CD-ROM Features
Valuable Source Code
Plus CGI Programming
Tools & Utilities

for the working programmer

FOUNDATIONS™ of

World Wide Web Programming WITH HTML & CGI

Get Complete Coverage of Forms-Handling CGI

Learn How to Write a Database Front End

Find Special Coverage of WWW Programming Languages: Perl, Java, Python, C, and UNIX Shells

PROGRAMMERS™ PRESS

CD-ROM Features
Valuable Source Code
Plus CGI Programming
Tools & Utilities

for the
working
programmer

FOUNDATIONS™ of

World Wide Web Programming WITH HTML & CGI

Get Complete Coverage of Forms-Handling CGI

Learn How to Write a Database Front End

Find Special Coverage of WWW Programming Languages: Perl, Java, Python, C, and UNIX Shells

ED TITTEL, MARK GAITHER,
SEBASTIAN HASSINGER, & MIKE ERWIN

PROGRAMMERS™
PRESS

FOUNDATIONS™ of

World Wide Web Programming WITH HTML & CGI

Foundations of World Wide Web Programming with HTML and CGI

Published by
IDG Books Worldwide, Inc.
An International Data Group Company
919 East Hillsdale Boulevard, Suite 400
Foster City, CA 94404

Library of Congress Catalog Card No.: 95-077530

ISBN 1-56884-703-3

Printed in the United States of America

Second Printing, November, 1995
10 9 8 7 6 5 4 3 2

Distributed in the United States by IDG Books Worldwide, Inc.

Published in the United States of America

Table of Contents

CHAPTER

6 The Common Gateway Interface

The Hidden Life of HTML

Online WWW documents lead a completely different life than their conventional, print-bound counterparts. They can change with a single keystroke and the effects are immediately visible to the whole world. Since server programs can generate custom-built Web documents on-the-fly in response to interaction with end users, this makes online WWW publishing much more responsive and open-ended than traditional publishing.

Quick response to change and dynamic behavior are the underpinnings of the power behind the WWW. But unless you implement the right approach to building and maintaining Web documents, your information can just lie stagnant and be virtually "dead to the world."

Dynamic behavior is a vital aspect of the Web that is sometimes hard to identify or appreciate. This is particularly true when documents are created on-the-fly, usually in response to some event on the WWW. Although what appears to a user in response to a query may look like "just another Web page," it might actually be an evanescent document created for one-time use in direct response to that query.

The same is often true for the results of a Web search, or even when clicking a link or a button on a particular page. The result is a custom-built response, tailored around previous interaction with the user (or from information provided by users, even if they're blissfully unaware of the underlying communications involved).

What makes all this possible is a special mechanism that supports the dynamic creation of HTML documents. This mechanism is based on

the invocation of external applications that run under the auspices of a WWW server, called by the browser in the form of an ordinary-looking URL on the Web document currently in use. Each of these external applications can be tailor-made to deliver customized, on-demand HTML documents.

Today, many commercial institutions realize the wealth and value of information that resides in their customer and product databases. Many of them want to provide this ancillary information for public consumption. Some key questions they're grappling with are: "How can we make this information accessible?" and "What's the best way to present it for easy consumption and use?" and "How can we make money on this thing?"

The WWW provides the infrastructure and mechanisms for seamless integration of this kind of information, along with typical sales documents like spec sheets, brochures, testimonials, technical documents, and whatnot (which we'll refer to rather loosely as *collaterals*). A key technology underlying the WWW permits static information to be queried and imported on demand and also supports custom construction of Web documents on-the-fly. This technology is called the Common Gateway Interface (CGI), the subject of this chapter.

What Is the Common Gateway Interface?

The Common Gateway Interface (CGI) supplies the middleware among WWW servers, external databases, and information sources. CGI scripts or programs may be considered to be extensions to the core functionality of a WWW server, like that supplied in the CERN or NCSA *httpd* packages. CGI applications perform specific information-processing, retrieval, and formatting tasks on behalf of their WWW servers. The CGI interface defines a method for the Web server to accommodate additional programs and services that may be used to access external applications from within the context of any active Web document.

