CHAPTER TWO

WEB STORE SETUP

Understanding the Setup File

In the Library directory, there are six sample setup files, which you can use to explore the possible store interfaces which may be defined. The six sample setup files are named:

1. web_store.setup.html
2. web_store.setup.db
3. web_store.setup.db.table
4. web_store.setup.frames
5. web_store.setup.frames.javascript
6. web_store.setup.frames.vbscript

These files go over the six most common paradigms used for stores that we have seen in the past, but you can certainly be a lot more creative than we have been. We offer these merely as demonstrations of the flexibility of this script.

1. The HTML store uses predefined HTML product pages to create the interface for the client. The administrator of this type of store has quite a bit of artistic leeway in how product pages are created but must follow a basic format in assigning values to the <INPUT> tag Name arguments. These arguments are discussed in Chapter 3.

2. The Database version of the store grabs the data to be displayed from a pipe-delimited flatfile ASCII text database. It then uses a format string to format the data for display in the browser window. You still have a great deal of artistic leeway in how data is presented but every product must be displayed in the format defined by the format variable. This is unlike the HTML store, which may have separate formats for each product on a page or between separate products pages. In the default setup file for the database store, we defined the format variable so that it will display products in the same way that the HTML store does. However, unlike the HTML store, product pages need not be created in advance because they will be generated on the fly by the script when the database file is queried for the product information.
3. The Query-based table display store is a secondary setup of the Database-based store. In this store, products are simply displayed with a table of database rows accessed by keyword search routines. The look is very generic, but this format may save space and time for customers merely interested in getting a quick list of items in a large store. More interestingly, we use this example to demonstrate the powerful searching tools available when using a database back end. The default front page of this store offers keyword as well as high and low price-range searching.
4. The Frames store is another template that we have provided as a sample. The frames version in the example uses a database method of creating the product pages. However, we could have also used the HTML store method of displaying pages inside the frames. The script presents the store as a frames-based site, including a main display frame, a table of contents frame, and a return to frontpage frame.
5. The JavaScript store is really an extension of option 4, but it allows the customer to do calculations on the product display screen using Netscape's JavaScript language inside the Netscape Web browser. This is the setup file hardcoded into **web_store.cgi** by default.
6. The VBScript store is also an extension of option 4, but allows the user to do calculations on the product display screen using Microsoft's VBScript language in Internet Explorer.

Which of the six setup files are actually used depends on what value you set for the setup file in **web_store.cgi**. At line 86 of **web_store.cgi**, you will see the following line:

```
&require_supporting_libraries (__FILE__, __LINE__,
     "./Library/web_store.setup.frames.javascript");
```

This line tells the script to load up **web_store.setup.frames.javascript**. If you change this to **web_store.setup.html** or **web_store.setup.db,** you will use the alternative setups instead. Eventually, you can settle on just one setup file and delete the others. They are not necessary, but are included as examples.

NOTE

If you are using a Windows NT or Windows 95 server, you may need to do additional modifications to the Web store to make it run. These changes are discussed in Chapter 1.

For the most part, these six setup files are the same. Most of the variables defined in them will be consistent for all the interfaces. Thus, in this chapter we will go over all the variables at one time, pointing out differences when they occur. The specific ways the store can be configured will be covered in more detail in the chapters that follow this one.

NOTE

All variables defined in the setup file are prefixed with **sc_** to denote that they are global shopping cart variables.

Global File Location Variables

Every setup file begins by defining the locations of supporting files that are needed throughout the life of the application. These paths can either be relative to the present working directory of **web_store.cgi** or can be hard-coded as absolute paths. In our default setup, we set paths relative to **web_store.cgi** so that when we define a path as **./Library/file.name** we are referring to a file called **file.name** in a directory called **Library**, which is a subdirectory of the directory containing **web_store.cgi**.

NOTE

If you are using a virtual server, be very careful with paths. What "you" see as your path when you log into your account may not be what the "Web server" sees when it executes **web_store.cgi**. The best thing to do is contact your Internet Service Provider (ISP) and ask them for a detailed explanation of how directories are mapped relative to your account and the Web server's account. In addition, many Windows NT servers do not run the script with the current working directory where the script resides. Possible solutions to these problems are discussed in Chapter 1 regarding the configuration of Windows NT/Windows 95–based Web servers.

In actual use, there is rarely any reason to change these variables from the values that are given in the default distribution. For most servers, the values should work just fine if you keep filenames and relative locations the same as in the distribution.

The only reasons you may want to change them is if you want to rename the files in question to be consistent with local filename standards, you are not allowed to use sub-directories in your ISP's **cgi-bin** directory, or you are on a virtual server with path mapping and the relative paths actually map the wrong locations. For example, your documents may be located in **/home/smithj/www/cgi-bin**. But a "virtual" Web server may see documents located in **/www/cgi-bin** since the server has been configured in such a way as to encapsulate what it sees so that the Web server sees only directories contained in your home directory. References like **./Library** will seem right to you, and will work if you run the scripts from the command line with your own user account, but they will be meaningless to the Web server, which may see a different relative path.

ISPs often do this because it enhances security to make sure that the Web server you are using cannot touch the files created by any other Web server instance on a shared machine.

If you are using a virtual Web server, you will want to get instructions from your ISP on how to use it for running CGI scripts. Remember, there are many different ways that a Web server can be set up. At some point it makes the most sense simply to ask your service provider how scripts should be configured at your site instead of experimenting with all the options and hoping something will work. The moral of this story is that if you want to change these path and filename location variables, feel free—but be careful.

- **$sc_cgi_lib_path** is the location of **cgi-lib.pl,** which is used to read and parse incoming form data.
- **$sc_mail_lib_path** is the location of **mail-lib.pl,** which is used to mail nonencrypted email.
- **$sc_html_search_routines_library_path** is the location of **web_store_html_search.pl**, which is used in case of a keyword search request by the customer using an HTML-based Web store.
- **$sc_html_setup_file_path** is the location of **web_store_html_lib.pl** in which subroutines for printing out much of the Web stores HTML are defined.
- **$sc_db_lib_path** is the location of **web_store_db_lib.pl**, which contains the database search routines
- **$sc_order_lib_path** is the location of **web_store_order_lib.pl**, which contains the routines that are used to process orders.
- **$sc_pgp_lib_path** is the location of **pgp-lib.pl**, which has a routine to automatically encrypt final cart orders for sending in email or logging to a file. You must have previously installed PGP on your Web server and configured it for use. Setting up and using PGP with the Web store will be discussed further in Chapter 9.
- **$sc_user_carts_directory_path** is the location of the subdirectory used to store the customers' shopping carts.
- **$sc_data_file_path** is the location of the ASCII text file database of items. This variable is necessary only if the store uses a database to generate product pages or if the store administrator wishes to check all orders derived from HTML-based product pages against a back-end database file. The reason you might want to do this is because, in theory, someone could hack the data between the product page and the order form and thus manipulate such fields as price. If you did not have a watchful eye over each order filled, someone might be able to sneak some hefty discounts right under your nose. This is discussed in more detail in Chapter 8, regarding order processing.
- **$sc_options_directory_path** is the location of the subdirectory that contains options files. These files are used to store the HTML code for options that may accompany items in a store. This variable is needed

only if you are using a database-based store. If you are using an HTML-based interface, you will hard code the options HTML directly into the product pages themselves.

