CHAPTER

9

Using Perl for CGI Programming

This chapter introduces the Common Gateway Interface (CGI) and the use of Perl for CGI programming. It begins with an overview of CGI, explaining how CGI programs are linked to Web documents and how results are returned to clients from CGI programs. Next, the format of query strings is described. Then it discusses the `CGI.pm` module, which provides a simple and efficient way to write CGI programs in Perl. A CGI program to process a product order form is then presented. This is followed by another complete example, including the XHTML code to describe a survey form and the two CGI programs that are used to provide the processing required for the form. The chapter closes with an introduction to cookies and how they are created and used in Perl CGI programs.

9.1 The Common Gateway Interface

XHTML is a markup language and, as such, cannot by itself describe computations, allow interaction with the user, or provide access to a database. Yet these are things that are commonly needed in Web-based systems. Computations are required to provide support for all kinds of Web commerce. The Web also is now a common way to access databases—for example, for making reservations for transportation services. Such applications obviously require interaction with clients. One early response to these needs was to develop a technique for a browser to access the software resources of the server machine. Using this approach, a browser can run programs indirectly on the server, including database access systems for the databases that reside on the server. These server-based programs communicate back to the client through HTTP. The protocol that is used between a browser and software on the server is called the *Common Gateway Interface*.

Before discussing CGI further, it is useful to take a brief look at its alternatives. There are now several popular approaches to providing server-side computation, including support for dynamic documents. As just discussed, the first of these uses the CGI to run software on the server that is completely separate from markup documents. CGI is part of a general approach that is often called LAMP, which is an acronym for Linux, Apache, and MySQL. The 'P' part of the acronym is generic—it represents one of the server-side programming and/or scripting languages, Perl, PHP, or Python. This chapter discusses using Perl and CGI. Chapter 11, "Introduction to PHP," discusses using PHP as an XHTML-embedded, server-side scripting language. This book does not cover Python, another scripting language. MySQL, a database management system, as used in conjunction with Perl, Java, and PHP in Web database access systems, is discussed in Chapter 13, "Database Access through the Web."

A second approach to providing server-side computation and dynamic documents is Microsoft's product, ASP. The latest incarnation of ASP is ASP.NET, an important part of the .NET computing platform. ASP.NET documents are mixtures of XHTML markup and either programming language code or references to external programs. These documents are compiled into classes in which the XHTML code is replaced by output statements that produce that XHTML code. ASP.NET documents can use any of the .NET programming languages. ASP.NET is discussed in Chapter 12, "Introduction to ASP.NET."

Yet another approach to providing server-side computation and dynamic documents is Java servlets and Java Server Pages (JSP). Servlets are Java classes with close connections to XHTML documents, which allow them to interact with Web clients conveniently. JSP is similar to ASP.NET in that programming code can be embedded in markup documents and programs can be referenced in documents to provide computational support. The primary difference between JSP and ASP.NET is that JSP restricts the developer to Java rather than a range of programming and scripting languages. JSP documents are converted to Java servlets before they are compiled. Both servlets and JSP are discussed in Chapter 10, "Servlets and Java Server Pages."

Now we can return to CGI. An HTTP request to run a CGI program is like any other HTTP request except that the requested file can be identified by the server as being a CGI program. Servers can identify CGI programs by their addresses on the server or by their filename extensions. When a server receives a request for a CGI program, it does not return the file—it executes the program in the file and returns that program's output.

A CGI program can produce results in a number of different forms. Most often, an XHTML document is returned. When a new document is generated, it consists of two parts: an HTTP header and a body. The header must be complete if the response is going directly to the client. It can be partial if the response is going to the client through the server, which completes the header. This is the usual process. The form of the body, which is specified in the header, can be XHTML, plain text, or even an image or audio file.

One common way for a browser user to interact with a Web server is through forms. A form is presented to the user, who is invited to fill in the text boxes and click the buttons of the form. The user submits the form to the server by clicking its *Submit* button. The contents of the form are encoded and transmitted to the server. The server must use a program to decode the transmitted form contents, perform whatever computation is necessary on the form data, and produce its output. Figure 9.1 shows the communications configuration for CGI.

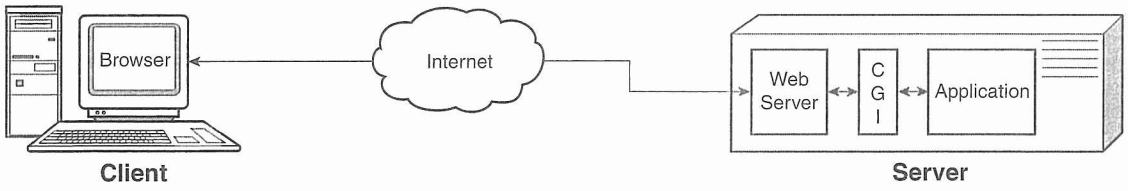


Figure 9.1 Communications and computation using CGI

Both CGI programs and client-side scripts, such as JavaScript, support dynamic documents. However, client-side scripts cannot access files and databases on the server.

9.2 CGI Linkage

This section describes how the connection between an XHTML document being displayed by a browser and a CGI program on the server is established.

CGI programs often reside in one specific directory on the server, `cgi-bin`, which is usually a subdirectory of `public_html`[1]. However, the Web server administrator can choose a different directory name and place for CGI

1. `public_html` is the directory in which the publicly accessible XHTML document files are stored.

programs. The only requirement is that the CGI programmer knows the location of CGI programs. The examples in this chapter assume that CGI programs are stored in a directory named `cgi-bin`, which is a subdirectory of the directory in which XHTML documents are stored.

A server can be configured to recognize the requested file as being a program by the extension on the file's name. For example, if the extension on the requested file's name is `.pl`, that could indicate that it is a Perl CGI program. Another extension that indicates the file is a Perl program is `.cgi`. Because CGI programs are invoked by the server, their access protection code must be set to allow this.

If a CGI program has been compiled and is in machine code, the server can invoke it directly. However, the compiled versions of Perl programs normally are not saved and are not in machine code anyway, so the `perl` system must be invoked on every Perl CGI program.[2] For UNIX-based Web servers, this can be done by adding a special line to the beginning of the program that specifies that `perl` must be run, using the remainder of the program file as input data to `perl`. This special line also must specify the location of `perl`. On UNIX systems, `perl` is often stored in `/usr/local/bin`. If so, the first line is as follows:

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

The `#!` specifies that the program whose location follows must be executed on the rest of the file. And, of course, `/usr/local/bin/perl` is the path to the `perl` system. If `perl` resides in some other directory, that directory's name is used in place of `/usr/local/bin`. Because `!` is called "bang" in the UNIX world, the `#!` line is often called "shebang."

Windows-based Web servers do not require the shebang line in Perl CGI programs.

An XHTML document can specify a call to a CGI program using an anchor tag (`<a>`), which must include a hypertext reference attribute (`href`). In this case, a CGI program call is similar to a link to an XHTML document. The value of `href` must be the complete URL of the CGI program's filename. Consider the following anchor tag:

```
<a href = "./cgi-bin/reply.cgi">
Click here to run the CGI program reply.cgi
</a>
```

The content of the anchor tag specifies the actual link in the XHTML. In this example, `cgi-bin`, which is a subdirectory of where the calling XHTML document is stored, is the directory where CGI programs are stored, and `reply.cgi` is the name of the CGI program. Some educational institutions now use a system called CGI Wrap to allow students to run CGI programs in a relatively secure environment This system allows the CGI program to be in the

2. Systems are available that produce and save compiled versions of Perl programs for use on Web servers. For example, look at `http://perl.apache.org/list/mod_perl.html`.

student's directory rather than on the server system. For CGI Wrap, Perl CGI programs must use the `.cgi` extension to their filenames.

The most common application of CGI programs is form processing, so most CGI programs are not called from XHTML documents with anchor tags. Rather, they are called as a side effect of *Submit* button clicks, with the CGI program being specified on the action attribute of the form tag. This is exemplified in Section 9.4.2.

