16

Web Programming with CGI

Objectives

- To understand the Common Gateway Interface (CGI) protocol.
- To understand the Hypertext Transfer Protocol (HTTP) and to use HTTP headers.
- To understand a Web server's functionality.
- To introduce the Apache HTTP Server.
- To request documents from a Web server.
- To implement a simple CGI script.
- To send input to CGI scripts using XHTML forms.

This is the common air that bathes the globe.
Walt Whitman

The longest part of the journey is said to be the passing of the gate.
Marcus Terentius Varro

Railway termini... are our gates to the glorious and unknown. Through them we pass out into adventure and sunshine, to them, alas! we return.
E. M. Forster

There comes a time in a man's life when to get where he has to go—if there are no doors or windows—he walks through a wall.
Bernard Malamud

Outline

16.1 Introduction

With the advent of the World Wide Web, the Internet gained tremendous popularity. This greatly increased the volume of requests users made for information from Web sites. It became evident that the degree of interactivity between the user and the Web site would be crucial. The power of the Web resides not only in serving content to users, but also in responding to requests from users and generating Web content dynamically.

In this chapter, we discuss specialized software—called a *Web server*—that responds to client (e.g., Web browser) requests by providing resources (e.g., XHTML[1] documents). For example, when users enter a *Uniform Resource Locator* (*URL*) address, such as **`www.deitel.com`**, into a Web browser, they are requesting a specific document from a Web server. The Web server maps the URL to a file on the server (or to a file on the server's network) and returns the requested document to the client. During this interaction, the Web server and the client communicate through the platform-independent *Hypertext Transfer Protocol* (*HTTP*), a protocol for transferring requests and files over the Internet (i.e., between Web servers and Web browsers).

1. The Extensible HyperText Markup Language (XHTML) has replaced the HyperText Markup Language (HTML) as the primary way of describing Web content. Readers not familiar with XHTML should read Appendix E, Introduction to XHTML, before reading this chapter.

Our Web-server discussion introduces the *Apache HTTP Server*. For illustration purposes, we use Internet Explorer to request documents and, later, to display content returned from "CGI scripts."

16.2 HTTP Request Types

HTTP defines several request types (also known as *request methods*), each of which specifies how a client makes requests from a server. The two most common are *get* and *post*. These request types retrieve and send client form data from and to a Web server. A form is an XHTML element that may contain text fields, radio buttons, check boxes and other graphical user interface components that allow users to enter data into a Web page. Forms can also contain hidden fields, not exposed as GUI components. A *get* request is used to send data to the server. A *post* request also is used to send data to the server. A *get* request sends form data as part of the URL (e.g., **`www.searchsomething.com/search?query=`***userquery*). In this fictitious request, the information following the **`?`** (**`query=`***userquery*) indicates user-specified input. For example, if the user performs a search on "Massachusetts," the last part of the URL would be **`?query=Massachusetts`**. A *get* request limits the *query string* (e.g., **`query=Massachusetts`**) to a predefined number of characters. This limit varies from server to server. If the query string exceeds this limit, a *post* request must be used.

Software Engineering Observation 16.1

The data sent in a post *request is not part of the URL and cannot be seen by users. Forms that contain many fields often are submitted via a* post *request. Sensitive form fields, such as passwords, usually are sent using this request type.*

An HTTP request often sends data to a *server-side form handler* that processes the data. For example, when a user participates in a Web-based survey, the Web server receives the information specified in the form as part of the request and processes the survey in the form handler.

Browsers often *cache* (save on a local disk) Web pages for quick reloading, to reduce the amount of data that the browser needs to download. However, browsers typically do not cache the responses to *post* requests, because subsequent *post* requests might not contain the same information. For example, users participating in a Web-based survey may request the same Web page. Each user's response changes the overall results of the survey, thus the information presented in the resulting Web page is different for each request.

Web browsers often cache the server's responses to *get* requests. A static Web page, such as a course syllabus, is cached in the event that the user requests the same resource again.

16.3 Multi-Tier Architecture

A Web server is part of a *multi-tier application*, sometimes referred to as an *n*-tier application. Multi-tier applications divide functionality into separate tiers (i.e., logical groupings of functionality). Tiers can be located on the same computer or on separate computers. Figure 16.1 presents the basic structure of a three-tier application.

The *information tier* (also called the *data tier* or the *bottom tier*) maintains data for the application. This tier typically stores data in a *relational database management system (RDBMS)*. For example, a retail store might have a database of product information, such

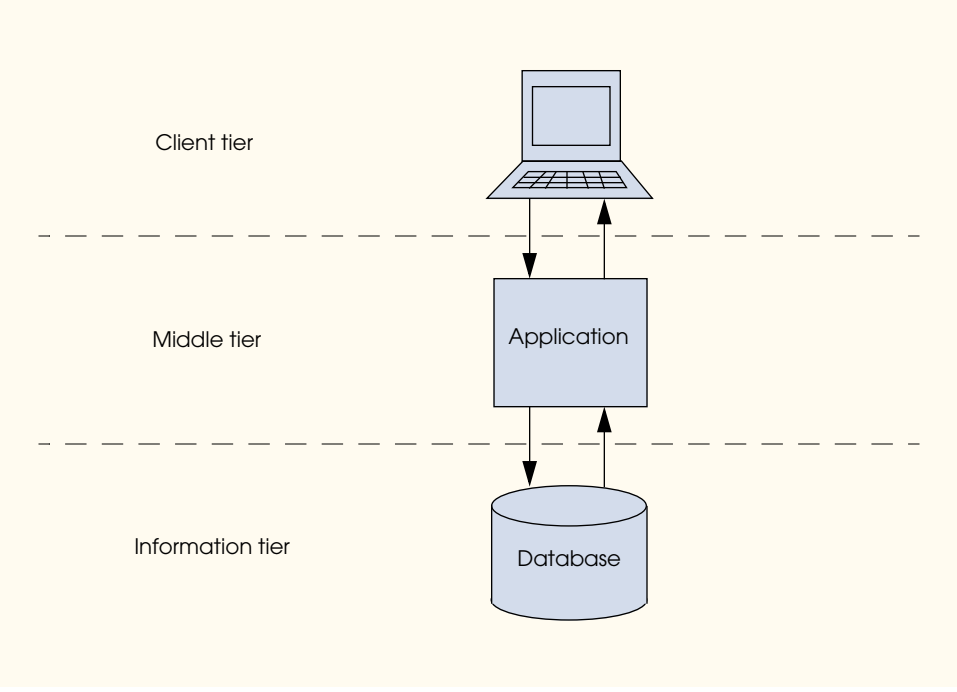


Fig. 16.1 Three-tier application model.

as descriptions, prices and quantities in stock. The same database also might contain customer information, such as user names for logging into the online store, billing addresses and credit-card numbers.

The *middle tier* implements *business logic* and *presentation logic* to control interactions between application clients and application data. The middle tier acts as an intermediary between data in the information tier and the application clients. The middle-tier *controller logic* processes client requests from the top tier (e.g., a request to view a product catalog) and retrieves data from the database. The middle-tier presentation logic then processes data from the information tier and presents the content to the client. In Web-based applications, the middle tier presentation logic typically presents content as XHTML documents.

Business logic in the middle tier enforces *business rule*s and ensures that data is reliable before updating the database or presenting data to a user. Business rules dictate how clients can and cannot access application data and how applications process data.

The *client tier*, or *top tier*, is the application's user interface. Users interact directly with the application through the user interface. The client interacts with the middle tier to make requests and to retrieve data from the information tier. The client then displays to the user the data retrieved from the middle tier.

16.4 Accessing Web Servers

To request documents from Web servers, users must know the URLs at which those documents reside. A URL contains a machine name (called a *host name*) on which the Web serv-

er resides. Users can request documents from *local Web servers* (i.e., ones residing on user's machines) or *remote Web servers* (i.e., ones residing on machines across a network).

Local Web servers can be accessed in two ways: through the machine name, or through **`localhost`**—a host name that references the local machine. We use **`localhost`** in this chapter. To determine the machine name in Windows Me, right-click **My Network Places**, and select **Properties** from the context menu to display the **Network** dialog. In the **Network** dialog, click the **Identification** tab. The computer name displays in the **Computer name:** field. Click **Cancel** to close the **Network** dialog. In Windows 2000, right click **My Computer** and select **Properties** from the context menu to display the **System Properties** dialog. In the dialog, click **Network Identification**. The **Full Computer Name:** field in the **System Properties** window displays the computer name. In Windows XP, select **Start > Control Panel > Switch to Classic View > System** to view the **System Properties** dialog. In the dialog, select the **Computer Name** tab.

A domain name represents a group of hosts on the Internet; it combines with a host name (e.g., **`www`**—World Wide Web) and a *top-level domain (TLD)* to form a *fully qualified host name*, which provides a user-friendly way to identify a site on the Internet. In a fully qualified host name, the TLD often describes the type of organization that owns the domain name. For example, the **`com`** TLD usually refers to a commercial business, whereas the **`org`** TLD usually refers to a non-profit organization. In addition, each country has its own TLD, such as **`cn`** for China, **`et`** for Ethiopia, **`om`** for Oman and **`us`** for the United States.

Each fully qualified host name is assigned a unique address called an *IP address*, which is much like the street address of a house. Just as people use street addresses to locate houses or businesses in a city, computers use IP addresses to locate other computers on the Internet. A *domain name system (DNS) server*, a computer that maintains a database of host names and their corresponding IP addresses, translates fully qualified host names to IP addresses. This translation is referred to as a *DNS lookup*. For example, to access the Deitel Web site, type the hostname (**`www.deitel.com`**) into a Web browser. The DNS server translates **`www.deitel.com`** into the IP address of the Deitel Web server (i.e., **`63.110.43.82`**). The IP address of **`localhost`** is always **`127.0.0.1`**.

16.5 Apache HTTP Server[2]

The Apache HTTP server, maintained by the Apache Software Foundation, is currently the most popular Web server because of its stability, cost, efficiency and portability. It is an open-source product that runs on Unix, Linux and Windows platforms.

To download the Apache HTTP server, visit **`www.apache.org`**.[3] For instructions on installing Apache, visit **`www.deitel.com`**. After installing the Apache HTTP server, start the server by selecting the **Start** menu, then **Programs > Apache HTTP Server 2.0.39 > Control Apache Server > Start**. If the server starts successfully, a command-prompt window opens, and states that the service is starting (Fig. 16.2). To stop the Apache HTTP server, select **Start > Programs > Apache HTTP Server 2.0.39 > Control Apache Server > Stop**.

2. This section applies to Windows 98/NT/2000/Me/XP, Unix and Linux users.
3. In this chapter, we use version 2.0.39.

Fig. 16.2 Starting the Apache HTTP server.

16.6 Requesting XHTML Documents

This section shows how to request an XHTML document from the Apache HTTP server. In the Apache HTTP server directory structure, XHTML documents must be saved in the **`htdocs`** directory. On Windows platforms, the **`htdocs`** directory resides in **`C:\Program Files\Apache Group\Apache`**; on Linux platforms, the **`htdocs`** directory resides in the **`/usr/local/httpd`** directory.[4] Copy the **`test.html`** document from the Chapter 16 examples directory on the book's CD-ROM into the **`htdocs`** directory. To request the document, launch a Web browser, such as Internet Explorer, Netscape or equivalent and enter the URL in the **Address** field (i.e., **`http://localhost/test.html`**). Figure 16.3 shows the result of requesting **`test.html`**. [*Note*: In Apache, the root of the URL refers to the default directory, **`htdocs`**, so we do not enter the directory name in the **Address** field.]

16.7 Introduction to CGI

The *Common Gateway Interface (CGI)* is a standard for enabling applications (commonly called *CGI programs* or *CGI scripts*) to interact with Web servers and (indirectly) with clients (e.g., Web browsers). CGI is often used to generate *dynamic Web content* using client input, databases and other information services. A Web page is dynamic if its content is generated programmatically when the page is requested, unlike *static Web content*, which is not generated programmatically when the page is requested (i.e., the page already exists before the request is made). For example, we can use CGI to have a Web page ask users for their ZIP codes, then redirect users to another Web page that is specifically for people in

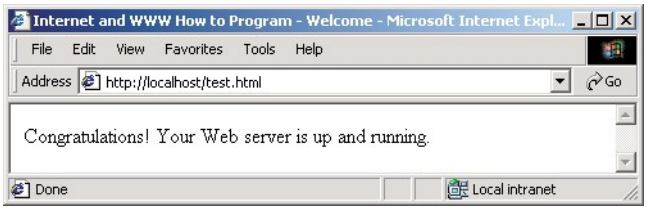


Fig. 16.3 Requesting **`test.html`** from Apache.

4. Linux users may already have apache installed by default. The **`htdocs`** directory may be found in a number of places depending on the Linux distribution.

that geographical area. In this chapter, we introduce the basics of CGI and use C++ to write our first CGI scripts.

The Common Gateway Interface is "common" in the sense that it is not specific to any particular operating system (such as Linux or Windows) or to any one programming language. CGI was designed to be used with virtually any programming language. Thus, CGI scripts can be written in C, C++, Perl, Python or Visual Basic without difficulty.

CGI was developed in 1993 by *NCSA* (*National Center for Supercomputing Applications*—**www.ncsa.uiuc.edu**) for use with its popular *HTTPd Web server*. Unlike Web protocols and languages that have formal specifications, the initial concise description of CGI written by NCSA proved simple enough that CGI was adopted as an unofficial standard worldwide. CGI support was incorporated quickly into other Web servers, including Apache.

16.8 Simple HTTP Transaction

Before exploring how CGI operates, it is necessary to have a basic understanding of networking and how the World Wide Web works. In this section, we will examine the inner workings of the Hypertext Transfer Protocol (HTTP) and discuss what goes on behind the scenes when a browser requests and then displays a Web page. HTTP describes a set of *methods* and *headers* that allows clients and servers to interact and exchange information in a uniform and predictable way.

A Web page in its simplest form is an XHTML document, which is a plain text file that contains markings (*markup* or *elements*) that describe the structure of the data the document contains. For example, the XHTML

```
<title>My Web Page</title>
```

indicates to the browser that the text between the ***<title>*** *start element* and the ***</title>*** *end element* is the title of the Web page. XHTML documents also can contain *hypertext* information (usually called *hyperlinks*), which create links to other Web pages or to other locations on the same page. When a user activates a hyperlink (usually by clicking it with the mouse), the Web browser "follows" the hyperlink by loading the new Web page (or a different part of the same Web page).

Each XHTML file available for viewing over the Web has a *URL (Universal Resource Locator)* associated with it—an address of sorts. The URL contains information that directs a browser to the resource (most often a Web page) that the user wishes to access. For example, consider the URL

```
http://www.deitel.com/books/downloads.html
```

The **http://** indicates that the Web browser should request the resource using the Hypertext Transfer Protocol. The middle portion, **www.deitel.com**, is the *hostname* of the server. The hostname is the name of the computer where the resource resides; likewise, this computer is usually referred to as the *host*, because it houses and maintains the resource.

The name of the resource being requested, **/books/downloads.html** (an XHTML document), is the remainder of the URL. This portion of the URL specifies both the name of the resource (**downloads.html**) and its path (**/books**). The path could represent an actual directory in the Web server's file system. However, for security reasons, the path often is a *virtual directory*. In this case, the server translates the path into a real

location on the server (or even on another computer), thus hiding the true location of the resource. In fact, it is even possible that the resource is created dynamically and does not reside anywhere on the server computer. As we will see, URLs also can be used to provide input to a program on the server.