- **$sc_html_product_directory_path** is the location of the subdirectory containing the HTML product pages and/or other HTML pages that aid in store navigation. Such aid is needed for both the HTML and database-based stores because list of product pages (without products but with lists of links that point to products) are used by both the HTML stores and the Database stores.
- **$sc_html_order_form_path** is the location of the HTML order form, which the customer will use to enter their shipping information.
- **$sc_store_front_path** is the location of the HTML front page for your store.
- **$sc_counter_file_path** is the location of the counter file that you will use to keep track of unique database row numbers for every item in the customers cart.
- **$sc_error_log_path** is the location of the flat-file error-log data file.
- **$sc_access_log_path** is the location of the flat-file access-log data file.
- **$sc_main_script_url** is the URL of **web_store.cgi**. This can either be relative or absolute. Remember, this is a URL used by the Web browser. It is not a real directory path on your server.
- **$sc_order_script_url** is the URL of the script that processes orders. If you are using a secure server setup, you may need to store this in a secure directory other than the one the rest of the script is contained in. This can also be either relative or absolute.

Database Definition Variables

These fields are used for the routines using the database-based version of the cart. They are also used for the HTML store if database verification of the products being ordered is turned on. Primarily, they define the format of your database so that the Web store script knows the location of each field of data. The most obvious need is for the script to know the field corresponding to an in order to make price calculations on items that have been added to customers' carts. The variables are:

- **%db** is an associative array that contains a mapping of your own customer-defined fields to the index number of the fields as they appear in the flat-file database. For example:

  ```
  $db{"product_id"}  = 0;
  $db{"product"}     = 1;
  $db{"price"}       = 2;
  $db{"name"}        = 3;
  $db{"image_url"}   = 4;
  $db{"description"} = 5;
  $db{"options"}     = 6;
  ```

  The code given above shows that **product_id** is the first field in the database. Remember, fields start counting at *zero!* An example of the database row corresponding to the above setup might be as follows:

  ```
  001|Product 1|1.00|1|<IMG SRC = "1.gif">|Number
  1|%%OPTION%%opt.html
  ```

- **@sc_db_display_fields** is an array containing descriptive headers for fields in the database we wish to display to users when they do a query search. Notice that these headers need not include every field defined in **%db**. You may display only some of the information contained in the database row if you so desire. The array takes advantage of the indexes defined in **%db** and, in addition, each descriptive element in this array corresponds to the index numbers in **@sc_db_index_for_display**.
- **@sc_db_index_for_display** is an array containing the index numbers of the database fields that correspond to the **display_fields** array. To access those index numbers, we just utilize the **%db** associative array. There must be one index number in this array for every descriptive element in **@sc_db_display_fields**.
- **@sc_db_index_for_defining_item_id** is an array containing the database fields that correspond to the fields from the database that you wish to associate with a customers cart when they select that item for purchase.

These will be used to determine which fields will make up each item row in the customers cart. If you do not put a database field in this array, it won't be put in the cart nor will be available for display when customers view their cart contents. However, it is essential that the price and options database fields become incorporated into the cart because these are used for cart subtotal calculations.

- **$sc_db_index_of_price** is an index to the field in the database that contains the price. This is used by the Web store to decide how to calculate and display money. In the example above, this value would be 2. Remember that even if you do not display price, its location must still be defined here.
- **@sc_db_query_criteria** is an array containing the criteria that can be used to search on the database with. This is a powerful search mechanism. The array contains pipe-delimited fields inside each list item. These fields are discussed in detail in Chapter 5. The fields are the following:

1. *Form variable name* This is the variable name which you want to associate with the Name parameter of any form field you use to gather customer data. For example, if you have the following text box for getting a keyword:

   ```
   <INPUT TYPE = text NAME = keywords>
   ```

   then the form variable **Name** would be **keywords**.
2. *Index into the database that this criteria applies to* This list corresponds to the **%db** associative array in the same way that **@db_index_for_display** does. Thus, if you want the Description and Name fields to be searched by keyword, the form_data Name would be **keywords** as above and the index into the database would be "3,5".
3. *Operator for comparison* This field is used by the script to determine what logical criteria to apply when searching the database. Possible values include: >, <, >= ,<= ,=, and != (not equal). The operators are compared the following way:

   ```
   form_variable OPERATOR database_field_value
   ```

That is, field 1 described above is the left side of the operator and field 2 is the right side of the operator.

4. *Data type of the field* This field determines how the operator in field 3 gets applied to the data. The data type can be: date, number, or string If the data type is a date, then the operator for comparison is done after the form value and the fields being compared in the database are converted to dates. If the data type is a number, then the operator for comparison is based on numerical Perl if operators (>, <, ==, etc.). If the data type is a string, then the operator for comparison is done based on string Perl if operators (gt, lt, eq, ne, etc.) with one exception. If the datatype is a string and the operator is =, then the search that is performed becomes a more flexible search using the following logic.

First, all the words in the customer-supplied search string are split apart and searched as separate keywords in the text of the fields. By default, the search on “string = string” is a pattern-match search and is not case-sensitive. If you want this special string, “=” combination searching to be case-sensitive and to match on whole words only, you must set up two new form variables: “case_sensitive” and “exact_match.”

If “exact_match” is “on” (the value of a checked checkbox) then the combination of string,”=” in the query criteria array will match on whole words only. If “case_sensitive” is “on” then the combination of string,”=”in the query criteria array must have matching case values (upper/lower).

This has been a lot of logic all at once. Perhaps a few examples are in order. They will demonstrate the three main cases that you generally want to set up a **query_criteria** array for:.

Case 1

General keyword search through the database.

Set the first field equal to your keywords form variable (keywords). You will need something like the following in your HTML page:

```
<INPUT TYPE = "text" NAME = "keywords"
      SIZE = "40"  MAXLENGTH = "40">
```

Next, set the second field equal to the field numbers of the database file you want to search. Since you are going to do a keyword search through multiple fields in the database, you want to comma-delimit these For example, assume that we use the sample **%db** defined a few pages back. We might use 1,3,5,6 where database fields 4 and 2 are not searched by keyword. After all, why would we want to keyword search on price and image URL?

