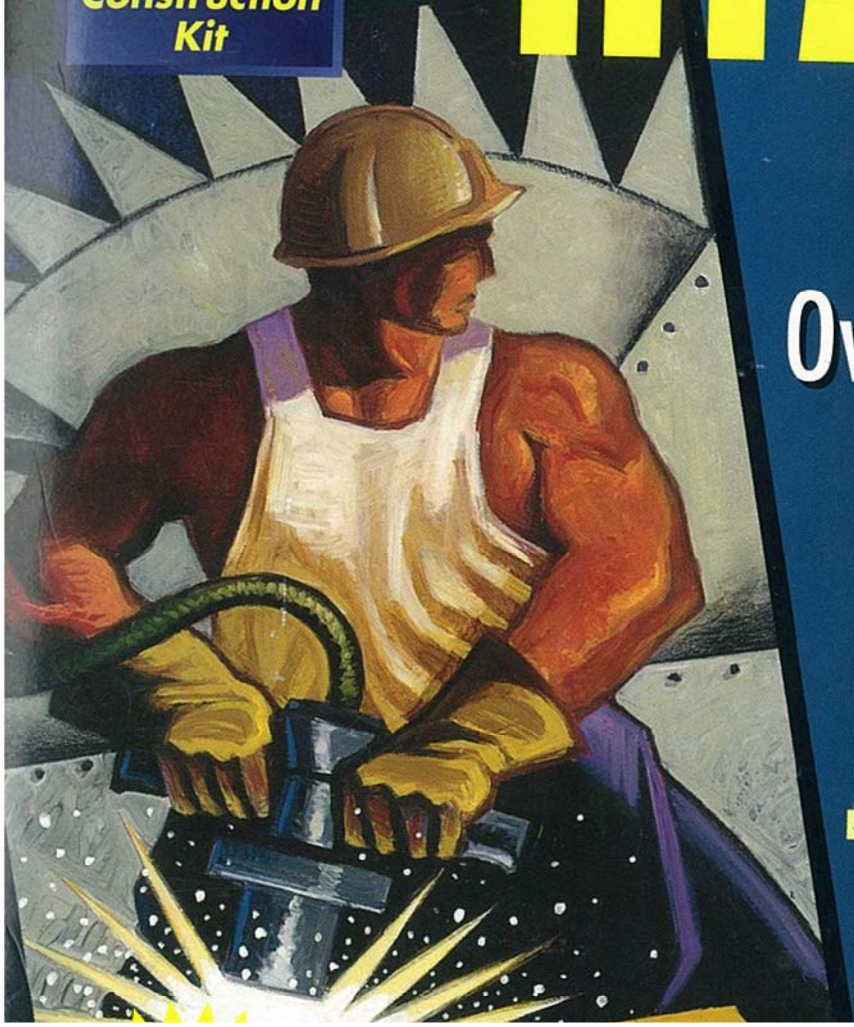
WEB PUBLISHER'S CONSTRUCTION KIT *with* HTML 3.2

Second Edition of best-selling *HTML Web Publisher's Construction Kit*



Publishing Your Own HTML Pages on the Internet

INCLUDES

NETMANAGE CHAMELEON, SLIPKNOT, HOTMETAL, HTML ASSISTANT, WINHTTPD, LVIEW PRO, WHAM, CGI SCRIPTS FOR UNIX AND WINDOWS, NCSA'S HTTPD FOR UNIX, AND MORE!

WEB PUBLISHER'S CONSTRUCTION KIT *with* HTML 3.2

Second Edition of best-selling ***HTML Web Publisher's Construction Kit***

Publishing Your Own HTML Pages on the Internet

INCLUDES
NETMANAGE CHAMELEON, SLIPKNOT, HOTMETAL, HTML ASSISTANT, WINHTTPD, LVIEW PRO, WHAM, CGI SCRIPTS FOR UNIX AND WINDOWS, NCSA'S HTTPD FOR UNIX, AND MORE!

COVERS THE LATEST VERSIONS OF NETSCAPE AND HTML

Includes CD-ROM

WAITE GROUP PRESS™

DAVID FOX, TROY DOWNING

Publisher • **Mitchell Waite**
Associate Publisher • **Charles Drucker**

Acquisitions Manager • **Jill Pisoni**

Editorial Director • **John Crudo**
Managing Editor • **John Crudo**
Copy Editor • **Scott Calamar, LightSpeed Publishing**
Technical Editor • **Miko Matsumura**

Production Director • **Julianne Ososke**
Production Manager • **Cecile Kaufman**
Production Editor • **Mark Nigara**
Cover Design and Production • **Sestina Quarequio and Karen Johnston**
Cover Illustration • **Rafael Lopez**
Illustrations • **Kristin Peterson**
Production • **Michele Cuneo**

Printed in the United States of America
95 96 97 98 • 10 9 8 7 6 5 4 3 2 1

Fox, David, 1973-
Web publisher's construction kit with HTML 3.2 / David Fox, Troy Downing.
p. cm.
Includes index.
ISBN: 1-57169-079-4
1. HTML (Document markup language) 2. World Wide Web (Information retrieval system)
I . Downing, Troy. II. Title.
QA76.76.H94F693 1996
025.04--dc20

96-30564
CIP

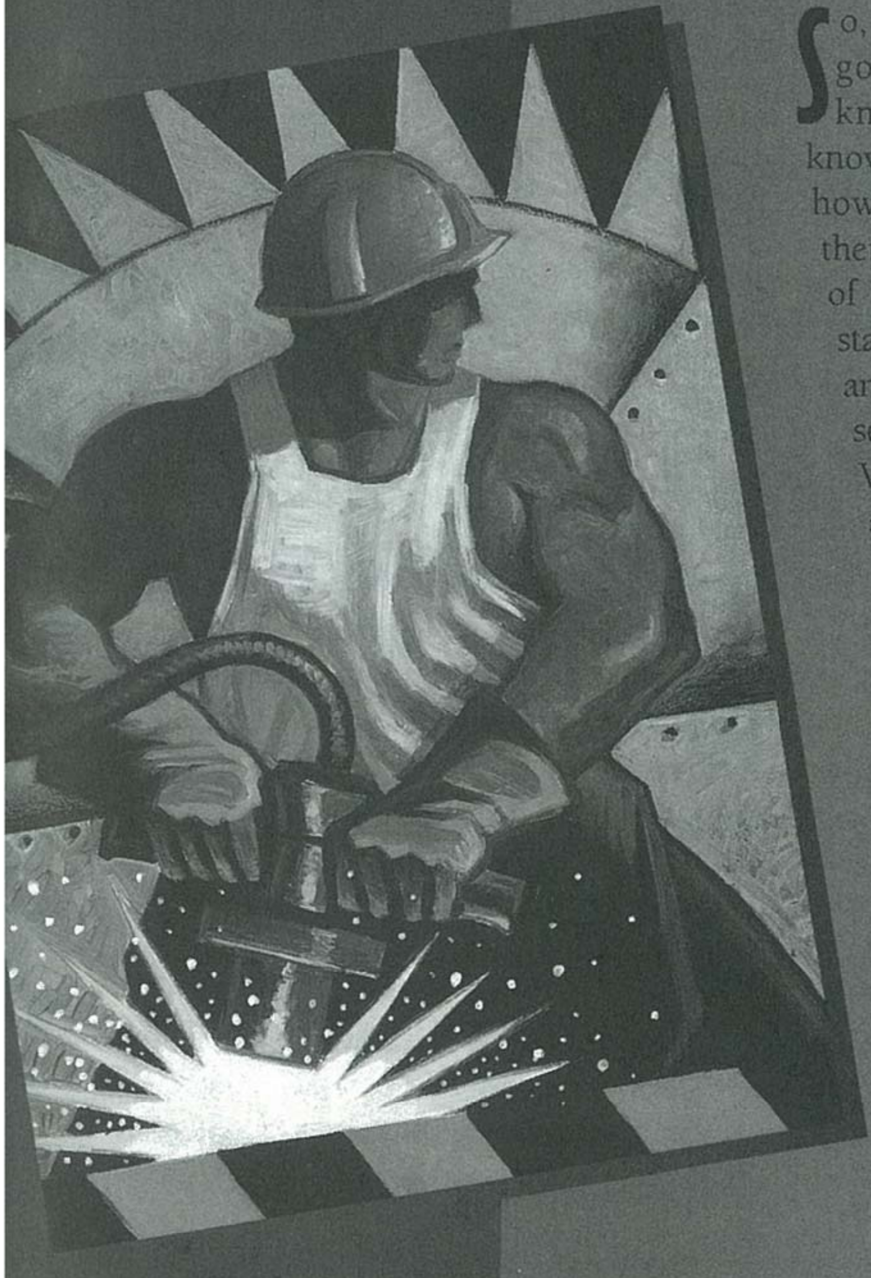
TABLE OF CONTENTS

CONTENTS

15

CGI SCRIPTS

15



So, by now you've probably got a pretty good understanding of HTML. You know how to create documents, you know what a form looks like, you know how to navigate the Web. What else is there? Only the single most exciting part of Web development, CGI scripts! CGI stands for Common Gateway Interface, and the short explanation is that a CGI script is a stand-alone program on your Web server, and is capable of doing anything that your server can, usually sending the results to a client Web browser. In fact, a CGI script can be any stand-alone program or script that can be executed on your system as long as it can send one of the standard HTML header types as its standard output. By "standard output" I mean anything that your program would normally send to your screen.

Sounds pretty exciting, doesn't it? In this chapter, you'll get a description of what a CGI script can do, what it can use as its input, and how it can format its output, and then you'll see some sample scripts that you can edit to suit your needs or just use as is. For the most part, the samples will be written in C and will assume that your server is running on a standard Unix platform, but if you are running a server on a Macintosh or Windows PC, you'll find a few tidbits that you can use as well. The main reason Unix is given preferential treatment as the Web server platform of choice is simple: Unix HTTP servers are the most common and most robust HTTP servers available as of this writing. That's not to say that you can't get satisfactory results from a Macintosh or Windows server; in fact there are many intriguing Web sites that are running on both of these platforms. It's just that the Unix side works better at this point in the history of the Web. Well, enough on that, let's get started.

LESSON #1: WHAT IS CGI?

The Common Gateway Interface—or CGI—is a method that lets you access external programs on a Web server and usually send the results to a Web browser. (There are situations in which you want the script to do some processing on your server, but not send data back to the client.)

These programs can be any executable code, script, or program supported by the operating system that runs your server. The CGI code to call an external program can be a shell script, or a batch file, an AppleScript file, a C program, a PASCAL program, compiled BASIC...literally anything that will run as a stand-alone executable Script on your system. Many CGI developers use shell scripts; others prefer Perl and C. Choose what works best for you! Just to make things simple, this chapter refers to all CGI code files as CGI scripts—or simply as "scripts"—whether they are written in a scripting language or in a compiled or interpreted programming language.

A server executes a CGI script based on a user request from a Web browser, as diagrammed in Figure 15-1. This request can be as simple as selecting a hyperlink that points to an executable item, or it can be a search request using the <ISINDEX> tag, or it can involve clicking the Submit button from within an HTML form. The parameters the script has available to it depend on how it was accessed. There are also many parameters available to scripts via environment variables that are set by the server. You'll get details on all of this in the Writing Your Own Scripts section.

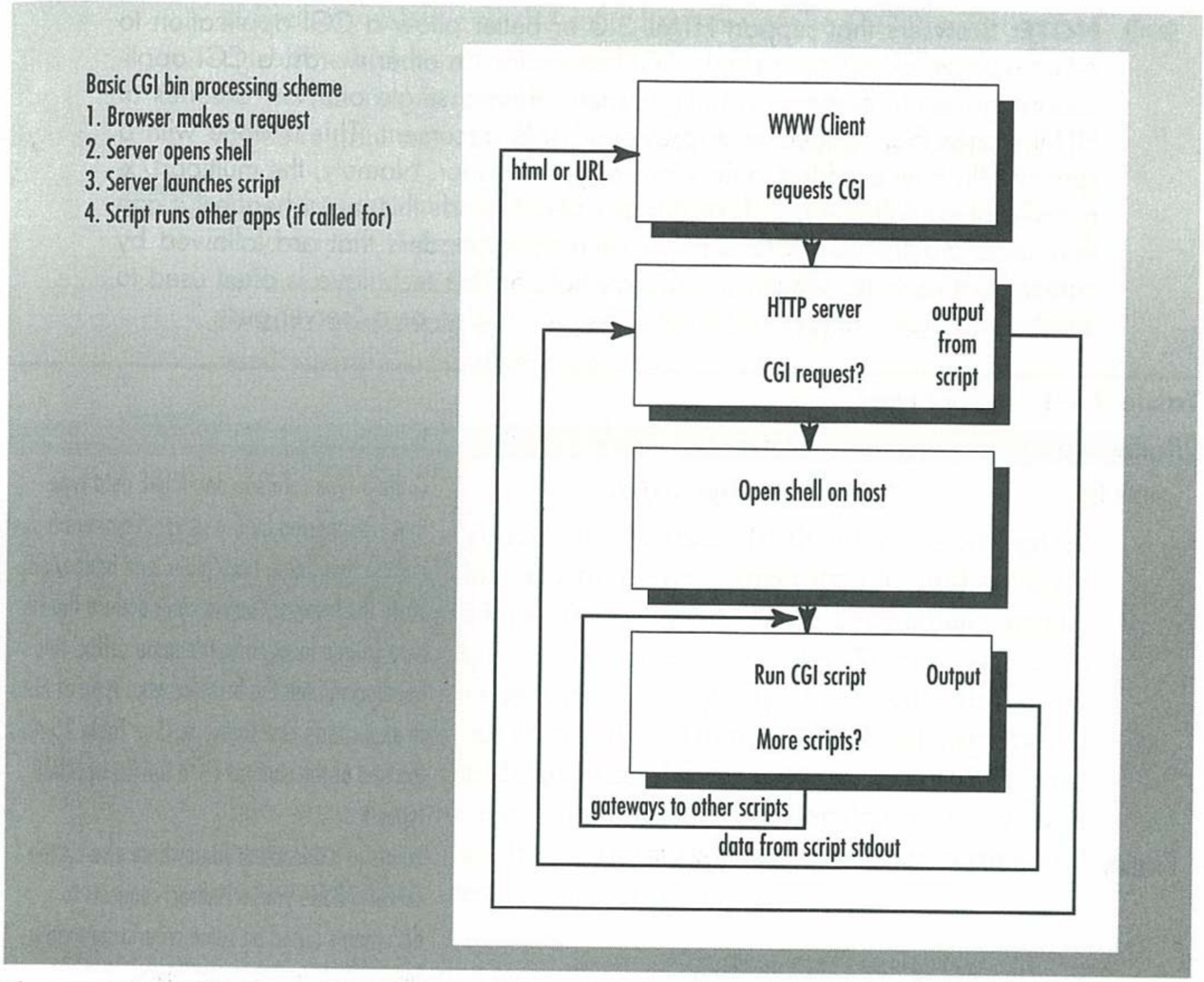


Figure 15-1 Diagram of a CGI session

A CGI script must produce an output header even if no data is to be forwarded to the Web browser. The HTML header must be the first thing that a script sends as output and must be followed by a blank line or carriage return. The header tells the server what kind of data to expect, if any; the server in turn tells the client that invoked the script what to expect. Currently, there are three types of headers. These headers are mutually exclusive—that is, you can't have more than one header for any one request. (See note for exception.) Valid header types are Content-Type, Location, and Status (see Table 15-1).

