CHAPTER FIFTEEN

THE ORDER LIBRARY

The Order Library (**web_store_order_lib.pl**) is the Web store's interface to all the order-processing-related routines. This includes everything from the initial display of the order form to the complex processing involved in calculating shipping, discounts, and sales tax.

The two main subroutines that are called from the Web store are **display_order_form** and **process_order_form**. There are also several other routines that act as scaffolding for these two procedures.

Display_order_form Subroutine

The **display_order_form** subroutine displays the order form to the user. It does not use any parameters except for variables that have been configured in the Setup file to affect the order form display. These variables have been previously discussed in Chapter 8.

```
sub display_order_form {
```

$line, **$subtotal**, **$total_quantity**, **$total_measured_quantity**, **$text_of_cart**, and **$hidden_fields_for_cart** are all declared local to this subroutine.

```
  local($line);
```

```
  local($subtotal);
  local($total_quantity);
  local($total_measured_quantity);
  local($text_of_cart);
  local($hidden_fields_for_cart);
```

First, the order form HTML file is opened. The filename and path for this file is taken from the **$sc_html_order_form_path** defined in the Setup file. If there is an error opening the file, then it is reported to the script using the **file_open_error** routine.

```
open (ORDERFORM, "$sc_html_order_form_path") ||
    &file_open_error("$sc_html_order_form_path",
      "Display Order Form File
Error",__FILE__,__LINE__);
```

Next, every line of the order form file is read into **$line** using a while loop. This line is then parsed to see if it should display as is or if some piece of cart information needs to be substituted in place of special tags. These special tags will be discussed below.

```
while (<ORDERFORM>) {
    $line = $_;
```

If the <FORM> tag is encountered, then it is replaced with a form tag that is generated based on the value in the Setup file. In addition to this new form tag, hidden fields for the current **cart_id** and **page** that the customer previously visited are placed in the form.

```
    if ($line =~ /<FORM/i) {
      print qq!
      <FORM METHOD = "post" ACTION =
"$sc_order_script_url">
      <INPUT TYPE = "hidden" NAME = "page"
             VALUE = "$form_data{'page'}">
      <INPUT TYPE = "hidden" NAME = "cart_id"
             VALUE = "$form_data{'cart_id'}">\n!;
      $line = "";
    } # End of If Form tag found
```

If there was a tag stating where the cart contents should appear, then the cart contents are inserted in place of the line. The regular expression that is matched against the line is expected to match a string such as "<H2>CART CONTENTS HERE</H2>". This string gets replaced with the customer's cart contents on the order form when it gets displayed.

```
if ($line =~ /<h2>cart.*contents.*h2>/i) {
```

The **display_cart_table** subroutine is called with **orderform** for a parameter to let the routine know that it is dealing with the Display order form routine. The **display_cart_table** subroutine returns the subtotal, total quantity of items in the cart, total measured quantity of the measurement field specified in the Setup file, and the ASCII text of the cart. The ASCII text of the cart is used for logging and emailing the orders.

```
($subtotal,
 $total_quantity,
 $total_measured_quantity,
 $text_of_cart) =
           &display_cart_table("orderform");
```

display_calculations is then called in order to print the shipping, discount, and sales tax calculations where it is appropriate to do so based on the values in the Setup file. The ASCII text of the calculations will be appended to **$text_of_cart** for future logging and emailing of orders.

```
 $text_of_cart =
   &display_calculations($subtotal,"before",
         $text_of_cart);
```

Of course, since the script is displaying the cart contents at this time, it needs to make the **$line** blank so that it does not display. The special tags that get replaced with the new form information and the cart contents should not be displayed to the user. They are merely placeholders for the information that the customer actually wants to see.

```
   $line = "";
}
```

Finally, the **$line** variable is printed. After the while loop, the routine ends.

```
    print $line;
  } # End of Parsing Order Form

} # End of display_order_form
```