The following document calls a trivial Perl CGI program through its anchor tag:

```
<?xml version = "1.0" encoding = "utf-8"?>
<!DOCTYPE html PUBLIC "-//w3c//DTD XHTML 1.1//EN"
 "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">

<!-- reply.html
     A trivial document to call a simple Perl CGI program
     -->
<html xmlns = "http://www.w3.org/1999/xhtml">
  <head>
    <title> XHTML to call the Perl CGI program, reply.cgi
    </title>
  </head>
  <body>
    <p>
      This is our first Perl CGI example
      <br /><br />
      <a href = "./cgi-bin/reply.cgi">
      Click here to run the CGI program, reply.cgi
      </a>
    </p>
  </body>
</html>
```

We have implied that the CGI program is always on the server that sent the XHTML document to the browser. However, the `href` attribute can be assigned the URL of any CGI program on any Web server on the Internet.

The obvious next step is to describe the CGI program, `reply.cgi`, which simply returns a greeting to the browser (through the server). The connection from a CGI program to the client is through standard output. Therefore, anything the CGI program sends to standard output is sent to the server, which sends it on to the client. In a Perl CGI program, this means that the `print` function is used to communicate to the client.

As previously stated, the HTTP response to the client can be in a variety of different forms, but it must begin with the HTTP header. The CGI program

often supplies only the first line of the header; the remainder is added by the server before the response document is sent to the client. The first line of the header specifies the form of the response as a MIME content type. In this book, we consider only XHTML as the response content type, which is specified with `text/html`.

The line following the one-line header *must* always be blank. The blank line indicates the end of the HTTP header. Therefore, the first output from all Perl CGI programs that return partial headers and XHTML response bodies is generated with the following statement:

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

Since the output of a CGI program is ultimately for the browser, it must be in the form of an HTML (or XHTML) document. Such a document is created in Perl by using `print` statements to output the XHTML text.

If the only task of a CGI program is to send one line of information, most of the program will be used to generate the required tags for the XHTML file. The following is the Perl program `reply.cgi`:

```
#!/usr/local/bin/perl -w
# reply.cgi
# This CGI program returns a greeting to the client

print "Content-type: text/html \n\n",
 "<?xml version = '1.0' encoding = 'utf-8'?> \n",
 "<!DOCTYPE html PUBLIC '-//w3c//DTD XHTML 1.1//EN'\n",
 "'http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd'>\n",
 "<html xmlns = 'http://www.w3.org/1999/xhtml'>\n",
 "<head><title> reply.cgi example </title></head>\n",
 "<body>\n",
 "<h1> Greetings from your Web server! </h1>\n",
 "</body></html>";
```

9.3 Query String Format

HTTP requests made by the GET and POST methods transmit the data associated with the request to the Web server. This data is in the form of a character string called a *query string*. This section describes the format of a query string.

Query strings can be handwritten and used directly with GET and/or POST HTTP methods. However, the great majority of query strings are those that code form data and are constructed by the browser when the *Submit* button of the form is clicked. The following paragraphs describe query strings in terms of form data.

For each element (widget) in the form that has a value, the widget's name and that value are coded as a character string of the form *name=value* and

included in the query string. A widget's value is always a character string. For example, if the value of a radio button named `payment` is `discover`, it is coded in the query string as follows:

```
payment=discover
```

If the form has more than one widget, their name/value pairs are separated by ampersands (`&`):

```
caramel=7&payment=discover
```

If there are special characters in the value of a widget, they are coded as a percent sign (`%`) followed by a two-character hexadecimal number that is the ASCII code for that character.[3] For example, the hexadecimal ASCII code for the exclamation point (`!`) is `21`, so that character is coded as `%21`. Likewise, because the hexadecimal ASCII code for a space is `20`, spaces are coded as `%20`. Suppose we had a form whose only widgets were a radio button collection named `payment` whose value was `visa` and a text widget whose name was `saying` and whose value was `"Eat your fruit!"`. The query string for this form would be as follows:

```
payment=visa&saying=Eat%20your%20fruit%21
```

Some browsers replace spaces in form data values with plus signs (+). For these, the previous example would be coded as follows:

```
payment=visa&saying=Eat+your+fruit%21
```

Recall from Chapter 2, "Introduction to XHTML," that the `method` attribute of `<form>` specifies one of the two techniques, `get` or `post`, used to pass the form data to the server. `get` is the default, so if no `method` attribute is given in the `<form>` tag, `get` will be used. The alternative technique is `post`. In both techniques, the form data is coded into a query string when the user clicks the *Submit* button. When the `get` method is used, the browser attaches the query string to the URL of the CGI program, so the form data is transmitted to the server with the URL. The browser inserts a question mark at the end of the actual URL just before the first character of the query string so that the server can easily find the beginning of the query string. The server removes the query string from the URL and places it in the environment variable `QUERY_STRING`, where it can be accessed by the CGI program. The `get` method can also be used to pass parameters to the server when forms are not involved (this cannot be done with `post`). The main disadvantage of the `get` method is that some servers place a limit on the length of the URL string and truncate any characters past the limit. So, if the form has more than a few widgets, `get` is not a good choice.

The following is an example of a URL with a query string:

```
http://cs.ucp.edu/cgi-bin/rws/test.cgi?sender=bob&day=2
```

3. When the `GET` method is used, the query string becomes part of a URL, in which certain special characters can cause problems.

When the post method is used, the query string is passed through standard input to the CGI program so that the CGI program can simply read the string. The length of the query string is passed through the environment variable CONTENT_LENGTH. There is no length limitation for the query string with the post method, so it is obviously the better choice when there are more than a few widgets in the form.

The following code, as part of a CGI program that handles GET or POST requests, fetches the query string and places it is $query_string, regardless of which method was used.

```
$request_method = $ENV{'REQUEST_METHOD'};
if ($request_method eq "GET") {
   $query_string = $ENV{'QUERY_STRING'};
}
elsif ($request_method eq "POST") {
   read(STDIN, $query_string, $ENV{'CONTENT_LENGTH'})
}
else {
   print "Error - the request method is illegal \n";
   exit(1);
}
```

Notice that although the value of the method attribute of the form element in XHTML must be lowercase, when that value is stored in the REQUEST_METHOD environment variable, it is in uppercase.

9.4 The CGI.pm Module

Much of what a CGI program must do is routine—that is, it is nearly the same for all CGI programs. Among these common tasks are creating the required tags, such as <html> and </html>, and creating form elements such as buttons and lists. Therefore, it is natural to have standard routines to do these things. CGI.pm, which was developed by Lincoln Stein, is a Perl module of functions for these common tasks.[4] The CGI.pm module is part of the standard Perl distribution. Its documentation can be obtained with the following command:

```
perldoc CGI
```

A Perl program specifies that it needs access to a particular module with the use declaration. In the case of CGI.pm, only a part of the module is usually needed. The part we need here, which is the most often used part, is named :standard. The following declaration provides access to this collection of resources from CGI.pm:

```
use CGI ":standard";
```

4. A Perl module can be thought of as a library of Perl functions.

9.4.1 Common CGI.pm Functions

Many of the functions in CGI.pm produce XHTML tags. In these cases, the functions have the names of their associated XHTML tags. These functions are called *shortcuts*. Shortcut functions produce their tags by using the parameters passed to them for the attributes and content of the tag. They may also generate embedded tags. If there are no attributes and no embedded tags, the only parameter is the content of the tag. A shortcut function that takes no parameters may be called without the parentheses. For example, the following call to br:

```
br;
```

returns

```
<br/>
```

Note that the shortcut functions do not produce any output—they simply return strings. So, to get a break in the output, use the following:

```
print br;
```

A shortcut function may take several different kinds of parameters, including a scalar literal or variable, a reference to a hash, or a reference to an array or list. When there is just one parameter, the CGI function call has the same form as that for any other Perl function. For example, in the following call, the only parameter is the content of the tag:

```
print h1("This is the real stuff");
```

