Part 1: Introduction

Perl is a scripting language that is used pervasively for Common Gateway Interface (CGI) programming on the World Wide Web. Perl has its roots in the Unix operating system, and since a large number of Web servers run on Unix, using Perl on Unix for CGI scripting is a very common configuration, if not the most common.

In this tutorial, we introduce CGI programming in Perl, for nonprogrammers. (If you are looking for a complete online Perl reference, see the Perl FAQ at http://www.perl.com/pub/v/faqs.)

A Simple Perl Script

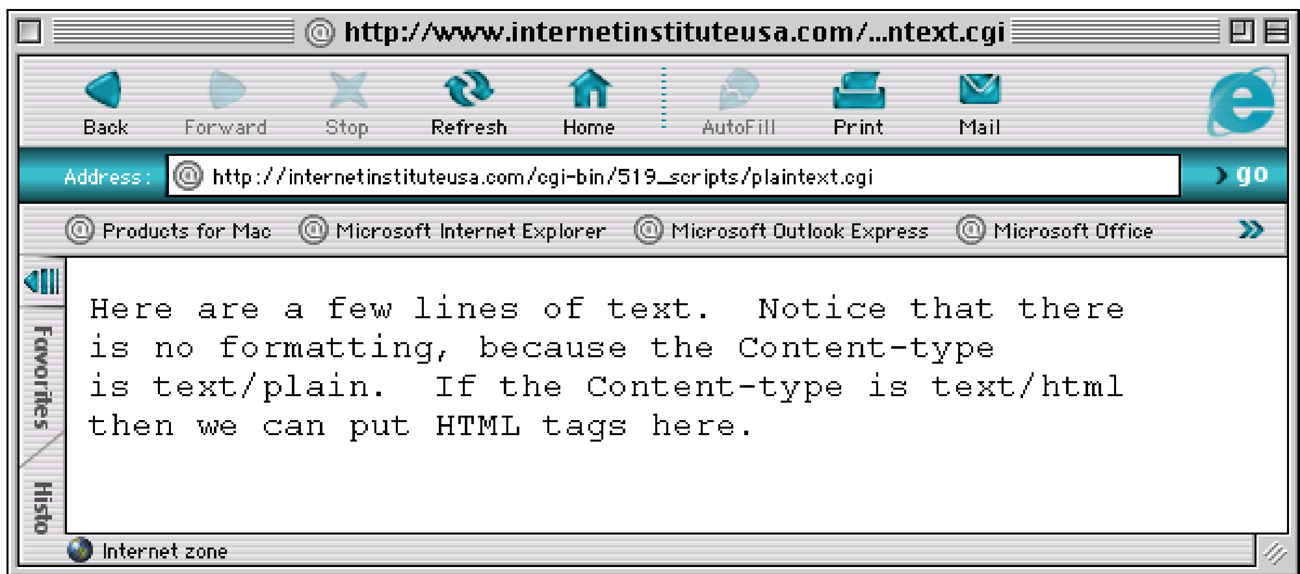
Here is an example of a simple Perl script that prints out a few lines. This is not much different than having an HTML Web page that contains these same lines, except that the script produces these same lines anew each time it is executed:

```
#!/usr/bin/perl
# The name of this file is "plaintext.cgi".
#
# NOTE: This file must be placed in the cgi-bin directory, and CGI
# must be enabled on the server machine, for this to work.
# Otherwise, the HTML code will be printed directly onto the screen.
#
# Don't forget to change the mode of this file to be readable and
# executable by all, so that it can be executed by a Web server.
# On a Unix machine, you can make this file readable and executable
# by all with:
#              chmod  a+rx  plaintext.cgi
# which can be executed from within an ftp session with:
#              ftp> !chmod a+rx plaintext.cgi
# By the way, comments begin with a pound sign (#).
#

print <<END;
Content-type: text/plain

Here are a few lines of text.  Notice that there
is no formatting, because the Content-type
is text/plain.  If the Content-type is text/html
then we can put HTML tags here.
END
```

Visit http://internetinstituteusa.com/cgi-bin/519_scripts/plaintext.cgi to see it work (see image on next page).

Let's analyze the program:

The first line identifies the location of the Perl interpreter, which is typically located in /usr/bin/perl on a Unix system. The special sequence `#!` that begins the first line indicates that the



location of the interpreter follows. These two characters must be the first two characters in the file: there can be no blank lines above this first line nor any blank space before these characters appear.

Except for the special sequence `#!` discussed above, any line that begins with a pound sign (`#`) starts a comment, which can be used to document the code, but has no influence on the behavior of the program. Anything that appears after the pound sign is ignored, through the end of the line.

The `print` command prints the text that follows. If what follows the print command appears within double quotes and ends with a semicolon, as below:

```
print "This prints out at runtime.";
```

then only the double-quoted text will be printed.

The syntax that appears in the above program:

```
print <<END;
Content-type: text/plain

Here are a few lines of text.  Notice that there
is no formatting, because the Content-type
is text/plain.  If the Content-type is text/html
then we can put HTML tags here.
END
```

prints everything that appears between the first and second instance of `END`. The character string "`END`" is not special; any other string can be used that does not appear in the text itself.

The "`Content-type`" line informs a Web browser of the type of file that the Web server will send. There must be a blank line that follows the `Content-type` line. This is a requirement of how CGI communicates between a server-side script and the Web server itself: it is not a requirement of Perl. We will look at the `Content-type` line in more detail below.

MIME Types

Multipurpose Internet Mail Extensions (MIME) types were created to allow graphics, sound, video, and other information to be sent via email. In addition to electronic mail, the HyperText Transport Protocol (HTTP), which is the protocol used for the World Wide Web, also adopted MIME.

We saw an example of the "`text`" MIME type above. MIME types also have subtypes, such as "`plain`" in the example above. Here are a few examples of type/subtype combinations:

`text/plain` -- Plain text
`text/html` -- HTML text
`image/gif` -- Graphics Interchange Format (GIF) image

A Web server sends a MIME type to the Web browser in a `Content-type` line, which is how the Web browser knows how it should handle the type of information it is being sent by the server. The Web server has a configuration file that tells it what MIME type/subtype pair to assign to each file type, based on filename extensions. For example, if a Web server receives a request from a Web browser to send file "`foo.gif`", then the server would send the browser a `Content-type` of "`image/gif`", unless it is configured to send a different `Content-type` for a `.gif` filename extension.

CGI scripts need to be handled differently than ordinary file, since a CGI script can produce any MIME type. A typical default situation is for a Web server to assume that a MIME type of `text/plain` is being produced by a CGI script, unless the CGI script explicitly produces its own `Content-type` line, as our script above does for `text/plain`.

The CGI script below sends a MIME type of `text/html` to the Web browser, and the result is that the Web browser interprets the embedded HTML tags appropriately:

```
#!/usr/bin/perl
# The name of this file is "htmltext.cgi".
# Don't forget to put a blank line after the Content-type line.

print <<END;
Content-type: text/html

<HTML><HEAD>
<TITLE>This is HTML Formatting</TITLE>
</HEAD>
<BODY>
<H1>This is HTML Formatting</H1>
The Content-type is changed to text/html, which tells
the Web browser to look for HTML tags and
<B>format</B> the HTML text as appropriate <I>for the tags</I>.
<P>Here is a hyperlink<A HREF="http://internetinstituteusa.com">
http://internetinstituteusa.com</A>
</BODY></HTML>
END
```

Click on http://internetinstituteusa.com/cgi-bin/519_scripts/htmltext.cgi to see it work.

Browsers handle some MIME types natively, like the types listed above. For other MIME types, the browser will launch an external "helper application" that exists on the end-user system. Types or subtypes that begin with "`x-`" are experimental, and you can use this feature to create your own MIME type. This is a common approach for cummunicating information that may have no standard MIME type. The basic idea is to create a new experimental MIME type for the new type of information, and then update the end-user's browser so that it opens the appropriate helper application to view the data.

Let's create our own MIME type/subtype that we will call “`text/x-iiusa00xx`.” The Perl script shown below makes use of this new MIME type:

```
#!/usr/bin/perl
# The name of this file is "iiusatext.cgi".
print <<END;
Content-type: text/x-iiusa00xx

The text that appears here is of type "text/x-iiusa00xx".
It will be read by a word processor application on
the Web browser system, after the end-user chooses
a word processor.
END
```

Unless your Web browser already knows about MIME type `text/x-iiusa00xx`, it will give you the option of saving the file to disk, or locating an application that can open it. On a Windows system, choose a simple word processor like notepad. On a Macintosh, choose SimpleText. On a Unix system, choose the cat command, which is located in /bin/cat. The word processor/display program will be launched by the Web browser, and the text that is produced by the print command will be displayed. For Netscape, and possibly other browsers, this selection will be made permanently for this Content-type. (You can change this in the Web browser by selecting "Preferences..." from the menu and modifying the "applications" mappings.)

Visit http://internetinstituteusa.com/cgi-bin/519_scripts/iiusatext.cgi to see it work.

On a Unix machine, you can select the option to save the file to disk and then view it with:

```
> more iiusatext.cgi
```

Executing a Command on the Server Machine: Calendar Script

So far, our CGI scripts only send static information to a Web browser: the content is the same from one invocation to the next. This is where the power of Perl can come into play: we can execute a CGI script via a Web server, and feed the output of the program to a Web browser.

The Perl script shown below executes the Unix `cal` command, which prints a calendar for the current year. The current year is obtained by executing the Unix `date` command.

```
#!/usr/bin/perl
# The name of this file is "calendar.cgi".
#
# This example comes from "How to Set Up and Maintain a Web Site,"
# 2/e, Lincoln D. Stein, Addison Wesley, (1997), p. 474.

$CAL='/usr/bin/cal';
$DATE='/bin/date';

# Fetch the current year using the Unix date command

$year = 2000 +`$DATE +%y`;

# Fetch the text of the calendar using the cal command

chop($calendar_text=`$CAL $year`);

# Print it all out now
print <<END;
Content-type: text/html

<HTML><HEAD>
<TITLE>Calendar for Year $year</TITLE>
</HEAD><BODY>
<H1>Calendar for Year $year</H1>
<PRE>
$calendar_text
</PRE>
<HR>
<A HREF="http://internetinstituteusa.com">Internet Institute USA</A>
</BODY></HTML>
END
```

Visit http://internetinstituteusa.com/cgi-bin/519_scripts/calendar.cgi to see it work (see next page).

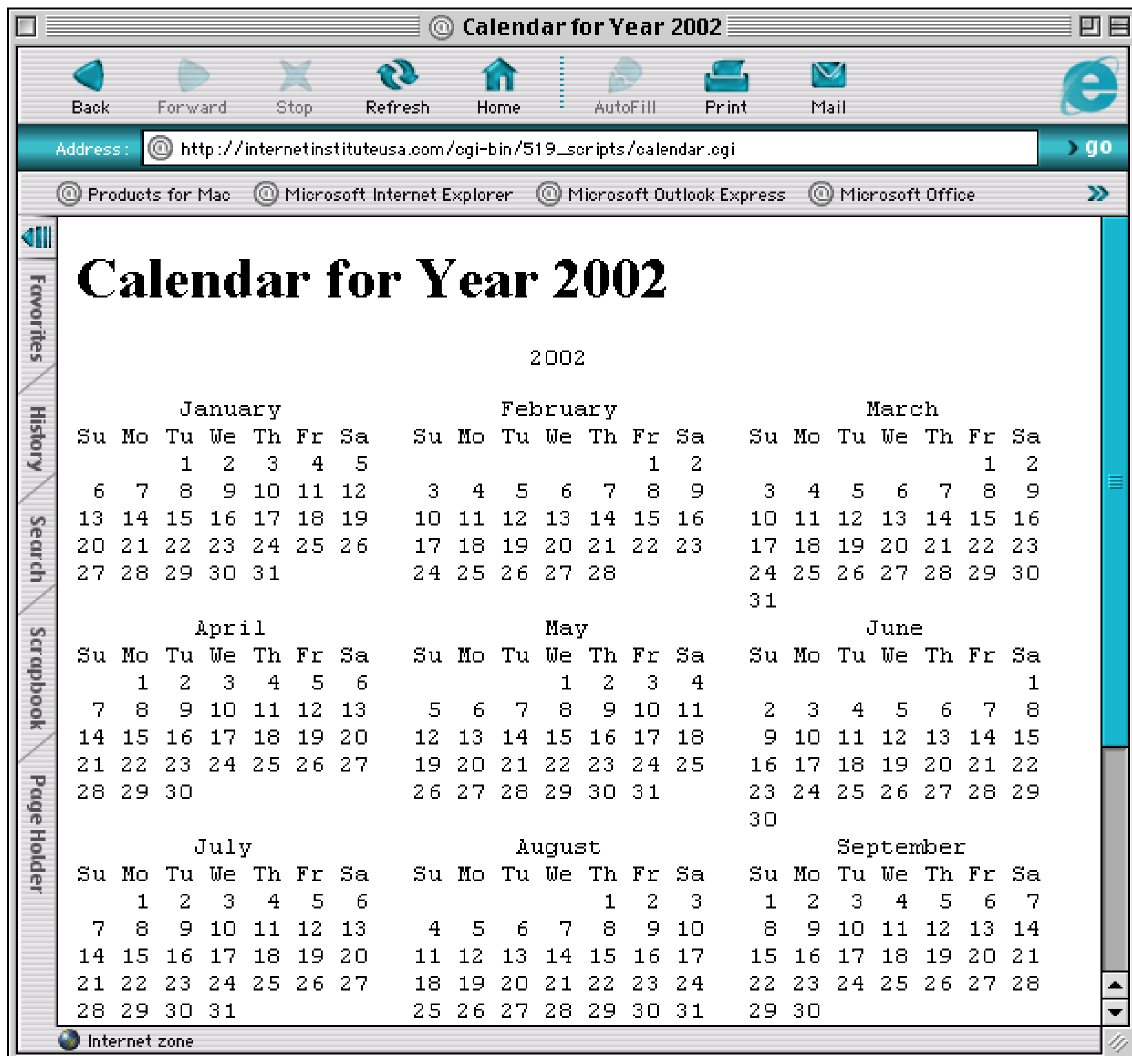
Let's analyze the program:

There are a number of new elements in this Perl program. The dollar sign (`$`) is special, and indicates that the name of a variable follows, which holds data that can change at runtime.

The line:

```
$CAL='/usr/bin/cal';
```

places the string "`/usr/bin/cal`" into variable "`$CAL`". With this construct, the variable name `$CAL` can now be used throughout the remainder of the program, as if `/usr/bin/cal` appeared in its place. The advantage of this use of variable `$CAL` is that we only need to make a single change to the variable `$CAL` to get the entire program to use a version of the `cal` program that is located somewhere else on the system. It is a simple convenience mechanism, but a powerful one.



Likewise, the line:

```
$DATE='/bin/date';
```

assigns the string `/bin/date` to variable `$DATE` so that `$DATE` can be used in its place elsewhere in the program. Notice that a program statement like:

```
$DATE='/bin/date';
```

always ends in a semicolon (`;`). Comments, which begin with a pound sign (`#`), do not need to end in a semicolon (notice that comments are interspersed throughout the program.)

On a Unix system, the command:

```
date +%y
```

will produce the last two digits of the current year. So, if the current year is 2001, then "`date +%y`" will produce "`01`". This is the way that the Unix command `date` works: it has nothing to do with Perl directly.

The Unix `cal` command will produce a 12-month calendar printout, but it needs to see the full 4-digit year as a parameter. We need to somehow construct a 4-digit year "`2001`" from the 2-digit year "`01`" which is produced by the date command. In a Perl program, enclosing a string of characters in backticks (`` ` ``) as in:

```
$year = 2000 +`$DATE +%y`;
```

causes the enclosed string to be executed as a command on the host computer, and the output produced by that command will go into variable `$year` for this example. The character string "`00`" is different than the number 00, but Perl will figure out what is intended by the construct `` 2000 +`$DATE +%y` `` and perform the expected calculation of adding 2000 to the current year, so that variable `$year` will end up with a 4-digit year. Note that this is not Y2K compliant! Starting in year 2100, the date will roll back to 2000.

Notice that variables can simply be brought into existence by naming them. We simply named `$year` above, and Perl does the right thing and creates the variable, assigns it a value after the line is executed, and allows the variable's existence to persist until the program terminates.

