

● **Generating class boilerplate with string eval (cont'd)**

<code>\$student = new Student;</code>	<i>Let's use the constructor.</i>
<code>\$student->firstname('Joseph');</code>	<i>Now set first name,</i>
<code>\$student->lastname('Hall');</code>	<i>last name,</i>
<code>\$student->id('7777');</code>	<i>and id.</i>
<code>print "Name = ", \$student->firstname(),</code>	<code>Name = Joseph Hall</code>
<code> " ", \$student->lastname(), "\n";</code>	
<code>print "Id = ", \$student->id(), "\n";</code>	<code>Id = 7777</code>

It might seem that this sort of application absolutely requires the use of string eval, but it doesn't. You can achieve the same effect by using closures (see Item 29), along with assignments to typeglobs to give them globally visible names. Closures are a more difficult mechanism for most programmers to understand, though, and in this case eval is probably the best way to go.

For a real module providing this kind of functionality, check out `Class::Template`.

Item 55: Know when, and when not, to write networking code.

One of the many features that makes Perl an attractive programming language is its built-in support for TCP/IP programming. Perl and TCP/IP mix especially well because Perl's powerful text processing capabilities are very helpful in dealing with text-based Internet protocols like SMTP, NNTP, and HTTP.

Perl's support for network programming is so complete that you can write any conceivable type of Internet network application in it. Anything that you can express in C also can be expressed in Perl. You can write a web server or a news client from the ground up if you like. You can write a DNS server. You could even rewrite *sendmail*. The necessary capabilities are all there.

Don't write low-level code when you can use modules instead

While it's certainly possible to write network applications from the ground up, you should consider using existing modules to support your efforts. For example, if you want to fetch a Web page, the following will suffice:

```
use LWP::Simple;
$page = get 'http://www.effectiveperl.com/';
```

(Yes, that's right—two lines!)

Of course, you could always start out with calls to `socket`, `bind`, `connect`, `bone up on HTTP`, and so on, but I think you'll agree that this is easier. HTTP is particularly well served by Perl modules, but there are also modules for working with FTP, NNTP, SMTP, and many other Internet protocols and standards. If you want to learn more about Perl's Internet modules, you should begin by looking at `libwww-perl` (the World Wide Web library, also called LWP) and `libnet` (a collection of Internet protocol modules).

When you do write low-level networking code, don't use anachronisms

Of course, your application may be one where you are forced to write low-level networking code. For example, you might be working on a CGI script that connects to and exchanges data with a server application (possibly also written in Perl) via TCP/IP.

Networking or “sockets” code isn't easy to understand the first time you encounter it. To avoid starting from scratch, you will be tempted to look for an example to use as a starting point. This is a good idea, but you should be careful to work from an up-to-date example. Many of the older examples of sockets code in Perl that are floating around the net have various limitations, inefficiencies, and/or bugs. Let's discuss some of these problems, and how to avoid them.

First, you should always use the `Socket` module, or perhaps `IO::Socket`.² Among other things, the `Socket` module defines constants for protocol numbers and the like that will be correct for your environment. Older code may contain something like this:

```
$pf_inet = 2;  
$sock_stream = 1;  
$tcp_proto = 6;
```

These hard-coded values worked for many programmers for a long time, but once large numbers of people started using Perl on Solaris (System V) machines, network applications written in this manner began failing with mysterious “protocol not supported” messages. These values do not work

2. `IO::Socket` provides an object-oriented interface to the built-in socket functions as well as to some of the `Socket` module. This example could be rewritten to use `IO::Socket`, but it would not look dramatically different. `IO::Socket` was somewhat new at the time this example was written, so I've stuck to plain old `Socket` for now.

on all operating systems. The right way to write this code is to use the functions defined by the Socket module:

```
my $proto = getprotobyname 'tcp';
socket SERVER, PF_INET,
    SOCK_STREAM, $proto
    or die "socket: $!";
```

*Establish a socket with the
filehandle SERVER.*

Thus, we get the constant for the inet domain from the “constant” function PF_INET, the constant for the stream type from the function SOCK_STREAM, and the protocol number from the function getprotobyname. Another thing you may see in older code is the use of the pack operator to create the binary addresses that the various sockets functions require:

The old way:

```
$port = 2345;
$addr = pack 'S n a4 x8',
    $pf_inet, $port, "\0\0\0\0";
bind SERVER, $addr or
    die "bind: $!\n";
```

*This isn't exactly easy to read
or remember.*

The Socket module defines functions that do this for you in a more readable and maintainable way:

The new way:

```
$port = 2345;
bind SERVER,
    sockaddr_in($port, INADDR_ANY)
    or die "bind: $!";
```

