1 Advanced Perl Applications

In the previous article in this series, I gave a brief introduction to some of the features of perl that make it so attractive for rapidly developing applications like data formatting and report generating. In this article, I'd like to expand the range of possible applications by introducing the idea of packages. Basically, packages are collections of perl functions, either distributed with perl, or contributed by other authors. You include packages in your program with the `require` directive; perl then searches the directories stored in the `@INC` array to find the necessary programs, and then includes them. After that, you can refer to the functions in the packages in your program, and greatly expand the range of what you can do with your perl programs.

To illustrate these points, I'll provide simple programs to perform two interesting tasks; the first will use the system-provided chat2 package to communicate with the University of Michigan weather server, and the second will use Steven Brenner's cgi-lib package (available at `http://www.bio.cam.ac.uk/cgi-lib`) to produce and process an HTML form suitable for a web browser.

2 Communication with other Computers

The University of Michigan operates a computer which provides weather reports for the entire United States. In normal use, you telnet to the machine `madlab.sprl.umich.edu` through port 3000, and make a variety of menu choices to find the information you want. After displaying a banner, the following prompt appears:

```
Press Return for menu, or enter 3 letter forecast city code:
```

At this point, I would select the code "`sfo`" for my local forecast, which would be displayed on the screen. After the forecast, it displays a menu entitled "`CITY FORECAST MENU`" and prompts for a menu selection with the phrase "`Selection:`". I thought it would convenient to have a program that I could execute which would supply the city code, print the forecast, and then exit from the menu, without any intervention on my part.

To do this task in perl, one feature I repeatedly used is the ability to change the definition of the input record separator. This variable, which is known as `$/` in perl, defaults to newline, so that each record read by the `<>`

input operator will normally read one line of text. Since the menu prompts from the weather server don't have newlines, I redefine the separator to read up until the next prompt, so at each stage, I can read everything I want with one execution of the input operator. Since the chat2 package opens a socket, I can use the same perl file handle to both read from and write to the remote computer.

To use the chat2 package, you first open the required port using the function `&chat'open_port`. The apostrophe in the name indicates that the `open_port` function is part of the chat package. This should return a non-null value, unless the program was unable to make a connection. Next, you associate a local file handle with the one defined within the chat2 package; from that point on, reading from the filehandle is equivalent to reading the output from the remote machine and printing to the filehandle is equivalent to typing responses to be interpreted by the remote machine. Here's the perl program:

```
#!/usr/local/bin/perl
require 'chat2.pl';

&chat'open_port("madlab.sprl.umich.edu",3000) || die "Couldn't open port";
*S = *chat'S;

# Change input separator to read up until the first prompt
$/ = "3 letter forecast city code:";
$str = <S>;

# The plus sign suppresses paging on this particular server
print S "sfo+\n";

# Now we can read the input until it says "CITY FORECAST MENU", so we'll
# make that the record separator
$/ = "CITY FORECAST MENU";

# read up to the prompt, and eliminate the final line
($str = <S>) =~ s#\n.*$/##;

print $str;   # print the weather report
```

```
$/ = "Selection:";
$str = <S>;
# Send an "x" to exit, and we're done!
print S "x\n";
```

For communicating with standard protocols (like NNTP, HTTP, etc.), port numbers can be determined by using the system call `getservbyname` (provided in perl) or by looking at the file `/etc/services`.

3 Processing HTML Forms

The general topics of HTML, forms and configuration of HTTP servers is beyond the scope of this article. An excellent introduction to the general topic of HTML can be found at `http://www.ncsa.uiuc.edu/General/Internet/WWW/HTMLPrimer.html` an "instantaneous" introduction to forms and CGI scripts can be found at `http://kuhttp.cc.ukans.edu/info/forms/forms-intro.html`.

The basic idea behind the CGI interface is that if you point your web browser at an executable script, and if the HTTP server on the machine where the script is located permits that script to execute, the script will be run with its printed output going back to the web browser which originally called it. HTTP servers are generally configured to only allow scripts in certain directories to be executed, so check with your server's administrator to find out where to install your scripts. The cgi-lib package provides a routine called `&ReadParse` which gets the input from a form and creates an associative array, indexed by the name of the input fields in the form, which contains the values which were entered in that particular field. Since the script's output is displayed in the browser which called it, most CGI programmers write what are known as "comboforms"; that is a script which both displays the form on the browser, and processes the results.

Before this starts sounding too complicated, we should look at an example, because it really is remarkable how simple it is to produce a comboform. In the example below, the perl subroutine `doform` is written to display the blank form. The `PrintHeader` subroutine of the cgi-lib package prints the necessary "Content-type" line to inform the world that it is producing an HTML document; all other fields need to be filled in by the script. The program starts by calling the `&ReadParse` routine. If the `submit` field of the form has a value, that means that the program is being called from a form; otherwise, it's being called directly, and it simply needs to print the necessary HTML to display the form. To keep this example simple, we'll just have one fill-in-the-blank field, and a set of three checkboxes, but any of the more complex form elements can be easily accomodated with this scheme. Similarly, the results are just printed back to the browser, but the full range of perl's capabilities could be used to do something more interesting with this information.

```
#!/usr/local/bin/perl

require 'cgi-lib.pl';

# The ReadParse routine from cgi-lib.pl creates an associative array
# containing all the variables entered from the html form

&ReadParse(*input);

if($input{"submit"}){
        print &PrintHeader;
        print("<html><title>Results from Sample Form</title><body>\n");
        print("<h1>Thanks for using the Sample Form!</h1><br>\n");
        foreach $k (keys(%input)){
                printf("%s:<BR>%s<P>",$k,$input{$k});
                }
        print("</body></html>");
        }
else {
        &doform;
        }


sub doform{
# The action field in the form below is the location of this script.

print &PrintHeader;
print <<EOF;
```

```
<html>
<title>Sample Form</title><body>
<h3 align=center><b>Welcome to the Sample Form!</b></h3>
<form method="post" action="http://wherever/cgi-bin/sampleform.pl">
Fill in the blank:<br>
My name is <input name="name" size=40>
<p>
<P>Choose your favorite color:<br>
  <input name="check" type=checkbox value="Blue"> Blue<br>
  <input name="check" type=checkbox value="Green"> Green<br>
  <input name="check" type=checkbox value="Red"> Red<br>
<p>
<input type=submit name="submit" value="submit">
</form>
</body>
</html>
EOF
}
```

4 Further Resources

This article doesn't begin to show you the wide variety of packages and scripts that are available to extend perl's cababilities. As always, the first place to start is the CPAN archives (`http://www.perl.com/perl/CPAN/CPAN.html`). Another source of scripts is the Metronet archive at `http://www.metronet.com/perlinfo/scripts`