Finally, set the operator to **=** and the data type to **string** so that the keyword search is done using pattern matching and is case insensitive. Then **@sc_db_query_criteria** would be equal to:

```
("keywords|1,2,3,4,6|=|string)
```

With this setup, when a customer submitted a keyword like **letter a**, the script would search fields 1, 3, 5, and 6 in that database.

Case 2

Suppose you just want to do a search on a product category and include that search term within URLs in a front page such as the following:

```
web_store.cgi?product=Vowels
```

In Chapter 4 you will see that this is in fact how we develop product pages on the fly for the Database store.

First, set the form variable equal to the above product. Second, set the second field equal to the field in the database corresponding to a product name (eg 1). Then, set the operator to **=**, and the data type to **string** to do a keyword search that is case insensitive. After these changes, **@sc_db_query_criteria** would be equal to:

```
("product|1|=|string")
```

Case 3

Finally, suppose you want to make a frontpage form in which several fields in the database are being searched for different types of keywords (or key numbers). That is, suppose you want to allow the user to search on a price range, plus do a keyword search on the description field. To solve this problem, **@sc_db_query_criteria** would be set equal to

```
("price_low_range|2|<=|number",
"price_high_range|2|>=|number",
"description|5|=|string");
```

Notice that we set up two form variables for allowing the price range searching. This is because we need to allow the customer to enter both the low range and the high range of the price they want to search for (assuming that price is database field number 2). Note, also, that the **price_low_range|2|<=** means that the database row returns a match if and only if the value of the **price_low_range** form field is less than or equal to the value of field 2 in the database row.

Thus if you have a database row with a price of $15.00, then entering the low range as **10.00** in the form field will return a match because 10.00 is less than 15.00, but if you enter a low range of **16.00**, a match would not be returned because 16.00 is not less than or equal to 15.00.

Finally, we set the description form variable to be a keyword (=,string) search on database field 5. The form itself would have these fields as HTML:

```
Lowest Price To Search For:
<INPUT TYPE = "text" NAME=price_low_range VALUE = "">
Highest Price To Search For:
<INPUT TYPE = "text" NAME=price_high_range VALUE = "">
Enter Keywords To Search For In Description:
<INPUT TYPE = "text" NAME=description VALUE = "">
```

NOTE

Only criteria that is entered on the form is queried against. If the customer leaves one or more fields blank (or you neglect to place them on the form), then those fields never get queried.

Of the criteria entered on the form, all the criteria must be satisfied for that row before the row will be considered safe to display to the customer.

The same goes for the **string,=** data type, operator combination (special case for keywords). When the string is split into keywords separated by white space, all the keywords must be found in the database field before the program will consider it a valid match.

This logic is there because we want to provide the capability of letting users narrow down the query as they enter more data into the form. This query logic is covered in more detail in Chapter 5.

$sc_db_max_rows_returned is the maximum number of rows you will allow to be displayed to the user as the result of a query. If the query gets above this number, the customer is presented with a message letting them know that they need to narrow their query down.

Cart Definition Variables

These variables define the structure of the customer carts.

The **%cart** associative array is similar to the **%db** associative array except that it is specific to the structure of the customers cart. It is defined via the following steps.

First, the first field is always the quantity of the purchased item. Subsequent fields are the same fields defined in the **@sc_db_index_for_defining_item_id** variable. This is done because whatever is defined in this array, becomes the part of the product defined in the customers cart. The field before the next to last always contains the options that the customer has chosen. The next to last field is always the price after options have been calculated in with the normal price. Finally, the last field is always a computer-generated, unique identifier to distinguish cart line items from each other.

If we use the same **%db** definition from the last section, and define **@sc_db_index_for_defining_item_id** as follows:

```
@sc_db_index_for_defining_item_id =
  ($db{"product_id"} ,
   $db{"product"} ,
   $db{"price"} ,
   $db{"name"} ,
   $db{"image_url"} ,
   $db{options} );Scc 3 bot
```

Then we would set the **%cart** associative array as follows:

```
$cart{"quantity"}             = 0;
$cart{"product_id"}           = 1;
$cart{"product"}              = 2;
$cart{"price"}                = 3;
```

```
$cart{"name"}                = 4;
$cart{"image_url"}           = 5;
$cart{"options"}             = 6;
$cart{"price_after_options"}  = 7;
$cart{"unique_cart_line_id"}  = 8;
```

- **$sc_cart_index_of_price** is the original database price of the item, which in this example is 3.
- **$sc_cart_index_of_price_after_options** is the price after the customer-selected options have been chosen—which, in the above example, equals 7.
- **$sc_cart_index_of_measured_value** is the index value of whatever field you want to use as a measured value in other calculations (specifically for calculating such things as shipping). For example, you may have a weight field that you wish to total in the cart so that the shipping cost changes with weight. Complex shipping logic is covered in Chapter 8.
- **@sc_cart_display_fields** is an array containing the descriptive names of the headers for displaying the cart fields. This follows the same logic as **@sc_db_display_fields** but is specific to **%cart** instead of **%db**.
- **@sc_cart_index_for_display** is an array containing an index into the cart for every field to be displayed to customers when they are viewing their own carts. These numbers must correspond to the descriptive names in **@sc_cart_display_fields**.
- **$sc_cart_index_of_item_id** is the index to the unique product i.d., which in the example above is equal to 1.
- **$sc_cart_index_of_quantity** is the index to the field of the the customer-submitted quantity for the item they are currently purchasing, which in the example above is 0. Remember, fields start counting at zero in Perl.

Order Form Definition Variables

The nuts and bolts of custom designed order form logic is covered in depth in Chapter 8. However, a brief overview appears here:

- **%sc_order_form_array** is the associative array of form variables, used on the order form to send in an order, such as asking for the user's name, address, and more. It maps a form field name with a descriptive name so that a legible email will be produced later.

- **@sc_order_form_required_fields** is an array containing the form field names (as defined in **%sc_order_form_array**) that are required fields. The order will not be processed without these field names being entered on the form.
- **$sc_order_with_hidden_fields** is "yes" or "no." If you want to submit orders to another server or to a MAILTO: url, then you can use this option to make sure that hidden fields are actually generated with the contents of the cart in them.

The following variables tell the ordering part of the script how to calculate shipping, discount, and sales tax and in what order.

The values are numerical (1, 2, 3, or 0 if we do not want to process anything). First, we determine whether these item (sales tax, shipping, and discount) are even calculated at all at either the display order form or at the point where the order form has been submitted for processing (**process_form**).