The term *gateway* describes the relationship between the WWW server and the external applications that handle data access and manipulation chores on its behalf. This gateway incorporates enough built-in

intelligence to guarantee successful communication between the server and the external application. Normally, a gateway handles information requests in some orderly fashion and then returns an appropriate response — for example, an HTML document generated on-the-fly that includes the results of a query applied against an external database.

In other words, CGI allows a WWW server to provide information to WWW clients that would not otherwise be available to those clients in a readable form. This could, for example, allow a WWW client to issue a query to an Informix database and receive an appropriate response in the form of a custom-built Web document. That's why we believe that CGI's main purpose is to support communications with information and services outside the normal purview of WWW, ultimately to produce HTTP objects from non-HTTP objects that a WWW client can render. In short, CGI opens up a door to the whole world of information and services that would otherwise be inaccessible to Web clients and servers alike.

Gateways can be designed and deployed for a variety of purposes, but the most common is the handling of <ISINDEX> and <FORM> HTTP requests. Some common uses of CGI include:

- Gathering user feedback about your product line or your WWW server through an HTML form.
- Dynamic conversion of your system's documentation (for example, *man* pages) into HTML, for easy Web-based access.
- Querying Archie or WAIS databases and rendering results as HTML documents.

Working in tandem with the HTTP server application (*httpd*), CGI applications service requests made by WWW clients by accepting requests for services at the server's behest, handling those requests, and sending appropriate responses back to the client. A client HTTP request consists of the following elements:

- a Universal Resource Identifier (URI)
- a request method
- other important information about the request provided by the transport mechanism of the HTTP

In the sections that follow, you'll have a chance to understand all of these elements in detail as you become familiar with the structure and function of a client HTTP request.

The CGI Specification: Extending HTTP

The CGI specification was created and documented by the main HTTP server authors: Tony Sanders, Ari Luotonen, George Phillips, and John Franks. These folks discovered that they didn't want to keep adding functionality to their HTTP servers every day, in keeping with the needs of one particular Web activity or another. They decided to build a clearly defined core of WWW server functionality and to provide a way to extend services and capabilities from there.

They needed an application programming interface (API) available to any Perl, C, or shell hacker willing to learn the appropriate interface details. Hence, the creation of CGI as a formal, rigorous specification that includes some nasty notation and grammar.

For more information about the CGI specification, please consult the following URL:

```
http://hoohoo.ncsa.uiuc.edu/cgi/interface.html
```

In the subsections that follow, you'll have a chance to cover the elements of the specification, as we prepare you to build CGI programs of your own.

A Cast of Environment Variables

Environment variables are entities that exist within a particular user's computer environment. Many of these variables attain their values whenever a user logs into a computer, or when the computer is booted for a single-tasking operating system like DOS. Environment variables

are pervasive within UNIX, where they are used to pass information to applications from the runtime environment.

Because the environment persists even as execution threads come and go, environment variables sometimes function as placeholders, to pass data from one application to another within the same user session. In the case of CGI, environment variables known to the server and CGI are used to pass data about an HTTP request from a server to the CGI application. These variables are accessible to both the server and any CGI application it may invoke.

The WWW server may also use command-line arguments, as well as environment variables, to pass data about an information request from a WWW client. This allows specific control over program parameters or execution to originate with the server, as well as providing a mechanism for clients to pass short input sequences to the CGI program involved. Likewise, these variables may also be set (or assigned their values) when the server actually executes a CGI application.

ACCESSING CGI VARIABLES

The requirements of a specific WWW server control how CGI applications access variables. If a variable does not have a value, this indicates a zero-length value (NULL in the UNIX realm). If the variable is not defined in the server's system (omitted or missing), it is assumed to have a zero-length value and hence, is also assumed to be NULL. As with variable access, the system requirements for the WWW server dictate the means of value representation for CGI variables.

Likewise, the names for environment variables are system specific. For historic reasons, we will discuss UNIX environment variables in detail. Please notice that these names are case sensitive, so they must be reproduced exactly in order to reference the correct information. Table 6-1 is a quick reference to the standard environment variables that are specific to each request handled by a CGI application.