In the next line of code, the Unix command "`/usr/bin/cal 2001`" is executed (assuming the current year is 2001):

```
chop($calendar_text=`$CAL $year`);
```

A new variable `$calendar_text` is created to hold the output of the cal command. The portion of the program statement that is enclosed in parentheses: "`` $calendar_text=`$CAL $year` ``" will do the job by itself, if we place a semicolon (`;`) at the end (do not forget that a semicolon ends every program statement.) As shown above, a Perl built-in function named `chop` will operate on `$calendar_text`. `chop` removes the last character in `$calendar_text`. The last character is a "newline" character, which brings the cursor back to the left side of the screen in a Unix console window. Since the output will be displayed in a Web browser instead of a console window, this newline character can be removed. If we leave out `chop` and use the line:

```
$calendar_text=`$CAL $year`;
```

instead of:

```
chop($calendar_text=`$CAL $year`);
```

then an extra blank line will appear at the end of the calendar. For this example, no one would notice, but it is good programming style to keep the code clean and remove extraneous characters that are not needed.

All that is left now is to print the calendar using the `print <<END;` construct that we used above. We can embellish it with HTML code as we like:

```
print <<END;
Content-type: text/html
```

```
<HTML><HEAD>
<TITLE>Calendar for Year $year</TITLE>
</HEAD><BODY>
<H1>Calendar for Year $year</H1>
<PRE>
$calendar_text
</PRE>
<HR>
<A HREF="http://internetinstituteusa.com">Internet Institute USA</A>
</BODY></HTML>
END
```

Notice that the position of variable `$calendar_text`, which holds the entire output of the Unix `cal` program, is embedded in the middle of other text. We can intermix HTML code and Perl variables to achieve complex behavior. This is just a simple example: we will see more complex examples as we continue through the tutorial.

Environment Variables

Our Perl scripts up to this point behave the same way, regardless of what may be different in the Web browser. The calendar program above behaves differently when the year changes, but all browsers see the same change when it takes place on January 1 and then there are no more changes for another year.

We can customize the response of a Perl script to correspond to information a Web browser makes available through CGI. This information is contained in *environment variables*, which differ from Perl variables. Environment variables are created by the runtime environment, which for this discussion, is a Web server. The environment variables are made available to a Perl program in the associative array `%ENV`, which is described below.

Perl has three data types:

- the *scalar*, which we have already seen in the `$variable_name` usage above. A scalar can hold a single item, such as a number, or a string of characters (there may be many characters, but they are contained in a single character string.) Here is an example assignment:

  ```
  $year = 2001;
  ```

- the *array*, which can hold a number of items, such as scalars. An array name starts with a commercial at sign (`@`), as in `@baz`. Individual elements of an array can be accessed by indexing into the array, with the first element starting at index 0: `$foo = $baz[0]`. The square brackets indicate an element of array `@baz`. Here is an example assignment:

  ```
  @baz = (10, 25, 44);
  ```

- the *associative array*, which is the same thing as an array, except that each element (the *value*) is indexed by a name (the *key*) instead of by a number. The name of an associative array starts

with a percent sign (%), and an element is accessed using curly braces {}. So, an associative array %teams can be initialized with three key-value pairs like this:

```
%teams = ('Scarlet Knights' => 'Rutgers', 'Huskies' => 'UCONN', 'Eagles'
=> 'Boston College');
```

and then a statement like:

```
$name_of_college = $teams{'Huskies'};
```

will place "UCONN" in $name_of_college.

Here is a Perl script that lists environment variables that are available via CGI, which are in associative array %ENV:

```
#!/usr/bin/perl

print "Content-type: text/plain\n\n";

print "CGI revision used by server: ", $ENV{'GATEWAY_INTERFACE'}, "\n";
print "Server name: ", $ENV{'SERVER_NAME'}, "\n";
print "Name of server software: ", $ENV{'SERVER_SOFTWARE'}, "\n";
print "Protocol: ", $ENV{'SERVER_PROTOCOL'}, "\n";
print "Server port: ", $ENV{'SERVER_PORT'}, "\n";
print "Request method: ", $ENV{'REQUEST_METHOD'}, "\n";
print "Path info: ", $ENV{'PATH_INFO'}, "\n";
print "Translated path: ", $ENV{'PATH_TRANSLATED'}, "\n";
print "Script name: ", $ENV{'SCRIPT_NAME'}, "\n";
print "Document root: ", $ENV{'DOCUMENT_ROOT'}, "\n";
print "Query string: ", $ENV{'QUERY_STRING'}, "\n";
print "Remote host: ", $ENV{'REMOTE_HOST'}, "\n";
print "Remote address: ", $ENV{'REMOTE_ADDR'}, "\n";
print "Authentication method: ", $ENV{'AUTH_TYPE'}, "\n";
print "Remote user: ", $ENV{'REMOTE_USER'}, "\n";
print "Remote identification (RFC 931): ", $ENV{'REMOTE_IDENT'}, "\n";
print "Content type: ", $ENV{'CONTENT_TYPE'}, "\n";
print "Content length: ", $ENV{'CONTENT_LENGTH'}, "\n";
print "Email address of remote user: ", $ENV{'HTTP_FROM'}, "\n";
print "MIME types client can accept: ", $ENV{'HTTP_ACCEPT'}, "\n";
print "Browser software: ", $ENV{'HTTP_USER_AGENT'}, "\n";
print "Document that points to CGI script: ", $ENV{'HTTP_REFERER'}, "\n";
```

Visit http://internetinstituteusa.com/519_scripts/environ.cgi to see it work.

Let's analyze the program:

The newline character (represented by the two characters \n) is new here. In our earlier examples, we used the

```
print <<END;
```

construct, which printed everything up to the second `END`. The `Content-type` line must be followed by a blank line, which we simply included directly in the text in the earlier examples.

This example uses separate print statements, and the newline character must be embedded in the sequence. The reason there are two newlines in:

```
print "Content-type: text/plain\n\n";
```

is that one newline ends the `Content-type` line, and the other newline corresponds to the required blank line. Any blank lines that appear in the Perl program are consumed by the Perl interpreter. So, blank lines can be inserted in a Perl program to improve readability, but will not be sent to a Web browser unless they are either explicitly inserted with the \n sequence or appear as blank lines in a "`print <<END;`" construct.

The print built-in function prints everything up to the semicolon (`;`), which terminates the statement, and so an arbitrary number of comma-separated elements can span several lines for a single print statement. But we could just as well have written the script in the style we used in our earlier examples:

```
#!/usr/bin/perl

print <<END;
Content-type: text/plain

CGI revision used by server: $ENV{'GATEWAY_INTERFACE'}
Server name: $ENV{'SERVER_NAME'}
Name of server software: $ENV{'SERVER_SOFTWARE'}
Protocol: $ENV{'SERVER_PROTOCOL'}
Server port: $ENV{'SERVER_PORT'}
Request method: $ENV{'REQUEST_METHOD'}
Path info: $ENV{'PATH_INFO'}
Translated path: $ENV{'PATH_TRANSLATED'}
Script name: $ENV{'SCRIPT_NAME'}
Document root: $ENV{'DOCUMENT_ROOT'}
Query string: $ENV{'QUERY_STRING'}
Remote host: $ENV{'REMOTE_HOST'}
Remote address: $ENV{'REMOTE_ADDR'}
Authentication method: $ENV{'AUTH_TYPE'}
Remote user: $ENV{'REMOTE_USER'}
Remote identification (RFC 931): $ENV{'REMOTE_IDENT'}
Content type: $ENV{'CONTENT_TYPE'}
Content length: $ENV{'CONTENT_LENGTH'}
Email address of remote user: $ENV{'HTTP_FROM'}
MIME types client can accept: $ENV{'HTTP_ACCEPT'}
Browser software: $ENV{'HTTP_USER_AGENT'}
Document that points to CGI script: $ENV{'HTTP_REFERER'}

END
```

Visit http://internetinstituteusa.com/cgi-bin/519_scripts/environ2.cgi to see it work.

We can embed information in the URL that we would like to make available to our Perl script. This

information is simply passed to the script by the Web server, as long as it appears after the name of the script. For example, if we add "/this_text" at the end of the URL, then it will appear in the `PATH_INFO` environment variable. Try it out, and take a look at the "Path info" field:

```
http://internetinstituteusa.com/cgi-bin/519_scripts/environ2.cgi/this_text
```

We will not cover all of the remaining environment variables in this tutorial: see Chapter 4 of *Perl and CGI for the World Wide Web*, by Elizabeth Castro, Peachpit Press, (1999) for detailed explanations. We will, however, make use of environment variables when we explore fill-out forms, next.

Query Strings

In the previous section, we saw an example of how information from a Web browser can be sent to a CGI script by way of the `PATH_INFO` environment variable. Another method of sending information to a CGI script is by way of the `QUERY_STRING` environment variable, which can be accomplished by appending a question mark to the URL for the CGI script, followed by whatever information we would like the Web browser to send to the CGI script.

As an example, let's allow the user to choose between executing the Unix `date` command and the Unix `cal` command. We can embed the desired `QUERY_STRING` in the URL. For example, if we append "`?date`" to the URL above, then we will have:

```
http://internetinstituteusa.com/cgi-bin/519_scripts/environ2.cgi?date
```

Go ahead and visit this hyperlink, and look for the `QUERY_STRING` environment variable, which should contain the text: "`date`".

Here is a Perl program that executes either the `date` command or the `cal` command, depending on the `QUERY_STRING` environment variable:

```
#!/usr/bin/perl

$CAL='/usr/bin/cal';
$DATE='/bin/date';

$query_string = $ENV{'QUERY_STRING'};

if ($query_string eq "date") {
    print "Content-type: text/plain\n\n";
    print `$DATE`;
    }

else {

# Fetch the current year using the Unix date command

$year = 2000 +`$DATE +%y`;

# Fetch the text of the calendar using the cal command
```

```
chop($calendar_text=`$CAL $year`);

# Print it all out now
print <<END;
Content-type: text/html

<HTML><HEAD>
<TITLE>Calendar for Year $year</TITLE>
</HEAD><BODY>
<H1>Calendar for Year $year</H1>
<PRE>
$calendar_text
</PRE>
<HR>
<A HREF="http://internetinstituteusa.com">Internet Institute USA</A>
</BODY></HTML>
END
}
```

Let's analyze the program:

This is essentially the same code as our calendar program above, except that the contents of the `QUERY_STRING` environment variable are placed in `$query_string`, and a decision is made as to whether `date` is executed, if it is specified in `QUERY_STRING`, or whether `cal` is executed, which will happen for all other cases, even if something other than `date` or `cal` appears in `QUERY_STRING` (as the program is now written).

The construct:

```
    if (condition) {
         # Do this
         }

     else {
         # Do this instead
     }
```

is new in this example. The full form is actually:

```
    if (condition) {
         # Do this
         }

     elsif (a different condition) {
         # Do this instead
     }

     else {
         # Do this as the default
     }
```

in which the `elsif` clause can appear several times with different conditions, or not at all.

The condition on which the `if/elsif/else` construct is based can be a test for equality, as indicated with the `eq` keyword above. The `eq` operator tests for string equality, whereas the `neq` operator tests for string inequality. The `==` operator tests for numerical equality, whereas the `!=` operator tests for numerical inequality.

The corresponding URLs are:

http://internetinstituteusa.com/cgi-bin/519_scripts/date_or_cal.cgi?date

http://internetinstituteusa.com/cgi-bin/519_scripts/date_or_cal.cgi?cal

Go ahead and try each of the hyperlinks above. This is a single CGI script that exhibits different behavior depending on what information the client sends to it via the `QUERY_STRING` environment variable.

Fill-Out Forms

A fill-out form is a more powerful CGI mechanism that lets the Web browser create the `QUERY_STRING` information "on-the-fly", based on information that the user enters. As a simple example, let's create a Perl script that allows the user to select the year that the `cal` program uses. Here is our new Perl program:

```
#!/usr/bin/perl
# Input looks like "year=2001"

$CAL='/usr/bin/cal';

# Get the year from the user

$query_string = $ENV{'QUERY_STRING'};
($field_name, $year) = split (/=/, $query_string);

# Fetch the text of the calendar using the cal command

chop($calendar_text=`$CAL $year`);

# Print it all out now
print <<END;
Content-type: text/html

<HTML><HEAD>
<TITLE>Calendar for Year $year</TITLE>
</HEAD><BODY>
<H1>Calendar for Year $year</H1>
<PRE>
$calendar_text
</PRE>
<HR>
<A HREF="http://internetinstituteusa.com">Internet Institute USA</A>
</BODY></HTML>
END
```

We need to write HTML text that puts up a fill-in box where the user can type in the year, and we also need to write HTML text that will invoke the CGI script. The following code will accomplish this:

```
<FORM ACTION="http://internetinstituteusa.com/cgi-bin/519_scripts/calendar2.cgi"
METHOD="GET">
Enter a year: <INPUT TYPE="text" NAME="year" SIZE=10>
<P>
<INPUT TYPE="submit" VALUE="Make a Calendar (GET Method)">
</FORM>
```

Here is what the user sees (go ahead, type in a year and click on the "`Make a Calendar`" button. Just for fun, take a look at September, 1752.)

Let's analyze the program:

Looking at the HTML text first, we see the `FORM` tag specifies the location of the CGI script through the use of the `ACTION` keyword. The `METHOD` specifies `GET`, which we will use as the default for the moment. We will explore aspects of the GET and POST methods in the next section.

The `INPUT` tag creates an element that causes the Web browser to take an action based on what the user does. We see two different applications of the `INPUT` tag. One application is for a `TYPE` of text, in which a fill-in field is created that holds up to 20 characters. We give an arbitrary name to this field of `year` using the `NAME` keyword. Since there can be any number of `INPUT` fields, the `NAME` attribute is important for distinguishing among the various fields in the Perl script. The corresponding `QUERY_STRING` that the Perl script sees is:

```
year=2001
```

(assuming the user enters 2001 in the fill-in field.)

The other `INPUT` tag is for the "`Make a Calendar`" button that causes the form data to be sent to the Web server when the user selects it. The `</FORM>` keyword ends the fill-out form.

Now let's take a look at the Perl script. As in the previous example, we capture the query string in variable `$query_string` with the statement:

```
$query_string = $ENV{'QUERY_STRING'};
```

We need to extract the year from the `$query_string` variable, which at this point contains:

```
year=2001
```

We can use the Perl built-in function `split` for this:

```
($field_name, $year) = split (/=/, $query_string);
```

The `split` function looks for a specific character string, the equals sign (=) for this case, and places the text that appears to the left and right of this character into variables `$field_name` and `$year`, respectively. For this program, `$field_name` is not used, but we include it anyway to show how `split` works.

The remainder of the Perl program is the same as the previous version of the calendar program.

GET vs. POST Methods

In the previous example, we used the GET method for the fill-out form. With the GET method, all of the information sent from the Web browser to the Web server is included in the URL. So, the URL generated by the fill-out form above is actually (for year 2001): http://internetinstituteusa.com/cgi-bin/519_scripts/calendar2.cgi?year=2001.

As the complexity of fill-out forms increases, the GET method becomes less practical because Web servers generally have a limit on the size of the URL that can be accepted, such as 256 characters. The POST method does not embed query string information in the URL, but instead, sends it after the URL has been sent, as part of the input stream to the Web server.

For the POST method, we need to find the

```
year=2001
```

string in the input stream rather than in the `QUERY_STRING` environment variable. The Perl program shown below does the same thing as the calendar program shown above, except that it uses the POST method.

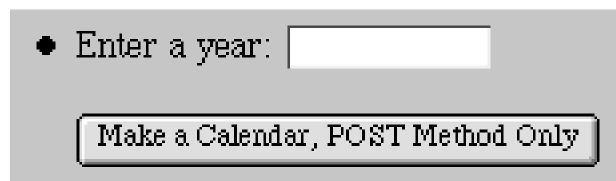
```
#!/usr/bin/perl

$CAL='/usr/bin/cal';

# Get the year from the user

$size_of_form_information = $ENV{'CONTENT_LENGTH'};

read (STDIN, $form_info, $size_of_form_information);

($field_name, $year) = split (/=/, $form_info);

# Fetch the text of the calendar using the cal command

chop($calendar_text=`$CAL $year`);
```

```
# Print it all out now
print <<END;
Content-type: text/html

<HTML><HEAD>
<TITLE>Calendar for Year $year</TITLE>
</HEAD><BODY>
<H1>Calendar for Year $year</H1>
<PRE>
$calendar_text
</PRE>
<HR>
<A HREF="http://internetinstituteusa.com">Internet Institute USA</A>
</BODY></HTML>
END
```

The `METHOD` attribute in the corresponding HTML text specifies the `POST` method:

```
<FORM ACTION="http://internetinstituteusa.com/cgi-bin/519_scripts/calendar3.cgi"
METHOD="POST">
Enter a year: <INPUT TYPE="text" NAME="year" SIZE=10>
<P>
<INPUT TYPE="submit" VALUE="Make a Calendar, POST Method Only">
</FORM>
```

Now try it out; it should work the same as for the GET method:



There is no guarantee that the HTML page that invokes the CGI script will use the GET or the POST method, and it is good programming style to create a Perl script that will behave correctly regardless of which method is used. Here is a modified version of the calendar program, that works correctly for both the GET and POST methods:

```
#!/usr/bin/perl

$CAL='/usr/bin/cal';

$request_method = $ENV{'REQUEST_METHOD'};

# Get the year from the user

if ($request_method eq "GET") {
    $form_info = $ENV{'QUERY_STRING'};
    }
else {
```

```
    $size_of_form_information = $ENV{'CONTENT_LENGTH'};
    read (STDIN, $form_info, $size_of_form_information);
    }

($field_name, $year) = split (/=/, $form_info);

# Fetch the text of the calendar using the cal command

chop($calendar_text=`$CAL $year`);

# Print it all out now
print <<END;
Content-type: text/html

<HTML><HEAD>
<TITLE>Calendar for Year $year</TITLE>
</HEAD><BODY>
<H1>Calendar for Year $year</H1>
<PRE>
$calendar_text
</PRE>
<HR>
<A HREF="http://internetinstituteusa.com">Internet Institute USA</A>
</BODY></HTML>
END
```

Now try it out; it should work the same for both the GET and POST methods:

- Enter a year:

Make a Calendar (POST Method)

Part 2: Form Processing

In this part of the tutorial, we go deeper into CGI programming using Perl. We cover more fill-out form elements like radio buttons, checkboxes, and textfields, and more sophisticated Perl constructs such as server redirection, subroutines,and query processing.