Variables that we do not calculate at a given time are given a value of 0 to show that they never enter into the calculation. Otherwise, they are given an order number of 1, 2, or 3 to show in what order they are calculated.

Here are the variables:

```
$sc_calculate_discount_at_display_form
$sc_calculate_discount_at_process_form
$sc_calculate_shipping_at_display_form
$sc_calculate_shipping_at_process_form
$sc_calculate_sales_tax_at_display_form
$sc_calculate_sales_tax_at_process_form
```

And below are a few examples of usage.

Example 1

We want shipping, discount, and sales tax to be calculated off of the original subtotal and then added to the subtotal all at once on the order form:

```
$sc_calculate_discount_at_display_form = 1;
$sc_calculate_shipping_at_display_form = 1;
$sc_calculate_sales_tax_at_display_form = 1;
```

Because they are all set to 1, they are all calculated at the same time and then added to the subtotal.

Example 2

We want the sales tax to get calculated first, added to the subtotal and then the discount and shipping gets calculated on the order form:

```
$sc_calculate_discount_at_display_form = 2;
$sc_calculate_shipping_at_display_form = 2;
$sc_calculate_sales_tax_at_display_form = 1;
```

Because **sales_tax** is 1, it gets calculated before the other values and then added to the subtotal. Then, the discount and shipping (value = 2) are calculated together and added to the subtotal based on the subtotal from above.

NOTE

If you are calculating something on the order form level, you need to calculate it on the **process_form** level as well; otherwise, the information will not be sent with the order in email.

NOTE

If a particular field such as shipping is dependent on the user entering a value into the order form, it cannot be calculated on the order form level. This is because the customer has not entered the form field value yet! Thus, you will need to set shipping to 0 at the **display_form** stage but set it to the appropriate value at the **process_form** stage since the customer will have, by then, entered the form value on the order form display.

- **@sc_order_form_shipping_related_fields** is an array containing the names of those form variables on the order form that will be used in calculating shipping. If you are calculating shipping without regard to form values, leave this array empty.
- **@sc_order_form_discount_related_fields** is an array containing the names of those form variables on the order form that will be used in calculating a discount for the customer. If you are calculating a discount without regard to form values, leave this array empty.
- **@sc_shipping_logic** is an array containing the logic for applying the shipping cost to the order. Each criteria is a separate list element. The fields within the criteria are pipe-delimited (|).

The values of the criteria are equal whole values (such as UPS or 5 or 11) or they can be ranges separated by hyphens (1-5, 1-, -5). If a number is left off one end of the hyphen, then the range is open-ended up to the value defined by the hyphen. For example, 5- signifies anything greater than or equal to 5.

The first fields correspond to the fields in the **@sc_order_form_shipping_related_fields** array. If this array is empty, then no fields in **@sc_shipping_logic** will correspond to the shipping.

The next field is the subtotal amount to compare against if you are determining shipping cost based on the total sum of money needed to purchase what is in the cart.

The next field after that is the quantity of items to compare against for determining shipping based on quantity.

The next field after quantity is the measured total of items based on the measured field index determined in the cart setup above.

The final field is the cost of the shipping if the criteria is matched in the above fields. If the value is followed by a % symbol, then the value of the shipping will be a percent of the current subtotal instead of a whole dollar amount.

Lets summarize this logic with a couple of examples. The first example will demonstrate a simple shipping based on quantity. The customer will simply choose items and the quantity of those items will determine the shipping price. Thus, **@sc_order_form_shipping_fields** will be empty because customers will not be able to change the shipping logic based on anything they fill out on the order form:

```
@sc_order_form_shipping_related_fields = ();
```

Next, we define the shipping logic:

```
@sc_shipping_logic = ("|11-||5",
  "|1-10||10");
```

In this example, the related fields are empty, so the logic fields start with the comparison of the subtotal. There is nothing in the logic field for subtotal, so it is not compared in determining shipping.

The next field has **11-**, and this is the quantity comparison field, so the quantity must be eleven or greater. For the second row (**1-10**), the quantity is compared to see if it falls within the range of 1 through 10.

The next field is blank (measured value) and because it is blank, it never enters into the matching check.

The final values (5 and 10) for the elements of the array are the dollar amounts that the shipping would be if the criteria was satisfied on the row. Thus, if the quantity is **1-10**, the shipping is $10.00. If the quantity is greater than or equal to 11, the shipping is $5.00.

In the second example, shipping is dependent upon form variables. The client is able to choose between either UPS or FedEx. The shipping logic depends on which company the client chooses. Since the variable is customer-defined, we need the following form tag in the order form:

```
<SELECT NAME = "22-shipping">
<OPTION>UPS
<OPTION>FEDEX
</SELECT>
```

We will also need to define our setup variables as follows:

```
@sc_order_form_shipping_related_fields =  ("22-shipping");
@sc_shipping_logic =  ("ups|1-10||5",
                       "ups|11-||10%",
                       "fedex|1-10||20",
                       "fedex|11-||30%");
```

Since there is one related form variable (**22-shipping**), the first field of the shipping logic is corresponds to the value of the **22-shipping** form variable. The rest of the fields are assigned in a manner similar to our first example..

If the form value of **22-shipping** is **UPS** and the quantity is greater than or equal to **1** and less than or equal to **10**, then shipping is $5. If the form value of **22-shipping** is **UPS** and the quantity is greater than **11**, then shipping is 10% of the current subtotal for the cart. If the form value of **22-shipping** is **FEDEX** and the quantity is greater than or equal to **1** and less than or equal to **10**, then shipping is $20. If the form value of **22-shipping** is **FEDEX** and the quantity is greater than **11**, then shipping is 30% of the current subtotal for the cart.

- **@sc_discount_logic** is an array containing the logic for applying a discount to the order. The discount is calculated as a dollar amount. Do not make the amounts negative. The Web store subtracts the values in this array from the subtotal when the discount is calculated.
- **$sc_sales_tax** is the value of sales tax. For example, Maryland has 5% sales tax, so this would be **.05** for Maryland residents.

