Part Two

Understanding the Nuts and Bolts

CHAPTER ELEVEN

THE MAIN SCRIPT

Defining the Operating Environment

Like any Perl script, the first line of the main script lets the Web server know to execute this script using the Perl interpreter and where it is located. As discussed in Chapter 1, you may have to modify this line to reflect the location of the Perl interpreter on your local server.

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

Next, Perl is told to bypass its own buffer so that the information generated by this script will be sent immediately to the browser.

```
$| = 1;
```

Then, the HTTP header is sent to the browser. This is done early for two reasons. First, it will be easier to debug the script while making modifications or customizing because we will be able to see exactly what the script is doing and avoid the unhelpful "500 Server Error" warnings.

Secondly, the HTTP header is sent out early so that the browser will not "time out" in case the script takes a long time to complete its work.

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

Next, the script executes a few subroutines that define the environment in which the script will operate. First, the **web_store.setup** file is read in order to assign global setup variables. Notice that, in the distribution, we have six setup files by default. We will use **web_store.setup.frames.javascript** as our basic example.

Secondary supporting libraries are loaded using **require_supporting_libraries**, which is passed the current filename as well as the current line number. This subroutine uses these values to generate useful error messages in case the script is unable to read in the files requested.

- **web_store.setup.*** defines many global variables for this script relative to the local server and installation.
- **$sc_cgi_lib_path** is the location of **cgi-lib.pl**, used to parse incoming form data.
- **$sc_html_setup_file_path** is the location of **web_store_html_lib.pl**, used to define various customizable HTML interface headers, footers, and pages.
- **$sc_mail_lib_path** is the location of **mail-lib.pl**, used to mail nonencrypted mail to the administrator about use of the script.

```
&require_supporting_libraries (__FILE__, __LINE__,
      "./Library/web_store.setup.db");
&require_supporting_libraries (__FILE__, __LINE__,
                     "$sc_cgi_lib_path",
                     "$sc_html_setup_file_path",
                     "$sc_mail_lib_path");
```

The incoming form data is then read and parsed using **&read_and_parse_form_data** which simply uses the ReadParse subroutine in **cgi-lib.pl** to parse the incoming form data into the associative array, **%form_data**:

```
&read_and_parse_form_data;
```

This subroutine is discussed in greater detail later. Once we have parsed the incoming form data, we can assign the values of administrative variables to scalars, local to this script.

- **$page** will contain the path location of any pages that this script is required to display. This may be the store frontpage, order form, or any number of product or category pages used for store navigation.
- **$search_request** is the value of the form button used when a customer submits search terms used to generate a dynamic custom product page.
- **$cart_id** is the i.d. number of the customer's unique cart containing all of the items they have ordered so far. The specifics of cart generation and maintenance are covered in greater depth in the next section.
- **$sc_cart_path** is the actual path of the shopping cart combining both **$sc_user_carts_directory_path** defined in **web_store.setup** and **$cart_id** that may be coming in as form data.

All four of these variables are crucial state variables which must be passed as form data from every instance of this script to the next.

```
$page = $form_data{'page'};
$search_request = $form_data{'search_request_button'};
$cart_id = $form_data{'cart_id'};
$sc_cart_path = "$sc_user_carts_directory_path/$cart_id.cart";
```

The incoming form data is then submitted to some security checks. **error_check_form_data** is a subroutine that checks the just-parsed incoming form data to make sure that the script is only being used to display proper pages (typically **.html**, **.html**, or **.htm**).

This is an important security precaution. Later in this script, we are going to use a variable called **page** to communicate the page in our store we want to display to the customer.

The danger lies in the fact that a customer might "fake" a request to the script by editing the **page** variable in the HTML or in the encoded URL. For example, they might manually reassign a page from, say, **Vowels.html** to "../../../etc/passwd"! As you can imagine, this could end up displaying your password file to the browser window if you are running a UNIX-based Web server. Thus, we need to make sure that only appropriate files can be displayed by the store. Typically, these are only HTML pages you will have designed. The subroutine itself will be discussed in greater detail later.

```
&error_check_form_data;
```

Finally, the script assigns a unique cart i.d. if none has been submitted as form data. Every customer who is using the application must be assigned a unique cart that will contain their specific shopping list. Obviously, customers will not share carts.

These carts are actually short, flat file, text databases stored by default in the **User_carts** subdirectory in the format **[SOME RANDOM NUMBER].cart**. These files contain information about which items the customer has ordered and how many of each item they ordered.

Once customers enters the store, they are assigned their own unique cart. For the rest of their stay, the script will make sure that it matches customers with their carts, no matter which page they go to.

It does this matching by continually passing along the location of the cart (**$cart_id**) as either hidden form data or URL encoded information, depending on whether the customer uses a Submit button or a hyperlink to navigate through the store. As long as the customer follows the path provided by the application, she will never lose her cart.

Thus, before anything else, the script must check to see if the customer has already received a unique shopping cart. If so, it will be coming in as form data and have been just assigned to **$cart_id**. If the script has not received a shopping cart i.d. number as form data (**$cart_id eq ""**), it means that the customer has not yet received a unique shopping cart.

In such a case, the script must assign a new cart i.d. number. However, as a matter of good housekeeping, the script will first delete old carts that have been abandoned by using the **delete_old_carts** subroutine documented later in this script. Then, it will assign customers their own fresh new cart using **assign_a_unique_shopping_cart_id**, which is also discussed later.

```
if ($cart_id eq "")
  {
  &delete_old_carts;
  &assign_a_unique_shopping_cart_id;
  }
```

Decoding the Main Script Logic

Now that the script has created the entire environment in which it must operate, it is time to provide the logic for it to determine what it should do. Figure 11.1 shows the logic of this determination.

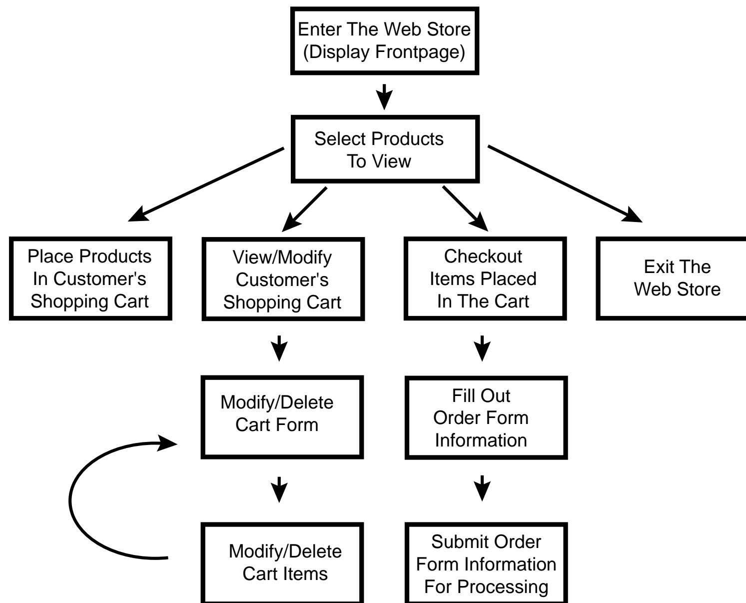


Figure 11.1 The Main Logic of web_store.cgi.

The logic is broken down into a series of "if" tests. Specifically, the script checks the values of incoming administrative form variables (mainly supplied from the Submit buttons on dynamically generated HTML forms) and performs its operations depending on whether those administrative variables have values associated with them or not. The basic format for such an "if" test follows the syntax:

```
if (the value of some submit button is not equal to
    nothing)
  {
  process that type of request;
  exit;
  }
```

For example, consider the first case in which the customer has clicked on the **Add to Cart** Submit button denoted with the NAME value of the **add_to_cart_button**.

```
elsif ($form_data{'add_to_cart_button'} ne "")
 {
 &add_to_the_cart;
 exit;
 }
```

Because the Submit button will have some caption value like **Add this item to my cart**, when the script reaches this line, it will answer true to the test. Since the customer can only click on one Submit button at a time, we can be assured that only one operation will answer true because only one submit NAME variable will have a value.

The beauty of using the not equal (**ne**) test is that, regardless of what the Submit button actually says (it might say, "Add a wiener dog to the chopping block"), the if test will still be satisfied if the customer has clicked the button, since whatever the VALUE is, it will certainly not be equal to "nothing". Of course, this assumes that you do not rename the NAME argument of the Submit buttons. If you do so, you must harmonize the variable you use on the input forms, with the variables used here to test. To repeat, the NAME of the buttons cannot change, but you can change the caption VALUE of them.

Similarly, if you wish to have graphical Submit buttons instead of the ugly default buttons supplied by the browser, you will have to modify the if tests so that they follow the standard image map test:

```
if ($form_data{'some_button.x'} ne "")
 {
 &do some subroutine;
 exit;
 }

where the HTML code looks like the following:
<INPUT TYPE = "image" NAME = "some_button"
       SRC = "Images/button.gif" BORDER = "0">
```

Thus, if the button actually has an X-dimension value (any x-dimension value at all), it means that the button had been clicked.