Table 6-1
Standard Environment Variables Quick Reference

Environment Variable	Description
AUTH_TYPE	access authentication type
CONTENT_LENGTH	size in decimal number of octets of any attached entity
CONTENT_TYPE	the MIME type of an attached entity
GATEWAY_INTERFACE *	server's CGI spec version
HTTP_(string)	client header data
PATH_INFO	path to be interpreted by CGI app
PATH_TRANSLATED	virtual to physical mapping of file in system
QUERY_STRING	URL-encoded search string
REMOTE_ADDR	IP address of agent making request
REMOTE_HOST	fully qualified domain name of requesting agent
REMOTE_IDENT	identity data reported about agent connection to server
REMOTE_USER	user ID sent by client
REQUEST_METHOD	request method by client
SCRIPT_NAME	URI path identifying a CGI app
SERVER_NAME *	server name; host part of URI; DNS alias
SERVER_PORT	server port where request was received
SERVER_PROTOCOL	name and revision of request protocol
SERVER_SOFTWARE *	name and version of server software

* not request-specific (set for all requests).

AUTH_TYPE

This variable is used to provide various levels of server access security. If your server supports user authentication, this environment variable value indicates the protocol-specific authentication method used to validate a user.

HTTP provides a simple challenge–response authorization mechanism. This mechanism can be used by a server to challenge a client's request or by a client to provide authentication information to a server. The variable's value is deciphered from the HTTP *auth-scheme* token in the client's request HTTP header. If no access authorization is required, the value is set to NULL.

For example, a server may require a client to provide a user ID and password to access a specific part of its Web. In order to make this

happen, the client must set the AUTH_TYPE environment variable to a value that matches the supported protocol of the server. For HTTP applications, the most common value is:

```
AUTH_TYPE = Basic
```

This provides a basic authentication scheme where the client must authenticate itself with a user ID and password pair. For more information on this (and other) environment variables, please consult section 10.1 of the HTTP 1.0 specification at the following URL:

```
http://www.w3.org/hypertext/WWW/Protocols/HTTP1.0/HTTP1.0/draft-ietf-http-spec.html#BasicAA
```

CONTENT_LENGTH

This variable's value equals the decimal number of octets for the attached entity, if any. If there is no attached entity, it is set to NULL. For example:

```
CONTENT_LENGTH = 9
```

CONTENT_TYPE

The value of this variable indicates the media type of the attached entity, if any. If there is no attached entity, it is also set to NULL. HTTP 1.0 uses MIME Content-Types, which provide an extensible and open mechanism for data typing and for data type negotiation.

With HTTP, you can represent different media like text, images, audio, and video. Within such media, there often exist various encoding methods. For instance, images can be encoded as jpeg, gif, ief, or tiff. The value of the CONTENT_TYPE variable for an attached gif image file could therefore be:

```
CONTENT_TYPE = image/gif
```

For HTTP requests, a client may provide a list of acceptable media types as part of its request header. This allows the client more autonomy in its use of unregistered media types. Although data providers are strongly encouraged to register their unique media types and subtypes with the Internet Assigned Numbers Authority (IANA), this does provide additional flexibility. But since both client and server must be able to

handle unregistered items, media types registered with the IANA are preferred over unregistered extensions.

See Table 6-2 for the seven standard predefined MIME Content-Types. (The primary subtype is presented in italics.)

Table 6-2
MIME Registered Content-Types Used by HTTP

Type	Subtypes	Description
application	*octet-stream*	transmit application data
text	*plain*	textual information
multipart	*mixed*, alternative, digest, parallel	multiple parts of independent data types
message	*rfc822*, partial, external-body	an encapsulated message
image	gif, jpeg	image data
audio	*basic*	sound data
video	*mpeg*	video data

application

This MIME Content-Type describes transmitted application data. This is typically uninterpreted binary data. The primary subtype is 'octet-stream' intended to be used in the case of uninterpreted binary data. An additional subtype, 'PostScript', is defined for transporting PostScript documents within MIME bodies.

Subtype Specifications

Whenever an environment variable supports subtyping, note that some subtype specification is *mandatory* for that variable. That is, there are no default subtypes.