Display_calculations Subroutine

The **display_calculations** subroutine displays all the cart-specific calculations such as shipping costs, discounts, and sales tax. It gets passed the current subtotal, whether we are displaying calculations at the order form or at a point where the order form is being processed, and the actual text of the cart.

```
sub display_calculations {
  local($subtotal,
        $are_we_before_or_at_process_form,
        $text_of_cart) = @_;
```

The first thing that the script does is call the **calculate_final_values** routine, which will calculate the sales tax, shipping, and discount. It also calculates the grand total after all of these calculations have been applied. The reason this subroutine also calculates the grand total is that the shipping, discount, and sales tax are not guaranteed to be applied all at once to the subtotal of the cart. The **calculate_final_values** routine takes care of that type of logic. The routine takes the subtotal, total quantity of items in the cart, the total measured quantity of the measured field specified in the Setup file, and a flag indicating whether the order form is being displayed or processed. These values are necessary for calculating the shipping, discount, and sales tax as well as generating a new grand total.

```
local  ($final_shipping,
          $final_discount,
          $final_sales_tax,$grand_total) =
    &calculate_final_values($subtotal,
                        $total_quantity,
                        $total_measured_quantity,
                      are_we_before_or_at_process_form);
```

If there is a final shipping value, then the script prints it and appends it to the text of the cart using the **format_text_field** function to justify the text of the cart.

```
if ($final_shipping > 0) {
  $final_shipping = &display_price($final_shipping);
  print "Shipping: $final_shipping<P>";
  $text_of_cart .= &format_text_field("Shipping:") .
    "= $final_shipping\n\n";
};
```

If there is a final discount value, then it is printed to the user's Web browser and appended to the text of the cart using the **format_text_field** routine.

```
if ($final_discount > 0) {
  $final_discount = &display_price($final_discount);
  print "Discount: $final_discount<P>";
  $text_of_cart .= &format_text_field("Discount:") .
    "= $final_discount\n\n";
}
```

If there is a final sales tax value, it is printed and appended to the text of the cart using the **format_text_field** routine.

```
if ($final_sales_tax > 0) {
  $final_sales_tax = &display_price($final_sales_tax);
  print "Sales Tax: $final_sales_tax<P>";
  $text_of_cart .= &format_text_field("Sales Tax:") .
    "= $final_sales_tax\n\n";
}
```

The real grand total is displayed. Then it is also appended to the text of the cart, just as the calculations above were.

```
$grand_total = &display_price($grand_total);
print "Grand Total: $grand_total<P>";
$text_of_cart .= &format_text_field("Grand Total:") .
 "= $grand_total\n\n";
```

Finally, the ASCII text of the cart is returned to the caller of this subroutine.

```
  return ($text_of_cart);
} # end of display_calculations
```

Format_text_field Subroutine

The **format_text_field** subroutine is a very simple routine. It merely accepts a text field and formats it so that it is left-justified with a field width of 25 charcaters. Basically, a string of 25 characters (**“” x 25**) is truncated by the string contained in `svalue` using the `substr` command.

```
sub format_text_field {
  local($value) = @_;

  return($value . substr((" " x 25), length($value)));

} # End of format_text_field
```

An example of what the formatted order looks like follows:

```
Description         = The letter A
Options             = Times New Roman 0.00, Red 0.00
Price After Options = $15.98
Quantity            = 1
Item Subtotal       = $15.98

Description         = The letter E
Options             = Times New Roman 0.00, Red 0.00
Price After Options = $12.98
Quantity            = 1
Item Subtotal       = $12.98

Subtotal:           = $28.96

Shipping:           =  $5.00

Discount:           =  $1.00

Sales Tax:          =  $8.60

Grand Total:        = $41.56
```

Process_order_form Subroutine

The **process_order_form** subroutine takes the contents of the order form that the user filled out previously and processes it as an actual order. This order is then logged and/or emailed to the store owner as specified in the Setup file.

```
sub process_order_form {
```