Finally, note that every if test is concluded with an exit statement. This is because once the script has executed the routine specified in the Submit button, it is through with its work and should exit immediately and wait for the next request.

Get used to the idea that this script is self-referencing. The application itself contains many miniroutines that all refer back this script. Every instance of the script executes maybe only 20 percent of the routines in the whole file, but in the lifetime of the application, most, if not all, of the routines are executed. Next, let

us look at each of the routines that this application must execute. There are ten cases that might apply to a customer's request.

1. **Adding an Item to the Shopping Cart** One request that the script may have to handle is that of adding an item to a shopping cart. Once the customer has decided to purchase an item, he or she will have added a quantity to the text box and clicked on the **add this item** Submit button. So we must be prepared to add items to the customer's cart. Additions are handled with the **add_to_the_cart** subroutine to be discussed later.
2. **Displaying the Customer's Cart with Cart Manipulation Options** On the other hand, the customer may have already been adding items, realized she went over budget, and decided to reduce the quantities of some of the items she had chosen or even delete them altogether from her cart.

   We need to send her an HTML form with which she can choose whether to delete or modify, as well as send her a table depicting the current contents of the shopping cart. This is all done using the **display_cart_contents** subroutine at the end of this file.
3. **Displaying the Change Quantity Form** Yet another function that this script may be asked to perform is modifying the quantities of some of the items in the customer's cart. If the customer has asked to make a quantity modification, the script must give her a form so that she can specify the changes she wants made.

   The form is fairly simple. We will use the same basic table presentation that we used in the **display_cart_contents** subroutine, except that we will add another column of text input fields to submit a new quantity for every row in the cart. These text input fields, however, will use as their NAME argument, the unique cart row number for every row. Consider the following cell definition:

   ```
   <TD><INPUT TYPE = "text" NAME = "219" SIZE ="3"></TD>
   ```

   When the customer submits a quantity change, they will be submitting a **cart_row_number "219"** associated with a quantity value (the value

submitted in the text field). We will use the cart row number to figure out exactly which item in the cart should be modified.

4. **Changing the Quantity of Items in the Cart** Once the customer has typed in some quantity changes and submitted the information back to this script, we must make the modifications to the database. This is done with the **modify_quantity_of_items_in_cart** subroutine discussed below.

5. **Displaying the Delete Item Form** Instead, perhaps the customer asked to delete an item rather than modify the quantity. If this is the case, the script must display a form very similar to the one for modification. The only difference is that we will use checkboxes for each item instead of text boxes because in the case of deletion, the customer need only select items to delete rather than specify a specific quantity.

   As in the case of modification, the script associates the NAME argument of the checkboxes with the cart row number of the item they represent. Thus, the syntax will resemble the following:

```
<TD><INPUT TYPE = "checkbox" NAME = "220"></TD>
```

   where **"220"** is the cart row number of some element that can be deleted. We will handle the display of the delete item form using the **output_delete_item_form** subroutine to be discussed later.

6. **Deleting Items from the Cart** Once the customer submits some items to delete, the script must also be able to delete them from the cart. This is done with the **delete_from_cart** subroutine to be discussed later.

7. **Displaying the Order Form** Further, the script must be able to display the order form for the customer if that is what she wants to see. The script uses the **display_page** subroutine, which will be discussed later, to display a predesigned order form.

8. **Submitting the Order** Once the customer fills out the order form, she may submit the order for final processing. Final processing involves calculating order logic (shipping method, tax rates, discounts, etc.), sending the order to the order processing administrator, and letting the customer know that all was completed successfully. All orders are processed by the **process_order_form** subroutine, which is designed to handle all of these chores.

There is one catch to order processing: secure servers. Many stores have secure server functionality in which specific directories are designed to handle encrypted communication between server and browser. In this case, the CGI script handling the order processing must be physically located inside the secure directory.

If this is your situation, then you must make a mirror copy of the application directories and place them all inside the secure area. Then, you will set **$sc_order_script_url** in the Setup file equal to the secured mirror of the script. Then, the script will dynamically reference the secured location instead of the insecure location for order processing. In actuality, only the order-processing routine will be executed in the Secure directory, but we copy the whole script there for simplicity's sake. If you are not running a secure server, you may just set **$sc_order_script_url** equal to **web_store.cgi** and continue the regular self-referencing behavior. This is discussed in greater detail in Chapter 8.

9. **Displaying Products, Categories, and Miscellaneous Pages** If the script is getting in a value for page or for product, it means that it is being asked to navigate through the store. The page variable is used to locate a pre-designed HTML page that the store should display to the customer. In the case of an HTML store, the page value will be used to point to both pages with products as well as pages with lists of products. Also in the case of the HTML store, there will be no need for the product variable since the product variable is specific to the Database store, which must be able to interpret between a list-of-products–type page and an actual product page. This is because when the database version creates a page to view, it needs to generate the page on the fly. Thus, the database searches for the product in the database. If the script needs to display a list-of-products–type page with sublinks to actual product pages within a similar group, it should not go to the database. Instead, it actually needs to display a list page just as the HTML store would do.

   The product value will be used by the database shopping cart to pull out the list of products that the customer is interested in seeing. Think of this as a type of hard-coded search of the database where the administrator may define a category to search for in the URL string that requests a product page view. Consider the following hyperlinks as examples.

```
web_store.cgi?page=Letters.html&cart_id=98.123
web_store.cgi?page=Numbers.html&cart_id=8708496.3559
web_store.cgi?product=Numbers&cart_id=2196655.5107
```

The first case could be used in either the HTML- or Database-based version. The script would display the HTML page **Letters.html** which would be a list of products type of page. In our distribution example page, **Letters.html** contains links to both **Vowels.html** and **Consonants.html**.

The second URL would be used for an HTML-based store. It would cause this script to display the pre-designed product page, **Numbers.html**.

Finally, the last line would be used for a Database-based cart system and would cause this script to search through the database for all items with **Numbers** in the category field (by default, this is the second field in **data.file**)

Thus, there are two ways that products can be displayed with this script. The first way is for the store administrator to create a delimited data file with all the data to be displayed incorporated in database rows. The contents of these rows will be displayed according to the format defined in the **$sc_product_page_row** variable in **web_store_html_lib.pl**. The second way is for the administrator to create HTML pages directly with the same data already incorporated into some desired interface.

The administrator specifies which method she will use by setting the variable **$sc_use_html_product_pages** in the Setup file. If this variable is set to "yes", it means that the script should simply output a predesigned HTML product page. Anything else, and it will expect a database.

The **display_products_for_sale** subroutine (to be discussed later) does just that. However, there is one catch to the presentation of an HTML page. If the customer is doing a keyword search, we will have to generate a list of pages on which their keyword was found using the **html search** subroutine located in **web_store_html_search.pl**.

10. **Display the Front Page** Finally, if all else has failed, it means that we are simply being asked to display the store front page, because no other routines remain to test. To display the store front page, we will access the **output_frontpage** subroutine discussed later.

Now that we have defined the cases, let us go through them one by one as code.

```
if ($form_data{'add_to_cart_button'} ne "")
  {
  &add_to_the_cart;
  exit;
  }

elsif ($form_data{'modify_cart_button'} ne "")
  {
  &display_cart_contents;
  exit;
  }

elsif ($form_data{'change_quantity_button'} ne "")
  {
  &output_modify_quantity_form;
  exit;
  }

elsif ($form_data{'submit_change_quantity_button'} ne "")
  {
  &modify_quantity_of_items_in_cart;
  exit;
  }

elsif ($form_data{'delete_item_button'} ne "")
  {
  &output_delete_item_form;
  exit;
  }

elsif ($form_data{'submit_deletion_button'} ne "")
  {
  &delete_from_cart;
  exit;
  }

elsif ($form_data{'order_form_button'} ne "")
  {
  &require_supporting_libraries (__FILE__, __LINE__,
                              "$sc_order_lib_path");
  &display_order_form;
  exit;
  }
```

```
elsif ($form_data{'submit_order_form_button'} ne "")
  {
  &require_supporting_libraries (__FILE__, __LINE__,
                              "$sc_order_lib_path");
  &process_order_form;
  exit;
  }

elsif (($page ne "" || $form_data{'product'} ne "") &&
      ($form_data{'return_to_frontpage_button'} eq ""))
  {
  &display_products_for_sale;
  exit;
  }

else
  {
  &output_frontpage;
  exit;
  }
```

Well, that's it. That is just the end of the main body of logic. From here on, we will define the logic of the subroutines called with the "if" tests above.