Now we consider how a browser, when given a URL, performs a simple HTTP transaction to retrieve and display a Web page. Figure 16.4 illustrates the transaction in detail. The transaction is performed between a Web browser and a Web server.

In Step 1 of Fig. 16.4, the browser sends an HTTP request to the server. The request (in its simplest form) looks like the following:

```
GET /books/downloads.html HTTP/1.1
Host: www.deitel.com
```

The word ***GET*** is an *HTTP method,* that indicates the client wishes to retrieve a resource. The remainder of the request provides the name and path of the resource (an XHTML document) and the protocol's name and version number (**`HTTP/1.1`**).

Any server that understands HTTP (version 1.1) will be able to translate this request and respond appropriately. Step 2 of Fig. 16.4 shows the results of a successful request. The

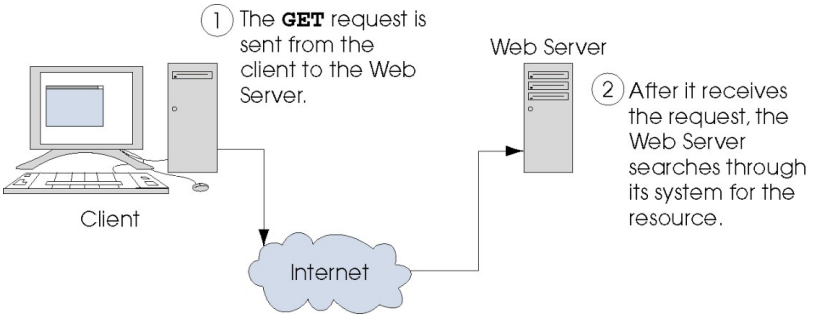


Fig. 16.4 Client interacting with server and Web server. Step 1: The *get* request, **`GET /books/downloads.htm HTTP/1.1`**. (Part 1 of 2.)

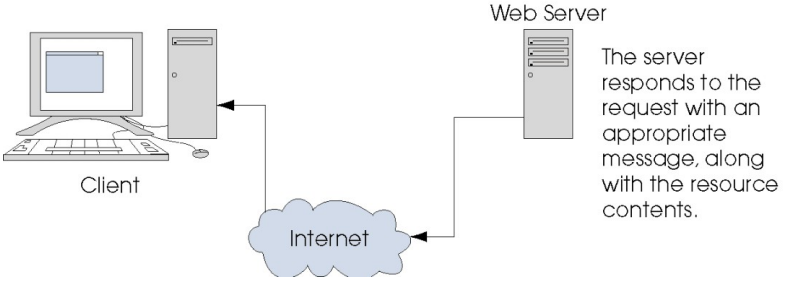


Fig. 16.4 Client interacting with server and Web server. Step 2: The HTTP response, **`HTTP/1.1 200 OK`**. (Part 2 of 2.)

server first responds with a line indicating the HTTP version, followed by a numeric code and a phrase describing the status of the transaction. For example,

```
HTTP/1.1 200 OK
```

indicates success;

```
HTTP/1.1 404 Not found
```

informs the client that the requested resource was not found on the server in the specified location.

The server then sends one or more *HTTP headers,* which provide information about the data being sent to the client. In this case, the server is sending an XHTML document, so the HTTP header reads

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

The information in the **Content-Type** header identifies the *MIME (Multipurpose Internet Mail Extensions) type* of the content. Each type of data sent from the server has a MIME type by which the browser determines how to process the data it receives. For example, the MIME type **text/plain** indicates that the data contains text that should be displayed without attempting to interpret any of the content as XHTML markup. Similarly, the MIME type **image/gif** indicates that the content is a GIF image. When this MIME type is received by the browser, it attempts to display the data as an image.

The headers are followed by a blank line, which indicates to the client that the server is finished sending HTTP headers. The server then sends the contents of the requested XHTML document (e.g., **downloads.html**). The connection is terminated when the transfer of the resource is complete. The client-side browser interprets the XHTML it receives and renders (or displays) the results.

16.9 Simple CGI Script

As long as an XHTML file on the server remains unchanged, its associated URL will display the same content in clients' browsers each time the file is accessed. For that content to change (e.g., to include new links or the latest company news), someone must alter the file manually on the server, probably with a text editor or Web-page-design software.

This need for manual change is a problem for Web page authors who want to create interesting and dynamic Web pages. To have a person continually alter a Web page is tedious. For example, if you want your Web page always to display the current date or weather conditions, the page would require continuous updating.

It is fairly straightforward to write a C++ program that outputs the current time and date (to the monitor of the local computer). In fact, this requires only a few lines of code:

```
time_t currentTime;     // time_t defined in <ctime>
time( &currentTime );
// asctime and localtime defined in <ctime>
cout << asctime( localtime( &currentTime ) );
```

C++ library function ***localtime***, when passed a **time_t** variable (e.g., **currentTime**) returns a pointer to a structure containing the "broken-down" local time (i.e., days,

hours, etc. are placed in individual structure members). Function **asctime**, which takes a pointer to a structure containing "broken-down" time, returns a string such as

```
Wed Jul 31 13:10:37 2002
```

What if we wish to send the current time to a client's browser window for display (rather than outputting it to the screen)? CGI makes this possible by allowing the server to redirect the output of a program to the Web server itself, sending the output to a client's browser. Redirection of output allows output (e.g., from a **cout** statement) to be sent somewhere other than the screen.

Figure 16.5 shows the full program listing for our first CGI script. Note that the program consists mainly of **cout** statements (lines 15–29). Until now, the output of **cout** always has been displayed on the screen. However, technically speaking, the default target for **cout** is *standard output*. When a C++ program is executed as a CGI script, the standard output is redirected by the Web server to the client Web browser. To execute the program, we placed the compiled C++ executable file in the Web server's **cgi-bin** directory. For the purpose of this chapter, we have changed the executable file extension from **.exe** to **.cgi**.[5] Assuming that the Web server is on your local computer, you can execute the script by typing

```
http://localhost/cgi-bin/localtime.cgi
```

in your browser's **Address** or **Location** field. If you are requesting this script from a remote Web server, you will need to replace **localhost** with the server's hostname or IP address.

```
1  // Fig. 16.5: localtime.cpp
2  // Displays the current date and time in a Web browser.
3
4  #include <iostream>
5
6  using std::cout;
7
8  #include <ctime>
9
10 int main()
11 {
12    time_t currentTime;  // variable for storing time
13
14    // output header
15    cout << "Content-Type: text/html\n\n";
16
17    // output XML declaration and DOCTYPE
18    cout << "<?xml version = \"1.0\"?>"
19         << "<!DOCTYPE html PUBLIC \"-//W3C//DTD XHTML 1.0 "
20         << "Transitional//EN\" \"http://www.w3.org/TR/xhtml1"
21         << "/DTD/xhtml1-transitional.dtd\">";
```

Fig. 16.5 First CGI script. (Part 1 of 2.)

5. On a server running Microsoft Windows, the executable may be run directly in **.exe** form.

```
22
23     time( &currentTime );  // store time in currentTime
24
25     // output html element and some of its contents
26     cout << "<html xmlns = \"http://www.w3.org/1999/xhtml\">"
27          << "<head><title>Current date and time</title></head>"
28          << "<body><p>" << asctime( localtime( &currentTime ) )
29          << "</p></body></html>";
30
31     return 0;
32
33  } // end main
```

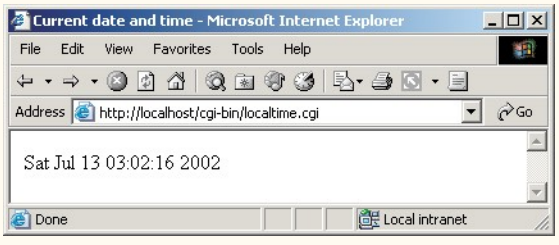


Fig. 16.5 First CGI script. (Part 2 of 2.)

The notion of standard output is similar to that of standard input, which we have seen frequently referenced with the expression **cin**. Just as standard input refers to the standard method of input into a program (normally, the keyboard), standard output refers to the standard method of output from a program (normally, the screen). It is possible to redirect (or *pipe*) standard output to another destination. Thus, in our CGI script, when we output an HTTP header (line 15) or XHTML elements (lines 18–21 and 26–29), the output is sent to the Web server, as opposed to the screen. The server sends that output to the client over HTTP, which interprets the headers and elements as if they were part of a normal server response to an XHTML document request.

Figure 16.6 illustrates this process in more detail. In Step 1, the client requests the resource named **localtime.cgi** from the server, just as it requested **downloads.html** in the previous example. If the server was not configured to handle CGI scripts, it might just return the contents of the C++ executable file to the client, as if it were any other document. However, based on the Web server configuration, the server executes **localtime.cgi** and sends the CGI program's output to the Web browser.

A properly configured Web server, however, will recognize that certain resources should be handled differently. For example, when the resource is a CGI script, the script must be executed by the server. A resource usually is designated as a CGI script in one of two ways: either it has a special filename extension (such as **.cgi** or **.exe**) or it is located in a specific directory (often **cgi-bin**). In addition, the server administrator must give permission explicitly for remote clients to be able to access and execute CGI scripts.[6]

6. If you are using the Apache HTTP Server and would like more information on configuration, consult the Apache home page at **www.apache.org**.

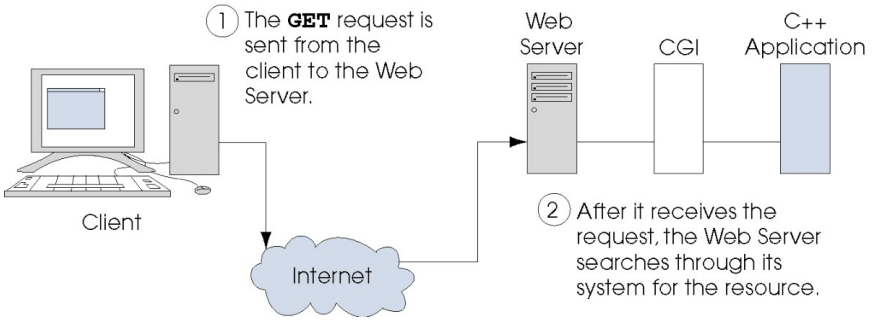


Fig. 16.6 Step 1: The *get* request, **`GET /cgi-bin/localtime.cgi HTTP/1.1`**. (Part 1 of 4.)

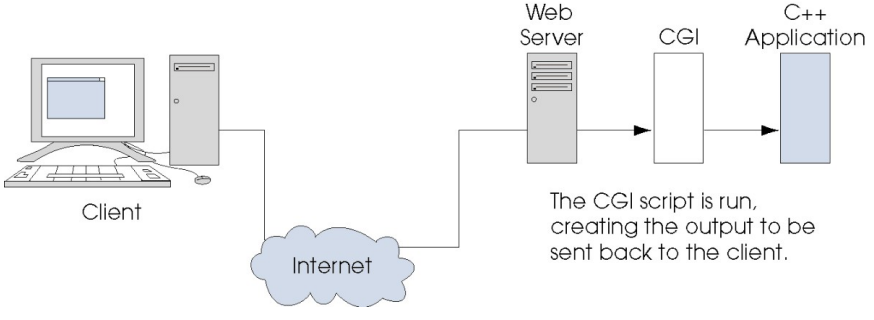


Fig. 16.6 Step 2: The Web server starts the CGI script. (Part 2 of 4.)

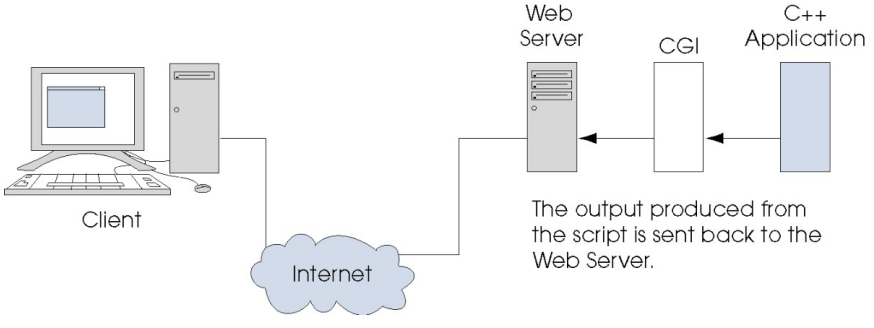


Fig. 16.6 Step 3: The output of the script is sent to the Web server. (Part 3 of 4.)

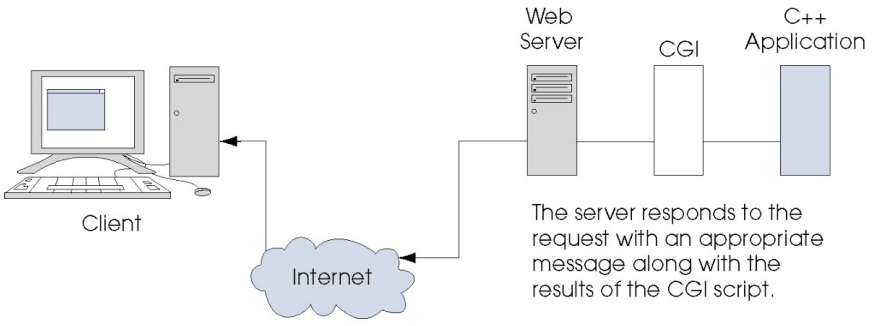


Fig. 16.6 Step 4: The HTTP response, **`HTTP/1.1 200 OK`**. (Part 4 of 4.)

In Step 2 of Fig. 16.6, the server recognizes that the resource is a CGI script and executes the script. In Step 3, the three **`cout`** statements (lines 15, 18–21 and 26–29 of Fig. 16.5) are executed, and the text is sent to the standard output and is returned to the Web server. Finally, in Step 4, the Web server adds a message to the output that indicates the status of the HTTP transaction (such as **`HTTP/1.1 200 OK`**, for success) and sends the entire output from the CGI program to the client.

The client-side browser then processes the XHTML output and displays the results. It is important to note that the browser is unaware of what has transpired on the server. In other words, as far as the browser is concerned, it requests a resource like any other and receives a response like any other. The client receives and interprets the script's output, just as if it were a simple, static XHTML document.

In fact, you can view the content that the browser receives by executing **`localtime.cgi`** from the command line, as we normally would execute any of the programs from the previous chapters. [*Note:* The file extension must be changed to **`.exe`** prior to executing from the command line on a system running Windows]. Figure 16.7 shows the output. For the purpose of this chapter, we formatted the output for readability.

```
Content-Type: text/html

<?xml version = "1.0"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns = "http://www.w3.org/1999/xhtml">
   <head>
      <title>Current date and time</title>
   </head>

   <body>
      <p>Mon Jul 15 13:52:45 2002</p>
   </body>
</html>
```

Fig. 16.7 Output of **`localtime.cgi`** when executed from the command line.

Notice that, with the CGI script, we must output the **`Content-Type`** header, whereas, for an XHTML document, the Web server would include the header.

To review, a CGI program prints the **`Content-Type`** header, a blank line and the data (XHTML, plain text, etc.) to standard output. The Web server retrieves this output, inserts the HTTP response to the beginning and delivers the content to the client. Later, we will see other content types that may be used in this manner, as well as other headers that may be used in addition to **`Content-Type`**.

The program of Figure 16.8 outputs the *environment variables* that the Web server provides when executing the CGI script. These variables contain information about the client and server environment, such as the type of Web browser being used and the location of the document on the server. Lines 15–24 initialize an array of **`string`** objects with the CGI environment variable names. Line 41 begins the XHTML table in which the data will be displayed.