NOTE: Browsers that support HTML 3.0 or better allow a CGI application to return multiple objects in a single CGI transaction. In other words, a CGI application can return a series of images rather than a single one, or, a series of HTML pages that replace each previous HTML document. This is done with a special MIME type added to the Content-type: header. Namely, the multipart/x-mixed-replace MIME type. If a CGI application sends this initial header, it can then send an arbitrary number of Content-type: headers that are followed by content that replaces whatever was sent before. This technique is often used to create slide-show animations that are often referred to as a "server push."

Table 15-1 Header types

Header Type	Format	Description
Content-Type	Content-Type: xxx/xxx	Content-type refers to any MIME data type that is supported by the server. Common types include text/html, text/plain, and image/gif. Since the browser/server can't deduce the file type from a location or filename suffix, this heading will tell the browser what type of data to expect and how to use it. (See Table 15-4 at the end of the chapter for a full list of MIME types.)
Location	Location: /path/doc	Points to a document somewhere else on the server. Allows you to redirect requests to documents based on some criteria sent via a form or environment variable.
Status	Status: nnn XXXX	Can be used to run a script, without sending a new page to the client. Can also be used to send an error message or other information to the client.

What Can I Do with a CGI Script?

A CGI script can do anything allowed on the host system as long as it sends one of the three header types listed in Table 15-1. It can access other programs, open files, read from files, create graphics, dial your modem, call your mother, do database searches, send e-mail, you name it. The only rules are:

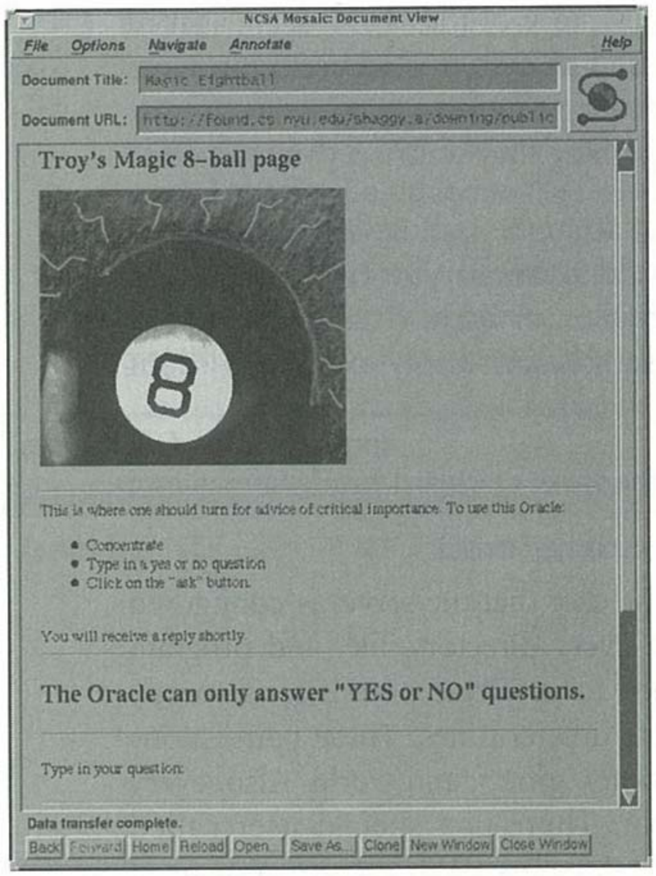
HTML The script has to be in a place designated by the server for CGI scripts, or it has to have a special suffix that the server is configured to recognize as a legal CGI script. Most systems store CGI scripts in a directory relative to the root directory of the HTTP server called cgi-bin, which is set up so that only certain trusted users can write to it. This avoids the obvious security problems of allowing anonymous remote users to execute anything they want on your system.

- The script can take its parameters from the standard input (by standard input, I mean what would normally be typed in at the keyboard), the environment variables, or both. (It is not necessary to take user input at all; the script can simply execute without needing any more information.)
- The script must output one of the three standard header types as a normal text string.
- The script must be runable by the user that the server is configured to run as. (On a Unix machine every directory, file, and program has a set of permissions attached to it. These permissions specify who can read, change, or execute different files. These permissions are divided into three groups: owner, group, and world. Also, every process must run as some user. There is a special user called "nobody" that is the default user for most Web servers. You must make sure that the user "nobody," or the user that your server is configured to run as, has permission to execute your scripts and read/write to any files that the script may use.)

CGI scripts are used for doing all of the "cool" stuff on the Net. There are sites that have interactive robots you can control with a Web browser, sites that allow you to control cameras and take pictures of remote places, sites that create graphic images on-the-fly, serve maps, open X clients on your machine and send you live video feeds, access huge databases, order submarine sandwiches, and ask questions of the Web's own version of the Magic Eightball, as shown in Figure 15-2. All of this is possible through the use of CGI scripts.

LESSON #2: THE STANDARD CGI SCRIPTS

As you read this, CGI scripts are coming to life all over the world. Some are special purpose, some are useful utilities, some are interesting, and some are just plain silly. Among these scripts, there are a handful that have become standard at most Web sites. If you download NCSA's HTTPD or copy it from the CD-ROM that came with this book, you'll find that it includes two


NCSA Mosaic: Document View
File Options Navigate Annotate Help
Document Title: Magic Eightball
Document URL: http://found.cs.nyu.edu/shaggy.a/downing/public
Data transfer complete.
Back Forward Home Reload Open... Save As... Clone New Window Close Window

Troy's Magic 8-ball page

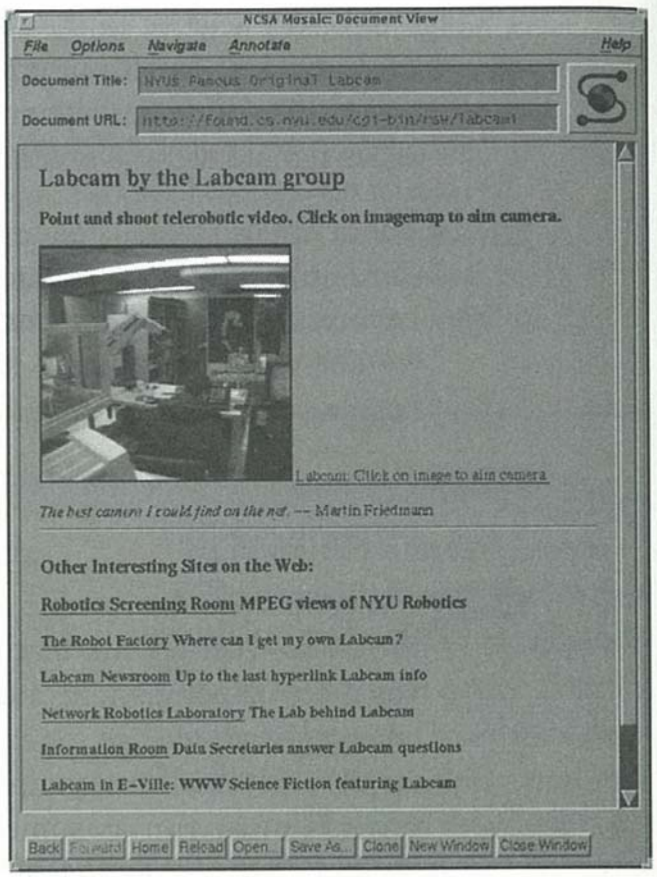
This is where one should turn for advice of critical importance. To use this Oracle:

- Concentrate
- Type in a yes or no question
- Click on the "ask" button.

You will receive a reply shortly.

The Oracle can only answer "YES or NO" questions.

Type in your question:


NCSA Mosaic: Document View
File Options Navigate Annotate Help
Document Title: NYUS Famous Original Labcam
Document URL: http://found.cs.nyu.edu/cgi-bin/rsw/labcam1
Back Forward Home Reload Open... Save As... Clone New Window Close Window

Labcam by the Labcam group

Point and shoot telerobotic video. Click on imagemap to aim camera.

Labcam: Click on image to aim camera

The best camera I could find on the net. -- Martin Friedmann

Other Interesting Sites on the Web:

Robotics Screening Room MPEG views of NYU Robotics

The Robot Factory Where can I get my own Labcam?

Labcam Newsroom Up to the last hyperlink Labcam info

Network Robotics Laboratory The Lab behind Labcam

Information Room Data Secretaries answer Labcam questions

Labcam in E-Ville: WWW Science Fiction featuring Labcam

Figure 15-2 Two nifty services, courtesy of CGI scripts

CGI directories, cgi-bin and cgi-src. The cgi-bin directory contains many demos and useful CGI scripts, and cgi-src contains the source code for these scripts so you can customize them—or just learn from them.

The CD-ROM scripts include C programs, shell scripts, and Perl scripts that do a few helpful little tasks for you. A few of them are even necessary for your server to have all the utility that you expect of it, like processing image maps. Table 15-2 describes all the scripts in the cgi-bin directory that comes standard with NCSA's HTTPD.

Table 15-2 Standard CGI scripts

Name	Type	Description
archie	shell script	Gateway to an archie server.
calendar	shell script	Gateway to the Unix calendar utility.
date	shell script	Calls the system date and sends it as an HTML doc.
finger	shell script	Gateway to the Unix finger utility.
fortune	shell script	Gateway to the Unix fortune utility.

Name	Type	Description
imagemap	C program	Handles imagemaps in HTML documents, taking the X and Y coordinates from the user, and forwarding a URL based on a map file created by the map developer.
jj	C program	Processes an order form from a submarine sandwich shop, then opens a pipe to a mailer, and faxes the order out of a fax modem. (A useful example, even if you're not selling sandwiches.)
nph-test-cgi	shell script	Echoes back the names and values of the environment variables. (Good for testing forms, or just figuring out what's going on.)
phf*	C program	Creates a fill-in form interface for a CSO ph database. (Great for looking up names/addresses on ph servers.)
post-query*	C program	Echoes the name/value pairs of a form that uses the POST method.
query*	C program	Echoes the name/value pairs of a form that uses the GET method.
test-cgi*	shell script	Echoes the names and contents of the environment variables.
test-cgi.tcl*	tclsh script	Echoes the names and contents of the environment variables.
uptime*	shell script	Gateway to the Unix uptime command. Will print the time that the system has been running.
wais.pl*	perl script	Provides an <ISINDEX> front end for WAIS searches.

Some of these are C programs, some are TCL scripts, and some are Perl scripts, but they all function in the same way. They get executed by the HTTP server, take their parameters (if any) from standard input or environment variables, and output at least a header when they're done.

As you can see, there are a few utilities here that you may find useful, such as a WAIS or finger gateway, and some that are necessary in some form to allow valuable utilities like imagemap, and still others that are intended simply as learning tools to demonstrate how to write a CGI script in the TCL scripting language. They all come standard with the NCSA HTTPD server software, and it's good to know what's there and how it can be used, either as a learning tool or utility. Later on, this chapter will describe many other special-purpose and form-handling scripts.

LESSON #3: HOW TO USE PRE-EXISTING SCRIPTS

OK, I know what a CGI script is, I know what one looks like, I know the names of a few standard scripts; now how do I use them?

The answer to this is simple. You access a CGI script in the same way you access any other URL: create a hyperlink in a document that points to a CGI script, or use the "open URL" option on your browser, or include the URL in the METHOD attribute of a form (as described in Chapter 14). Since most installations require all CGI scripts to be in one protected directory (namely cgi-bin), the following examples all use this convention.

Constructing a CGI URL

A URL that points to a CGI script follows the same conventions as other URLs that point to HTTP servers. It contains a protocol type (HTTP), the name of the server that will execute the script and forward the results, and the name and path of the CGI script to be executed. The simple generic format to create a CGI script URL is

```
http://machinename/cgi-bin/myprogram
```

In this form, the URL will open an HTTP connection to the "machinename" server, the server will then invoke the "myprogram" script from the standard cgi-bin directory, and will forward the results of the execution of "myprogram" back to the Web browser. There are also ways to include query and path information along with the URL. Query information can be appended to the URL separated by a question mark (?).

```
http://machinename/cgi-bin/myprogram?whoareyou
```

Here the HTTP server sets the environment variable QUERY_STRING to the value "whoareyou" when it executes the "myprogram" script. The script can then access the query data through the environment and make a decision based on the "whoareyou" value that was stored in the QUERY_STRING variable.

To include path information in the URL, simply append the relative path to the URL. For example:

```
http://machinename/cgi-bin/myprogram/people/docs
```

This will invoke the script and assign the value "/people/docs" to the environment variable PATH_INFO. It will also resolve the address from a virtual path to a physical path and store that value into PATH_TRANSLATED. For example, if your server root is set to "/usr/httpd" and "/people/docs" is sent along with the CGI URL, the server will assign PATH_INFO the value "/people/docs" and "/usr/httpd/people/docs" will be assigned to PATH_TRANSLATED.

The following listing is a sample HTML document that calls a CGI script called test-cgi on a server called myserver.com. If you have a standard HTTPD installation like the one on the CD-ROM that comes with this book, you should be able to replace "myserver.com" with the name of the machine running the server software and use this document.

```
<TITLE>test-cgi</TITLE>
<H1>Test CGI</H1>
<HR>
<A HREF="http://myserver.com/cgi-bin/test_cgi">Click here to run test-
cgi</A><BR>
<HR>
test-cgi should return a virtual HTML document that contains the names
of environment variables and their values on the HTTP server speci-
fied.<BR>
```

Specifying a Script Within a Form

When specifying a script to act on form data submitted from a client, construct the URL in the same way as before, that is

```
http://machinename/cgi-bin/programname
```

The only difference is where to place it within the HTML document. Specifically, include the script URL in the ACTION attribute of the <FORM> tag.

```
<FORM ACTION="http://machinename/cgi-bin/programname">
```

NOTE: The ACTION attribute in a <FORM> tag is optional. If there is no ACTION=*URL* attribute specified, the CGI script will be assumed to have the same URL as the document containing the <FORM> tag. This is useful when creating CGI scripts that generate the HTML documents that contain the forms that they process.

Later in this chapter, the section headed Handling Form Data will give you the details on creating a CGI script that will process the HTML form.

Specifying a Script from the "Open URL" Interface

Executing a script directly from your Web browser is as simple as selecting the Open URL or Open Location option in your browser and entering the

URL for the script. The URL can contain any of the standard URL conventions—see Chapter 1, Catching the Internet in a Web, if you need a review. In particular, you can include optional port numbers if you need them, and you have to escape any special characters that may be required to specify the path or filename of the CGI script.

LESSON #4: HOW TO USE A SCRIPT TO ACCESS OTHER APPLICATIONS

Why is it called the Common "Gateway" Interface? Well, the answer is simple: The Common Gateway Interface was originally intended as a "gateway" between WWW clients and other programs that could be run remotely on your server. Many CGI scripts, especially those that access databases, simply execute another application on the server and redirect its output with whatever formatting changes are required to the HTTP server and then to the client that requested the script.