- **$sc_sales_tax_form_variable** is the name of a form variable that will be used on the order form to determine if the sales tax is applicable.
- **@sc_sales_tax_form_value** are the possible (case-insensitive) values that the form variable above should be equal to in order to apply sales tax (such as **md** or **maryland**) .
- **$sc_order_email** is the email address to send orders to. Don't forget to escape any **@** signs. Thus:

  ```
  you@yourdomain.com
  ```

  must be written as:

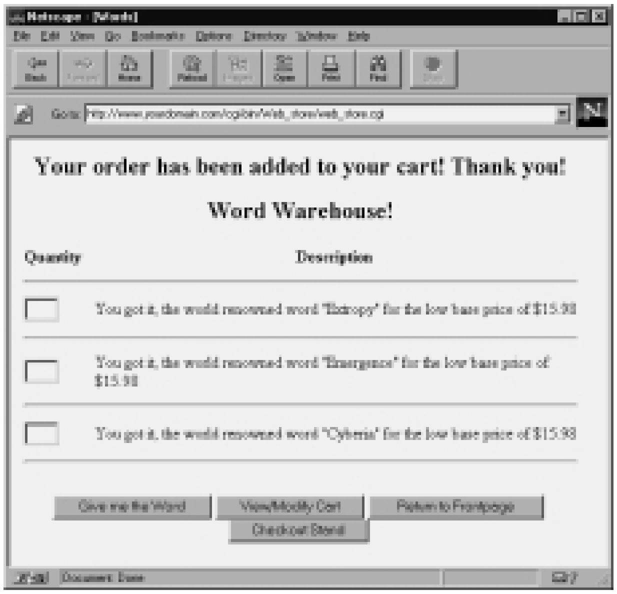
  ```
  youyourdomain.com
  ```
- **$sc_send_order_to_email** should be set equal to **yes** if you want orders sent to the above email address.
- **$sc_send_order_to_log** should be set equal to **yes** if you want the orders to be recorded in a local log file.
- **$sc_order_log_file** is the path and filename of the logfile where you want orders recorded if the above variable is **yes**.
- **$sc_order_check_db** should be set equal to **yes** if you want to use the database routines to double-check that the customer has not attempted to fool around with the database by entering in values for items based on form manipulation. If this variable is yes, the script double checks to see if the price in the cart is the same as the recorded price in the current database file.
- **$sc_use_pgp** should be set equal to **yes** if you want to use the PGP library to communicate with PGP for encrypting orders. You must have previously installed PGP on your system and set up your public/private key pairs. This is discussed in Chapter 9.
- **$sc_pgp_temp_file_path** is the path where you want the PGP program to generate temporary files. This should be a directory that is writable to the Web server.

Store Option Variables

- **$sc_use_html_product_pages** defines whether or not you want the script to be an HTML or Database store. If you set this to **yes**, the script will generate navigation using predesigned HTML pages in the

Html/Products subdirectory. If you set it to **no**, it will generate product pages dynamically from the d**ata.file** in the **Data_files** subdirectory.

- **$sc_should_i_display_cart_after_purchase** determines where the customer will go when they hit the **add this item to my cart** submit button. If this variable is set to **yes**, the customer will be sent to the cart display page where they can see the current contents of their cart (including the new item). If you set this equal to **no**, the customer will be sent back to whatever product page display they were just at with a note at the top of the page reminding the customer that the item had been added to the cart.
- **$sc_shall_i_let_client_know_item_added** defines whether or not the customer should be specifically told that the items they just ordered were added to the cart. If it is set to **yes** they will be told that items have been purchased. This is useful in the case that **$sc_should_i_display_cart_after_purchase** has been set to **no** because then customers have some feedback after being transported back to the page from which they just ordered (Figure 2.1).


Netscape - [Words]
File Edit View Go Bookmarks Options Directory Window Help
Back Forward Home Reload Images Open Print Find Stop
Go to: http://www.yourdomain.com/cgi-bin/Web_store/web_store.cgi

Your order has been added to your cart! Thank you!

Word Warehouse!

Quantity	Description
	You got it, the world renowned word "Entropy" for the low base price of $15.98
	You got it, the world renowned word "Emergence" for the low base price of $15.98
	You got it, the world renowned word "Cyberia" for the low base price of $15.98

Give me the Word
View/Modify Cart
Return to Frontpage
Checkout Stand
Document: Done

Figure 2.1 Customer feedback after order placement.

- **$sc_item_ordered_message** is the message that the customer receives if you have set **$sc_shall_i_let_client_know_item_added** equal to **yes**.
- **$sc_shall_i_email_if_error** defines whether or not the administrator receives an email whenever an error occurs. If this is set to **yes** the administrator will receive email whenever an error occurs during the processing of the script. If it is set to **no**, the administrator will receive nothing in email but is still free to review the error logs if the variable below is set to **yes**.
- **$sc_shall_i_log_errors** defines whether or not the script should log errors when they occur. If it is set to **yes**, all errors will be logged in the **error.log** file. If it is set to **no**, no logging will occur.
- **$shall_i_log_accesses** defines whether or not the script will log new access to **access.log.** If the variable is set to **yes**, all new accesses will be logged. If it is set to **no**, no accesses will be logged. Be careful with your log files. They may grow quickly so you may want to review them often and, if necessary, rotate them at some regular interval.

HTML Search Variables

These variables are used by HTML-based Web stores for document search requests:

- **$sc_root_web_path** defines the actual location of the HTML product pages that are being searched for keywords. This is not a URL, it is the actual path of the root HTML directory as the server would see it.
- **$sc_server_url** is the URL to the root HTML directory of product pages.
- **@sc_unwanted_files** is the list of all file extensions which should not be searched by the HTML search routines. Thus, if **.cgi** is in the list, the keyword search routine will not search any file with the file extension **.cgi**. Of course, you really should not store anything but HTML product pages under your HTML root directory.

Error Message Variables

- **$sc_page_load_security_warning** is the content of the error message sent to the browser window if someone attempts to view files that do not satisfy the **@acceptable_file_extensions_to_display** test.
- **$sc_randomizer_error_message** is the error message displayed to the browser if the application is not able to find a unique cart i.d. number.

Miscellaneous Variables

- **$sc_admin_email** is the email address to which error notifications are sent.
- **$sc_shall_i_email_if_error** is **yes** if you want to email to **$sc_admin_email** if an error occurred. Do not forget to escape any **@** signs in the **$sc_admin_email** variable. Thus:

  ```
  you@yourdomain.com
  ```

  must be written as:

  ```
  youyourdomain.com
  ```

- **@acceptable_file_extensions_to_display** lists the extensions that you will allow this application to display to the browser window. The goal is to restrict browsers to the absolute minimum files possible so that their ability to hack is limited to files that are publicly available anyway.
- **$sc_money_symbol** is the monetary symbol that you would like to display with prices. Make sure to escape the dollar sign ($) if you wish to use that symbol. Thus:

  ```
  $sc_money_symbol = "";
  ```

  is the correct syntax, *not:*

  ```
  $sc_money_symbol = "$";
  ```

- **$sc_money_symbol_placement** defines whether or not you want the amount to be placed in front or in back of the money symbol. If you set this variable equal to **back,** the monetary symbol will be placed after the number. If you set it to **front,** it will be placed in front. Thus, you might have $12.56 (front) or 12.56 $US (back).