Radio Buttons, Checkboxes, Textfields, Password Fields, Scrolled Lists, and Menus

In addition to the single-line text field and the submit button that we saw previously, there are a number of other fill-out form elements such as: password fields, reset buttons, radio buttons, checkboxes, menus, scrolled lists, and multiline textfields.

Here is an example (on the next page) of a fill-out form that uses all of these elements, as well as the single-line text field and the submit button that we have previously seen: (Go ahead and fill in the fields at http://internetinstituteusa.com/cgi-bin/519_scripts and click on the "Submit the Form" button, and see what the server sends back.)

Here is the corresponding HTML code:

```
<FORM ACTION="http://internetinstituteusa.com/cgi-bin/519_scripts/
sampleform.cgi" METHOD="GET">
Enter your username: <INPUT TYPE="text" NAME="Username" SIZE=30>
<P>
Enter your password: <INPUT TYPE="password" NAME="Password" SIZE=10>
<P>
Select days you are available:
Monday <INPUT TYPE="checkbox" NAME="Monday">
Tuesday <INPUT TYPE="checkbox" NAME="Tuesday">
Wednesday <INPUT TYPE="checkbox" NAME="Wednesday">
Thursday <INPUT TYPE="checkbox" NAME="Thursday">
Friday <INPUT TYPE="checkbox" NAME="Friday">
<P>
Select a semester:<BR>
Fall: <INPUT TYPE="radio" NAME="Semester" VALUE="Fall" CHECKED><BR>
Spring: <INPUT TYPE="radio" NAME="Semester" VALUE="Spring"><BR>
Summer: <INPUT TYPE="radio" NAME="Semester" VALUE="Summer"><BR>
<P>
Pay by:
<SELECT NAME="paymethod" SIZE=1>
<OPTION SELECTED>Pay by check
<OPTION>Pay by P.O.
<OPTION>Bill me
</SELECT>
<P>
Which courses?
<SELECT NAME="course" SIZE=5 MULTIPLE>
<OPTION>Beginning HTML
<OPTION>Intermediate HTML
<OPTION>Advanced HTML
<OPTION>CGI Programming
```

Enter your username:

Enter your password:

Select days you are available: Monday Tuesday Wednesday Thursday Friday

Select a semester:
Fall:
Spring:
Summer:

Pay by: Pay by check

Which courses? Beginning HTML
Intermediate HTML
Advanced HTML
CGI Programming
Web Technology

Give us any special instructions:

Type a message here.

Submit the Form Clear All Fields

```
<OPTION>Web Technology
<OPTION>Java for Beginners
<OPTION>Java for Programmers
<OPTION>Graphic Java
</SELECT>
<P>
Give us any special instructions:
<TEXTAREA ROWS=10 COLS=40 NAME="instructions">
Type a message here.
</TEXTAREA>
<P>
<CENTER>
<INPUT TYPE="submit" VALUE="Submit the Form">
<INPUT TYPE="reset" VALUE="Clear All Fields">
</CENTER>
</FORM>
```

And here is an example of a Perl script that parses the form input:

```
#!/usr/bin/perl
# The name of this file is "sampleform.cgi".

$request_method = $ENV{'REQUEST_METHOD'};

# Get the form data

if ($request_method eq "GET") {
    $form_info = $ENV{'QUERY_STRING'};
    }
else {
    $size_of_form_information = $ENV{'CONTENT_LENGTH'};
    read (STDIN, $form_info, $size_of_form_information);
    }

print <<END1;
Content-type: text/html

<HTML><HEAD>
<TITLE>Key-Value Pairs for Sample Fill-Out Form</TITLE>
</HEAD><BODY>
<H1>Key-Value Pairs for Sample Fill-Out Form</H1>
<PRE>
END1

@key_value_pairs = split(/&/, $form_info);

foreach $key_value (@key_value_pairs) {
    ($key, $value) = split (/=/, $key_value);
    $value =~ tr/+/ /;
    $value =~ s/%([\dA-Fa-f][\dA-Fa-f])/pack ("C", hex ($1))/eg;
    print $key, " = ", $value, "\n";
    }

print <<END2;
</PRE>
<HR>
<A HREF="http://internetinstituteusa.com">Internet Institute USA</A>
</BODY></HTML>
END2
```

Let's analyze the program:

As with the previous examples, the first line identifies the location of the Perl interpreter, which is typically located in /usr/local/bin/perl on a Unix system. The special sequence `#!` that begins the first line indicates that the location of the interpreter follows. As a reminder: these two characters must be the first two characters in the file: there can be no blank lines above this first line nor any other blank space before these characters appear.

The line:

```
$request_method = $ENV('REQUEST_METHOD');
```

gets the request method from the environment variable associative array `%ENV`. The `if/else` clauses that follow get the form input either from the URL or from the input stream, for the GET and POST methods, respectively.

After sending the `Content-type` and some initial HTML text to the Web browser with the `print/END1` construct, all of the key/value pairs are copied into array `@key_value_pairs` with the line:

```
@key_value_pairs = split(/&/, $form_info);
```

The form input that the CGI script sees is made up of ampersand (`&`) separated key-value pairs, that are separated by equals signs (`=`). For example, here is a URL sent using the GET method:

```
http://internetinstituteusa.com/cgi-bin/519_scripts/sampleform.cgi?
Username=Guest&Password=foo&Monday=on&Wednesday=on&Semester=Spring&
paymethod=Pay+by+P.O.&course=Intermediate+HTML&
course=Web+Technology&instructions=Here+is+a+message.%0D%0A
```

(This is a single line: it has been broken into 4 lines so that it will print better.) Notice that each key-value pair contains an equals sign, and that key-value pairs are separated by ampersands.

The `split` line above finds all of the key-value pairs, but we still need to separate the keys from the values. The line:

```
foreach $key_value (@key_value_pairs) {
```

steps through each key/value pair in array `@key_value_pairs` and copies the key-value pair into `$key_value`. We apply the `split` built-in function within the `foreach` loop to separate the keys from the values in the line:

```
($key, $value) = split(/=/, $key_value);
```

This copies the key into `$key` and the value into `$value`, using the equals sign (`=`) as the marker that identifies where the split takes place.

The next two lines:

```
$value =~ tr/+/ /;
$value =~ s/%([\dA-Fa-f][\dA-Fa-f])/pack ("C", hex ($1))/eg;
```

are somewhat cryptic on first glance, so let's go over them in detail. The first line uses the translate (`tr`) operator to change all plus signs (`+`) to spaces. Take a look at the URL above again. The portion near the end:

```
instructions=Here+is+a+message.
```

should actually be:

```
instructions=Here is a message.
```

The Web browser encodes spaces into printable characters so that they are not misinterpreted as command separators by the Web server. Our CGI script needs to undo the Web browser translation, and the `tr` operator takes care of translating spaces.

There are other special characters that also get translated. For example, a carriage return character and a newline character are typically paired at the end of a line. These are single-byte ASCII characters that appear at positions 13 and 10 in the ASCII character chart. In hexadecimal, these are positions `D` and `A`, and the Web browser encodes these characters using the 3-character sequences "`%0D`" and "`%0A`", respectively. We need to translate these 3-character sequences back into their single-character form on the server side.

The substitute (`s`) operator performs this task. The general form is:

```
variable =~ s/look_for_this/replace_with_this/options;
```

Refer again to the line:

```
$value =~ s/%([\dA-Fa-f][\dA-Fa-f])/pack ("C", hex ($1))/eg;
```

The string that needs to be matched starts with a percent sign (`%`), followed by any digit or character from A through F or a through f, followed by a second digit or character from A through F or a through f. (This is because hexadecimal digits go from 0 - 15 in base 10, which is represented as 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F in base 16, with lower case a - f being treated the same as upper case A - F.)

The (`$1`) syntax causes the matched pattern that is enclosed in parentheses (without the percent sign) to be placed in variable `$1`. The `pack` built-in function takes a first argument that identifies the format (`C` specifies an unsigned character will be produced), and a second argument that for this case is the hexadecimal representation of the matched string (via the `hex` built-in function), and produces the actual character. The `e` option evaluates the substitution string, so that the `pack` function with its arguments is actually evaluated, rather than being treated like the string “`pack`”, and the `g` option replaces all instances of the matched pattern in the string.

The remainder of the Perl script sends the trailing HTML text to the Web browser.

Madlibs Example

So far, our scripts simply print the data out in a web page, but we would like to do more. If we need to extract any one particular item from the user's form information, this can be difficult using the script above, but we can add a simple line to the code to pick out individual information:

```
$FORM{$key} = $value;
```

Adding this line to the `for/each` loop is one method to take information from the user's input and do something else with it.

For example, take a look at the HTML to generate a simple form:

```
<html>
<head>
<title>A Form To Try</title>
</head>

<body bgcolor="#FFFFCC">

<form action="http://internetinstituteusa.com/cgi-bin/519_scripts/madlibs.cgi"
method="POST">

<h3 align="center">This form will result in a custom madlibs page.</h3>

<pre>
       Your Name: <input type="text" name="username">
        Your Age: <input type="text" name="age">
  Your Shoe Size: <input type="text" name="shoesize">
  Favorite Smell: <input type="text" name="favesmell">
</pre>

<input type="submit" value="Send it On!"><br><br>
<input type="reset" value="Start Over!">

</form>

</body>
</html>
```

We have to keep track of what names we give to the input variables, but otherwise, this is pretty easy. We just call for the form item in our script like this: `$FORM{'name_of_form_input'}` and the script drops in the value that the user typed into that input.

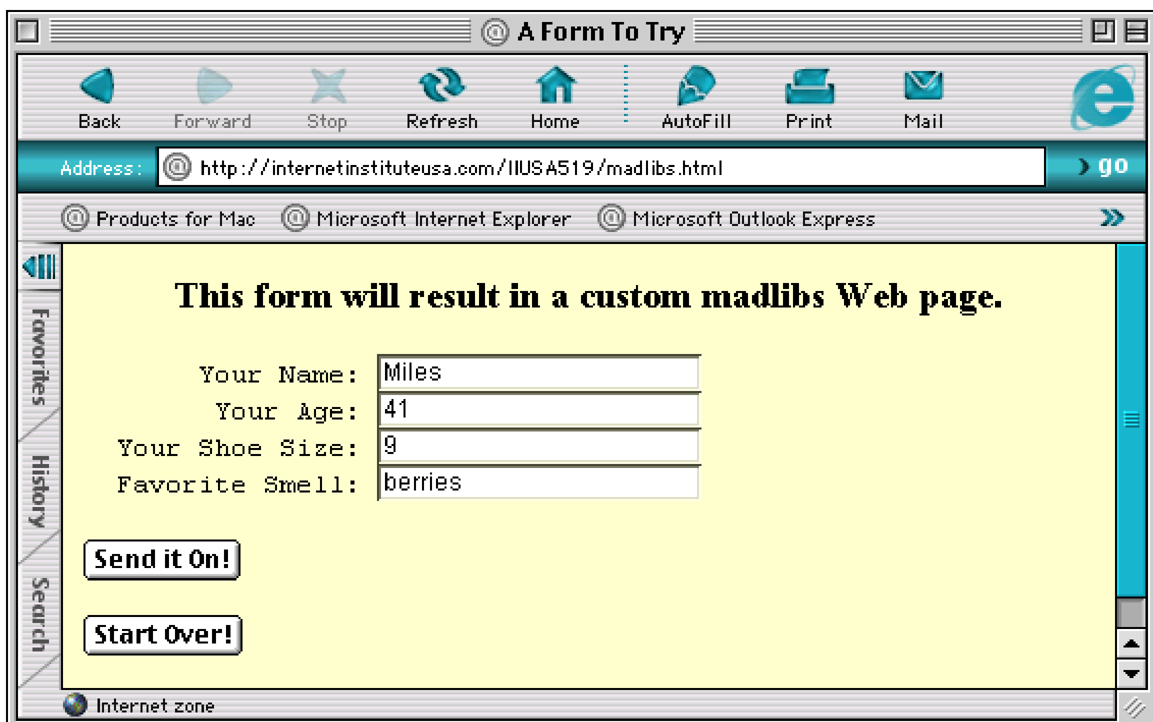
```
#!/usr/bin/perl
# The name of this file is "madlibs.cgi".

$request_method = $ENV{'REQUEST_METHOD'};

# Get the form data

if ($request_method eq "GET") {
    $form_info = $ENV{'QUERY_STRING'};
    }
else {
    $size_of_form_information = $ENV{'CONTENT_LENGTH'};
    read (STDIN, $form_info, $size_of_form_information);
    }

print <<END1;
Content-type: text/html

<HTML><HEAD>
<TITLE>Silly Form Results</TITLE>
</HEAD><BODY>
<PRE>
END1

@key_value_pairs = split(/&/, $form_info);
```

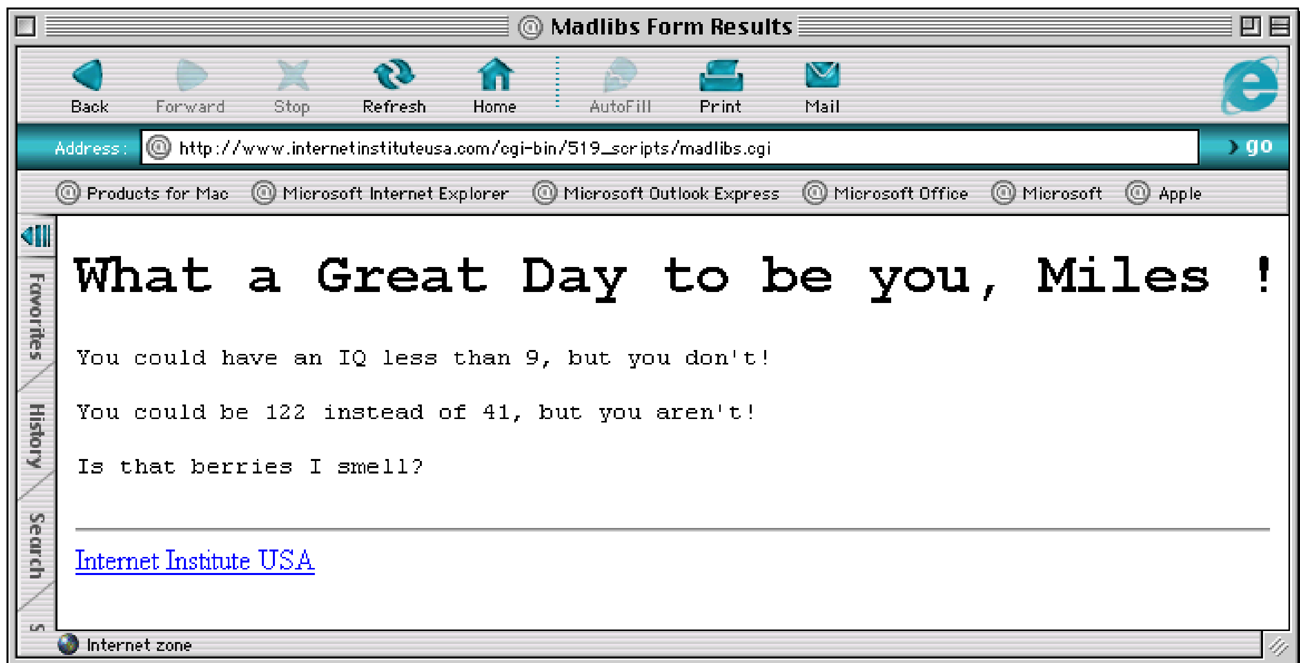
```
foreach $key_value (@key_value_pairs) {
    ($key, $value) = split (/=/, $key_value);
    $value =~ tr/+/ /;
    $value =~ s/%([\dA-Fa-f][\dA-Fa-f])/pack ("C", hex ($1))/eg;
    $FORM{$key} = $value;
}

print "<h1>What a Great Day to be you, $FORM{'username'} !</h1>";
print "<p>You could have an IQ less than $FORM{'shoesize'}, but you don't!</p>";
print "<p>You could be 122 instead of $FORM{'age'}, but you aren't!</p>";
print "<p>Is that $FORM{'favesmell'} I smell?</p>";

print <<END2;
</PRE>
<HR>
<A HREF="http://internetinstituteusa.com">Internet Institute USA</A>
</BODY></HTML>
END2
```

Server Redirection

When a Web site is moved to a different host computer, the old site should inform the end-user of the change. However, we may only want to make the change temporary: we might use one location during the day and another location at night. How can we do this in a way that the end-user is unaware of the change? The "`Location:`" server directive can be sent by a CGI script to a Web server, to cause the server to generate a Web page that tells the browser to look someplace else for the requested file. For example, if a user requests file "`foo.html`", then a server directive of

```
Location: "http://remus.rutgers.edu/"
```





will tell the Web server to generate a message that can be displayed on a Web browser informing the end-user that the file has moved.