As a simple example, let's take a quick look at the finger script that was mentioned in the standard script table (Table 15-2). Finger is a standard Unix utility that allows you to locate users and/or machines and retrieve information about them. Don't worry if you don't understand the entire script; we'll cover that in the next section.

NOTE: In the following code examples, the HTML code generated is not complete. Some important tags were left out in order to keep the examples short and simple. Namely, the output of the CGI applications should include <HTML>, <HEAD>, and <BODY> tags just as any well-structured HTML document would contain.

The finger gateway is written in the Unix shell scripting language, so first we give it a shell script header, and define a constant that points to the actual finger program in the Unix filesystem.

```
#! /bin/sh

FINGER=/usr/ucb/finger
```

Notice that the FINGER constant is assigned the entire path of the program that it will be running. Next, we send the server a standard header. We'll use the Content-Type header and specify the "text/html" MIME type. (For a complete list of MIME types, see Table 15-4 at the end of the chapter.) This will inform the client that we plan on sending it straight ASCII text, and that the text should be interpreted as HTML code. It is important to specify a header of some type, as most servers and browsers will return an error message if they don't get a header. We send the header to the server simply by writing it to the standard output. (In the examples of this chapter, standard output is the same as printing directly to the screen or console. The HTTPD server intercepts and redirects this output as necessary.) An easy way to do this in a shell script is with the "echo" command. Note the blank echo line after the header. This is necessary with most servers and should always be included.

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

Now we are ready to start sending the output from the finger script to the server. Just to make it look a little nicer, we add a <TITLE> tag and a short description of the output as follows:

```
echo <TITLE>Finger Gateway</TITLE>
echo <H1>Finger Gateway</H1>
echo This will finger our HTTP server
echo
```

Now, we execute finger, with a <PRE> tag added just before and a </PRE> right after to make it look a little nicer to the user:

```
echo <PRE>
$FINGER
echo</PRE>
```

Since finger will automatically send its results to the standard output, this text goes to the server and then to the browser as part of an HTML document. This simplified finger gateway looks like this:

```
#! /bin/sh
FINGER = /usr/ucb/finger
echo Content-type: text/html
echo
echo <TITLE>Finger Gateway</TITLE>
echo <H1>Finger Gateway</H1>
echo This will finger our HTTP server
echo
<PRE>
$FINGER
</PRE>
```

Since finger usually works best with parameters, such as user@machine, it's nice to be able to pass along a parameter supplied by the user. The full finger gateway uses the <ISINDEX> tag to get a username and machinename from the user, and passes these along to the Unix finger utility. A listing of the complete finger gateway is listed below. Some parts may be unfamiliar, but these will be discussed in the Writing Your Own Scripts section.

```
#! /bin/sh
#This script comes standard with NCSA's HTTPD
FINGER=/usr/ucb/finger
echo Content-type: text/html
echo
if [ -x $FINGER ]; then
       if [ $# = 0 ]; then
               cat << EOM
<TITLE>Finger Gateway</TITLE>
<H1>Finger Gateway</H1>
<ISINDEX>
This is a gateway to "finger". Type a user@host combination in your
browser's search dialog.<P>
EOM
       else
               echo \<PRE\>
               $FINGER "$*"
       fi
else
       echo Cannot find finger on this system.
fi
```

WARNING! Remember to make your shell scripts executable. In Unix this means that you must type

```
chmod a+x scriptname
```

for any new shell script you create.

It should start becoming clear how simple and powerful a gateway script can be. A script can point to any executable file on your server and execute it. All data sent to the standard output—either your script or the file(s) it executes—will be forwarded to the client and interpreted as the MIME type specified in the Content-Type header. This is a lot of power and should be used with caution. You don't want users imposing potentially (or deliberately) destructive scripts on your server, so it is usually a good idea, on a shared system, to allow only a few trusted users to create CGI scripts.

There are a few mechanisms in place to help protect you. One is the ability to require CGI scripts to be in a specific directory, and the other is the ability to require CGI scripts to have a specific suffix. Either method prevents anonymous users from being able to write URLs that point to your server and run whatever they want (Say, rm-r * for example. Not a pleasant thought.) Don't worry! The risk potential is there, but if you install your server with some thought, you should be able to avoid such mishaps. If you are concerned about security, read the security section in Chapter 21, HTML Assistant.

LESSON #5: WRITING YOUR OWN SCRIPTS

OK, enough talk, let's see some action. We're going to try the learning-by-example method here, so let's just get a few things out of the way first. In order to run CGI scripts, make sure the following infrastructure is in place:

- You have an HTTP server installed at your site.
- The HTTP server has been configured to allow CGI scripts.
- You, or someone you know, has write permission in the cgi-bin directory on this server, unless the server has been configured to allow CGI scripts elsewhere.

That in mind, writing a CGI script is a six-step process:

1. Write the script and compile it if necessary. (Obviously, you don't compile a shell script.)
2. Have the script moved into the cgi-bin directory (or equivalent).
3. Make sure the script is executable. (The Unix command is chmod a+x scriptname.)
4. Write a reference URL or form to access the script.
5. Debug the script.
6. Publish the script. (Tell your audience about it, or create links to it.)

Writing a Simple Script

Let's start with a simple script: an interactive <ISINDEX> form that will ask the user to input his or her name, and then echo back a short greeting to the user's browser. This example assumes that you are using the NCSA HTTPD server software from the CD-ROM that came with the book.

We will use the Unix shell scripting language to write it. To start, using your favorite text editor, create the following file:

```
echoname.sh
```

Since this is a shell script, we will start it with a standard shell script header. The first line is:

```
#! /bin/sh
```

Now, to avoid having problems interpreting the data, or getting error messages for not being specific, we add the HTML header. In this case, we are returning text that we want interpreted as HTML code. The header for this type of data is just the MIME type for HTML code. Add the following to your file to print the header information:

```
echo Content-Type: text/html
echo
```

The extra "echo" is necessary. A blank line is used to separate the header from the actual content. Next, we want to create the HTML code that is sent to the browser. We will share some code, and the rest will be unique depending on whether there was user input or not. First we create the common HTML. Notice that in a shell script, the greater-than and less-than brackets (< >) are reserved symbols and must be escaped with backslashes (\). So if you want to print a less-than symbol using echo, you would use "echo \<"—"echo <" won't work by itself. You can avoid messing with backslashes by enclosing the entire string in double-quotes. With that in mind, add the following lines to your file:

```
echo "<TITLE>Echoname example</TITLE>"
echo "<ISINDEX>"
echo "<H1>CGI script example</H1>"
echo Any name typed into the query window will be echoed to the
screen.
echo "<HR>"
```

Now we want to create a fork—one side allows the user to input a name, and the other displays a message if the user types in a name. In this example, since we are using the <ISINDEX> tag, we can assume that the command line parameter count is greater than zero if the user entered a value, and zero if not. The following lines will check for a value on the command line and print a prompt message if there were no parameters.

```
if [ $# = 0] ; then
        echo Please enter your name in the query window.\<BR\>
else
        echo Hello $*, Welcome to our server.
fi
```

Now, if this script is called without parameters, it will print the message "Please enter your name in the query window." When called with parameters, it will print the message "Hello [parameters], Welcome to our server." The entire script looks like this:

```
#! /bin/sh
echo Content-Type: text/html
echo
echo "<TITLE>Echoname example</TITLE>"
echo "<ISINDEX>"
echo "<H1>CGI script example</H1>"
echo Any name typed into the query window will be echoed to the
screen.
echo "<HR>"
if [ $# = 0 ]; then
        echo Please enter your name in the query window.\<BR\>
else
        echo Hello $*, Welcome to our server.\<BR\>
fi
```

NOTE: The <ISINDEX> tag uses the GET method to pass data from the browser to the calling script. This means that the encoded input data is stored in the QUERY_STRING environment variable. But the <ISINDEX> query will also list the unencoded values on the command line of the calling script. In the previous script, the $* operator refers to the command line arguments.

Before we can test the script, we need to make sure it is executable. At the Unix command prompt, type

```
chmod a+x echoname.sh
```

This will make the shell script executable, a necessity if you want to be able to run this script. Now to test it, type

```
echoname.sh
```

This should produce the results:

```
Content-Type: text/html

<TITLE>Echoname example</TITLE>
<ISINDEX>
<H1>CGI script example</H1>
Any name typed into the query window will be echoed to the screen.
<HR>
Please enter your name in the query window. <BR>
```

Now test the script with a command line argument. Try the following:

```
echoname.sh Troy
```

The result should be:

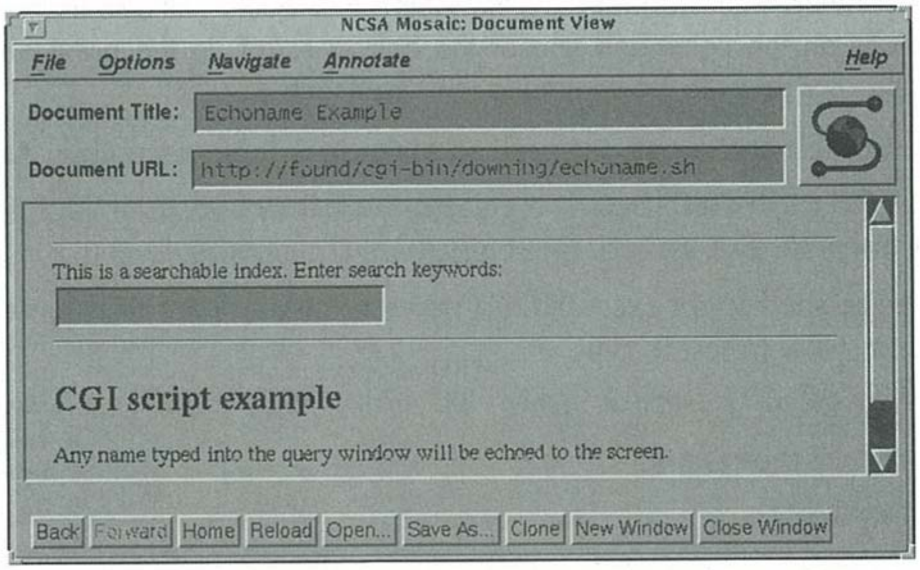
```
Content-Type: text/html

<TITLE>Echoname example</TITLE>
<ISINDEX>
<H1>CGI script example</H1>
Any name typed into the query window will be echoed to the screen.
<HR>
Hello Troy, Welcome to our server.
```

We are now ready to place the file into the correct directory and try it out with a Web browser. Copy the file into the appropriate directory for CGI scripts on your server. If you're using NCSA HTTPD from the CD-ROM, this is the cgi-bin directory. In any case, the command will be something like:

```
cp echoname.sh /usr/httpd/cgi-bin
```

Now, let's try it out with a browser. Take your favorite Web browser and open the following URL, substituting the name of your server and the exact path to your CGI directory as necessary:

```
http://yourserver/cgi-bin/echoname.sh
```



Document Title: Echoname Example

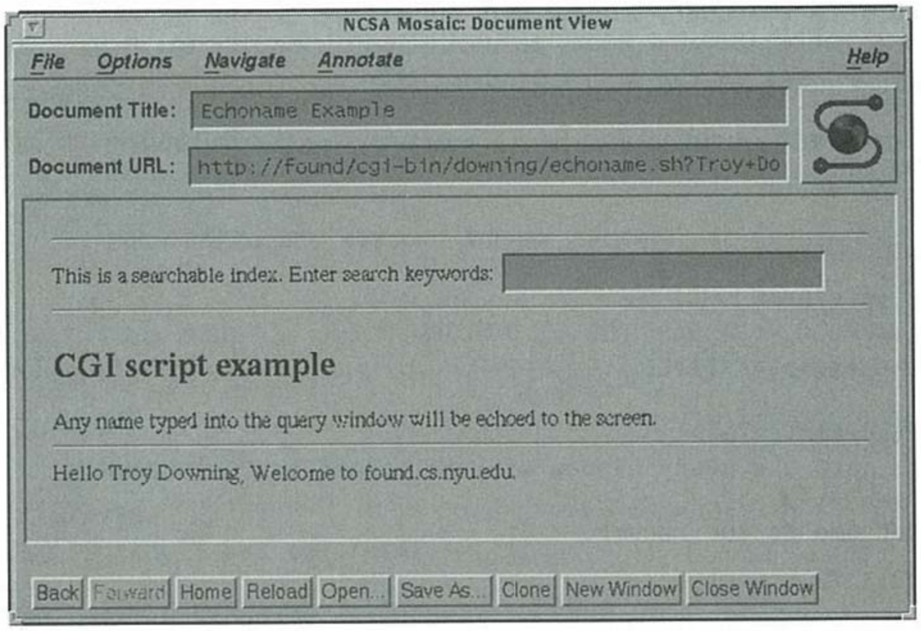
Document URL: http://found/cgi-bin/downing/echoname.sh

This is a searchable index. Enter search keywords:

CGI script example

Any name typed into the query window will be echoed to the screen.

Figure 15-3 echoname.sh with no arguments



Document Title: Echoname Example

Document URL: http://found/cgi-bin/downing/echoname.sh?Troy+Do

This is a searchable index. Enter search keywords:

CGI script example

Any name typed into the query window will be echoed to the screen.

Hello Troy Downing, Welcome to found.cs.nyu.edu.

Figure 15-4 echoname.sh with "Troy Downing" as the argument

Opening this URL should cause your HTTP server to execute the script "echoname.sh" and send back an HTML page with a query window as in Figure 15-3.

Now try typing a name into the query window. The resulting screen should look something like Figure 15-4.

Congratulations! You've just created your first CGI script. Doesn't do a whole lot, but it shows how simple script writing can be. This script could have been created just as easily in another scripting language or a compiled language such as C or PASCAL. The next example will show how to use environment variables to get information about the user and the user's environment.

Using Environment Variables in Scripts

Whenever a server launches a CGI script, a new shell is launched and a number of environment variables are set with information about the data being sent, the client software, the client machine, even the username in some authentication schemes. See Table 15-3 for a list of environment variables set on the NCSA HTTPD server. As a simple exercise, we are going to add a few lines of code to the previous echoname.sh script to make use of some environment variables. The only lines we are going to change are the few at the end that print the "hello" message. The two variables we will use to demonstrate this are SERVER_NAME and REMOTE_HOST. SERVER_NAME is set to the name of the machine that is running the HTTP server and REMOTE_HOST is the name of the machine that is making the HTTP request. Let's make the following changes to echoname.sh:

```
#!/bin/sh
echo Content-Type: text/html
echo
echo "<TITLE>Echoname example</TITLE>"
echo "<ISINDEX>"
echo "<H1>CGI script example</H1>"
echo Any name typed into the query window will be echoed to the
screen.
echo "<HR>"
if [ $# = 0 ]; then
        echo "Please enter your name in the query window.<BR>"
else
        echo Hello $* from $REMOTE_HOST, Welcome to \
        $SERVER_NAME.\<BR\>
fi
```

Notice the addition to the last "echo" line. We've added the two environment variables to our greeting. Now execution of this file should result in a reply string that looks something like:

```
Hello Troy from play.cs.nyu.edu. Welcome to www.nyu.edu.
```

You may or may not want to use the environment variables in this way. Environment variables are particularly useful when processing forms. In many cases, you need them for retrieving data from forms using the GET method, and for determining the length of the data block when using the POST method. This will all be explained in greater detail in the section on form handling coming up next. Before going on to forms, let's take a look at a few more scripts.