Load Supporting Libraries

The subroutine **require_supporting_libraries** is used to load some of the supporting files that this script will take advantage of. The subroutine takes a list of arguments beginning with the current filename, the current line number, and the list of files that must be loaded using the following syntax:

```
&require_supporting_libraries (__FILE__, __LINE__,
                         "file1", "file2",
                         "file3"...);
```

__FILE__ and **__LINE__** are special Perl variables that contain the current filename and line number respectively. We will continually use these two variables throughout the rest of this script in order to generate useful error messages.

```
sub require_supporting_libraries
  {
```

The incoming file and line arguments are split into the local variables **$file** and **$line**, while the file list is assigned to the local list array **@require_files**. **$require_file**, which will just be a temporary holder variable for our processing, is also defined as a local variable.

```
local ($file, $line, @require_files) = @_;
local ($require_file);
```

Next, the script checks to see if every file in the **@require_files** list array exists (**-e**) and can be read (**-r**). If so, the script goes ahead and loads it with the require command.

```
 foreach $require_file (@require_files)
    {
    if (-e "$require_file" && -r "$require_file")
      {
      require "$require_file";
      }
```

If not, the scripts sends back an error message that will help the administrator isolate the problem with the script.

```
 else
      {
      print "I am sorry but I was unable to require
             $require_file at line $line in $file.
             Would you please make sure that you have
             the path correct and that the permissions
             are set so that I have read access?  Thank
             you.";
      exit;
      }
    } # End of foreach $require_file (@require_files)
  } # End of sub require_supporting_libraries
```

Read and Parse Form Data

read_and_parse_form_data is a short subroutine responsible for calling the **ReadParse** subroutine in **cgi-lib.pl** to parse the incoming form data. The script also tells **cgi-lib** to prepare that information in the associative array named **%form_data**, which we will be able to use for the rest of this script.

read_and_parse_form_data takes no arguments and is called with the following syntax:

```
&read_and_parse_form_data;
```

The code reads as follows:

```
sub read_and_parse_form_data
  {
  &ReadParse(*form_data);
  }
```

Error Check Form Data

error_check_form_data is responsible for checking to make sure that only authorized pages are viewable using this application. It takes no arguments and is called with the following syntax:

```
&error_check_form_data;
```

The routine simply checks to make sure that the page variable extension is not one defined in the Setup file as an appropriate extension such as **.html** or **.htm**, or that there is no customer-definable page being requested for display. If, however, these conditions exist, the script will send a warning to the customer, append the error log, and exit.

@acceptable_file_extensions_to_display is an array of acceptable file extensions defined in the Setup file. To be more—or less—restrictive, just modify this list. Specifically, for each extension defined in the Setup file, if the value of the page variable coming in from the form (**$page**) is like the extension (**=~ /$file_extension/**) or (||) there is no value for page (**eq ""**), we will set **$valid_extension** equal to "yes."

```
sub error_check_form_data
  {
  foreach $file_extension
          (@acceptable_file_extensions_to_display)
    {
    if ($page =~ /$file_extension/ || $page eq "")
```

```
      {
      $valid_extension = "yes";
      }
    }
```

Next, the script checks to see if **$valid_extension** has been set to “yes.”

If the value for page satisfied any of the extensions in **@acceptable_file_extensions_to_display**, the script will have set **$valid_extension** equal to “yes” in the previous routine. If the value is set to “yes,” the subroutine will go on with it’s work. Otherwise it will exit with a warning and write to the error log if appropriate.

Notice that we pass three parameters to the **update_error_log** subroutine. The subroutine gets a warning, the name of the file, and the line number of the error.

$sc_page_load_security_warning is a variable set in **web_store.setup**. If you want to give a more or less informative error message, you are welcome to change the text there.

```
if ($valid_extension ne "yes")
    {
    print "$sc_page_load_security_warning";
    &update_error_log("PAGE LOAD WARNING",  __FILE__,
                      __LINE__);
    exit;
    }
  }
```

Deleting Old Carts

delete_old_carts is a subroutine used to prune the Carts directory, cleaning out all the old carts after a time interval defined in the Setup file. It takes no arguments and is called with the following syntax:

```
&delete_old_carts;
```

The code is explained below:

```
sub delete_old_carts
  {
```

The subroutine begins by grabbing a listing of all of the customer-created shopping carts in the **User_carts** directory. It then opens (**opendir**) the directory and reads the contents (**readir**) using grep and pattern-matching for **.cart** to grab every file in the **User_carts** directory. Then it closes the directory. If the script has any trouble opening the directory, it will output an error message using the **file_open_error** subroutine to be discussed later. To the subroutine, it will pass the name of the file that had trouble, as well as the current routine in the script having trouble, the filename, and the current line number.

```
 opendir (USER_CARTS, "$sc_user_carts_directory_path")
   || &file_open_error("$sc_user_carts_directory_path",
                          "Delete Old Carts", __FILE__,
                       __LINE__);
  @carts = grep(/\.cart/,readdir(USER_CARTS));
  closedir (USER_CARTS);
```

Then, for every cart in the directory, the script deletes the cart if it is older than half a day. The **-M** file test returns the number of days since the file was last modified. Since the result is in terms of days, if the value is greater than the value of **$sc_number_days_keep_old_carts** set in **web_store.setup**, the file is deleted using the unlink function.

```
foreach $cart (@carts)
    {
    if (-M "$sc_user_carts_directory_path/$cart" >
            $sc_number_days_keep_old_carts)
      {
      unlink("$sc_user_carts_directory_path/$cart");
      }
    }
  } # End of sub delete_old_carts
```

Assigning a Shopping Cart

assign_a_unique_shopping_cart_id is a subroutine used to assign a unique cart i.d. to every new customer. It takes no arguments and is called with the following syntax:

```
&assign_a_unique_shopping_cart_id;
```

The code is explained below:

```
sub assign_a_unique_shopping_cart_id
  {
```

First, the script checks to see if the administrator has asked it to log all new customers. If so, it gets the current date using the **get_date** subroutine discussed later, opens the access log file for appending, and prints all of the environment variable values as well as the current date and time to the access log file.

However, to protect ourselves from multiple, simultaneous writes to the access log, the script uses the **lockfile** routine documented at the end of this file, passing it the name of a temporary lock file to use.

Remember that there may be multiple simultaneous executions of this script because there may be many people shopping all at once. It would not do if one customer was able to overwrite the information of another customer if both accidentally wanted to access the log file at the same exact time.

```
if ($sc_shall_i_log_accesses eq "yes")
      {
      $date = &get_date;
      &get_file_lock("$sc_access_log_path.lockfile");
      open (ACCESS_LOG, ">>$sc_access_log_path");
```

Using the Keys function, the script grabs all the keys of the **%ENV** associative array and assigns them as elements of **@env_keys**. It then creates a new row for the access log, which will be a pipe-delimited list of the date and all the environment variables and their values, using the **.=** operator to continually append fields to the **$new_access** variable.

```
 @env_keys = keys(%ENV);

      $new_access = "$date\|";
      foreach $env_key (@env_keys)
        {
        $new_access .= "$ENV{$env_key}\|";
        }
```

The script then takes off the final pipe, adds the new access to the log file, closes the log file, and removes the lock file.

```
  chop $new_access;
  print ACCESS_LOG "$new_access\n";
  close (ACCESS_LOG);
  &release_file_lock("$sc_access_log_path.lockfile");
  }
```

Now that the new access is recorded, the script assigns the customers their own unique shopping carts. To do so, it generates a random (**rand**) 8-digit (100000000) integer (**int**) and then appends to that string the current process i.d. (**$$**) separated by a period. Additionally, the **srand** function is seeded with the time and the current process i.d. in order to produce a more random random number. **$sc_cart_path** is also defined, now that we have a unique cart i.d. number.

```
srand (time|$$);
$cart_id = int(rand(10000000));
$cart_id .= ".$$";
$sc_cart_path = "$sc_user_carts_directory_path/$cart_id.cart";
```

However, before we can be absolutely sure that we have created a unique cart, the script must check the existing list of carts to make sure that there is not one with the same value.

It does this by checking to see if a cart with the randomly generated i.d. number already exists in the **User_carts** subdirectory. If one does exist (**-e**), the script grabs another random number using the same routine as above and checks again.