- **$sc_current_century** is the current century. Note that if the year is 1997, you should set this value to **20**. The script will subtract one from this number when it prints out the current year.
- **$sc_number_days_keep_old_carts** defines how long old carts are kept in the **User_carts** subdirectory. The trick to this is that you want to leave carts in the directory long enough for the clients to finish shopping, but not so long that your hard disk usage gets too large. We recommend the compromise of half a day, which is writen as **.5.**
- **$sc_no_frames_button** defines the contents of any buttons you wish to display in nonframes implementations. This is usually the case with a button like **Return to Frontpage,** which loads the store frontpage again. The problem with the Frames version is that Submit buttons may not take a TARGET, so if the customer is able to hit a Return to frontpage button, they will cause the main window to be divided into a subframe with its own table of contents and main window. Thus, if you are running a Frames implementation, this variable should be empty. Otherwise, you may add the Return to frontpage button, which is shown by default.
- **$sc_product_display_title** is the text that should appear between the **<TITLE>** and **</TITLE>** tags on the dynamically generated database-based product pages.
- **$sc_product_display_header** is the header HTML used to display products in the database implementations. Notice that we use the **%s** characters to substitute for any given product information. The script will substitute product data for each product in place of **%s** when the variable is actually used.

NOTE

There must be a **%s** for every item in **@sc_db_index_for_display** because those elements will be what gets substituted for each **%s** in the order they are defined in the array.

- **$sc_product_display_footer** is the footer for each product
- **$sc_product_display_row** is the **%s** embedded product row variable.

Using web_store_check_setup.cgi

Using **Web_store_check_setup.cgi** is relatively easy. All you have to do is configure it to use the correct Setup file and then run it from your Web browser. This CGI script is designed to look into how your Setup file is defined and automatically check to see that all the directories and files have the appropriate permissions set to allow the Web server to run the script properly.

The first step is to change the setup file definition. Recall that in order to use a different Setup file in **web_store.cgi**, we had to edit **web_store.cgi** and change the actual setup file that was being referenced. You have to do the same with **web_store_check_setup.cgi**.

Line 34 of this file says the following:

```
$sc_setup_file = "./Library/web_store.setup.frames";
```

Change this variable to reflect the actual setup file you are using.

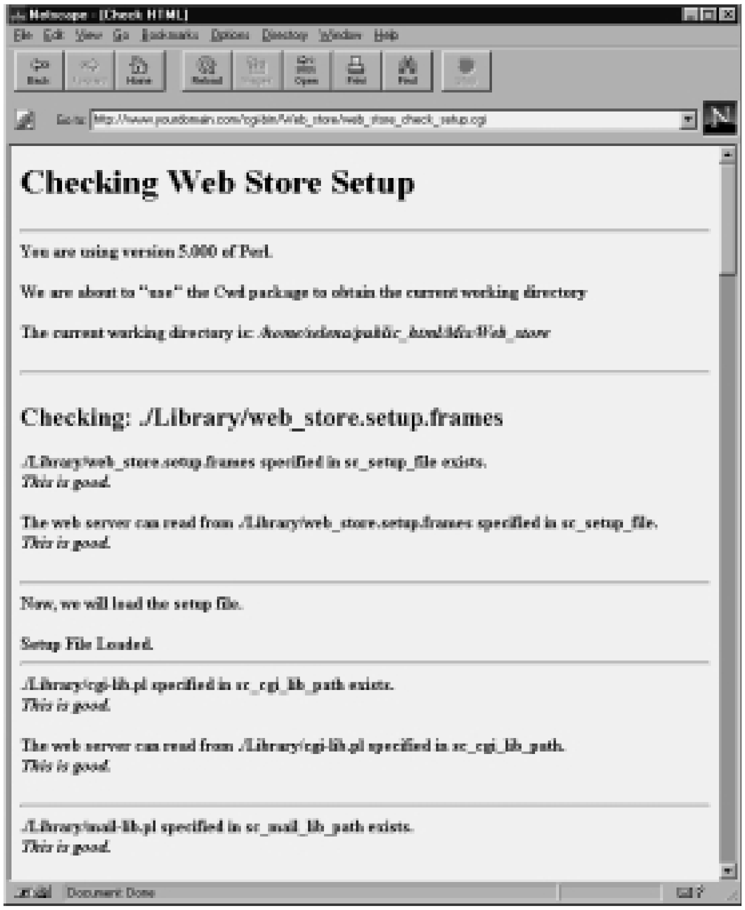
Then, all you have to do is run the script through your browser. For example, if your Web server is called **www.yourdomain.com**, and your Web store scripts are located under **cgi-bin/Web_store,** then you would call the check setup script with the following URL:

```
http://www.yourdomain.com/cgi-bin/Web_store/web_store_check_setup.cgi
```

Figure 2.2 shows an example of the output you should expect to see from the Setup checker script. The number-one setup problem usually encountered is that the scripts are not seen by the Web server as relative to the directory where the script is actually located. Thus, the first thing that the Web store Setup checking script does is print out the current working directory. The current working directory should be the same physical directory that the script is located in. If it is not, then you will need to configure the setup variables as absolute paths instead of paths relative to what you thought was the current working directory, or you can add a **chdir** command to the **web_store.cgi** script as we described in Chapter 1 under the possible Windows NT server Setup configurations.

If the current working directory is fine, the setup file will be required and then all the setup path/location related variables will be checked to see if the

Web server has permission to do what it needs to do to those directories. For example, the user carts subdirectory is checked to see if it is readable, executable, and writable by the Web server, but the libraries subdirectory is merely checked to see if it is readable and executable. If any permissions seem incorrect, a message will be printed out to let you know that there is a potential problem with running the script.


Netscape - [Check HTML]
File Edit View Go Bookmarks Options Directory Window Help
Back Forward Home Reload Images Open Print Find Stop
Go to: http://www.yourdomain.com/cgi-bin/Web_store/web_store_check_setup.cgi
Document: Done

```
Checking Web Store Setup

You are using version 5.000 of Perl.

We are about to "use" the Cwd package to obtain the current working directory

The current working directory is: /home/selena/public_html/Mix/Web_store

Checking: ./Library/web_store.setup.frames

./Library/web_store.setup.frames specified in sc_setup_file exists.
This is good.

The web server can read from ./Library/web_store.setup.frames specified in sc_setup_file.
This is good.

Now, we will load the setup file.

Setup File Loaded.

./Library/cgi-lib.pl specified in sc_cgi_lib_path exists.
This is good.

The web server can read from ./Library/cgi-lib.pl specified in sc_cgi_lib_path.
This is good.

./Library/mail-lib.pl specified in sc_mail_lib_path exists.
This is good.
```

Figure 2.2 Sample output from **web_store_check_setup.cgi**.