Lines 45–48 output each row of the table. Let us examine each of these lines closely. Line 45 outputs an XHTML **`<tr>`** (table row) start tag, which indicates the beginning of a new table row. Line 48 outputs its corresponding **`</tr>`** end tag, which indicates the end of the row. Each row of the table contains two table cells. Each row contains the name of an environment variable and the data associated with that variable. The **`<td>`** start tag (line 45) begins a new table cell. The **`for`** loop (line 44) iterates through each of the 24 **`string`** objects. Each environment variable's name is output in the left table cell. The value associated with the environment variable is output by calling

```
1  // Fig. 16.8: environment.cpp
2  // Program to display CGI environment variables.
3  #include <iostream>
4
5  using std::cout;
6
7  #include <string>
8
9  using std::string;
10
11 #include <cstdlib>
12
13 int main()
14 {
15    string environmentVariables[ 24 ] = {
16       "COMSPEC", "DOCUMENT_ROOT", "GATEWAY_INTERFACE",
17       "HTTP_ACCEPT", "HTTP_ACCEPT_ENCODING",
18       "HTTP_ACCEPT_LANGUAGE", "HTTP_CONNECTION",
19       "HTTP_HOST", "HTTP_USER_AGENT", "PATH",
20       "QUERY_STRING", "REMOTE_ADDR", "REMOTE_PORT",
21       "REQUEST_METHOD", "REQUEST_URI", "SCRIPT_FILENAME",
22       "SCRIPT_NAME", "SERVER_ADDR", "SERVER_ADMIN",
23       "SERVER_NAME","SERVER_PORT","SERVER_PROTOCOL",
24       "SERVER_SIGNATURE","SERVER_SOFTWARE" };
25
26    // output header
27    cout << "Content-Type: text/html\n\n";
```

Fig. 16.8 Retrieving environment variables via function **`getenv`**. (Part 1 of 3.)

```
28
29       // output XML declaration and DOCTYPE
30       cout << "<?xml version = \"1.0\"?>"
31            << "<!DOCTYPE html PUBLIC \"-//W3C//DTD XHTML 1.0 "
32            << "Transitional//EN\" \"http://www.w3.org/TR/xhtml1"
33            << "/DTD/xhtml1-transitional.dtd\">";
34
35       // output html element and some of its contents
36       cout << "<html xmlns = \"http://www.w3.org/1999/xhtml\">"
37            << "<head><title>Environment Variables</title></head>"
38            << "<body>";
39
40       // begin outputting table
41       cout << "<table border = \"0\" cellspacing = \"2\">";
42
43       // iterate through environment variables
44       for ( int i = 0; i < 24; i++ )
45          cout << "<tr><td>" << environmentVariables[ i ]
46               << "</td><td>"
47               << getenv( environmentVariables[ i ].data() )
48               << "</td></tr>";
49
50       cout << "</table></body></html>";
51
52       return 0;
53
54    } // end main
```

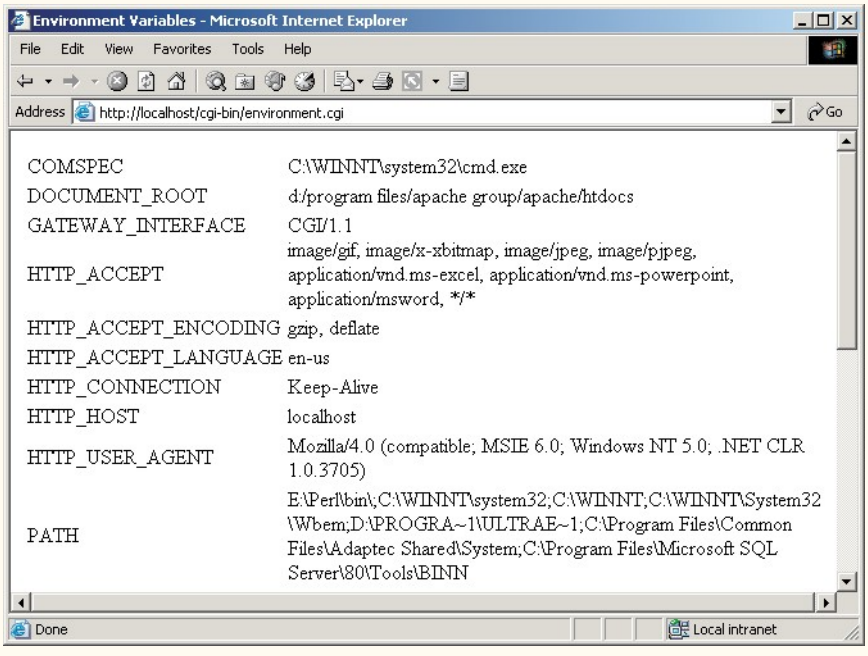


Fig. 16.8 Retrieving environment variables via function **`getenv`**. (Part 2 of 3.)

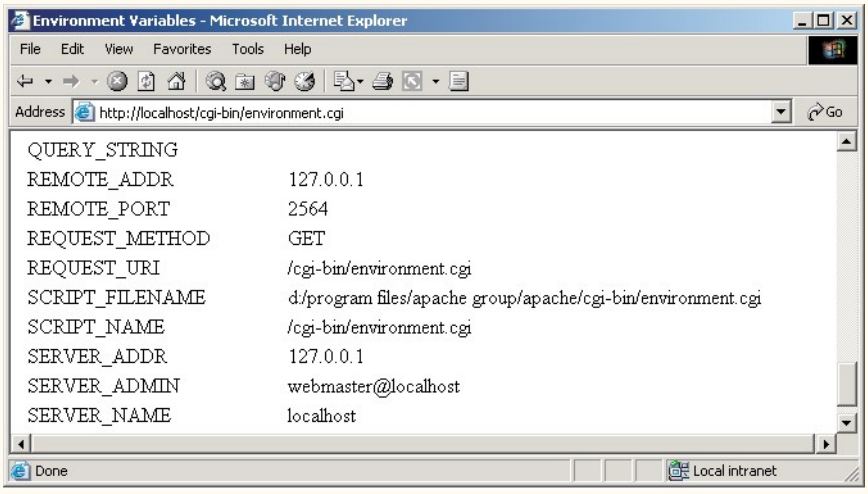


Fig. 16.8 Retrieving environment variables via function **`getenv`**. (Part 3 of 3.)

function ***`getenv`*** of **`<cstdlib>`** and passing it the string value returned from the function call **`environmentVariables[ i ].data()`**. Function **`data`** returns a C-style **`char *`** string containing the contents of the **`environmentVariables[ i ]`** `string`.

Common Programming Error 16.1

Forgetting to place a blank line after a header is a logic error.

16.10 Sending Input to a CGI Script

Though preset environment variables provide much information, we would like to be able to supply any type of data to our CGI scripts, such as a user's name or a search-engine query. The environment variable ***`QUERY_STRING`*** provides a mechanism to do just that. The **`QUERY_STRING`** variable contains information that is appended to a URL in a *get* request. For example, the URL

`www.`*somesite*`.com/cgi-bin/script.cgi?state=California`**

causes the Web browser to request a resource from **`www.`***somesite*`.com`**. The resource is a CGI script (**`cgi-bin/script.cgi`**). The Web server stores the data following the **`?`** (**`state=California`**) in the **`QUERY_STRING`** environment variable. The query string provides parameters that customize the request for a particular client. Note that the question mark is not part of the resource requested, nor is it part of the query string. It serves as a delimiter (or separator) between the two.

Figure 16.9 shows a simple example of a CGI script that reads data passed through the **`QUERY_STRING`**. Note that data in a query string can be formatted in a variety of ways. The CGI script reading the string must know how to interpret the formatted data. In the example in Fig. 16.9, the query string contains a series of name-value pairs delimited by ampersands (**`&`**), as in

`name=Jill&age=22`

In line 15 of Figure 16.9, we pass **"QUERY_STRING"** to function **getenv**, which returns the query string and assigns it to **string** variable **query**. After outputting a header, some XHTML start tags and the title (lines 21–29), we test if **query** contains data (line 34). If not, we output a message instructing the user to add a query string to the URL. We also provide a link to a URL that includes a sample query string. Query-string data may be specified as part of a hyperlink in a Web page when encoded in this manner. The contents of the query string are output on line 42.

```
1  // Fig. 16.9: querystring.cpp
2  // Demonstrating QUERY_STRING.
3  #include <iostream>
4
5  using std::cout;
6
7  #include <string>
8
9  using std::string;
10
11 #include <cstdlib>
12
13 int main()
14 {
15    string query = getenv( "QUERY_STRING" );
16
17    // output header
18    cout << "Content-Type: text/html\n\n";
19
20    // output XML declaration and DOCTYPE
21    cout << "<?xml version = \"1.0\"?>"
22         << "<!DOCTYPE html PUBLIC \"-//W3C//DTD XHTML 1.0 "
23         << "Transitional//EN\" \"http://www.w3.org/TR/xhtml1"
24         << "/DTD/xhtml1-transitional.dtd\">";
25
26    // output html element and some of its contents
27    cout << "<html xmlns = \"http://www.w3.org/1999/xhtml\">"
28         << "<head><title>Name/Value Pairs</title></head>"
29         << "<body>";
30
31    cout << "<h2>Name/Value Pairs</h2>";
32
33    // if query contained no data
34    if ( query == "" )
35       cout << "Please add some name-value pairs to the URL "
36            << "above.<br/>Or try "
37            << "<a href=\"querystring.cgi?name=Joe&age=29\">"
38            << "this</a>.";
39
40    // user entered query string
41    else
42       cout << "<p>The query string is: " << query << "</p>";
43
```

Fig. 16.9 Reading input from **QUERY_STRING**. (Part 1 of 2.)

```
44      cout << "</body></html>";
45
46      return 0;
47
48   } // end main
```

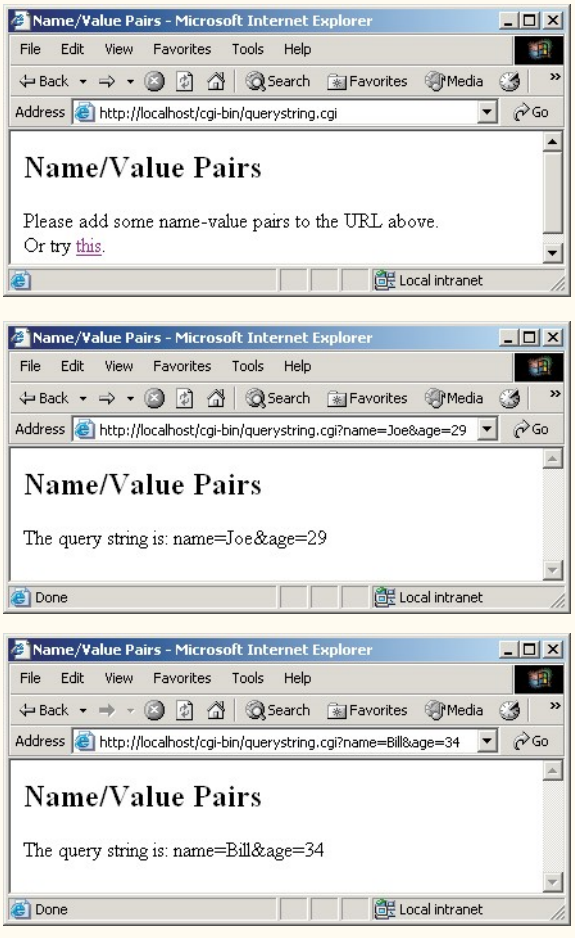


Fig. 16.9 Reading input from **`QUERY_STRING`**. (Part 2 of 2.)

16.11 Using XHTML Forms to Send Input

Having a client enter input directly into a URL is not a user-friendly approach. Fortunately, XHTML provides the ability to include *forms* on Web pages that provide a more intuitive way for users to input information to be sent to a CGI script.

The ***`form`*** element encloses an XHTML form. The **`form`** element generally takes two attributes. The first attribute is ***`action`***, which specifies the action to take when the user submits the form. For our purposes, the **`action`** usually will be to call a CGI script to process the form's data. The second attribute used in the **`form`** element is **`method`**. The method attribute identifies the type of HTTP request to use when the browser submits the

form to the Web server. In this section, we will show examples using both methods to illustrate them in detail.

An XHTML form may contain any number of form elements. Figure 16.10 gives a brief description of several form elements.

Figure 16.11 demonstrates a basic XHTML form using the HTTP *get* method. The form is output in lines 35–38 with the **`form`** element. Notice that attribute **`method`** has the value **`"get"`** and attribute **`action`** has the value **`"getquery.cgi"`** (i.e., the script actually calls itself to handle the form data once they are submitted).

The form contains two **`input`** fields. The first (line 36) is a single-line text field (**`type = "text"`**) named **`word`**. The second (line 37) displays a button, labeled **Submit Word**, to submit the form data.

Element name	`type` attribute value (for `input` elements)	Description
`input`	`text`	Provides a single-line text field for text input.
	`password`	Like **`text`**, but each character typed by the user appears as an asterisk (**`*`**).
	`checkbox`	Displays a checkbox that can be checked (true) or unchecked (false).
	`radio`	Radio buttons are like checkboxes, except that only one radio button in a group of radio buttons can be selected at a time.
	`button`	A push button.
	`submit`	A push button that submits form data according to the form's **`action`**.
	`image`	The same as **`submit`**, but displays an image rather than a push button.
	`reset`	A push button that resets form fields to their default values.
	`file`	Displays a text field and button that allow the user to specify a file to upload to a Web server. When clicked, the button opens a file dialog that allows the user to select a file.
	`hidden`	Hidden form data that can be used by the form handler on the server. These **`input`**s are not visible to the user.
`select`		Drop-down menu or selection box. This element is used with the **`option`** element to specify a series of selectable items.
`textarea`		This is a multiline text field in which text can be input or displayed.

Fig. 16.10 XHTML form elements.

```
// Fig. 16.11: getquery.cpp
// Demonstrates GET method with XHTML form.
#include <iostream>

using std::cout;

#include <string>

using std::string;

#include <cstdlib>

int main()
{
   string nameString = "";
   string wordString = "";
   string query = getenv( "QUERY_STRING" );

   // output header
   cout << "Content-Type: text/html\n\n";

   // output XML declaration and DOCTYPE
   cout << "<?xml version = \"1.0\"?>"
        << "<!DOCTYPE html PUBLIC \"-//W3C//DTD XHTML 1.0 "
        << "Transitional//EN\" \"http://www.w3.org/TR/xhtml1"
        << "/DTD/xhtml1-transitional.dtd\">";

   // output html element and some of its contents
   cout << "<html xmlns = \"http://www.w3.org/1999/xhtml\">"
        << "<head><title>Using GET with Forms</title></head>"
        << "<body>";

   // output xhtml form
   cout << "<p>Enter one of your favorite words here:</p>"
        << "<form method = \"get\" action = \"getquery.cgi\">"
        << "<input type = \"text\" name = \"word\"/>"
        << "<input type = \"submit\" value = \"Submit Word\"/>"
        << "</form>";

   // query is empty
   if ( query == "" )
      cout << "<p>Please enter a word.</p>";

   // user entered query string
   else {
      int wordLocation = query.find_first_of( "word=" ) + 5;

      wordString = query.substr( wordLocation );

      // no word was entered
      if ( wordString == "" )
         cout << "<p>Please enter a word.</p>";

```

Fig. 16.11 Using **GET** with an XHTML form. (Part 1 of 3.)

```
54          // word was entered
55          else
56             cout << "<p>Your word is: " << wordString << "</p>";
57       }
58
59       cout << "</body></html>";
60
61       return 0;
62
63    } // end main
```

Fig. 16.11 Using **GET** with an XHTML form. (Part 2 of 3.)