Using the **$cart_count** variable, the script executes this algorithm three times. If it does not succeed in finding a unique cart i.d. number, the script assumes that there is something seriously wrong with the randomizing routine and exits, warning the customer on the Web and the administrator using the **update_error_log** subroutine discussed later.

```
    $cart_count = 0;
    while (-e "$sc_cart_path")
      {
      if ($cart_count == 3)
```

```
    {
    print "$sc_randomizer_error_message";
    &update_error_log("COULD NOT CREATE UNIQUE CART
                      ID", __FILE__,
                        __LINE__);
    exit;
    }

  srand (time|$$);
  $cart_id = int(rand(10000000));
  $cart_id .= ".$$";
  $cart_count++;

  } # End of while (-e $sc_cart_path)
```

Now that we have generated a truly unique i.d. number for the new customer's cart, the script may go ahead and create it in the **User_carts** subdirectory. If there is a problem opening the new cart, we will output an error message with the **file_open_error** subroutine discussed later.

```
 open (CART, ">$sc_cart_path") ||
         &file_open_error("$sc_cart_path",
                          "Assign a Shopping Cart",
  __FILE__, __LINE__);
  }
```

Displaying the Frontpage

output_frontpage is used to display the frontpage of the store. It takes no arguments and is accessed with the following syntax:

```
&output_frontpage;
```

The subroutine simply utilizes the **display_page** subroutine, which is discussed later, to output the frontpage file, the location of which is defined in **web_store.setup**. **display_page** takes four arguments: the frontpage HTML file path, the routine calling it, the current filename, and the current line number.

```
sub output_frontpage
```

```
  {
  &display_page("$sc_store_front_path",
                "Output Frontpage", __FILE__,
                __LINE__);
  }
```

Adding to the Shopping Cart

The **add_to_the_cart** subroutine is used to add items to the customer's unique cart. It is called with no arguments with the following syntax:

```
&add_to_the_cart;
```

The code is explained below:

```
sub add_to_the_cart
  {
```

The script must first figure out what the customer has ordered. It begins by using the **%form_data** associative array given to it by **cgi-lib.pl**. It takes all of the keys of the **form_data** associative array and drops them into the **@items_ordered** array.

An associative array *key* is like a variable name, whereas an associative array *value* is the value associated with that variable name. The benefit of an associative array is that you can have many of these key/value pairs in one array.

Conveniently enough, you will notice that input fields on HTML forms will have associated NAMES and VALUES corresponding to associative array KEYS and VALUES. Since each of the text boxes in which the customer could enter quantities were associated with the database i.d. number of the item that they accompany (as defined in the **display_page** routine at the end of this script), the HTML should read

```
<INPUT TYPE = "text" NAME = "1234">
```

for the item with database i.d. number **"1234"** and

```
<INPUT TYPE = "text" NAME = "5678">
```

for item **"5678"**.

If the customer orders **"2"** of **"1234"** and **"9"** of **"5678"**, then the incoming data will be a list of **"1234"** and **"5678"** such that **"1234"** is associated with **"2"** in the **%form_data** associative array and **"5678"** is associated with **"9"**. The script uses the keys function to pull out just the keys. Thus, **@items_ordered** would be a list like (1234, 5678, ...).

```
@items_ordered = keys (%form_data);
```

Next the script goes through the list of items ordered one by one.

```
  foreach $item (@items_ordered)
    {
```

However, there are some incoming items that do not need to be processed. Specifically, we do not care about **cart_id**, **page**, **keywords**, **add_to_cart**, or whatever incoming administrative variables exist because these are all values set internally by this script. They will be coming in as form data just like the customer-defined data, and we will need them for other things, just not to fill up the customer's cart. In order to bypass all of these administrative variables, we use a standard method for denoting incoming items. All incoming items are prefixed with the tag **item-**. When the script sees this tag, it knows that it is seeing an item to be added to the cart.

Similarly, items that are actually options information are prefixed with the word **option**. We will also accept those for further processing. And of course, we will not need to worry about any items which have empty values. If the shopper did not enter a quantity, then we will not add it to the cart.

```
    if (($item =~ /^item-/i ||
        $item =~ /^option/i) &&
        $form_data{$item} ne "")
      {
```

Once the script has determined that the current element (**$item**) of **@items_ordered** is indeed a nonadministrator item, it must separate out the items that have been ordered from the options which modify those items. If $item begins with the keyword **option**, which we set specifically in the HTML defining the option, the script will add (**push**) that item to the array called **@options**. The script also removes the **item-** flag since it will not be necessary now that the item has passes the "if" test above.

```
$item =~ s/^item-//i;
if ($item =~ /^option/i)
  {
  push (@options, $item);
  }
```

On the other hand, if it is not an option, the script adds it to the array **@items_ordered_with_options**, but adds both the item and its value as a single array element.

The value will be a quantity and the item will be something like **item-0001|12.98|The letter A** as defined in the HTML file. Once we extract the initial **item-** tag from the string using regular expressions (**$item =~ s/^item-//i;**) and append the quantity, the resulting string would be something like the following:

```
2|0001|12.98|The letter A
```

where **2** is the quantity.

However, the quantity value must satisfy a few conditions before it is considered valid. First, it must be a digit. That is, we do not want the customers trying to enter values like “a”, “-2”, “.5” or “1/2”. They might play havoc on the ordering system. A sneaky customer might even gain a discount because you were not reading the order forms carefully. Second, the script will disallow any zeros (**$form_data{$item} == 0**). In both cases the customer will be sent to the subroutine **bad_order_note** located in **web_store_html_lib.pl**.

```
        else
          {
          if (($form_data{"item-$item"} =~ /\D/) ||
              ($form_data{"item-$item"} == 0))
            {
         &bad_order_note;
            }
          else
            {
            $quantity = $form_data{"item-$item"};
            push (@items_ordered_with_options,
                  "$quantity\|$item\|");
         }
        }
        } # End of if ($item ne "$variable"
    } #End of foreach $item (@items_ordered)
```

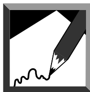
Next, the script goes through the array **@items_ordered_with_options** one item at a time in order to modify any item that has had options applied to it. Recall that we just built the **@options** array with all the options for all the items ordered. Now the script will need to figure out which options in **@options** belong to which items in **@items_ordered_with_options**.

```
foreach $item_ordered_with_options
        (@items_ordered_with_options)
  {
```

First, clear out a few variables that we are going to use for each item. **$options** will be used to keep track of all of the options selected for any given item. **$option_subtotal** will be used to determine the total cost of each option. **$option_grand_total** will be used to calculate the total cost of all ordered options. **$item_grand_total** will be used to calculate the total cost of the item ordered factoring in quantity and options.

```
  $options = "";
  $option_subtotal = "";
  $option_grand_total = "";
  $item_grand_total = "";
```

Then, the script splits out **the $item_ordered_with_options** variable into its fields.


NOTE

We have defined the index location of some important fields in **web_store.setup**. Specifically, the script must know the index of quantity, item i.d. and item price within the array. It will need these values in particular for further calculations.

The script also changes all occurrences of **~qq~** to a double quote (") character, **~gt~** to a greater than sign (**>**), and **~lt~** to a less than sign (**<**). The reason that this must be done is so that any double quote, greater than, or less than characters used in URL strings can be stuffed safely into the cart and passed as part of the NAME argument in the Add Item form. Consider the following item name which must include an image tag.

```
<INPUT TYPE = "text"  NAME = "item-0010|Vowels|15.98|The letter
A|~lt~IMG SRC = ~qq~Html/Images/a.jpg~qq~ ALIGN = ~qq~left~qq~~gt~"
```

Notice that the URL must be edited. If it were not, how would the browser understand how to interpret the form tag? The form tag uses the double quote, greater than, and less than characters in its own processing.

```
$item_ordered_with_options =~ s/~qq~/\"/g;
$item_ordered_with_options =~ s/~gt~/\>/g;
$item_ordered_with_options =~ s/~lt~/\</g;

@cart_row = split (/\|/, $item_ordered_with_options);
$item_quantity = $cart_row[$sc_cart_index_of_quantity];
$item_id_number = $cart_row[$sc_cart_index_of_item_id];
$item_price = $cart_row[$sc_cart_index_of_price];
```

Then, for every option in **@options**, the script splits up each option into its fields. Once it does both splits, the script can compare the name of the item with the name associated with the option. If they are the same, it knows that this is an option meant to enhance this particular item.

```
foreach $option (@options)
  {
  ($option_marker, $option_number,
   $option_item_number) = split
     (/\|/, $option);
```

If the script finds a match, it records the option information contained in the **$option** variable.

```
if ($option_item_number eq "$item_id_number")
 {
```

Since it must apply this option to this item, the script splits out the value associated with the option and appends it to **$options**. Once it has gone through all of the options, using **.=**, the script will have one big string containing all the options so that it can print them out. Note that in the form on which the customer chooses options, each option is denoted with the form

```
NAME = "A|B|C" VALUE = "D|E"
```

where

- "A" is the option marker "option."
- "B" is the option number. You might have multiple options all modifying the same item. Option number identifies each option uniquely .
- "C" is the option item number, the unique item i.d. number that the option modifies.
- "D" is the option name—the descriptive name of the option.
- "E" is the option price.

