Bonus Chapter E

Internet Programming 2: CGI

In the last chapter we saw how to set up HTML pages containing information that could be viewed both locally and across networks. While this is a very good way of publishing information, it is all essentially static information. To supply dynamic information, we need to allow the user, through the Web page, to interact with programs on the server in a dynamic way.

In this chapter, you'll learn how to allow the viewer to send information back to the server and how the server can pass this information to programs, which can then respond to the client in a dynamic way. We are going to look only at server-side processing, that is, where code executes away from the client. We are also going to concentrate on stand-alone server-side programs, rather than scripts embedded in Web pages, such as those used by PHP.

It is also possible to write client-side programs, usually referred to as Dynamic HTML (DHTML), where scripts execute on the client, a topic we are not going to pursue here.

The interface specification that defines how information can be passed from browsers back to the server is called the **Common Gateway Interface**, usually abbreviated to **CGI**. The program accepting such information from a browser, commonly referred to as a **CGI program**, can process this information and use the HTTP protocol to send commands or dynamic documents back to the browser.
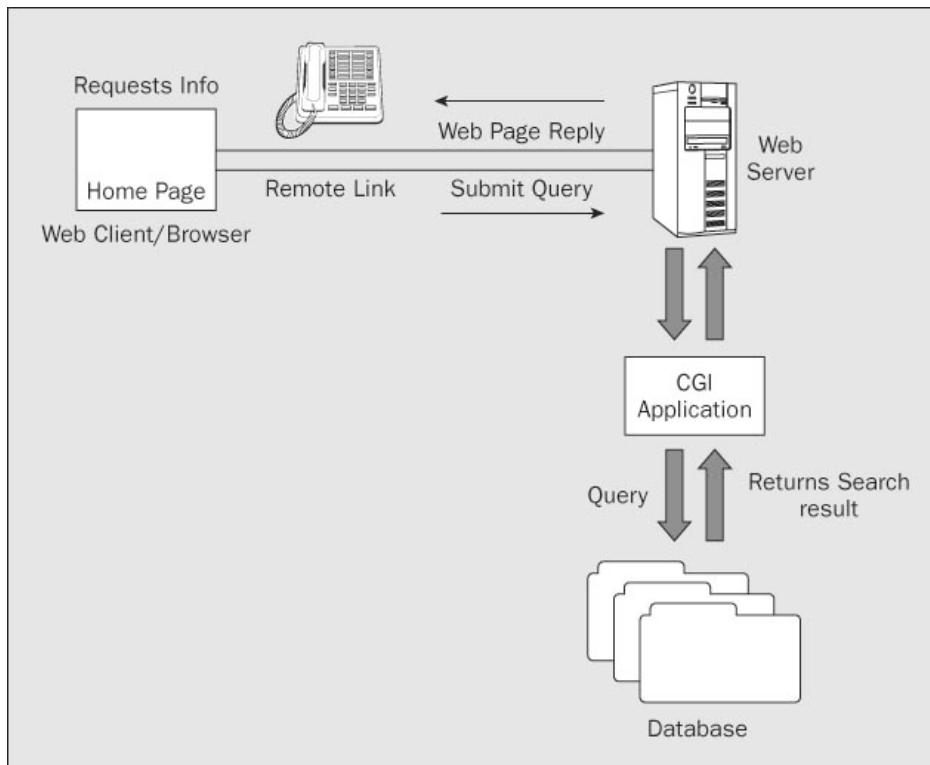
Before we can look at the CGI, we need to look at an HTML construct we haven't yet encountered: the `FORM` element.

FORM Elements

In the HTML that we've seen so far, all the tags have been concerned with controlling the way information is structured for display to the client, but they haven't provided for any input from the browser, other than the selection of hypertext links.

We can set up fields that allow user input using the `<FORM>` tag. This is a composite tag. It can contain several other tags that are only valid inside a form. The `<FORM>` tag and its embedded tags, the form elements, have the following syntax:

```
<FORM ACTION= METHOD= ENCTYPE= >
  <INPUT NAME= TYPE= VALUE= SIZE= MAXLENGTH= CHECKED>
  <SELECT NAME= SIZE= MULTIPLE>
    <OPTION SELECTED VALUE= >
  </SELECT>
  <TEXTAREA NAME= ROWS= COLS= > </TEXTAREA>
</FORM>
```

In addition, we can also use the more common tags that we met in the last chapter inside the `<FORM>` tag. Notice that almost all the tags have a `NAME` attribute, which will be used by the server. We'll return to the processing of the `NAME` attribute when we look at general processing of forms by the server later in the chapter.

As you can see, `<FORM>` is a quite complex tag. Let's look at each element in turn.

The FORM Tag

The **\<FORM>** tag starts an HTML form. It takes three attributes:

❑ The ACTION attribute specifies the URL of the program to invoke to process this form.

❑ The METHOD attribute takes either the value GET or the value POST.

❑ The ENCTYPE attribute is usually omitted, unless you wish to include a file to be sent with the form. If this is the case, then the attribute is set to multipart/form-data, but this is not commonly used. If you are just submitting the form, then use x-www-form-urlencoded, which is the default.

Collectively, these attributes control how information is to be passed back to the server. The ACTION value must point to a program that can be invoked on the server; these are normally stored in a separate subdirectory of the HTTP server from normal pages, which is almost always called cgi-bin. The METHOD attribute controls how information is transmitted to the program on the server. We'll return to programs in cgi-bin and the METHOD attribute later.

The INPUT Tag

The \<INPUT> tag defines input types. The appearance and behavior of the input is controlled with the TYPE attribute. Supported values for the TYPE attribute include the following:

TEXT

When the TYPE attribute takes the value TEXT, the browser will display a single line box into which we can enter text. The NAME attribute gives the field a name, which will be used when the form is processed on the server. The SIZE attribute gives the size of the field on the web page; MAXLENGTH gives the maximum input size allowed. If this is larger than SIZE, the field will scroll to allow input. The VALUE attribute allows a default string to be entered in the field when it's first displayed.

Here's a fragment of HTML, illustrating the TEXT style of input:

```
Please enter your
<BR>
salutation: <INPUT TYPE=TEXT NAME=sal SIZE=5 MAXLENGTH=10 VALUE="Mr. ">
<BR>
first name: <INPUT TYPE=TEXT NAME=fname SIZE=20 MAXLENGTH=30>
<BR>
second name: <INPUT TYPE=TEXT NAME=sname SIZE=20 MAXLENGTH=30>
```

PASSWORD

The PASSWORD value is the same as the TEXT value, except that the contents of the field can't be seen in the browser. It's not very secure as a password, because the contents of the field are passed as plain text to the server, so anyone intercepting the transfer of the form information to the server will be able to read the password. However, it's useful to prevent other people from seeing what's being typed into the field.

Here's a fragment of HTML, illustrating the PASSWORD type:

```
Password: <INPUT TYPE=PASSWORD NAME=passwd SIZE=8>
```

HIDDEN

Setting TYPE=HIDDEN results in a text field that the user doesn't see on the browser screen and can't enter information into. This is used by server programs. Like the PASSWORD type, it's not very secure.

Here's a fragment of HTML, illustrating a HIDDEN field:

```
<INPUT TYPE=HIDDEN NAME=camefrom VALUE="foo/bar/baz.html">
```

CHECKBOX

The CHECKBOX type allows the user to select several of a number of options. Checkbox fields are grouped by specifying several of them with the same NAME attribute. The browser will allow the user to select from the set of checkbox fields having the same name.

The additional attribute CHECKED provides default selections. The VALUE attribute is used in returning the information to the server. To allow the user to see what's being selected, you should associate some text with each checkbox.

Here's a fragment of HTML, illustrating the CHECKBOX type of field. It shows the user a list of areas of the world and allows multiple selections:

```
<FORM ACTION="program" METHOD=POST>
<BR>
Please indicate which areas of the world you would like
to visit:<BR>
<INPUT TYPE=CHECKBOX NAME=cb VALUE="1">Asia<BR>
<INPUT TYPE=CHECKBOX NAME=cb VALUE="2" CHECKED>Africa<BR>
 <INPUT TYPE=CHECKBOX NAME=cb VALUE="3">North America<BR>
<INPUT TYPE=CHECKBOX NAME=cb VALUE="4">South America<BR>
<INPUT TYPE=CHECKBOX NAME=cb VALUE="5">Antarctica<BR>
<INPUT TYPE=CHECKBOX NAME=cb VALUE="6">Europe<BR>
<INPUT TYPE=CHECKBOX NAME=cb VALUE="7" CHECKED>Australasia<BR>
</FORM>
```

RADIO

The RADIO type of <INPUT> tag is very similar to the CHECKBOX type, except that only one option may be chosen and only one of a set may be marked CHECKED. If no default is specified, the browser will make the first of the set checked when the radio buttons are first shown.

Here's a fragment of HTML, illustrating the RADIO type of field. It again shows the user a list of areas of the world, but this time allows only one to be selected:

```
<FORM ACTION="program" METHOD=POST>
<BR>
Please indicate in which area of the world you
live:<BR>
<INPUT TYPE="RADIO" NAME=cb VALUE="1">Asia<BR>
<INPUT TYPE="RADIO" NAME=cb VALUE="2">Africa<BR>
<INPUT TYPE="RADIO" NAME=cb VALUE="3" CHECKED>North America<BR>
<INPUT TYPE="RADIO" NAME=cb VALUE="4">South America<BR>
<INPUT TYPE="RADIO" NAME=cb VALUE="5">Antarctica<BR>
<INPUT TYPE="RADIO" NAME=cb VALUE="6">Europe<BR>
<INPUT TYPE="RADIO" NAME=cb VALUE="7">Australasia<BR>
</FORM>
```

IMAGE

The IMAGE type of <INPUT> tag allows the selection of an (x,y) coordinate from an image, giving a similar input type to image maps. It's very rarely used, as image maps are almost always preferred, so we won't consider it further here.

SUBMIT

The SUBMIT type causes a button to be displayed by the browser. When this button is selected, the contents of the form are sent back to the server for processing. Almost all forms have a SUBMIT button, the label for which is given by the VALUE attribute.

Here's an example:

```
<INPUT TYPE=SUBMIT NAME=processnow VALUE="Submit Form">
```

RESET

TYPE=RESET also causes a button to be displayed by the browser. When this button is selected, the form elements are reset to the values displayed when the form was first loaded. This is done by the browser and doesn't cause any interaction with the server. The RESET type can also take a VALUE attribute. Here's an example:

```
Clear the form: <INPUT TYPE=RESET VALUE="Reset Form">
```

The SELECT Tag

The <SELECT> tag allows the user to select from a list of values. It has the attributes NAME, MULTIPLE, and SIZE.

The NAME attribute, as usual, is used when information is returned to the server. The attribute MULTIPLE allows more than one option to be selected, but only one option can be seen at a time. It's not often used, because checkboxes are often a better choice. The SIZE attribute specifies the number of items that may be selected. The default is one and gives a selection list from which one element can be selected.

Each item in a <SELECT> tag is specified with an <OPTION> tag, with the attribute VALUE giving the value returned to the server when a selection is made. The optional attribute SELECTED specifies a default value.

An example of the SELECT tag is

```
<SELECT NAME="Speed">
<OPTION VALUE="vslow">9600 or slower
<OPTION VALUE="slow">14400
<OPTION SELECTED VALUE="OK">28800
<OPTION VALUE="quick">better than 28000
</SELECT>
```

The TEXTAREA Tag

The <TEXTAREA> tag allows multiple input lines to be entered and returned to the server. In addition to the usual NAME attribute, it has ROWS and COLS attributes to specify the size of the area. Browsers usually take the ROWS and COLS to specify the visible size of the text area and add scroll bars to allow text input outside this area.

Here's an example of <TEXTAREA>:

```
Enter your address:<BR>
<TEXTAREA NAME="address" ROWS=5 COLS=50>
</TEXTAREA>
```

If text is entered between the opening and closing <TEXTAREA> tags, it appears as default text in the text area when the page is displayed.

A Sample Page

In preparation for the rest of the chapter and to summarize the last section, it's time for a Try It Out. Imagine, if you will, a virtual travel agent, which you ask for information on the various parts of the world you might like to visit and to which you give your modem speed (to determine the graphical content of the brochures). The part of the world you live in determines which branch of the company handles your request.