This is the old style of browser behavior. Newer Web browsers do not display the message: they simply make a second request for the file, using the new address, and the end-user will generally not even be aware that a server redirection has taken place.

Here is an example, in which we will attempt to access a file on internetinstituteusa.com, but we will be sent to www.yahoo.com instead. Click on the following hyperlink:

```
http://internetinstituteusa.com/cgi-bin/519_scripts/redirect.cgi
```

Notice that the URL above shows the host machine as `internetinstituteusa.com` and that the URL displayed in the Web browser shows a host machine of `www.yahoo.com`.

Here is what the script looks like:

```
#!/usr/bin/perl
print "Location: http://www.yahoo.com", "\n\n";
```

Reading from a File

CGI/Perl Becomes more powerful when we begin to make use of other files on the server. To start with we will look at reading from an existing file. Suppose that we have a file named "`directory.dat`" on the server and that the file looks like this:

```
Jim Thomas|Communication|SCILS 415|husatonic@scils.rutgers.edu|732-555-8826
Andrew Mayer|Library and Inf. Studies|DeWitt 415|jahlov@scils.rutgers.edu|732-555-7501
Jennifer Chromos|Journalism and Mass Media|DeWitt 417|ante@scils.rutgers.edu|732-555-7369
Belle J. Starr|Library and Inf. Studies|SCILS 418|belkin@scils.rutgers.edu|732-555-8585
Sumner Sampson|Communication|SCILS 419|bigboy@scils.rutgers.edu|732-555-7910
```

The file is a little list of names, departments, office locations, email addresses, and telephone numbers with each item separated by a | character. It does not matter what kind of a character we use as a separator, as long as our code interprets it appropriately.

We can read in this information with a few lines of code:

```
$filename = ”directory.dat“;
open(INF,$filename);
@indata = <INF>;
close(INF);
```

This code reads a file with a *handle* `INF`, reading the information into an array named `@indata` and then closing the handle (thus closing the file).

Once we have the information from the file in an array, we can use it just like any other array in Perl. In this case, we are taking one line at a time, splitting the line up along the (|) character and dropping the resulting chunks of information into different variable names (`$facultyname, $department, $roomnumber`, *etc.*)

After we do this, the information is formatted into a table for the user. The Perl code is shown below:

```
#!/usr/bin/perl

$filename = "directory.dat";

print <<HEADER;
Content-type:text/html

<html><head><title>Staff Directory</title></head>

<body bgcolor="#FFFFCC">

HEADER

open(INF,$filename);
@indata = <INF>;
close(INF);

print "<table border=1>";
print "<tr><th>Faculty Name</th><th>Department</th><th>Room Number</th>
<th>Email</th><th>Phone Number</th></tr>\n";

foreach $i (@indata) {
        chop($i);
        ($facultyname,$department,$roomnumber,$email,$phonenumber) = split(/\|/
,$i);

        print "<tr>";
        print "<td>$facultyname</td>";
        print "<td>$department</td>";
        print "<td>$roomnumber</td>";
```

```
        print "<td><a href=\"mailto:$email\">$email</a></td>";
        print "<td>$phonenumber</td>";
        print "</tr>\n";
}
print "</table>";
print "</body></html>";
```

Give it a try! Visit the read.cgi hyperlink at: http://internetinstituteusa.com/cgi-bin/519_scripts

Writing to a File

It would be useful if we can also modify the `directory.dat` data file from the Web. We have seen a form processing script for reading from a file above, but now we will create a form that instead of just sending back a Web page with the appropriate information, we will write to a file using the following lines of code:

```
open(OUTF,">>directory.dat");

print OUTF "$FORM{'facultyname'}|$FORM{'department'}|$FORM{'roomnumber'}
|$FORM{'email'}|$FORM{'phonenumber'}\n";
close(OUTF);
```

What we are doing is opening up the file with a handle `OUTF`, and printing the information to `directory.dat` using the `FORM{'name_of_form_input'}` construct and a `print` statement.

After the information is written into the file we close it using the handle `OUTF`.

The remaining code in the program redirects the script to another web resource, in this case to the CGI script `mydirectory.cgi` which parses the `directory.dat` file into an HTML page for the user to see that the information was added.

```
#!/usr/bin/perl

#Get the form data

$request_method = $ENV{'REQUEST_METHOD'};

if ($request_method eq "GET") {
    $form_info = $ENV{'QUERY_STRING'};
    }
else {
    $size_of_form_information = $ENV{'CONTENT_LENGTH'};
    read (STDIN, $form_info, $size_of_form_information);
    }

@pairs = split(/&/, $form_info);

foreach $pair (@pairs) {
        ($name, $value) = split(/=/, $pair);
        $value =~ tr/+/ /;
```

```
        $value =~ s/%([\dA-Fa-f][\dA-Fa-f])/pack ("C", hex ($1))/eg;
        $FORM{$name} = $value;
}

#write everything into the directory file

open(OUTF,">>directory.dat");

print OUTF "$FORM{'facultyname'}|$FORM{'department'}|$FORM{'roomnumber'}
|$FORM{'email'}|$FORM{'phonenumber'}\n";
close(OUTF);
print "Location:http://internetinstituteusa.com/cgi-bin/519_scripts/
mydirectory.cgi\n\n";
```

This is the HTML code for the form that writes to `directory.dat`:

```
<html>
<head>
<title>Administer your Database</title>
</head>

<body bgcolor="#FFFFCC">

<form action="admindirectory.cgi" method="post">

<h2>This form will add people to our directory</h2>

<pre>

Faculty Name: <input type="text" name="facultyname">
  Department: <select name="department">
<option>Communication</option>
<option>Library and Information Studies</option>
<option>Journalism and Mass Media</option>
</select>
        Room: <input type="text" name="roomnumber">
       Email: <input type="text" name="email">
Phone Number: <input type="text" name="phonenumber">
</pre>

<input type="submit" value="add to database">

</form>

</body>
</html>
```

Try it for yourself! Visit the hyperlink at:

`http://internetinstituteusa.com/519_scripts/administer.html`

Subroutines

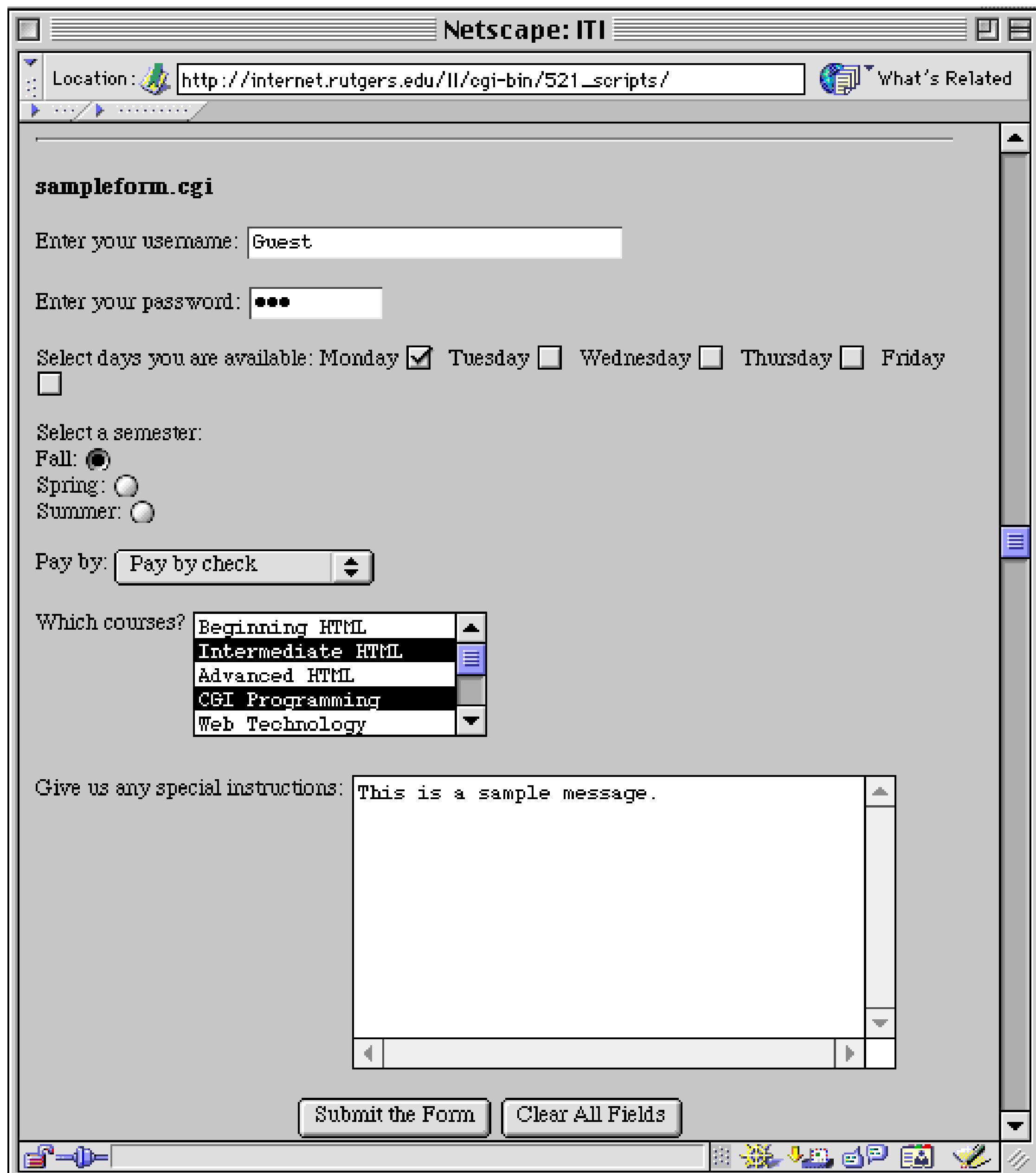
As the complexity of the program grows, following good coding practice, we should break up the

program into smaller subroutines that encapsulate logically distinct portions of the code. This helps make the program more readable, and also helps to reuse portions of the code instead of repeating it in-line.

First, we take a look at the Username/Password pair, and accept only the pair: `Guest/baz`. Try anything other than that pair shown below at http://internet.rutgers.edu/II/cgi-bin/519_scripts, and click on the submit button to see how it fails, and then try it with the correct Username/Password pair of `Guest/baz` to see how it succeeds:

Here is the Perl script that processes the form. Not only does it parse the form data, but it checks for a valid Username/Password pair, and sends a copy of the form data to `iti00xx@internet.rutgers.edu` via email:

```
#!/usr/bin/perl
# The name of this file is "sampleform2.cgi".

$request_method = $ENV{'REQUEST_METHOD'};

# Get the form data

if ($request_method eq "GET") {
    $form_info = $ENV{'QUERY_STRING'};
    }
else {
    $size_of_form_information = $ENV{'CONTENT_LENGTH'};
    read (STDIN, $form_info, $size_of_form_information);
    }

@key_value_pairs = split(/&/, $form_info);

$has_password = 0;
$has_username = 0;

foreach $key_value (@key_value_pairs) {
    ($key, $value) = split (/=/, $key_value);
    $value =~ tr/+/ /;
    $value =~ s/%([\dA-Fa-f][\dA-Fa-f])/pack ("C", hex ($1))/eg;

    # The Username/Password pair is hardwired to: "Guest"/"baz"
    if ($key eq 'Username') {
        $has_username = 1;
        if (!($value eq 'Guest')) {
            &return_error(500, "Bad username", $value);
            }
        }
    elsif ($key eq 'Password') {
        $has_password = 1;
        if (!($value eq 'baz')) {
            &return_error(500, "Bad password", $value);
            }
        }
    }

    # Did someone submit a bad form?
```




```
    if (($has_username == 0) || ($has_password == 0)) {
        &return_error(500, "Bad form", "Missing username or password");
        }

   &print_all_OK;

exit(0);

# Here is the error subroutine

sub return_error {
    local ($status, $keyword, $message) = @_;

    print "Content-type: text/html", "\n";
```

```
    print "Status: ", $status, " ", $keyword, "\n\n";

    print <<End_Of_Error;

<HTML>
<HEAD>
    <TITLE>CGI Program - Unexpected Error</TITLE
</HEAD>
<BODY
<H1>$keyword</H1>
<HR>$message<HR>
</BODY>
</HTML>

End_Of_Error

    exit(1);
}

# If the username and password check OK, then we invoke this routine

sub print_all_OK {
print <<END1;
Content-type: text/html

<HTML><HEAD>
<TITLE>Processing a Fill-Out Form</TITLE>
</HEAD><BODY>
<H1>Processing a Fill-Out Form</H1>
This confirms that your form was processed, with the following
parameters:
<PRE>
END1

foreach $key_value (@key_value_pairs) {
    ($key, $value) = split (/=/, $key_value);
    $value =~ tr/+/ /;
    $value =~ s/%([\dA-Fa-f][\dA-Fa-f])/pack ("C", hex ($1))/eg;
    if (!($key eq 'Username') && !($key eq 'Password')) {
        print $key, " = ", $value, "\n";
        }
    }

print <<END2;
</PRE>
<HR>
<A HREF="http://internetinstituteusa.com">Internet Institute USA</A>
</BODY></HTML>
END2
# Replace your_address@company.com with your email address, and you will
# receive an email message for each form submitted.  Be mindful of
# the backslash that appears before the commercial at sign (@), which
# is needed so that it does not look like an array, which
# also starts with (@).  You will need to replace your_address@company.com
# in two places, and also change the "From:" line to correspond to
# your account on internet.rutgers.edu.
```

```
open(SENDMAIL, "| /usr/lib/sendmail -fyour_address\@company.com -t -n -oi");
print SENDMAIL <<End_Of_Mail;
From: iti00xx <iiusa00xx\@internetinstituteusa.com>
To: your_address\@company.com
Subject: Confirmation of processed form
This confirms that your form was processed, with the following
parameters:

End_Of_Mail
    foreach $key_value (@key_value_pairs) {
        ($key, $value) = split (/=/, $key_value);
        $value =~ tr/+/ /;
        $value =~ s/%([\dA-Fa-f][\dA-Fa-f])/pack ("C", hex ($1))/eg;
        if (!($key eq 'Username') && !($key eq 'Password')) {
            print SENDMAIL $key, " = ", $value, "\n";
            }
        }
    }
```

Let's analyze the program:

The lines:

```
$has_password = 0;
$has_username = 0;
```

create two variables that will be set to `1` if the script finds a Password and Username key/value pair in the form data. We make use of these variables later in the code.

Within the `foreach` loop, every key/value pair is checked for a Username key and a Password key. If either is found and does not match the value being checked, then the subroutine `return_error` is invoked (which we will study later). Notice that the name of the subroutine is `return_error`, but that it is invoked with the ampersand as in: `&return_error`.

If the `foreach` loop completes with no errors, then that means the user did not send an invalid Username or an invalid Password to the server, which is the expected situation. However, a malicious user could simply create a new fillout form that has no Username/Password entry and send that to the server, and when that happens, the `foreach` loop will complete with no error conditions and give the end-user access without seeing a valid Username and Password!

To prevent this situation, the `$has_username` and `$has_password` variables are set to 1 in the `foreach` loop when at least one of each has been seen. Then, when the `foreach` loop finishes, the code:

```
    if (($has_username == 0) || ($has_password == 0)) {
        &return_error(500, "Bad form", "Missing username or password");
        }
```

checks to make sure that a Username/Password pair has been seen, and calls the error processing

subroutine `return_error` if there is a problem.

If all is well with the Username and Password, then subroutine `print_all_ok` is invoked, and the script then exits.