For example, consider this option from the default **Vowels.html** file that modifies item number **0001**:

```
<INPUT TYPE = "radio" NAME = "option|2|0001"
       VALUE = "Red|0.00" CHECKED>Red<BR>
```

This is the second option modifying item number **0001**. When displayed in the display cart screen, it will read **Red 0.00**, and will not affect the cost of the item.

```
($option_name, $option_price) =
         split (/\|/,$form_data{$option});
$options .= "$option_name $option_price,";
```

But the script must also calculate the cost changes due to options. To do so, it takes the current value of **$option_grand_total** and adds to it the cost of the current option. It then formats the result to two decimal places using the **format_price** subroutine discussed later and assigns the new result to **$option_grand_total**.

```
  $unformatted_option_grand_total = $option_grand_total
                                  + $option_price;
  $option_grand_total =
        &format_price($unformatted_option_grand_total);

  } # End of if ($option_item_number eq
 } # End of foreach $option (@options)
```

Next, the script takes off the last comma in **$options**. Look a few lines up; you will see that a comma is added to the end of each option. The last option should not display the last comma.

```
    chop $options;
```

Then, the script adds a space after each comma so that the display looks more neatly formatted.

```
$options =~ s/, /, /g;
```

The counter subroutine, discussed later, is then called and sent the location of the counter file defined in the Setup file as well as the file name and current line number.

This routine will return one variable called **$item_number**, that the script can use to identify a shopping cart item. This must be done so that when we modify and delete from the cart, we will know exactly which item to affect. We cannot rely simply on the unique database i.d. number because a customer may purchase two of the same item but with different options. Unless there is a separate, unique cart row i.d. number, how would the script know which to delete if the customer asked to delete one of the two? Thus, the counter subroutine assures that each cart row can be uniquely identified.

```
$item_number = &counter ($sc_counter_file_path,
                         __FILE__, __LINE__);
```

Finally, the script makes the last price calculations and appends every ordered item to **$cart_row**.

A completed cart row might look like the following:

```
2|0001|Vowels|15.98|Letter A|Times New Roman 0.00|15.98|161
```

The code is shown below:

```
$unformatted_item_grand_total = $item_price +
                                $option_grand_total;
$item_grand_total =
        &format_price("$unformatted_item_grand_total");

    foreach $field (@cart_row)
      {
      $cart_row .= "$field\|";
      }

$cart_row .= "$options\|$item_grand_total\|$item_number\n";
    } # End of foreach $item_ordered_with_options.....
```

When the script is through appending all the items to **$cart_row**, it opens the customer's shopping cart and adds the new items. If there is a problem opening the file, it will call **file_open_error** subroutine to handle the error reporting.

```
open (CART, ">>$sc_cart_path") ||
    &file_open_error("$sc_cart_path",
                     "Add to Shopping Cart",
                     __FILE__, __LINE__);
print CART "$cart_row";
close (CART);
```

Then, the script sends the customer back to a previous page. There are two pages that the customer can be sent to: the last product page they were on or the page displaying the customer's cart. Which page the customer is sent to depends on the value of **$sc_should_i_display_cart_after_purchase**, which is defined in the Web store Setup file. If the customer should be sent to the display cart page, the script calls **display_cart_contents**; otherwise it calls **display_page** if this is an HTML store cart or **create_html_page_from_db** if this is a Database store cart.

```
if ($sc_use_html_product_pages eq "yes")
  {
  if ($sc_should_i_display_cart_after_purchase
      eq "yes")
    {
    &display_cart_contents;
    }
  else
    {
    &display_page("$sc_html_product_directory_path/$page",
                  "Display Products for Sale");
    }
  }
else
  {
  if ($sc_should_i_display_cart_after_purchase eq "yes")
    {
    &display_cart_contents;
    }
  else
    {
    &create_html_page_from_db;
    }
  }
}
```

Output Modify Quantity Form

output_modify_quantity_form is the subroutine responsible for displaying the form that customers can use to modify the quantity of items in their cart. It is called with no arguments with the following syntax:

```
&output_modify_quantity_form;
```

The subroutine is explained below:

```
sub output_modify_quantity_form
  {
```

The subroutine begins by outputting the HTML header using **standard_page_header**, adds the modify form using **display_cart_table**, and finishes off the HTML page with **modify_form_footer**. All of these subroutines and their parameters are discussed in **web_store_html_lib.pl**.

```
  &standard_page_header("Change Quantity");
  &display_cart_table("changequantity");
  &modify_form_footer;
  }
```

Modify Quantity of Items in the Cart

The **modify_quantity_of_items_in_cart** subroutine is responsible for making quantity modifications in the customer's cart. It takes no arguments and is called with the following syntax:

```
&modify_quantity_of_items_in_cart;
```

The subroutine is discussed below:

```
sub modify_quantity_of_items_in_cart
  {
```

First, the script gathers the keys as it did previously for the **add_to_cart** routine, checking to make sure the customer entered a positive integer (not fractional and not less than one).

```
@incoming_data = keys (%form_data);
foreach $key (@incoming_data)
    {
    if ((($key =~ /[\d]/) &&
        ($form_data{$key} =~ /\D/)) ||
        $form_data{$key} eq "0")
      {
      &update_error_log("BAD QUANTITY CHANGE",
                        __FILE__, __LINE__);
      &bad_order_note;
      }
```

Just as in the **add_to_cart** routine previously, the script will create an array (**@modify_items**) of valid keys.

```
    unless ($key =~ /[\D]/ &&
            $form_data{$key} =~ /[\D]/)
      {
      if ($form_data{$key} ne "")
        {
        push (@modify_items, $key);
        }
      }
    } # End of foreach $key (@incoming_data)
```

Then, the script must open up the customer's cart and go through it line by line. Problems opening the file are, as usual, handled by **file_open_error**.

```
  open (CART, "$sc_cart_path") ||
    &file_open_error("$sc_cart_path",
                     "Modify Quantity of Items in the
                     Cart", __FILE__, __LINE__);
```

As the script goes through the cart, it will split each row into its database fields placing them as elements in **@database_row**. It will then grab the unique cart row number and subsequently replace it in the array.

The script needs this number to check the current line against the list of items to be modified. This list will be made up of all the cart items that are being modified.

Finally, the script chops the newline character off the cart row number.

```
while (<CART>)
  {
  @database_row = split (/\|/, $_);
  $cart_row_number = pop (@database_row);
  push (@database_row, $cart_row_number);
       chop $cart_row_number;
```

Next, the script checks to see if the item number submitted as form data is equal to the number of the current database row.

```
foreach $item (@modify_items)
 {
 if ($item eq $cart_row_number)
   {
```

If so, it means that the script must change the quantity of this item. It will append this row to the **$shopper_row** variable and begin creating the modified row. That is, it will replace the old quantity with the quantity submitted by the customer (**$form_data{$item}**). Recall that **$old_quantity** has already been shifted off the array.

```
$shopper_row .= "$form_data{$item}\|";
```

Now the script adds the rest of the database row to **$shopper_row** and sets two flag variables. **$quantity_modified** lets us know that the current row has had a quantity modification for each iteration of the while loop.

```
 foreach $field (@database_row)
   {
   $shopper_row .= "$field\|";
   }

 $quantity_modified = "yes";
 chop $shopper_row; # Get rid of last pipe symbol but
     # not the newline character

 } # End of if ($item eq $cart_row_number)
} # End of foreach $item (@modify_items)
```

If the script gets this far and **$quantity_modified** has not been set to "yes," it knows that the above routine was skipped because the item number submitted from the form was not equal to the current database i.d. number. Thus, it knows that the current row is not having its quantity changed and can be added to **$shopper_row** as is. Remember, we want to add the old rows as well as the new modified ones.

```
if ($quantity_modified ne "yes")
  {
  $shopper_row .= $_;
  }
```

Finally, the script clears out the **quantity_modified** variable so that next time the while loops start, it will have a fresh test.

```
 $quantity_modified = "";
 } # End of while (<CART>)
close (CART);
```

At this point, the script has gone all the way through the cart. It has added all of the items without quantity modifications as they were, and has added all the items with quantity modifications but made the modifications. The entire cart is contained in the **$shopper_row** variable.