$subtotal, **$total_quantity**, **$total_measured_quantity**, **$text_of_cart**, and **$required_fields_filled_in** are declared local to this routine. These variables are used in a similar context to the way they were used in the **display_order_form** subroutine. The only different variable is **$required_fields_filled_in**. This variable is added because on the order form that was filled out, there may be some required fields that the user must enter before the form is submitted. If the required fields were not entered, then the script will print an error message letting the user know that they need to fill in more fields on the order form.

```
local($subtotal, $total_quantity,
      $total_measured_quantity,
      $text_of_cart,
      $required_fields_filled_in);
```

The first thing the routine does is print the header of the process order form.

```
print qq!
<HTML>
<HEAD>
<TITLE>Processing The Order Form</TITLE>
</HEAD>
</BODY>
!;
```

Then, the cart table is displayed to the user just as it was displayed in the **display_order_form** subroutine, except the routine is told that it is being called from the part of the Web store currently processing orders by sending it the

"process order" parameter. Then, **display_calculations** is also called in the same manner as the **display_order_form** routine, except that this routine is also passed information letting it know that the order form is being processed.

```
($subtotal,
 $total_quantity,
 $total_measured_quantity,
 $text_of_cart) =
    &display_cart_table("process order");

$text_of_cart =
    &display_calculations($subtotal,"at",
         $text_of_cart);
```

Next, the required fields processing is performed before actually processing the order. By default, **$required_fields_filled_in** is set to "yes," indicating that the routine starts out assuming all the fields were filled in. Then, the script goes through each required field one by one to see if it was filled in. If any one of the fields were not filled in, the **$required_fields_filled_in** flag is set to "no" and a message is displayed to the user letting them know exactly which fields they forgot to fill in.

```
$required_fields_filled_in = "yes";
foreach $required_field (@sc_order_form_required_fields) {
  if ($form_data{$required_field} eq "") {
    $required_fields_filled_in = "no";
    print "<H2>You forgot to fill in " .
      $sc_order_form_array{$required_field} . ".</H2>\n";
  }
} # End of checking required fields
print "<HR>\n";
```

If the required fields were filled in, the rest of the order is actually processed.

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

The ASCII text of the cart stored in **$text_of_cart** is appended with all the values that the user entered into the order form. The **format_text_field** routine is used to make sure that the form values are lined up and left justified.

```
foreach $form_field (sort(keys(%sc_order_form_array))) {
  $text_of_cart .=
    &format_text_field($sc_order_form_array{$form_field})
    . "= $form_data{$form_field}\n";
}
$text_of_cart .= "\n";
```

If PGP has been configured so that the Web store is using it, then the text of the cart is translated to a PGP-encrypted state using the **pgp-lib.pl** discussed in Chapter 9 and Chapter 18.

```
 if ($sc_use_pgp =~ /yes/i) {
  &require_supporting_libraries(__FILE__, __LINE__,
    "$sc_pgp_lib_path");
$text_of_cart = &make_pgp_file($text_of_cart,
                "$sc_pgp_temp_file_path/$$.pgp");
$text_of_cart = "\n" . $text_of_cart . "\n";
}
```

If the order is configured to be sent through email, then the **send_mail** routine in **mail-lib.pl** is used to send the email.

```
if ($sc_send_order_to_email =~ /yes/i) {
  &send_mail($sc_order_email,$sc_order_email,
             "Web Store Order", $text_of_cart);
}
```

If the setup is configured to send the order to a Log file, then the following section of the routine appends the Order log file with the text of the order. Each order is delimited with a line of 40 hyphens (-).

```
if ($sc_send_order_to_log =~ /yes/i) {
  open (ORDERLOG, ">>$sc_order_log_file");
  print ORDERLOG "-" x 40;
  print ORDERLOG $text_of_cart;
  print ORDERLOG "-" x 40 . "\n";
  close (ORDERLOG);
}
```

Then, the user is notified that the order was a success.

```
  print "<H2>Your Order Has Been Sent!</H2>\n";
} else {
```

If the required fields were not filled in, then the user is notified that the order was not sent.

```
   print "<H2>Your Order Has Not Been Sent!</H2>\n";
}
```