This produces the following:

```
<h1> This is the real stuff </h1>
```

CGI programs sometimes create and send forms back to the client to gather information. In the following paragraphs, we use some examples of CGI.pm functions that produce widgets. Tags can have both content and attributes. In the previous example of the h1 function, the parameter becomes the content of <h1>. Each attribute of a tag is passed in the name/value pair form that is used in a hash literal, where the name and value are separated by =>. The name of the attribute is the key in the pair, and the attribute value is its value. The attribute names are preceded by minus signs. For example, consider the following call to textarea:

```
print textarea(-name => "Description",
               -rows => "2",
               -cols => "35"
);
```

This produces the following:

```
<textarea name="Description" rows="2" cols="35">
</textarea>
```

If both attributes and content are passed to a shortcut function, the attributes are specified in a hash literal, which must be the first parameter. For example, the statement

```
print a({-href => "fruit.html"},
      Press here for fruit descriptions");
```

generates

```
<a href="fruit.html"> Press here for fruit descriptions </a>
```

One convenient characteristic of the shortcut functions is that the tags and their attributes are distributed over the parameters to the function. For example, consider the following call:

```
print ol(li({-type => "square"},
      ["milk", "bread", "cheese"]));
```

Notice that a reference to the literal array, `("milk", "bread", "cheese")`, is passed as a parameter to `li`. In this example, the `li` tag and its attribute are distributed over the list items. This produces the following:[5]

```
<ol>
    <li type="square" milk</li>
    <li type="square" bread</li>
    <li type="square" cheese</li>
</ol>
```

As another example, consider the following call to `radio_group`, in which the collection name is specified just once, and the values can be specified as a reference to an array:

```
print radio_group(-name => 'colors',
              -value => ['blue', 'green', 'yellow', 'red'],
              -default => 'blue');
```

This produces the following:

```
<input type="radio" name="colors" value="blue" checked /> blue
<input type="radio" name="colors" value="green" /> green
<input type="radio" name="colors" value="yellow" /> yellow
<input type="radio" name="colors" value="red" /> red
```

The `CGI.pm` functions that are not shortcuts often produce the boilerplate that is part of every XHTML document. The first thing most CGI programs must do is produce the content type line. The `header` function does precisely this. Specifically,

```
print header;
```

5. Actually, the produced XHTML is returned as a single string.

creates

```
Content-type: text/html;charset=ISO-8859-1
-- blank line --
```

Notice that `CGI.pm` includes the `charset` value `ISO-8859-1` in the output of `header`. We do not include `charset` in the HTTP headers of our non-`CGI.pm` examples because the default character set, ISO-8859-1, is fine. If the content type is not `text/html`, the alternative content type can be specified as the parameter to `header`.

The next required XHTML output of a CGI program is the head part of the document. This is generated by the `start_html` function, whose parameter specifies the title of the document. For example, the call

```
print start_html("Paul's Gardening Supplies");
```

produces

```
<!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" lang="en-US"
            xml:lang="en-US">
<head><title> Paul's Gardening Supplies </title>
<meta http-equiv="Content-Type" content="text/html;
                  charset=iso-8859-1"> </head>
<body>
```

Section 9.3 describes the format of the query string. Any form-processing program must first extract the form data from the query string. This means it must be split into its name/value pairs, which then must be split to access the values. Any characters coded as hexadecimal ASCII must be decoded, and any plus signs must be replaced by spaces. The whole process of decoding the query string and setting local variables to the values in the query string can be done by calls to the `CGI.pm` function, `param`. For example, if the query string is

```
name=Bob%20Brumbel&payment=visa
```

then the following statement sets `$name` to `"Bob Brumbel"`:

```
my $name = param("name")
```

So, for the whole query string, the following statement could be used:

```
my($name, $payment) = (param("name"), param("payment"));
```

This sets the local variables `$name` and `$payment` to the values of the form values for `"name"` and `"payment"` (`"Bob Brumbel"` and `"visa"`, respectively).

The last XHTML code required in a document is generated with `end_html`, which produces `</body> </html>`.

The `CGI.pm` functions for creating XHTML tables are discussed in Section 9.5.

9.4.2 A Complete Form Example

The following XHTML code describes a form for taking sales orders for popcorn. Three text widgets are used at the top of the form to collect the buyer's name and address. These are placed in a borderless table to force the text boxes to align vertically. A second table is used to collect the actual order. Each row of this table names a product with the content of a `<td>` tag, displays the price with another `<td>` tag, and uses a text widget with `size` set to `2` to collect the quantity ordered. The payment method is input by the user through one of four radio buttons. Recall that this form also appears in Chapter 2.

```
<?xml version = "1.0" encoding = "utf-8"?>
<!DOCTYPE html PUBLIC "-//w3c//DTD XHTML 1.1//EN"
  "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">

<!-- popcorn.html
     This describes popcorn sales form page>
     -->
<html xmlns = "http://www.w3.org/1999/xhtml">
  <head> <title> Popcorn Sales Form </title>
  </head>
  <body>
    <h2> Welcome to Millenium Gynmastics Booster Club Popcorn
         Sales
    </h2>

<!-- The next line gives the address of the CGI program -->
    <form action = "./cgi-bin/popcorn.cgi"
          method = "post">
      <table>

<!-- Text widgets for name and address -->
        <tr>
          <td> Buyer's Name: </td>
          <td> <input type = "text" name = "name" size = "30">
          </td>
        </tr>
        <tr>
          <td> Street Address: </td>
          <td> <input type = "text" name = "street" size = "30">
          </td>
        </tr>
        <tr>
          <td> City, State, Zip: </td>
          <td> <input type = "text" name = "city" size = "30">
```

```
          </td>
        </tr>
      </table>
      <br />
      <table border = "border">

<!-- First, the column headings -->
        <tr>
          <th> Product Name </th>
          <th> Price </th>
          <th> Quantity </th>
        </tr>

<!-- Now, the table data entries -->
        <tr>
          <td> Unpopped Popcorn (1 lb.) </td>
          <td> $3.00 </td>
          <td> <input type = "text" name = "unpop" size ="2">
          </td>
        </tr>
        <tr>
          <td> Caramel Popcorn (2 lb. cannister) </td>
          <td> $3.50 </td>
          <td> <input type = "text" name = "caramel" size = "2">
          </td>
        </tr>
        <tr>
          <td> Caramel Nut Popcorn (2 lb. cannister) </td>
          <td> $4.50 </td>
          <td> <input type = "text" name = "caramelnut" size = "2">
          </td>
        </tr>
        <tr>
          <td> Toffey Nut Popcorn (2 lb. cannister) </td>
          <td> $5.00 </td>
          <td> <input type = "text" name = "toffeynut" size = "2">
          </td>
        </tr>

      </table>
      <br />

<!-- The radio buttons for the payment method -->
      <h3> Payment Method: </h3>
      <p>
```

```
        <label>
          <input type = "radio" name = "payment" value = "visa"
                 checked = "checked" /> Visa <br />
        </label>
        <label>
          <input type = "radio" name = "payment"
                 value = "mc" />  Master Card <br />
        </label>
        <label>
          <input type = "radio" name = "payment"
                 value = "discover"/> Discover <br />
        </label>
        <label>
          <input type = "radio" name = "payment"
                 value = "check" /> Check <br />
        </label>
      </p>

<!-- The submit and reset buttons -->
      <p>
        <input type = "submit" value = "Submit Order" />
        <input type = "reset" value = "Clear Order Form" />
      </p>
    </form>
  </body>
</html>
```

Figure 9.2 shows a browser display of `popcorn.html`. Figure 9.3 shows an example of this form after it has been filled out.