*No more mysterious pack
strings.*

Server applications are often written so that they spawn child processes to handle incoming connections. Any time you create child processes you need to do something to ensure that they do not become “zombies.” One way to do this is to set up a SIGCHLD (child died) signal handler in the parent process. Whenever a child process exits, control transfers to the signal handler, which should then call wait to get rid of the zombie. You may have seen a variety of versions of this code, but one reasonably safe version looks something like this:

This is a slightly overblown signal handler:

```
sub REAPER {
    $SIG{CHLD} = \&REAPER;
    wait;
}
$SIG{CHLD} = \&REAPER;
```

*Reinstall if System V.
Install handler.*

It isn't necessary (or recommended) to reinstall the handler within the handler subroutine itself so long as you are on a BSD system or one that is POSIX-compliant. Nowadays the news here is likely to be good. Try the following:

```
use Config;
print "handlers stay put\n" if
    $Config{d_sigaction} eq "define";
```

You should skip reinstalling the handler if you believe your scripts will be run only on systems with POSIX signals (generally a safe bet):

```
sub REAPER { wait }
$SIG{CHLD} = \&REAPER;
```

Or, even more succinctly:

```
$SIG{CHLD} = sub { wait };           A really short version!
```

The reason to avoid the assignment to %SIG within the handler is that Perl does not yet have “safe” interrupts and may not have them for some time to come.³ So long as this is true, the less that goes into a signal handler, the better. You may have considered using:

```
$SIG{CHLD} = 'IGNORE';             Bad, bad, BAD!
```

Please don't. It works on some System V machines but you will be experiencing “Night of the Living <defunct>” on other platforms.

An example

Let's develop a pair of simple TCP/IP applications. We will write a server called `psd` that will run the `ps` command locally. The result will be returned to a client called `rps`. If you have to write both a client and a server, it's usually easiest to start writing the server, because you can probably test it using `telnet` as a client. Here is a bare-bones, slightly buggy first cut at `psd`:

● `psd`: A `ps` daemon

```
use strict;
use Socket;
```

3. Maybe I spoke too soon. As this book goes to press, a version of Perl with a safe exception-handling thread is being tested. This is “exceptionally” good news!

● **psd: A ps daemon (cont'd)**

```
my $port = 2001;
my $proto = getprotobyname 'tcp';
my $ps = '/usr/ucb/ps';
```

Or wherever it is.

```
socket SERVER, PF_INET, SOCK_STREAM, $proto
  or die "socket: $!";
```

Create a socket with filehandle SERVER, family INET, type STREAM, protocol TCP.

```
bind SERVER, sockaddr_in($port, INADDR_ANY)
  or die "bind: $!";
```

Bind socket to port 2001, allowing connections on any interface.

```
listen SERVER, 1 or die "listen: $!";
print "$0 listening to port $port\n";
```

Begin queueing connections.

```
for (;;) {
  accept CLIENT, SERVER;
```

Take a connection from the queue. It become the bidirectional filehandle CLIENT. Run ps and send the output to the client.

```
  print CLIENT `ps`;
```

Close down the connection and get another one.

```
  close CLIENT;
}
```

You can test this version of psd by running it in the background from the command line, then telnet-ing to the assigned port:

```
% psd &
[1] 29321
psd listening to port 2001
% telnet localhost 2001
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
  PID TT          S  TIME COMMAND
 10582 pts/7      S   0:01 -tcsh
... blah blah blah...
Connection closed by foreign host.
%
```

There are a couple of problems with this code that we ought to fix. If you start this version of psd, connect to it at least once, then kill it (with Control-C), and then try to restart it immediately, it may die with an error message along the lines of “address already in use.” If you wait a while, though, it will run fine. What is happening is that one or more closed connections in the TIME_WAIT state (a perfectly normal condition) are preventing the call to bind from succeeding, because bind will not by default allow more than one socket to use the same name (address and port number) at

the same time. After a few minutes, the closed connections time out completely and the name becomes available for reuse again.

Another problem is that this server will accept only a single connection at a time. The easy and customary way to have a server accept multiple simultaneous connections is to spawn a new child process to handle each incoming connection.

We will get back to the server shortly. Let's take a look at our client, `rps`:

● **rps: A remote ps client**

<pre>use strict; use Socket; my \$remote_host = shift or die "\$0: no hostname\n"; my \$port = 2001; my \$ip = inet_aton \$remote_host or die "unknown host: \$remote_host"; my \$proto = getprotobyname 'tcp'; socket PSD, PF_INET, SOCK_STREAM, \$proto or die "socket: \$!"; connect PSD, sockaddr_in(\$port, \$ip) or die "connect: \$!"; print while <PSD>; close PSD or die "close: \$!";</pre>	<p><i>Translate hostname into numeric address.</i></p> <p><i>Get TCP protocol number.</i></p> <p><i>Create a socket with filehandle PSD, family INET, type STREAM, protocol TCP.</i></p> <p><i>Establish a connection to the supplied port and address using socket psd.</i></p> <p><i>Read from remote psd.</i></p> <p><i>All done.</i></p>
---	--