Finally, the HTML footer for the process order page is printed and the subroutine is done.

```
print "<HR>\n";
print qq!
</BODY>
</HTML>
!;
} # End of process_order_form
```

Calculate_final_values Subroutine

The **calculate_final_values** routine calculates the shipping, sales tax, and discount logic. It also applies these values to the subtotal in order to produce a genuine grand total for the cart. There are several parameters passed to the script that are needed to determine the calculations. These parameters include the current subtotal, the total quantity of items in the cart, the total measured quantity of the measured field specified in the Setup file, and the status of whether we are calculating these values at the order form or at the point where the contents of the order form are being processed by the Web store.

```
sub calculate_final_values {
  local($subtotal,
        $total_quantity,
        $total_measured_quantity,
        $are_we_before_or_at_process_form) = @_;
```

The following lines of code specify local variables that will be used throughout the processing of this subroutine.

```
local($temp_total) = 0;
local($grand_total) = 0;
local($final_shipping, $shipping);
local($final_discount, $discount);
local($final_sales_tax, $sales_tax);
```

$calc_loop is set to zero. This variable keeps track of looping through the calculations in order to see whether shipping, sales tax, or discount should be calculated at a time where the value of **$calc_loop** reaches either 1, 2, or 3. A description of this logic is provided in Chapter 8.

```
local($calc_loop) = 0;
```

$temp_total is initialized to equal the subtotal of the cart. This total is updated after every cycle through the calculation loop in order to determine what the final grand total is.

```
$temp_total = $subtotal;
```

The reason that there are three cycles of calculation is that there are three things to calculate: shipping, discount, and sales tax. It is conceivable that all three of these values may need to be calculated separately and applied to the subtotal after each calculation. Examples of this situation are given in Chapter 8. The for loop given below starts the three cycles.

```
for (1..3) {
```

At the beginning of the loop, the calculated values of the shipping, discount, and sales tax are set to zero. **$calc_loop** is set equal to the current loop count.

```
    $shipping = 0;
    $discount = 0;
    $sales_tax = 0;
    $calc_loop = $_;
```

In addition to the calculation logic being different depending on the cycle number of the loop, the logic is also dependent on whether the order form is being displayed or the contents of the order form have been submitted and are

currently being processed. The reason this matters is that some of the calculations may be dependent on values that have been entered onto the form. For example, shipping costs may depend on the user selecting a different shipping type.

```
if ($are_we_before_or_at_process_form =~
     /before/i) {
```

The following code calculates all the items if the routine has been called before the process order form (at the order form display). For each of the possible values to calculate, the Setup file variables that configure which cycle of the loop the values are calculated in are compared. If the cycle number (**$calc_loop**) matches the Setup variable, then that particular type of calculation is performed.

```
if ($sc_calculate_discount_at_display_form ==
    $calc_loop) {
  $discount =
    &calculate_discount($temp_total,
      $total_quantity,
      $total_measured_quantity);
} # End of if discount gets calculated here
if ($sc_calculate_shipping_at_display_form ==
    $calc_loop) {
  $shipping =
    &calculate_shipping($temp_total,
      $total_quantity,
      $total_measured_quantity);
} # End of shipping calculations
if ($sc_calculate_sales_tax_at_display_form ==
    $calc_loop) {
  $sales_tax =
    &calculate_sales_tax($temp_total);
} # End of sales tax calculations
```

The same logic that is explained above is used below, the only exception that the calculations are done at the stage where the order form is being processed instead of being displayed.

```
} else {
 if ($sc_calculate_discount_at_process_form ==
     $calc_loop) {
   $discount =
     &calculate_discount($temp_total,
       $total_quantity,
```

```
            $total_measured_quantity);
      } # End of if discount gets calculated here
      if ($sc_calculate_shipping_at_process_form ==
          $calc_loop) {
        $shipping =
          &calculate_shipping($temp_total,
            $total_quantity,
            $total_measured_quantity);
      } # End of shipping calculations
      if ($sc_calculate_sales_tax_at_process_form ==
          $calc_loop) {
        $sales_tax =
          &calculate_sales_tax($temp_total);
      } # End of sales tax calculations
    } # End if we are before or at process order form
```