All type, subtype, and parameter names are case insensitive. For example, 'IMAGE', 'Image', and 'image' are all equivalent. The only constraint on subtype names is that their uses must not conflict. It would be undesirable to have two different uses of 'Content-Type: application/grunge'. The name provides a simple mechanism for publicizing its uses, not constraining them.

The simplest and most common action when a client requests 'application' MIME Content-Type data is to offer to write the information to a file. For example, the environment variable:

```
CONTENT_TYPE = application/octet-stream
```

would result in the client asking the user if he or she would like to save the binary data to a local file on his or her hard drive.

Other expected uses for 'application' could include things like spreadsheet data, data for mail-based scheduling systems, compressed binary data such as *gzip*, and languages for "active" e-mail.

text

This MIME Content-Type describes textual data. The primary subtype is 'plain'. This unformatted text requires no special software to render the text, although the indicated character set must be supported by the client. Text subtypes are forms of enriched text where software may enhance the presentation of the text. The most common non-registered text subtype used on the Web is 'html (text/html). This software must not be required in order to grasp the general idea of the text's content. One such possible subtype is 'richtext' which was introduced by RFC 1341.

An example of this environment variable is:

```
CONTENT_TYPE = text/plain; charset=us-ascii
```

For more information about 'richtext' and RFC 1341 see this URL:

```
http://www.cis.ohio-state.edu/htbin/rfc/rfc1341.html
```

Please note that since its publication, RFC 1341 has been superseded by RFC 1521, which incorporates some new material not mentioned in 1341. For current implementation details, please consult the most current version of any RFC. The full collection, which includes pointers from obsolete to current versions, can be accessed through the following URL:

```
http://www.cis.ohio-state.edu/hypertext/information/rfc.html
```

multipart

This MIME Content-Type describes data consisting of multiple parts of independent data types. There are four common subtypes.

1. 'mixed' is the primary subtype.
2. 'alternative' represents the same data type in multiple formats.
3. 'parallel' for parts intended to be viewed at the same time.
4. 'digest' for parts of type 'message'.

message

This MIME Content-Type subtype is an encapsulated message. A MIME body of Content-Type 'message' is itself all or part of a fully formatted message. This message must comply with the Internet Mail Header protocol described in RFC 822. This RFC 822–compliant message may contain its own completely different MIME Content-Type header field.

There are three subtypes:

1. 'rfc822' is the primary subtype.
2. 'partial' is defined for partial messages that allow fragmented transmission of MIME bodies deemed too large for normal mail transport facilities.
3. 'external-body' is for specifying large MIME bodies by reference to an external data source such as an image repository.

Here's an example of a standard e-mail text message:

```
CONTENT_TYPE = message/rfc822
```

image

This MIME Content-Type describes image data. The MIME Content-Type image requires a display device such as a graphical display, a FAX machine, or a printer to view the image data. There are two subtypes that are defined for two widely used image formats, jpeg and gif.

For example, data of a gif encoded image would be:

```
CONTENT_TYPE = image/gif
```

audio

This MIME Content-Type describes audio or sound data. The primary subtype is 'basic'. Audio requires an audio output device such as a speaker to display the contents.

An example of a MIDI–attached MIME body would be:

```
CONTENT_TYPE = audio/x-midi
```

NOTE: the "x" prefix declares this MIME Content-Type subtype to be an extended data type.

video

This MIME Content-Type describes video data. Video requires the capability to display moving images. This is typically accomplished with specialized hardware and software. The primary subtype is 'mpeg'.

An example of a video data entity would be:

```
CONTENT_TYPE = video/mpeg
```

For more information about the IANA, see this URL:

```
http://ds.internic.net/rfc/rfc1590.txt
```

GATEWAY_INTERFACE

This environment variable represents the version of the CGI specification to which the server that supplies this value complies. This variable is not request-specific and is set for all HTTP requests. An example value would be:

```
GATEWAY_INTERFACE = CGI/1.1
```

http_(string)

CGI environment variables beginning with the string 'HTTP_' contain HTTP header information supplied by the client. These are the MIME Content-Types that the client will accept, as supplied in HTTP request

headers. Each data type in this list should be separated by a comma as required in the HTTP 1.0 specification.