Table 15-3 Environment Variables

Variable Name	Description
SERVER_SOFTWARE	The name and version number of the server software that is serving the request, and running the CGI script. Format: name/version.
SERVER_NAME	The server's hostname, alias, or IP address depending on the particular installation.
GATEWAY_INTERFACE	Revision number of the gateway interface. Format: CGI/revision #.
SERVER_PROTOCOL	The protocol name and revision of the protocol that the request came in with. Format: protocol/revision.
SERVER_PORT	The port number that the server is accepting requests through. (Usually port 80.)
REQUEST_METHOD	The method of the request. Normally POST or GET.
PATH_INFO	The path information that came along with the request. Normally, this information was appended to the end of the URL that called the CGI script.
PATH_TRANSLATED	The physical mapping that is derived from the virtual path supplied in PATH_INFO.
SCRIPT_NAME	The path and file name of the script.
QUERY_STRING	The value of a query URL or a form that was sent using the GET method is stored here. The QUERY_STRING is url-encoded, unless the query was invoked with the <ISINDEX> tag, then the "name" of the field is omitted and only the value is assigned to QUERY_STRING variable. In <ISINDEX> calls, the unencoded value will also be passed along to the script as command line parameters.
REMOTE_HOST	The host name of the machine making the request. Either the DNS name or alias.
REMOTE_ADDR	The IP address of the REMOTE_HOST.
AUTH_TYPE	The authentication method used to validate users for protected scripts.
REMOTE_USER	The user name making the request. This value is only set if user authentication has been used.
REMOTE_IDENT	The user ID for a remote user in some authentication schemes.
CONTENT_TYPE	The MIME type of the data being served.
CONTENT_LENGTH	The number of bytes of content being sent by the client.
HTTP_ACCEPT	The MIME types that the client will accept. Format type/type, type/type,...
HTTP_USER_AGENT	The browser that the client is using.

As an exercise and a utility to see what your environment variables are being set to, we will write a short shell script that simply returns the values of all the main environment variables.

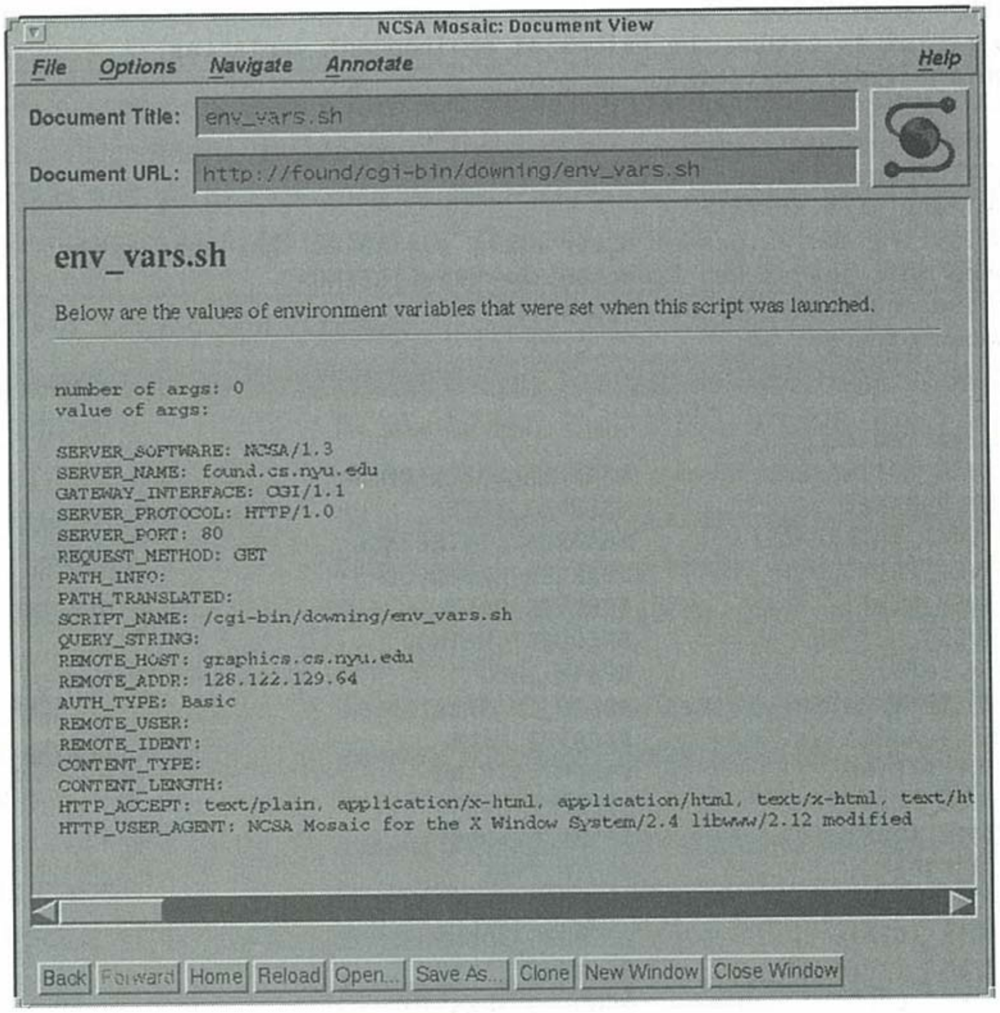
```
#!/bin/sh
#simple script to return the values of environment variables.
echo Content-Type: text/html
echo
#simple header info
echo "<TITLE>env_vars.sh</TITLE>"
echo "<H1>env_vars.sh</H1>"
echo "Below are the values of environment variables that were set"
echo "when this script was launched.<b><HR><LISTING>"
#were there any command-line arguments?
echo number of args: $#
echo value of args: $*
echo
#now the variables
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 QUERY_STRING:              $QUERY_STRING
echo REMOTE_HOST:               $REMOTE_HOST
echo REMOTE_ADDR:               $REMOTE_ADDR
echo AUTH_TYPE:                 $AUTH_TYPE
echo REMOTE_USER:               $REMOTE_USER
echo REMOTE_IDENT:              $REMOTE_IDENT
echo CONTENT_TYPE:              $CONTENT_TYPE
echo CONTENT_LENGTH:            $CONTENT_LENGTH
echo HTTP_ACCEPT:               $HTTP_ACCEPT
echo HTTP_USER_AGENT:           $HTTP_USER_AGENT
```

Be sure to make this script executable and put it into the correct directory for CGI scripts. If you include this script as the action for a form, or just call the script directly from your favorite browser, it will list the contents of the environment variables that we listed in the script. Typically, the results will look something like Figure 15-5.

Location and Status Headers

The Content-Type header we've been discussing tells the browser to expect a stream of data of a certain type, but sometimes you don't want to create a data stream at all. If you want your script to simply redirect clients to a different location based on the machine they are connecting from or the browser they are using, use the Location header. You can also use a Location header to point the browser to a different URL. The format is simple:

```
Location: http://foo.com
```



Document Title: env_vars.sh

Document URL: http://found/cgi-bin/downing/env_vars.sh

env_vars.sh

Below are the values of environment variables that were set when this script was launched.

```
number of args: 0
value of args:

SERVER_SOFTWARE: NCSA/1.3
SERVER_NAME: found.cs.nyu.edu
GATEWAY_INTERFACE: CGI/1.1
SERVER_PROTOCOL: HTTP/1.0
SERVER_PORT: 80
REQUEST_METHOD: GET
PATH_INFO:
PATH_TRANSLATED:
SCRIPT_NAME: /cgi-bin/downing/env_vars.sh
QUERY_STRING:
REMOTE_HOST: graphics.cs.nyu.edu
REMOTE_ADDR: 128.122.129.64
AUTH_TYPE: Basic
REMOTE_USER:
REMOTE_IDENT:
CONTENT_TYPE:
CONTENT_LENGTH:
HTTP_ACCEPT: text/plain, application/x-html, application/html, text/x-html, text/ht
HTTP_USER_AGENT: NCSA Mosaic for the X Window System/2.4 libwww/2.12 modified
```

Figure 15-5 env_vars.sh results

Like the Content-Type header, the Location header requires a blank line after it. The Location header can be followed by any valid URL. To use this header in a shell script, it would look something like:

```
echo Location: ../downing/funstuff.html
echo
```

NOTE: You cannot mix header types. Every header must be either Content-Type, Location, or Status.

The following script will redirect a request based on the browser making it. A Netscape browser will get a file formatted for Netscape, other browsers will get a default page.

```
#!/bin/sh
#This will send the location of a file based on the
#client browser
FILENAME="default.html"
#default.html is the standard HTML file we want to serve
#if the user is using Netscape, we will redirect to
```

```
#nsversion.html
if [ "$HTTP_USER_AGENT" = "Mozilla"]; then
        FILENAME="nsversion.html"
fi
        echo Location: ../htdocs/$FILENAME
        echo
```

If you want to run a CGI script without having any change appear on the user's browser, use the Status header. If the script returns a Status header with the status number set to 204 and the string "No Response" attached, the calling browser will simply stay on the page that the request was made from. In other words, the browser does nothing, even though the server ran a remote script based on the browser's request. We can take care of some task, say add a line to a database, without changing the user's current page. The following script will add the machine name that the request came from, and the name of the browser used, to a database in the "logs" directory relative to the cgi-bin directory.

```
#! /bin/sh
#This will add the machine name and browser
#name of a client to a database
LOGPATH="../logs/browser.dat"
echo $REMOTE_HOST $HTTP_USER_AGENT >> $LOGPATH
#now send the status to the browser
echo Status: 204 No Response
echo
```

Assuming that you have created a file called /logs/browser.dat, this script will add the remote host name and the browser name to this file and terminate, sending a status code back to the client. The client will stay on the page where the call came from. There are a number of status codes that are sent from a server to a browser. Most of them aren't very useful in cgi scripts but are used to tell the browser that a file was not found or that the user doesn't have permission to access a certain file. I'm sure you've all seen the "404 Not Found" error message—this was status number 404.

Security with CGI Scripts

BEWARE! Watch out for characters that have special meanings to the shell, such as %.<,>... A client can enter these characters into input fields and sometimes compromise your system if you don't handle them carefully. Any user-supplied data that is used as a command line argument can take advantage of this problem. An easy, but not foolproof way to handle some of these problems is to include the command line parameters in double quotes so that any special characters will be treated as literals rather than shell directives. See Chapter 21, HTML Assistant, for more information on server security.

Most HTTP servers and clients have certain security features built in, but you may occasionally want to try protecting a document by having a CGI script ask the user for a password of some sort. The following script will give very rudimentary security to a script; it's listed here as an example of how you might implement such a scheme, even though it's not necessarily a completely secure solution. It will print a message prompting the user to input a password. Since this is a single field of input, we will use the <ISINDEX> tag.

NOTE: If you wanted to add a more secure password field in a form, it would make sense to use <FORM> tags instead of the <ISINDEX> tag and use the <INPUT TYPE="password"> tag to prevent characters from being echoed to the screen.

This script will return a "failure" page if it receives an incorrect password. If it gets the correct password, it will redirect the browser to another location.

```
#!/bin/sh
#simple password script
PASSWORD=Schmoo
PROTECTEDFILE=/usr/me/securefile.html
if [ $# = 0 ]
then
        cat << EOM
                Content-Type: text/html
                <TITLE>Password script</TITLE>
                <ISINDEX>
                <H1>This page is protected. Enter password</H1>
EOM
elif [ "$#" = "$PASSWORD" ]
then
        echo Location: $PROTECTEDFILE
        echo
else
        cat << EOM2
                Content-Type: text/html

                <TITLE> FAIL!</TITLE>
                <ISINDEX>
                <H1>Password failed! Try again.</H1>
EOM2
fi
```

Remember—passwd.sh is meant as a demonstration to base other schemes on; it's far from the most secure way to protect a page or server. If you are interested in security, read the security section in Chapter 21, HTML Assistant.

LESSON #6: HANDLING FORM DATA

And now, (drum roll please), the moment you have all been waiting for... Form Handling! In Chapter 14, Interactivity, we learned all about one half of the form scheme: how to write the HTML code that describes a form interface and how the form sends its data. Unfortunately, you can't do much with a form without having some sort of program that can accept the data that is passed by a form and do something with it. To clear the air about using form data, there is some good news and some bad news. The bad news: The form data is sent in an encoded data block that can be a pain to decode into its component parts. The good news: This is such a common task among CGI scripters that people have already written a number of form-decoding utilities. A useful little collection of C functions comes with NCSA's HTTPD; you can just plug these functions into your C programs, and voilà! the task is done. For those of you who are not C programmers, there are also plenty of utilities that can be used with shell scripts, TCL scripts, and most of the common scripting and programming languages.

What Does Form Data Look Like?

There are two methods that a form can use to pass data to a script, GET and POST. As a quick guideline, use POST whenever possible, and use GET only for indexes and single-parameter forms.

When a form sends its data using the GET method, the data is encoded and stored in the environment variable QUERY_STRING. With the POST method, data is sent along through the standard input stream of the script. (By standard input, I mean what would normally be typed in from the keyboard. The HTTPD server redirects that data to the script as if it were being typed in.) In either case, the string is URL encoded. All variables and their values are paired together with equals (=) signs, then all of the name/value pairs are concatenated and separated with ampersands (&). The spaces are replaced with plus (+) signs, and the special characters are escaped. (Backslashes don't work here as they did in shell scripts); in this context, "escaped" means the character is represented by a percent sign (%) followed by the hexadecimal ASCII representation for that character.)

NOTE: Some definitions may be useful here. American Standard Code for Information Interchange (ASCII) is a code assigning unique numbers to the standard printable and control characters. This code can be read by virtually any computer in operation today. Hexadecimal, or base 16, is a numbering system using 16 as the base instead of 10. Numbers from 10 to 15 are represented by the first five letters of the alphabet.)