Finally, for this cycle only, the new temporary total is calculated. In addition, the final discount, shipping, and sales tax values are assigned to other variables if the values have been calculated during this cycle. This assignment is done because the values of the calculations are lost every time the cycle starts again.

```
    $final_discount = $discount if ($discount > 0);
    $final_shipping = $shipping if ($shipping > 0);
    $final_sales_tax = $sales_tax if ($sales_tax > 0);
    $temp_total = $temp_total - $discount
                    + $shipping + $sales_tax;
  } # End of $calc_loop
```

The last thing the routine does is return the final calculated values along with the new grand total.

```
   return ($final_shipping,
          $final_discount,
          $final_sales_tax,
          &format_price($grand_total));
} # end of calculate_final_values
```

Calculate_shipping Subroutine

The **calculate_shipping** subroutine calculates the shipping by taking the shipping logic variables and passing them by reference to a general logic calculation subroutine defined further down in this library. The **calculate_general_logic** routine is also passed all the parameters it needs to perform calculations such

as the subtotal, total quantity of items in the cart, and the total measured quantity of the measured field in the cart.

```
sub calculate_shipping {
  local($subtotal,
        $total_quantity,
        $total_measured_quantity) = @_;

  return(&calculate_general_logic(
           $subtotal,
           $total_quantity,
           $total_measured_quantity,
           *sc_shipping_logic,
           *sc_order_form_shipping_related_fields));
} # End of calculate_shipping
```

Calculate_discount Subroutine

The **calculate_discount** routine is defined the same way as the **calculate_shipping** routine described above. The only difference is that references to the discount logic variables are passed to the **calculate_general_logic** routine instead of to shipping-logic-related variables.

```
sub calculate_discount {
  local($subtotal,
        $total_quantity,
        $total_measured_quantity) = @_;

  return(&calculate_general_logic(
           $subtotal,
           $total_quantity,
           $total_measured_quantity,
           *sc_discount_logic,
           *sc_order_form_discount_related_fields));
} # End of calculate_discount
```

Calculate_general_logic Subroutine

The **calculate_general_logic** subroutine calculates shipping or discount values by being passed references to their criteria along with values that affect these

calculations such as the subtotal, total quantity of items in the cart, and total measured quantity of the measured field in the cart. ***general_logic** is a reference to an array the contains the criteria used to apply the calculation. ***general_related_form_fields** is a reference to an array containing information about form field values that may affect the criteria in the general logic array.

```
sub calculate_general_logic {
  local($subtotal,
        $total_quantity,
        $total_measured_quantity,
     *general_logic,
        *general_related_form_fields) = @_;
```

The following lines of code declare local variables that will be used throughout this routine.

```
  local($general_value);
  local($x, $count);
  local($logic);
  local($criteria_satisfied);
  local(@fields);
```

@related_form_values is an array that is assigned the values of all the form fields in the **@general_related_form_fields** array using a **foreach** loop to go through all the elements in the array. **$count** is used to keep track of which related form value is currently being assigned.

```
  local(@related_form_values) = ();

  $count = 0;
  foreach $x (@general_related_form_fields) {
    $related_form_values [$count] = $form_data{$x};
    $count++;
  }
```

Each element of the general logic array is processed using another **foreach** loop that places the current logic being processed into the **$logic** variable.

```
  foreach $logic (@general_logic) {
```

The routine starts off assuming that the criteria has been satisfied.

```
$criteria_satisfied = "yes";
```

Next, the definition of the logic being checked currently is split out of the **$logic** variable into the **@fields** array based on the pipe symbol (|) that delimits the logic fields.

```
@fields = split(/\|/, $logic);
```