The HTTP header name is converted to all uppercase characters. All "-" (hyphen) characters are replaced with "_" (underscore). Finally, the 'HTTP_' string is prepended, resulting in the set of HTTP_string environment variables.

The HTTP request header data sent by the client may be preserved, or it may be transformed, but it will not change its meaning. An example of this behavior occurs when a server may strip comment data from an HTTP request header. If necessary, the receiving server must transform the HTTP request header data to be consistent with the legal syntax for a CGI environment variable.

In the course of its operation, the WWW server handles many HTTP request headers. It is not required to create CGI environment variables for all received header data. In particular, the server should remove any headers containing authentication data such as 'Authorization'. This is valuable information that should not be available to other CGI applications. The following example identifies the MIME Content-Types that a server supports and services:

```
HTTP_ACCEPT = application/zip, audio/basic, audio/x-midi, application/x-rtf,
video/msvideo, video/quicktime, video/mpeg, image/targa, image/x-win-bmp,
image/jpeg, image/gif, application/postscript, audio/wav, text/plain,
text/html, audio/x-aiff, audio/basic, application/x-airmosaic-patch, applica-
tion/binary, application/http, www/mime
```

PATH_INFO

This variable represents extra path information, provided by a requesting client. It describes a resource to be returned by a CGI application once it completes successfully. Clients can access CGI applications using virtual pathnames, followed by extra information appended to this path. Because URLs use a special encoding method for character data, this path information will be decoded by the server if it comes from a URL before it is passed to a CGI application.

The value of the PATH_INFO variable can be one of the following:

1. It may be contained in some trailing part of the requesting client's URI.

2. It may be some string provided by the client to the server.
3. It may be the entire client's URI.

A CGI application cannot decipher how PATH_INFO was chosen or constructed based solely on its value. As a CGI programmer, you must determine the context in which this variable is to be used in your CGI application. An example of a valid PATH_INFO variable is:

```
PATH_INFO = /raising/arizona/
```

PATH_TRANSLATED

This variable represents the operating system path to a file on the system that a server would attempt to access for a client requesting an absolute URI containing the PATH_INFO data. The server provides a translated version of PATH_INFO, which takes the path data and does any virtual-to-physical file system mapping.

For security reasons, some servers do not support this variable and will set it to NULL. This variable need not be supported by a server. The algorithm a server uses to decode PATH_TRANSLATED is invariably operating-system dependent. Therefore, CGI applications utilizing PATH_TRANSLATED may have limited portability. Here's an example of an NCSA server's 'DocumentRoot' equal to '/u/Web' that builds on the preceding example:

```
PATH_TRANSLATED = /u/Web/raising/arizona/
```

QUERY_STRING

This variable represents a URL-encoded search string. The value of this variable follows the "?" character in the URL that referenced a CGI search application. The value of this variable is not decoded by the server, but is passed onto the CGI application untouched.

The information after the "?" in a URL can be added by one of the following:

- an HTML <ISINDEX> document
- an HTML <FORM> CGI application (using the GET method)

- manually appended by the HTML document's author in which the reference occurs, in an <A> (anchor statement)

The value of QUERY_STRING is encoded in the standard URL format of replacing a space with a "+" and encoding nonprintable characters with the hexadecimal encoding scheme of '%dd' where "d" is a digit. You will need to decode this hexadecimal information before you use it. (You will learn how to do this in Chapter 8.)

This variable is always set when there is query information requested by a client, regardless of any command-line decoding by the CGI application. An example of a client's query URL for a CGI search application ('places.pl') to locate a place named "Sunset Crater" could look like this:

```
http://www.flagstaff.az.us/cgi-bin/places.pl?Sunset_Crater
```

The resulting QUERY_STRING value for this URL would be:

```
QUERY_STRING = Sunset_Crater
```

REMOTE_ADDR

This CGI environment variable represents the Internet Protocol (IP) address for a requesting agent. This is not necessarily the address of the client, but could be the address of the host for that client, depending on the type of connection and operating system the client is using. Here's an example:

```
REMOTE_ADDR = 199.1.78.25
```

REMOTE_HOST

This variable represents the fully qualified domain name for a requesting agent. If the server cannot decipher this information, REMOTE_ADDR is set to NULL. Domain names are case insensitive. Here's an example:

```
REMOTE_HOST = ppp1.aus.sig.net
```

This is the address of the machine of one of the author's Internet service providers, where the client actually resides on his machine at

home. It communicates to the Internet through the host via the Point to Point Protocol (PPP) — in this case, using connection 1 (which is why the first term in the domain name is "pppP1"). Thus, the agent is not necessarily the client.