Encoding example: If a form had the following text input fields in its description: <INPUT TYPE="text" NAME="VAR1"> and <INPUT TYPE="text" NAME="VAR2">, and the strings typed into these text fields were "Troy Downing" and "Boo{TAB}Radley", the resulting encoded string would look like:

```
VAR1=Troy+Downing&VAR2=Boo%09Radley
```

To interpret the contents of the string, you would want to parse it into name/value pairs, replace the "+" with a space, and replace %09 with the {Tab} character. Simple enough? The following code has a number of C routines to do just that. Actually, the following code has a number of useful procedures that can be plugged into your CGI applications. These procedures generate common headers, simple HTML pages, and retrieve cookies.

cgiLib.c

```
/* cgiLib.c

   Troy Downing
   719 Broadway, 12th Floor
   New York, NY 10003
   (212) 998-3208

   downing@nyu.edu

   This is a library of common cgi decoding functions. It is meant to
be compiled and linked into most cgi applications. Feel free to
redistribute this source code, as long as this header remains as part
of the file. If you have any optimizations, bugs, or suggestions,
please send me email at the address above.

   cgiLib.c Copyright 1995, 1996 Troy Downing

*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "cgiLib.h"

/* this will decode all post data, and return a linked list
   This will only decode data coming in via standard in, so, it is
```

```
only good for decoding form data that was submitted with a "POST"
method.
*/
node_t* getcgidata() {

   char *buffer; /* tmp space for extracting url data */
    node_t *node, *root; /* the root, and actual nodes of the linked
list */
    int cont_len; /* length of form data, is decremented as data is
extracted */
    int first=0; /* used to determine if we have looped through this
yet */

   cont_len=atoi(getenv("CONTENT_LENGTH")); /* how much data? */

     while(cont_len) { /* loop through as long as there is still
cont_len */
      if(!first) { /* see if this is the first time */
        root = node = (node_t*)malloc(sizeof(node_t)); /*assign a
root*/
        first=1;
      } else {
           node->next = (node_t*)malloc(sizeof(node_t));
           node = node->next;
      }
      buffer = (char*)fmakeword(&cont_len); /* break the data block at
the first & */
      node->name=makeword(buffer); /* assign name to the name field */
      node->value=buffer; /* assign the data to the value field */
      plustospace(node->value); /* turn all + to spaces */
      unescape_url(node->value); /* fix the hex digits */
   }
   node->next=NULL; /* make sure this is the last node */
   return root; /* return the root of the linked list */

}

/* return a substring from stdin that is up to the next &. This
is what divides the urlencoded data stream into key=value chunks.
This will return a string that is the next key=value set in the stdin
stream.
*/
char *fmakeword(int *cl) {

       char stop='&'; /* the character that is used to delimit the
key=values */
    long wsize; /* the length of the data */
    char *word; /* the chopped data */
    int ll;    /* a counter */

    wsize = 32000; /* set the default size of the data */
    ll=0; /* set the counter */
```

Continued on next page

Continued from previous page

```
    word = (char *) malloc(sizeof(char) * (wsize + 1)); /* create the
data block to be returned. */

    while(1) { /* grab characters, increment the counter, and look for
the ampersand */
        word[ll] = (char)fgetc(stdin);
        if(ll==wsize) {
            word[ll+1] = '\0';
            wsize+=102400;
            word = (char *)realloc(word,sizeof(char)*(wsize+1));
        }
        --(*cl);
        if((word[ll] == stop) || (feof(stdin)) || (!(*cl))) {
            if(word[ll] != stop) ll++;
            word[ll] = '\0';
            return word;
        }
        ++ll;
    }
}

/* divide a key=value string into a key and a value */
char *makeword(char *line) {
    char stop = '=';
    int x,y;
    char *word = (char *) malloc(sizeof(char) * (strlen(line) + 1));

    for(x=0;((line[x]) && (line[x] != stop));x++)
        word[x] = line[x];

    word[x] = '\0';
    if(line[x]) ++x;
    y=0;

    while(line[y++] = line[x++]);
    return word;
}

/* convert all pluses '+' to spaces */
void plustospace(char *str) {
    register int x;

    for(x=0;str[x];x++) if(str[x] == '+') str[x] = ' ';
}

/* convert escaped characters of the form %XX where XX is a hex number
   representing an ASCII character value. This will convert the escape
   sequence back into the character that it represents
*/
void unescape_url(char *url) {
    register 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';
}

/* getval is sort of the equivalent to an associative array in Perl.
(${"keyname"}). This will take a linked list of name/value pairs and a
key. It will search for the key in the list, and return a value if the
key is found. Otherwise, it will return null.
*/
char* getval(node_t* node, char* name){

        while(node!=NULL) {
           if(node->name!=NULL)
             if(!strncmp(name,node->name,strlen(name)))
                return(node->value);
           node=node->next;
        }
        return NULL;
}

/* simple html header generator */
void htmlheader(char* title) {

        printf("Content-type: text/html\n\n");
        printf("<HTML><HEAD><TITLE>%s</TITLE></HEAD>\n",title);
}

/* closes most html pages */
void htmlfooter(){

        printf("</BODY></HTML>\n");
}


/* Generate a simple html page. */
void simplepage(char *title, char *message) {

        htmlheader(title);
        puts(message);
        htmlfooter();
}

/* print a list of all of the name/val pairs */
void printnamelist(FILE *fp, node_t *node) {

        while(node!=NULL) {
```

Continued on next page

Continued from previous page

```
            if(node->value!=NULL)
                fprintf(fp,"%s = %s\n",node->name,node->value);
            node=node->next;
        }
}
/* print a list of all of the values for name/val pairs */
void printlist(FILE *fp, node_t *node) {

        while(node!=NULL) {
            if(node->value!=NULL)
               fprintf(fp,"%s\n",node->value);
            node=node->next;
        }
}


/* generate a simple error page */
void errorpage(char* message) {

        htmlheader("Error!");
        printf("<h1 align=center>Error!</H1>\n%s",message);
        htmlfooter();
        exit(1);
}

/* set a full cookie. Be sure to add an extra carriage return after
the last call to a cookie.
*/
void setcookie(char* key, char *val, char *path, char *expire )
{
      printf("Set-Cookie: %s=%s; path=%s;
expires=%s\n",key,val,path,expire);
}

/* set up a simple cookie */
void setsimplecookie(char* key, char* val){

        printf("Set-Cookie: %s=%s\n",key,val);
}

/* generate a simple 1-cookie html header */
void cookieheader(char* title, char* key, char* val) {

        printf("Content-type: text/html\n");
        if(key!=NULL)
           printf("Set-Cookie: %s=%s\n\n",key,val);
        else
           printf("\n");

        printf("<HTML><HEAD><TITLE>%s</TITLE></HEAD>\n",title);
}
```

```
/* convert a hex number to a character. */
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);
}

void httpheader(void){
        printf("Content-type: text/html\n");
}

/* this will build a linked list of name/value pairs from
   the cookie environment variable
*/

node_t* getcookiedata(void) {

        char *buffer;
        char *buffer2;
        node_t *list;
        node_t *node;
        char cookie[BUFSIZ];
        int first=TRUE;

        list=NULL;

        if(getenv("HTTP_COOKIE")==NULL)
            return NULL;


memcpy(cookie,getenv("HTTP_COOKIE"),strlen(getenv("HTTP_COOKIE")));

        if(cookie!=NULL){
           while(TRUE){
              if(first){
                list=node=(node_t*)malloc(sizeof(node_t));
                first=FALSE;
                buffer=strtok(cookie,";");
                } else {
                node->next=(node_t*)malloc(sizeof(node_t));
                  node=node->next;
                buffer=strtok(NULL,";");
                }
              if(buffer==NULL) break;

                if(buffer[0]==' ') {
                  buffer2=buffer+1;
              }
```

Continued on next page

Continued from previous page

```
                else
                    buffer2=buffer;

                  node->name=makeword(buffer2);
                  node->value=buffer2;
            }
        }

        return list;
}
```

The following is the header file that goes along with cgiLib.c.

```
/* cgiLib.h

   Troy Downing
   719 Broadway, 12th Floor
   New York, NY 10003
   (212) 998-3208

   downing@nyu.edu

   cgiLib.h Copyright 1995, 1996 Troy Downing

*/

#ifndef __CGILIB_H
#define __CGILIB_H
struct url_node {
        char *name;
        char *value;
        struct url_node* next;
};
typedef struct url_node node_t;

#ifndef TRUE
enum {FALSE, TRUE};
#endif

node_t* getcgidata(void);
char* fmakeword(int* cl);
char* makeword(char* buff);
void plustospace(char* buff);
void unescape_url(char* url);
char x2c(char* c);
void printlist(FILE *fp, node_t*);
void printnamelist(FILE *fp, node_t*);
char* getval(node_t* node, char* name);
void setcookie(char* key, char *val, char *path, char *expire );
void httpheader(void);
node_t* getcookiedata(void);
void htmlheader(char* title);
void setsimplecookie(char* key, char* val);
void htmlfooter(void);
void simplepage(char* title, char* message);
void errorpage(char* message);
#endif
```

Right now, you're probably thinking, "Wow, what a mess! What am I supposed to do with all of that stuff?" Well, don't worry; you don't have to look at the cgiLib.c code again. All we will do is compile it into an object file once, then link it to whatever programs we write that use its functions. Before we jump in, lets get familiar with the prototypes in the header file. The following prototypes are extracted from the cgiLib.h file in the previous code listing. The prototypes here will be followed by brief explanations of their functions, and the parameters that they require.

HTML `node_t* getcgidata(void);`

The getcgidata() function is probably the most important one in this list. This function is sort of the magic "black box" that will turn a huge block of urlencoded data into a list of names and values that can be easily accessed in your programs. This function returns a node_t structure that is also defined in this header. file. This structure is used in other functions later on for accessing the form data.

HTML `char* getval(node_t* node, char* name);`

This getval() function is probably the second most useful in this library. The getval() function takes a node_t structure such as the one returned by the getcgidata() function, and a character string representing a name. It will return a value that corresponds to the name that is passed as a parameter.

HTML `void printlist(FILE *fp, node_t*);`

The printlist() function is used to print all of the values that were passed in as either urlencoded data, or in the form of cookies. It requires a file pointer and a node_t structure such as the one returned by getcgidata(). The file pointer that is passed will often be the special "stdout" pointer for printing the results directly to standard output.

HTML `void printnamelist(FILE *fp, node_t*);`

The printnamelist() function is exactly the same as printlist() with one exception—the names of the name/value pairs are printed as well as the values.

HTML
```
void setcookie(char* key, char *val, char *path,
    char *expire );
```

The setcookie() function is used for setting html cookies in a Browser.

HTML `void httpheader(void);`

This prints a simple HTML header.

```
void htmlfooter(void);
```

This prints a standard HTML closing tag.

```
node_t* getcookiedata(void);
```

The getcookiedata() function is similar to the getcgidata() function, except it returns a node_t structure that contains name/value pairs that were passed as cookies rather than as urlencoded data. The getval() function may be used to retrieve that data.

```
void htmlheader(char* title);
```

The htmlheader() function prints a standard HTTP header followed by an HTML header with the title of the document set to the value passed as the "title" parameter.

```
void setsimplecookie(char* key, char* val);
```

This sets a simple cookie name/value pair on a Browser.

```
void simplepage(char* title, char* message);
```

The simplepage() function takes a title and a message and will generate a complete HTML document with those parameters.

```
void errorpage(char* message);
```

The errorpage() function is similar to the simplepage() function, but is meant for sending error messges to the user's browser.

```
char* fmakeword(int* cl);
```

fmakeword is used internally by the cgiLib functions. There is really no nead to access it in your programs. This is used mainly by the getcgidata() function.

```
char* makeword(char* buff);
```

makeword() is also used internally by the getcgidata() function. In general, this function divides the name/value pairs into their component parts.

```
void plustospace(char* buff);
```

plustospace() is an internal function that converts all pluses (+) to spaces.

```
void unescape_url(char* url);
```

unescape_url() is an internal function that turns escaped characters of the form %XX and turns them into the ASCII character that they represent.

HTML `char x2c(char* c);`

x2c() is used internally by the unescape_url() function.

So, in general, the functions in the cgiLib.c file do all of the dirty work associated with CGI applications programming. There are functions for decoding urlencoded CGI data, grabbing cookie data, setting cookies, and generating simple HTML pages. Now, before we compile this library, let's take a quick look at what it's doing.

Obviously, the most useful function in this library is the getcgidata() function. (Assuming we are decoding form data.) So, what exactly is this function doing for us? Well, the quick answer is that it is taking a block of data that looks something like:

```
fname=Troy+B.&lname=Downing&add=719%0DBroadway&phone=555+1212
```

and turning it into something useful like:

```
fname   Troy B.
lname   Downing
add     719 Broadway
phone   555 1212
```

So, let's think back about HTML forms for a second. A form urlencodes it's data before sending it to a CGI application for processing. So, all field names and their values have been stuck together and separated by an equals (=) sign, all of these name/value pairs have been stuck together and separated by an ampersand (&), all spaces have been converted to plus signs (+), and any special characters have been escaped to a hexidecimal representation of the form %XX.

Our getcgidata() function undoes all of that mess and gives us a nice linked list of the name/value pairs. In the linked list, we can search for a name, and we should be able to find the value that was associated with it. (The getval() function does just that.) So, let's take a look at this function:

```
node_t* getcgidata() {
```

All this tells us is the name of the function and that it returns a pointer to a node in our linked list. (Don't worry if you don't understand this, the other functions of this library work with this structure directly.)

```
char*  buffer;
node_t *node, *root;
int cont_len;
int first=0;
```

That was simple enough, just defining some variables that this function will use.

```
cont_len=atoi(getenv("CONTENT_LENGTH"));
```

Now, you may recall that all data from a form using the POST method will be sent to our CGI application via the standard input. Well, we need to know how much data to read from the standard input. Well, it just so happens that the server will set an environment variable called CONTENT_LENGTH to the exact number of bytes that the urlencoded data block take up. So, if we grab this value, we know exactly how many bytes to read from standard input. Here we are assigning this number to the *cont_len* variable.

```
while(cont_len) {
```

Here we keep looping through our read routines while there is still data to be read. Every time we take some information from standard input, we subtract the number of bytes read from the cont_len variable. Once this number hits zero, we are through.

```
if(!first) {
/* ... */
}
```

This is just an initialization so that we create a new linked list node the first time we run through this loop.

```
buffer= (char*)fmakeword(&cont_len);
```

This function just grabs data from the standard input until it reaches an ampersand, or the end of the input stream. The cont_len variable is decremented by the number of bytes read.

```
node->name=makeword(buffer);
```

This will take the name/value pair that was stored in the buffer variable, and divide it at the equal sign (=). Then the first part, which is the name is stored to the node structure and the second part which is the value is stored in the buffer variable.

```
node->value=buffer;
```

This assigns the value that was in the buffer to the node.

```
plustospace(node->value);
```

This will convert all of the plus signs (+) back into spaces.

```
unescape_url(node->value);
```

This will turn all of the escaped characters back into themselves. That is, if there were any escaped characters in a particular value.

The rest of this function just loops through the previous functions until there is no more data to be read from standard input. Once this is complete, the linked list is terminated with a NULL pointer, and the root of the list is returned. So, an assignment such as:

```
node_t* list;
list = getcgidata();
```

will assign the linked list to the list variable which is then ready to be used in a call to *getval(list,"keyword")*. Pretty simple.

Now the best part of this whole thing. Now that you know how it works, you never have to look at it again. In general, you will never need to make changes to the functions in cgiLib.c, and can just plug them into any new CGI program that you write. Now, let's compile this library and start using it.

You should create a directory somewhere to contain the source code for your CGI scripts. Move the cgiLib.c and cgiLib.h files into this directory. Now, using your favorite C compiler, compile the cgiLib.c file into an object file called cgiLib.o. If you use the gcc compiler, all you have to type to create this object file is

```
gcc-o .cgiLib.c -o cgiLib.o
```

This would compile the code and save it in the same directory as cgiLib.c. OK, that's taken care of. Now, the best way to show how to use these functions is to demonstrate them in a CGI script example. The following program will create a Magic Eightball game on your server. For those of you who have never heard of the Magic Eightball, it is a black plastic ball full of blue liquid, with an icosahedron floating inside. The bottom of the ball has a window, in which one of the sides of the icosahedron is visible. There are 20 different answers to "Yes" or "No" questions written on the sides of the icosahedron. The user asks the Eightball a question, shakes the ball, and then turns it upside down to read its response.