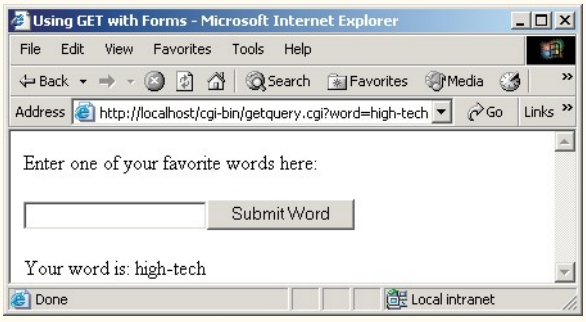


Fig. 16.11 Using **GET** with an XHTML form. (Part 3 of 3.)

The first time the script is executed, there should be no value in **QUERY_STRING** (unless the user has appended the query string to the URL). Once the user enters a word into the **word** field and presses **Submit Word**, the script is requested again. This time, the name of the input field (**word**) and the value entered by the user are placed in the **QUERY_STRING** variable by the browser. That is, if the user enters the word "**technology**" and presses the **Submit Word**, **QUERY_STRING** is assigned the value **word=technology** and the query string is appended to the URL in the browser window.

During the second execution of the script, the query string is decoded. Lines 46–48 in Fig. 16.11 search **query** for the first occurrence of **word=**, using **string** method **find_first_of**, which returns an integer value corresponding to the location in the **string** where the first match was found. A value of **5** is added to the location to move the position in the **string** to the first character of the user's favorite word. Method **substr** (line 48) returns the remainder of the **string** starting at the location specified by **wordLocation**, which is then assigned to **wordString**. Line 51 determines whether the user entered a word. If so, line 56 outputs the word entered by the user.

The two previous examples used *get* to pass data to the CGI scripts through an environment variable. Web browsers typically interact with Web servers by submitting forms using HTTP *post*. CGI programs read the contents of *post* requests using standard input. For comparison purposes, let us now reimplement the application of Fig. 16.11, using **POST** (as in Fig. 16.12). Notice that the code in the two figures is virtually identical. The XHTML form indicates that we are now using the **POST** method to submit the form data.

```
1  // Fig. 16.12: post.cpp
2  // Demonstrates POST method with XHTML form.
3  #include <iostream>
4
5  using std::cout;
6  using std::cin;
7
8  #include <string>
9
10 using std::string;
11
```

Fig. 16.12 Using **POST** with an XHTML form. (Part 1 of 4.)

```
#include <cstdlib>

int main()
{
   char postString[ 1024 ] = ""; // variable to hold POST data
   string dataString = "";
   string nameString = "";
   string wordString = "";
   int contentLength = 0;

   // content was submitted
   if ( getenv( "CONTENT_LENGTH" ) ) {
      contentLength = atoi( getenv( "CONTENT_LENGTH" ) );

      cin.read( postString, contentLength );
      dataString = postString;
   } // end if

   // output header
   cout << "Content-Type: text/html\n\n";

   // output XML declaration and DOCTYPE
   cout << "<?xml version = \"1.0\"?>"
        << "<!DOCTYPE html PUBLIC \"-//W3C//DTD XHTML 1.0 "
        << "Transitional//EN\" \"http://www.w3.org/TR/xhtml1"
        << "/DTD/xhtml1-transitional.dtd\">";

   // output XHTML element and some of its contents
   cout << "<html xmlns = \"http://www.w3.org/1999/xhtml\">"
        << "<head><title>Using POST with Forms</title></head>"
        << "<body>";

   // output XHTML form
   cout << "<p>Enter one of your favorite words here:</p>"
        << "<form method = \"post\" action = \"post.cgi\">"
        << "<input type = \"text\" name = \"word\" />"
        << "<input type = \"submit\" value = \"Submit Word\" />"
        << "</form>";

   // data was sent using POST
   if ( contentLength > 0 ) {
      int nameLocation =
         dataString.find_first_of( "word=" ) + 5;

      int endLocation = dataString.find_first_of( "&" ) - 1;

      // retrieve entered word
      wordString = dataString.substr( nameLocation,
         endLocation - nameLocation );

      // no data was entered in text field
      if ( wordString == "" )
         cout << "<p>Please enter a word.</p>";
```

Fig. 16.12 Using **POST** with an XHTML form. (Part 2 of 4.)

```
65
66          // output word
67          else
68             cout << "<p>Your word is: " << wordString << "</p>";
69
70       } // end if
71
72       // no data was sent
73       else
74          cout << "<p>Please enter a word.</p>";
75
76       cout << "</body></html>";
77
78       return 0;
79
80    } // end main
```

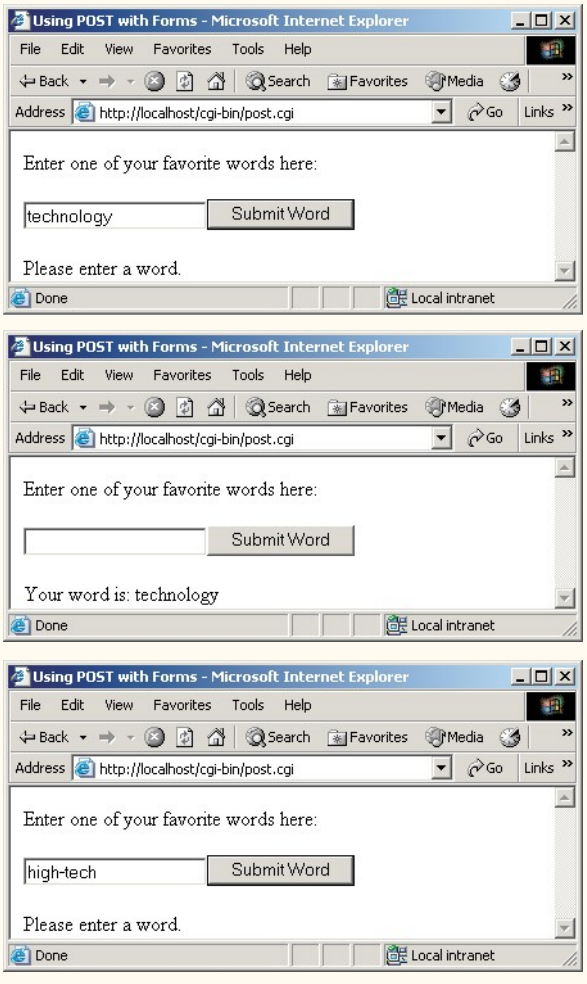


Fig. 16.12 Using **POST** with an XHTML form. (Part 3 of 4.)

Fig. 16.12 Using **`POST`** with an XHTML form. (Part 4 of 4.)

The Web server sends *post* data to a CGI script via standard input. The data is encoded (i.e., formatted) just as in **`QUERY_STRING`** (that is, with name-value pairs connected by equals signs and ampersands), but the **`QUERY_STRING`** environment variable is not set. Instead, the **`POST`** method sets the environment variable ***`CONTENT_LENGTH`***, to indicate the number of characters of data that were sent in the *post* requests.

The value of the **`CONTENT_LENGTH`** environment variable is used by the CGI script to process the correct amount of data. Line 23 determines whether **`CONTENT_LENGTH`** contains a value. Line 24 reads in the value and converts it to an integer by calling **`<cstdlib>`** function **`atoi`**. Line 26 calls function **`cin.read`** to read characters from standard input and stores the characters in array **`postString`**. Line 27 converts **`postString`**'s data to a **`string`** by assigning it to **`dataString`**.

In earlier chapters, we read data from standard input using an expression such as

```
cin >> data;
```

The same approach might work in our CGI script as a replacement for the **`cin.read`** statement. Recall that **`cin`** reads data from standard input up to and including the first newline character, space or tab, whichever comes first. The CGI specification does not require a newline to be appended after the last name-value pair. Although some browsers append a newline or **`EOF`**, they are not required to do so. If **`cin`** is used with a browser that sends only the name-value pairs (as per the CGI specification), **`cin`** must wait for a newline that will never arrive. In this case, the server eventually "times out" and the CGI script terminates. Therefore, **`cin.read`** is preferred over **`cin`**, because the programmer can specify exactly how much data to read.

The CGI scripts from this section, while useful for explaining how *get* and *post* operate, do not include many of the features described in the CGI specification. For example, if we enter the words **`didn't translate`** into the text field and click the **`submit`** button, the script informs us that our word is **`didn%27t+translate`**.

What has happened here? Web browsers *URL encode* the XHTML form data they send. This means that spaces are replaced with plus signs, and other symbols (e.g., apostrophes) are translated into their ASCII value in hexadecimal format and preceded with a percent sign. URL encoding is necessary because URLs do not allow certain characters, such as spaces and apostrophes.

16.12 Other Headers

We mentioned in Section 16.9 that there are several HTTP headers in addition to the **Content-Type** header. A CGI script can supply other HTTP headers in addition to **Content-Type**. In most cases, the server passes these extra headers to the client without executing them. For example, the following ***Refresh*** *header* redirects the client to a new location after a specified amount of time:

```
Refresh: "5; URL = http://www.deitel.com/newpage.html"
```

Five seconds after the Web browser receives this header, the browser requests the resource at the specified URL. Alternatively, the **Refresh** header can omit the URL, in which case it will refresh the current page after the given time has expired.

The CGI specification indicates that certain types of headers output by a CGI script are to be handled by the server, rather than be passed directly to the client. The first of these is the ***Location*** *header*. Like **Refresh**, **Location** redirects the client to a new location:

```
Location: http://www.deitel.com/newpage.html
```

If used with a relative (or virtual) URL (i.e., **Location: /newpage.html**), the **Location** header indicates to the server that the redirection is to be performed on the server side without sending the **Location** header back to the client. In this case, it appears to the user as if the document rendered in their Web browser was the resource they requested.

The CGI specification also includes a ***Status*** *header*, which instructs the server to output a corresponding status header line (such as **HTTP/1.1 200 OK**). Normally, the server will send the appropriate status line to the client (adding, for example, the **200 OK** status line in most cases). However, CGI allows programmers to change the response status. For example, sending a

```
Status: 204 No Response
```

header indicates that, although the request was successful, the client should not display a new page in the browser window. This header might be useful if you want to allow users to submit forms without relocating to a new page.

We have now covered the fundamentals of the CGI specification. To review, CGI allows scripts to interact with servers in three basic ways:

1. through the output of headers and content to the server via standard output;
2. by the server's setting of environment variables (including the URL-encoded **QUERY_STRING**) whose values are available within the script (via **getenv**); and
3. through **POST**ed, URL-encoded data that the server sends to the script's standard input.

16.13 Case Study: An Interactive Web Page

Figure 16.13 and Fig. 16.14 show the implementation of a simple interactive portal for the fictional Bug2Bug Travel Web site. The example queries the client for a name and password, then displays information about weekly travel specials based on the data entered. For simplicity, the example does not encrypt the data sent to the server.

Figure 16.13 displays the opening page. It is a static XHTML document containing a form that **POST**s data to the **portal.cgi** CGI script (line 16). The form contains one field each to collect the user's name (line 18) and the user's password (line 19). [*Note:* This XHTML document was placed in the document directory of the Web server.]

Figure 16.14 contains the CGI script. First, let us examine how the data is retrieved from standard input and stored in **string**s. The **string** library **find** function searches **dataString** (line 30) for an occurrence of **namebox=**. Function **find** returns a location in the string where **namebox=** was found. To retrieve the value associated with

```
1  <?xml version = "1.0"?>
2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
3  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
4
5  <!-- Fig. 16.13: travel.html    -->
6  <!-- Bug2Bug Travel Homepage    -->
7
8  <html xmlns = "http://www.w3.org/1999/xhtml">
9     <head>
10       <title>Bug2Bug Travel</title>
11    </head>
12
13    <body>
14       <h1>Welcome to Bug2Bug Travel</h1>
15
16       <form method = "post" action = "/cgi-bin/portal.cgi">
17          <p>Please enter your name:</p>
18          <input type = "text" name = "namebox" />
19          <input type = "password" name = "passwordbox" />
20          <p>password is not encrypted</p>
21          <input type = "submit" name = "button" />
22       </form>
23
24    </body>
25 </html>
```

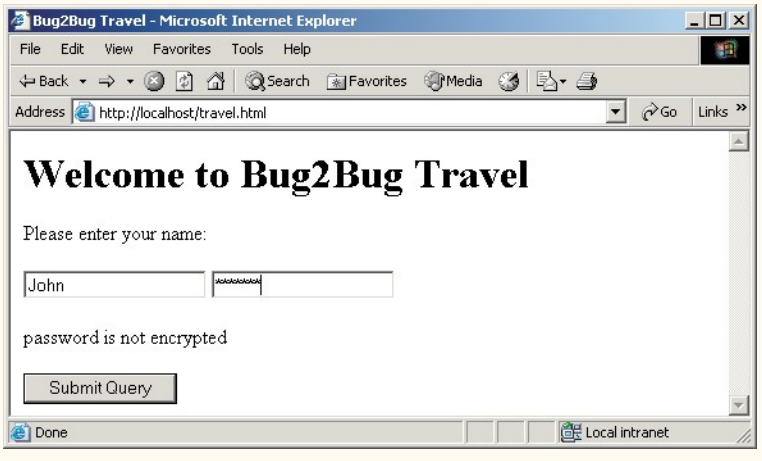


Fig. 16.13 Interactive portal to create a password-protected Web page.

namebox=—the value entered by the user—the position in the string moves forward **8** characters. Recall that a query string contains name-value pairs separated by equals signs and ampersands. To find the ending location for the data we wish to retrieve, we search for the **&** character on line 31. The program now contains an integer "pointing" to the starting location. The length of the entered word is determined by the calculation **endNamelocation - namelocation**. On lines 37–41, we assign the form-field values to variables **nameString** and **passwordString**. We use **nameString** in line 58 to output a personalized greeting to the user. The current weekly specials are displayed in lines 58–62. (In this example, we include this information as part of the script.)

If the member password is correct, additional specials are output (lines 66–67). If the password is incorrect, the client is informed that the password was invalid.

Note that we use a combination of a static Web page and a CGI script here. We could have incorporated the opening XHTML form and the processing of the data into a single CGI script, as we did in previous examples in this chapter. We ask the reader to do this in Exercise 16.8.