REMOTE_IDENT

This variable represents the remote user's name retrieved from the server. This RFC 931 (Authentication Server) identification must be supported by the HTTP server. It is highly recommended that use of this CGI environment variable be limited to logging actions. The current RFC 931 documentation implies that the HTTP server must attempt to retrieve the identification data from the requesting agent if it supports this feature. Finally, the value of this variable is not appropriate for user authentication purposes. An example would be:

```
REMOTE_IDENT = inmate.state.prison.az.us
```

REMOTE_USER

The value of this environment variable is the authenticated user's name. Two conditions must apply for this variable's value to be set:

1. The server must support user authentication.
2. The CGI application must be protected from unauthorized access.

If the AUTH_TYPE variable's value is 'Basic', then the value for REMOTE_USER will be the user's identification sent by the requesting client:

```
REMOTE_USER = gail_snokes
```

REQUEST_METHOD

The value of this CGI environment variable represents the method with which the client request was made. For HTTP 1.0, this is GET, HEAD, POST, PUT, DELETE, LINK, and UNLINK. The method name is case sensitive. An example for a client posting HTML form data to a server is:

```
REQUEST_METHOD = POST
```

SCRIPT_NAME

This variable's value is the URI path that identifies a CGI application. It is the virtual path to a CGI application being executed by a server. In this example, the client's query URL of a CGI application that calculates an engine's maximum horsepower is:

```
http://www.muscle.cars.org/cgi-bin/calc_max_hp.pl
```

and the corresponding SCRIPT_NAME variable's value is:

```
SCRIPT_NAME = /cgi-bin/calc_max_hp.pl
```

SERVER_NAME

This variable represents the server's hostname, DNS alias, or IP address as it would appear in URLs. This variable is not request-specific and is set for all requests. An example is:

```
SERVER_NAME = www.hal.com
```

SERVER_PORT

This variable is the port on which a client request is received. This is deciphered from the appropriate part of a URL. Most HTTP server implementations use port 80 as the default port number. URLs that do not explicitly specify a port can be accessed via port 80. For example, the following URL specifies that port 8119 is the designated HTTP server port:

```
http://www.chevelle.org:8119/index.html
```

SERVER_PROTOCOL

This variable's value represents the name and revision of the information protocol that the requesting client utilizes. The protocol is similar to the URL scheme used by a requesting client, and it is case insensitive. An example is:

```
SERVER_PROTOCOL = HTTP/1.0
```

SERVER_SOFTWARE

This variable represents the name and version of the information server software. This variable is not request-specific and is set for all requests:

```
SERVER_SOFTWARE = NCSA/1.4
```

THE CGI COMMAND LINE

Some operating systems support methods for providing data to CGI applications via the command line. This technique is used only in the case of an HTML <ISINDEX> query.

A server identifies this type of request by the GET method, accompanied by a URI search string that does not contain any *unencoded* "=" characters. This method trusts the clients to encode the "=" character in an <ISINDEX> query. This practice was considered safe at the time of the design of the CGI specification and requires that you, as a CGI programmer, be careful to observe this restriction.

A server parses this string into words using the rules in the CGI specification, as follows:

1. The server decodes the query information by first splitting the string on the "+" characters in the URL (which map to spaces, following URL encoding techniques).
2. The server then performs any additional decoding before placing each split string into an array named 'argv'.
   - Typically, this consists of separating out individual variable strings by splitting the input stream at each ampersand ("&").
   - Then, the resulting substrings must be split again (at the equal sign "=" character) to establish name and value pairs, where the left-hand-side value provides the name of the variable, and the right-hand-side value supplies its corresponding value.