In this version of the Eightball, the user types a question into a form on their Web browser. A click of the Submit button encodes the question and sends it to a server. The server randomly picks one of the 20 replies that would normally be written on the icosahedron, and returns an HTML document with the answer. This CGI script also adds the question and answer to a log file so that users can select the "log" page and look at all of the extremely useful advice that the Eightball has given.

Forms generally have two parts, an HTML part and a CGI script part. This example lists the CGI script first since that's what we're working on here, then it gives a few notes on how to compile the script, and finally lists the corresponding HTML form description.

```
#include <stdio.h>
#include "cgiLib.h"
#define CHOICES 20
#define LOG          "/usr/httpd/logs/eightball_log.html"
void main(void) {
        node_t* cgidata;
        unsigned char rn; /*random number */
        FILE *questions; /* log file */

        static char *message[]={
           "Yes",
           "No",
           "Maybe",
           "My Reply is Yes",
           "Reply Hazy, Try again",
           "Concentrate and ask again",
           "Definitely",
           "Signs point to YES",
           "Ask again later",
           "Without a doubt",
           "It is certain",
           "Outlook not so good",
           "My reply is No",
           "Don't count on it",
           "Outlook good",
           "Most Likely",
           "Very doubtful",
           "My sources say no",
           "You may rely on it",
           "It is decidedly so"
        };

        if(!(questions=fopen(LOG,"a")))
            errorpage("Can't open log file"); /* catch an error*/

        /* seed the random generator */
        srand(getpid());

        /*get a random number between 0 and 19 */
        rn = rand()%20;

        cgidata = getcgidata(); /* now wasn't that simple? The
                                   previous statement just
                                     urldecoded all of the data and
                                          stored it in cgidata */
        /* now for the content... First, let's write to the log*/

        fprintf(questions,"<PRE>%s </PRE>",
                           getval(cgidata,"question"));
        fprintf(questions,"<B> %s</B><HR>\n",message[rn]);
        fclose(questions);

        /* now let's generate the response page */

        htmlheader("Oracle Response"); /* print a header */
        printf("The Oracle has thought long and hard about ");
        printf("your question and has come to the following ");
        printf("answer:<BR>\n");
        printf("<H1>%s</H1>\n",message[rn]);

        /* finish off the HTML document */
        htmlfooter();
}
```

Note that you will have to change the path names and filenames to match the ones your system uses. For example, if you put the log file in a different location, you will have to change the HREF that points to it in the bottom of the file.

To install this file, compile it and link it with the cgiLib.o object file that we created earlier in this section. If you're following the example exactly, type

```
gcc eightball.c cgiLib.o -o /usr/httpd/cgi-bin/eightball
```

Note the path to the cgi-bin directory. This eliminates the step of moving the compiled program into the correct directory. If you compile your version locally, remember to either move it into the correct directory, or ask your system administrator to move it for you if you don't have write privileges to make changes there.

Now, let's take a look at the form that calls this script. Remember that you will have to change the machine name and path to match your own installation.

```
<HTML>
<HEAD>
<TITLE>Magic Eightball</TITLE>
</HEAD>
<BODY>
<H1>Troy's Magic 8-ball page</H1>
<IMG SRC="/shaggy.a/downing/public-html/8-ball.gif"><P>
<HR>
This is where one should turn for advice of critical
importance. To use this Oracle:
<UL>
<LI>Concentrate
<LI>Type in a yes or no question
<LI>Click on the "ask" button.
```

Continued on next page

Continued from previous page

```
</UL><BR>
You will receive a reply shortly.<BR>
<HR>
<H1> The Oracle can only answer "YES or NO" questions.</H1>
<HR>
<FORM ACTION="http://found.cs.nyu.edu/cgi-bin/downing/eightball"
METHOD="POST">
Type in your question:<P>
<HR>
<TEXTAREA NAME="question" ROWS=2 COLS=60></TEXTAREA>
<HR>
<INPUT TYPE="submit" VALUE="Ask 8-ball"><INPUT TYPE="reset"
VALUE="Clear Entry">
</FORM>
<HR>
<H5>I can't be held responsible for bad advice given
by this oracle</H5>
<H4>Accept no imitations! This is the Original WWW 8Ball and not a
cheap imitation!</H4>
<A href=/logs/eightball_log.html>Read Log</A>
</BODY>
</HTML>
```

To make this work, you must make sure that all of the hrefs are pointing to the path/names of the files you created on your system. The paths and filenames listed here are only examples. To see the Eightball in action in its original home, point your Web browser at:

```
http://found.cs.nyu.edu/downing/eightball.html
```

The Eightball example only deals with a single variable that contains the question string. What happens when we need to deal with several name/value pairs? Well, the following example works with the basic "feed-back" page form that was described in Chapter 14, which allows a user to submit comments via their browser. This version adds the comments to a database, but this could easily be changed to a mailer that would mail the results to the form's owner.

Here is the description of the database handler that goes with the form description in Chapter 14. It will take the form data, return an HTML page to the browser confirming receipt of the data, write the data to a database, and send e-mail informing the author that new data has been added.

```
#include <stdio.h>
#include <stdlib.h>
#include "cgiLib.h"

#define MAILER "/usr/lib/sendmail"
#define ADDRESS "user@somewhere.com"

void main(void){
        node_t* root;
        FILE *mail;
        char address[BUFSIZ];


        root=getcgidata();


        /* put the user response page together */
        htmlheader("Mailer Page");

        printf("Thanks for your submission.<BR>");
        printf("The contents of your submission follow:");
        printf("<PRE>\n");
        printnamelist(stdout,root);
        htmlfooter();

        /* construct the mail address */
        sprintf(address,"%s %s",MAILER, ADDRESS);
        mail=popen(address,"w"); /* open a pipe to mail */

        if(mail==NULL)
           errorpage("Couldn't open mailer\n");
           /* send a page if the mail pipe failed*/

        fprintf(mail,"Subject: WebMail!\n");
        /* set the subject line in the mailer */

        printnamelist(mail,root);
        /*print all of the cgi data to the mailer */

        fclose(mail); /* close the mail pipe */

}
```

Here's an exercise. To get an idea of what your encoded strings look like to your CGI scripts, write the following shell script:

```
#include <stdio.h>
#include <stdlib.h>
void main(int argc, char *argv[])
{

        int cont_len,index;
        char c;

        cont_len = atoi(getenv("CONTENT_LENGTH"));

        print("Content-type: text/html\n\n");
        printf("<listing>\n");
   for(index=0;index<cont_len;index++)
   {
        c = getchar();
```

Continued on next page

Continued from previous page

```
        printf("%c",c);
    }
}
```

Compile this program and place it in the cgi-bin directory. Now, using one of the forms you've created, or one of the forms described in this book, change the <ACTION> tag to point at this program. (You must use the POST method for this to work.) Now, when you submit a form that points to this script, it should return the entire encoded string that your form sent it. This can be a useful tool if your decoding algorithms aren't doing what you expect.

HTTP Cookies

Mmmmm Cookies! Well, um, actually, HTTP cookies aren't as tasty as one may hope. But that doesn't mean that they should be taken lightly. HTTP cookies are the single most powerful mechanism at a CGI programmer's disposal for maintaining client-side state. So, what is a cookie? The quick answer: "An HTTP cookie is persistent, client-side state, that is assigned through a standard HTTP header."

Well, what's so cool about that? Let's think about the standard HTTP transaction between a browser and a CGI application. First, the browser opens a connection to an HTTP server and requests a CGI application. The server locates the CGI application, executes it, sends it data, and takes its results from the CGI application's standard output. The server then forwards the output from the CGI application to the browser that requested it in the first place, and kills the connection.

Once the connection is closed, the CGI application has no way of keeping track of who had accessed it. The CGI application lies dormant until a request is made for its services, then it springs to life, and goes back to sleep. There is no persistence of state kept in the CGI application's memory, since it exits, and no longer has a connection to any particular client browser anyway.

In many circumstances, it is useful to keep track of what transactions a particular client has made. For instance, say you have an Internet shopping service. Part of the functionality of this service is that you'd like your clients to be able to shop around and add items to a virtual shopping bag. Once the client is done shopping, you want him or her to be able to "check out," or pay for the items in the virtual shopping bag.

Well, the problem is keeping track of items as they are added to the shopping bag. Your CGI application could generate a new HTML form with hidden input tags after every transaction, and these hidden tags could contain the names of items in the bag. Or you could keep a local database on your server that assigned a unique ID that was hidden in an HTML form and track items on your server. Or, use cookies.

Cookies allow you to store information on a client's browser that is not part of the HTML content. You can set the persistence of cookies to last throughout a single session, or for some arbitrarily large amount of time.

Many of you no doubt have seen customizable Web sites. These are Web sites where you select what information you want to see, and how you want to see it. Amazingly enough, when you come back to that same site the next day, it remembers all of the preferences you set. These work with cookies. The site maintains a database of your preferences and assigns your browser a unique ID number. This number is hidden in your browser as a cookie so anytime that you access the site, the server at the site can determine who you are, and what your preferences are.

So, how does it work? Simply, a cookie is set on a browser with a standard HTTP header. This works exactly the same as the mechanism we use to tell the server what type of content we are sending. The cookie data, in it's simplest form, is formatted as a name/value pair. To clear things up a bit, the following code bit is an HTTP header for an HTML document that will set a cookie name "fname" and the value of "Troy" on the client's browser:

```
Content-type: text/html
Set-Cookie: fname=Troy
```

You can have an arbitrary number of "Set-Cookie:" lines in a single HTTP header. The server distinguishes the header from the content by looking for a blank line seperating the two. This is why an HTTP header must always be followed by an extra blank line. For example, the following ouput would set a number of cookies, and then begin sending content:

```
Content-type: text/html
Set-Cookie: name=Bob
Set-Cookie: add=Smith Street
Set-Cookie: phone=555-1212

<HTML>
<HEAD>
...
```

Notice the extra line between the header and the content. If this line is omitted, this script will not work. The browser will be sent an error message of the form "Server Error 500", which doesn't mean a whole lot to anyone.

Relevance

The next interesting part of a cookie is setting its relevance. By default, the cookies that you set will only be sent to documents with the same base

URL of the CGI application that set the cookie. So, normally, only the site that set a cookie can get the data back. Sometimes, you may want to set the cookie's relevance to a specific path on your server. This is useful if you have a number of applications that may use the same names for cookies. This is done by adding a "path" attribute to the cookie header. The attributes in the header are separated by semi-colons (;). So, to set the path of a cookie to be relevant to /cgi-bin/downing/, you could use the following cookie header:

```
Set-Cookie: name=Troy; path=/cgi-bin/downing/
```

Once issued, this cookie would only be sent to CGI applications in the /cgi-bin/downing/ directory.

Persistence

By default, a cookie will remain in effect throughout the session that it was created in. In other words, the cookie is forgotten once the client's browser is restarted. So, how do you set the persistence? Well, there is an "expires" attribute that can be added to the cookie header. The expires attribute is in an odd format. The format is in the form Wdy, 11-Jun-98 12:15:00 GMT. More generically, "Weekday, DD-Mon-YR HH:MM:SS GMT", where the first part is the weekday, the DD is the day of the month, Mon is an abbreviation for the name of the month, YR is the last 2 digits of the year and HH:MM:SS are the hours, minutes, and seconds of the expiration. All times are recorded in Greenwich Mean Time (GMT). So, the following cookie header would remain persistent until June 11, 1998 at 12:15 Greenwich Mean Time:

```
Set-Cookie: name=Troy; expires=Wdy, 11-Jun-98 12:15:00 GMT
```

Retrieving Cookie Data

So, now you know how to set cookies, the obvious question is: "How do I get the data back from the browser?" All relevant cookie data is automatically sent to a CGI application through an environment variable called "HTTP_COOKIE". The form is "key=val; key=val; ..." for an arbitrary number of keys. So, getting cookie values is as simple as reading the HTTP_COOKIE environment variable and parsing it into name/value pairs.

Using the cgiLib Library with Cookies

The cgiLib.c library has a number of functions for setting and retrieving cookie values. Setting cookies can be as simple as calling a setcookie() function. The setcookie() function takes four parameters representing the name, value, path, and expiration of a cookie. Here is a code sample for setting a cookie with the library:

```
httpheader();
setcookie("name", "Troy", "/cgi-bin/downing/","Wdy, 11-Jun-98 12:15:00
GMT");
printf("\n");
```

Notice that the httpheader() function was issued first to print a "Content-type" header, and that the entire HTTP header was followed by a printf("\n"); to print an extra line after the header. Often, you only want to set a temporary cookie, in which case you can use the setsimplecookie() function with a name and value parameter. For example:

```
httpheader();
setsimplecookie("Name", "Troy");
setsimplecookie("Phone","555-1212");
printf("\n");
```

There is one other simple function that will create an HTTP header and set a cookie value all in one fell swoop: cookieheader(). This is sent a document title parameter, and a cookie name and value. This function will automatically print the "Content-type" header, and follow the cookie with the extra blank line. For example:

```
cookieheader("Cookie Page", "name", "Troy");
/* content follows*/
```

So as you can see, setting cookies is pretty simple. Extracting cookies is almost identical to extracting the urlencoded CGI data. The getcookiedata() function returns the root of a linked list that can be used in any of the getval(), printlist(), or printnamelist() functions. So, here is a code fragment that gets the cookie data from the HTTP_COOKIE environment variable, parses it into a linked list, and prints the value of the cookie named "fname":

```
node_t * cookie;
cookie = getcookiedata();
htmlheader("test page");
printf("The value of the cookie fname = %s\n", getval(cookie,"fname));
htmlfooter();
```

Couldn't be simpler. Instead of using the getval() function in a printf() statement, we could have printed all of the cookie name/value pairs with the following statement:

```
printnamelist(stdout,cookie);
```

The printnamelist() function takes a stream or file handle, and the root of a linked list. It then traverses the linked list and prints out all of the names and values that it contains.

Finally, here is a simple program that sets a number of cookies and then prints their values:

```
#include <stdio.h>
#include "cgiLib.h"
void main(void) {
    node_t* cookie;
    cookie=getcookiedata();
    httpheader();
    setsimplecookie("name","Troy");
    setsimplecookie("add", "719 Broadway");
    setsimplecooke("phone", "555-1212");
    printf("\n");
    printf("<HTML><HEAD><TITLE>Cookie Test</TITLE></HEAD>\n");
    printf("<BODY>\nThe submitted cookies follow:<PRE>/n");
    printnamelist(stdout,cookie);
    printf("<\PRE>\n");
    htmlfooter();
}
```

As you can see, cookies are a powerful mechanism for storing information on a user's browser. They are also very simple to use, especially if you use the prepackaged functions listed above. Common uses for cookies are storing password information, keeping track of a particular user (look in your browser's cookie database on your local machine, you may be surprised to see how many services have assigned you an ID number so that they can keep track or your browsing habits), or keeping track of items that you may select during a shopping spree. The applications are enormous, and I'm sure you'll come up with your own intriguing requirements for client-side state tracking.