Try It Out—A Simple Query Form

1. Start with a basic HTML template, like the one below, and save it as `cgi1.html`.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN"
        "http://www.w3.org/TR/REC-html40/loose.dtd">
<HTML>
<HEAD>
<TITLE>A Simple HTML Form Document</TITLE>
</HEAD>
<BODY>
<H1>A demonstration of an HTML form</H1>
<P>

</P>
</BODY>
</HTML>
```

2. Now create the FORM element that fills the rest of the page. It starts with the TEXT and PASSWORD type elements:

```
<FORM ACTION="program" METHOD=GET>

Enter your name: <INPUT NAME=name TYPE=TEXT SIZE=20 MAXLENGTH=40>
and password: <INPUT NAME=passwd TYPE=PASSWORD SIZE=8 MAXLENGTH=8><BR>
<BR><BR>
```

3. For all the different parts of the world the customer might want to visit, we provide a series of checkboxes:

```
Please indicate which areas of the world you would like
to visit:<BR>
<INPUT TYPE="CHECKBOX" NAME=cb VALUE="1">Asia
<INPUT TYPE="CHECKBOX" NAME=cb VALUE="2">Africa
<INPUT TYPE="CHECKBOX" NAME=cb VALUE="3">North America
<INPUT TYPE="CHECKBOX" NAME=cb VALUE="4">South America
<INPUT TYPE="CHECKBOX" NAME=cb VALUE="5">Antarctica
<INPUT TYPE="CHECKBOX" NAME=cb VALUE="6">Europe
<INPUT TYPE="CHECKBOX" NAME=cb VALUE="7">Australasia
<BR><BR>
```

4. To save a little time, you can copy and modify the last section to use radio buttons to find out the customer's home continent:

```
Please indicate in which area of the world you
live:<BR>
<INPUT TYPE="RADIO" NAME=rb VALUE="1">Asia<BR>
```

```
<INPUT TYPE="RADIO" NAME=rb VALUE="2">Africa<BR>
<INPUT TYPE="RADIO" NAME=rb VALUE="3" CHECKED>North America<BR>
<INPUT TYPE="RADIO" NAME=rb VALUE="4">South America<BR>
<INPUT TYPE="RADIO" NAME=rb VALUE="5">Antarctica<BR>
<INPUT TYPE="RADIO" NAME=rb VALUE="6">Europe<BR>
<INPUT TYPE="RADIO" NAME=rb VALUE="7">Australasia<BR>
<BR><BR>
```

5. Now we use the `<SELECT>` tags for finding out the user's modem speed:

```
Please indicate your modem speed:
<SELECT NAME="Speed">
<OPTION value="none">No modem
<OPTION value="vslow">9600 or slower
<OPTION value="slow">14400
<OPTION SELECTED value="OK">28800
<OPTION value="quick">better than 28000
</SELECT>
<BR><BR>
```

6. Use the `TEXTAREA` input type for the customer's address:

```
Please enter your address:<BR>
<TEXTAREA NAME="address" ROWS=5 COLS=50>
</TEXTAREA>
<BR><BR>
```

7. Finally, use the `Submit` and `Reset` buttons and the closing `<FORM>` tag:

```
<INPUT TYPE=RESET VALUE="Clear fields">
<CENTER>
<INPUT TYPE=SUBMIT VALUE="Send Information">
</CENTER>
</FORM>
```

Having typed all this in, if you view the form in your web browser, you should see something like this:

Netscape: A Simple HTML Form Document

File   Edit   View   Go   Communicator   Help

Back   Forward   Reload   Home   Search   Netscape   Print   Security

Bookmarks Location: http://localhost/blp2e/cgi1.html   What's Related

WebMail Contact People Yellow Pages Download Find Sites Channels

A demonstration of an HTML form

Enter your name: [   ]   and password: [   ]

Please indicate which areas of the world you would like to visit:
☐Asia ☐Africa ☐North America ☐South America ☐Antarctica ☐Europe ☐Australasia

Please indicate in which area of the world you live:
◌Asia
◌Africa
◌North America
◌South America
◌Antarctica
◌Europe
◌Australasia

Please indicate your modem speed: [28800   ▭]

Please enter your address:
[   ]

100%

How It Works

The document starts with conventional HTML, just as we have seen before. We then start a form with the <FORM> tag. For now, we're not concerned with the ACTION and METHOD attributes.

We then ask the user to enter a name and password, using the TEXT and PASSWORD type <INPUT> tags. Since we didn't specify any line breaks, the text simply continues on the same line.

We then break the text with the <BR> tag, provide a prompt, and use a CHECKBOX type of <INPUT> tag to allow the user to select multiple entries. We use the same NAME=cb field in each of the <INPUT> tags to tell the browser that these entries should be grouped together.

After another paragraph break, we use a group of RADIO type <INPUT> tags to allow the user to select a single option. Again, we group them together by using the same NAME=rb for each of the buttons in a group. We also use the CHECKED attribute to specify a default.

The modem speed element demonstrates a different type of selection, the SELECT type. The appearance of this element changes significantly depending on the platform the browser is running on. The one shown here is the OSF/Motif style SELECT.

We then use <TEXTAREA> tags to prompt the user for some free-form information, in this case a postal address.

Finally, we provide a reset button for clearing the form to default values and a submit button for sending the information to the server. Note that the <CENTER> tag used here is depreciated in HTML 4, but serves our purpose, since we wish to stick to the basics of HTML rather than get involved in more complex issues such as style sheets and XHTML.

Sending Information to the WWW Server

Now that we've seen how to program a browser to allow the user to enter information and send it back to the server, we need to look at the other side of the system, how the information is transmitted to the server and how it's processed once there.

When the Submit button is selected, the browser encodes all the information in the form and sends it to the server using the HTTP protocol. The server software is then responsible for invoking the appropriate program on the server computer and passing the information to it. The program on the server is often called a CGI (or cgi-bin) program, since the interface between the WWW server software and the program is defined by the CGI specification. There are alternatives to invoking a separate program, commonly called "in-process" handling. In-process handling is the purpose of the ISAPI interface to IIS, the NSAPI interface to Netscape server, and also they way Java servlets execute. At the end of this chapter, we will see our own "in-process" handling, where we see how to embed a Perl interpreter in the Apache Web server, so we can write cgi Perl programs without having to invoke Perl each time a cgi program needs executing.

Information Encoding

We've already met the ENCTYPE attribute of the <FORM> tag that specifies the encoding used to return information to the server. In HTML, this is normally omitted, as the default of application/x-www-form-urlencoded is usually the only encoding required.[AU: Edit OK? If not, please clarify.]

When the data in a form is prepared for transmission to a server, a sequence of transformations is applied to encode the data:

The form field names and values are "escaped." This means that in each field and value pair, spaces are replaced with + characters. Non-alphanumeric characters are replaced by their hexadecimal digits, in the form %HH, where HH is the ASCII hex value of the character. Line breaks are encoded as %0D%0A.

The fields are listed in the order they appear in the form, with the name and value fields joined by an = character. The name/value field pairs are then joined together with & characters. The ; character is often accepted in place of the & character, although this is not part of the standard.

This encoding replaces the form information with a single stream of characters (with no whitespace) holding all the information entered into the form.

Server Program

We now need to look at the programs on the server. These are normally stored in a subdirectory of the main HTTP server called `cgi-bin` (since they are binary programs that implement the CGI interface). The program to be invoked is specified by the `ACTION` attribute of the `<FORM>` tag like this:

```
<FORM ACTION="/cgi-bin/myprogram" METHOD=POST>
```

Security

A word of warning about security is appropriate at this point. When you allow people to send form information to your server, you're allowing them to specify some arbitrary data and pass it to a program they're invoking on your server. This is a potential security problem. Depending on how the CGI program handles the data and if you allow it to invoke other programs, some people may find ways of starting programs you did not wish them to run. Look for the latest version of the WWW security FAQ and be very careful about allowing programs in `cgi-bin` to invoke any other programs.

For starters, here are a few tips on things to avoid:

❑ Don't trust the client. You have no control over which browser they're using and what it allows to be sent to your program.

❑ Don't use the shell or Perl `eval` statement in CGI programs, since it potentially allows an arbitrary string to be used to invoke a command shell.

❑ Be careful invoking other programs, especially using `popen` and `system`, which might allow the client to cause the wrong program to be invoked by clever use of the input data.

Having said that, there are many, many Internet service providers offering WWW services that allow customers to write their own CGI programs, with very few scare stories about unauthorized access. You should be aware of the security problem, but don't let it stop you setting up a Web server!

Writing a Server-Side CGI Program

There are almost no restrictions on what language you can use to write CGI programs. In this chapter, we'll mostly use C, but it's perfectly possible to write CGI programs using shell scripts, Tcl, and possibly the most popular option, Perl.

A server program can receive information in three ways: through environment variables, from the command line, and from the standard input. The method used is controlled by the `METHOD` attribute of the `<FORM>` tag.

Where `METHOD=GET`, the encoded form information is normally passed to the CGI program on the command line. This can cause problems on some systems where there may be a limit on the amount of such information that can be passed. Instead, for all but the simplest requests, many servers pass this information in the environment variable `QUERY_STRING` rather than the command line.

Where `METHOD=POST`, the form information is made available for reading on the standard input.

> **Historically, the GET method was used for forms that "had no side effects," normally in forms implementing a query on the server, and the POST method for more complex forms that could result in changes on the server.**

There's a special way of passing information to a CGI program without using a form at all. This is to append the information to the URL, separated by a ? character. We'll see more of this later.

Environment Variables

Several important pieces of information are passed to the CGI program as **environment variables**. These are the same whatever the method of transferring the form data. The standard variables are

Variable Name	Description
SERVER_SOFTWARE	Information about the server software that received the request and invoked the program.
SERVER_NAME	The server's hostname, or IP address.
GATEWAY_INTERFACE	The revision of the CGI specification that the server implements.
SERVER_PROTOCOL	The revision of the protocol used when the request was received from the client.
SERVER_PORT	The port on which the request was received. This is normally 80 for WWW servers.
REQUEST_METHOD	This is the method used to make the request GET or POST.
PATH_INFO	Some additional information about the path to the CGI program.
PATH_TRANSLATED	The physical path to the CGI program.
SCRIPT_NAME	The name of the script being executed.

Table Continued on Following Page

Variable Name	Description
REMOTE_HOST	The name of the host computer the request came from.
REMOTE_ADDR	The IP address of the host the request came from.
AUTH_TYPE	These are used where the server supports user authentication.
REMOTE_USER	The user name passed by the client.
REMOTE_IDENT	The remote user name. This is not reliable and is rarely used.
CONTENT_TYPE	The content type of the information being transferred. This is usually application/x-wwwform-urlencoded.
CONTENT_LENGTH	The number of bytes of data being passed to the program. This should always be used in preference to looking for a null terminator or end of file when reading input where METHOD=POST.

The `CONTENT_TYPE` and `CONTENT_LENGTH` may not be set when `METHOD=GET`.

In addition, some servers may add further variables. These should start with `HTTP_` to avoid clashes with names that may be added by later versions of the HTTP protocol and CGI specification.

The Apache server, and many others, also provide

`QUERY_STRING`	This contains the information passed to the CGI program for `METHOD=GET`, or where the information was passed as part of the URL.

Putting It All Together

We now know almost enough to write our first CGI program, which will show the values of the environment variables when the client request was made. There are libraries that can do this for you, but it's instructive to show how it would be done by hand, since that will help you to understand the principles involved.

We don't yet quite understand how, after processing the information from the client, our CGI program can return a result to it. In actual fact the standard output of the cgi program is fed back to the browser, but before we can send HTML to the client, we must send some HTTP header information. For now, it's enough to know that writing `Content-type: text/plain` and a blank line followed by the text you wish to display is sufficient for the browser to display simple text on the screen.

We'll use a shell program for this example, because that's the easiest method for the processing we need at this stage.

Try It Out—Our First CGI Program

1. We're going to write `cgi1.sh`, so we'll comment that into the header:

```
#!/bin/sh
# cgi1.sh
# A simple script for showing environment variable information passed to a CGI
program.
```

2. We start the output to the browser with the two lines that we were given above:

```
echo Content-type: text/plain
echo
```

3. Next, we want to display the arguments:

```
echo argv is "$*".
echo
```

4. Then we show the environment variables under which the CGI request was made, the meat of this first program:

```
echo SERVER_SOFTWARE=$SERVER_SOFTWARE
echo SERVER_NAME=$SERVER_NAME
echo GATEWAY_INTERFACE=$GATEWAY_INTERFACE
echo SERVER_PROTOCOL=$SERVER_PROTOCOL
echo SERVER_PORT=$SERVER_PORT
```
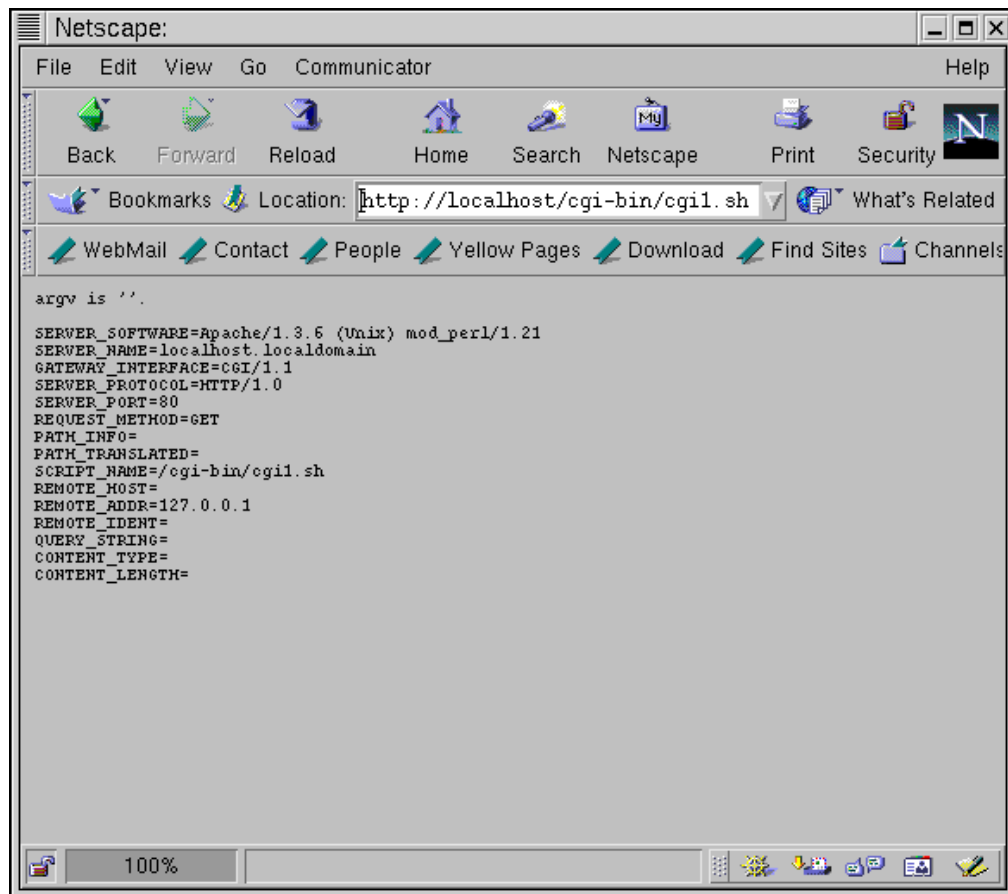
```
echo REQUEST_METHOD=$REQUEST_METHOD
echo PATH_INFO=$PATH_INFO
echo PATH_TRANSLATED=$PATH_TRANSLATED
echo SCRIPT_NAME=$SCRIPT_NAME
echo REMOTE_HOST=$REMOTE_HOST
echo REMOTE_ADDR=$REMOTE_ADDR
echo REMOTE_IDENT=$REMOTE_IDENT
echo QUERY_STRING=$QUERY_STRING
echo CONTENT_TYPE=$CONTENT_TYPE
echo CONTENT_LENGTH=$CONTENT_LENGTH
    exit 0
```

5. We need to change our HTML example page (let's call it `cgi2.html`) so that the request references our shell script. To do this, we just need to change the `ACTION` attribute right after the HTML header:

```
<FORM ACTION="/cgi-bin/cgi1.sh" METHOD=POST>
```

6. Lastly, since we must access these scripts via our WWW server in order for the CGI program to be invoked correctly, we must now copy the files into the appropriate directories. For the `cgi1.sh` program, this should be in the `cgi-bin` subdirectory of the server setup; for the HTML, it's with the other HTML files. We must also ensure the `cgi1.sh` program is executable.

We can now access our form via the server. When we press the `Submit` button on the form, this is what we see:

```
argv is ''.

SERVER_SOFTWARE=Apache/1.3.6 (Unix) mod_perl/1.21
SERVER_NAME=localhost.localdomain
GATEWAY_INTERFACE=CGI/1.1
SERVER_PROTOCOL=HTTP/1.0
SERVER_PORT=80
REQUEST_METHOD=GET
PATH_INFO=
PATH_TRANSLATED=
SCRIPT_NAME=/cgi-bin/cgi1.sh
REMOTE_HOST=
REMOTE_ADDR=127.0.0.1
REMOTE_IDENT=
QUERY_STRING=
CONTENT_TYPE=
CONTENT_LENGTH=
```

How It Works

When the user presses the Submit button, the form data is sent to the server, which then invokes the cgi1.sh program. cgi1.sh uses the standard output to return data to the browser for it to display. Notice that ACTION has what appears to be an absolute path in it, but the server automatically prepends the location of the server files before invoking the program.

Note that we can't yet see any of the actual data from the elements on the form yet, just the environment in which the program was invoked.

We can now slightly extend the CGI script to try and access the data being passed to the program. Since we used METHOD=POST, this data will appear on the standard input.
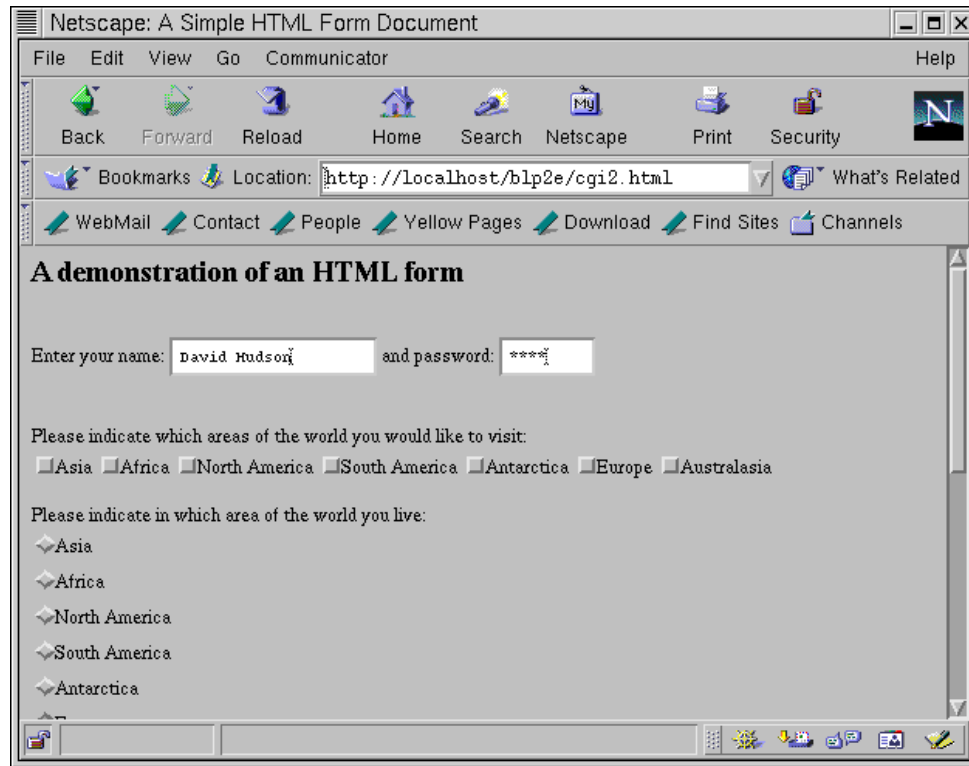
Try It Out—Reading the Data

Save a copy of cgi1.sh as cgi2.sh. We'll now make a very simple change to our script to read and return the input. Add the following code to the end of the script:

```
echo The data was:
read x
while [ "$x" != "" ]; do
    echo $x
    read x
```
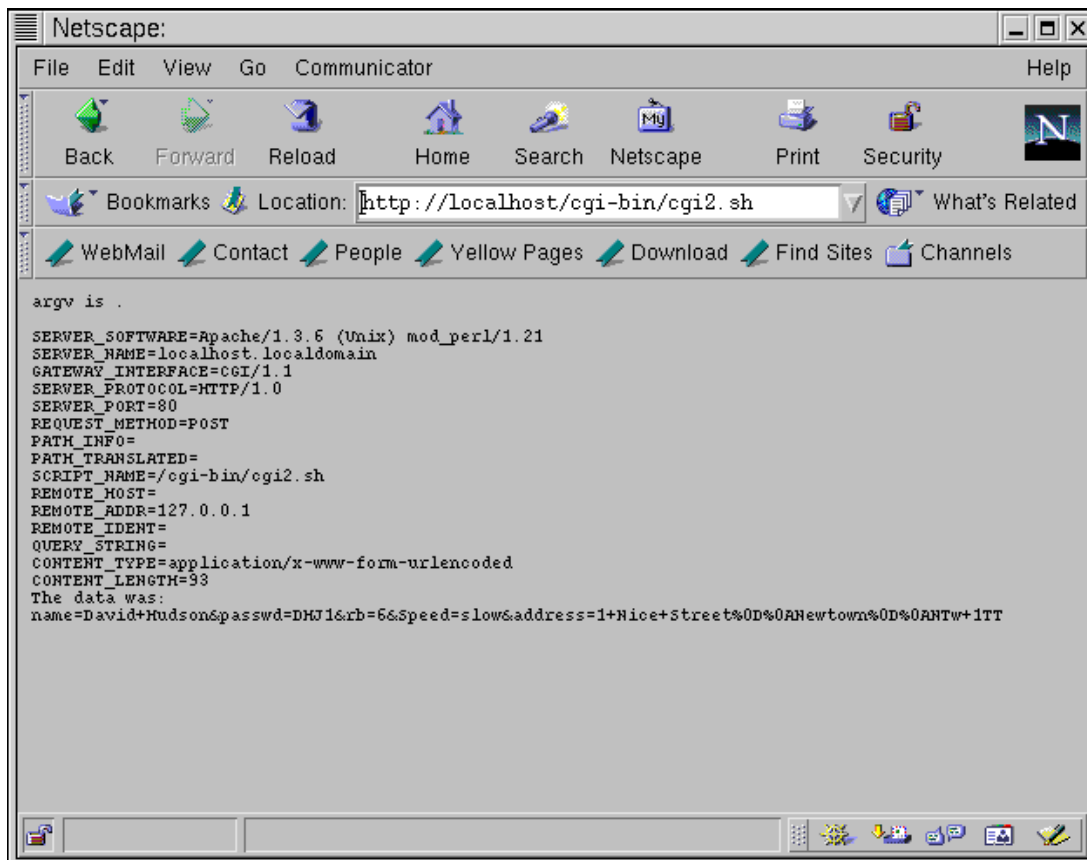
```
done
```

We also change the `ACTION` attribute of our HTML form (call it `cgi2.html`) to reference the new CGI program, `cgi2.sh`.

When we submit, the screen looks like this:

The output we get is

```
Netscape:
File   Edit   View   Go   Communicator                                    Help

  Back   Forward   Reload   Home   Search   Netscape   Print   Security

  Bookmarks   Location: http://localhost/cgi-bin/cgi2.sh   What's Related

  WebMail   Contact   People   Yellow Pages   Download   Find Sites   Channels

argv is .