Understanding the HTML Library

For the most part, **web_store_html_lib.pl** contains all of the customizable HTML aspects of this application. If you want to change the look and feel of the Web store installation, you will probably have to make changes to this file.

Usually, this is as simple as cutting and pasting some of your template HTML in place of our default code. However, there are a few things that you should know about this file before you start changing it.

Taking Precautionary Measures

First, as always, make a backup of the original file before you start doing anything. Also, when you make changes, do so as a series of small changes. Make many small changes, running the script each time. That way you will not get caught making so many changes that you will not be able to easily recall at which point you made your error.

Using the "qq" Custom Quote Delimiter Feature

This library uses the **qq** Perl feature liberally. That means that you need to be aware of what the delimiter is in every bit of HTML code displayed. For the most part, we use the standard qq! and **qq~** when we are going to display a long bit of HTML code. The qq parameter will change the delimiter of a print string from double quote marks (**"**) to whatever follows the **qq**. This is done to allow double quote marks (which are very common in HTML) to be included in strings without requiring us to escape them everywhere using the backslash (character. Escaping the double-quote characters inside of HTML code makes the HTML look quite messy. Recall that normally we use the double quote marks to delineate the beginning and the ending of a text string to be printed. However, using **qq!** we can replace the double-quotes as delimiters with the **!** character.

Thus:

```
print "She said hi";
```

would become

```
print qq!She said "Hi"!;
```

Notice that in the first case, we had to escape the double quote marks in order to use the double quote marks as string delimiters. Since we use a lot of double quote marks in our HTML, it is easier to just change the delimiter to the bang (!) sign or the tilde (~) sign.

However, though the qq method does solve the problem of embedded double quotes, it does not solve the problem of Perl special characters. If you wish to include a dollar sign ($) or an *at* sign (@), for example, you will need to escape them with a backslash (\). Thus, to display:

```
send all $$$ to selena@eff.org
```

you must use backslashes as follows:

```
send all  to selenaeff.org
```

The reason for this is that the dollar and at signs have special meanings to Perl other than to denote price or an email address; specifically, to name scalar variables and list arrays.

If you are getting a **document contains no data** error message, it is a good bet you forgot to escape a Perl special character in this file.

Another thing to keep in mind is that we have included some variables within the default displays. For example, in **product_page_header** discussed later, we use both **$page_title** as well as **$hidden_fields** within the HTML. You probably want to leave these the way they are and write your own template HTML "around" them. If you delete them, make sure to keep a log of the changes you made so that you can retrieve them if you want them later.

Finally, a quick note about library syntax is in order since it will help you understand what is happening.

There are three levels of routines that you will be faced with in the Web store:

```
Individual routines
Application Specific Subroutines
Inter-Application Libraries
```

Individual Routines

Routines are merely pieces of code that perform some sort of action in a specific way. For example, an algorithm that adds two numbers can be expressed as:

```
x + y.
```

However, its use is very specific. To add 31,289 and 23,990, you would use the following Perl code:

```
$sum = 31289 + 23990;
```

This is a pretty simple routine. But it is also pretty specific to just one case. Programs typically consist of many routines like this put together. However, there comes a time when one wants to make the routine generic so that one can call it over and over again in other parts of the program without having to rewrite the routine. This is where application-specific subroutines come in.

Application-specific Subroutines

Routines that are general enough that they are used several times in the same application should usually be placed in a subroutine. A subroutine encapsulates an algorithm so that other parts of the program can refer to it by its subroutine name. Consider the addition routine from above.

What if we needed to add various numbers several times through our program, but did not want to create a separate piece of code for each instance of addition? If the algorithm were four or five lines long, it would be annoying to type the similar lines over and over again. Even worse, if you ended up changing the logic of the algorithm, you would have to hunt down every occurrence of the algorithm and change each one. Maintaining a program like this could become a nightmare, since many errors could arise from forgetting to change one of the lines of code in any of the duplicate routines or from changing one of them incorrectly.

When faced with such a circumstance, a programmer can create a "subroutine" within the application that can be used again and again by other parts of the program. **web_store.cgi** uses subroutines liberally. In fact, 90 percent of the program is actually contained in one subroutine or another.

In order to create and use subroutines in Perl, we need three things: a subroutine reference, a subroutine identifier, and a parameter list of variables to pass to the subroutine.

NOTE

Subroutine definitions can be placed anywhere in the script even if the call to that subroutine appears earlier in the program. This is because Perl first goes through the entire script and compiles references to all subroutines before it actually starts running the script. Although the subroutines may be placed anywhere in the program, their definitions are usually placed at the end of the script since it makes the main code easier to find and read.

In order to use a subroutine, the ampersand symbol (&) precedes the name of the routine. This tells Perl to look for the subroutine in the program and call it. For example, **&AddNumbers** would direct Perl to execute the **AddNumbers** subroutine.

However, we will also need to be able to send the subroutine some information. Specifically, we will need to send the subroutine parameters which it will use to customize its output. If we want to get the sum of 2 and 3, for example, we can pass 2 and 3 to the subroutine using the following format:

```
 &AddNumbers(2,3);
```

The definition of the subroutine itself is marked off in the program using a **sub** marker and the code belonging to the routine is delimited with curly brackets ({}). The following example shows what the **AddNumbers** subroutine definition might look like:

```
sub AddNumbers
  {
  local($first_number, $second_number) = @_;
  print $first_number + $second_number;
  }
```

Note the third line above. We use the **local** keyword to make sure that the **$first_number** and **$second_number** variables will be considered local to only that subroutine. That is, the subroutine will not affect any other variables that may be called **$first_number** or **$second_number** in other subroutines within the program.

Also, in Perl, @_ is a special name for the list of parameters that have been passed to the function. **$first_number** is set equal to the first parameter and **$second_number** is set equal to the second parameter in the @_ list of parameters. If the routine is called by **&AddNumbers(2,3)**, 2 and 3 will be assigned to **$first_number** and **$second_number**, respectively.

NOTE

It is important to use local variables within subroutines so that they do not overwrite variables used by the main script. In complex scripts that use dozens of variables, you may easily forget which variables you are using. Using local variables assures that if you do end up using the same name for a variable, you can keep them separated. Whenever you want to add numbers, you can simply use the subroutine call **&AddNumbers(x,y)** instead of writing out each addition individually. As an added bonus, if you need to change the logic of the addition algorithm, you need only change it in the subroutine.