Before we move on to the subroutine code, let's review how the subroutines are invoked. The `return_error` subroutine is invoked in three places, using different arguments each time:

```
&return_error(500, "Bad username", $value);
&return_error(500, "Bad password", $value);
&return_error(500, "Bad form", "Missing username or password");
```

The `print_all_ok` subroutine is invoked only once, and is passed no arguments:

```
&print_all_OK;
```

Now let's take a look at the subroutine code.

A subroutine begins with the `sub` keyword, followed by the subroutine name, followed by the statements that make up the subroutine, which are enclosed in matching curly braces, as in:

```
sub return_error {
    # Perl statements go here

    }
```

The `return_error` subroutine captures its arguments into "local" variables with the line:

```
local ($status, $keyword, $message) = @_;
```

By "local" variable, what is meant is that the variable can only be accessed within the subroutine. Its *scope* does not extend beyond the boundaries of the subroutine, and no other subroutines nor any future instances of this same subroutine can access these variables. Future instances will use the same variable names, but the contents of the variables will change for each new invocation. The predefined variable `@_` is an array that contains the arguments passed to the subroutine.

After obtaining the parameters passed to it, the `return_error` subroutine prints an error message and then exits.

The `print_all_OK` subroutine sends HTML code back to the end-user, confirming that the form was received and that all is OK. The subroutine also sends the form data to `iti00xx@internet.rutgers.edu` via email. This is a fictitious email address, and so the code as shown will not work as intended. However, if the email address is changed to any valid email address, including an email address that the user might provide in a fill-out form, then the form data will be emailed to that address.

Part 3: Interaction with a Line Oriented Database

In this part of the tutorial, we cover CGI interaction with a Structured Query Language (SQL) queriable database.

The Database Format

A surprisingly common database format is “flat” and “text-oriented.” What is meant by "flat" is that records are stored sequentially in a file, with no hierarchy or structure among the records (although the records themselves have internal structure). What is meant by “text-oriented” is that the information stored in each record has no special encoding. The data is entirely human-readable, although it may not be formatted particularly well for readability.

The reason that this is surprising is that there are much better formats and methods of organizing information that have been in use for years. In practice, many databases contain only hundreds or thousands of records, in which a more efficient structure would not be noticed with the speed of today's computers (and the relative sluggishness of networks). The text-oriented organization makes the database easier to manage and convert to other forms, and so this seemingly unsophisticated approach is in fact, effective and commonly in use.

A Microsoft Excel spreadsheet can be saved in a number of formats. One format separates records by “newline” characters, and separates records with commas. This is referred to as “comma delimited” (`.csv`) format. This is also the same form used by the Sprite database that we will use here, which is available as a Perl module. (By the way: the comma delimited format is the same format used by the course database for Internet Institute USA at `http://iiusa.cc`).

A problem with this format is that as the records extend past about 80 characters, it becomes difficult to use an ordinary text editor to modify the records (which is not a good idea anyway, since manually editing records invites disasters that would not happen if specialized database access routines that ensure record formats are preserved are used exclusively). The problem is that a newline only occurs at the end of a record. To make our work simpler when setting up a line-oriented database (as this format is called), we will use a visually simpler format with a new line for every record, and then write a Perl program that translates the simpler multi-line format into the single-line format.

An Image Database

We will create a database that stores URLs for images, which can be located anywhere on the World Wide Web, along with a description for each image and a list of keywords that will be used when searching the image database. For an example of how this works, see:

```
http://internetinstituteusa.com/cgi-bin/519_scripts/Pictures.cgi
```

The simplified format is shown below:

```
ImageLocation
Description
Keywords
-=-
http://waldo.wi.mit.edu/WWW/examples/Ch9/pictures/camel.gif
This is a picture of camels in the foreground, loaded with supplies.+
There are snow covered mountains in the background.  This comes from+
Lincoln Stein\'s collection.
camel mountain desert
-=-
http://waldo.wi.mit.edu/WWW/examples/Ch9/pictures/alps.gif
A skier and a dog are in the foreground, looking over the peaks of the Alps.+
From Lincoln Stein\'s collection.
mountain Alp dog ski
-=-
http://waldo.wi.mit.edu/WWW/examples/Ch9/pictures/cactus.gif
A cactus flower is blooming.  From Lincoln Stein\'s collection.
cactus flower
-=-
http://www.internet.rutgers.edu/ITI/Press/Focus/248a-1.jpg
The Hill 248 laboratory.
workstation computer desk
-=-
http://www.cs.rutgers.edu/~murdocca/Executive.gif
An image that Miles scanned in from an issue of Scientific American.+
Forgot who the artist is.
executive robot automaton
```

Each record is separated by the symbol "`-=-`" on a line all by itself. The first record contains the names of the fields, one field per line. Keep in mind that this not the line-oriented database, but rather, a preliminary version that has one field per line (which makes it easier to read while we create the database.) There are three fields in each record in the database, named "`ImageLocation`", "`Description`", and "`Keywords`". Each field is on its own line, but even this does not simplify readability enough: some of the descriptions extend for more than one line. The solution is to append a plus sign "+" to each line except the last line that makes up a field. Note that Perl special characters like single quotes and double quotes need to be escaped by prepending them with a backslash.

The Perl script `ConvertToSprite.pl` which converts this file into the line-oriented format is shown below:

```
#!/usr/bin/perl
# Convert a text database file that uses the newline "\n" to separate fields
# and "-=-" to separate records into a file that uses "," to separate fields
# and "\n" to separate records.
#
# This is used in preparing a file for the Sprite format.  Note that the
# source file needs to escape single or double quotes by prepending with
# a backslash.  So, "John's car" becomes "John\'s car".
#
# The first line in the file lists the fields.  The records follow.
#
# The name of this file is "ConvertToSprite.pl".
```

```
#

open (INFILE, 'PictureSource.db') || die "Cannot open PictureSource.db\n";
open (OUTFILE, '>PictureFinal.db') || die "Cannot create PictureFinal.db\n";

$/ = "\n-=-\n";  # The delimiter between records, which contain newlines.

while (<INFILE>) {
    chomp;          # Remove -=-
    s/,/\\,/g;      # Convert any ',' to '\,'
    s/\n/,/g;       # Convert newline to ','
    s/,$//;         # Remove final ','

    # A plus sign (+) at the end of the line is a continuation character.
    # The plus sign is converted to a space character, and a newline is
    # not produced at the end because the field has not yet ended.
    # This allows a single field to extend over several lines.

    s/\+,/ /g;     # Convert '+' to ' '
    print OUTFILE "$_\n";  # Print converted line to OUTFILE
    }

close (INFILE);
close (OUTFILE);
```

Let's go over the `ConvertToSprite.pl` program line by line (by the way, Perl programs end with the suffix "`.pl`" by convention. However, CGI scripts, even when they are written in Perl, end in "`.cgi`" also by convention, although this is a configurable option for the Web server that can be changed.) Starting from the first line and working our way down, the first line encountered is:

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

The two-character sequence "`#!`" is a flag to the operating system, that the location of the interpreter follows. A CGI script can be written in other languages than Perl, and so this line tells the operating system where to find the interpreter that will read the rest of the file.

Continuing down the code, blank lines, and lines that begin with a pound sign (`#`) are ignored by the Perl interpreter.

This program reads a file named `PictureSource.db`, converts it into the line-oriented format, and stores the result in `PictureFinal.db`. (This program is intended to be executed from the Unix command line rather than via CGI, since it is executed just once at the time that the database is created. From that point onward, all changes are made via CGI.) The input file is opened with the line:

```
open (INFILE, 'PictureSource.db') || die "Cannot open PictureSource.db\n";
```

If the file cannot be opened, then the Perl `die` routine is called, which aborts the program at that point.

The output file is created with the line:

```
open (OUTFILE, '>PictureFinal.db') || die "Cannot create PictureFinal.db\n";
```

The single right angle bracket (>) indicates that the file will be created anew, wiping out any previous contents if the output file already exists. A double right angle bracket (>>) would indicate that the file should be appended, rather than overwritten. Without a single or double angle bracket, the file is opened for reading, as for `INFILE` above.

`INFILE` now serves as the "handle" for the input file `PictureSource.db`, and `OUTFILE` serves as the handle for the output file `PictureFinal.db`.

The two-character sequence "`$/`" is a Perl special variable that holds the input record separator. By default, the input record separator is a newline. For our situation, each record is made up of multiple lines, and the character string "`-=-`" is the record separator. There is a newline before and after this separator, and so just to be cautious that "`-=-`" does not naturally occur somewhere in the input, we define the input separator as:

```
$/ = "\n-=-\n";  # The delimiter between records, which contain newlines.
```

Note that there cannot be any other white space character immediatley before or after "`-=-`", otherwise this will not work.

The `while` loop reads one more record from the input file on each iteration, and converts it into a single-line record. The Perl special variable "`$_`" references the default input and pattern matching space. Thus, all of the activity within the while loop operates on `$_` by default, which is why `$_` appears as the source for the `print` line.

```
while (<INFILE>) {
    chomp;          # Remove -=-
    s/,/\\,/g;      # Convert any ',' to '\,'
    s/\n/,/g;       # Convert newline to ','
    s/,$//;         # Remove final ','

    # A plus sign (+) at the end of the line is a continuation character.
    # The plus sign is converted to a space character, and a newline is
    # not produced at the end because the field has not yet ended.
    # This allows a single field to extend over several lines.

    s/\+,/ /g;     # Convert '+' to ' '
    print OUTFILE "$_\n";  # Print converted line to OUTFILE
    }
```

Finally, the input and output files are closed with:

```
close (INFILE);
close (OUTFILE);
```

On a Unix system, any open files are automatically closed when a process ends, however, explicitly closing a file is the clean way to do it.

Suppose that the source file is named "`PictureSource.db`", that the converted file is named "`PictureFinal.db`", and that the Perl program is in a file named "`ConvertToSprite.pl`". The Perl program must be executable, and this can be done on a Unix system (named "internet", thus the command line shown below) with:

```
internet> chmod a+rx ConvertToSprite.pl
```

We can convert the source database format into the target comma delimited format with:

```
internet> ConvertToSprite.pl > PictureFinal.db
```

The converted file looks like this (it looks like there are multiple lines below, but in fact, there are only 6 records, one for the first record that gives the field names and 5 for the images):

```
ImageLocation,Description,Keywords
http://waldo.wi.mit.edu/WWW/examples/Ch9/pictures/camel.gif,This is a picture of
 camels in the foreground\, loaded with supplies. There are snow covered mountai
ns in the background.  This comes from Lincoln Stein\'s collection.,camel mounta
in desert
http://waldo.wi.mit.edu/WWW/examples/Ch9/pictures/alps.gif,A skier and a dog are
 in the foreground\, looking over the peaks of the Alps. From Lincoln Stein\'s c
ollection.,mountain Alp dog ski
http://waldo.wi.mit.edu/WWW/examples/Ch9/pictures/cactus.gif,A cactus flower is
blooming.  From Lincoln Stein\'s collection.,cactus flower
http://www.internet.rutgers.edu/ITI/Press/Focus/248a-1.jpg,The Hill 248 laborato
ry.,workstation computer desk
http://www.cs.rutgers.edu/~murdocca/Executive.gif,An image that Miles scanned in
 from an issue of Scientific American. Forgot who the artist is.,executive robot
 automaton
```

The converted database is then accessed through CGI by a Perl program that we will study below.

The Web server is typically configured to run with minimal permissions, and will thus be unable to read or modify the database (which we own) unless we change the permissions to allow access by others. Unfortunately, this makes the database vulnerable to perusal and potential damage by anyone else who has an account on the same system, but secure solutions are not easy to implement with CGI, and we will simply live with this security problem. Here is how to change the permissions on a Unix system so that the converted database (not the original, unconverted database!) is readable and modifiable by both the owner and the Web server (and as a result, by anyone else with an account on the same system):

```
% chmod a+rw PictureFinal.db
```

The converted file is the same as would be produced by saving a Microsoft Excel file in comma delimited (`.csv`) format.

The Database Code

In this section, we look at a program that accesses the line-oriented database. This program demon-

strates the "Web site in a file" concept, in which all of the pages in the Web site are created by a single CGI script. There are no other Web pages that make up this Web site, although the user perceives that there are four Web pages in this site (plus a few error message Web pages).

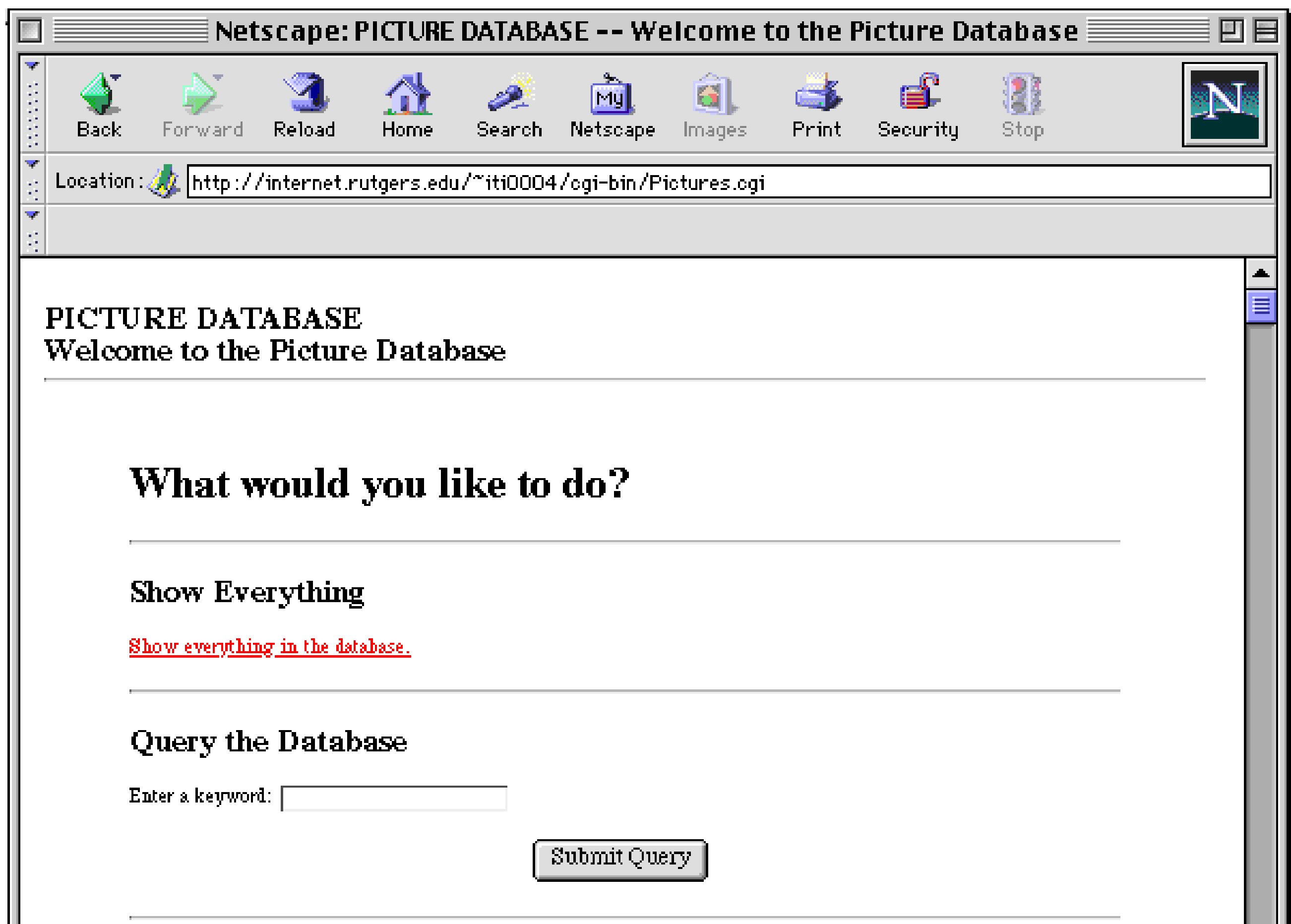
We will assume that the program is stored in a file named "`Pictures.cgi`" in the cgi-bin directory, so it would be accessible via the World Wide Web as `http://internetinstituteusa.com/~itiwxyz/cgi-bin/Pictures.cgi` in which `itiwxyz` is replaced with the owner's account (this assumes the programmer has an account on machine `internet.rutgers.edu`. For other computers, the hostname will need to change and also possibly, the path to `Pictures.cgi`.) A working copy of this program can be found at:

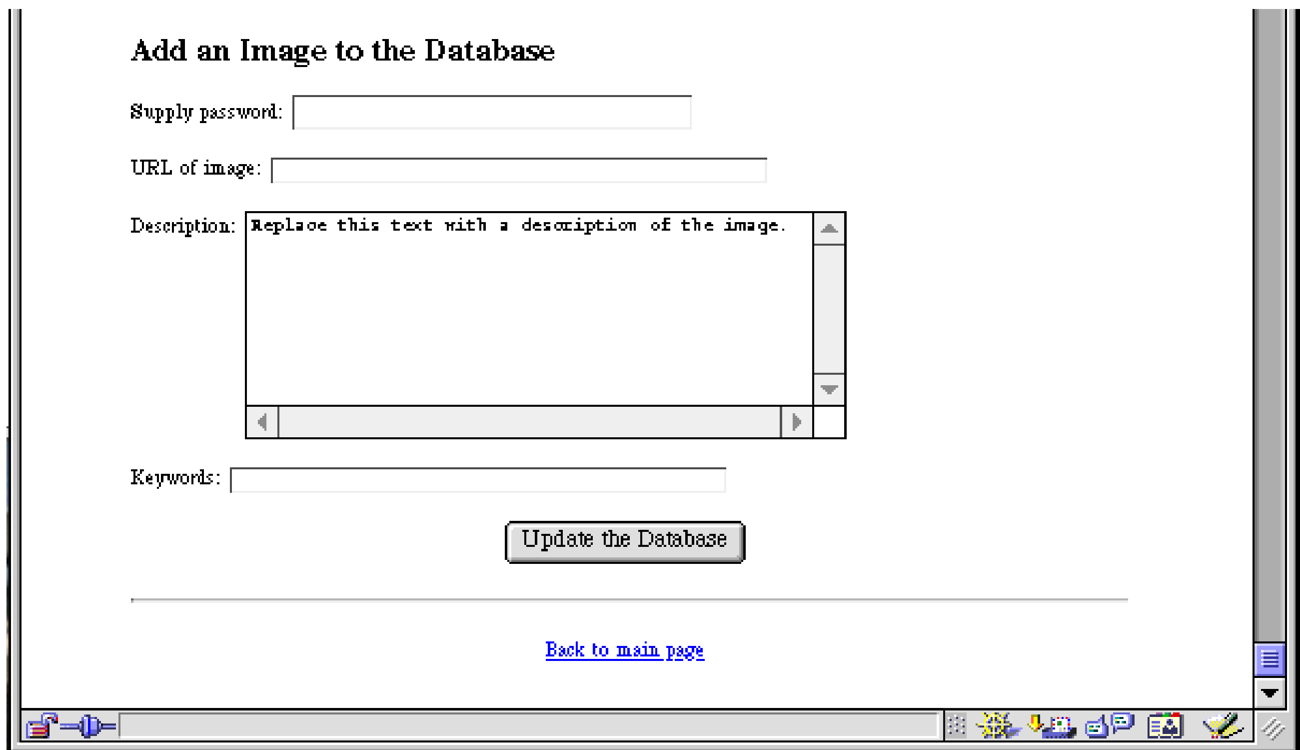
`http://internetinstituteusa.com/cgi-bin/519_scripts/Pictures.cgi`

The initial screen produced by the URL above is shown (in parts) below and on the next page.

The code that follows is based on the Sprite Perl library developed by Shishir Gundavaram. The Sprite library allows a line-oriented database that can be queried with SQL (Structured Query Language) commands, which is a common approach to accessing databases of all kinds, not just line-oriented databases.

```
#!/usr/bin/perl

# The Sprite library comes from the CPAN archive
```






```
# at ftp://ftp.cis.ufl.edu/pub/perl/CPAN in
# modules/by-authors/Shishir_Gundavaram.  Examples
# of how to use Sprite are in "CGI Programming on
# the World Wide Web" by Shishir Gundavaram, O'Reilly
# & Associates, (1996).

# This comes from the Sprite author, in a README file in the distribution:
# Sprite, v3.21
# -------------
#
# What is it? No, it's not the soda that claims, "Image is Nothing, Thirst
# is Everything" :-) However, it is a useful module that allows you to access
# text-delimited databases, such as those exported from applications like
# MS Excel, using a _small_ subset of SQL.
#
# Sprite has some cool advantages, like allowing you to embed Perl's powerful
# regular expressions -- not those wimpy ones that come with your commercial
# relational database engine -- into your queries.
#
# [...]
#
# Shishir Gundavaram
# April 21, 1998

#
# The following code is written by Miles Murdocca, February 1999
#

#
# The next line tells the Perl interpreter to look in the Sprite-3.21
# directory to find libraries, in addition to the default library path.
# You will need to change this path to correspond to the place where
```

```
# you install Sprite on your system.  On internet.rutgers.edu at Rutgers
# University, Sprite is already installed at this location so there is
# nothing more for you to do if you are using that system.
#
use lib '/www/cgi-bin/519_scripts/Sprite-3.21';

# The next line selects a specific library we will use (there are
# two Sprite libraries in the library path above.  We will use Sprite.pm,
# which is indicated by simply specifying "Sprite" as below.)

use Sprite;

# The PATH_INFO environment variable extracts information
# from the URL.  So, if the name of this file is
# "Pictures.cgi", then a URL such as:
# http://internet.rutgers.edu/~iti00xy/cgi-bin/Pictures.cgi/Update
# will put "/Update" into the PATH_INFO environment variable.
#
# Here are the supported PATH_INFO fields:
#
# / -- (No additional path information) Show the initial page.
# /ShowAll -- Show everything in the database.
# /Query -- Execute a query
# /Update -- Show the update page
# /(Anything else) -- Show the initial page
#

# Change the next line to correspond to your cgi-bin
$DOCUMENT_ROOT = "http://internetinstituteusa.com/cgi-bin/519_scripts";

# Change the next line to correspond to your login id
$webmaster = "iiusa2xyz\@internetinstituteusa.com";

# Change the next line if you change the name of the database.
# *** VERY IMPORTANT *** The way we are doing it, *you* are the owner of
# this file, and so, the Web server cannot modify it unless you change
# the permissions to be (somewhat dangerously) writable by all.  From the
# Unix command line, use:
#
# internet> chmod a+rw PictureFinal.db

# A slightly better solution has the Web server create the file, but
# that is not a lot better.  A better solution is to run the server with
# your permissions and also check for a password.
#
$DATABASE_LOCATION = "PictureFinal.db";

# The password is used in process_update, for adding entries to the database
#
$PASSWORD = "foo";

$path_info = $ENV{'PATH_INFO'};

# Delete leading "/" from $path_info, so "/Update" becomes "Update".  Notice
# that "/" is a special character used in Perl patten matching, and so we need
# to escape it with a backslash.
```

```
($discard, $path_info) = split(/\//, $path_info);

# Associative array (a.k.a. hash) "FORM" below will hold the key/value pairs

# For PATH_INFO = "/Query"
if ($path_info eq "Query") {
    &parse_form_data(*FORM);
    &print_query();
    }
# For PATH_INFO = "/Update"
elsif ($path_info eq "Update") {
    &parse_form_data(*FORM);
    &process_update();
    }
# For PATH_INFO = "/ShowAll"
elsif ($path_info eq "ShowAll") {
    &parse_form_data(*FORM);
    &print_showall();
    }
# For PATH_INFO = all other cases.
else {
    &print_main_form();
    }

exit(0);


##### THIS IS THE END OF THE MAIN PROGRAM.  SUBROUTINES START HERE.  ######
```

This ends the main program. The subroutines follow.

From an organizational standpoint, the main routine uses the `PATH_INFO` environment variable to determine which Web page is requested by the user. A different subroutine is called depending on whether `PATH_INFO` is `/Query`, `/Update`, `/ShowAll`, or anything else.

Each routine that prints a Web page creates it in three parts: the beginning HTML code, the body, and the trailing HTML code. The header for each generated Web page is produced with the same routine:

```
    &print_HTML_Header("Welcome to the Picture Database");
```

in which the string passed to the routine is used to create a title. The trailer is produced the same for each Web page, with:

```
    &print_HTML_Trailer;
```

The code that produces the previous screen image is shown below:

```
#
# ----- SUBROUTINE print_main_form
#
# Create an HTML page for the main page.
#
```

```
sub print_main_form {
    &print_HTML_Header("Welcome to the Picture Database");

     print <<HTML_Middle;
<H1>What would you like to do?</H1>
<P>
<HR>
<H2>Show Everything</H2>
<A HREF="$DOCUMENT_ROOT/Pictures.cgi/ShowAll">
Show everything in the database.</A>
<P>
<HR>
<P>
<H2>Query the Database</H2>
<FORM ACTION="Pictures.cgi/Query" METHOD="POST">
<P>Enter a keyword:
<INPUT TYPE="text" NAME="keyword" SIZE=20>
<P>
<CENTER>
<INPUT TYPE="submit" VALUE="Submit Query">
</CENTER>
</FORM>
<P>
<HR>
<P>
<H2>Add an Image to the Database</H2>
<P>
<FORM ACTION="Pictures.cgi/Update" METHOD="POST">
Supply password:
<INPUT TYPE="password" NAME="Password" SIZE=30>
<P>URL of image:
<INPUT TYPE="text" NAME="URL" SIZE=45>
<P>Description:
<TEXTAREA ROWS=10 COLS=50 NAME="Description">
Replace this text with a description of the image.
</TEXTAREA>
<P>Keywords:
<INPUT TYPE="text" NAME="Keywords" SIZE=45>
<P>
<CENTER>
<INPUT TYPE="submit" VALUE="Update the Database">
</CENTER>
</FORM>
<P>
HTML_Middle

    &print_HTML_Trailer;
    }
```

When the `/ShowAll` “Web page” is selected, the image shown below is produced. The code that generates the `/ShowAll` Web page follows.

```
#
# ----- SUBROUTINE print_showall
```

```
#
# Create an HTML page that lists full picture info.
#

sub print_showall {
    # Create a new Sprite database "object"
    $rdb = new Sprite();

    # Read in all records in the database, which are separated by commas.
    $rdb->set_delimiter("Read", ",");
    @data = $rdb->sql("select * from $DATABASE_LOCATION");
    $rdb->close();

    $status = shift(@data); # status, which is set by rdb->sql, is OK if nonzero
    $no_elements = scalar(@data);  # Find how many records were retrieved.

    if (!$status) {
        &return_error(500, "Database Error", "Sprite database error.");
        } elsif (!$no_elements) {
        &return_error(500, "Database Error", "No records.");
        }

    &print_HTML_Header("Show Entire Database");
```

```
    # Each database record has three fields:
    # ImageLocation,Description,Keywords

    foreach $record (@data) {
        ($ImageLocation, $Description, $Keywords) = @$record;

        print "<HR>\n";
        print "<A HREF=\"$ImageLocation\">\n";
        print "$ImageLocation</A>";
        print "<P><B>Description:</B> $Description<BR>\n";
        print "<B>Keywords:</B> $Keywords<BR><P>\n";
        }

    &print_HTML_Trailer;
    }
```

The remaining subroutines are shown below.

```
#
# ----- SUBROUTINE print_query
#
# Create an HTML page that results from a query.
#

sub print_query {
    # Create a new Sprite database "object"
    $rdb = new Sprite();

    # Read in all records in the database, which are separated by commas.
    $rdb->set_delimiter("Read", ",");

    @data = $rdb->sql(
      "select * from $DATABASE_LOCATION where (Keywords =~ /$FORM{'keyword'}/)"
      );
    $rdb->close();

    $status = shift(@data); # status, which is set by rdb->sql, is OK if nonzero
    $no_elements = scalar(@data);  # Find how many records were retrieved.

    if (!$status) {
        &return_error(500, "Database Error", "Sprite database error.");
        } elsif (!$no_elements) {
        &return_error(500, "Database Error", "No records.");
        }

    &print_HTML_Header("Show Query Results");

    # Each database record has three fields:
    # ImageLocation,Description,Keywords

    foreach $record (@data) {
        ($ImageLocation, $Description, $Keywords) = @$record;
```

```
        print "<HR>\n";
        print "<A HREF=\"$ImageLocation\">\n";
        print "$ImageLocation</A>";
        print "<P><B>Description:</B> $Description<BR>\n";
        print "<B>Keywords:</B> $Keywords<BR><P>\n";
        }

    &print_HTML_Trailer;
    }

#
# ----- SUBROUTINE process_update
#
# Add a record to the database.
#

sub process_update {

    # Check for the password:
    if (!($FORM{'Password'} eq $PASSWORD)) {
        &return_error(500, "Password error",
            "You are not authorized to make changes to the database.");
        }

    # Create a new Sprite database "object"
    $rdb = new Sprite();

    # Prepare the database for writing
    $rdb->set_delimiter("Read", ",");
    $rdb->set_delimiter("Write", ",");

    $rdb->sql(<<End_of_Insert) || &database_error();

insert into $DATABASE_LOCATION
    (ImageLocation, Description, Keywords)
values
    ('$FORM{'URL'}', '$FORM{'Description'}', '$FORM{'Keywords'}')
End_of_Insert

    $rdb->close($DATABASE_LOCATION);

    &print_main_form();
    }

#
# ----- SUBROUTINE parse_form_data
#
# Read in the parameters uploaded from the browser to the server.  If
# there are multiple entries with the same name, like:
#       FILENAME=file1.gif
#       FILENAME=file2.gif
# then the entries are catenated like this:
#       file1.gif\0file2.gif
# and the catenated entries are assigned to key FILENAME.
#
sub parse_form_data {
    # "@_" is a magic variable that holds argument passed to the subroutine.
```

```
    local(*FORM_DATA) = @_;

    # When a variable is declared "local", its scope is limited to
    # the subroutine.
    local($request_method, $query_string, @key_value_pairs,
        $key_value, $key, $value);

    # This same code will work for either the GET or POST methods
    $request_method = $ENV{'REQUEST_METHOD'};

    if ($request_method eq "GET") {
        $query_string = $ENV{'QUERY_STRING'};
    } elsif ($request_method eq "POST") {
        read (STDIN, $query_string, $ENV{'CONTENT_LENGTH'});
    } else {
        &return_error (500, "Server Error",
            "Browser uses unsupported method");
    }

    # The browser uploads information like this:
    # key1=value1&key2=value2&key3=value3
    # The next line breaks the input into key/value pairs.
    @key_value_pairs = split(/&/, $query_string);

    # Now, break each key/value pair into a key and a value.  Also, undo
    # any damage the browser had to do encoding special characters like
    # spaces, plus signs, etc.
    foreach $key_value (@key_value_pairs) {
        ($key, $value) = split(/=/, $key_value);
        $value =~ tr/+/ /;
        $value =~ s/%([\dA-Fa-f][\dA-Fa-f])/pack ("C", hex ($1))/eg;

        if (defined($FORM_DATA{$key})) {
            $FORM_DATA{$key} = join("\0", $FORM_DATA{$key}, $value);
            } else {
            $FORM_DATA{$key} = $value;
            }
        }
    }


#
# ----- SUBROUTINE return_error
#
sub return_error {
    local($status, $keyword, $message) = @_;

    print"Content-type: text/html", "\n";
    print"Status: ", $status, " ", $keyword, "\n\n";

    print <<End_of_Error;

<HTML>
<HEAD>
    <TITLE>CGI Program - Unexpected Error</TITLE>
</HEAD>
<BODY>
```

```
<H1>$keyword</H1>
<HR>$message<HR>
Please contact $webmaster for more information.
</BODY>
</HTML>

End_of_Error

    exit(1);
    }

#
# ----- SUBROUTINE print_HTML_Header
#
# Print the header for a Web page
#
sub print_HTML_Header {
    local ($title) = @_;
     print <<HTML_Header;
Content-type: text/html

<HTML>
<HEAD>
<TITLE>PICTURE DATABASE -- $title</TITLE></HEAD>
<BODY BGCOLOR="#FFFFFF" TEXT="#000000" LINK="#0000FF" VLINK="#FF0000"
ALINK="#808080">
<BASEFONT FACE="Verdana,Arial,Helvetica">
<P>
<br>
<P>
<table border="0" width="550">
  <tr>

<td align="left">
<FONT size=+2><B>PICTURE DATABASE<BR>$title</B>

<HR>
<P>

<P>
</FONT>
</td>
<P>
  </tr>
  <tr>
    <td><blockquote>
      <p align="left"><font face="Arial,Helvetica" size=+0>

HTML_Header
    }


#
# ----- SUBROUTINE print_HTML_Trailer
#
# Print the trailer for a Web page
#
```

```
sub print_HTML_Trailer {

    print <<HTML_Trailer;
<P>
<HR>
<P>
<CENTER>

<A HREF="$DOCUMENT_ROOT/Pictures.cgi">Back to main page</A>
</CENTER>
    </td>
  </tr>
</table>
<P>
</BODY></HTML>
HTML_Trailer
    }

#
# --------- Subroutine database_error
#
# Simple error routine.
#

sub database_error {
        &return_error(500, "Database Error", "Sprite (Insert) database error.");
}
```

A summary of subroutines, and their operation, is given below.