The actual cart still has the old values, however. To change the cart completely the script must overwrite the old cart with the new information and send the customer back to the view cart screen with the **display_cart_contents** subroutine (to be discussed below). Notice the use of the write operator (**>**) instead of the append operator (**>>**).

```
open (CART, ">$sc_cart_path") ||
&file_open_error("$sc_cart_path",
                "Modify Quantity of Items in the Cart",
                __FILE__, __LINE__);
print CART "$shopper_row";
close (CART);

&display_cart_contents;
} # End of if ($form_data{'submit_change_quantity'}...
```

Displaying the Delete Item

The **output_delete_item_form** subroutine is responsible for displaying the HTML form that the customers can use to delete items from their cart. It takes no arguments and is called with the following syntax:

```
&output_delete_item_form;
```

As it did when it printed the modification form, the script uses several subroutines in **web_store_html_lib.pl** to generate the header, body, and footer of the Delete form.

```
sub output_delete_item_form
  {
  &standard_page_header("Delete Item");
  &display_cart_table("delete");
  &delete_form_footer;
  } # End of if ($form_data{'delete_item'} ne "")
```

Deleting Items from the Cart

The job of **delete_from_cart** is to take a set of items submitted by the customer for deletion and actually delete them from the customer's cart. The subroutine takes no arguments and is called with the following syntax:

```
&delete_from_cart;
```

As with the modification routines, the script first checks for valid entries. This time it only needs to make sure that it filters out the extra form keys. It is not necessary to make sure that it has a positive integer value as well, because unlike with a text entry, customers have less ability to enter bad values with checkbox submit fields.

```
sub delete_from_cart
  {
  @incoming_data = keys (%form_data);
  foreach $key (@incoming_data)
    {
```

We still want to make sure that the key is a cart row number and that it has a value associated with it. If the key is actually an item which the customer has asked to delete, the script will add it to the **@delete_items** array.

```
    unless ($key =~ /[\D]/)
      {
      if ($form_data{$key} ne "")
        {
        push (@delete_items, $key);
        }
      } # End of unless ($key =~ /[\D]/...
    } # End of foreach $key (@incoming_data)
```

Once the script has gone through all the incoming form data and collected the list of all items to be deleted, it opens up the cart and gets the **$cart_row_number** and **$db_id_number** as it did in the modification routines previously.

```
  open (CART, "$sc_cart_path") ||
     &file_open_error("$sc_cart_path",
                      "Delete Item From Cart",
                      __FILE__, __LINE__);
  while (<CART>)
    {
    @database_row = split (/\|/, $_);
    $cart_row_number = pop (@database_row);
    $db_id_number = pop (@database_row);
    push (@database_row, $db_id_number);
    push (@database_row, $cart_row_number);
    chop $cart_row_number;
```

Unlike modification, all we need to do for deletion is check to see if the current database row matches any submitted item for deletion. If it does not match, the script adds it to **$shopper_row**. If it is equal, it does not. Thus, all the rows will be added to **$shopper_row** except for the ones that should be deleted.

```
    $delete_item = "";
    foreach $item (@delete_items)
      {
      if ($item eq $cart_row_number)
        {
```

```
        $delete_item = "yes";
        }
      } # End of foreach $item (@add_items)
    if ($delete_item ne "yes")
      {
      $shopper_row .= $_;
      }
    } # End of while (<CART>)
  close (CART);
```

Then, as it did for modification, the script overwrites the old cart with the new information and sends the customer back to the view cart page with the **display_cart_contents** subroutine, which will be discussed later in this chapter.

```
 open (CART, ">$sc_cart_path") ||
      &file_open_error("$sc_cart_path",
                      "Delete Item From Cart",
  __FILE__, __LINE__);
  print CART "$shopper_row";
  close (CART);
  &display_cart_contents;
  } # End of if ($form_data{'submit_deletion'} ne "")
```

Displaying Products for Sale

display_products_for_sale is used to dynamically generate the product pages that the customer will want to browse through. However, there are two cases within it.

First, if the store is an HTML-based store, this routine will either display the requested page or, in the case of a search, perform a search on all the pages in the store for the submitted keyword. Second, if this is a Database-based store, the script will use the **create_html_page_from_db** to output the product page requested or to perform the search on the database. The subroutine takes no arguments and is called with the following syntax:

```
&display_products_for_sale;
```

The script first determines which type of store this is. If it turns out to be an HTML-based store, the script will check to see if the current request is a keyword search or simply a request to display a page. If it is a keyword search, the script will require the HTML search library and use the **html_search** subroutine within it to perform the search.

```
sub display_products_for_sale
  {
  if ($sc_use_html_product_pages eq "yes")
    {
    if ($form_data{'search_request_button'} ne "")
      {
      &standard_page_header("Search Results");
      require "$sc_html_search_routines_library_path";
      &html_search;
      &html_search_page_footer;
      exit;
      }
```

However, if the store is HTML-based but there is no current keyword, the script simply displays the page as requested with **display_page**, which will be discussed below.

```
&display_page("$sc_html_product_directory_path/$page",
                   "Display Products for Sale", __FILE__,
              __LINE__);
    }
```

On the other hand, if **$sc_use_html_product_pages** was set to "no," it means that the administrator wants the script to generate HTML product pages on the fly, using the format string and the raw database rows. The script will do so using the **create_html_page_from_db** subroutine, which will be discussed in the next section.

```
  else
    {
    &create_html_page_from_db;
    }
  }
```

Creating Dynamically Generated Product Pages from the Database

create_html_page_from_db is used to generate the navigational interface for database-based stores. It is used to create both product pages and list of products pages. The subroutine takes no arguments and is called with the following syntax:

```
&create_html_page_from_db;
```

First, the subroutine defines a few working variables that will remain local to this subroutine.

```
sub create_html_page_from_db
  {
  local (@database_rows, @database_fields, @item_ids,
         @display_fields);
  local ($total_row_count, $id_index, $display_index);
  local ($row, $field, $empty, $option_tag,
         $option_location, $output);
```

Next, the subroutine checks to see if there is actually a page that must be displayed. If there is a value for the page variable incoming as form data (for example, a list of products page), the script will simply display that page with the **display_page** subroutine and exit.

```
  if ($page ne "" &&
      $form_data{'search_request_button'} eq "")
    {
    &display_page
("$sc_html_product_directory_path/$form_data{'page'}",
                  "Display Products for Sale",
                  __FILE__, __LINE__);
    exit;
    }
```

If there is no page value, then the script knows that it must generate a dynamic product page using the value of the product form variable to query the database. First, the script uses the **product_page_header** subroutine in order to dynamically generate the product page header. The subroutine takes the value of the page we have been asked to display so that it can display something useful in the <TITLE></TITLE> area.

The **product_page_header** subroutine is located in **web_store_html_lib.pl** and **$sc_product_display_title** is defined in the Setup file.

```
  &product_page_header($sc_product_display_title);
    if ($form_data{'add_to_cart_button'} ne "" &&
     $sc_shall_i_let_client_know_item_added eq "yes")
      {
      print "$sc_item_ordered_message";
      }
```

Next, the database is queried for rows containing the value of the incoming product variable in the correct category as defined in the Web Store setup file. The script uses the **submit_query** subroutine in **web_store_db_lib.pl**, passing to it a reference to the list array **@database_rows**.

submit_query returns a descriptive status message if there is a problem and a total row count for diagnosing if the maximum rows returned variable is exceeded.

```
  if (!($sc_db_lib_was_loaded =~ /yes/i)) {
    &require_supporting_libraries (__FILE__, __LINE__,
                                   "$sc_db_lib_path");
  }
($status,$total_row_count) = &submit_query(*database_rows);
```

Now that the script has the database rows to be displayed, it will display them.

First, the script goes through each database row contained in **@database_rows** splitting it into its fields. For the most part, in order to display the database rows, the script will simply need to take each field from the database row and substitute it for a **%s** in the format string defined in the Web store Setup file.

However, in the case of options that will modify a product, the script must grab the code from an Options file. The special way that options are denoted in the database are by using the format **%%OPTION%%option.html** in the data file. The use of this flag is covered in more detail in Chapter 4 and includes two important items of information, as follows.

First, it begins with **%%OPTION%%**. This is a flag that will let the script know that it needs to deal with this database field as if it were an option. When the script sees the flag, it will then look what follows the flag to see which file it should load. Thus, in this example, the script would load the file **option.html** for display.

NOTE

Why go through all the trouble? Basically, because we need to create a system that will handle large, very likely similar, chunks of HTML code within the database. If there are options on product pages, it is probable that they will be repeated fairly often. For example, every item in a database might have an option like tape, cd, or lp. By creating one **options.html** file, we could easily put all the code into one shared location and not need to worry about typing it in for every single database entry.

```
foreach $row (@database_rows)
  {
  @database_fields = split (/\|/, $row);
  foreach $field (@database_fields)
    {
```

For every field in every database row, the script simply checks to see if it begins (^) with **%%OPTION%%**. If so, it splits the string into three pieces of information: an empty string, an OPTION tag, and the location of the option to be used. Then the script resets **$field** to null because it is about to overwrite it.

```
    if ($field =~ /^%%OPTION%%/)
      {
      ($empty, $option_tag, $option_location) =
      split (/%%/, $field);
      $field = "";
```

The Option file is opened and read. Every line of the Option file is appended to the **$field** variable and the file is closed again. However, the current product i.d. number is substituted for the **%%PRODUCT_ID%%** flag, which is a mandatory tag contained in all options files.

```
open (OPTION_FILE,
"$sc_options_directory_path/$option_location") ||
&file_open_error ("$sc_options_directory_path/$option_location",
"Display Products for Sale", __FILE__,
__LINE__);

while (<OPTION_FILE>)
  {

s/%%PRODUCT_ID%%/$database_fields[$sc_db_index_of_product_id]/g;
  $field .= $_;
  }
close (OPTION_FILE);
} # End of if ($field =~ /^%%OPTION%%/)
} # End of foreach $field (@database_fields)
```

Finally, the database fields (including the Option field that has been recreated) are stuffed into the format string **$sc_product_display_row** and the entire formatted string is printed to the browser along with the footer.