LESSON #7: SAMPLE SCRIPTS FOR UNIX, WINDOWS, AND MACINTOSH SERVERS

This final section offers a number of educational, useful, and/or interesting CGI scripts, with something for whatever platform you're likely to be using. Most of them come from public archive sites around the Internet community, and their authors deserve great thanks for making them publicly available.

Unix

Unix is generally the CGI programmer's operating system of choice. Most of the more intriguing scripts seem to have been written for Unix servers. This doesn't mean that they can't be revised to run on other servers; in fact, many of them can be modified to run on other platforms with very little work. It may take some doing to convert a shell script to an AppleScript program, but the C programs should port quite nicely. Note that any C programs that are listed below that contain "#include "cgiLib.h"" must be linked with the cgiLib.c functions listed earlier in this chapter. Well, enjoy!

fax_mailer.c

```
/*      This works with a specific fax-modem terminal that
        the author runs locally. The fax modem takes e-mail as
        its input. This may need to be modified to work on your
        system.

        HTML Fax Utility. Should be run as a cgi file under
        HTTPD. Takes the variable string supplied from the
        HTML "Form" submittal and parses it into an e-mail
        address.... The order that the variables appear in the
        html form is important. They must appear in the form
        in the same order that they are listed in the defines
        below, that is, AT..FROM. (Not the best approach for the
        job; it'll get fixed at some point.) This will produce
        an e-mail message of the form:

        /FN=995-4122/AT=Troy_Downing/O=NYU/@text-fax.nyu.edu

        text-fax.nyu will parse the "to" line and create a fax
        cover page and attempt to fax it to the number listed
        after /FN=. Currently, all spaces are converted to
        underscores in the address but not in the body.

        The fax is mailed from nobody@yourserver.com and this
        will appear on the header as the sender. So, Its
        important to include the FROM string so that the
        recipient will know who it came from....

*/

#include <stdio.h>
#include <stdlib.h>
#include "cgiLib.h"

#define MAILER  "/usr/bin/mail"
#define LOGFILE "/usr/logs/fax.log"

#define FAX     "@text-fax.myserver.com"

void main(int argc, char *argv[])
{
       node_t* data;
       char address[256];
       FILE *mdata, *log;

       /* get form data */
       data = getcgidata();
```

Continued on next page

Continued from previous page

```
        /* put together e-mail address. The getval() function is
        passed the linked list that we created with the getcgidata()
        function, and then the names of the input fields are passed to
        retrieve the values that were input in the html form */
        sprintf(address, "%s -s \"%s\" \"/FN=%s/AT=%s/O=%s/OU=%s/\"%s",
                MAILER, getval(data,"subject"), getval(data,"FaxNo"),
                getval(data,"addressee"),getval(data,"Organization"),
                getval(data,"address"),FAX);

        /* open a pipe to the mailer */
        if (!(mdata=popen(address,"w")))
                errorpage("Couldn't open the mailer");

        /* Send the mail message to the fax machine*/
        fprintf(mdata,"%s\n", getval(data,"body"));
        fprintf(mdata,"Message Sender: %s\n",
                getval(data,"from"));
        pclose(mdata);

        /*write the data to the log file*/
        log=fopen(LOGFILE, "a");
        fprintf(log,"----------\n");
        printnamelist(log,data);
        fprintf(log,"----------end--\n");
        fclose(log);

        /* send a response page to the user */
        htmlheader("Fax Sent!");
        printf("<h1>Mail sent!</h1>\n" );
        printf("content follows:<p><hr><PRE>\n");
        printnamelist(stdout,data);
        printf("</PRE>");

        /* end html document */
        htmlfooter();


}
```

fax_mailer.html

```
<HTML>
<HEAD>
<title>FAX Utility</title>
</HEAD>
<BODY>
<h1>FAX Mailer</h1>
<hr>
This form will send a FAX to the fax number/recipient that is
supplied. Currently it will only work with area codes 212 and 718. If
```

```
you haven't used this before, please read the <a
href="http://found.cs.nyu.edu/downing/fax_instruct.html">instruc-
tions</a>.<p>
***REQUIRED fields are marked with a *<p>
<hr>

<form   action="http://found.cs.nyu.edu/cgi-bin/downing/fax"
METHOD="POST">
<pre>
To:*            <input type="text" name="addressee" size=30><p>
Sub:*           <input type="text" name="subject" size=30><p>
Fax number:*    <input type="text" name="FaxNo" size=15><p>
Organization:   <input type="text" name="Organization" size=30><p>
Address:        <input type="text" name="address" size=30><p>
</pre>
<hr>
Body:<p>
<TEXTAREA name="body" COLS=80 ROWS=10></TEXTAREA>
<p>
<hr>
From:<input type="text" name="from" size=35><p>
<input type="submit" value="submit">
</form>
</BODY>
</HTML>
```

mailer.c

This next script will send e-mail submitted via a form. It can be useful if users can't use mailto: URLs with their browsers.

```
/* This is a simple mail program. It uses the cgiLib library
   and is configurable in the html, so it's useful for a number
   of users at a site. It expects a form such as:

   <FORM ACTION=http://foo.com/cgi-bin/mailer/downing@foo.com>
   <INPUT name="email" >
   <INPUT ...>
   ...
   <FORM>

   there can be an arbitrary number of input elements in the
   form. If one of the input fields has the name "email", then
   the contents of the field will be used as a return address,
   otherwise the server's address will be used. The email
   address that will be sent the results of the form submission
   is appended to the URL in the ACTION attribute. This makes it
   possible for a number of users to use this same program by
   only changing the URL in the ACTION attribute to point to
   their particular email address
*/

#include <stdio.h>
#include <stdlib.h>
#include "cgiLib.h"
```

Continued on next page

Continued from previous page

```
#define MAILER "/usr/lib/sendmail"

void main(void){

        node_t* root;
        FILE *mail;
        char address[BUFSIZ];


        root=getcgidata();


        /* put the user response page together */
        htmlheader("Test Page");
        printf("<BODY><PRE>");
          printf("Thanks for your submission %s<BR>\n",
            getval(root, "fname"));
          printf("We have your email address as: %s<BR>\n",
            getval(root,"email"));
          printf("The contents of your submission follow:\n");


        printnamelist(stdout,root);
        htmlfooter();

        /*construct the mail address */
        sprintf(address,"%s %s",MAILER, getenv("PATH_INFO")+1);

        /* if an email address wasn't provided, send an error*/
        if(getenv("PATH_INFO")==NULL)
         errorpage("You must supply an email address \
                   appended to the action url. Otherwise, \
                   this mailer has no idea who to mail to...");

        mail=popen(address,"w"); /* open a pipe to mail */

        if(mail==NULL)
           errorpage("Couldn't open mailer\n");
           /* send a page if the mail pipe failed*/

        fprintf(mail,"Subject: WebMail!\n");
        /* set the subject line in the mailer */

        if(getval(root,"email")!=NULL)
        /* if email is a var, make that the ret. add*/
              fprintf(mail,"Reply-to: %s\n",
                      getval(root,"email"));

        printnamelist(mail,root);
          /*print all of the cgi data to the mailer */
        fclose(mail); /* close the mail pipe */

}
```

mailer.html

```
<HTML>
<HEAD>
<TITLE>Mail Interface</TITLE>
</HEAD>
<BODY>
<H1>Mailer</H1>
<P>Please send us a message:</P>
<FORM ACTION="http://foo.com/cgi-bin/mailer/downing@foo.com"
      METHOD="POST">
<PRE>
Name:   <INPUT name="name">
Email:  <INPUT name="email">
<TEXTAREA rows=10 cols=60 name="message">
Type your message here
</TEXTAREA>
<INPUT type="submit">
</PRE>
</FORM>
</BODY>
</HTML>
```

names.c

```
/* This program parses a form that contains 3 input fields: a
   name suggestion for my expected new baby, a sex which could
   be male/female/either, and a comment
   the suggestions are added to a database that I'm keeping
   until I actually have to name the kid.

   addendum: The baby was born 3-25-95 and named Morgan
*/

#include <stdio.h>
#include <stdlib.h>
#include "cgiLib.h"

#define NAMES           "/usr/pub/names"

void main(void)
{

       char address[256];
       FILE *names;
       node_t* data;

     data=getcgidata();

       names = fopen(NAMES, "a");

      htmlheader("Name Submission");
```

Continued on next page

Continued from previous page

```
        fprintf(names,"%s\t\t%s\t%s\n",getval(data,"name"),
                getval(data,"sex"),getval(data,"comment"));
        fclose(names);

        printf("<h1>Thanks for your suggestion!</h1> \n");
        printf("<h2>I will take the name \"%s\" into serious \
            consideration!<p>",getval(data,"name"));
        printf("<a href=\"http://foo.com/pub/names\">");
        printf("List of names</a><p>");

          htmlfooter();

}
```

names.form

```
<form action="http://myserver.com/cgi-bin/names" METHOD="POST">
Please help suggest a name. Enter a name, comment if you want, and
sex.<BR>
name:<input type="text" NAME="name" size=15>
<select name="sex">
 <option>Male
 <option>Female
 <option>Either
 </select>
comment:<input type="text" name="comment" size=30>
<input type="submit" value="submit name">
<a href="/usr/pub/names">List of names so far...</a>
</form>
```

Easy Counter

Most Web servers support server-side programs, which are a way of creating small bits of HTML on-the-fly. These are similar to cgi scripts, but they do not need to be called via a form or hyperlink—they can be called from the source of your HTML document.

Server-side scripts are a marvelously easy way of adding a *counter* to your Web page. A counter is a simple program that keeps track of how many people have visited your site. Whenever your Web page is loaded, the counter is incremented. You can then print a message onto your page similar to:

```
1,238 folk have visited this Web page since January 1, 1996.
```

STYLE TIP: Counters are highly over-used. Given that warning, you're still welcome to use one but be aware that some people find them annoying.

To add a counter to your Web page, split your HTML document into two documents. The first file should contain all the HTML markup you want to appear before the counter appears. The second file should contain the rest of the markup.

For instance, you might have two files as follows:

file1.html

```
<HTML>
<HEAD>
<TITLE>My counter Page</TITLE>
</HEAD>
<BODY>
A counter page. <P>
<HR>
```

file2.html

```
folk have visited this Web page since January 1, 1996.<P>
<HR>
Pretty neat, eh?
</BODY>
</HTML>
```

Now create a text file which contains the initial counter value (i.e., 0). This file shold be named *count.txt*.

count.txt

```
0
```

NOTE: Make sure that all the files are readable and writable by the Web server. Use the chmod 644 command to give the file the correct mode. For example:

```
chmod 644 counter.txt
```

You'll now use a simple server-side program to stitch these three files together. The C code for such a program is as follows:

```
/**********************************************
counter.c:
A script to add a counter to your Web page.
**********************************************/
#define HTML_HEADER     "(entire local path to)/test.head"
#define HTML_COUNTER    "(entire local path to)/test.count"
#define HTML_FOOTER     "(entire local path to)/test.foot"

void    print_file(char *file)
{
        FILE    *fp = fopen(file,"r+");
        char    line[1024];
        line[0] = '\0';
        while (fscanf(fp, "%[^\n]s", line) != EOF)
        {
    fgetc(fp);
    printf("%s\n", line);
    line[0] = '\0';
        }
        fclose(fp);
}
```

Continued on next page

Continued from previous page

```
void    increment_counter(char *file)
{
             int     counter;
             FILE    *fp = fopen(file,"r+");
             fscanf(fp, "%d", &counter);
             /* Increment the counter */
             counter++;
             /* Save the new value to the count.txt file. */
             fseek(fp, 0L, 0);
             fprintf(fp, "%d\n", counter);
             fclose(fp);
             /* Spit out the count number. */
             printf("%d", counter);
}

void main(int argc, char **argv)
{
       printf("Content-type: text/html\n\n");
       print_file("file1.html");
       increment_counter("count.txt");
       print_file("file2.html");
}
```

You can compile the program (assuming you're using the Unix *cc* compiler) as follows:

```
cc counter.c -o /cgi-bin/counter.html
```

Modify the output path, if necessary, so that the output is put in your server's *cgi-bin* directory. You can name this file anything you want, *counter.html* is just a suggestion.

To test the file, just run the *counter.html* file by typing its name. The correct HTML output should appear on your screen if everything seems fine. You can now have people access your counter directly. For example, if your cgi-bin directory is at: *http://www.smartypants.com/cgi-bin* then your new counting Web page can be accessed at: *http://www.smartypants.com/cgi-bin/counter.html*

If you wish, you can create an alias so that when people seem to be accessing a Web page from your regular Web directory they'll actually be accessing the *counter.html* program in the cgi-bin directory.

Move to the *config* directory of your Web server and edit the *srm.conf* file. Add this line to the ScriptAlias section:

```
ScriptAlias /mypage.html /cgi-bin/counter.html
```

The first value should be the directory where your Web pages are stored. You can change *mypage.html* to any name you'd like people to access. The second value should be the actual cgi-bin directory on your Web server.

Once your server is restarted, your counter page will be accessed whenever somebody calls *http://www.smartypants.com/mypage.html*

Server Push

Most late-model browsers, such as the Netscape Navigator and Microsoft's Internet Explorer, allow dynamic content in the form of a "Server Push" or a "Client Pull." What the Server Push allows you to do is send a series of objects to the client rather than a single one. Normally, a CGI script sends a single type of data, say a text/html document or a GIF image and once this object has been passed on to the client, the connection is broken. With a Server Push, the connection is left open while the server sends a series of objects and is not closed until the script terminates. This is particularly useful using the multipart/x-mixed-replace MIME type with graphic images. This will allow you to send a graphic image to the client browser and then immediately replace it with another. By stringing a series of images like this together, you can create a sort of inline animation on the client's browser without using an external "helper application."

The Client Pull is similar to the Server Push but rather than having the server send another object, the client requests a series of objects after a specified interval. This comes in handy if you want the client to keep checking a document or script for changes automatically and for orchestrating "guided tours" without requiring user intervention. The Client Pull is implemented as an HTML tag that is interpreted by the client's browser. An example of a Server Push and a Client Pull follow.