SERVER_SOFTWARE=Apache/1.3.6 (Unix) mod_perl/1.21
SERVER_NAME=localhost.localdomain
GATEWAY_INTERFACE=CGI/1.1
SERVER_PROTOCOL=HTTP/1.0
SERVER_PORT=80
REQUEST_METHOD=POST
PATH_INFO=
PATH_TRANSLATED=
SCRIPT_NAME=/cgi-bin/cgi2.sh
REMOTE_HOST=
REMOTE_ADDR=127.0.0.1
REMOTE_IDENT=
QUERY_STRING=
CONTENT_TYPE=application/x-www-form-urlencoded
CONTENT_LENGTH=93
The data was:
name=David+Hudson&passwd=DHJ1&rb=6&speed=slow&address=1+Nice+Street%0D%0ANewtown%0D%0ANTw+1TT
```

How It Works

We now read the entire standard input stream and return it to the browser. Remember that simply reading the standard input until it stops isn't an officially supported method of accessing the data. It does, however, serve our purposes here as an example.

> Real programs should always use the CONTENT_LENGTH environment variable to determine the amount of data to process. This is the way the CGI specification states the length must be determined, perhaps for environments that don't have an easy way of detecting end-of-file on the standard input.

There are two important things to notice in the information the CGI program received:

❑ The data has been encoded so that it forms a continuous string with no whitespace.

❑ There's no information about the checkbox part (where would you like to visit) of the form. This is because no boxes were checked, so the browser was free to omit that part of the data. This is an advantage in many ways, because it reduces the amount of data being transferred across the network, but it's important that form decoding software take account of the possibility of empty fields being omitted. Browsers may arbitrarily suppress unused fields.

To complete our look at HTML forms, change the METHOD type to GET in our HTML sample, now cgi3.html. If we now invoke the form, the information is passed to cgi2.sh in a slightly different way, so we see
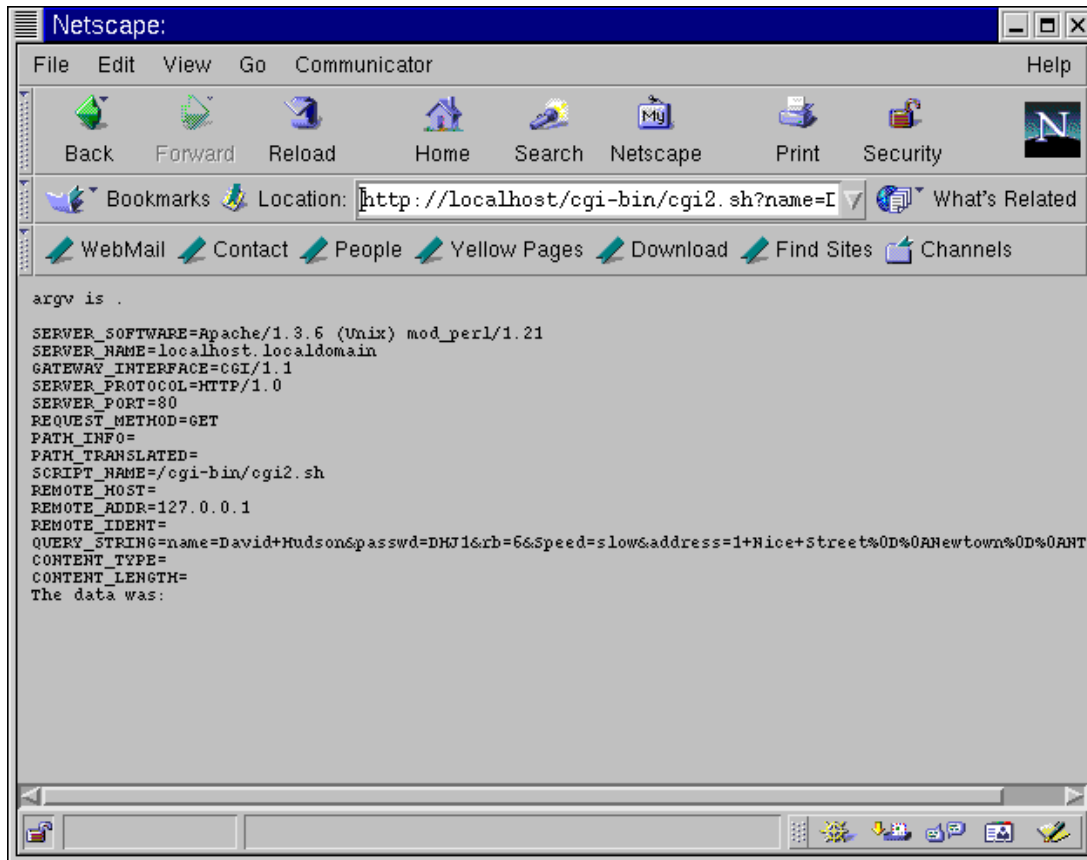
The information is encoded in the same way, but the CONTENT_LENGTH parameter is no longer supplied. It's now made available in the variable QUERY_STRING, not on the standard input. METHOD=GET is easier to process, but METHOD=POST is to be preferred for all but the simplest forms.
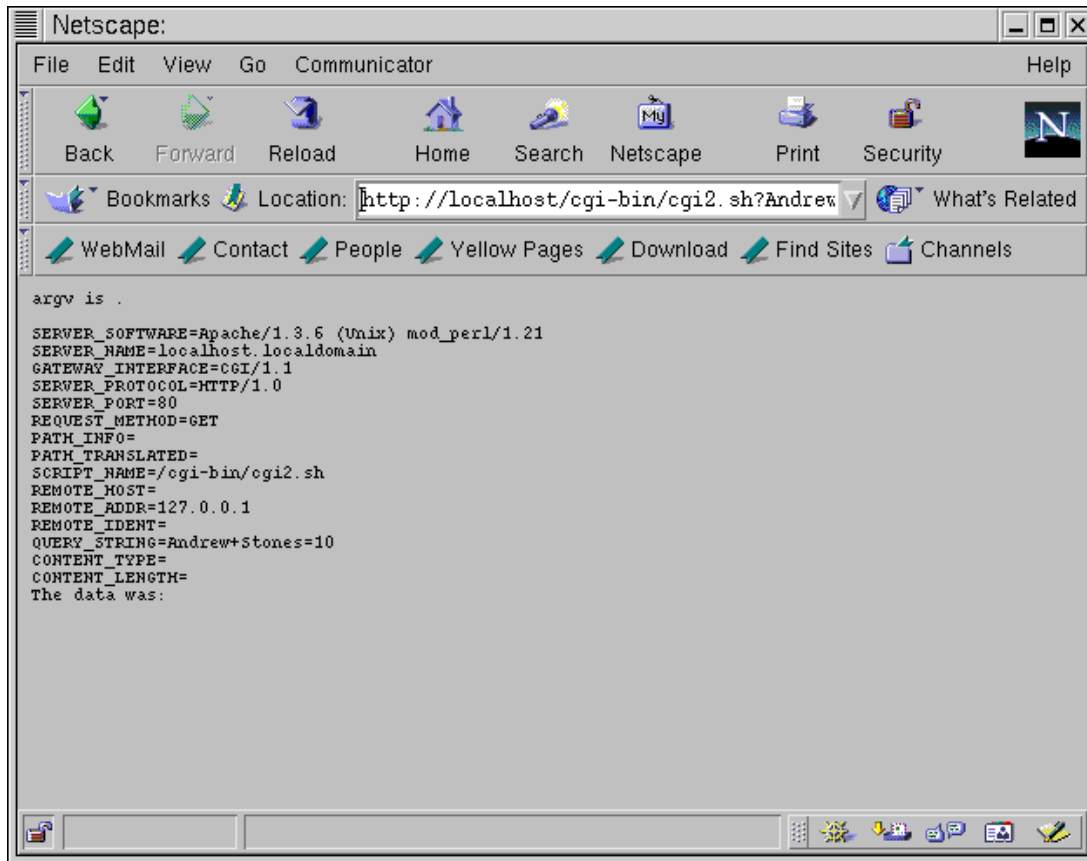
CGI Programs Using Extended URLs

Before we move on to decoding the data passed to the CGI program, we'll look at the remaining way of sending data to a CGI program: appending it to a URL. The server program separates the main part of the URL (the bit before the ?) and uses it as the program to invoke. It passes the remaining part of the URL as an argument to the program, as though it had come from a form.

Invoke `cgi2.sh` again, this time using the URL
`http://localhost/cgi-bin/cgi2.sh?Andrew+Stones=10`.

Now the CGI program sees

```
Netscape:
File   Edit   View   Go   Communicator                                    Help

  Back   Forward   Reload    Home    Search   Netscape    Print   Security

  Bookmarks   Location: http://localhost/cgi-bin/cgi2.sh?Andrew    What's Related

  WebMail   Contact   People   Yellow Pages   Download   Find Sites   Channels

argv is .

SERVER_SOFTWARE=Apache/1.3.6 (Unix) mod_perl/1.21
SERVER_NAME=localhost.localdomain
GATEWAY_INTERFACE=CGI/1.1
SERVER_PROTOCOL=HTTP/1.0
SERVER_PORT=80
REQUEST_METHOD=GET
PATH_INFO=
PATH_TRANSLATED=
SCRIPT_NAME=/cgi-bin/cgi2.sh
REMOTE_HOST=
REMOTE_ADDR=127.0.0.1
REMOTE_IDENT=
QUERY_STRING=Andrew+Stones=10
CONTENT_TYPE=
CONTENT_LENGTH=
The data was:
```

This method of passing information to CGI programs is often used where the server has generated the HTML that the client is viewing and wishes to encode some extra data inside it. An example would be Web pages that require user authentication. Since each request for a page is treated separately by the server, in theory, the user would have to enter their password every time a page was accessed. Instead, the user is given a special URL to access which encodes the user name and password or an authenticated pass key.

Suppose a user, Jenny Stones, with a password `secret` (obviously not a very worldly-wise user . . .), wishes to access the weather page of an online newspaper that asks her to register before she can have access. Rather than presenting her with a form that asks for her name and password, the page could ask her to enter a special URL of the format

`http://www.paper.com/cgi-bin/access?user=Jenny+Stones&passwd=secret&page=weather`

Although this looks a bit cumbersome, it can be stored as a browser bookmark or shortcut and then accessed with a single selection. There is then no need to re-enter the name and password each time the page is accessed. It also saves Jenny Stones having to remember or write down the user name and password for all the online services to which she has access.

The drawback is that the password is stored as plain text, so this isn't a secure way of storing names and passwords or passing them across the network. In practice, a feature called "cookies" is often used to store a user's access rights to a site, which saves them having to remember passwords. The drawback is that the cookie is stored by the browser on the local machine, so if you register at home and get a cookie stored on your home PC, it's not going to help you access the site if you browse it from work on a different PC. We will not be looking at cookies further in this chapter.

Decoding the Form Data

We've seen how to access the data returned by a form, so now we must decode it into a more usable format. This can be quite involved. However, when you've solved the problem once, you can reuse the decoding software again and again. There are already programs that decode form data available on the Internet in a range of languages including Tcl, Perl, C, and C++. Some of the examples are in the public domain; others have slightly more restrictive licenses.

Don't worry; we're going to decode the data using our own program. To do this, we need to perform the opposite of the encode and in the reverse order. Since we already know the encoding rules, it's just a case of writing the software to implement them. Let's call our first attempt, decode1.c. To keep things simple, we will impose some fixed limits on the number and sizes or parameters we can process.

Try It Out—A CGI Decode Program in C

1. After adding in the standard headers and a couple of constants, we define a data structure, name_value_st, which can hold one field name and one value of that field. Then we declare the prototypes of the functions that we're going to use later.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#define FIELD_LEN 250 /* how long each name or value can be */
#define NV_PAIRS 200  /* how many name=value pairs we can process */

typedef struct name_value_st {
    char name[FIELD_LEN + 1];
    char value[FIELD_LEN + 1];
} name_value;

name_value name_val_pairs[NV_PAIRS];

static int get_input(void);
static void send_error(char *error_text);
static void load_nv_pair(char *tmp_buffer, int nv_entry_number_to_load);
static char x2c(char *what);
static void unescape_url(char *url);
```

2. On its own, the main function doesn't do a great deal. It calls get_input to set the ball rolling by loading the name_val_pairs structure and sends the decoded information back to the client, starting with the Content-type and a blank line:

```
int main(int argc, char *argv[])
{
```

```
    int nv_entry_number = 0;

    if (!get_input()) {
        exit(EXIT_FAILURE);
    }

    printf("Content-type: text/plain\r\n");
    printf("\r\n");

    printf("Information decoded was:-\r\n\r\n");
    while(name_val_pairs[nv_entry_number].name[0] != '\0') {
        printf("Name=%s, Value=%s\r\n",
                name_val_pairs[nv_entry_number].name,
                name_val_pairs[nv_entry_number].value);
        nv_entry_number++;
    }
    printf("\r\n");
    exit(EXIT_SUCCESS);
}
```

3. Next, we define a function, `get_input`, that does just what it says. We need to discover whether the request is of type POST or GET and then copy the input to a data block called `ip_data`:

```
static int get_input(void)
{
    int nv_entry_number = 0;
    int got_data = 0;
    char *ip_data = 0;
    int ip_length = 0;
    char tmp_buffer[(FIELD_LEN * 2) + 2];
    int tmp_offset = 0;
    char *tmp_char_ptr;
    int chars_processed = 0;

    tmp_char_ptr = getenv("REQUEST_METHOD");
    if (tmp_char_ptr) {
      if (strcmp(tmp_char_ptr, "POST") == 0) {
                tmp_char_ptr = getenv("CONTENT_LENGTH");
                if (tmp_char_ptr) {
                    ip_length = atoi(tmp_char_ptr);
                    ip_data = malloc(ip_length + 1); /* allow for NULL character */
                    if (fread(ip_data, 1, ip_length, stdin) != ip_length) {
              send_error("Bad read from stdin");
              return(0);
            }
                    ip_data[ip_length] = '\0';
                    got_data = 1;
                }
      }

    }

    tmp_char_ptr = getenv("REQUEST_METHOD");
    if (tmp_char_ptr) {
      if (strcmp(getenv("REQUEST_METHOD"), "GET") == 0) {
        tmp_char_ptr = getenv("QUERY_STRING");
                if (tmp_char_ptr) {
                    ip_length = strlen(tmp_char_ptr);
                    ip_data = malloc(ip_length + 1); /* allow for NULL character */
                    strcpy(ip_data, getenv("QUERY_STRING"));
                    ip_data[ip_length] = '\0';
                    got_data = 1;
                }
      }
    }
```

```
    if (!got_data) {
        send_error("No data received");
        return(0);
    }

    if (ip_length <= 0) {
        send_error("Input length not > 0");
        return(0);
    }
```