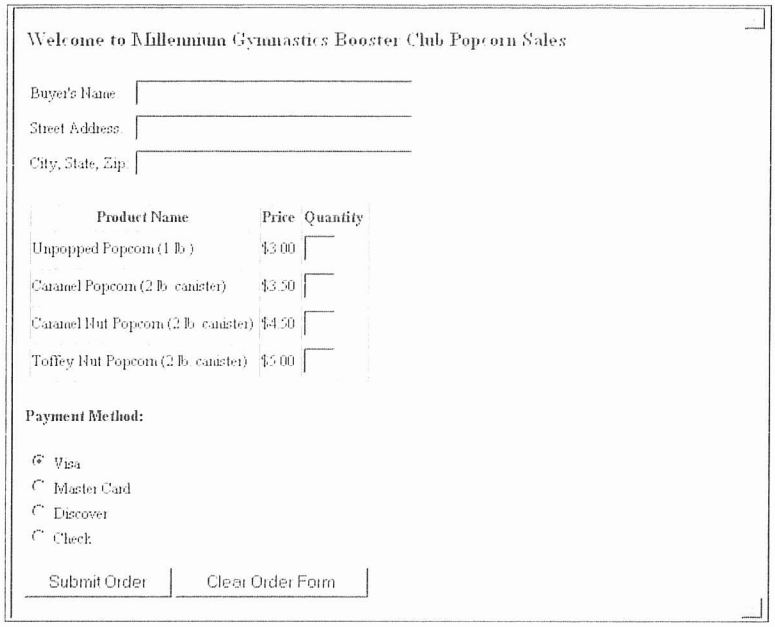


Figure 9.2 Display of `popcorn.html`

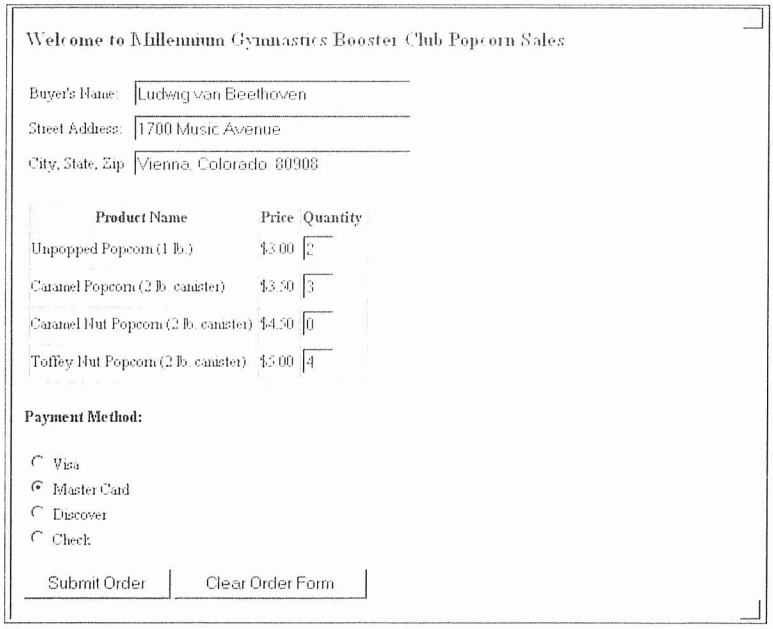


Figure 9.3 Display of `popcorn.html` after the form is filled out

We now can consider the CGI program that will process the data from the popcorn order form. The program must compute the costs of the ordered items, determine the total sale amount, and send them back to the user as an XHTML document. The program must determine what was ordered and compute its cost. For the item orders, the program must use the values to compute the total price and total number of items in the order. Finally, this program must build the XHTML code to reply to the user who placed the order. The complete program, called `popcorn.cgi`, follows:

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

# popcorn.cgi
# A CGI program to process the popcorn sales form

use CGI ":standard";

#>>> Initialize total price and total number of purchased items
$total_price = 0;
$total_itemi = 0;

#>>> Produce the header part of the HTML return value
print header;
print start_html("CGI-Perl Popcorn Sales Form, using CGI.pm");

#>>> Set local variables to the parameter values
my($name, $street, $city, $payment) =
      (param("name"), param("street"),
       param("city"), param("payment"));
my($unpop, $caramel, $caramelnut, $toffeynut) = (param("unpop"),
        param("caramel"), param("caramelnut"),
        param("toffeynut"));

#>>> Compute the number of items ordered and the total cost
$total_price = 3.0 * $unpop + 3.5 * $caramel + 4.5 * $caramelnut +
               5.0 * $toffeynut;
$total_items = $unpop + $caramel + $caramelnut + $toffeynut;


#>>> Produce the result information for the browser and
#>>>  finish the page
print h3("Customer:"), "\n";
print "$name<br/>\n", "$street <br/>\n", "$city <br/><br/>\n";
print "<b>Payment method:</b> $payment <br/><br/>\n";
print h3("Items ordered:"), "\n";
```

```
if ($unpop > 0) {print "Unpopped popcorn: $unpop <br/>\n";}
if ($caramel > 0) {print "Caramel popcorn: $caramel <br/>\n";}
if ($caramelnut > 0)
  {print "Caramel nut popcorn: $caramelnut <br/>\n";}
if ($toffeynut > 0)
  {print "Toffey nut popcorn: $toffeynut <br/>\n";}
print "Thank you for your order <br/><br/>\n";
print "<b>Your total bill is:</b> \$ $total_price <br/> \n";
print end_html;
```

Figure 9.4 shows the results of running `popcorn.cgi` on the filled out form shown in Figure 9.3.

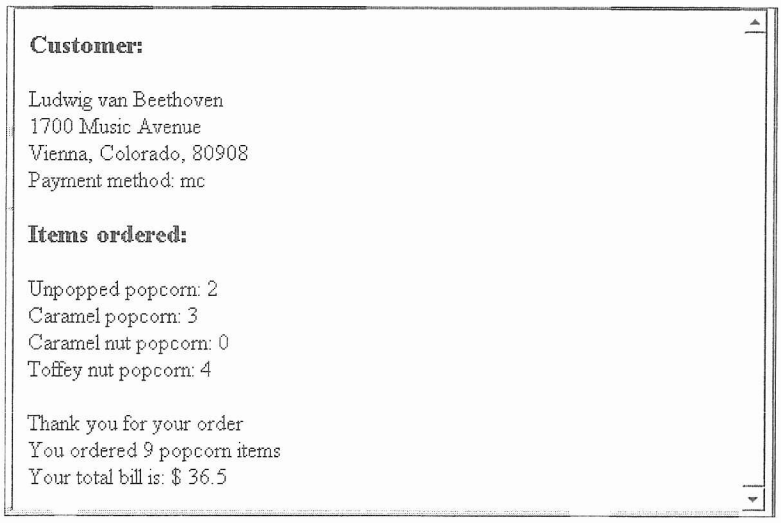


Figure 9.4 Output of `popcorn.cgi`

9.5 A Survey Example

It is common for a Web document to conduct a survey of the opinions or choices of the people who visit the document. The document to gather the information is similar to the popcorn sales example. The most important difference in the CGI program that processes the inputs for a survey is that the data must be stored between visits. The obvious place to store such data is on the disk on the server. This file can reside in the directory where CGI programs are stored. We use a file (rather than a database) in this example because of the simplicity of the data that needs to be stored. However, it many cases a database would be a better choice.

For this example, we use two CGI programs: one to process a client's response to the survey form and one to produce the latest totals from the survey, which will be produced when the user clicks a link in the document.

One of the complications with this application is that more than one user may submit survey forms concurrently, which could lead to corruption of the data if the changes occur at about the same time. For example, suppose two clients simultaneously request executions of a CGI program that reads a number from a file, increments the number, and rewrites the file. Consider the following scenario: The initial value in the file is 27. Two browsers simultaneously call the CGI program. The three steps of the CGI program (read the file, increment the value from the file, rewrite the file) can become interleaved. One possible sequence of these steps is as follows. (We name the two browser's executions of the CGI program CGI1 and CGI2.)

1. CGI1 reads the file into its variable, `counter` (value is 27).
2. CGI2 reads the file into its variable, `counter` (value is 27).
3. CGI1 increments its `counter` to 28.
4. CGI1 rewrites the file, which now has 28.
5. CGI2 increments its `counter` to 28.
6. CGI2 rewrites the file, which still has 28.

So, although the file now should have 29, it has 28.

Such multiple simultaneous accesses can be prevented by requiring the programs that access the file to lock the file before accessing it and unlock it when they are finished.[6] The Perl function `flock` can be used to both lock and unlock the file. `flock` takes two parameters: the filehandle of the file to be locked or unlocked, and the specified process (lock or unlock).

Actually, locking the file with `flock` does not directly block all accesses to the file; rather, it prevents any other process from acquiring the file's lock. So, it effectively blocks other processes from accessing the file only if they first attempt to lock the file. For our example we assume that no other process will attempt to use the file, so this works.

The values for the parameter to `flock` are defined in the Perl `Fcntl` module, which is made accessible with the following:

```
use Fcntl qw(:DEFAULT :flock);
```

Because closing a file implicitly unlocks it, we only need the parameter value for a write lock, which is `LOCK_EX`. The following example illustrates the process of updating a file using locking:

```
use Fcntl qw(:DEFAULT :flock);
open(TAX_DATA, "+<taxdata") or
   die "TAX_DATA could not be opened";
```

6. Locking the file means that all other accesses to it are temporarily blocked. Unlocking it makes the file available to other accesses.