In doing so, however, we must format the fields correctly. Initially, **@display_fields** is created containing the values of every field to be displayed, including a formatted price field.

```
@display_fields = ();
@temp_fields = @database_fields;
foreach $display_index (@sc_db_index_for_display)
  {
  if ($display_index == $sc_db_index_of_price)
    {
    $temp_fields[$display_index] =
      &display_price($temp_fields[$display_index]);
    }
  push(@display_fields,
       $temp_fields[$display_index]);
  }
```

Then, the elements of the NAME field are created so that customers will be able to specify an item to purchase. We are careful to substitute double quote marks (") and greater and less than signs (>,<) for the tags **~qq~**, **~gt~**, and **~lt~**. The reason that this must be done is so that any double quote, greater than, or less than characters used in URL strings can be stuffed safely into the cart and passed as part of the NAME argument in the "add item" form. Consider the following item name that must include an image tag.

```
<INPUT TYPE = "text"  NAME = "item-0010|Vowels|15.98|The letter
A|~lt~IMG SRC = ~qq~Html/Images/a.jpg~qq~ ALIGN = ~qq~left~qq~~gt~"
```

Notice that the <IMG SRC> reference was edited. If it were not, how would the browser understand how to interpret the form tag? The form tag uses the double quote, greater than, and less than characters in its own processing, so the value of NAME of the tag cannot contain these special characters without this substitution.

```
@item_ids = ();
foreach $id_index (@sc_db_index_for_defining_item_id)
  {
  $database_fields[$id_index] =~ s/\"/~qq~/g;
  $database_fields[$id_index] =~ s/\>/~gt~/g;
  $database_fields[$id_index] =~ s/\</~lt~/g;
  push(@item_ids, $database_fields[$id_index]);
  }
```

Finally, **$sc_product_display_row** is created with the two arrays using **printf** to apply the formatting.

```
  printf ($sc_product_display_row,
          join("\|",@item_ids),
          @display_fields);
  } # End of foreach $row (@database_rows)
  &product_page_footer($status,$total_row_count);
  exit;
  }
```

Displaying the Contents of the Cart

display_cart_contents is used to display the current contents of the customer's cart. It takes no arguments and is called with the following syntax:

```
&display_cart_contents;
```

The subroutine begins by defining some working variables as local to the subroutine.

```
sub display_cart_contents
  {
  local (@cart_fields);
  local ($field, $cart_id_number, $quantity,
         $display_number, $unformatted_subtotal,
         $subtotal, $unformatted_grand_total,
         $grand_total);
```

Next, as when we created the modification and deletion forms for cart manipulation, we will use the routines in **web_store_html_lib.pl** to generate the header, body, and footer of the cart page. However, unlike with the modification and deletion forms, we will not need an extra table cell for the checkbox or text field. Thus, we will not pass anything to **display_cart_table**. We will simply get a table representing the current contents of the customer's cart.

```
  &standard_page_header("View/Modify Cart");
  &display_cart_table("");
  &cart_footer;
  exit;
  } # End of sub display_cart_contents
```

Handling File Open Errors

If there is a problem opening a file or a directory, it is useful for the script to output some information clarifying what problem has occurred. This subroutine is used to generate those error messages. **file_open_error** takes four arguments: the file or directory that failed, the section in the code in which the call was made, the current file name, and the line number. It is called with the following syntax:

```
&file_open_error("file.name", "ROUTINE", __FILE__, __LINE__);
```

The subroutine simply uses the **update_error_log** subroutine discussed later to modify the error log and then uses **CgiDie** in **cgi-lib.pl** to gracefully exit the application with a useful debugging error message sent to the browser window.

```
sub file_open_error
  {
  local ($bad_file, $script_section, $this_file,
         $line_number) = @_;
  &update_error_log("FILE OPEN ERROR", $this_file,
                    $line_number);
  &CgiDie ("I am sorry, but I was not able to access
           $bad_file in the $script_section routine of
           $this_file at line number $line_number.
           Would you please make sure the path is
           correctly defined in web_store.setup and
           that the permissions are correct.")
  }
```

Displaying a Pre-designed HTML Page

display_page is used to filter HTML pages through the script and display them to the browser window. **display_page** takes four arguments: the file directory that failed, the section in the code in which the erroneous call was made, the current file name, and the line number, and is called with the following syntax:

```
&file_open_error("file.name", "ROUTINE", __FILE__, __LINE__);
```

The subroutine begins by opening the requested file for reading, exiting with **file_open_error** if there is a problem as usual.

```
sub display_page
  {
  local ($page, $routine, $file, $line) = @_;
  open (PAGE, "$page") ||
        &file_open_error("$page", "$routine", $file,
        $line);
```

It then reads in the file one line at a time. However, on every line it looks for special tag sequences that it knows must be replaced in order to maintain the state information necessary for the workings of this script. Specifically, every form must include a page and a **cart_id** value and every URL hyperlink must have a **cart_id** value added to it.

Raw pre-designed HTML pages must include the following tag lines if they are to filter properly and pass along this necessary state information. All forms must include two hidden field lines with the substitution "tags" embedded as follows:

```
<INPUT TYPE = "hidden" NAME = "cart_id"
       VALUE = "%%cart_id%%">
<INPUT TYPE = "hidden" NAME = "page"
       VALUE = "%%page%%">
```

When the script reads in these lines, it will see the tags **"%%cart_id%%"** and **"%%page%%"** and substitute them for the actual **page** and **cart_id** values which came in as form data. Similarly, it might see the following URL reference:

```
<A HREF = "web_store.cgi?page=Letters.html&cart_id=">
```

In this case, it will see the **cartid=** flag and substitute the correct and complete **cartid=[SOME NUMBER]**.

```
  while (<PAGE>)
    {
    s/cart_id=/cart_id=$cart_id/g;
    s/%%cart_id%%/$cart_id/g;
    s/%%page%%/$form_data{'page'}/g;
```

Next, it checks to see if the **add_to_cart_button** button has been clicked. If so, it means that we have just added an item and are returning to the display of the product page. In this case, we will sneak in an additional confirmation message right after the **<FORM>** tag line so that the customer will know that the item was successfully added.

```
    if ($form_data{'add_to_cart_button'} ne "" &&
        $sc_shall_i_let_client_know_item_added eq
        "yes")
      {
      if ($_ =~ /<FORM/)
        {
        print "$_";
        print "$sc_item_ordered_message";
        }
      }
```

If it is any other line, simply print it out to the browser window. Once we have gone through all of the lines in the file, the HTML will be complete and filtered.

```
    print $_;
    }

  close (PAGE);
  } # End of sub display_page
```

Getting a Unique Cart Row Number with Counter

The **counter** subroutine is used to keep track of unique cart database i.d. numbers so that every item in every cart will be uniquely identifiable. The subroutine takes three arguments, the name of the counter file defined in the Setup file, the current filename, and the current line number. It is called with the following syntax:

```
&counter ($sc_counter_file_path, __FILE__, __LINE__);
```

First, the subroutine assigns to the local variable **$counter_file**, the filename that we passed to this subroutine from the main script. It also defines **$file**, **$line**, and **$item_number** as local.

```
sub counter
  {
local($counter_file, $file, $line) = @_;
local ($item_number);
```

Next, the script checks to see if the Counter file exists. If it does not, it attempts to create it.

```
  if (!(-e $counter_file))
    {
    open(COUNTER_FILE, ">$counter_file") ||
        &file_open_error("$counter_file", "Counter",
        $file, $line);
    print COUNTER_FILE "1\n";
    close(COUNTER_FILE);
    }
```

Next, the script opens the Counter file. If the Counter file cannot be opened, however, **&file_open_error** is called as usual.

```
  open (COUNTER_FILE, "$counter_file") ||
       &file_open_error("$counter_file", "Counter",
       $file, $line);
```

Then, the script checks to see what number the counter is currently on and assigns that value to **$item_number**.

```
  while (<COUNTER_FILE>)
    {
    $item_number = "$_";
    }
  close (COUNTER_FILE);
```

It then adds 1 to that number, changes the Counter file to reflect the addition, returns the number to the main script, and closes the Counter file.

```
  $item_number += 1;
  open (NOTE, ">$counter_file") ||
      &file_open_error("$counter_file", "Counter",
      $file, $line);
```

```
  print NOTE "$item_number\n";
  close (NOTE);
  return $item_number;
  } # End of sub counter
```

Updating the Error Log

update_error_log is used to append to the error log if there has been a process executing this script and/or email the administrator. The subroutine takes three arguments: the type of error, the current filename, and the current line number. It is called with the following syntax:

```
&update_error_log("WARNING", __FILE__, __LINE__);
```

The subroutine begins by assigning the incoming arguments to local variables and defining some other local variables to use during its work.