4. By getting to this point, we know that we have the encoded data stored in `ip_data`. Next, we need to extract the NAME and VALUE components of the HTML submission and decode them individually. We know that & symbols represent the breaks between NAME=VALUE pairs, so we use them to split them up and pass the result to the function `load_nv_pair` to decode them one at a time.

```
    memset(name_val_pairs, '\0', sizeof(name_val_pairs));
    tmp_char_ptr = ip_data;
    while (chars_processed <= ip_length && nv_entry_number < NV_PAIRS) {

        /* copy a single name=value pair to a tmp buffer */
        tmp_offset = 0;
        while (*tmp_char_ptr &&
                *tmp_char_ptr != '&' &&
                tmp_offset < FIELD_LEN) {
            tmp_buffer[tmp_offset] = *tmp_char_ptr;
            tmp_offset++;
            tmp_char_ptr++;
            chars_processed++;
        }
        tmp_buffer[tmp_offset] = '\0';

        /* decode and load the pair */
        load_nv_pair(tmp_buffer, nv_entry_number);

        /* move on to the next name=value pair */
        tmp_char_ptr++;
        nv_entry_number++;
    }
    return(1);
}
```

5. You'll notice that we pass all errors to a function called `send_error`, which sends an error string back to the client. It's not too difficult, so we'll define it next:

```
static void send_error(char *error_text)
{
    printf("Content-type: text/plain\r\n");
    printf("\r\n");
    printf("Woops:- %s\r\n", error_text);
}
```

6. The actual decoding is handled by the function `load_nv_pair`, which we called in `get_input`. It breaks down the NAME=VALUE pairs further and places the NAME and VALUE into separate sections of our data structure and then calls another function, `unescape_url`, to continue the decoding.

```
/* Assumes name_val_pairs array is currently full of NULL characters */
```

```
static void load_nv_pair(char *tmp_buffer, int nv_entry)
{
    int chars_processed = 0;
    char *src_char_ptr;
    char *dest_char_ptr;

    /* get the part before the '=' sign */
    src_char_ptr = tmp_buffer;
    dest_char_ptr = name_val_pairs[nv_entry].name;
    while(*src_char_ptr &&
          *src_char_ptr != '=' &&
          chars_processed < FIELD_LEN) {

        /* Change a '+' to a ' ' */
        if (*src_char_ptr == '+') *dest_char_ptr = ' ';
        else *dest_char_ptr = *src_char_ptr;
        dest_char_ptr++;
        src_char_ptr++;
        chars_processed++;
    }

    /* skip the '=' character */
    if (*src_char_ptr == '=') {

        /* get the part after the '=' sign */
        src_char_ptr++;
        dest_char_ptr = name_val_pairs[nv_entry].value;
        chars_processed = 0;
        while(*src_char_ptr &&
              *src_char_ptr != '=' &&
              chars_processed < FIELD_LEN) {

            /* Change a '+' to a ' ' */
            if (*src_char_ptr == '+') *dest_char_ptr = ' ';
            else *dest_char_ptr = *src_char_ptr;
            dest_char_ptr++;
            src_char_ptr++;
            chars_processed++;
        }
    }

    /* Now need to decode %XX characters from the two fields */
    unescape_url(name_val_pairs[nv_entry].name);
    unescape_url(name_val_pairs[nv_entry].value);
}
```

7. The function `unescape_url` then decodes the special (nonalphanumeric) characters in the text from their `%HH` representation by running through the array and calling the function `x2c`:

```
/* this routine borrowed from the examples that come with the NCSA server */
static void unescape_url(char *url)
{
    int x,y;
    for (x=0,y=0; url[y]; ++x,++y) {
        if ((url[x] = url[y]) == '%') {
            url[x] = x2c(&url[y+1]);
            y += 2;
        }
    }
    url[x] = '\0';
}

/* this routine borrowed from the examples that come with the NCSA server */
static char x2c(char *what)
{
    register char digit;
    digit = (what[0] >= 'A' ? ((what[0] & 0xdf) - 'A')+10 : (what[0] - '0'));
```

```
    digit *= 16;
    digit += (what[1] >= 'A' ? ((what[1] & 0xdf) - 'A')+10 : (what[1] - '0'));
    return(digit);
}
```
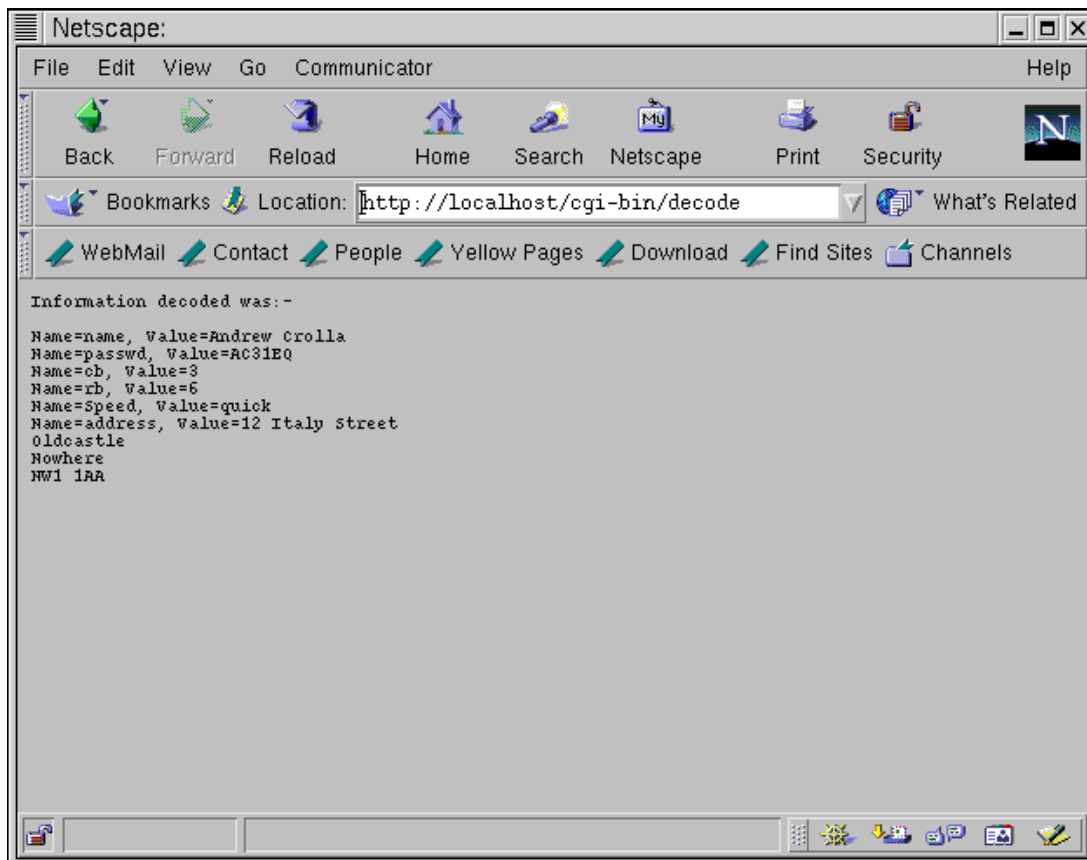
We compile this program, copy it into the `cgi-bin` directory, and change our HTML form so that the `FORM ACTION` is now `/cgi-bin/decode1` and the `METHOD=POST`. This is now `cgi5.html`.

Here's the form information that we submit:

After submitting the form, we see

As you can see, the new line has been preserved.

How It Works

That was quite a large amount of C code, so let's segment it and look at how each piece works. We start by declaring a structure to hold each of the name/value pairs and then declare a global array to store them in. We impose some arbitrary limits on the sizes, because it makes the example rather easier to understand. A real-world program should perhaps use a linked list of nodes, each containing a pointer to a `malloced` area with the name and value in it, thus removing the arbitrary size restrictions.

We then call the routine `get_input` to load the `name_val_pairs` array. This takes several steps. First we check which method was used to post the data and transfer the data into some space pointed to by `ip_data`. Now that the data is in a single place, we can use the same code to process both `GET` and `POST` methods.

The first stage is to break the single string so we can work on a single `name=value` string at a time. Remember that these are separated by `&` characters. We then call `load_nv_pair` once per name/value pair, which splits the pair by looking for the = character. It also replaces + characters with spaces.

Once the name and value strings have been separated, we can remove the special coding where characters are stored as a pair of hex bytes, as `%XX`. To do this, we use two routines, `x2c` and `unescape_url`.

The `x2c` and `unescape_url` routines appear (usually verbatim) in almost all C and C++ form decoding software. As far as we can tell, they came originally from the examples provided with the NCSA WWW server. We've never seen a copyright message on them or any other comment claiming ownership. Since they are so widely distributed and solve the problem, we use them again here, trusting that the attribution to the NCSA server software is correct.

Now that we've processed the input, we simply print out the pairs. Notice that we precede our output with a `Content-type` line and blank line as before and that all lines are terminated with a carriage return and a new line.

Returning HTML to the Client

Until now, we've just been sending back plain text to the client from our CGI program. While this works, it doesn't look very attractive.

We'll now look at generating HTML, so that the CGI program's output looks more like a normal Web page. In fact, if we're careful, there's no reason why a user should be able to tell the difference between a static HTML page returned from the server and one dynamically generated by a CGI program.

Because the CGI program has full control over data sent back to the client, it can send many different types of data, not just text. It does this using the `Content-type:` control line that we saw earlier, which controls the MIME type and subtype that the client browser expects to receive. Rather than using the `text/plain` MIME type and subtype information that we used for text, we can use a `text/html` type. This allows us to send HTML to the client, which it will then decode and display just like a normal HTML page.

We could simply write code like this:

```
printf("<H1>A heading</H1>\r\n");
```

but it wouldn't be very elegant. A much better approach is to write some utility functions to cater for some of the lower level functionality that we need to send HTML tags to the client.

Here are some examples of lower level routines. The function `html_content` sends a string to the client to let it know that we'll be sending HTML to it:

```
static void html_content(void)
{
    printf("Content-type: text/html\r\n\r\n");
}
```

`html_start` then starts the HTML header section, sends a title string, and starts the HTML body section:

```
static void html_start(const char *title)
{
    printf("<HTML>\r\n");
    printf("<HEAD>\r\n");
    printf("<TITLE>%s</TITLE>\r\n", title);
    printf("</HEAD>\r\n");
    printf("<BODY>\r\n");
}
```

We can use two more functions, `html_header` and `html_text`, simply as convenient utility routines to print header text of a specified level and ordinary body text, respectively.

```
static void html_header(int level, const char *header_text)
{
    if (level < 1 || level > 6) level = 6; /* force the level to a valid number */
    if (!header_text) return;
    printf("<H%d>%s</H%d>\r\n", level, header_text, level);
}

static void html_text(const char *text)
{
    printf("%s\r\n", text);
}
```

Finally, `html_end` terminates the HTML.

```
static void html_end(void)
{
    printf("</BODY>\r\n");
    printf("</HTML>\r\n");
}
```

Notice that we terminate all lines with both a carriage return and a newline character, rather than the more usual UNIX convention of terminating lines with a single newline character. A minimal CGI program could then consist of these routines, something as simple as

```
html_content();
html_start("Example");
html_header(1, "Hello World");
html_text("Pleased to be here!");
html_end();
```

We can modify our earlier program to return HTML information rather than plain text by adding the five functions defined above to the end of our `decode1.c` program and making a few other changes. Call it `decode2.c`.