```
flock(TAX_DATA, LOCK_EX) or
   die "TAX_DATA could not be locked";
chomp($tax = <TAX_DATA>);
# update $tax
# Rewind the file so the write goes to its beginning
seek(TAX_DATA, 0, 0) or
  die "TAX_DATA could not be rewound";
print TAX_DATA $tax or
  die "TAX_DATA could not be rewritten";
close TAX_DATA or die "TAX_DATA could not be closed";
```

The survey form collects votes on what consumer electronics device the client is most likely to purchase in the next six months. The form will also collect the age and gender of the client. A visitor to the document can vote, see the current voting results, or both. So the document needs two links, one to the CGI program that records a vote and one that produces a table of current results. The survey document is described by the following code:

```
<?xml version = "1.0" encoding = "utf-8"?>
<!DOCTYPE html PUBLIC "-//w3c//DTD XHTML 1.1//EN"
 "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">

<!-- conelec.html
     A document to present the user with a consumer electronics
     purchasing survey form -->
<html xmlns = "http://www.w3.org/1999/xhtml">
  <head>
    <title> Consumer Electronics Purchasing Survey </title>
  </head>
  <body>
    <form action = "./cgi-bin/conelec1.cgi" method = "post">
      <h2> Welcome to the Consumer Electronics Purchasing Survey
      </h2>
      <p />
      <h4> Your Age Category: </h4>
      <p>
        <label>
          <input type = "radio" name = "age" value = "0"
                 checked = "checked" /> 10-25 <br/>
        </label>
        <label>
          <input type = "radio" name = "age" value = "1"/>
            26-40 <br/>
        </label>
        <label>
```

```
      <input type = "radio" name = "age" value = "2"/>
        41-60 <br/>
    </label>
    <label>
      <input type = "radio" name = "age" value = "3"/>
        Over 60 <br /> <br />
    </label>
  </p>
  <h4> Your Gender: </h4>
  <p>
    <label>
      <input type = "radio" name = "gender" value = "0"
             checked = "checked" /> Female <br/>
    </label>
    <label>
      <input type = "radio" name = "gender" value = "4"/>
        Male <br /> <br />
    </label>
  </p>
  <h4> Your Next Consumer Electronics Purchase will be: </h4>
  <p>
    <label>
      <input type = "radio" name = "vote" value = "0"/>
        Conventional TV <br />
    </label>
    <label>
      <input type = "radio" name = "vote" value = "1"/>
        HDTV <br />
    </label>
    <label>
      <input type = "radio" name = "vote" value = "2"/>
        VCR <br />
    </label>
    <label>
      <input type = "radio" name = "vote" value = "3"/>
        CD player <br />
    </label>
    <label>
      <input type = "radio" name = "vote" value = "4"/>
        Mini CD player/recorder <br />
    </label>
    <label>
      <input type = "radio" name = "vote" value = "5"/>
        DVD player <br/>
    </label>
```

```
        <label>
          <input type = "radio" name = "vote" value = "6"
                 checked = "checked" /> Other <br /> <br/>
        </label>

        <input type = "submit" value = "Submit Order" />
        <input type = "reset" value = "Clear Order Form"/>
      </p>
    </form>

    <hr/>
    <p>
      To see the results of the survey so far, click
      <a href = "./cgi-bin/conelec2.cgi"> here </a>
    </p>
  </body>
</html>
```

Figure 9.5 shows the display of `conelec.html`.

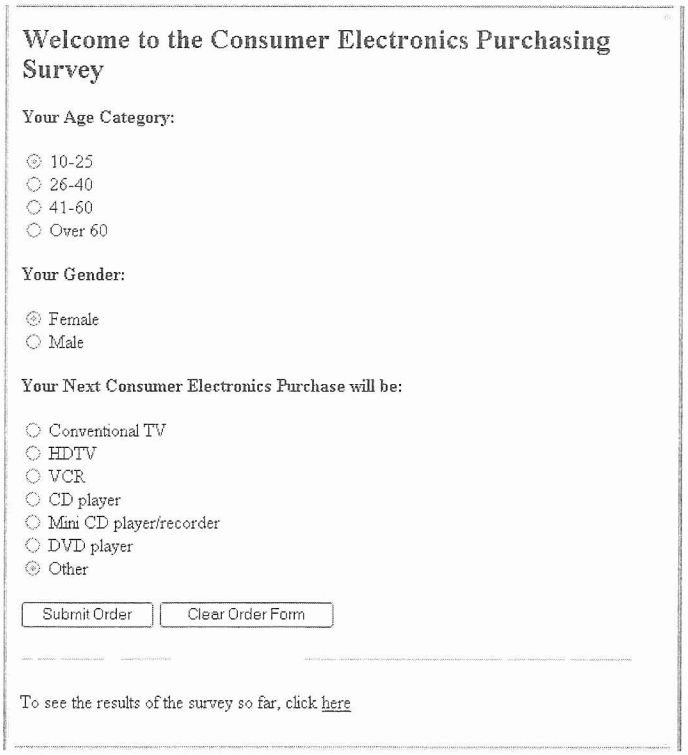


Figure 9.5 Display of `conelec.html`

Before we can develop either of the two CGI programs for this application, we must design the file format to be used to store the data. The simple format we will use is eight strings that represent the rows of voting totals. The first four rows store the votes from female visitors; the last four are for the male visitors. Initially, the file will have eight lines, each being a string of seven zeros (one for each possible vote), separated by spaces. The file must be created and initialized to all zeros before the survey page is made public. This can be done with a text editor because it is simple text.

Note in `conelec.html` that the values `0`, `1`, `2`, and `3` are used for the radio buttons for selecting the age group. Also, the values `0` and `4` are used for the gender selection. These values were chosen so that the index of the element of the vote data array that must be changed can be computed by adding the age group value and the gender value. For example, a male voter (gender = `4`) who selects the 41–60 age group (age = `2`) affects the element with the index `6`.

It is always a good idea to check that file operations succeeded. In Chapter 8, "The Basics of Perl," the `die` function is used with `open` to produce an error message and terminate the program when `open` fails. In a CGI program, this is a bad idea because if a file operation fails and the program dies, the client will not know what happened. Instead, a message must be returned to the client indicating the problem. In the two CGI programs for the survey form document, a function named `error` is included to produce a message and terminate the program. Because the message must go to the client, it must be in the form of XHTML.

The CGI program that collects and records the voting data is named `conelec1.cgi`. Its tasks are as follows:

1. Get the form values.
2. Determine which row of the file must be modified.
3. Open, lock, and read the survey data file.
4. Split the affected data string into numbers and store them in an array.
5. Modify the affected array element and join the array back into a string.
6. Rewind the data file (with `seek`).
7. Rewrite and close the survey data file.

We already know how to get the widget values from the query string, and reading and writing the survey data file is relatively simple. Its lines will be read into an array of strings, one array element per row of the file. Each of the strings stores seven numbers. Because each submitted voting form changes just one element of the file, only one row must be converted from a string to an array of numbers. As previously explained, the row to be changed can be determined by adding the `age` and `gender` form data values. The `gender` form value determines which half of the file is affected. The `age` form data will determine which of the four rows of the `gender`'s half is affected. The actual change amounts to splitting the row into an array of numbers, adding 1 to one of its elements,

which is determined by the particular vote, and joining the resulting array back together as a string.

The following is the CGI program to implement the processes previously described:

```
#!/usr/local/bin/perl
# conelec1.cgi
# This CGI program processes the consumer electronics survey form
# and updates the file that stores the survey data, survdat.dat
use CGI ":standard";
use Fcntl qw(:DEFAULT :flock);

# error - a function to produce an error message for the client
#         and exit in case of input/output errors
sub error {
    print start_html();
    print "Error - input/output error in conelec1.pl <br/>";
    print end_html();
    exit(1);
}

#>>> Begin the main program
#>>> Get the form values
my($age, $gender, $vote) = (param("age"), param("gender"),

#>>> Produce the header for the reply page - do it here so error
#>>>  messages appear on the page
print header;

#>>> Set $index to the line index of the current vote
$index = $age + $gender;

#>>> Open and lock the survey data file
open(SURVDAT, "+<survdat.dat") or error();
flock(SURVDAT, LOCK_EX) or error();

#>>> Read the survey data file, unlock it, and close it
for ($count = 0; $count <= 7; $count++) {
   chomp($file_lines[$count] = <SURVDAT>);
}

#>>> Split the line into its parts, increment the chosen device,
#>>>  and put it back together again
@file_votes = split / /, $file_lines[$index];
```

```
$file_votes[$vote]++;
$file_lines[$index] = join(" ", @file_votes);

#>>> Rewind the file for writing
seek(SURVDAT, 0, 0) or error();

#>>> Write out the file data and close it (which also unlocks it)
foreach $line (@file_lines) {
    print SURVDAT "$line\n";
}

close(SURVDAT);

#>>> Build the web page to thank the survey participant
print start_html("Thankyou");
print "Thank you for participating in our survey <br/> <br/> \n";
print end_html;
```

We now can develop the program, named `conelec2.cgi`, which produces the current status of the voting in the consumer electronics preference survey. Because there are vote totals in 56 different categories, it is best that the results are displayed in a table. Building this table in XHTML would be tedious without `CGI.pm`, and even with this module it is nontrivial.

Tables with `CGI.pm` are introduced through the following example: Suppose you want a CGI program to return a table of sales figures for the sales personnel Mary, Freddie, and Spot, including numbers for each of the five workdays in a week. The people will be the rows of the table; the days of the week will be the columns. Assume the sales figures are in three arrays, `@marysales`, `@freddiesales`, and `@spotsales`, each of which has five elements. The column titles are `Salesperson` (for the title row), `Monday`, `Tuesday`, `Wednesday`, `Thursday`, and `Friday`. The row titles for the three arrays are `Mary`, `Freddie`, and `Spot`.

The `table` function creates an entire table, which is specified with its parameters. If the table is to have a border, that is specified with the attribute `border`. The table's caption, if there is one, is created with the `caption` function:

```
caption("Sales Figures");
```

Each row of a table is created by a call to `Tr`. This shortcut is spelled with an uppercase first letter because, if it were not, it would be mistaken for the Perl transliterate operator, `tr`. The other parameters for `Tr` specify the row data of the table. The title row is built with a call to `th`, and the data rows are built with calls to `td`. All three functions, `Tr`, `th`, and `td`, take either references or arrays as parameters. Therefore, if a literal list is to be passed, it must be constructed

with brackets. If a parameter is an array, its name must be backslashed in the actual parameter list to specify its address.

Consider the following examples of the `th` shortcut function:

```
print th(['apples', 'oranges', 'grapes']);
```

produces

```
<th> apples </th>
<th> oranges </th>
<th> grapes </th>
```

and

```
@days = ('Saturday', 'Sunday');
print th(\@days);
```

produces

```
<th> Saturday </th>
<th> Sunday </th>
```

The `td` function is exactly like the `th` function.

In many cases, we want to build a table row that consists of a `th` element and several `td` elements and treat the whole thing as a single parameter to `Tr`. To treat these as a single row string, the returned values of the calls to `td` and `th` can be catenated together. For example, the code

```
print th('9:15').td('French', '', 'French', '');
```

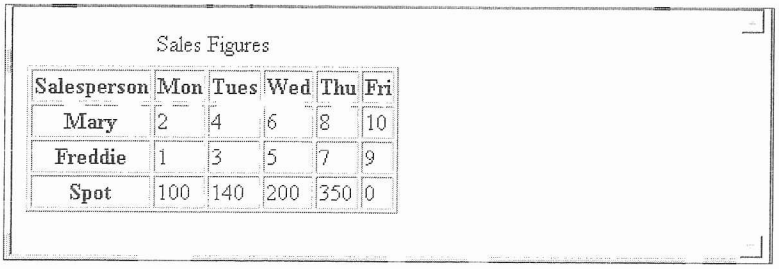
produces

```
<th> 9:15 </th> <td> French </td> <td> </td>
                <td> French </td> <td> </td>
```

Let us now consider a complete table specification.

```
# easy_table.pl
table({-border => "border"},
      caption("Sales Figures"),
       Tr(
         [th(["Salesperson", "Mon", "Tues", "Wed",
              "Thu", "Fri"]),
          th("Mary").td(\@marysales),
          th("Freddie").td(\@freddiesales),
          th("Spot").td(\@spotsales),
       ]
      )
     );
```

The browser display of the XHTML produced by this call to `table` is shown in Figure 9.6.



Sales Figures

Salesperson	Mon	Tues	Wed	Thu	Fri
Mary	2	4	6	8	10
Freddie	1	3	5	7	9
Spot	100	140	200	350	0

Figure 9.6 A table produced with `table`

Rows can have horizontal and vertical alignment attributes, specified by `align` and `valign`, respectively. These can be specified as the first parameter in the call to `Tr`.

Now we can get back to the development of the second survey-handling CGI program, `conelec2.cgi`. The basic tasks of `conelec2.cgi` are as follows:

1. Open the file and read the lines of the file into an array of strings.
2. Split the first four rows (responses from females) into arrays of votes for the four age groups.
3. Unshift row titles into the vote rows (making them the first elements).
4. Create the column titles row with `th` and put its address in an array.
5. Use `td` on each row of votes.
6. Push the addresses of the rows of votes onto the row address array.
7. Create the table using `Tr` on the array of row addresses.
8. Repeat steps 2 to 7 for the last four rows of data (responses from males).

Because the rows of the two result tables are nearly the same for the two tables, they are built with a subprogram, `make_rows`. This function takes one parameter, which is `0` for the female results table and `4` for the male results table. `make_rows` builds the table rows in the `@rows` array, which is globally accessed by the function and the calling program. The following is the CGI program to produce the vote totals tables:

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

# conelec2.cgi - display the survey results

#>>> make_rows - a subprogram to make the rows of an output table
sub make_rows {
    my $index = $_[0];

#>>> Split the input lines for females into age arrays
    @age1 = split(/ /, $vote_data[$index]);
```

```
    @age2 = split(/ /, $vote_data[$index + 1]);
    @age3 = split(/ /, $vote_data[$index + 2]);
    @age4 = split(/ /, $vote_data[$index + 3]);

#>>> Add the row titles to the age arrays
    unshift(@age1, "10-25");
    unshift(@age2, "26-40");
    unshift(@age3, "41-60");
    unshift(@age4, "Over 60");

#>>> Create the column titles in HTML by giving their address to the
#>>> th function and storing the return value in the @rows array
    @rows = th(\@col_titles);

#>>> Now create the data rows with the td function
#>>>  and add them to the row addresses array
   push(@rows, td(\@age1), td(\@age2), td(\@age3), td(\@age4));
}  #>>> end of the make_rows subprogram

#>>> error - a function to produce an error message for the client
#>>>         and exit in case of open errors
sub error {
    print start_html;
    print "Error - input/output error in conelec2.pl <br/>";
    print end_html;
    exit(1);
}  #>>> end of the error subprogram

use CGI qw(:standard);

#>>> Make the column titles array
@col_titles = ("Age Group", "Conventional TV", "HDTV", "VCR",
               "CD player", "MiniCD player/recorder", "DVD player",
               "Other");
print header;

#>>> Open and read the survey data file
open(SURVDAT, "<survdat.dat") or error();
@vote_data = <SURVDAT>;

#>>> Create the beginning of the result Web page
print start_html("Survey Results");
print h2("Results of the Consumer Electronics Purchasing Survey");
print "<br />";

#>>> Create the rows of the female survey results table
```

```
make_rows(0);

#>>> Create the table for the female survey results
#>>>  The address of the array of row addresses is passed to Tr
print table({-border => "border"},
            caption(h3("Survey Data for Females")),
            Tr(\@rows)
           );

#>>> Create the rows for the male results table
make_rows(4);

#>>> Create the table for the male survey results
#>>>  The address of the array of row addresses is passed to Tr
print "<br /><br />";
print table({-border => "border"},
            caption(h3("Survey Data for Males")),
            Tr(\@rows)
           );

print end_html;
```

Figure 9.7 shows the display of the table of voting results constructed by `conelec2.cgi` after some voting has occurred.