Performance Tip 16.1

It is always much more efficient for the server to provide static content rather than execute a CGI script, because it takes time for the server to load the script from hard disk into memory and execute the script (whereas an XHTML file needs to be sent only to the client). It is a good practice to use a mix of static XHTML (for content that generally remains unchanged) and CGI scripting (for dynamic content). This practice allows the Web server to respond to clients more efficiently than if only CGI scripting were used.

```
1  // Fig. 16.14: portal.cpp
2  // Handles entry to Bug2Bug Travel.
3  #include <iostream>
4
5  using std::cout;
6  using std::cin;
7
8  #include <string>
9
10 using std::string;
11
12 #include <cstdlib>
13
14 int main()
15 {
16    char postString[ 1024 ] = "";
17    string dataString = "";
18    string nameString = "";
19    string passwordString = "";
20    int contentLength = 0;
21
22    // data was posted
23    if ( getenv( "CONTENT_LENGTH" ) )
24       contentLength = atoi( getenv( "CONTENT_LENGTH" ) );
25
```

Fig. 16.14 Interactive portal handler. (Part 1 of 3.)

```
26      cin.read( postString, contentLength );
27      dataString = postString;
28
29      // search string for input data
30      int namelocation = dataString.find( "namebox=" ) + 8;
31      int endNamelocation = dataString.find( "&" );
32
33      int password = dataString.find( "passwordbox=" ) + 12;
34      int endPassword = dataString.find( "&button" );
35
36      // get values for name and password
37      nameString = dataString.substr( namelocation,
38         endNamelocation - namelocation );
39
40      passwordString = dataString.substr( password, endPassword -
41         password );
42
43      // output header
44      cout << "Content-Type: text/html\n\n";
45
46      // output XML declaration and DOCTYPE
47      cout << "<?xml version = \"1.0\"?>"
48           << "<!DOCTYPE html PUBLIC \"-//W3C//DTD XHTML 1.0 "
49           << "Transitional//EN\" \"http://www.w3.org/TR/xhtml1"
50           << "/DTD/xhtml1-transitional.dtd\">";
51
52      // output html element and some of its contents
53      cout << "<html xmlns = \"http://www.w3.org/1999/xhtml\">"
54           << "<head><title>Bug2Bug Travel</title></head>"
55           << "<body>";
56
57      // output specials
58      cout << "<h1>Welcome " << nameString << "!</h1>"
59           << "<p>Here are our weekly specials:</p>"
60           << "<ul><li>Boston to Taiwan ($875)</li>"
61           << "<li>San Diego to Hong Kong ($750)</li>"
62           << "<li>Chicago to Mexico City ($568)</li></ul>";
63
64      // password is correct
65      if ( passwordString == "coast2coast" )
66         cout << "<hr /><p>Current member special: "
67              << "Seattle to Tokyo ($400)</p>";
68
69      // password was incorrect
70      else
71         cout << "<p>Sorry. You have entered an incorrect "
72              << "password</p>";
73
74      cout << "</body></html>";
75      return 0;
76
77   } // end main
```

Fig. 16.14 Interactive portal handler. (Part 2 of 3.)

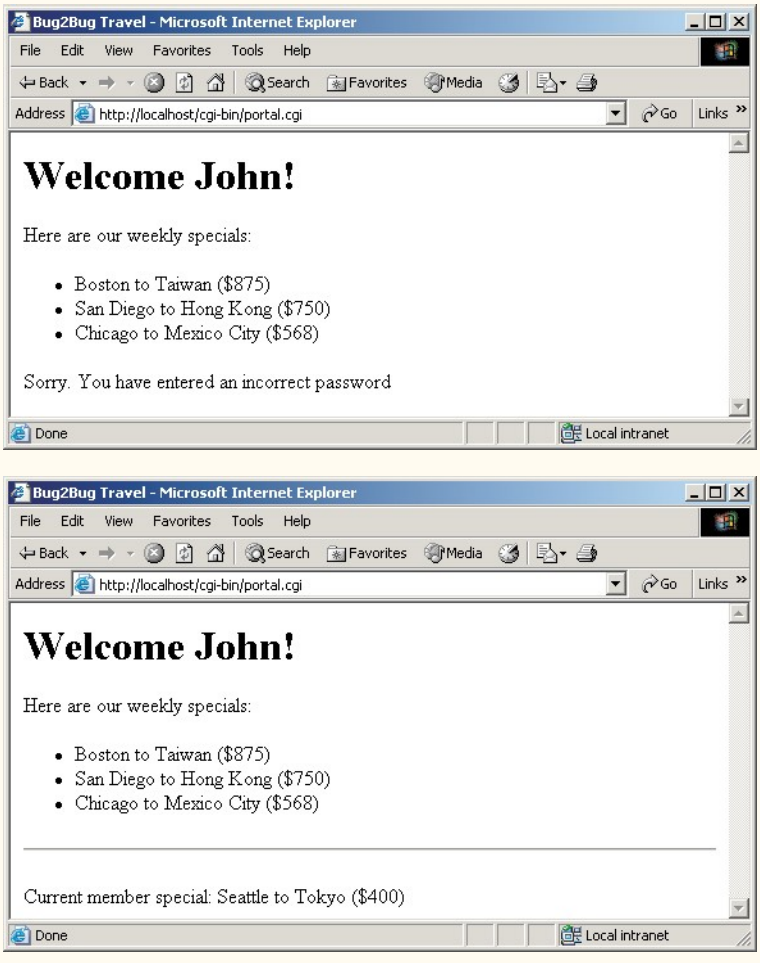


Fig. 16.14 Interactive portal handler. (Part 3 of 3.)

16.14 Cookies

In the last two sections, we discussed two ways in which information may be passed between programs (or executions of the same program) through a browser. This section concentrates on storing state information on the client computer with *cookies*. Cookies are essentially small text files that a Web server sends to your browser, which then writes the cookies onto your computer. Many Web sites use cookies to track a user's progress through their site (as in a shopping-cart application) or to help customize the site for an individual user.

Cookies do not break into your computer, nor do they erase your hard drive. However, they can be used to identify users and keep track of how often users visit a site or what users buy at a site. For this reason, cookies are considered to be a security and privacy concern. Popular Web browsers provide support for cookies. These browsers also allow users who are concerned about their privacy and security to disable this support. Most major Web sites use cookies. As a programmer, you should be aware of the possibility that cookies might

be disabled by your clients. Figure 16.15, Fig. 16.16 and Fig. 16.17 use cookies to store and manipulate information about a user.

Figure 16.15 is an XHTML page that contains a form in which values are to be input. The form posts its information to **writecookie.cgi** (Fig. 16.16). This program retrieves the data contained in the **CONTENT_LENGTH** environment variable. Line 24 of Fig. 16.16 declares and initializes **string expires** to store the expiration date of the cookie (i.e., how long the cookie resides on the client's machine). This value can be a string, like the one in this example, or it can be a relative value. For instance, **+30d** sets the cookie to exist for 30 days. For the purposes of this chapter the expiration date is deliberately set to expire in 2010 to ensure that the program will run properly well into the future. You may set the expiration date of this example to any future date as needed. The browser deletes cookies when they expire.

```
1  <?xml version = "1.0"?>
2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
3     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
4
5  <!-- Fig. 16.15: cookieform.html -->
6  <!-- Cookie Demonstration        -->
7
8  <html xmlns = "http://www.w3.org/1999/xhtml">
9     <head>
10       <title>Writing a cookie to the client computer</title>
11    </head>
12
13    <body>
14       <h1>Click Write Cookie to save your cookie data.</h1>
15
16       <form method = "post"
17          action = "/cgi-bin/writecookie.cgi">
18
19          <p>Name:<br />
20             <input type = "text" name = "name" />
21          </p>
22
23          <p>Age:<br />
24             <input type = "text" name = "age" />
25          </p>
26
27          <p>Favorite Color:<br />
28             <input type = "text" name = "color" />
29          </p>
30
31          <p>
32             <input type = "submit" name = "button" />
33          </p>
34       </form>
35
36    </body>
37 </html>
```

Fig. 16.15 XHTML document containing a form to post data to the server (Part 1 of 2.)

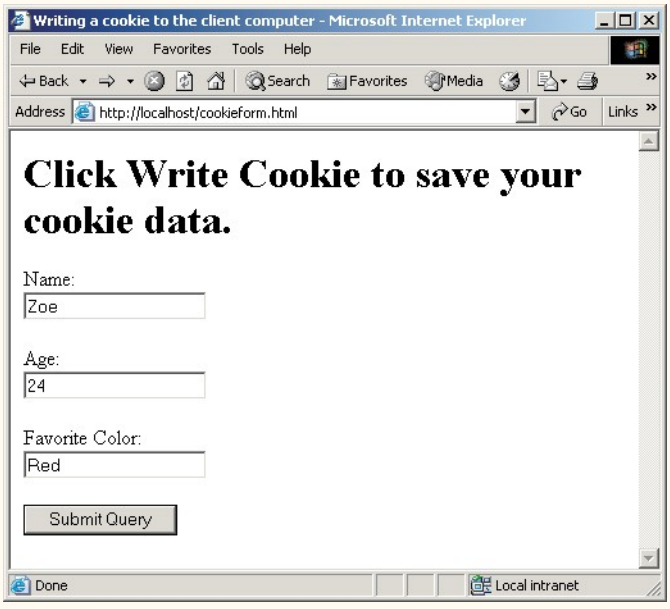


Fig. 16.15 XHTML document containing a form to post data to the server (Part 2 of 2.)

```
1  // Fig. 16.16: writecookie.cpp
2  // Program to write a cookie to a client's machine.
3  #include <iostream>
4
5  using std::cout;
6  using std::cin;
7
8  #include <cstdlib>
9  #include <string>
10
11 using std::string;
12
13 int main()
14 {
15    char query[ 1024 ] = "";
16    string dataString = "";
17    string nameString = "";
18    string ageString = "";
19    string colorString = "";
20
21    int contentLength = 0;
22
23    // expiration date of cookie
24    string expires = "Friday, 14-MAY-10 16:00:00 GMT";
25
```

Fig. 16.16 Writing a cookie. (Part 1 of 3.)

```
26      // data was entered
27      if ( getenv( "CONTENT_LENGTH" ) ) {
28         contentLength = atoi( getenv( "CONTENT_LENGTH" ) );
29
30         // read data from standard input
31         cin.read( query, contentLength );
32         dataString = query;
33
34         // search string for data and store locations
35         int nameLocation = dataString.find( "name=" ) + 5;
36         int endName = dataString.find( "&" );
37
38         int ageLocation = dataString.find( "age=" ) + 4;
39         int endAge = dataString.find( "&color" );
40
41         int colorLocation = dataString.find( "color=" ) + 6;
42         int endColor = dataString.find( "&button" );
43
44         // get value for user's name
45         nameString = dataString.substr( nameLocation, endName -
46            nameLocation );
47
48         // get value for user's age
49         if ( ageLocation > 0 )
50            ageString = dataString.substr( ageLocation, endAge -
51               ageLocation );
52
53         // get value for user's favorite color
54         if ( colorLocation > 0 )
55            colorString = dataString.substr( colorLocation,
56               endColor - colorLocation );
57
58         // set cookie
59         cout << "Set-Cookie: Name=" << nameString << "age:"
60              << ageString << "color:" << colorString
61              << "; expires=" << expires << "; path=\n";
62
63      } // end if
64
65      // output header
66      cout << "Content-Type: text/html\n\n";
67
68      // output XML declaration and DOCTYPE
69      cout << "<?xml version = \"1.0\"?>"
70           << "<!DOCTYPE html PUBLIC \"-//W3C//DTD XHTML 1.0 "
71           << "Transitional//EN\" \"http://www.w3.org/TR/xhtml1"
72           << "/DTD/xhtml1-transitional.dtd\">";
73
74      // output html element and some of its contents
75      cout << "<html xmlns = \"http://www.w3.org/1999/xhtml\">"
76           << "<head><title>Cookie Saved</title></head>"
77           << "<body>";
78
```

Fig. 16.16 Writing a cookie. (Part 2 of 3.)

```
79      // output user's information
80      cout << "<p>The cookies have been set with the following"
81           << " data:</p>"
82           << "<p>Name: " << nameString  << "<br/></p>"
83           << "<p>Age:"   << ageString   << "<br/></p>"
84           << "<p>Color:" << colorString << "<br/></p>"
85           << "<p>Click <a href=\"/cgi-bin/read_cookie.cgi\">"
86           << "here</a> to read saved cookie data:</p>"
87           << "</body></html>";
88
89      return 0;
90
91   } // end main
```

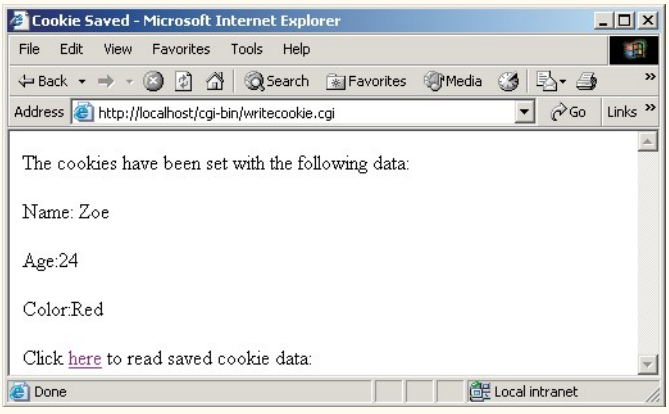


Fig. 16.16 Writing a cookie. (Part 3 of 3.)

After obtaining the data from the form, the program creates a cookie (lines 59–61). In this example, we create a cookie by adding a line of text containing the name-value pairs of the posted data, delimited by a colon character (`:`). The line must be output before the header is written to the client. The line of text begins with the **`Set-Cookie:`** header, indicating that the browser should store the incoming data in a cookie. We set three attributes for the cookie: a name-value pair containing the data to be stored, a name-value pair containing the expiration date and a name-value pair containing the URL of the server domain (e.g., **`www.deitel.com`**) for which the cookie is valid. For this example, **`path`** is not set to any value, making the cookie readable from any server in the domain of the server that originally wrote the cookie. Lines 66–87 send a Web page indicating that the cookie has been written to the client.

Portability Tip 16.1

Web browsers store the cookie information in a vendor-specific manner. For example, Internet Explorer stores cookies as text files in the ***Temporary Internet Files*** *directory on the client's machine. Netscape stores its cookies in a single file named* **`cookies.txt`**.

Figure 16.17 reads the cookie written in Fig. 16.16 and displays the information. When a request is made from the client Web browser, the Web browser locates any cookies previously written by the server to which the request is being made. These cookies are sent by

the browser as part of the request. On the server, the environment variable **HTTP_COOKIE** stores the client's cookies sent as part of the request. Line 20 calls function **getenv** with the **HTTP_COOKIE** environment variable as the first parameter. The value returned is stored in **dataString**. The name-value pairs are decoded and stored in strings on lines 23–36 according to the **name:value** encoding scheme used in Fig. 16.16. The contents of the cookie are output as a Web page on lines 39–58.

Software Engineering Observation 16.2

Cookies present a security risk. If unauthorized users gain access to a computer, they can examine the local disk and view files, which include cookies. For this reason, sensitive data, such as passwords, should never be stored in cookies.

```
1  // Fig. 16.17: readcookie.cpp
2  // Program to read cookie data.
3  #include <iostream>
4
5  using std::cout;
6  using std::cin;
7
8  #include <cstdlib>
9  #include <string>
10
11 using std::string;
12
13 int main()
14 {
15    string dataString = "";
16    string nameString = "";
17    string ageString = "";
18    string colorString = "";
19
20    dataString = getenv( "HTTP_COOKIE" );
21
22    // search through cookie data string
23    int nameLocation = dataString.find( "Name=" ) + 5;
24    int endName = dataString.find( "age:" );
25
26    int ageLocation = dataString.find( "age:" ) + 4;
27    int endAge = dataString.find( "color:" );
28
29    int colorLocation = dataString.find( "color:" ) + 6;
30
31    // store cookie data in strings
32    nameString = dataString.substr( nameLocation, endName -
33       nameLocation );
34    ageString = dataString.substr( ageLocation, endAge -
35       ageLocation);
36    colorString = dataString.substr( colorLocation );
37
38    // output header
39    cout << "Content-Type: text/html\n\n";
40
```

Fig. 16.17 Program to read cookies from the client's computer. (Part 1 of 2.)

```
41      // output XML declaration and DOCTYPE
42      cout << "<?xml version = \"1.0\"?>"
43           << "<!DOCTYPE html PUBLIC \"-//W3C//DTD XHTML 1.0 "
44           << "Transitional//EN\" \"http://www.w3.org/TR/xhtml1"
45           << "/DTD/xhtml1-transitional.dtd\">";
46
47      // output html element and some of its contents
48      cout << "<html xmlns = \"http://www.w3.org/1999/xhtml\">"
49           << "<head><title>Read Cookies</title></head>"
50           << "<body>";
51
52      // data was found
53      if ( dataString != "" )
54         cout << "<h3>The following data is saved in a cookie on"
55              << " your computer</h3>"
56              << "<p>Name: "  << nameString  << "<br/></p>"
57              << "<p>Age: "   << ageString   << "<br/></p>"
58              << "<p>Color: " << colorString << "<br/></p>";
59
60      // no data was found
61      else
62         cout << "<p>No cookie data.</p>";
63
64      cout << "</body></html>";
65
66      return 0;
67
// end main
```

Fig. 16.17 Program to read cookies from the client's computer. (Part 2 of 2.)