Try It Out—Returning HTML to the Client

1. Add the following function prototypes for the functions that we've just placed at the end. They can go after the ones that we've already defined:

```
static int get_input(void);
static void send_error(char *error_text);
static void load_nv_pair(char *tmp_buffer, int nv_entry_number_to_load);
static char x2c(char *what);
static void unescape_url(char *url);
static void html_content(void);
static void html_start(const char *title);
static void html_end(void);
static void html_header(int level, const char *header_text);
static void html_text(const char *text);
```

2. Most of the changes happen in the `main` function, since it deals with almost all the output. Basically, it's a matter of defining a buffer for the text to be stored in before it gets displayed in HTML format and then changing the output commands to make use of our new functions:

```
int main(int argc, char *argv[])
{
```

```
    char tmp_buffer[4096];
    int nv_entry_number = 0;

    if (!get_input()) {
            exit(EXIT_FAILURE);
    }

    html_content();
    html_start("Form decoding test");

    html_header(2, "Information decoded");
    while(name_val_pairs[nv_entry_number].name[0] != '\0') {
        sprintf(tmp_buffer, "Name=%s, Value=%s",
                name_val_pairs[nv_entry_number].name,
                name_val_pairs[nv_entry_number].value);
        html_text(tmp_buffer);
        html_text("<BR><BR>");
        nv_entry_number++;
    }
    html_end();
    exit(EXIT_SUCCESS);
}
```
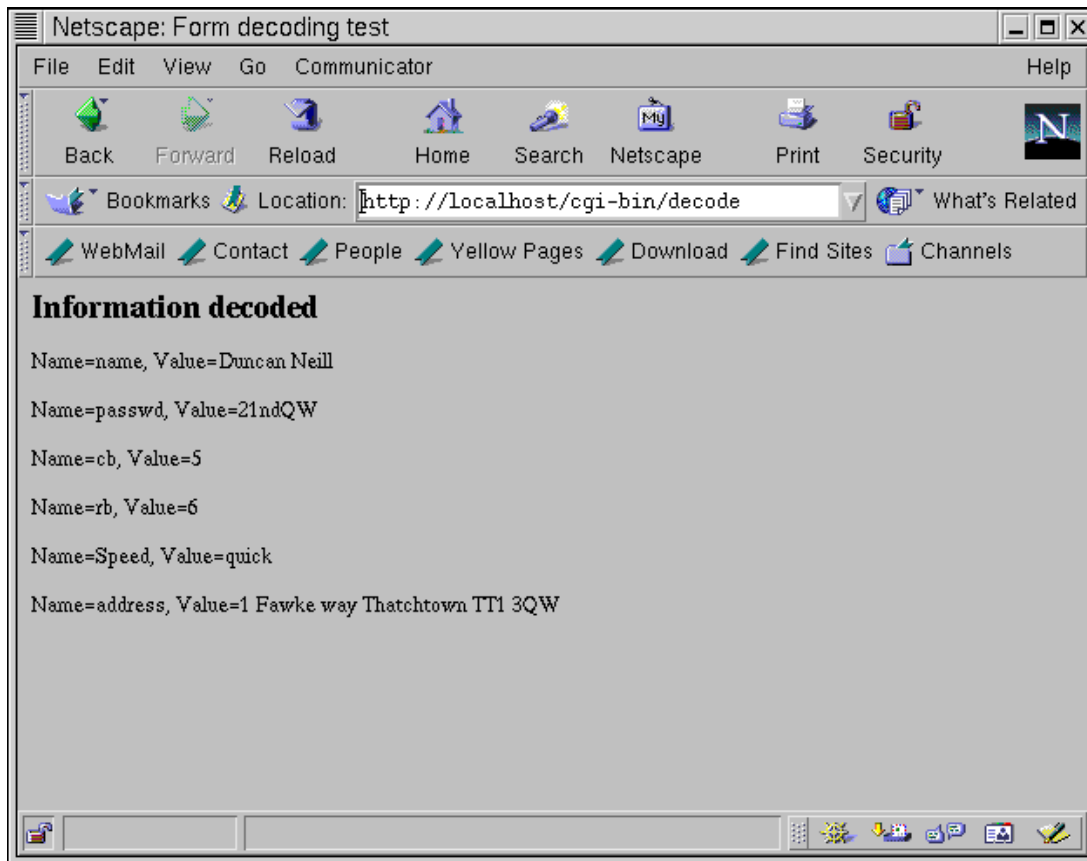
3. The only other function that we need to change is `send_error`, the output from which must now also be in HTML format:

```
static void send_error(char *error_text)
{
    html_content();
    html_start("Woops");
    html_text(error_text);
    html_end();
}
```

After completing these changes (and changing the reference in the Web page, of course), we get the following output. Your hard-earned HTML expertise gives a slightly more professional look:

Netscape: Form decoding test

File Edit View Go Communicator   Help

Back Forward Reload Home Search Netscape Print Security

Bookmarks Location: `http://localhost/cgi-bin/decode` What's Related

WebMail Contact People Yellow Pages Download Find Sites Channels

Information decoded

Name=name, Value=Duncan Neill

Name=passwd, Value=21ndQW

Name=cb, Value=5

Name=rb, Value=6

Name=Speed, Value=quick

Name=address, Value=1 Fawke way Thatchtown TT1 3QW

Tips and Tricks

Here are a few hints and tips that you may find useful in writing your own CGI programs.

Making Sure the CGI Program Exits

During the execution of your CGI program or when it's sending back its response to the client, the client could exit or the network could fail. If this happens, your program may get into a state where it's waiting to send output on the standard output that it can never complete. To avoid this, it's wise to ensure that there's a fail-safe way for the program to exit.

A good way of doing this is to trap the SIGALRM signal when the program starts; then, before doing any output, call `alarm` to generate a signal in, say, 30 seconds. Normally, a CGI program will run for a very short period of time, often less than a second. Thus, if the signal handling routine is ever called, we can assume that a serious problem has occurred. Since this could well be that output has blocked, it's probably best to just exit the CGI program, rather than try to send error information back to the client. Alternatively, if the server system allows, we could use nonblocking I/O to avoid stalling.

Redirecting the Client

A common trick is for the CGI program to redirect the request to a different Web page, depending on the information that was passed. It does this by sending status messages back to the client. A range of status codes are defined, but only a very few are useful in CGI programs. Here we'll consider only status 302, which tells the client the page has temporarily moved.

Suppose we have a form that asks the user if they like jazz or classical music and, depending on the answer to the question, we wish to direct the user to different pages. If we generate the resulting page on the fly in the CGI program, we're now stuck with coding pages in programs, rather than using HTML files.

A much simpler solution is to ask the client to move to a different page. Suppose we decode the information in a CGI program and end up with a variable `result` that contains either `jazz` or `classical`. We can direct the client to the appropriate page like this:

```
printf("Status: 302\r\n");
if (strcmp(result, "jazz") == 0) {
    printf("Location: /jazz.html\r\n");
}
else {
    printf("Location: /class.html\r\n");
}
exit(EXIT_SUCCESS);
```

Notice that this is the first information we send back, so we don't start the response with a `Content-type:` line. When the client sees the `Status: 302`, it believes the URL it requested has moved to a temporary new home. It looks for a following `Location:` line to tell it what that location is. When the `Location:` line arrives the client simply requests the specified URL.

Dynamic Graphics

Many WWW pages have an access or hits counter. Since, in general, these appear on pages consisting of normal HTML, how does the up-to-date graphical access counter get there?

Often, the answer is to use an `<IMG>` tag, but to specify the SRC (usually a `.gif` or `.jpg` file) as a `cgi-bin` program. When the client accesses the base HTML page, it finds the `<IMG>` tag and tries to fetch the image specified. By making the `cgi-bin` program one that can generate images on the fly, a dynamic image can be returned.

A full discussion of this is beyond the scope of this chapter, but if you wish to investigate this further, you should search for a graphics library called `gd`, which lets you generate graphics from code.

Hiding Context Information

Often, a server would like to pass information between one form and the next. This might be a user's name, a customer code, or some other information.

Since each form is processed completely separately from any other form, we need to "hide" information inside the form. We can do this using the hidden fields that we met earlier in the chapter, or by appending information to the URL after a `?` character.

By declaring one or more hidden fields and giving them a default value when a CGI-generated HTML form is sent to the client, we can hide the information in the form in a way not immediately visible to the user. When the user has finished with the form and sends it back to the server, the CGI program can use the values of the hidden fields to extract the context information.

Notice that the fields are hidden, not secret. The user can still view the information by viewing the source to the HTML. Hidden fields are not a good place to hide secret passwords!

Data appended to a URL after a ? character is treated as though the user used a form with METHOD=GET. The CGI program can then decode the additional information and use it as context information. We'll demonstrate this in the following example application.

An Application

This is what you've been waiting for; we're now going to see how we can adapt our database application to be accessible over the Web. In this example, we're only going to implement read access to the database. This allows us to demonstrate the principle of accessing the database through a CGI script, but keeps the application reasonably simple. You could extend it to allow the database to be updated if you wished.

We'll start with the application that we developed in Chapter 7, using the dbm database, but replacing the app_ui.c file with a new front end, app_html.c. We'll also use the HTML functions that we wrote earlier in the chapter.

Try It Out—An HTML Database Interface

1. This is the full listing of the program app_html.c. The main function starts the database and then calls one of the other two functions, depending on whether a CD catalog number is specified on the URL command line:

```
#include <stdio.h>
#include <string.h>

#include "cd_data.h"
#include "html.h"

const char *title = "HTML CD Database";
const char *req_one_entry = "CAT";
void process_no_entry(void);

void process_cat(const char *option, const char *title);
```

```
void space_to_plus(char *str);

int main(int argc, char *argv[])
{
    if (!database_initialise(0)) {
        html_content();
        html_start(title);
        html_text("Sorry, database could not initialize");
        html_text("<BR><BR>");
        html_text("Please mail <AHREF=\"mailto:webmaster@anyhost.com\">
                                webmaster</A> for assistance");
        html_end();
        exit(EXIT_SUCCESS);
    }
    if (!get_input()) {
        database_close();
        html_content();
```

```
        html_start(title);
        html_text("Sorry, METHOD not POST or GET");
        html_text("Please mail <AHREF=\"mailto:webmaster@anyhost.com\">
                            webmaster</A> for assistance");
        html_end();
        exit(EXIT_SUCCESS);
    }
    html_content();
    html_start(title);
```

2. If there was a legitimate query string, display the tracks for that CD. Otherwise, list all the CDs in the database:

```
    if (strcmp(name_val_pairs[0].name, req_one_entry) == 0) {
        process_cat(name_val_pairs[0].name, name_val_pairs[0].value);
    }
    else {
        process_no_entry();
    }
    html_end();
    database_close();
    exit(EXIT_SUCCESS);
}
```

3. If we get here, no entry has been selected. Show a standard screen, listing all the CDs in the database:

```
void process_no_entry(void)
{
    char tmp_buffer[120];
    char tmp2_buffer[120];
    cdc_entry item_found;
    int first_call = 1;
    int items_found = 1;

    html_header(1, "CD database listing");
    html_text("Select a title to show the tracks");
    html_text("<BR><BR><HR><BR><BR>");

    while(items_found) {
        item_found = search_cdc_entry("", &first_call);
        if (item_found.catalog[0] == '\0') {
            items_found = 0;
        }

        else {
            sprintf(tmp_buffer, "Catalog: %s", item_found.catalog);
            html_text(tmp_buffer);
            html_text("<BR><BR>");
            strcpy(tmp2_buffer, item_found.catalog);
            space_to_plus(tmp2_buffer);
            sprintf(tmp_buffer, "Title: <A HREF=\"/cgi-bin/cddb/cdhtml?CAT=%s\">
                                %s</A>", tmp2_buffer, item_found.title);
            html_text(tmp_buffer);
            html_text("<BR><BR>");
            sprintf(tmp_buffer, "Type: %s", item_found.type);
            html_text(tmp_buffer);
            html_text("<BR><BR>");
            sprintf(tmp_buffer, "Artist: %s", item_found.artist);
            html_text(tmp_buffer);
            html_text("<BR><BR><HR><BR><BR>");
        }
    }
}
```

4. If we get here, the additional parameters have already been parsed:

```
void process_cat(const char *name_type, const char *cat_title)
{
    char tmp_buffer[120];
    cdc_entry cdc_item_found;
    cdt_entry cdt_item_found;
    int first = 1;
    int track_no = 1;

    if (strcmp(name_type, req_one_entry) == 0) {
```

5. This code shows a screen with one entry, defined by `cat_title`:

```
        html_header(1, "CD catalog entry");
        html_text("<BR><BR>");
        html_text("Return to: <A HREF=\"/cgi-bin/cddb/cdhtml\">list</A>");
        html_text("<BR><BR>");
        html_text("<HR>");

        cdc_item_found = search_cdc_entry(cat_title, &first);
        if (cdc_item_found.catalog[0] == '\0') {
            html_text("Sorry, couldn't find item ");
            html_text(cat_title);
        }
        else {
            sprintf(tmp_buffer, "Catalog: %s", cdc_item_found.catalog);
            html_text(tmp_buffer);
            html_text("<BR><BR>");
            sprintf(tmp_buffer, "Title: %s", cdc_item_found.title);
            html_text(tmp_buffer);
            html_text("<BR><BR>");
            sprintf(tmp_buffer, "Type: %s", cdc_item_found.type);
            html_text(tmp_buffer);
            html_text("<BR><BR>");
            sprintf(tmp_buffer, "Artist: %s", cdc_item_found.artist);
            html_text(tmp_buffer);
            html_text("<BR><BR>");
            html_text("<OL>");
```

```
            cdt_item_found = get_cdt_entry(cdc_item_found.catalog, track_no);
            while(cdt_item_found.catalog[0] != '\0') {
sprintf(tmp_buffer, "<LI> %s", cdt_item_found.track_txt);
                html_text(tmp_buffer);
                track_no++;
                cdt_item_found = get_cdt_entry(cdc_item_found.catalog,track_no);
            }
            html_text("</OL>");
            html_text("<HR>");
        }
    }
}
```

6. This short function lives up to its name by accepting a string and changing any occurrence of the space character to a + symbol:

```
void space_to_plus(char *str)
{
    while (*str) {
        if (*str == ' ') *str = '+';
        str++;
    }
}
```

How It Works
The first part of the process is to check that the database will initialize. If this fails, we print some simple HTML to allow the user to mail the Webmaster of the site to report the problem. We then check to see whether there was a query string whose first component name was `req_one_entry` (which is set to be

the string `CAT`). If there was, we call the routine `process_cat` to print out the tracks for a single entry. Otherwise, we call `process_no_entry` to list the known CDs in the database.