Finally, a subroutine can return a value to the main script by using the keyword **return** and assigning the value to a variable at the time of the call. Thus, the following code would set **$sum** equal to 5.

```
$sum = &AddNumber(2,3);
sub AddNumber
{
local($left_hand_side, $right_hand_side) = @_;
return ($left_hand_side + $right_hand_side);
}
```

Inter-application Libraries

Good design does not stop with the mere use of subroutines. Often, several different scripts will incorporate the use of similar routines into their design. Or similar or customizable subroutines will be broken out for easy access or modification (as in the case of **web_store_html_lib.pl** which brings all the customizable HTML into one place). In this case, it makes sense to remove the common routines from the programs and place them in a separate file of routines.

This file can then be loaded as a library of subroutines into each program as needed.

For example, most CGI applications will need a form gathering/parsing routine, a template for sending out the HTTP header, and perhaps another subroutine to generate template HTML code such as:

```
<HTML><HEAD><TITLE>My Script Title</TITLE></HEAD>
```

In this case, we use library files and "require" them from the main script. A library file in Perl is simply a text file containing subroutines that are shared in common by several different Perl scripts.

NOTE

In addition to all the normal Perl code for subroutines, a Perl library must end with a **1;** on the last line. This is because the REQUIRE command evaluates the script when it reads in the library routines. The last **1;** makes the script itself evaluate to TRUE. In other words, a return value of 1 in the Perl library means the library was read in successfully by the main Perl script.

In order for these library files to be usable by the script, the permissions must be set on the library file to make it readable, and the script must be in the Perl library path or its location must be explicitly referenced with its directory path location. For example, if we wanted to load Steven Brenner's **cgi-lib.pl** library into our script and **cgi-lib.pl** was located in the same directory as the script calling it or was in a directory included in the **@INC** array, we would use the following:

```
require "cgi-lib.pl";
```

When this is done, every subroutine in c**gi-lib.pl** becomes accessible to the main script as if it were actually written into the script's code. Thus we can simply reference a subroutine contained in **cgi-lib.pl** as we would any other subroutine in the main program.

NOTE

Library files need to be readable by the script that requires them. If your server is running as "Nobody" in reference to Chapter 1 on setting up script permissions, you may need to make the library files readable by the world.

Well, that was a lot of information to swallow at once. But now you are ready to go and play havoc with the HTML in this file that are contained as "subroutines" and are "required" by **web_store.cgi**. Do not forget to make a backup, but feel free to experiment. The store distributed by default is kept bland on purpose. By remaining simple, it makes customizing the interface easier for most people. It is your job to turn the default interface into your own unique store.

If you would like to see how others have customized the interface, check out Selena Sol's Script Archive. There you can explore the section called "Web Store Scripts in Action" located at http://www.edd.org/verict/Scripts/ to look at the dozens of existing real-world implementation of the Web store.

HTML Library Routines

The following is a list of all the subroutines that are defined in the HTML library. Use this as your road map to deciding what routines you need to change in order to alter the cosmetic look-and-feel of your specific Web store:

- **product_page_header** is used to display the shared HTML header used for database-generated product pages. It takes one argument, **$page_title**, which will be used to fill the data between the **<TITLE>** and **</TITLE>**. Typically, this value is determined by **$sc_product_display_title** in **web_store.setup**. The subroutine is called with the following syntax:

  ```
  &product_page_header("Desired Title");
  ```

- **product_page_footer** is used to generate the HTML page footer for database-based product pages. It takes two arguments, **$db_status** and **$total_rows_returned** and is called with the following syntax:

  ```
  &product_page_footer($status,$total_row_count);
  ```

- **html_search_page_footer** is used to generate the HTML footer for HTML-based product pages when the script must perform a keyword search and generate a list of hits. It is called with no arguments with the following syntax:

  ```
  &html_search_page_footer;
  ```

- **standard_page_header** is used to generate a standard HTML header for pages within either the HTML-based or Database-based stores. It takes a single argument and the title of the page to be displayed. Itis called with the following syntax:

  ```
  &standard_page_header("TITLE");
  ```

- **modify_form_footer** is used to generate the HTML footer code for the "Modify quantity of items in the cart" form page. It takes no arguments and is called with the following syntax:

  ```
  &modify_form_footer;
  ```

- **delete_form_footer** is used to generate the HTML footer code for the "Delete items from the cart" form page. It takes no arguments and is called with the following syntax:

  ```
  &delete_form_footer;
  ```

- **cart_footer** is used to generate the HTML footer code for the "View items in the cart" form page. It takes no arguments and is called with the following syntax:

  ```
  &cart_footer;
  ```

- **bad_order_note** generates an error message for users in case they have not submitted a valid number for a quantity. It takes no arguments and is called with the following syntax:

  ```
  &bad_order_note;
  ```

- **cart_table_header** is used to generate the header HTML for different views of the cart. It takes one argument, the type of view we are requesting and is called with the following syntax:

  ```
  &cart_table_header(TYPE OF REQUEST);
  ```

- The job of **display_cart_table** is to display the current contents of the customers cart for several different types of screens that all display the cart in some form or another. The subroutine takes one argument, the reason that the cart is being displayed, and is called with the following syntax:

  ```
  &display_cart_table("reason");
  ```

There are really only five values that **$reason_to_display_cart** should be equal to:

1. "" (view/modify cart screen)
2. **"changequantity"** (change quantity form)
3. **"delete"** (delete item form)
4. **"orderform"** (order form)
5. **"process order"** (order form process confirmation)

Notice that this corresponds closely to the list in **cart_table_header** because the goal of this subroutine is to fill in the actual cells of the table created by the **cart_table_header** subroutine.

- **cart_table_footer** is used to display the footer for cart table displays. It takes one argument, the pre shipping grand total and is called with the following syntax:

```
&cart_table_footer(PRICE);
```

- **make_hidden_fields** is used to generate the hidden fields necessary for maintaining state. It takes no arguments and is called with the following syntax:

```
&make_hidden_fields;
```

- **PrintNoHitsBodyHTML** is utilized by the HTML store search routines to produce an error message in case no hits were found based on the customer-defined keywords. It is called with no arguments and the following syntax:

```
&PrintNoHitsBodyHTML;
```

- **PrintBodyHTML** is utilized by the HTML store search routines to produce a list of hits. These hits will be the pages that had the customer-defined keywords within them. The subroutine takes two arguments, the filename as it will appear in the URL link as well as the text that should be visibly hyperlinked. It is called with the following syntax:

```
&PrintBodyHTML("file.name", "Title to be linked");
```

Summary

After finishing this chapter, you should have a general idea of how to set up the Web Store script. The following chapters will cover the specific types of installations that are used on the Internet and how using each of them affects the various configuration options for the Web Store. In addition, other chapters will take certain topics that have been covered lightly here such as ordering logic and explain them with the depth necessary for more advanced Web Store setups.