a table named testdb), you merely need to follow essentially the same steps as used with other database interfaces in Ruby: Load the module, connect to the database, and then do your work with the tables. You'll probably want a way of executing queries, getting the results of a "select" operation back, and working with transactions. Here's an example:

```
require 'postgres'
conn = PGconn.connect("",5432, "", "", "testdb")
conn.exec("create table rtest ( number integer default 0 );")
conn.exec("insert into rtest values ( 99 )")
res = conn.query("select * from rtest")
# res id [["99"]]
```

The `PGconn` class contains the `connect` method, which takes the typical database connection parameters, such as host, port, database, username, and login, but it also takes options and `tty` parameters in positions three and four. We have connected in our example to the Unix socket via a privileged user, so we don't need a username and password. Also, the host, options, and `tty` parameters are left empty. The port must be an integer, whereas the others are strings. An alias for this is the `new` method.

The next thing of interest is working with our tables; this requires some means to perform queries. The instance methods `PGconn#exec` and `PGconn#query` are just what we need.

The `exec` method sends its string parameter as a SQL query request to PostgreSQL, and it returns a `PGresult` instance on success. On failure, it raises a `PGError` exception.

The `query` method also sends its string parameter as a SQL query request to PostgreSQL. However, it returns an array on success. The returned array is actually an array of tuples. On failure, it returns `nil`, and error details can be obtained by the `error` method call.

A special method, called `insert_table`, is available for inserting values into a specific table. Despite the name, `insert_table` actually means "insert into table." This method returns a `PGconn` object. Here's an example:

Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

```
conn.insert_table("rtest",[[34]])
res = conn.query("select * from rtest")
# res is [["99"], ["34"]]
```

This inserts one row of values into the table rtest. For this simple example, there is only one column to begin with. Notice that the `PGresult` object `res` shows updated results with two tuples. We will discuss `PGresult` methods shortly.

Other potentially useful methods from the `PGconn` class include the following:

- `db`— Returns the connected database name.
- `host`— Returns the connected server name.
- `user`— Returns the authenticated username.
- `error`— Returns the error message about the connection.
- `finish`— Close the backend connection.
- `loimport(file)` — Imports a file to a large object; returns the `PGlarge` instance on success. On failure, this method raises the `PGError` exception.
- `loexport(oid, file)` — Saves a large object of `oid` to a file.
- `locreate([mode])` — Returns the `PGlarge` instance on success. On failure, it raises the `PGError` exception.
- `loopen(oid, [mode])` — Opens a large object of `oid`; returns the `PGlarge` instance on success. The mode argument specifies the mode for the opened large object, which is either `INV_READ` or `INV_WRITE` (if the mode is omitted, the default is `INV_READ`).

Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

`lounlink(oid)` — Unlinks (deletes) the `Postgres` large object of `oid`.

Notice that the last five methods of `PGconn` involve objects of the `PGlarge` class. The `PGlarge` class has specific methods for accessing and changing its own objects. (The objects are created as a result of the `PGconn` instance methods `loimport`, `locreate`, and `loopen` from the preceding list.)

Here is a list of `PGlarge` methods:

- `open([mode])` — Opens a large object. The mode argument specifies its mode (see `PGconn#loopen`).
- `close` — Closes a large object (also closed when it is garbage collected).
- `read([length])` — Attempts to read "length" bytes from a large object. If no length is given, all data is read.
- `write(str)` — Writes the string to the large object and returns the number of bytes written.
- `tell` — Returns the current position of the pointer.
- `seek(offset, whence)` — Moves the pointer to `offset`. The possible values for `whence` are `SEEK_SET`, `SEEK_CUR`, and `SEEK_END` (or `0`, `1`, and `2`).
- `unlink` — Deletes a large object.
- `oid` — Returns the large object `oid`.
- `size` — Returns the size of a large object.
- `export(file)` — Saves a large object of `oid` to a file.

Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

Of more interest to us are the instance methods of the `PGresult` class, which are created as the result of queries. Use `PGresult#clear` when finished with these objects to improve memory performance. Here's a list of these methods:

- `result`— Returns the query result tuple in the array.
- `each`— An iterator.
- `[]`— An accessor.
- `fields`— Returns the array of the fields of the query result.
- `num_tuples`— Returns the number of tuples of the query result.
- `fieldnum(name)`— Returns the index of the named field.
- `type(index)`— Returns an integer corresponding the type of the field.
- `size(index)`— Returns the size of the field in bytes. A value of `-1` indicates the field is variable length.
- `getvalue(tup_num, field_num)`— Returns the field value for the given parameters. `tup_num` is the same as row number.
- `getlength(tup_num, field_num)`— Returns the length of the field in bytes.
- `cmdstatus`— Returns the status string of the last query command.
- `clear`— Clears the `PGresult` object.

Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

Working with CSV Data

The CSV format is something you may have had to deal with if you have ever worked with spreadsheets or databases. Fortunately, Hiroshi Nakamura has created a module for Ruby and has made it available in the Ruby Application Archive.

This is not a true database system. However, we felt that a discussion of it fits here better than anywhere else.

The CSV module (`csv.rb`) will parse or generate data in CSV format. The module author defines this format as follows:

```
Record separator: CR + LF
Field separator: comma (,)
Quote data with double quotes if it contains CR, LF, or comma
Quote double quote by prefixing it with another double quote (" -> "")
Empty field with quotes means null string (data,"",data)
Empty field without quotes means NULL (data,,data)
```

There are two ways to use this module: Parse/create single lines, and read/write through a file parsing/creating records sequentially. In the latter case, you will be dealing with arrays of column data objects instead of arrays of strings. The latter method requires the use of the record separator and the `isNull` flag of the column data object.

First, let's look at the handling of single, nonterminated lines. The method `CSV::parse` will parse the specified CSV-formatted string as a single line and return an array of strings; the method `CSV::create` will take the specified array of strings and create a single CSV-formatted line.

Suppose we have a data file `data.csv`, as shown here:

```
"name","age","salary"
"mark",29,34500
"joe",42,32000
"fred",22,22000
"jake",25,24000
"don",32,52000
```

We can process this file as follows:

```
require 'csv'
IO.foreach("data.csv") {  |f| p CSV::parse(f.chomp) }
# Output:
# ["name", "age", "salary"]
# ["mark", "29", "34500"]
# ["joe", "42", "32000"]
# ["fred", "22", "22000"]
```


Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

```
# ["jake", "25", "24000"]
# ["don", "32", "52000"]
```

We could also process each array and write it back to a file:

```
File.open("newdata.csv","w") do |file|
  IO.foreach("data.csv") do |line|
     a=CSV::parse line.chomp
     if a[1].to_i > 20
       s=a[2].to_i
       a[2]=s*1.1  # 10% raise
     end
    file.puts CSV::create(a)
  end
end
```

Now let's take a look at the multiline parsing commands. These commands handle data differently and do expect to see the record terminator CR-LF. The `parseLine` method populates an array with objects that contain the parsed field data and a flag for null data. Likewise, the `createLine` method uses such objects to generate CSV-formatted output.

To demonstrate, we can use the `data.csv` file again:

```
require 'csv'

file_str = File.open("data.csv").read
rec = []; pos=nil; i=-1
c,pos = CSV::parseLine(file_str,
                       pos.to_i,
                       rec[(i+=1)]=[]) until pos == 0
```

Note that if your data records are not separated by CR-LF, there will be a discrepancy. To ensure that your data records are separated by CR-LF, you might perform `#gsub!(/\n/,"\r\n")` on the file string beforehand.

You will still have to go through another step before you can use the information encapsulated inside the column data objects. However, this would be the only means available if you wish to detect null values. Otherwise, it would be better to use the simpler `CSV::parse` method because large files would have to be parsed line by line anyway.

Here is an example going the other way:

```
cd=[]
cd[0]=CSV::ColData.new
cd[0].data="joe"
cd[0].isNull=false
cd[1]=CSV::ColData.new
```


Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17

```
cd[1].data=27
cd[1].isNull=false
cd[2]=CSV::ColData.new
cd[2].data=32000
cd[2].isNull=false

csv_str=""
c = CSV::createLine(cd, 3, csv_str)
```

This will produce a CSV-formatted line including the CR-LF terminator. It seems like extra work, if you ask us.

Any material here is always superseded by online documentation. For more information, see the Ruby Application Archive.

Interfacing to Other Databases

Space does not permit us to delve into all the database interfaces available. Furthermore, at the time of this writing, these were of varying levels of maturity.

We'll just mention that there are several other libraries and tools in various stages of development. These will allow interfacing to Oracle, Interbase, mSql, LDAP, and others. There is also a usable ODBC driver. As always, refer to the Ruby Application Archive for the latest software and documentation.

Summary

That ends our look at files, I/O, and external data manipulation in Ruby. As always, more information can be found in any Ruby reference, and the latest versions of utilities and libraries can be found in the Ruby Application Archive.

The next chapter is a little more esoteric. In it, we discuss the dynamic features of Ruby and a number of techniques involving Ruby-specific object-oriented programming (OOP).

Ruby Way, The By Hal Fulton ISBN: 0-672-32083-5 Publisher: Sams
Print Publication Date: 2001/12/17