`print_main_form` Creates the top-level Web page.
`print_showall` Creates a Web page that shows the database contents.
`print_query` Creates a Web page that allows the user to query the database.
`process_update` Creates a Web page that allows the user to update (add a record to) the database.
`parse_form_data` Reads data from the Web browser and breaks it into key-value pairs.
`return_error` Creates a Web page that indicates an error.
`print_HTML_Header` Prints the leading HTML text for a Web page.
`print_HTML_Trailer` Prints the trailing HTML text for a Web page.
`database_error` Prints an error message.

Part 4: Advanced Topics

About Perl Modules

The Sprite library is packaged into a Perl *module*. A module is identified with a `.pm` extension (for **P**erl **m**odule) as in `Sprite.pm`. The module can be used by invoking the `use` command. The first `use` line below identifies the path to a directory where modules can be found on the server:

```
use lib '/www/cgi-bin/519_scripts/Sprite-3.21';
```

The `use` line below identifies one particular module that we will use. Note that the argument to `use` is `Sprite` even though the actual filename for the module is `Sprite.pm`.

```
use Sprite;
```

A *package* allows the name space within a module to be segregated from the name space in the program that uses it. For example, if there is a variable `$foo` is the Sprite module,which is implemented as a package, then there is no danger that we might accidentally overwrite it by unknowingly choosing a variable name of `$foo` in our own program because the package has its own name space.

If the module is *object-oriented*, which is the case for Sprite, then we can create an instance of the module (an instance is an object) with the line:

```
$rdb = new Sprite();
```

The subroutines and variables in the Sprite module can then be accessed through the `$rdb` handle, which is how the name space for this instance of Sprite can be segregated from the main program. For example, the Sprite `set_delimiter` subroutine, which defines the delimiter that separates fields in input records (a comma for this case), can be accessed like this:

```
$rdb->set_delimiter("Read", ",");
```

Although this mechanism keeps the name spaces separate, only some of the Sprite module is implemented as a package. Several variables are common across multiple instances of Sprite, and this will create unpredicatble behavior if two instances of the Sprite database are used simultaneously by the same program.

Socket Programming: Background on The Internet

In the early days of computing, computers were centralized facilities that contained most or all of the resources used by the populations they serviced. Data was transferred between computers via media (punched paper cards, paper tapes, magnetic tapes, and magnetic disks), hand-carried by an operator. As the number of computers increased, and costs shifted away from hardware and more toward labor, it became economical to directly link computers so that resources could be shared. This is what networking is about. Here, we take a look at architectural aspects of computer networks in preparation for *socket programming*, which is a methodology for network programming.

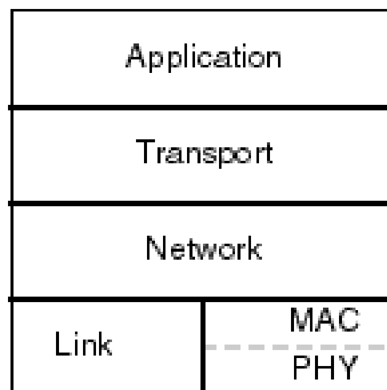
The Internet Model

In a telecommunication system there may be many sources and many destinations. An example of this form of communication is a long distance telephone network. For every telephone to be reachable from every other telephone, there must be a path, or channel, between each source and destination. If there are 10^7 telephones in New York City and 10^7 telephones in Chicago, then for everyone in one city to be able to call everyone in the other city, $10^7 \times 10^7 = 10^{14}$ channels must exist between the cities. Fortunately, not everyone in New York City wants to talk with everyone in Chicago at the same time, and a smaller number of channels between New York City and Chicago can be shared among all telephones in those cities. On the other hand, there must be at least one line from each telephone to the telephone company's central office, and there must be a sufficient number of lines between central offices to handle the maximum number of simultaneously held conversations.

A small number of physical connections, on the order of a few to a few thousand depending on whether fibers or wires are used, are all that are needed to connect the cities because it is never the case that everyone in one city wants to call someone in the other city at the same time. The information carrying capacity of the connections (called bandwidth) is shared among all of the users so that a dramatic reduction in cost is realized. A control mechanism must be created, however, so that the bandwidth can be shared properly.

Layering in the TCP/IP Protocol Suite

An "internet" is a collection of interconnected networks. The "Internet" is probably the most well-known internet, using the TCP/IP protocol and IP addresses in what is known as the TCP/IP protocol suite (more on this below). The traditional 7-layer OSI model is simplified somewhat in the Internet, which can be thought of as having only 4 layers, as illustrated by the protocol stack shown below. At the bottom of the protocol stack is the Link layer, which is made up of the medium access control (MAC) and physical (PHY) sublayers. The Link layer resolves contention for the medium when more than one device wants to transmit, manages the logical grouping of bits into frames, and implements error protection.



The Link layer is responsible for simply getting a frame of bits from one machine to a directly connected machine. This is fine for point-to-point communication between two cooperating processes on different machines. In order for multiple processes to share the same link, however, a protocol is needed to coordinate which data goes to what process. This is the responsibility of the Network layer, which is implemented with the Internet Protocol (IP) for the Internet.

The Network layer deals with hop-by-hop communication. The Transport layer deals with end-to-end communication, in which there may be a number of intervening systems between the sender and

receiver. The Transport layer deals with retransmission (for errors, or packets dropped due to congestion), sequencing (packets may arrive out-of-order), flow control (applying back-pressure to the source to relieve congestion) and error protection (the Link layer does not do enough error protection on its own.) For the Internet, the Transport layer is implemented with the Transmission Control Protocol (TCP). The TCP/IP combination at the Network and Transport layers is the predominent Internet protocol suite. Any other appropriate protocols can be used at the Link and Application layers, and there are also other protocols used within the Network and Transport layers.

At the Application layer, a process can exchange data with another process anywhere on the Internet and treat the connection as if it is a file on the local system, reading and writing bytes with ordinary read and write system calls, frequently implemented by *sockets*, which are pathways to the network through the operating system.

Internet Addresses

Every interface on the Internet has a unique IP address. Version 4 of the IP protocol, known as IPv4, is still widely used but is gradually being replaced by IPv6 which uses addresses that are four times larger, and has several enhancements and simplifications to IPv4. An example of an IPv4 address, shown in "dotted decimal notation" is shown below:

```
165.230.140.67
```

Each number that is delimited by a dot is an unsigned byte in the range from 0 through 255. The equivalent bit pattern for the IPv4 address shown above is then:

```
10100101.11100110.10001100.10000011
```

The leftmost bits determine the class of the address. The figure below shows the five IPv4 classes. Class A has 7 bits for the network identification (ID) and 24 bits for the host ID. There can thus be at most 27 class A networks and 224 hosts on each class A network. A number of these addresses are reserved, and so the number of addresses that can be assigned to hosts is fewer than the number of possible addresses.

Class B addresses use 14 bits for the network ID and 16 bits for the host ID. Class C addresses use 21 bits for the network ID and 8 bits for the host ID. Class D addresses are used for *multicast* groups, to which an end-system that has a class A, B, or C address subscribes, and thereby receives all network traffic intended for that group. This is an efficient mechanism for sending the same packets to multiple subscribers, without flooding the network with broadcasts, and without the sender needing to keep track of all of the current subscribers. Class E addresses are unused.

The available supply of IPv4 addresses will run out soon after the year 2000, and so it is important that IPv6 be widely adopted soon. Already, many networks reuse IP addresses that are simultaneously in use elsewhere (using a protocol that allows for sharing of IP addresses), and others assign IP addresses only for the duration of a session (such as for a dialup line through a modem.)

Ports

Loosely speaking, a *port* is how a process is known to the world. A port number identifies the source process, and a port number also identifies the destination process. Strictly speaking, the port identi-

fies a network entry point for a process. Ports 0-1023 are *well-known ports* for server processes. For example, the telnet port is 23. On a Unix or Windows system, the following command:

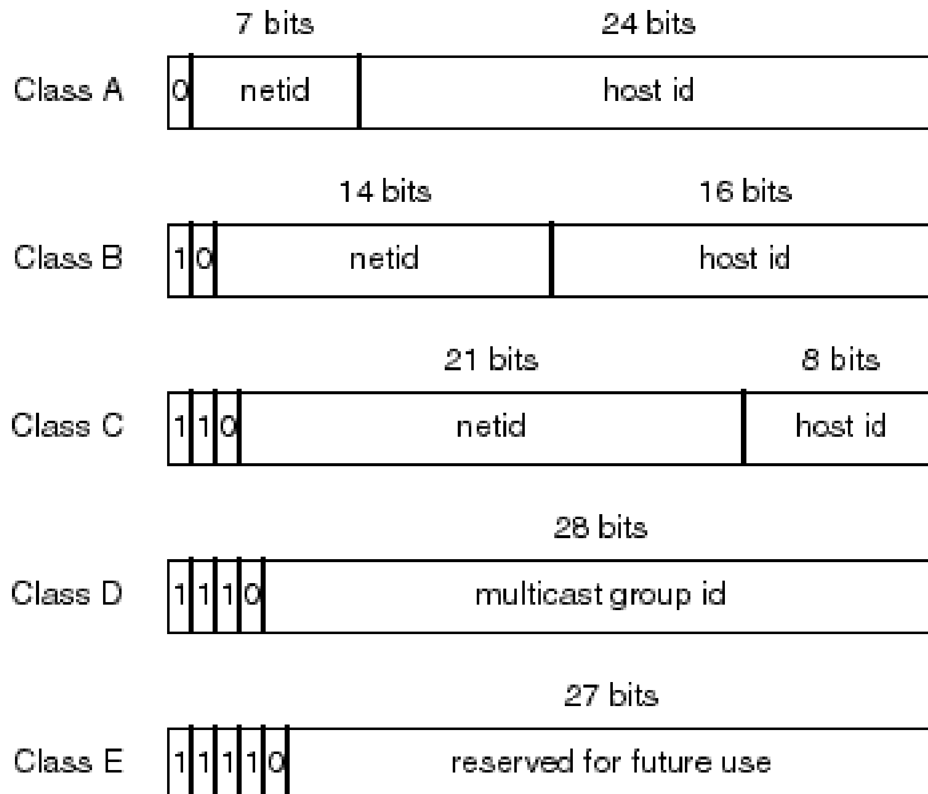
```
% telnet internetinstituteusa.com 23
```

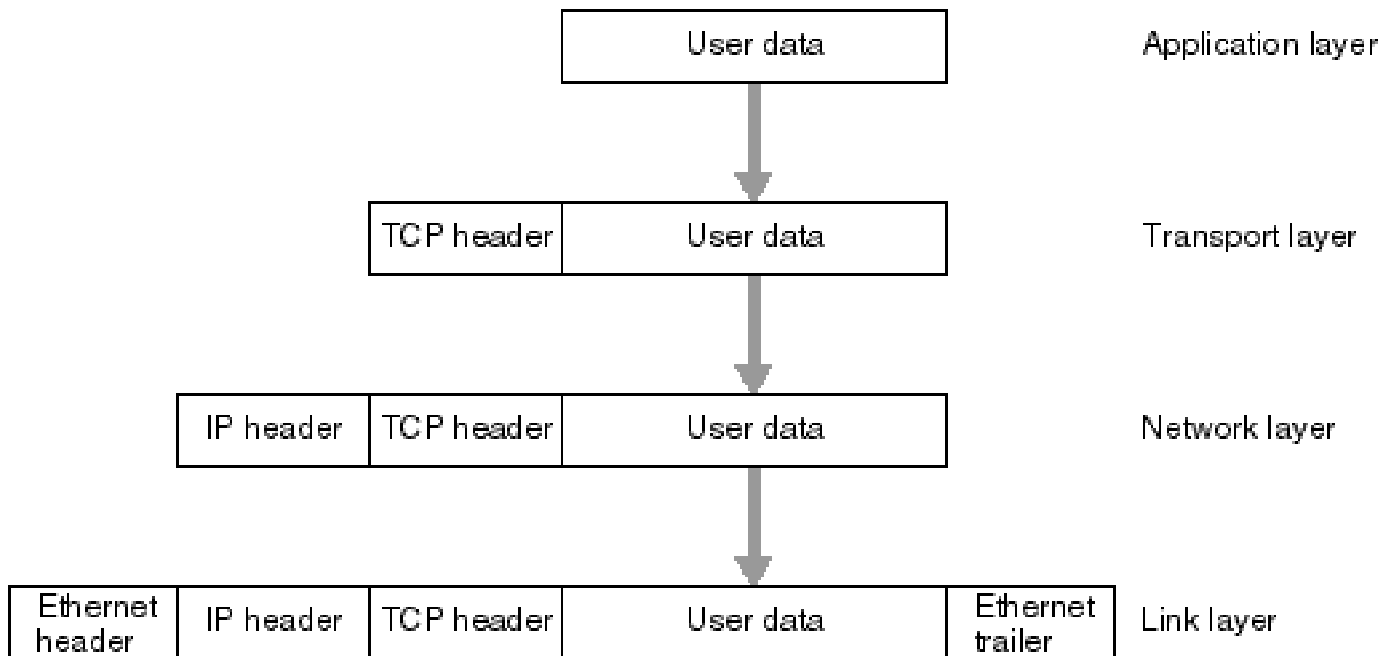
will connect the user to system `internetinstituteusa.com`. If the 23 is not present on the command line, then 23 is assumed. If 23 is replaced with another port, such as 13 for the daytime server, then a different process will be reached, with different resulting behavior.

Encapsulation

Network data is encapsulated as it passes through the network layers, as illustrated in the figure on the next page. The user data is sent to the network using similar read and write system calls that would be used for reading and writing files. The application layer sends user data to the Transport layer, where the operating system adds a TCP header that identifies the source and destination ports, forming a TCP segment. The TCP segment is passed down to the network layer, where the TCP segment is repackaged into IP datagrams, each with an IP header identifying the source and destination systems. The IP datagrams are sent to the Link layer, where the datagrams are encapsulated into Ethernet frames (for this example). The reverse process takes place on the receiving system.

A single TCP segment may be decomposed into a number of IP datagrams, that are independently routed through the Internet. Each IP datagram contains the source and destination IP addresses (in the IP header), the source and destination ports (in the TCP header), and the protocol at the next layer of encapsulation (in the IP header – TCP is only one of the transport layer protocols used in the Internet.) Collectively, these five parameters uniquely identify each IP datagram as it traverses the Internet, which helps ensure that the datagrams arrive at the correct receiving process.





The Domain Name System

The Domain Name Systems (DNS) is a distributed database that maps between hostnames and IP addresses, and provides mail routing information. For example, `internetinstituteusa.com` maps to 216.3.196.244 (and vice versa), and both names: `internetinstituteusa.com` www.internet.rutgers.ed and `www.internetinstituteusa.com` map to 216.3.196.244. The DNS is responsible for interacting with programs that need to map between names and addresses.

Each domain (like `internetinstituteusa.com`) maintains its own database of information, and runs a server that other systems across the Internet can query. Access to the DNS is provided through a *resolver* which is embodied in library routines that are silently linked into high-level programs that access the network.

The Network Information Center (NIC, also known as the InterNIC) manages the top-level domains, and delegates authority for second level domains. Within a *zone*, a local administrator maintains the name server database. There must be a primary name server, which loads its database from a file, and secondary name servers, which get their information from the primary name server. *Caching* is used, so that a query that causes other servers to be contacted does not cause future queries to cause additional contacts to other servers.

The World Wide Web

The World Wide Web (or simply, the “Web”) is made up of client processes (Web browsers) and Web servers running the HyperText Transport Protocol (HTTP), at the Application layer of the Internet. As distinctions get blurred in everyday usage, it is important to keep in mind that the Web is built on top of the Internet – the Web is not the Internet itself.

In 1989, Tim Berners-Lee at CERN (the European high-energy physics facility) developed a text based Web, for exchanging technical documents among colleagues. In February 1993, the National Center for Supercomputing Applications (NCSA) at the University of Illinois at Urbana-Champaign

released a graphical version of the Mosaic Web browser, as well as an HTTP server, both free of charge, and the Web exploded to where it is today.

Socket Programming: Methods

Sockets provide a means for network programming, in which writing to and reading from processes over a network is treated similar to the way that writing to and reading from files on a local file system is handled.

In the first of three client/server examples below, a sample TCP client/server pair is shown. The client program simply connects to the server, prints whatever the server sends, and then ends.