Results of the Consumer Electronics Purchasing Survey

Survey Data for Females

Age Group	Conventional TV	HDTV	VCR	CD player	MiniCD player/recorder	DVD player	Other
10-25	3	0	0	2	0	0	1
26-40	0	2	0	0	0	0	0
41-60	0	0	2	0	0	0	0
Over 60	0	0	0	0	0	0	0

Survey Data for Males

Age Group	Conventional TV	HDTV	VCR	CD player	MiniCD player/recorder	DVD player	Other
10-25	1	3	0	0	0	0	0
26-40	0	0	0	0	1	0	0
41-60	0	0	0	0	2	3	0
Over 60	0	0	0	2	0	0	0

Figure 9.7 Survey results page

Debugging a CGI program can be difficult. In many cases, errors are reported to the user as server errors, with the error number 500, which provides no clue as to what happened. The best course of action in such cases is to find out from the system administrator the location of the server error log. On UNIX systems the file is often named `error_log`; on Windows and OS/2 systems, it is often `error.log`. Once you determine the location and name of the server error log file, examine the most recent errors, and find the error message that is yours (on UNIX systems, this is done with the `tail` command). The error log information should help determine the problem.

9.6 Cookies

A *session* is the time span during which a browser interacts with a particular server. A session begins when a browser becomes connected to a particular server. It ends when the browser ceases to be connected to that server because either it becomes connected to a different server or it is terminated. The HTTP protocol is essentially stateless—it includes no means to store information about a session that is available to a subsequent session. However, there are a number of different reasons why it is useful for the server to be capable of connecting a request made during a session to the other requests made by the same client during that session, as well as previous and subsequent sessions. Many Web sites now create profiles of clients by remembering which parts of the site are perused. Later sessions can use such profiles to target advertising to the client according to the client's past interests. Also, if the server recognizes a request as being from a client who has made an earlier request from the same site, it is possible to present a customized interface to that client. These situations require that information about clients is accumulated and stored.

Cookies provide a general approach to storing information about sessions on the browser system itself. The server is given this information when the browser makes subsequent requests for Web resources from the server. Cookies allow the server to present a customized interface to the client. They also allow the server to connect requests from a particular client to previous requests, thereby connecting sequences of requests into a session.

A *cookie* is a small object of information consisting of a name and a textual value. A cookie is created by some software system on the server, such as a CGI program. Every HTTP communication between a browser and a server includes a header, which stores information about the message. A message from a browser to a server is a request; a message from a server to a browser is a response. The header part of an HTTP communication can include cookies. So, every request sent from a browser to a server, and every response from a server to a browser, can include one or more cookies.

At the time it is created, a cookie is assigned a lifetime. When the time a cookie has existed reaches its associated lifetime, the cookie is deleted from the browser's host machine. Every browser request includes all of the cookies its host machine has stored that are associated with the Web server to which the

request is directed. Only the server that created a cookie can ever receive the cookie from the browser, so a particular cookie is information that is exchanged exclusively between one specific browser and one specific server. Because cookies are stored as text, the browser user can view, alter, or delete them at any time.

Because cookies allow servers to record browser activities, they are considered by some to be a privacy concern. Accordingly, browsers allow the client to change the browser setting to refuse to accept cookies from servers. This is clearly a drawback of using cookies—the clients that reject them render them useless.

The `CGI.pm` module includes support for cookies in Perl, primarily through the `cookie` function. The `cookie` function serves both to create cookies and to retrieve existing cookies from the HTTP header of a request. The form of a call to `cookie` to create a cookie is as follows:

```
cookie(-name => a_cookie_name,
       -value => a_value,
       -expires => a_time_value)
```

The `cookie` function can take three more parameters, but they are not discussed here. The cookie name can be any string. The value can be any scalar value, including references to arrays and hashes. These last two allow the creation of multiple cookies with one call. The `expires` value, which specifies the lifetime of the cookie, can be expressed in many different units. For example, `+3d` specifies three days. The other units are `s` for seconds, `m` for minutes, `h` for hours, `M` for months, `y` for years, and `now` for right now. A negative value for `expires` effectively kills the cookie.

A cookie must be placed in the HTTP header at the time the header is created. With `CGI.pm`, this means when the `header` function is called. This is done by passing the cookie as a parameter to `header`, as in the following example:

```
header(-cookie => $my_cookie);
```

When the `cookie` function is called with no parameters, it returns a hash of all of the cookies in the HTTP header of the current request. To retrieve the value of one cookie, the `cookie` function is called with the name of the cookie. For example, the following call:

```
$age = cookie('age');
```

gets the value of the cookie named `age`.

To display all of the cookies, both names and values, in a CGI program, we could use the following:

```
print "Cookie Name \t Cookie Value <br  />";
foreach $name (keys cookie()) {
    print "$name \t cookie($name) <br />";
}
```

Suppose we want to provide a greeting to all visitors, including a message giving the day of the week, month, and day of the month of their last visit. To do

this, we could use a cookie to record the time and date of each visit. For the first visit by a client, a message could provide a first-time greeting.

For our program, we need to get the current date. The Perl `time` function returns the current time in seconds since January 1, 1970. The `localtime` function is more useful for our problem. It calls `time` and converts the number of seconds into nine values. For example, consider the following call to `localtime`:

```
($sec, $min, $hour, $mday, $mon, $year, $wday, $yday,
 $isdst) = localtime;
```

This statement sets the nine values for the current time and date, corrected for the time zone in which the Web server computer is installed. The first three scalars in the list have obvious meanings. The others have the following meanings: `$mday` is the day of the month; `$mon` is the month, coded as `0` to `11`; `$year` is the number of years since 1900; `$wday` is the day of the week, coded as `0` to `6`, where `0` is Sunday; `$yday` is the day of the year; and `$isdst` is a Boolean that specifies whether the given time is in daylight savings time.

The following example displays all of the nine values returned by `localtime`:

```
# time_date.pl
#  Input: None
# Output: The nine values returned by localtime
#

($sec, $min, $hour, $mday, $mon, $year, $wday, $yday,
  $isdst) = localtime;

print "\$sec = $sec\n";
print "\$min = $min\n";
print "\$hour = $hour\n";
print "\$mday = $mday\n";
print "\$mon = $mon\n";
print "\$year = $year\n";
print "\$wday = $wday\n";
print "\$yday = $yday\n";
print "\$isdst = $isdst\n";
```

The output of `time_date.pl` is as follows:

```
$sec = 43
$min = 20
$hour = 10
$mday = 19
$mon = 2
```

```
$year = 105
$wday = 6
$yday = 77
$isdst = 0
```

For days of the week and month, we often want the names rather than numbers. This conversion is easy in Perl. The names can be put in a list, which can be subscripted by the value returned from `localtime`. For example, the following statement sets `$day_of_week` to the name of the current day of the week:

```
$day_of_week = (qw(Sunday Monday Tuesday Wednesday
                   Thursday Friday Saturday))[(localtime[6]];
```

The subscript 6 is that of the day of the week (`$wday`).

The CGI program for our problem of creating a greeting for visitors with the time and date will do the following:

1. Get the cookie named `last_time`.
2. Get the current day of the week, month, and day of the month and put them in a cookie named `last_time`.
3. Put the cookie in the header of the return document.
4. If there was no existing cookie, produce a welcome message for the first-time visitor.
5. If there was a cookie, produce a welcome message that includes the previous day of the week, month, and day of the month.