16.15 Server-Side Files

The other mechanism by which to maintain state information is to create *server-side files* (i.e., files that are located on the server or on the server's network). This mechanism is a slightly more secure method by which to maintain vital information. In this mechanism, only someone with access and permission to change files on the server can alter files. Figure 16.18 and Fig. 16.19 ask users for contact information then store it on the server. The file that is created by the script is shown in Fig. 16.20.

The XHTML document in Fig. 16.18 posts the form data to the CGI script in Fig. 16.19. In the CGI script, lines 46–106 decode the parameters that were sent by the client. Line 123 creates an instance of the output file stream (**`outFile`**) that opens a file for appending. If the file **`clients.txt`** does not exist, it is created. Lines 132–136 output the personal information to the file. (See Fig. 16.20 for the contents of the file.) The remainder of the program outputs an XHTML document that summarizes the user's information.

There are a few important points to make about this program. First, we do not perform any validation on the data before writing the data to disk. Normally, the script would check for bad data, incomplete data, etc. Second, our file is located in the **`cgi-bin`** directory, which is publicly accessible. If someone knew the filename, it would be relatively easy to access someone else's contact information.

```
1  <?xml version = "1.0"?>
2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
3  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
4
5  <!-- Fig. 16.18: savefile.html           -->
6  <!-- Form to input client information    -->
7
8  <html xmlns = "http://www.w3.org/1999/xhtml">
9     <head>
10       <title>Please enter your contact information</title>
11    </head>
12
13    <body>
14       <p>Please enter your information in the form below.</p>
15       <p>Note: You must fill in all fields.</p>
16       <form method = "post"
17          action = "/cgi-bin/savefile.cgi">
18          <p>
19             First Name:
20             <input type = "text" name = "firstname" size = "10" />
21             Last Name:
22             <input type = "text" name = "lastname" size = "15" />
23          </p>
24
25          <p>
26             Address:
27             <input type = "text" name = "address" size = "25" />
28             <br />
29             Town:
30             <input type = "text" name = "town" size = "10" />
31             State:
32             <input type = "text" name = "state" size = "2" />
33             <br/>
34             Zip Code:
35             <input type = "text" name = "zipcode" size = "5" />
36             Country:
37             <input type = "text" name = "country" size = "10" />
38          </p>
```

Fig. 16.18 XHTML document to read user's contact information. (Part 1 of 2.)

```
            <p>
               E-mail Address:
               <input type = "text" name = "email" />
            </p>
               <input type = "submit" value = "Enter" />
               <input type = "reset" value = "Clear" />
         </form>
      </body>
   </html>
```

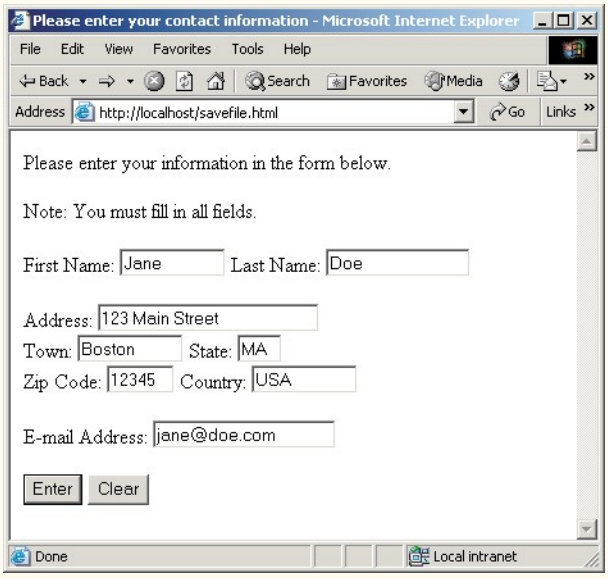


Fig. 16.18 XHTML document to read user's contact information. (Part 2 of 2.)

This script is not robust enough for deployment on the Internet, but it does provide an example of the use of server-side files to store information. Once the files are stored on the server, users cannot change the files unless they are allowed to do so by the server administrator. Thus, storing these files on the server is safer than storing user data in cookies. [*Note:* Many systems store user information in password-protected databases for higher levels of security.]

```
// Fig. 16.19: savefile.cpp
// Program to enter user's contact information into a
// server-side file.

#include <iostream>

using std::cerr;
```

Fig. 16.19 Creating a server-side file to store user data. (Part 1 of 5.)

```
using std::cout;
using std::cin;
using std::ios;

#include <fstream>

using std::ofstream;

#include <string>

using std::string;

#include <cstdlib>

int main()
{
   char postString[ 1024 ] = "";
   int contentLength = 0;

   // variables to store user data
   string dataString = "";
   string firstname = "";
   string lastname = "";
   string address = "";
   string town = "";
   string state = "";
   string zipcode = "";
   string country = "";
   string email = "";

   // data was posted
   if ( getenv( "CONTENT_LENGTH" ) )
      contentLength = atoi( getenv( "CONTENT_LENGTH" ) );

   cin.read( postString, contentLength );
   dataString = postString;

   // search for first '+' character
   int charLocation = dataString.find( "+" );

   // search for next '+' character
   while ( charLocation < string::npos ) {
      dataString.replace( charLocation, 1, " " );
      charLocation = dataString.find( "+", charLocation + 1 );
   } // end while

   // find location of firstname
   int firstStart  = dataString.find( "firstname=" ) + 10;
   int endFirst = dataString.find( "&lastname" );

   firstname = dataString.substr( firstStart,
      endFirst - firstStart );

```

Fig. 16.19 Creating a server-side file to store user data. (Part 2 of 5.)

```
61      // find location of lastname
62      int lastStart = dataString.find( "lastname=" ) + 9;
63      int endLast = dataString.find( "&address" );
64
65      lastname = dataString.substr( lastStart,
66         endLast - lastStart );
67
68      // find location of address
69      int addressStart = dataString.find( "address=" ) + 8;
70      int endAddress = dataString.find( "&town" );
71
72      address = dataString.substr( addressStart,
73         endAddress - addressStart );
74
75      // find location of town
76      int townStart = dataString.find( "town=" ) + 5;
77      int endTown = dataString.find( "&state" );
78
79      town = dataString.substr( townStart, endTown - townStart);
80
81      // find location of state
82      int stateStart = dataString.find( "state=" ) + 6;
83      int endState = dataString.find( "&zipcode" );
84
85      state = dataString.substr( stateStart,
86         endState - stateStart );
87
88      // find location of zip code
89      int zipStart = dataString.find( "zipcode=" ) + 8;
90      int endZip = dataString.find( "&country" );
91
92      zipcode = dataString.substr( zipStart, endZip - zipStart );
93
94      // find location of country
95      int countryStart = dataString.find( "country=" ) + 8;
96      int endCountry = dataString.find( "&email" );
97
98      country = dataString.substr( countryStart,
99         endCountry - countryStart );
100
101     // find location of e-mail address
102     int emailStart = dataString.find( "email=" ) + 6;
103     int endEmail = dataString.find( "&submit" );
104
105     email = dataString.substr( emailStart,
106        endEmail - emailStart );
107
108     // output header
109     cout << "Content-Type: text/html\n\n";
110
```

Fig. 16.19 Creating a server-side file to store user data. (Part 3 of 5.)

```
111    // output XML declaration and DOCTYPE
112    cout << "<?xml version = \"1.0\"?>"
113         << "<!DOCTYPE html PUBLIC \"-//W3C//DTD XHTML 1.0 "
114         << "Transitional//EN\" \"http://www.w3.org/TR/xhtml1"
115         << "/DTD/xhtml1-transitional.dtd\">";
116
117    // output html element and some of its contents
118    cout << "<html xmlns = \"http://www.w3.org/1999/xhtml\">"
119         << "<head><title>Contact Information entered"
120         << "</title></head><body>";
121
122    // output to file
123    ofstream outFile( "clients.txt", ios::app );
124
125    // file was not opened properly
126    if ( !outFile ) {
127       cerr << "Error: could not open contact file.";
128       exit( 1 );
129    } // end if
130
131    // append data to clients.txt file
132    outFile << firstname << " " << lastname << "\n"
133            << address << "\n" << town << " "
134            << state << " " << country << " "
135            << zipcode << "\n" << email
136            << "\n\n";
137
138    // output data to user
139    cout << "<table><tbody>"
140         << "<tr><td>First Name:</td><td>"
141         << firstname << "</td></tr>"
142         << "<tr><td>Last Name:</td><td>"
143         << lastname << "</td></tr>"
144         << "<tr><td>Address:</td><td>"
145         << address << "</td></tr>"
146         << "<tr><td>Town:</td><td>"
147         << town << "</td></tr>"
148         << "<tr><td>State:</td><td>"
149         << state << "</td></tr>"
150         << "<tr><td>Zip Code:</td><td>"
151         << zipcode << "</td></tr>"
152         << "<tr><td>Country:</td><td>"
153         << country << "</td></tr>"
154         << "<tr><td>Email:</td><td>"
155         << email << "</td></tr>"
156         << "</tbody></table>"
157         << "</body>\n</html>\n";
158
159    return 0;
160
161 } // end main
```

Fig. 16.19 Creating a server-side file to store user data. (Part 4 of 5.)

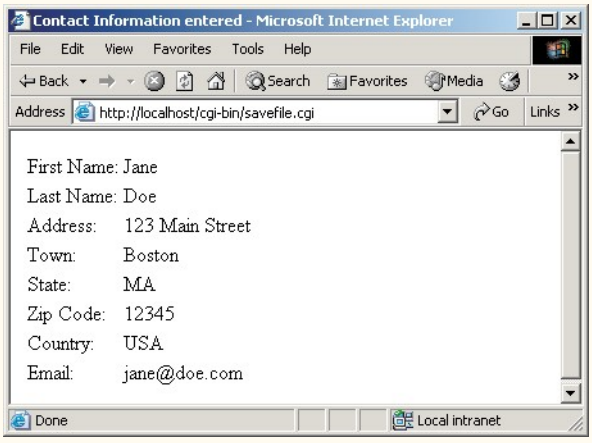


Fig. 16.19 Creating a server-side file to store user data. (Part 5 of 5.)

```
Jane Doe
123 Main Street
Boston MA USA 12345
jane@doe.com
```

Fig. 16.20 Contents of **`clients.txt`** data file.

16.16 Case Study: Shopping Cart

Many businesses' Web sites contain shopping-cart applications, which allow customers to buy items conveniently on the Web. The sites record what the consumer wants to purchase and provide an easy, intuitive way to shop online. They do so by using an electronic shopping cart, just as people would use physical shopping carts in retail stores. As users add items to their shopping carts, the sites update the carts' contents. When users "check out," they pay for the items in their shopping carts. To see a real-world electronic shopping cart, we suggest going to the online bookstore **`Amazon.com`** (**`www.amazon.com`**).

The shopping cart implemented in this section (Fig. 16.21–Fig. 16.24) allows users to purchase books from a fictitious bookstore that sells four books (see Fig. 16.23). This example uses four scripts, two server-side files and cookies.

Figure 16.21 shows the first of these scripts, the login page. This script is the most complex of all the scripts in this section. The first **`if`** condition (line 39) determines whether data was posted to the program. The second **`if`** condition (line 70) determines whether the **`dataString`** was set (i.e., the decoding completed successfully). The first time we run this program, both conditions fail, so lines 75–86 output an XHTML form to the user, as shown in the first screen capture of Fig. 16.21. When the user fills out the form and clicks the **login** button, **`login.cgi`** is requested again.

```
1  // Fig. 16.21: login.cpp
2  // Program to output an XHTML form, verify the
3  // username and password entered, and add members.
4  #include <iostream>
5
6  using std::cerr;
7  using std::cout;
8  using std::cin;
9  using std::ios;
10
11 #include <fstream>
12
13 using std::ifstream;
14 using std::ofstream;
15
16 #include <string>
17
18 using std::string;
19
20 #include <cstdlib>
21
22 void header();
23 void writeCookie();
24
25 int main()
26 {
27    char query[ 1024 ] = "";
28    string dataString = "";
29
30    // strings to store username and password
31    string userName = "";
32    string passWord = "";
33    string newCheck = "";
34
35    int contentLength = 0;
36    int endPassword = 0;
37
38    // data was posted
39    if ( getenv( "CONTENT_LENGTH" ) ) {
40
41       // retrieve query string
42       contentLength = atoi( getenv( "CONTENT_LENGTH" ) );
43       cin.read( query, contentLength );
44       dataString = query;
45
46       // find username location
47       int userLocation = dataString.find( "user=" ) + 5;
48       int endUser = dataString.find( "&" );
49
50       // find password location
51       int passwordLocation = dataString.find( "password=" ) + 9;
52
53       endPassword = dataString.find( "&new" );
```

Fig. 16.21 Program that outputs a login page. (Part 1 of 7.)

```
54
55          // new membership requested
56          if ( endPassword > 0 )
57             passWord = dataString.substr( passwordLocation,
58                endPassword - passwordLocation );
59
60          // existing member
61          else
62             passWord = dataString.substr( passwordLocation );
63
64          userName = dataString.substr( userLocation, endUser -
65             userLocation );
66
67       } // end if
68
69       // no data was retrieved
70       if ( dataString == "" ) {
71          header();
72          cout << "<p>Please login.</p>";
73
74          // output login form
75          cout << "<form method = \"post\" "
76               << "action = \"/cgi-bin/login.cgi\"><p>"
77               << "User Name: "
78               << "<input type = \"text\" name = \"user\"/><br/>"
79               << "Password: "
80               << "<input type = \"password\" "
81               << "name = \"password\"/><br/>"
82               << "New? <input type = \"checkbox\""
83               << " name = \"new\" "
84               << "value = \"1\"/></p>"
85               << "<input type = \"submit\" value = \"login\"/>"
86               << "</form>";
87
88       } // end if
89
90       // process entered data
91       else {
92
93          // add new member
94          if ( endPassword > 0 ) {
95             string fileUsername = "";
96             string filePassword = "";
97             bool nameTaken = false;
98
99             // open password file
100            ifstream userData( "userdata.txt", ios::in );
101
102            // could not open file
103            if ( !userData ) {
104               cerr << "Could not open database.";
105               exit( 1 );
106            } // end if
```