First, we look at the client, in which the expected usage is:

```
perl client.pl hostname portno
```

where `hostname` is replaced with the name of the host where the server is located, and `portno` is replaced with the port number of the server.

```
#!/usr/bin/perl -w

# Source: page 349 of "Programming Perl", Larry Wall, Tom Christiansen, and
# Randal L. Schwartz, O'Reilly, (1996).

require 5.002;
use strict;
use Socket;
my ($remote, $port, $iaddr, $paddr, $proto, $line);

$remote = shift || 'localhost';
$port = shift || 2345;  # random port
if ($port =~ /\D/) {$port = getservbyname($port, 'tcp') }
die "No port" unless $port;
$iaddr = inet_aton($remote) or die "no host: $remote";
$paddr = sockaddr_in($port, $iaddr);

$proto = getprotobyname('tcp');
socket(SOCK, PF_INET, SOCK_STREAM, $proto) or die "socket: $!";
connect(SOCK, $paddr) or die "connect: $!";
while (defined($line = <SOCK>)) {
    print $line;
}

close (SOCK) or die "close: $!";
exit;
```

The `-w` flag for the Perl interpreter turns on warning messages, which can be helpful while debugging. The `require` keyword brings in the 5.002 Perl module at runtime. The `use strict` construct forces references inside of modules to be fully qualified, which serves as an aid to ensure that name spaces are properly separated. For example, if variable `foo` is in the `Socket` module, then it must be referenced as `Socket::foo`. The `use Socket` construct brings in the `Socket` module

at compile time. The `my` keyword declares the variables that follow to be lexically scoped within the block. That is, these variables are visible to no other subroutines.

The shift operator extracts the next command line argument. If the hostname is not present, then `localhost` is used, which is the same host on which the client program runs. If the port number is not present, then a default of 2345 is used. Any available port above 1000 and below 32768 can be used, as long as it matches the port used by the server. If a port is given by its name, such as `smtp`, then `getservbyname` finds the integer port number associated with the name. Otherwise, the integer port number given on the command line is used.

The `inet` subroutine returns the Internet address of the server. The `sockaddr` subroutine returns an address structure that references the internet address/portno pair. The `getprotobyname` subroutine gets the integer associated with the TCP protocol. The `socket` call creates a socket with the given parameters. The socket at this point can be thought of as a telephone, waiting to be used. The `connect` call initiates a connection with the server through the socket, and can be thought of as dialing a telephone number and waiting for an answer. The code that follows reads lines from the server and prints them out until the server closes the connection, at which point the client closes the socket and the program exits.

And now, the server code:

```
#!/usr/bin/perl -Tw

# Source: page 349 of "Programming Perl", Larry Wall, Tom Christiansen, and
# Randal L. Schwartz, O'Reilly, (1996).

require 5.002;
use strict;
BEGIN { $ENV{PATH} = '/usr/ucb:/bin' }

use Socket;
use Carp;

sub logmsg {print "$0 $$: @_ at ", scalar localtime, "\n" }

my $port = shift || 2345;
my $proto = getprotobyname('tcp');
socket(Server, PF_INET, SOCK_STREAM, $proto) or die "socket: $!";
setsockopt(Server, SOL_SOCKET, SO_REUSEADDR, pack("l", 1))
    or die "setsockopt: $!";
bind(Server, sockaddr_in($port, INADDR_ANY)) or die "bind: $!";
listen(Server,SOMAXCONN) or die "listen: $!";

logmsg "server started on port $port";

my $paddr;

$SIG{CHLD} = \&REAPER;

for ( ; $paddr = accept(Client,Server); close Client) {
    my($port,$iaddr) = sockaddr_in($paddr);
    my $name = gethostbyaddr($iaddr,AF_INET);
```

```
    logmsg "connection from $name [",
        inet_ntoa($iaddr), "] at port $port";

    print Client "Hello there, $name, it's now ",
        scalar localtime, "\n";
}
```

The `BEGIN` keyword immediately executes, before the rest of the file is parsed. In this case, `BEGIN` causes the `PATH` environment variable to be set to `/usr/ucb:/bin`, which is needed to find the `localtime` program on the Unix server. The `use Carp` construct helps in error reporting – any errors will be reported according to where the error happens, which allows library routines to appear more like in-line code. The `REAPER` call is more significant for the multi-threaded server described next, but is used here because it may be needed even for a single-threaded version. When a server *forks* a process, then the process is maintained in the system process table of the host even after it dies, waiting for the parent process to die. In many situations, the parent process never dies (like for a Web server.) The `REAPER` eliminates these "zombie" processes. The remainder of the code is similar to the client and other Perl code shown earlier in this tutorial, except that the socket is created as a Server socket, which allows it to "sleep" until a client connects to it.

A problem with this server is that while it is servicing a client, it excludes access by all other clients. This may be a desired behavior for some servers, but for other servers, such as Web servers, multiple clients may need to be simultaneously serviced. This behavior can be accomplished by *forking* a copy of the server process for each client that connects to it, so that the main server is always ready to accept a new connection. A multi-threaded version (as this is called) of the server is shown below:

```
#!/usr/bin/perl -Tw

# Source: page 350 of "Programming Perl", Larry Wall, Tom Christiansen, and
# Randal L. Schwartz, O'Reilly, (1996).

require 5.002;
use strict;
BEGIN { $ENV{PATH} = '/usr/ucb:/bin' }

use Socket;
use Carp;
use FileHandle;

sub spawn; #forward declaration
sub logmsg {print "$0 $$: @_ at ", scalar localtime, "\n" }

my $port = shift || 2345;
my $proto = getprotobyname('tcp');
socket(Server, PF_INET, SOCK_STREAM, $proto) or die "socket: $!";
setsockopt(Server, SOL_SOCKET, SO_REUSEADDR, pack("l", 1))
    or die "setsockopt: $!";
bind(Server, sockaddr_in($port, INADDR_ANY)) or die "bind: $!";
listen(Server,SOMAXCONN) or die "listen: $!";

logmsg "server started on port $port";

my $waitedpid = 0;
```

```
my $paddr;

sub REAPER {
    $waitedpid = wait;
    $SIG{CHLD} = \&REAPER; # if you don't have sigaction(2)
    logmsg "reaped $waitedpid" . ($? ? " with exit $?" : "");
}
$SIG{CHLD} = \&REAPER;

for ( ; $paddr = accept(Client,Server); close Client) {
    my($port,$iaddr) = sockaddr_in($paddr);
    my $name = gethostbyaddr($iaddr,AF_INET);

    logmsg "connection from $name [",
        inet_ntoa($iaddr), "] at port $port";

    spawn sub {
        print Client "Hello there, $name, it's now ",
            scalar localtime, "\n";
    exec '/usr/games/fortune' or confess "can't exec fortune: $!";
    };
}

sub spawn {
    my $coderef = shift;

    unless (@_ == 0 && $coderef && ref($coderef) eq 'CODE') {
    confess "usage: spawn CODEREF";
    }

    my $pid;
    if (!defined($pid = fork)) {
        logmsg "cannot fork: $!";
        return;
    } elsif ($pid) {
        logmsg "begat $pid";
        return; # I'm the parent
    }
    # else I'm the child -- go spawn

    open(STDIN, "<&Client") or die "can't dup client to stdin";
    open(STDOUT, ">&Client") or die "can't dup client to stdout";
    STDOUT->autoflush();
    exit &$coderef();
    }
```

Setting Up a CGI Script on a Unix Server

In this section, we will create a CGI script that runs on `internetinstituteusa.com` that executes the "who" program on internet, and sends the output back to the user. This same procedure can be used on a different Unix based Web server by substituting the name `internetinstituteusa.com` with the name of the server machine.

Setting Up the cgi-bin Directory
(NOTE: At IIUSA, you can skip this section because these directories and the sample home page are automatically set up, similar to the way that some Internet service providers do it.)

In your home directory on `internet.rutgers.edu`, create a subdirectory called `public_html` that is readable and executable by all. This is the default for the Apache Web server, and is a configurable option that may be different on other systems. Your home directory should also be readable by all, or at least executable by all, otherwise, the `public_html` subdirectory will not be readable by all. Create your home page in file `index.html` in the `public_html` subdirectory using an ordinary text editor. Again, make sure that `index.html` is readable by all, as well as any other files that you make hypertext links to. Do not make the file executable, however. Only directories and executable files should be made executable on Unix.

You can make a file readable by all with:

```
% chmod a+r filename
```

You can make a directory readable and executable by all with:

```
% chmod a+rx public_html
```

The directory must be executable by all in order for the files to be readable by all, which is a peculiarity of how Unix handles permissions.

Type:

```
% man chmod
```

for more information on the `chmod` command.

When your home page is finished, it should be accessible to the outside world via the Web. If your login id is `iiusa00xy` and your home computer is "`internetinstituteusa.com`", then anyone with access to the Web, anywhere in the world, can reach your home page by using a browser like Netscape to open the Uniform Resource Locater (URL):

```
http://internetinstituteusa.com/~iiusa00xy/
```

If you do not include the closing slash at the end of the URL, it works fine without it because Web servers figure out what to do. The `internet.rutgers.edu` server uses a default target filename of `index.html` if you do not specify a filename with a "`.html`" extension after the closing slash (`/`). "`index.html`" is the name you should use for your home page. (Note that on some other systems, the default home page is `index.htm`, not `index.html`.)

The CGI scripts are located in a special directory in which the HTTP server knows to look. The location is a configurable option that differs from one system to another. On `internetinstituteusa.com`, the CGI script directory is located in `~username/public_html/cgi-bin`. So if your login id on internet is "`iiusa00xy`", then your cgi-bin directory is in `~iiusa00xy/public_html/cgi-bin`.

You can create the directory and set its permissions properly with:

```
> cd
> cd public_html
> mkdir cgi-bin
> chmod a+rx cgi-bin
```

Any CGI scripts that you place in that directory can contain a compiled C program, a shell script, a Perl script, a Java program, or any other executable or interpreted program. Note that all CGI programs on internet.rutgers.edu must be in files with names that end with "`.cgi`".

Common Gateway Interface (CGI)
For this part of the tutorial, we will create a CGI script that executes the Unix `who` program on internet.rutgers.edu and sends the output back to the end-user. The Unix `who` program gives a listing of who is currently logged into the system.

Here is an example of how it should work (click on the hyperlink):

`http://internetinstituteusa.com/cgi-bin/519_scripts/who.cgi`

You will need to create a Perl script that implements this operation and place it in your own account. Here is what the Perl code looks like:

```
#!/usr/bin/perl
# The name of this file is "who.cgi".

$names = `/usr/bin/who`;

print <<END;
Content-type: text/plain

Here is who is logged onto internetinstituteusa.com:

$names
END
```

You create this program on another system (like a Windows machine or a Macintosh) and transfer it to your internet.rutgers.edu account using FTP.

Using FTP to Transfer a Perl Script to Your IIUSA Unix Account
If you create the `who.cgi` file on a Windows machine (using `notepad` or some other word processor) or on a Macintosh (using `SimpleText` or another word processor) or on some other machine, then you can transfer it to `internetinstituteusa.com`using the FTP (file transfer protocol) program that comes standard in most telecommunications packages. FTP is built into Windows -- from the START menu, open an MS-DOS window and type the first line below that starts with "`ftp`"; you can use the "`cd`" command to change directories in the MS-DOS window, for example, use "`cd c:\`" to move to the top level of your C drive and then type "`dir`" to see what files and directories are visible at that level.

Here is an example of what to type:

```
> ftp internetinstituteusa.com
Connected to internetinstituteusa.com.
220-
220- This ftp server is for AUTHORIZED USE ONLY.  A valid email address
220- for anonymous ftp transfers is REQUIRED!  All transfers are logged.
220-
220- Use a dash (-) as the first character of your password to turn off
220- the continuation messages.
220-
220- Report problems to help@iiusa.cc
220-
220 internetinstituteusa.com FTP server (Version 4.3) ready.
Name (mybusiness.com:username): iiusa00xy
331 Password required for iiusa00xy.
Password:
230 User iiusa00xy logged in.
ftp> cd public_html
250 CWD command successful.
ftp> cd cgi-bin
250 CWD command successful.
ftp> ascii
200 Type set to A.
ftp> put who.cgi
200 PORT command successful.
150 Opening ASCII mode data connection for who.cgi.
226 Transfer complete.
local: who.cgi remote: who.cgi
199 bytes sent in 0.00062 seconds (3.1e+02 Kbytes/s)
ftp> quit
221 Goodbye.
```

The "`ascii`" line above should be used when transferring printable text. The keyword "`ascii`" should be replaced with "`binary`" when transferring images or other multimedia files. In some ftp programs, you may need to type these commands in upper case as "ASCII" and "BINARY", respectively.

If you are on a Windows machine, then when you go to the START menu and open an MS-DOS window, that will probably place you in the top-level `C:` directory. The sequence above assumes that your Perl script is also in that same directory (top level of your C drive).

Use "`quit`" to get out of FTP, and "`exit`" to close the MS-DOS window.

Then, after transferring your Perl script to your Rutgers account, telnet to internet, and after logging in, type:

```
cd public_html
cd cgi-bin
chmod a+rx who.cgi
```

This changes the mode of `who.cgi` (replace "`who.cgi`" with "`filename.cgi`" for a Perl script

named "`filename.cgi`") to be executable. This is required for systems at Rutgers, but if you use a Web hosting service, there may be no need to do this additional step of changing the mode to be executable. A simpler way to do this without opening a telnet window, is to do it before quitting the ftp program with:

```
ftp> !chmod a+rx who.cgi
```

You can now run the Perl script via the Web at:

```
http://internetinstituteusa.com/~iiusa00xy/cgi-bin/who.cgi
```

(Replace `iti00xy` with your account name).

Heads Up on CGI
A common mistake is to have the read or execute permissions set incorrectly. It takes some getting used to, but you have to remember that the HTTP server has no way of knowing that you are really you when you access your script. That is, the HTTP server treats every client as an outsider, even when the client user is you. For that reason, every HTML file that you want the browser to use must be readable by all, and all of the directories along the path from your home directory to the HTML file must be executable by all. Further, any CGI scripts/executables must be executable by all. A compiled CGI program does not need to be readable by all, but an interpreted program does need to be readable by all.

So: if the HTTP server treats all clients as an "outsider", then can my CGI script create a file in my cgi-bin directory or not? The answer is "no", unless you do some special things (which can be dangerous from a security standpoint) to allow it. The HTTP server runs under the username "no-body" (or "www" depending on how the server is configured), which is a user who does not have write permissions in your directories. You should write any temporary files to `/tmp`.

If you follow this thread a little farther, that means that "nobody" will own any created files, and you can not read or remove them unless nobody sets the permissions that way. So, any files that your CGI script creates in `/tmp` (or wherever) cannot be removed by you, but they *can* be removed by your CGI script when the HTTP server runs your CGI program.

It only takes a little thinking ahead to realize that "nobody" can read/write/destroy anything owned by nobody, whether they are files or executing processes created by your script or by anyone else's script on internet.rutgers.edu. For example, you can create a CGI script that kills all processes that belong to nobody, and thereby kill any CGI programs that belong to others.

Hopefully you can see the security risk here and understand why CGI is not generally made available to ordinary users by Internet service providers.

Projects:

After Part #1:

Here are a few experiments that should be accessible after working through Part #1 of this tutorial:

Implement your own server redirection CGI script on internet.rutgers.edu, using the Location: directive to send the Web browser to some other server, such as www.cnn.com.

Copy the sampleform2.cgi script to your own computer account, modify the section to send email to your own email account (which can be anywhere; it does not have to be at rutgers.edu), and also copy the HTML form to your own account (which does not have to be at rutgers.edu either) and modify it to access your Perl script instead of the default script used above.

After Part #2:

Create a fill-out form and a corresponding CGI Perl script that allows the user to select parameters for a calendar. The Perl script then either emails that calendar to a recipient, or displays it on the screen.

The fill-out form uses a single-line text field for an email address where the calendar will be sent. A pull-down list (that is, a "menu") gives the user a selection of years to choose (list a few years in the menu). Two radio buttons allow the user to select (1) that the calendar will be sent via email, or (2) that the calendar will only be displayed on the screen (do not use email in that case). For both cases, some sort of reply screen should be sent to the user (a confirmation that the email was sent, and/or an image of the calendar.)

The same Perl script should work for both the GET and POST methods. Provide two URL's on your fill-out form so that both methods can be exercised with a mouse click for each.

After Part #3 and Part #4:

Create a text database that allows records to be added or modified, and searched. The database should be accessible entirely via CGI. At a minimum, the database should:

1. Have a single CGI script that generates at least four different Web pages (after Part #3).
2. Prompt a user for a password when additions or modifications are made (after Part #3). Additions and modifications do not have to be implemented until Part #4 has been covered.
3. Allow searches to made on one or more fields (after Part #4).
4. Create a new Web page, with a unique name, that contains a record selected by the user. Send an email message to a recipient entered by the user informing them of the URL for the new page. This is similar to the electronic greeting cards that can be sent from http://www.bluemountain.com (after Part #4).

For an example of the database portion of the expected behavior, see:
`http://internetinstituteusa.com/cgi-bin/519_scripts/Pictures.cgi`

For more online information on Perl, including a complete reference of syntax and built-in functions, see: http://www.perl.com/pub/v/faqs.