You can now use `rps` instead of `telnet` to talk to `psd` (kind of a mouthful of Unix, isn't it?):

```
% psd &
[2] 29678
psd1 listening to port 2001
% rps localhost
PID TT      S  TIME COMMAND
10582 pts/7   S   0:01 -tcsh
... blah blah blah ...
%
```

Extra features are nice, up to a point, so let's add one. Let's allow the user to specify a `ps` option as an argument on the command line, which `rps` will pass on to `psd`. First, at the top of the file, add:

```
use FileHandle;
```

Then, before `my $remote_host...`, add:

```
my $option = shift if @ARGV[0] =~ /^-/;
```

Next, we have to send the option to the server. This will require changes to both rps and psd. The change to rps is simple. Before `print while <PSD>`, insert the following:

```
PSD->autoflush(1); # prettier than SELECT(PSD) and $| = 1
print PSD "$option\n";
```

We want to make sure that the option string we are sending gets sent; otherwise the server will hang. Here's psd, rewritten to incorporate an option string sent from rps and to support multiple connections:

● psd: A revised ps daemon

<pre>use strict; use Socket; my \$port = 2001; my \$proto = getprotobyname 'tcp'; my \$ps = '/usr/ucb/ps'; \$SIG{CHLD} = sub { wait }; socket SERVER, PF_INET, SOCK_STREAM, \$proto or die "socket: \$!"; setsockopt SERVER, SOL_SOCKET, SO_REUSEADDR, 1 or die "setsockopt: \$!"; bind SERVER, sockaddr_in(\$port, INADDR_ANY) or die "bind: \$!"; listen SERVER, 5 or die "listen: \$!"; print "\$0 listening to port \$port\n"; for (;;) { my \$addr = accept CLIENT, SERVER; my \$client_host = gethostbyaddr((unpack_sockaddr_in \$addr)[1], AF_INET); print "connection from \$client_host\n"; die "can't fork: \$!" unless defined (my \$kid = fork()); if (not \$kid) { my \$option = <CLIENT>; \$option =~ tr/a-zA-Z//cd; \$option = "-\$option" if \$option; print CLIENT `ps \$option`; exit; } }</pre>	<p><i>Dispose of zombies.</i></p> <p><i>Create a socket with filehandle SERVER, family INET, type STREAM, protocol TCP.</i></p> <p><i>Set SO_REUSEADDR so that we can establish multiple connections to this socket.</i></p> <p><i>Bind socket to port 2001, allowing connections on any interface.</i></p> <p><i>Begin queueing connections.</i></p> <p><i>Take a connection from the queue into CLIENT.</i></p> <p><i>Fork after accepting.</i></p> <p><i>Child here.</i></p> <p><i>Read option, then excise any unsafe stuff in it.</i></p> <p><i>Exit, lest child start accept-ing.</i></p>
--	---

● **psd: A revised ps daemon (cont'd)**

```

    } else {
      close CLIENT;
    }
  }
}

```

Parent here. We don't want to mess with CLIENT.

We've added a call to `setsockopt` that allows us to establish multiple connections on the same socket. This also will put an end to the `TIME_WAIT` behavior you may have observed before. The second parameter to `listen` has been increased to 5, which will allow us to have up to five connections queued up at once, that is, five connections that haven't yet been answered by `accept`.⁴

We now print out the hostname of the connecting machine and then fork a child process. The child process, which executes the code inside the first block of the `if` statement, reads the option sent by the client and tidies it up so that nothing bad will happen if someone sends an option like `' ; rm *'`. There is also a `SIGCHLD` handler so that we don't create zombies.

Obviously, you can go a lot farther in network programming than this simple example does. Perl supports all of the Unix networking features accessible from C, including other TCP/IP features (e.g., UDP) and Unix domain networking. As I pointed out earlier, Perl is especially convenient for dealing with text-based protocols because of its string handling and pattern matching features. In any event, as you embark on your next network programming project, remember to check the CPAN to see whether what you need has already been written. The code that you need may be there already, and if it is, it is likely to be reasonably well thought out and implemented.

Item 56: Don't forget the file test operators.

One of the more frequently heard questions from newly minted Perl programmers is, "How do I find the size of a file?" If this question is asked on the Perl newsgroup `comp.lang.perl.misc`, almost invariably there will be one response like the following:

```

($dev,$ino,$mode,$nlink,$uid,
 $gid,$rdev,$size,$atime,$mtime,
 $ctime,$blksize,$blocks) =
  stat($filename);

```

Poster must have been reading the stat man page.

4. The value 5 is a maximum in some operating systems. It is the "customary" value for the second argument to `listen`.