Fig. 16.21 Program that outputs a login page. (Part 2 of 7.)

```
107
108          // read username and password from file
109          while ( userData >> fileUsername >> filePassword ) {
110
111             // name is already taken
112             if ( userName == fileUsername )
113                nameTaken = true;
114
115          } // end while
116
117          // user name is taken
118          if ( nameTaken ) {
119             header();
120             cout << "<p>This name has already been taken.</p>"
121                  << "<a href=\"/cgi-bin/login.cgi\">"
122                  << "Try Again</a>";
123          } // end if
124
125          // process data
126          else {
127
128             // write cookie
129             writeCookie();
130             header();
131
132             // open user data file
133             ofstream userData( "userdata.txt", ios::app );
134
135             // could not open file
136             if ( !userData ) {
137                cerr << "Could not open database.";
138                exit( 1 );
139             } // end if
140
141             // write user data to file
142             userData << "\n" << userName << "\n" << passWord;
143
144             cout << "<p>Your information has been processed."
145                  << "<a href=\"/cgi-bin/shop.cgi\">"
146                  << "Start Shopping</a></p>";
147
148          } // end else
149       } // end if
150
151       // search for password if entered
152       else {
153
154          // strings to store username and password from file
155          string fileUsername = "";
156          string filePassword = "";
157          bool authenticated = false;
158          bool userFound = false;
159
```

Fig. 16.21 Program that outputs a login page. (Part 3 of 7.)

```
160          // open password file
161          ifstream userData( "userdata.txt", ios::in );
162
163          // could not open file
164          if ( !userData ) {
165             cerr << "Could not open database.";
166             exit( 1 );
167          } // end if
168
169          // read in user data
170          while ( userData >> fileUsername >> filePassword ) {
171
172             // username and password match
173             if ( userName == fileUsername &&
174                passWord == filePassword )
175                authenticated = true;
176
177             // username was found
178             if ( userName == fileUsername )
179                userFound = true;
180          } // end while
181
182          // user is authenticated
183          if ( authenticated ) {
184             writeCookie();
185             header();
186
187             cout << "<p>Thank you for returning, "
188                  << userName << "!</p>"
189                  << "<a href=\"/cgi-bin/shop.cgi\">"
190                  << "Start Shopping</a>";
191          } // end if
192
193          // user not authenticated
194          else {
195             header();
196
197             // password is incorrect
198             if ( userFound )
199                cout << "<p>You have entered an incorrect "
200                     << "password. Please try again.</p>"
201                     << "<a href=\"/cgi-bin/login.cgi\">"
202                     << "Back to login</a>";
203
204             // user is not registered
205             else
206                cout << "<p>You are not a registered user.</p>"
207                     << "<a href=\"/cgi-bin/login.cgi\">"
208                     << "Register</a>";
209
210          } // end else
211       } // end else
212    } // end if
```

Fig. 16.21 Program that outputs a login page. (Part 4 of 7.)

```
213
214    cout << "</body>\n</html>\n";
215    return 0;
216
217 } // end main
218
219 // function to output header
220 void header()
221 {
222    // output header
223    cout << "Content-Type: text/html\n\n";
224
225    // output XML declaration and DOCTYPE
226    cout << "<?xml version = \"1.0\"?>"
227         << "<!DOCTYPE html PUBLIC \"-//W3C//DTD XHTML 1.0 "
228         << "Transitional//EN\" \"http://www.w3.org/TR/xhtml1"
229         << "/DTD/xhtml1-transitional.dtd\">";
230
231    // output html element and some of its contents
232    cout << "<html xmlns = \"http://www.w3.org/1999/xhtml\">"
233         << "<head><title>Login Page</title></head>"
234         << "<body>";
235
236 } // end header
237
238 // function to write cookie data
239 void writeCookie()
240 {
241    string expires = "Friday, 14-MAY-04 16:00:00 GMT";
242    cout << "Set-Cookie: CART=; expires="
243         << expires << "; path=\n";
244
245 } // end writeCookie
```

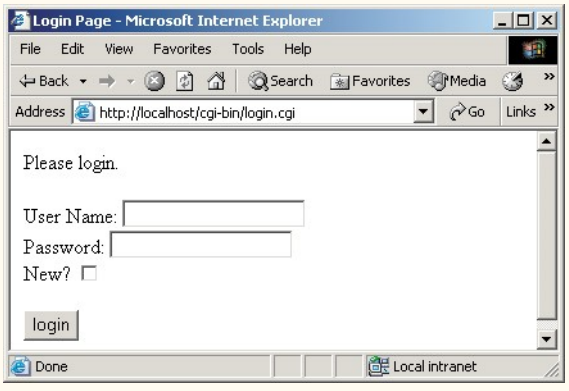
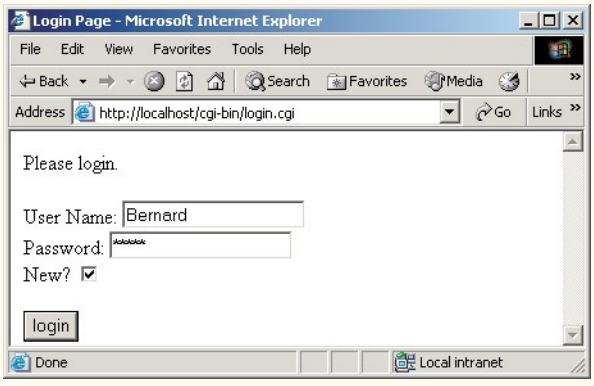


Fig. 16.21 Program that outputs a login page. (Part 5 of 7.)



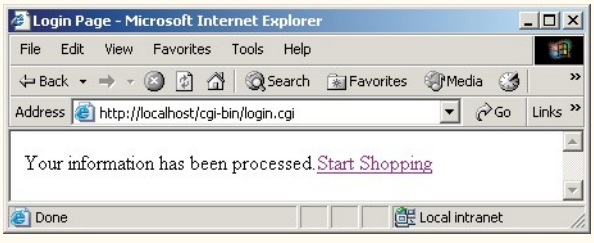


Fig. 16.21 Program that outputs a login page. (Part 6 of 7.)

Fig. 16.21 Program that outputs a login page. (Part 7 of 7.)

If the user checked the **New** checkbox on the Web page to create a new membership, the condition on line 94 evaluates to **`true`**. Next, we open **`userdata.txt`** (line 100)—the file that contains all the usernames and passwords. Lines 109–115 read through this file, comparing each username with the name entered. If the name is already in the list, lines 120–122 output a message to the user indicating that the name has been taken, and a link to the form is provided. Otherwise, the new user is added to the list. The file is opened again on line 133—this time for appending. Line 142 adds the new user information to **`userdata.txt`** in the format

```
Bernard
Jones
```

Each username and password is separated by a newline character. Lines 144–146 provide a hyperlink to the script of Fig. 16.22, which allows users to purchase items.

The last possible scenario for this script is for returning users (lines 152–211). This portion of the program executes when the user enters a name and password but does not select the **New** checkbox (i.e., the **`else`** of line 152 is evaluated). In this case, we assume that the user already has a username and password in **`userdata.txt`**. Lines 170–180 read through **`userdata.txt`** in an attempt to locate the username entered. If the username is found and the password entered is correct (lines 173–174), boolean variable **`authenticated`** is set to **`true`**. Line 183 determines whether the user has been authenticated. Function **`writeCookie`** is called to initialize the cookie and to remove existing data from prior sessions (line 184). The cookie, which is named **`CART`** (line 242), is used by other scripts to store book information. A message is output welcoming the user back to the Web site and providing a link to purchase books (**`shop.cgi`**) on lines 187–190.

If the user was not authenticated, the program determines whether the user was found (line 198). If the user was found but not authenticated, a message is output indicating that the password is invalid. A hyperlink is provided to the login page (**`<a href="/cgi-bin/login.cgi">`**), where the user can attempt to login again. If neither the username nor the password were found, an unregistered user has attempted to sign on (line 205). A message is output indicating that the user does not have the proper authorization to access the page, and lines 206–208 provide a link that allows the user to attempt another login.

Figure 16.22 uses the values in **`catalog.txt`** (Fig. 16.25) to output the items that the user can purchase. The **`while`** structure (lines 73–93) outputs a table containing the items. The last column for each row includes a button for adding the item to the shopping

cart. Hidden form fields are specified for each book and its associated information. Lines 73–77 output the different values for each book, and lines 83–93 output a form containing **submit** buttons for purchasing books.

When a user purchases a book, the **viewcart.cgi** script is requested, and the ISBN for the book to be purchased is sent to the script. Figure 16.23 begins by reading the value of the cookie stored on the user's system on line 38. Any existing cookie data is stored in **string cookieString** (line 39). The entered ISBN number from the form of Fig. 16.22 is stored in **string isbnEntered** (line 54). The script determines whether the cart already contains data (line 65). If not, the **cookieString** is given the value of the entered ISBN number (line 66). If the cookie already contains data, the entered ISBN is appended to the existing cookie data (line 70). The new book is stored in the **CART** cookie on lines 73–74. The cart's contents are output in a table by calling function **outputBooks** (line 95).

```
1  // Fig. 16.22: shop.cpp
2  // Program to display available books.
3  #include <iostream>
4
5  using std::cerr;
6  using std::cout;
7  using std::cin;
8  using std::ios;
9
10 #include <istream>
11
12 #include <fstream>
13
14 using std::ifstream;
15 using std::ofstream;
16
17 #include <string>
18
19 using std::string;
20
21 #include <cstdlib>
22
23 void header();
24
25 int main()
26 {
27    // variables to store product information
28    char book[ 50 ] = "";
29    char year[ 50 ] = "";
30    char isbn[ 50 ] = "";
31    char price[ 50 ] = "";
32
33    string bookString = "";
34    string yearString = "";
35    string isbnString = "";
36    string priceString = "";
```

Fig. 16.22 CGI script that allows users to buy a book. (Part 1 of 3.)

```
37
38      bool nameTaken = false;
39
40      // open file for input
41      ifstream userData( "catalog.txt", ios::in );
42
43      // file could not be opened
44      if ( !userData ) {
45         cerr << "Could not open database.";
46         exit( 1 );
47      } // end if
48
49      header(); // output header
50
51      // output available books
52      cout << "<center><br/>Books available for sale<br/>"
53           << "<a href=\"/cgi-bin/logout.cgi\">Sign Out"
54           << "</a><br/><br/>"
55           << "<table border = \"1\" cellpadding = \"7\" >";
56
57      // file is open
58      while ( userData ) {
59
60         // retrieve data from file
61         userData.getline( book, 50 );
62         bookString = book;
63
64         userData.getline( year, 50 );
65         yearString = year;
66
67         userData.getline( isbn, 50 );
68         isbnString = isbn;
69
70         userData.getline( price, 50 );
71         priceString = price;
72
73         cout << "<tr>"
74              << "<td>" << bookString << "</td>"
75              << "<td>" << yearString << "</td>"
76              << "<td>" << isbnString << "</td>"
77              << "<td>" << priceString << "</td>";
78
79         // file is still open after reads
80         if ( userData )
81
82            // output form with buy button
83            cout << "<td><form method=\"post\" "
84                 << "action=\"/cgi-bin/viewcart.cgi\">"
85                 << "<input type=\"hidden\" name=\"add\""
86                 << "value=\"true\"/>"
87                 << "<input type=\"hidden\" name=\"isbn\""
88                 << "value=\"" << isbnString << "\"/>"
89                 << "<input type=\"submit\""
```

Fig. 16.22 CGI script that allows users to buy a book. (Part 2 of 3.)

```
90                << "value=\"Buy\"/>\n"
91                << "</form></td>\n";
92
93          cout << "</tr>\n";
94
95       } // end while
96
97       cout << "</table></center></body></html>";
98       return 0;
99    }
100
101   // function to output header information
102   void header()
103   {
104      // output header
105      cout << "Content-Type: text/html\n\n";
106
107      // output XML declaration and DOCTYPE
108      cout << "<?xml version = \"1.0\"?>"
109           << "<!DOCTYPE html PUBLIC \"-//W3C//DTD XHTML 1.0 "
110           << "Transitional//EN\" \"http://www.w3.org/TR/xhtml1"
111           << "/DTD/xhtml1-transitional.dtd\">";
112
113      // output html element and some of its contents
114      cout << "<html xmlns = \"http://www.w3.org/1999/xhtml\">"
115           << "<head><title>Login Page</title></head>"
116           << "<body>";
117   } // end header
```

Fig. 16.22 CGI script that allows users to buy a book. (Part 3 of 3.)

Figure 16.24 allows the user to log out of the shopping-cart application. This script outputs a message to the user and calls **`writeCookie`** (line 20), thus erasing the current information in the shopping cart.

```
1   // Fig. 16.23: viewcart.cpp
2   // Program to view books in the shopping cart.
3   #include <iostream>
4
5   using std::cerr;
6   using std::cout;
7   using std::cin;
8   using std::ios;
9
10  #include <istream>
11
12  #include <fstream>
13
14  using std::ifstream;
15  using std::ofstream;
16
```

Fig. 16.23 CGI script that allows users to view their carts' content. (Part 1 of 5.)

```
#include <string>

using std::string;

#include <cstdlib>

void outputBooks( const string &, const string & );

int main()
{
   // variable to store query string
   char query[ 1024 ] = "";
   char *cartData; // variable to hold contents of cart

   string dataString = "";
   string cookieString = "";
   string isbnEntered = "";
   int contentLength = 0;

   // retrieve cookie data
   if ( getenv( "HTTP_COOKIE" ) ) {
      cartData = getenv( "HTTP_COOKIE" );
      cookieString = cartData;
   } // end if

   // data was entered
   if ( getenv( "CONTENT_LENGTH" ) ) {
      contentLength = atoi( getenv( "CONTENT_LENGTH" ) );
      cin.read( query, contentLength );
      dataString = query;

      // find location of isbn value
      int addLocation = dataString.find( "add=" ) + 4;
      int endAdd = dataString.find( "&isbn" );
      int isbnLocation = dataString.find( "isbn=" ) + 5;

      // retrieve isbn number to add to cart
      isbnEntered = dataString.substr( isbnLocation );

      // write cookie
      string expires = "Friday, 14-MAY-10 16:00:00 GMT";
      int cartLocation = cookieString.find( "CART=" ) + 5;

      // cookie exists
      if ( cartLocation > 0 )
         cookieString = cookieString.substr( cartLocation );

      // no cookie data exists
      if ( cookieString == "" )
         cookieString = isbnEntered;

```

Fig. 16.23 CGI script that allows users to view their carts' content. (Part 2 of 5.)

```
68          // cookie data exists
69          else
70             cookieString += "," + isbnEntered;
71
72          // set cookie
73          cout << "Set-Cookie: CART=" << cookieString << "; expires="
74               << expires << "; path=\n";
75
76       } // end if
77
78       // output header
79       cout << "Content-Type: text/html\n\n";
80
81       // output XML declaration and DOCTYPE
82       cout << "<?xml version = \"1.0\"?>"
83            << "<!DOCTYPE html PUBLIC \"-//W3C//DTD XHTML 1.0 "
84            << "Transitional//EN\" \"http://www.w3.org/TR/xhtml1"
85            << "/DTD/xhtml1-transitional.dtd\">";
86
87       // output html element and some of its contents
88       cout << "<html xmlns = \"http://www.w3.org/1999/xhtml\">"
89            << "<head><title>Shopping Cart</title></head>"
90            << "<body><center>"
91            << "<p>Here is your current order:</p>";
92
93       // cookie data exists
94       if ( cookieString != "" )
95          outputBooks( cookieString, isbnEntered );
96
97       cout << "</body></html>\n";
98       return 0;
99
100 } // end main
101
102 // function to output books in catalog.txt
103 void outputBooks( const string &cookieRef, const string &isbnRef )
104 {
105    char book[ 50 ] = "";
106    char year[ 50 ] = "";
107    char isbn[ 50 ] = "";
108    char price[ 50 ] = "";
109
110    string bookString = "";
111    string yearString = "";
112    string isbnString = "";
113    string priceString = "";
114
115    // open file for input
116    ifstream userData( "catalog.txt", ios::in );
117
```

Fig. 16.23 CGI script that allows users to view their carts' content. (Part 3 of 5.)

```
118    // file could not be opened
119    if ( !userData ) {
120       cerr << "Could not open database.";
121       exit( 1 );
122    } // end if
123
124    // output link to log out and table to display books
125    cout << "<a href=\"/cgi-bin/logout.cgi\">Sign Out";
126    cout << "</a><br><br>";
127    cout << "<table border = 1 cellpadding = 7 >";
128
129    // file is open
130    while ( userData ) {
131
132       // retrieve book information
133       userData.getline( book, 50 );
134       bookString = book;
135
136       // retrieve year information
137       userData.getline( year, 50 );
138       yearString = year;
139
140       // retrieve isbn number
141       userData.getline( isbn, 50 );
142       isbnString = isbn;
143
144       // retrieve price
145       userData.getline( price, 50 );
146       priceString = price;
147
148       int match = cookieRef.find( isbn );
149
150       // match has been made
151       if ( match > 0 || isbnRef == isbnString ) {
152
153          // output table row with book information
154          cout << "<tr>"
155               << "<form method=\"post\""
156               << "action=\"/cgi-bin/viewcart.cgi\">"
157               << "<td>" << bookString << "</td>"
158               << "<td>" << yearString << "</td>"
159               << "<td>" << isbnString << "</td>"
160               << "<td>" << priceString << "</td>";
161
162       } // end if
163
164       cout << "</form></tr>";
165
166    } // end while
167
168    // output link to add more books
169    cout << "<a href=\"/cgi-bin/shop.cgi\">Back to book list</a>";
170 } // end outputBooks
```

Fig. 16.23 CGI script that allows users to view their carts' content. (Part 4 of 5.)