The following is the Perl program to implement these actions:

```
#!/usr/bin/perl
# day_cookie.pl
#  A CGI-Perl program to use a cookie to remember the
#   day of the last login from a user and display it when run

use CGI ":standard";

#>>> Get the existing day cookie, if there was one
@last_day = cookie('last_time');

#>>> Get the current date and make the new cookie
$day_of_week = (qw(Sunday Monday Tuesday Wednesday Thursday
                   Friday Saturday)) [(localtime)[6]];
$month = (qw(January February March April May June July
             August September October November December))
          [(localtime)[4]];
$day_of_month = (localtime)[3];
@day_stuff = ($day_of_week, $day_of_month, $month);
```

```
$day_cookie = cookie(-name => 'last_time',
                     -value => \@day_stuff,
                     -expires => '+5d');

#>>> Produce the return document
#>>> First, put the cookie in the new header
print header(-cookie => $day_cookie);
print start_html('This is day_cookie.pl');

#>>> If there was no day cookie, this is the first visit
if (scalar(@last_day) == 0) {
    print "Welcome to you on your first visit to our site <br />";
}

#>>> Otherwise, welcome the user back and give the date of the
#>>>  last visit
else {
    ($day_of_week, $day_of_month, $month) = @last_day;
    print "Welcome back! <br /> ",
          "Your last visit was on ",
          "$day_of_week, $month $day_of_month <br />";
}
print end_html;
```

Figure 9.8 shows a browser display of a document returned by `day_cookie.pl`.

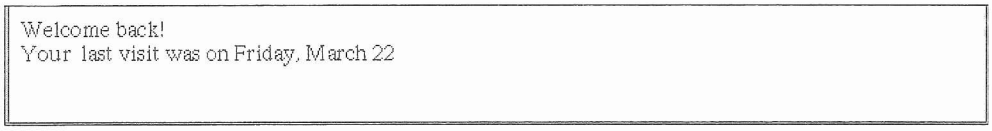


Figure 9.8 A document returned by `day_cookie.pl`

Summary

CGI is the interface between an XHTML document being displayed by a browser and a program that resides on the server. An XHTML document can specify a call to a CGI program with an anchor tag that includes the Internet address of the program. CGI programs can also be called as a side effect of a *Submit* button being clicked. A CGI program communicates with the XHTML document that called it by sending an XHTML document back to the browser

through the server. The XHTML tags in this document, as well as their content, are created by the CGI program through standard output.

XHTML forms are sections of documents that contain widgets, which are used to collect input from the user. The data specified in a form can be sent to the CGI program in either of two methods, `get` or `post`. The `get` method is fine for forms with relatively few widgets. It is also good for sending parameters to a CGI program when a form is not involved. The `post` method is more general because it has no limitation on the number of widgets included in the form. The coded data from a form is called a query string. With `get`, the query string is attached to the URL of the CGI program.

The values specified in a form are together called form data. When the *Submit* button is clicked, the query string, which is an encoded version of the form data, is created from the form data and sent to the server. Each widget that has a value is included in the query string. The format of each of these is a name/value pair, separated by an equal sign. These assignments are separated by ampersands. All special characters are coded by using a percent sign followed by the ASCII code for the character, in hexadecimal.

The `CGI.pm` module provides convenient aids to writing CGI programs in Perl. The shortcut functions of `CGI.pm` produce the tags after which they are named. Attribute values are passed to functions in `CGI.pm` using the form of the elements in a hash literal. The attribute names are preceded by minus signs. Parameters to functions follow the attribute values, if there are any. Tags and their attributes distribute over a list parameter. This list parameter must actually be the address of a list or array. Tables are built with `table`, table rows are built with `Tr`, table headings are built with `th`, and table data is created with `td`. `Tr`, `th`, and `td` allow references as parameters. The `header` function produces the first two lines of the return XHTML document. The `start_html` function produces the `<head>`, `<title>`, `</title>`, `</head>`, and `<body>` tags. Its parameter is used for the content of `<title>`. The `param` function takes a name as its parameter. It returns the value from the query string of that name. The `end_html` function produces the closing tags for `<body>` and `<html>`.

Cookies are small pieces of textual information that are exchanged between Web servers and browsers. They originate from server-based programs but are stored on browser systems. They are used to store information about a client between sessions the browser has with specific servers.

Review Questions

9.1 What are three categories of operations that are essential in Web documents but that cannot be done with XHTML?

9.2 On what system, the client or the server, do CGI programs reside when they are executed?

9.3 What forms are legal for the response from a CGI program?

9.4 What is the most common way for a client to provide information to the server?

9.5 What are the actions of the *Submit* button in a form?

9.6 How is the CGI program that processes the data provided by a form specified in the form?

9.7 Must a CGI program that processes a form reside on the server that provided the form to the client?

9.8 What part of the HTTP header is always provided as part of the response of a CGI program?

9.9 What is form data?

9.10 What is a query string?

9.11 How is a query string transmitted to the server with the `get` method?

9.12 How is a query string transmitted to the server with the `post` method?

9.13 What is the format of a query string that has multiple widget data values?

9.14 Why are special characters coded in query strings?

9.15 What is the purpose of the shortcuts in `CGI.pm`?

9.16 If both content and attribute values are passed to a shortcut, what is the format of these parameters?

9.17 Explain how tags and their attributes are distributed over the parameters to a shortcut function.

9.18 How are arrays passed to shortcut functions?

9.19 Why should a file to be read or written by a CGI program be locked against multiple simultaneous operations?

9.20 Where are cookies stored?

9.21 What is the form of the value of a cookie?

Exercises

9.1 Write an XHTML document that contains an anchor tag that calls a CGI program. Write the called CGI program, which returns a randomly chosen greeting from a list of five different greetings. The greetings must be stored as constant strings in the program. A random number between 0 and 4 can be computed with these lines:

```
srand;  # Sets the seed for rand
$number = int(rand 4);  # Computes a random integer 0-4
```

9.2 Modify the CGI program for Exercise 9.1 to count the number of visitors and display that number for each visitor.

9.3 Write an XHTML document to create a form with the following capabilities:

a. A text widget to collect the user's name

b. Four checkboxes, one each for the following items:

i. Four 100-watt light bulbs for $2.39

ii. Eight 100-watt light bulbs for $4.29

iii. Four 100-watt long-life light bulbs for $3.95

iv. Eight 100-watt long-life light bulbs for $7.49

c. A collection of three radio buttons that are labeled as follows:

i. Visa

ii. MasterCard

iii. Discover

9.4 Write a Perl CGI program that computes the total cost of the ordered light bulbs from Exercise 9.3, after adding 6.2 percent sales tax. The program must inform the buyer of exactly what was ordered, in a table.

9.5 Revise the survey sample CGI program of this chapter to make the table that displays the results of the survey have consumer electronics devices as its rows rather than its columns.

9.6 Revise the survey sample CGI program of this chapter to record the number of votes so far in the data file and display that count every time a vote is submitted or a survey result is requested. Also, change the output table so that its data is a percentage of the total votes for the particular age category.

9.7 Write an XHTML document to create a form that collects favorite popular songs, including the name of the song, the composer, and the performing artist or group. This document must call one CGI program when the form is submitted and another to request a current list of survey results.

9.8 Write a CGI program that collects the data from the form of Exercise 9.7 and writes it to a file.

9.9 Write a CGI program that produces the current results of the survey of Exercise 9.7.

9.10 Write an XHTML document to provide a form that collects names and telephone numbers. The phone numbers must be in the format ddd-ddd-dddd. Write a CGI program that checks the submitted telephone number to be sure it conforms to the required format and then returns a response that indicates whether the number was correct.

9.11 Modify the `day_cookie.pl` program to have it return the number of months, days, hours, and minutes since the last visit by the current client.

9.12 Write a CGI program that collects the name of every visitor (in a form text element). The program must create a cookie to save the visitor's name and include a brief personalized greeting to every repeat visitor.