This approach follows the UNIX method for passing command-line information to applications. 'argv' is an array of pointers to strings,

where the length of the 'argv' array is stored in the environment variable 'argc'. (Love that UNIX!)

If the server finds that it cannot send the 'argv' array to the CGI application, it will include *no* command-line information. Instead, it will provide the non-decoded query information within the environment variable QUERY_STRING. This failure may be attributed to internal limitations of the server like those imposed by *exec()* or */bin/sh* command-line restrictions. Another example of such a failure occurs when the number of arguments sent by the server to the CGI application may exceed the operating system's or server's limitations. It may also occur when a string in the 'argv' array is not a valid argument for the CGI application that it enters.

For demonstrated examples of command-line usage see this URL:

```
http://hoohoo.ncsa.uiuc.edu/cgi/examples.html
```

Make sure when you run these examples that you pay close attention to the values for the command-line variables 'argc' and 'argv'. We'd argue strongly that the first integrity check on accepting CGI input in a program is to make sure that 'argv' is non-zero, and that the number matches the expected number of arguments to your program!

It's also noteworthy that this command-line data-passing is seldom used for handling HTML forms, which can contain arbitrarily large amounts of data. HTML forms normally utilize environment variables to pass data using the POST method, because this method bypasses command-line length restrictions and the occasional data-passing problem that context-switching can sometimes cause in the UNIX runtime environment.

CGI DATA INPUT

For requests with data attached following the HTTP request header, like HTTP POST or PUT, the data is delivered to the CGI application using a standard input file descriptor. All CGI applications follow this method of reading data. For UNIX operating systems, this uses the so-called standard input device, commonly referred to as *stdin*.

The server will send a number of bytes of information, at least as long as the value specified by the variable CONTENT_LENGTH, using the *stdin* file descriptor to pass the data to the CGI application. The CGI

application is not required to read all of this data, but neither must it attempt to read more than CONTENT_LENGTH bytes.

The server also provides the CONTENT_TYPE for data passed to the CGI application. This permits the application to use this information to decide how to interpret the data it receives. At the end of the data stream, the server is not required to transmit an end-of-file marker; both server and application assume that reading will cease immediately after the CGI application reads the number of bytes specified by the value of CONTENT_LENGTH.

For example, let's assume you are working with an HTML form that provides its data using the <FORM> METHOD="POST". Let's say this form's results are 15 bytes encoded, and look like "one=bob&two=cat" (in decoded form).

In this example, the server will set CONTENT_LENGTH to '15' and CONTENT_TYPE to 'application/x-www-form-urlencoded'. The first byte on the script's standard input file descriptor will be the character "O", followed by the rest of the encoded string.

CGI DATA OUTPUT

A CGI application always returns *something* to the client that invokes it, or to the server. The CGI application sends its output to a standard output file descriptor. UNIX calls this *stdout*. The CGI application output might be a document generated by the application (typically HTML, destined for delivery to the client) or it might be data retrieval instructions to the server (e.g., to transfer a file to the client).

Output is usually returned in one of two ways. The first form is non-parsed header output. Using this form, a CGI application must return a complete HTTP response message.

The second form of returned data is parsed header output. A server is only required to support this form. In this case, the CGI application returns a CGI response message. This response consists of headers and a body, separated by a blank line. These headers are either CGI headers that will be interpreted by a server or they are HTTP headers to be included within the response message. Here, the body is optional; but if it is supplied, the body must be prefaced by a MIME Content-Type header.

If the body is excluded, the CGI application must transmit either a Location or Status header. The Location header is used to indicate to the server that the CGI application is returning a reference to a document rather than the document's bits and pieces. The Status header is utilized to provide information to the server about the status code the server should use in its response message to the client. For information about the syntax of these CGI headers, please consult the CGI specification (or at least Chapter 8, which covers CGI input and output in considerable detail).

Summary

In this chapter we presented the middleware needed to bridge the abyss between Web and non-Web information repositories. In the next chapter, we introduce you to some old and proven programming languages and also to some new and exciting programming languages that you can use to implement CGI applications.