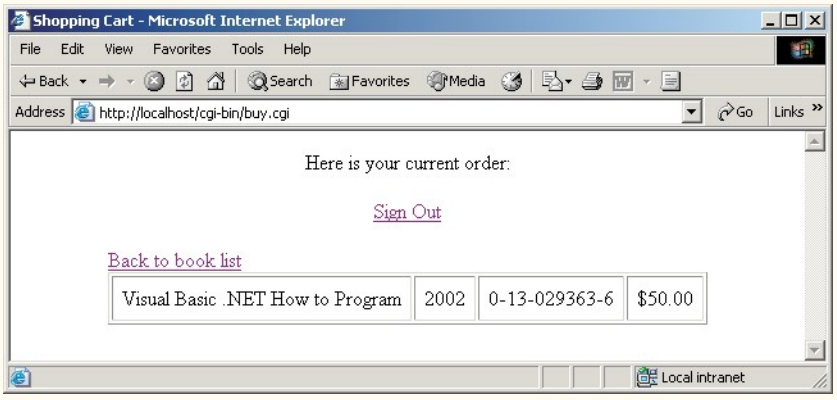


Fig. 16.23 CGI script that allows users to view their carts' content. (Part 5 of 5.)

Figure 16.25 shows the contents of the **`catalog.txt`** file. This file must reside in the same directory where the CGI scripts reside for this shopping-cart application to work correctly.

```
1  // Fig. 16.24: logout.cpp
2  // Program to log out of the system.
3  #include <iostream>
4
5  using std::cout;
6
7  #include <string>
8
9  using std::string;
10
11 #include <ctime>
12
13 #include <cstdlib>
14
15 void writeCookie();
16
17 int main()
18 {
19    // write the cookie
20    writeCookie();
21
22    // output header
23    cout << "Content-Type: text/html\n\n";
24
25    // output XML declaration and DOCTYPE
26    cout << "<?xml version = \"1.0\"?>"
27         << "<!DOCTYPE html PUBLIC \"-//W3C//DTD XHTML 1.0 "
28         << "Transitional//EN\"";
29
```

Fig. 16.24 Logout program. (Part 1 of 2.)

```
30      // output html element and its contents
31      cout << "<html xmlns = \"http://www.w3.org/1999/xhtml\">"
32           << "<head><title>Logged Out</title></head>"
33           << "<body>"
34           << "<center><p>You are now logged out<br>"
35           << "You will be billed accordingly<br>"
36           << "To login again, "
37           << "<a href=\"/cgi-bin/login.cgi\">click here</a>"
38           << "</body></html>\n";
39
40      return 0;
41
42   } // end main
43
44   // function to write cookie
45   void writeCookie()
46   {
47      // string containing expiration date
48      string expires = "Friday, 14-MAY-10 16:00:00 GMT";
49
50      // set cookie
51      cout << "Set-Cookie: CART=; expires=" << expires
52           << "; path=\n";
53
54   } // end writeCookie
```

Fig. 16.24 Logout program. (Part 2 of 2.)

16.17 Internet and Web Resources

Apache

www.apache.org
This is the product home page for the Apache HTTP server. Users may download Apache from this site.

www.apacheweek.com
This online magazine contains articles about Apache jobs, product reviews and other information concerning Apache software.

linuxtoday.com/stories/18780.html
This site contains an article about the Apache HTTP server and the platforms that support it. It also contains links to other Apache articles.

```
Visual Basic .NET How to Program
2002
0-13-029363-6
$50.00
C# How to Program
2002
0-13-062221-4
$49.95
C How to Program 3e
2001
0-13-089572-5
$50.00
Java How to Program 4e
2002
0-13-034151-7
$49.95
```

Fig. 16.25 Contents of **catalog.txt**.

CGI

www.gnu.org/software/cgicc/cgicc.html
This site contains a free open-source CGI library for creating CGI scripts in C++.

www.hotscripts.com
This site contains a rich collection of scripts for performing image manipulation, server administration, networking, etc. using CGI.

www.jmarshall.com/easy/cgi
This page contains a brief explanation of CGI for those with programming experience.

www.speakeasy.org/~cgires
This site contains a collection of CGI-related tutorials and scripts.

www.w3.org/CGI
This World Wide Web Consortium page discusses CGI security issues.

www.w3.org/Protocols
This World Wide Web Consortium site contains information on the HTTP specification and links to news, mailing lists and published articles.

SUMMARY

- Web servers respond to client requests by providing resources, such as XHTML documents.
- Web servers and clients communicate with each other via the platform-independent Hypertext Transfer Protocol (HTTP).
- The most common HTTP request types are *get* and *post*; these requests send client form data to a Web server.
- The *get* request sends form content as part of the URL; the *post* request attaches form contents to the end of an HTTP request. The data sent in a *post* request are not part of the URL and cannot be seen by the user.
- Browsers often cache Web pages for quick reloading. However, browsers typically do not cache the server's response to a *post* request, because the information might have changed.
- The information tier maintains data for the application in a database.

- A Web server is part of a multi-tier application—sometimes referred to as an *n*-tier application. A multi-tier application divides functionality into separate tiers. The three-tier application contains an information tier, a middle tier and a client tier.
- The middle tier implements business logic and presentation logic to control interactions between application clients and application data. A Web server is a middle-tier application.
- The client tier is the application's user interface. The client interacts with the middle tier to make requests and to retrieve data from the information tier. The client then displays data retrieved from the middle tier to the user.
- The Apache HTTP server, developed by the Apache Group, is the most popular Web server in use today. It runs on Windows and non-Windows platforms.
- A virtual directory is an alias for an existing directory on a local machine.
- The Common Gateway Interface (CGI) describes a set of protocols through which applications (commonly called CGI scripts or CGI programs) can interact with Web servers and interact (indirectly) with clients.
- CGI is "common" in the sense that it is not specific to any particular operating system (such as Linux or Windows) or to any one programming language.
- A Web page, in its simplest form, is nothing more than an XHTML document. This document is just a plain text file containing markings (markup or elements) that describe to a Web browser how to display and format the information in the document.
- Hypertext information creates links to different pages or to other portions of the same page.
- Any XHTML file available for viewing over the Internet has a URL (Universal Resource Locator) associated with it. The URL contains information that directs a browser to the resource that the user wishes to access.
- The hostname is the name of the computer where the resource resides and is translated into an IP address, which identifies the server on the Internet.
- To request a resource, the browser first sends an HTTP request message to the server. The server responds with a line indicating the HTTP version, followed by a numeric code and a phrase describing the status of the transaction. The server normally then sends one or more HTTP headers, which provide additional information about the data being sent. The header or set of headers is followed by a blank line, which indicates that the server finished sending HTTP headers. Then the server sends the contents of the requested resource, and the connection is terminated. The client-side browser interprets the XHTML it receives and displays the results.
- A properly configured Web server will recognize a CGI script and execute it. A resource usually is designated as a CGI script in one of two ways: Either it has a specific filename extension (such as **`.cgi`** or **`.exe`**), or it is located in a special directory (often **`/cgi-bin`**). The server administrator must give permission for remote clients to access and execute CGI scripts.
- When the server recognizes that the resource requested is a CGI script, the server executes the script. The output is piped to the Web server. Finally, the Web server adds an additional line to the output indicating the status of the HTTP transaction (such as **`HTTP/1.1 200 OK`**, for success) and sends the whole body of text to the client. The browser on the client side then interprets the output and displays the results appropriately.
- With a CGI script, programmers must include the **`Content-Type`** header explicitly, whereas with a normal XHTML document, the header would be added by the Web server.
- The CGI protocol for output to be sent to a Web browser consists of printing to standard output the **`Content-Type`** header, a blank line and the data (XHTML, plain text, etc.) to be output.
- CGI-enabled Web servers set environment variables that provide information about both the server's and the client's script-execution environment.

- The environment variable **QUERY_STRING** provides a mechanism that enables programmers to supply any sort of data to their CGI scripts. The **QUERY_STRING** variable contains information that is appended to a URL. A question mark character (**?**) delimits the resource requested from the query string.
- Data placed in a query string can be structured in a variety of ways, provided that the CGI script that reads the string knows how to interpret the encoded data.
- Forms provide another way for users to input information that is sent to a CGI script.
- The **<form>** element generally takes two attributes. The first attribute is **action**, which specifies the action to take when the user submits the form. The second attribute is **method**, which is either **GET** or **POST**.
- Using *get* with a form causes data to be passed to the CGI script through environment variable **QUERY_STRING**.
- The **POST** method enables CGI scripts to interact with servers via standard input.
- With **POST**, data is encoded just as with **QUERY_STRING**, but the **QUERY_STRING** environment variable is not set. Instead, the **POST** method sets the environment variable **CONTENT_LENGTH** to indicate the number of characters of data that are being sent or posted, then function **read** is used with **STDIN** to obtain the data.
- Web browsers encode the form data before it is sent. This means that spaces are replaced with plus signs, and certain other symbols (such as the apostrophe) are converted into their ASCII value equivalent and displayed in hexadecimal notation (preceded by a percent sign).
- A CGI script can supply HTTP headers in addition to **Content-Type**. In most cases, the server passes these extra headers to the client untouched.
- The CGI protocol indicates that certain types of headers output by a CGI script are to be handled by the server, rather than be passed directly to the client.
- Function **getenv** from library **<cstdlib>** returns a character array containing the value of the CGI environment variable passed to it.

TERMINOLOGY

action attribute of element form
Apache HTTP Server
asctime
<body> element
bottom tier
button type attribute for **input** element
cache
/cgi-bin directory
CGI (Common Gateway Interface)
.cgi file extension
CGI program
CGI script
CGI specification
checkbox type attribute for **input** element
client tier
CONTENT_LENGTH environment variable
Content-Type header
.cpp file extension
data tier
DNS lookup
domain name
domain name system (DNS)
dynamic vs. static Web content
dynamic Web content
Extensible HyperText Markup Language (XHTML)
file type attribute for **input** elements
filepath
form
form element
fully qualified host name
get (HTTP request)
getenv function of **<cstdlib>**
head element
hidden type attribute for **input** elements
host
hostname
htdocs directory

`html` element
HTTP (Hypertext Transfer Protocol)
HTTP connection
HTTP header
HTTP host
HTTP method
HTTP transaction
`HTTP_USER_AGENT` environment variable
HyperText Markup Language (HTML)
HyperText Transfer Protocol (HTTP)
`image` type attribute for **`input`** element
information tier
`input` element
IP address
local Web server
`localhost`
`localtime`
markup
`method` attribute of **`form`** element
middle tier
multi-tier application
n-tier application
open source
`password` type attribute for **`input`** element
pipe
post (HTTP request)
`QUERY_STRING` environment variable
`radio` type attribute for **`input`** element
redirect
remote Web server
request method
request type
`reset` type attribute for **`input`** element
`select` element
standard output
static Web content
`submit` type attribute for **`input`** element
`text` type attribute for **`input`** element
`textarea` element
`title` element
top tier
top-level domain (TLD)
URL (Universal Resource Locator)
virtual directory
Web server
XHTML
XHTML element
XHTML form
XHTML **`form`** element

SELF-REVIEW EXERCISES

16.1 Fill in the blanks in each of the following statements:

a) The two most common HTTP request types are _____ and _____.
b) Browsers often _____ Web pages for quick reloading.
c) In a three-tier application, a Web server is typically part of the _____ tier.
d) In the URL **`http://www.deitel.com/books/downloads.htm`**, the part that consists of **`www.deitel.com`** is the _____ of the server, where a client can find the desired resource.
e) A(n) _____ document is a text file containing markings that describe to a Web browser how to display and format the information in the document.
f) The environment variable _____ provides a mechanism for supplying data to CGI scripts.
g) A common way of reading input from the user is to implement _____.

16.2 State whether each of the following is *true* or *false*. If *false*, explain why.

a) Web servers and clients communicate with each other through the platform-independent HTTP.
b) Web servers often cache Web pages for reloading.
c) The information tier implements business logic to control the type of information that is presented to a particular client.
d) A dynamic Web page is a Web page that is not created programmatically.
e) We put data into a query string using a format that consists of a series of name-value pairs joined with exclamation points (**`!`**).

f) Using a CGI script is more efficient than using an XHTML document.
g) The *post* method of submitting form data is preferable when sending personal information to the Web server.

ANSWERS TO SELF-REVIEW EXERCISES

16.1 a) *get* and *post*. b) cache. c) middle. d) hostname. e) XHTML. f) **`QUERY_STRING`**. g) forms.

16.2 a) True. b) True. Web browsers often cache Web pages for quick reloading c) False. The middle tier implements business logic and presentation logic to control interactions between application clients and application data. d) False. A dynamic Web page is a Web page that is created programmatically. e) False. The pairs are joined with an ampersand (**`&`**). f) False. XHTML documents are more efficient than CGI scripts because XHTML documents do not need to be executed on the server side before they are output to the client. g) True.

EXERCISES

16.3 Define the following terms:
a) HTTP.
b) Multi-tier application.
c) Request method.

16.4 Explain the difference between the *get* request type and the *post* request type. When is it ideal to use the *post* request type?

16.5 Write a CGI script that prints the squares of the integers from 1 to 10 on separate lines.

16.6 Write a CGI script that receives as input three numbers from the client and returns a statement indicating whether the three numbers could represent an equilateral triangle (all three sides are the same length), an isosceles triangle (two sides are the same length) or a right triangle (the square of one side is equal to the sum of the squares of the other two sides.)

16.7 Write a soothsayer script that allows the user to submit a question. When the question is submitted, the script should choose a random response from a list of vague answers and return a new page displaying the answer.

16.8 Modify the program of Fig. 16.14 to incorporate the opening XHTML form and the processing of the data into a single CGI script (i.e., combine the XHTML of Fig. 16.13 into the CGI script of Fig. 16.14.) When the CGI script is requested initially, the form should be displayed. When the form is submitted, the CGI script should execute.

16.9 Modify the shopping-cart application to enable users to remove items from the cart.