- **$type_of_error** will be a text string explaining what kind of error is being logged.
- **$file_name** is the current filename of this script.
- **$line_number** is the line number on which the error occurred. Note that it is essential that the line number, stored in **__LINE__** be passed through all levels of subroutines so that the line number value will truly represent the line number of the error and not the line number of some subroutine for error handling.

```
sub update_error_log
  {
 local ($type_of_error, $file_name, $line_number) = @_;
 local ($log_entry, $email_body, $variable, @env_vars);
```

The list of the HTTP environment variables is culled into the **@env_vars** list array and **get_date** is used to assign the current date to **$date**.

```
  @env_vars = keys(%ENV);
  $date = &get_date;
```

Now, if the administrator has instructed the script to log errors by setting **$sc_shall_i_log_errors** in the Web store Setup file, the script will create an error log entry.

```
  if ($sc_shall_i_log_errors eq "yes")
    {
```

First, the new log entry row is created as a pipe-delimited list beginning with the error type, filename, line number, and the current date.

```
    $log_entry = "$type_of_error\|FILE=$file_name\|LINE=$line_num-
ber\|";
    $log_entry .= "DATE=$date\|";
```

Then, the error log file is opened securely by using the lock file routines in **get_file_lock** discussed later in this chapter.

```
    &get_file_lock("$sc_error_log_path.lockfile");
    open (ERROR_LOG, ">>$sc_error_log_path") ||
         &CgiDie("The Error Log Could Not Be Opened");
```

Next, the script adds to the log entry row, the values associated with all of the HTTP environment variables, and prints the whole row to the log file it then closes and opens for use by other instances of this script by removing the lock file.

```
    foreach $variable (@env_vars)
      {
      $log_entry .= "$ENV{$variable}\|";
      }

    print ERROR_LOG "$log_entry\n";
    close (ERROR_LOG);
    &release_file_lock("$sc_error_log_path.lockfile");
    } # End of if ($sc_shall_i_log_errors eq "yes")
```

Finally, the script checks to see if the administrator has instructed it to also send an email error notification to the administrator by setting the **$sc_shall_i_email_if_error** in the Web store Setup file.

If so, it prepares an e-mail with the same information contained in the log file row and mails it to the administrator using the **send_mail** routine in **mail-lib.pl**. Note that a common source of email errors lies in the administrator not

setting the correct path for **sendmail** in **mail-lib.pl** on line 42. Make sure that you set this variable there if you are not receiving your mail and you are using the **sendmail** version of the mail-lib package.

```
  if ($sc_shall_i_email_if_error eq "yes")
    {
    $email_body = "$type_of_error\n\n";
    $email_body .= "FILE = $file_name\n";
    $email_body .= "LINE = $line_number\n";
    $email_body .= "DATE=$date\|";
    foreach $variable (@env_vars)
      {
      $email_body .= "$variable = $ENV{$variable}\n";
      }
    &send_mail("$sc_administrator_email",
               "$sc_administrator_email", "Web Store
               Error", "$email_body");
    } # End of if ($sc_shall_i_email_if_error eq "yes")
  }
```

Getting the Date

get_date is used to get the current date and time and format it into a readable form. The subroutine takes no arguments and is called with the following syntax:

```
$date = &get_date;
```

Since it will return the value of the current date, you must assign it to a variable in the calling routine if you are going to use the value.

```
sub get_date
  {
```

The subroutine begins by defining some local working variables

```
local ($sec,$min,$hour,$mday,$mon,$year,
       $wday,$yday,$isdst,$date);
local (@days, @months);
@days = ('Sunday','Monday','Tuesday',
         'Wednesday','Thursday', 'Friday','Saturday');
@months = ('January','February','March','April',
           'May','June','July', 'August','September',
           'October', 'November','December');
```

Next, it uses the **localtime** command to get the current time from the value returned by the time command, splitting it into variables.

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

Then the script formats the variables and assign them to the final **$date** variable. Note that $**sc_current_century** is defined in **web_store.setup**. Since the 20th century is really 1900–1999, we will need to subtract 1 from this value in order to format the year correctly.

```
  if ($hour < 10)
    {
    $hour = "0$hour";
    }
  if ($min < 10)
    {
    $min = "0$min";
    }
  if ($sec < 10)
    { $sec = "0$sec";
    }
  $year = ($sc_current_century-1) . "$year";
  $date = "$days[$wday], $months[$mon] $mday, $year at
           $hour\:$min\:$sec";
  return $date;
  }
```

Displaying the Price

display_price is used to format the price string so that the store can take into account differing methods for displaying prices. For example, some countries use **$xx.yy**. Others may use **xx.yy UNIT**. This subroutine will use the **$sc_money_symbol_placement** and the **$sc_money_symbol** variables defined in **web_store.setup** to format the entire price string for display. The subroutine takes one argument, the price to be formatted, and is called with the following syntax:

```
$price = &display_price(xx.yy);
```

Where **xx.yy** is some number like “23.99”.

NOTE

The main routine calling this subroutine must prepare a variable for the returned formatted price to be assigned to.

```
sub display_price
  {
  local ($price) = @_;
  local ($format_price);

  if ($sc_money_symbol_placement eq "front")
    {
    $format_price = "$sc_money_symbol $price";
    }
  else
    {
    $format_price = "$price $sc_money_symbol";
    }

  return $format_price;
  }
```

Making a Lockfile

get_file_lock is a subroutine used to create a lockfile. Lock files are used to make sure that no more than one instance of the script can modify a file at one time. A lockfile is vital to the integrity of your data. Imagine what would happen if two or three people were using the same script to modify a shared file (such as the error log) and each accessed the file at the same time. At best, the data entered by some of the customers would be lost. Worse, conflicting demands could possibly result in the corruption of the file.

Thus, it is crucial to provide a way to monitor and control access to the file. This is the goal of the lockfile routines. When an instance of this script tries to access a shared file, it must first check for the existence of a lockfile by using the **file lock** checks in **get_file_lock**.

If **get_file_lock** determines that there is an existing lockfile, it instructs the instance that called it to wait until the lockfile disappears. The script then waits and checks back after some time interval. If the lockfile still remains, it continues to wait until some point at which the administrator has given it permission to just overwrite the file because some other error must have occurred.

If, on the other hand, the lockfile has disappeared, the script asks **get_file_lock** to create a new lock file and then goes ahead and edits the file.

The subroutine takes one argument, the name to use for the lock file, and is called with the following syntax:

```
&get_file_lock("file.name");
```

The code is explained below:

```
sub get_file_lock
  {
  local ($lock_file) = @_;
  local ($endtime);
  $endtime = 60;
  $endtime = time + $endtime;
```

We set endtime to wait 60 seconds. If the lockfile has not been removed by then, there must be some other problem with the file system. Perhaps an instance of the script crashed and could never delete the lock file.

```
   while (-e $lock_file && time < $endtime)
    {
    # Do Nothing
    }
  open(LOCK_FILE, ">$lock_file");
```

NOTE

If **flock** is available on your system, feel free to use it. **flock** is an even safer method of locking your file because it locks it at the system level. The above routine is "pretty good" and it will server for most systems. But if you are lucky enough to have a server with **flock** routines built in, go ahead and uncomment the next line and comment the one above.

```
# flock(LOCK_FILE, 2); # 2 exclusively locks the file
  }
```

Deleting a Lockfile

release_file_lock is the partner of **get_file_lock**. When an instance of this script is through using the file it needs to manipulate, it calls **release_file_lock**

to delete the lockfile that it put in place so that other instances of the script can get to the shared file. It takes one argument, the name of the lock file, and is called with the following syntax:

```
&release_file_lock("file.name");
```

The code is explained below

```
sub release_file_lock
  {
  local ($lock_file) = @_;
  # flock(LOCK_FILE, 8); # 8 unlocks the file
```

As we mentioned in the note in the discussion of **get_file_lock**, **flock** is a superior file locking system. If your system has it, go ahead and use it instead of the version here. Uncomment the above line and comment the two that follow.

```
  close(LOCK_FILE);
  unlink($lock_file);
  }
```

Formatting Prices

format_price is used to format prices to two decimal places. It takes one argument, the price to be formatted, and is called with the following syntax:

```
$price =&format_price(xxx.yyyyy);
```

Notice that the main calling routine must assign the returned formatted price to some variable for its own use. Also notice that this routine takes a value even if it is longer than two decimal places and formats it with rounding. Thus, you can utilize price calculations such as 12.99 * 7.985 (where 7.985 might be some tax value).

```
  sub format_price
  {
```

The incoming price is set to a local variable and a few working local variables are defined.

```
  local ($unformatted_price) = @_;
  local ($formatted_price);
```

The script then uses the rounding method in EXCEL, which is also used in **sprintf**.

```
$formatted_price = sprintf ("%.2f", $unformatted_price);
return $formatted_price;
}
```