Slide Show Animation—The following is the code for an inline animation script. It will send a series of GIF files to a browser, replacing each image with a subsequent one. It can easily be modified to use JPEG or XBM images as well. To work as intended, it should be called with an HTML document similar to the one that follows.

movie.c

```
/*      Multipart-mixed cgi demo.

        Troy Downing 1995

        This was written to demonstrate the multipart/mixed-replace
        capabilities that are now available with HTML 3.0 compliant
        browsers.

        This will send a series of images to a browser. Each image
        will replace the previous image giving the illusion of an
```

Continued on next page

Continued from previous page

```
        animation. This will work best on fast networks with small
        images of the same size/resolution.

        Just to make things simple, I've named the images 1..8.gif;
        this could be easily modified to deal with other
        filenames/types.
*/


#include <stdio.h>


#define BOUNDARY     "--ThisRandomString\n"    /*marks beginning of file*/
#define ENDING       "--ThisRandomString--\n"/*marks end of file*/
#define HEADER  "Content-type:multipart/x-mixed
replace;boundary=ThisRandomString"
#define TYPE "Content-type: image/gif"     /*mime type of file*/
#define IMAGETYPE  "gif"                   /*filename suffix*/
#define IMAGEDIR   "/usr/downing/gifs/"    /*contains the images*/
#define NAMELEN    256
#define REPEAT     8                       /*number of images to send*/
#define BUF_SIZE   1024                    /*number of bytes to read at
                                           once*/


void main(void)
{
        FILE   *f_spew;        /*points to image files*/
        char   file[NAMELEN];  /*holds image file name for use with ↵
fopen()*/
        char   buffer[BUF_SIZE]; /*buffer for reading file*/
        int counter,count,tries;

        printf("%s\n\n",HEADER);       /*print the multipart header*/

        counter=REPEAT+1;
        while(counter--)               /*cycle through images*/
        {
           printf("%s",BOUNDARY);      /*print beginning boundary*/
           printf("%s\n\n",TYPE);      /*print mime type for image*/

           sprintf(file,"%s%d.%s",IMAGEDIR,counter,IMAGETYPE);
 /*construct filename*/
           while((f_spew = fopen(file,"r"))==NULL) /*open file*/
           {
               if(tries--<0) break;
           }
           while (!feof(f_spew))       /*send data while not EOF*/
           {
           count = fread(buffer, 1, BUF_SIZE, f_spew);
           fwrite(buffer, 1, count, stdout);
           }

           fclose(f_spew);
```

```
            printf("%s",ENDING);        /*print ending boundary*/
        }

}
```

Now, here is the HTML page that calls this program. I'm assuming that the code was compiled as "movie.cgi" and placed in the cgi-bin directory.

```
<HTML>
<HEAD>
<TITLE>Movie Test</TITLE>
</HEAD>
<body>
    The following box should show a series of images. Click the
"reload" button to restart it.<BR>
<CENTER>
<TABLE border = 10><td><img src = "http://yourserver.com/cgi-bin/
movie.cgi"></td>
</TABLE>
</CENTER>
</BODY>
```

The following is an example of a Client Pull. The Client Pull is specified with an HTML <META> tag. For more information on the HTML 3 specs, see Chapter 10. This page will automatically reload itself every 30 seconds.

```
<HTML>
<HEAD>
<TITLE> Client Pull demo </TITLE>
<META http-equiv="Refresh" content=30>
</HEAD>
<BODY>
The following is a picture of our lab. It will
refresh every 30 seconds.<BR>
<IMG SRC="HTTP://ourserver.com/cgi-bin/take_pic.cgi">
</BODY>
```

Perl

Perl is a very popular scripting language for creating CGI applications. Since it is beyond the scope of this book to teach you the "Practical Expression and Report Language" (PERL), here are some code examples in Perl for creating simple CGI applications.

In the cgiLib.c library earlier in this chapter, there were a number of common functions for creating simple html documents and decoding URL data. The following code example is an almost identical library, but designed to be included in Perl scripts.

Perl is much slower than compiled C code, but has one advantage. It has a very powerful regular expression model for easily manipulating strings. So, the getcgidata() function in Perl, as you will soon see, is much shorter and more elegant than its C equivalent. Well enough talk, here's the library:

cgiLib.pl

```
# cgiLib.pl
# Troy Downing
# downing@nyu.edu
#
# This is a package or core subroutines that are helpful when
# writing cgi applications

# the following line must be present for "require"
# to succeed
1;

# htmlheader prints out a standard header, and <HEAD>... tags.
# syntax htmlheader("title");
sub htmlheader {

   local($title) = @_;

   print "Content-type: text/html\n\n";
   print "<HTML><HEAD><TITLE>$title</TITLE></HEAD>\n";
}

# textheader prints out a text/plain http header
sub textheader {

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

# imageheader prints out a standard http gif header
#syntax imageheader("gif");
sub imageheader {

   local($type) = @_;

   print "Content-type: image/$type\n\n";
}

# htmlfooter prints out a standard html doc closure
sub htmlfooter {
   print "\n</BODY></HTML>\n";
}

# getcgidata will decode the urlencoded data from a form or
# query submission
# it will return the decoded name/value pairs as a alist.
# By default, it
# will determine how to get the data
# (ie POST or GET) by examining the
# REQUEST_METHOD environment variable.
# It can be forced to use either
# POST or GET by passing it the value "POST" or
#"GET" as a parameter. ie:
# %alist = getcgidata("POST");

sub getcgidata {

   local($method) = @_;

   if($method eq "") {     # check to see if a value was passed
      $method = $ENV{'REQUEST_METHOD'};
   }
   if (($method eq "POST") || ($method eq "post")) {
      read(STDIN, $buffer, $ENV{'CONTENT_LENGTH'});
   } else {
      $buffer = $ENV{'QUERY_STRING'};
   }

   @nameValue = split(/&/, $buffer);
        #break up the urlencoded block

   foreach $pair (@nameValue) {
      ($name, $value) = split(/=/, $pair);
      #divide the name/value pairs
      $value =~ tr/+/ /; #replace all + with a space
      $value =~ s/%([a-fA-F0-9][a-fA-F0-9])/pack("C",
                 hex($1))/eg;
      #decode any escaped hex characters
      if ($value ne "") {
       $form{$name} = $value;
      }
   }
   return %form;
}

# simplePage generates a simple html page.
# It expects a title and
# a message string to use when generating the page

sub simplePage {

        local($title,$message) = @_;

        htmlheader($title);
        print("$message\n");
        htmlfooter();
}
```

Phew! That was almost as ugly as the cgiLib.c file! Well, the nice thing about it is, like the cgiLib.c file, you don't ever need to look at it again. Just include it in any CGI programs that you write in Perl. Here is an example program written in Perl that generates a simple HTML page:

```
#!/usr/local/bin/perl
require 'cgiLib.pl';
htmlheader("Test page");
print "<H1>Whoopy! A generated page!\n";
htmlfooter();
```

That wasn't so bad now was it? Here's a more complicated example. This is the equivalent to the mailer.c program earlier in this chapter. This will parse nearly any HTML form input, and send it as e-mail to an address specified in the ACTION attribute as path information. (ie: ACTION=http://foo.com/cgi-bin/mailit.pl/downing@foo.com)

mailit.pl

```
#!/usr/local/bin/perl
#
# Troy Downing
# downing@nyu.edu
#
# This is a basic mail application. It is generic and can be used by a
variety
# of installations using different recipient email addresses.
#
# the destination address is passed in the PATH_INFO variable, so the
form that
# uses this would look something like:
#
# <FORM action=/cgi-bin/mailit/downing@nyu.edu>
# email: <INPUT name=email>
# name:  <INPUT name=name>
# Mess:  <INPUT name=message>
# </FORM>
#
# if the name of one of your input fields is called "email" ,
# then this mailer will use that as the "Reply-To" field,
# otherwise, the server's return address will be used.

require 'cgiLib.pl';

%list = getcgidata();

if ($list{message} eq "") {
   simplePage("Mailit Error", "<H1>You Must supply a message!</H1>");
   exit();
}
```

```
htmlheader("Form Submission");

print "<H1 align=center>Thanks for your submission</H1>\n";
print "The file contents of your submission follow:<BR>\n<PRE>\n";


$mailaddr= $ENV{'PATH_INFO'};
$mailaddr =~ tr/\// /;   # This will replace the first /
open (MAIL,"| mail $mailaddr");

if($list{email} ne "") {
   print MAIL "Reply-To: $list{email}\n";
}

foreach (keys %list) {
        print "$_ = $list{$_}\n";
       print (MAIL "$_ = $list{$_}\n");
}
print "</pre>";
close (MAIL);

htmlfooter();
```

Now, you've got to admit that was pretty simple! All of the dirty work is hidden in the cgiLib.pl functions. This leaves you, the programmer, free to concentrate on what you want to program, rather than the tedious part of decoding urlencoded data streams. So, try taking the mailit.pl program above and modify it for your own needs. Once you start to play with it a bit, it should become clear how the cgiLib.pl functions work. Enjoy!

DOS/Windows Scripts

There are a number of DOS and Windows scripts available on the Internet archives. Many are written as batch files, C programs, compiled BASIC programs, or PASCAL programs. Since most of the C programs written for Unix can easily be modified to work on a DOS system, this section will concentrate on the batch files.

A few notes about file location: The standard directory for CGI scripts on a Unix server is in the cgi-bin directory relative to the server root directory. As described in Chapter 21, HTML Assistant, the HTTPD server is installed relative to a directory specified in the configuration files for a particular installation. This directory is usually something like C:\HTTPD and all of the document, configuration, and CGI files are in directories relative to this. Any reference to "server root" is referring to this directory. For example: Given the above installation, to say that the cgi-bin directory is relative to the server root directory is the same as saying C:\HTTPD\CGI-BIN. Likewise, the document root directory is the directory that you have configured your server to look for HTML documents in. Normally this would be something like C:\HTTPD\HTDOCS. The HTTPD server for Windows is very similar to the Unix version. Relative to server root, there is normally a cgi-win and a cgi-dos directory. You should place your DOS and Windows CGI scripts into one of these corresponding directories.

args.bat

```
rem
rem ************
rem * ARGS.BAT *
rem ************
rem
rem Script used in args.htm to illustrate argument transfer
rem
rem Bob Denny <rdenny@netcom.com>
rem 30-Apr-94
rem
rem Echo is OFF at script entry
rem
set of=%output_file%
echo Content-type:text/plain > %of%
echo. >> %of%
echo CGI/1.0 test script report: >> %of%
echo. >> %of%
echo argc = %#  argv: >> %of%
if NOT %#==0  echo %1 %2 %3 %4 %5 %6 %7 %8 >> %of%
if %#==0  echo {empty} >> %of%
echo. >> %of%
echo environment variables: >> %of%
echo REQUEST_METHOD=%REQUEST_METHOD% >> %of%
echo SCRIPT_NAME=%SCRIPT_NAME% >> %of%
echo QUERY_STRING=%QUERY_STRING% >> %of%
echo PATH_INFO=%PATH_INFO% >> %of%
echo PATH_TRANSLATED=%PATH_TRANSLATED% >> %of%
echo. >> %of%
if NOT %REQUEST_METHOD%==POST goto done
echo CONTENT_TYPE=%CONTENT_TYPE% >> %of%
echo CONTENT_FILE=%CONTENT_FILE% >> %of%
echo CONTENT_LENGTH=%CONTENT_LENGTH >> %of%
echo ---- begin content ---- >> %of%
type %CONTENT_FILE% >> %of%
echo. >> %of%
echo ----- end content ----- >> %of%
echo. >> %of%
:done
echo -- end of report -- >> %of%
```

demoindx.bat

```
rem
rem ****************
rem * DEMOINDX.BAT *
rem ****************
rem
rem Offers an ISINDEX document if no query arguments,
rem else reports on the "results" of the query.
rem
rem Bob Denny <rdenny@netcom.com>
rem 28-Apr-94
rem
set of=%output_file%
if NOT %#==0 goto shoquery
rem
rem No query, signal server to do redirect to ISINDEX demo doc.
rem
echo Location: /demo/isindex.htm > %of%
echo. >> %of%
goto done
rem
rem There were query arguments. Generate plain text report
(COMMAND.COM: BAH!)
rem
:shoquery
echo Content-type:text/plain > %of%
echo. >> %of%
echo Here is what the server would have fed to the back-end program:
>> %of%
echo. >> %of%
echo Number of query arguments = %# >> %of%
echo. >> %of%
echo Arguments: >> %of%
echo %1 %2 %3 %4 %5 %6 %7 %8 %9 >> %of%
:done
echo -- end of report -- >> %of%
```

Macintosh Scripts

Most Macintosh scripts are written as AppleScript programs. Here is an AppleScript CGI script and an HTML file to give you an idea of how Macintosh CGI scripting works. Dennis Wilkinson wrote the code you see here as an example of how to get MacHTTP to deal with data from forms supported by clients like XMosaic 2.0.

query.applescript

```
tell window 1 of application "Scriptable Text Editor"
        set contents to http_search_args
        return "<title>Server Query Response</title><h1>Hi!</h1>We get
        the picture. Thanks for the feedback.<P><address>Here</Address>"
end tell
```

query.html

```
<title>Feedback Form</title>
<h2>Submit your feedback</h2>
<form method=GET action="http:/form.script">
Name: <input name="username"><p>
E-Mail: <input name="usermail"><p>
<select name="feedback">
<option selected>I Love It!
<option>I'm Lost!
<option>I Hate It!
</select><p>
<input type=submit value="Submit your feedback now"><p>
<input type=reset value="Reset this form"><p>
</form>
```

Table 15-4 MIME types

MIME Type	File Extension(s)
application/activemessage	
application/andrew-inset	
application/applefile	
application/atomicmail	
application/dca-rft	
application/dec-dx	
application/mac-binhex40	
application/macwriteii	
application/msword	
application/news-message-id	
application/news-transmission	
application/octet-stream	bin
application/oda	oda
application/pdf	pdf
application/postscript	ai eps ps
application/remote-printing	
application/rtf	rtf
application/slate	
application/x-mif	mif
application/wita	
application/wordperfect5.1	
application/x-csh	csh
application/x-dvi	dv

MIME Type	File Extension(s)
application/x-hdf	hdf
application/x-latex	latex
application/x-netcdf	nc cdf
application/x-sh	sh
application/x-tcl	tcl
application/x-tex	tex
application/x-texinfo	texinfo texi
application/x-troff	t tr roff
application/x-troff-man	man
application/x-troff-me	me
application/x-troff-ms	ms
application/x-wais-source	src
application/zip	zip
application/x-bcpio	bcpio
application/x-cpio	cpio
application/x-gtar	gtar
application/x-shar	shar
application/x-sv4cpio	sv4cpio
application/x-sv4crc	sv4crc
application/x-tar	tar
application/x-ustar	ustar
audio/basic	au snd
audio/x-aiff	aif aiff aifc
audio/x-wav	wav
image/gif	gif
image/ief	ief
image/jpeg	jpeg jpg jpe
image/tiff	tiff tif
image/x-cmu-raster	ras
image/x-portable-anymap	pnm
image/x-portable-bitmap	pbm
image/x-portable-graymap	pgm
image/x-portable-pixmap	ppm
image/x-rgb	rgb
image/x-xbitmap	xbm
image/x-xpixmap	xpm

Continued on next page

Continued from previous page

MIME Type	File Extension(s)
image/x-xwindowdump	xwd
message/external-body	
message/news	
message/partial	
message/rfc822	
multipart/alternative	
multipart/appledouble	
multipart/digest	
multipart/mixed	
multipart/parallel	
text/html	html
text/plain	txt
text/richtext	rtx
text/tab-separated-values	tsv
text/x-setext	etx
video/mpeg	mpeg mpg mpe
video/quicktime	qt mov
video/x-msvideo	avi
video/x-sgi-movie	movie

WHAT NOW?

There are popular services that you can use in your Web publishing. The next chapter looks at some of the major non-HTTP services available through WWW browsers.