Information on a particular CD is given by the `process_cat` function, which we'll look at first, since it's slightly simpler. First, we send some HTML to add an anchor so the user can click to get back to the list of CDs. We then call the routine that we developed in Chapter 7, `search_cdc_entry`, to retrieve the information for the CD and send some more HTML to display the entry. After the header information, we use `search_cdt_entry` to scan the tracks. We list the tracks, using HTML to generate an ordered list, which gives us numbered entries.

If no catalog entry is selected, the function `process_no_entry` is invoked and it displays the standard screen that lists all the CDs in the database. The function searches the database using a blank string so that all the CDs are found. We then display them using HTML.

The "clever" part is in the lines

```
strcpy(tmp2_buffer, item_found.catalog);
space_to_plus(tmp2_buffer);
sprintf(tmp_buffer, "Title: <A HREF=\"/cgi-bin/cddb/cdhtml?CAT=%s\">%s</A>",
tmp2_buffer, item_found.title);
```

These create a copy of the catalog entry, but with spaces changed to + characters (as required by the standard rules for encoding `x-www-form-encoded` form data). We then generate an anchor line containing both the catalog name (to display on the screen) and the encoded version, so selecting the link calls the program `cgi-bin/cddb/cdhtml?CAT=CDA66374`. Remember that this invokes the `cdhtml` program, but passes `CAT=CDA66374` in the environment variable `QUERY_STRING`. The program detects this, decodes it, and uses it to select the `process_cat` function, with the name set to `CAT` and the value set to `CDA66374`, which it can then use to find and display the track information.

To create the full application, we've still got a little more work to do. For a start, we need to define all the functions that we used in `app_html.c`. Create a file called `html.c` and start it like this:

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#include "html.h"

name_value name_val_pairs[NV_PAIRS];
```

Then add in all the functions (except `main`, of course) that we defined in our `decode3.c` program. The remaining things we need are the two header files that we've `included` in our source code files, `html.h` and `cd_data.h`, and the additional source code file, `cd_access.c`.

`html.h` looks like this:

```
#define FIELD_LEN 250 /* how long can each name or value be */
#define NV_PAIRS 200 /* how many name=value pairs can we process */

/* This structure can hold one field name and one value of that field  */
typedef struct name_value_st {
    char name[FIELD_LEN + 1];
    char value[FIELD_LEN + 1];
} name_value;

extern name_value name_val_pairs[NV_PAIRS];

int get_input(void);
void send_error(char *error_text);
```

```
void load_nv_pair(char *tmp_buffer, int nv_entry_number_to_load);
char x2c(char *what);
void unescape_url(char *url);

void html_content(void);
void html_start(const char *title);
void html_end(void);
void html_header(int level, const char *header_text);
void html_text(const char *text);
```

but the other two need no effort at all since, in an excellent example of code reuse, they're exactly the same as the files of the same name that we used back in Chapter 7. To put everything together, we can use a short makefile:

```
all:    cdhtml

.c.o:
        gcc -g -c $?

html.o: html.c html.h
        gcc -g -c html.c

app_html.o: app_html.c cd_data.h html.h
        gcc -g -c app_html.c

cd_access.o: cd_access.c cd_data.h
        gcc -I/usr/include/db1 -g -c cd_access.c

cdhtml: app_html.o cd_access.o html.o
        gcc -o cdhtml -pedantic -g app_html.o cd_access.o \
html.o -ldb

install: cdhtml
        -echo Depending on your setup, you need to do something like...
        -echo cp cdhtml /usr/local/apache/cgi-bin/cddb
        -echo cp cdc_data.db /usr/local/apache/cgi-bin/cddb
        -echo cp cdt_data.db /usr/local/apache/cgi-bin/cddb
```
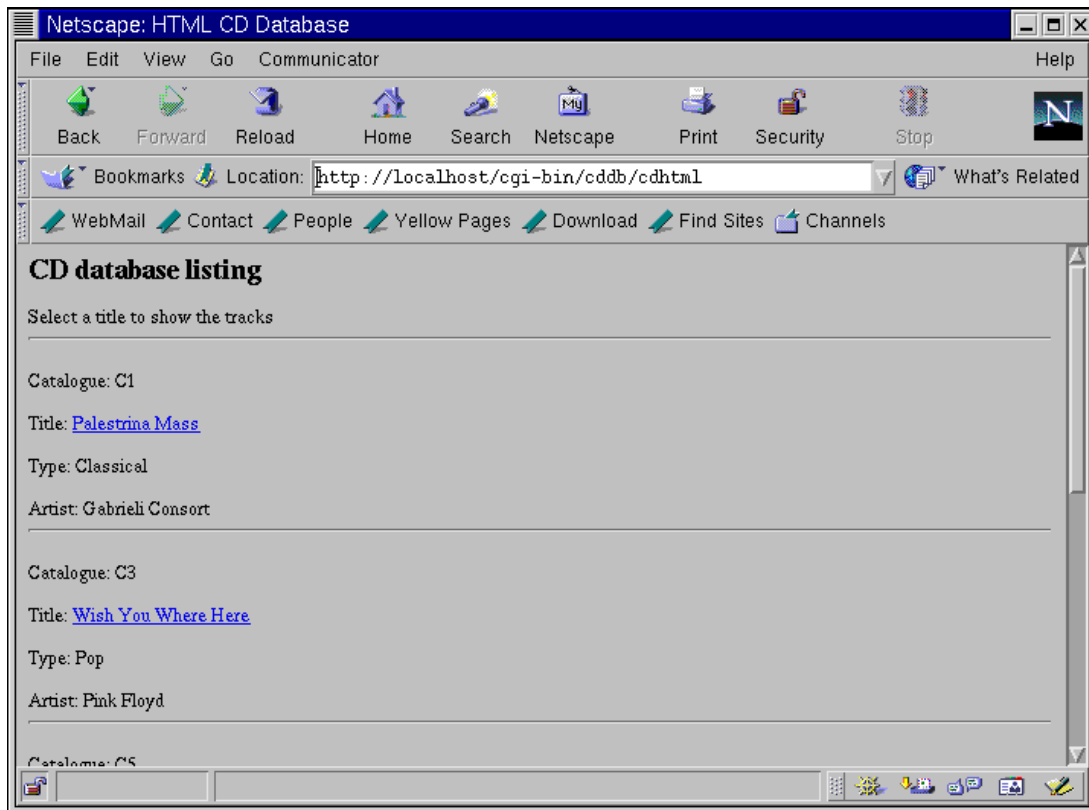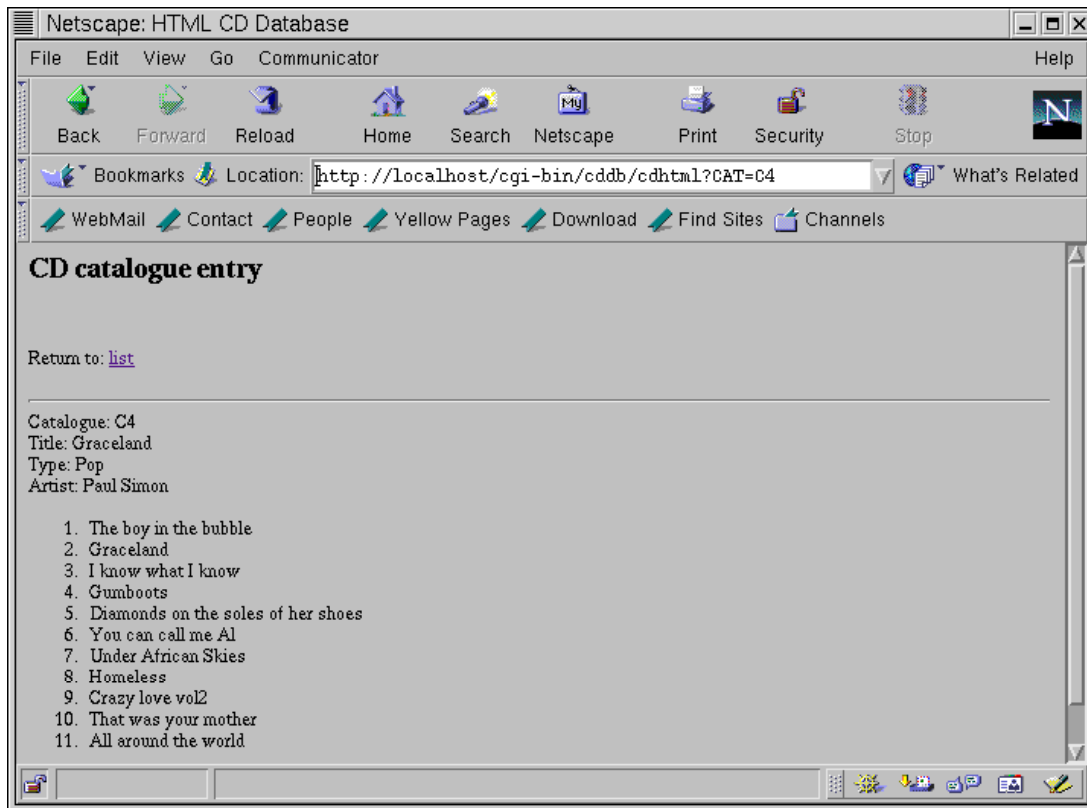
After we compile the program with make, we need to move it into the cgi-bin directory (don't forget the database cdc_data files from the earlier version of the application!), and we can access it using the URL http:localhost/cgi-bin/cddb/cdhtml.

This gives us a list of CDs, like this:

If we then click one of the highlighted links, we access the same executable, but this time with an appended query string. What we see are the tracks of the selected CD:

Netscape: HTML CD Database

File   Edit   View   Go   Communicator   Help

Back   Forward   Reload   Home   Search   Netscape   Print   Security   Stop

Bookmarks Location: http://localhost/cgi-bin/cddb/cdhtml?CAT=C4   What's Related

WebMail   Contact   People   Yellow Pages   Download   Find Sites   Channels

CD catalogue entry

Return to: list

Catalogue: C4
Title: Graceland
Type: Pop
Artist: Paul Simon

1. The boy in the bubble
2. Graceland
3. I know what I know
4. Gumboots
5. Diamonds on the soles of her shoes
6. You can call me Al
7. Under African Skies
8. Homeless
9. Crazy love vol2
10. That was your mother
11. All around the world

The `cdc_` and `cdt_` files that you've copied to the `/cgi-bin/cddb` directory must have read/write privileges set for everyone. Though `cdhtml` is read-only, the (unchanged) `cd_access.c` routines that open the database need to open the files for reading and writing.

If the thought of typing in all that code doesn't appeal to you, remember that the listings for this and all the other examples in the book are available from the Wrox Press Web site.

Perl

No book containing information about writing programs for a Linux-based Apache Web server would be complete without a mention of Perl. Although we have generally stuck to using C to be consistent with the main theme of this book, Perl is probably the most commonly used language for writing cgi programs on Linux and UNIX systems.

There are many Perl modules associated with writing cgi programs in Perl, most notably `CGI.pm`, which, amongst other things, does much of the hard work of parsing forms for you. However, we are going to leap forward to a more advanced topic, the powerful `mod_perl` facility. This effectively embeds a Perl interpreter inside the Apache Web server and allows it to process requests that involve executing Perl scripts without the overhead of starting the Perl interpreter each time a cgi program needs to be invoked.

In addition, you get access to some internal Apache state information, which helps you develop more complex generated Web pages and gives you the ability to change your Perl scripts without rebuilding the Web server each time you make a change. Indeed you can often make changes and just ask Apache to restart and reload your Perl scripts next time it is idle.

Unfortunately, there is a price you must pay. First of all, Perl scripts that you execute using `mod_perl` must be very carefully written   since they run inside the web server, you need to be quite careful about your use of variables and initialization. Secondly, the process sizes can get quite large, so you may need to significantly increase the memory on a server making heavy use of `mod_perl`. However, many sites judge that the power and flexibility of `mod_perl` vastly outweighs its drawbacks, and `mod_perl` is widely used on the Internet. You can even move your `mod_perl` scripts, normally unchanged, from Apache on a Linux or UNIX server, to Apache on Microsoft Windows.

We don't have anything like the space in this book to give you more than a quick taste of adding `mod_perl` support to Apache and writing your first Perl script for it. However, we do hope this brief look at a more advanced topic will encourage you to delve more deeply into this, and once you know the basics, there is plenty of helpful online documentation to guide you on your way.

The first thing you need is the sources for Apache. If your distribution came with a prebuilt Apache, but no sources, you may have to do what we did   uninstall the prebuilt Apache, fetch the sources of the latest stable version, compile them, and install them yourself. Don't worry; Apache is extremely easy to build and install, a matter of a few minutes work.

We assume you will be doing this on a test machine, and that once you have saved any work relating to your existing Apache Web server, you have removed it and are starting from a clean install. If your Linux distribution came with Apache and `mod_perl` ready built, you are in luck and can skip these steps. Otherwise, the safest course is to remove any Web server that came with your distribution and rebuild Apache from the sources, so we can link in the `mod_perl` additions. Don't worry; it's very easy.

First go to http://www.apache.org and fetch the latest sources; they are usually in a gzip-ed tar file with the version number included. Download this file to a convenient directory, unzip, and untar it. Read the `INSTALL` file that contains the instructions. If you want to install Apache in `/usr/local/apache`, a common location, `./configure --prefix=/usr/local/apache` is the first step; then you run `make`. If all looks well `su` to become root, and run `make install`. You then start your web server with the command `/usr/local/apache/bin/apachectr start` and away you go. We did say it wasn't hard!

Once you have Apache running from sources that you have built yourself, you are ready to install `mod_perl`. At the time of writing this is slightly more tricky than installing Apache, but not particularly difficult.

Assuming you already have Perl installed, but no additional modules, the first step in getting `mod_perl` working is to fetch additional modules from CPAN, the Comprehensive Perl Archive Network, which you can find at http://www.cpan.org. The minor inconvenience with `mod_perl` is that it relies on a large number of other modules, so each time you try installing `mod_perl`, it tends to complain that you need a different module installing first. But then again, you should be impressed at how much reuse is possible with Perl modules, so it's not all bad news! It gets more frustrating when some of those modules themselves require further modules. In the words of *The Hitch Hiker's Guide to the Galaxy* by Douglas Adams, "Don't Panic." There is an easy solution, and that solution is the CPAN Perl module.

Stage one is to fetch the FTP module. At the time of writing this is part of the `libnet` module. FTP to http://www.cpan.org and fetch a copy into an empty directory and extract the files. Then you need just four commands to get it installed:

```
perl Makefile.PL
make
make test
make install
```

Well, strictly speaking, you could omit the `make test`, but it's better to be safe than sorry. Now you have a Perl module that can "do" FTP for you. As it happens, it can also do SMTP, NNTP, and several other protocols, but FTP is the one that interests us right now.

Stage two is to install Andreas König's truly wonderful CPAN module. You configure this module once to tell it which CPAN sites are good download locations from your site; then it can take care of fetching modules from CPAN for you, working out dependant modules, fetching them, and installing them all in one go.

Repeat the steps for CPAN that you followed for `libnet`, except this time CPAN will ask you some questions. Don't worry; they are nice, easy ones such as "where can I make a directory to work in" and "tell me which continent you are on." Nothing too difficult.

Once you have CPAN installed, it's time for stage three, the Apache module bundle.

Since the CPAN module will need access to the Internet, but sometimes has to do some work in between, if you are on a dial-up link that has an automatic timeout, you may need to take some steps to keep the dial-up link alive. A simple `ping -i15 www.perl.com` will do the trick; just remember to kill it off once CPAN has finished it's magic, and allow the dial-up link to drop.

Start the CPAN module in interactive mode, with the command

```
perl -MCPAN -e shell
```

Perl will start, and you will be given an interactive `cpan>` prompt.

Enter the command

```
install Bundle::Apache
```

Then sit back and watch in amazement as all the hard work is done for you!

Don't worry if the final steps of installing `mod_perl` fail   we are going to run them again anyway, because we want to specify some extra options.

Once CPAN has finished, enter `exit` to return to the shell prompt.

Stage four (don't worry   we are getting there) is to re-fetch by hand the `mod_perl` module from CPAN, because we want different options from the defaults that the CPAN module gave us. As before, unpack it, but this time we want some extra command line options when we configure it.

```
perl Makefile.PL EVERYTHING=1 APACHE_PREFIX=/usr/local/apache
```

This means we want all `mod_perl` options, and our Apache Web server is installed in `/usr/local/apache`. If yours is elsewhere, you need to adjust the instructions as appropriate. Almost immediately you will be prompted for the source directory where you built Apache; type this in   you

need to include the final `/src` part of the tree. We built Apache in a local directory, so the answer on this author's machine was `/home/rick/apache/src/apache_1.3.9/src.`

The version number may well have increased by the time you get to read this.

You will then be asked questions about allowing Apache to be rebuilt. You probably want to answer yes to all these questions. Then we run `make`, to rebuild our `httpd` binary, which will be built in the Apache `src` directory.

When the compile steps have completed, you can run `make test` to test that you have built a `mod_perl`-enabled Apache server. It will run a special `httpd` using a different configuration file on a separate port, so it will not conflict with any existing web server running on the machine. Now it's possible that the test will fail. It did for us! At least this seemed to have been a problem with the testing itself, not the newly built `httpd`, so we kept going without further problems.

Now that we have an `httpd` program built with `mod_perl` support, the final stage is to install and configure it. Then we will be ready to write a test Perl module.

You can run `make install`, but our personal preference is to do this final step by hand, so that's what we will describe here.

Change into the `/usr/local/apache` directory, and `su` to become root.

If your web server is already running, stop it using the command

```
/usr/local/apache/bin/apachectr stop
```

Make a copy of the existing `bin/httpd` and `conf/httpd.conf` files, just in case.

Now overwrite the `bin/httpd` file with the one you just built in the apache/`src` directory. You should notice that the new `httpd` file is much larger than the old file.

There is just one thing left to do before we write our simple Perl module, which is to update the Apache configuration with some Perl additions.

There are several ways to do this, and what we are going to describe here is not the most efficient way of doing this, in terms of execution. However, it is probably the simplest way, and quite sufficient for learning about `mod_perl`.

Edit `httpd.conf` (remember to take a copy first). At a convenient location in the file, add some additional lines:

```
PerlFreshRestart On
<Location /blp-hello-perl>
        SetHandler perl-script
        PerlHandler Apache::Hello
</Location>
```

The `PerlFreshRestart On` tells Apache that when it is restarted, it should reload any Perl scripts. This is an option you need if you are debugging Perl scripts.

The next section, which looks a little like a section in an HTML document, is telling Apache that when it is asked for a document, `blp-hello-perl`, it needs to use the Perl script handler, and the Perl script it should invoke is `Hello`, in the module Apache.

Now we stop and start Apache, to let it reread the configuration file:

```
/usr/local/apache/bin/apachectr stop
/usr/local/apache/bin/apachectr start
```

Now we have our Apache server with an embedded Perl interpreter, and we have configured it to invoke a Perl script when we ask for a document called `blp-hello-perl`. It's time to Try It Out.

Try It Out—A mod_perl Module

The file we are going to create is `/usr/local/apache/lib/perl/Apache/Hello.pm`.

The directory `lib/perl` under the standard Apache install directory is one place `mod_perl` in Apache looks for Perl scripts. If you need to put it somewhere else, you need to add to the `httpd.conf` file an extra line: `PerlSetEnv PERL5LIB <comma separated list of places to look>`. The reason we created the additional `Apache` directory is that we are going to put our file in package call `Apache`. Here is our very first Perl module:

```perl
package Apache::Hello;

use strict;
use Apache::Constants ':common';

sub handler {
    my $r = shift;
    my $user_agent = $r->header_in('User-Agent');

    $r->content_type('text/html');
    $r->send_http_header;
    my $host = $r->get_remote_host;
    $r->print(<<END);
<HTML>
<HEAD>
<TITLE>HELLO</TITLE>
</HEAD>
<BODY>
<BR>
END

    $r->print("Hello $host with browser $user_agent \n\r<BR><BR>Welcome to Apache
running mod_perl\r\n");
    $r->print("</BODY>\r\n</HTML>\r\n");

    return OK;
}

1;
```

How It Works

We declare that we are a file, `Hello`, in module `Apache`. Then we turn on the `strict` option, always a good idea, and declare that we wish to use the common constants in the `Apache` module.

We declare a subroutine `handler`, which is a special name that Apache knows to invoke, to generate output for a web page.

The line `my $r = shift;` gets us a variable, `r`, containing the request object. This object has all the common routines we need to call to generate our page from Perl.

We then get the user agent that will have been passed in the HTTP request, send the content type, send a predefined HTTP header, and get the remote host name.

The final lines are simple Perl to generate a very basic HTML page that is returned to the browser simply by calling the print method of the requester object.

Well that's all the hard work done. There is just one thing left to do testing, and perhaps some debugging too.

Fire up your web browser and point it at http://localhost/blp-hello-perl.

If all went well, you should see
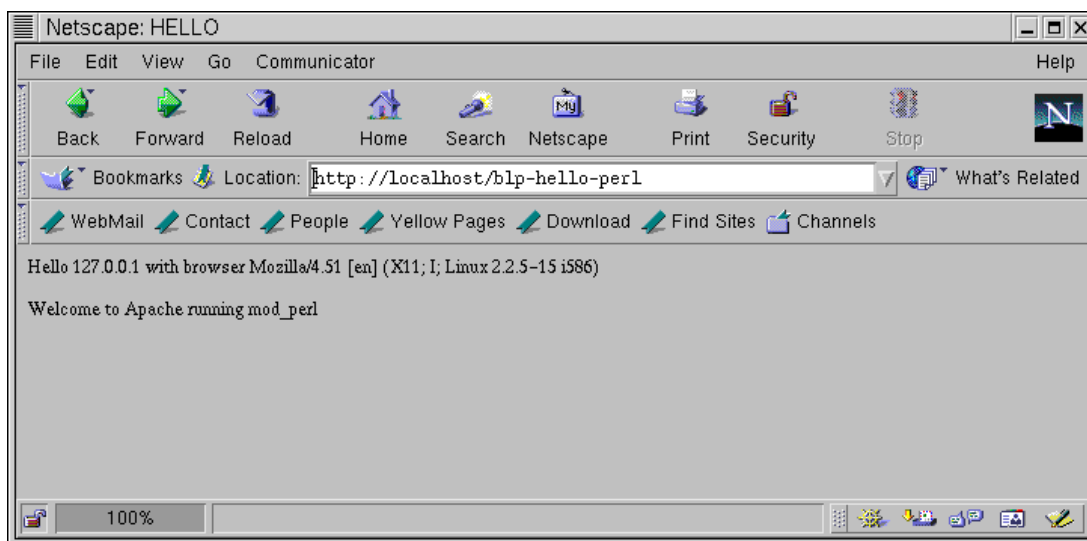
However, if it doesn't work the first time, don't worry. The first thing to try is a request to http://localhost/index.html. You should get the standard installation screen, telling you Apache is running OK. Assuming that worked, you should find the log files in `/usr/local/apache/logs`, which will give you some pretty good clues as to what went wrong.

Unfortunately, this very brief taster of `mod_perl` is all we have room for here. However, there is a huge amount of information available on the Web; just follow the links from http::/www.apache.org, and if you get serious with Apache modules, you may want to look out for other books on Apache modules.

Summary

This chapter has shown you how to create dynamic, interactive Web pages. A faster changing subject you're unlikely to come across, so there are ever-improving methods of displaying information on the Web. We hope this has given you the basis (in theory and application) to write your own simple CGI applications in C.

To recap, in this chapter we've covered

❑ The HTML to create forms for entering client-side information

❑ Encoding these forms for transmission

❑ The subsequent decoding using Perl or C

❑ Dynamically creating HTML pages with the CGI program's response

❑ CGI tips and tricks

❑ Using the CGI program to access our server database

❑ A brief glance at an advanced topic, `mod_perl`, for embedding a Perl interpreter in the Apache Web server