All the form variables are cycled through to see if any of the criteria matches the form values from the **@related_form_values** array. If any criteria are not satisfied, then **$criteria_satisfied** is set to "no".

```
for (1..@related_form_values) {
  if (!(&compare_logic_values(
        $related_form_values[$_ - 1],
        $fields[$_ - 1]))) {
     $criteria_satisfied = "no";
  }
} # End of loop through form values
```

Now that the related form values have been checked, they are shifted off of the criteria fields array (**@fields**). As each criteria field is applied, it gets shifted off so that the **compare_logic_values** routine that is called below can be called with the first element of the array each time.

```
for (1..@related_form_values) {
  shift(@fields);
}
```

The script is now ready to deal with comparing the general logic with the totals (subtotal, total quantity, total measured quantity). The first field in the logic array after the related form variables fields have been shifted off is the subtotal comparison field. The subtotal is compared against this criteria field. If the match fails, then **$criteria_satisfied** is set to "no."

```
if (!(&compare_logic_values(
      $subtotal,
```

```
      $fields[0]))) {
   $criteria_satisfied = "no";
}
```

The field is shifted off and the next comparison is done. This comparison is based on the total quantity of items.

```
shift (@fields);

if (!(&compare_logic_values(
      $total_quantity,
      $fields[0]))) {
   $criteria_satisfied = "no";
}
```

Again, a field is shifted off the array. This time, the criteria that is left on the array is compared against the total measured quantity value.

```
shift (@fields);
if (!(&compare_logic_values(
      $total_measured_quantity,
      $fields[0]))) {
   $criteria_satisfied = "no";
}
```

The last piece of criteria is shifted off the Fields array. After this occurs, the only item left on the **@fields** array will be the cost to calculate if the **$criteria_satisfied** flag is equal to "yes."

```
shift (@fields);
```

If **$criteria_satisfied** is equal to "yes," then the value of the calculation is calculated. If this value has a percent sign, then the value is calculated based on the subtotal that was passed to this routine. Otherwise, the actual value is considered to be the new calculated value.

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

  if ($fields[0] =~ /%/) {
    $fields[0] =~ s/%//;
```

```
        $general_value = $subtotal * $fields[0] / 100;
      } else {
        $general_value = $fields[0];
      }
    }

  } # End of foreach loop through shipping logic
```

Now that the shipping logic has been calculated, it is returned as a formatted value to the calling subroutine.

```
  return(&format_price($general_value));
} # End of calculate_general_logic
```

Calculate_sales_tax Subroutine

The **calculate_sales_tax** routine takes the current subtotal as a parameter and calculates the sales tax on the basis of that value plus any configuration variables that affect sales tax in the Setup file.

```
sub calculate_sales_tax {
  local($subtotal) = @_;
```

$sales_tax is defined as a local variable and initialized to **0** before the script performs any processing on it.

```
  local($sales_tax) = 0;
```

If the sales tax is dependent on a form variable, then the routine checks the value of that form variable against the possible values that have been designated in the **@sc_sales_tax_form_variable** array. A successful match results in the sales tax being calculated.

```
  if ($sc_sales_tax_form_variable ne "") {
    foreach $value (@sc_sales_tax_form_values) {
      if ($value =~
          /$form_data{$sc_sales_tax_form_variable}/i) {
        $sales_tax = $subtotal * $sc_sales_tax;
```

```
      }
    }
```

If the sales tax is not form variable-dependent, then the sales tax is always calculated.

```
  } else {
    $sales_tax = $subtotal * $sc_sales_tax;
  }
```

The routine ends by returning the formatted sales tax value.

```
  return (&format_price($sales_tax));
} # End of calculate sales tax
```

Compare_logic_values Subroutine

The **compare_logic_values** subroutine takes a value and checks whether it falls within the range or is equal to a comparison value. These two variables are the parameters to this subroutine (**$input_value** and **$value_to_compare**). The routine returns a **1** if the value is in the range or a **0** if the value does not match the value/range to compare.

```
sub compare_logic_values {
  local($input_value, $value_to_compare) = @_;
```

$lowrange and **$highrange** are declared as local variables. They will be used to break the comparison value into the lower limit and upper limit of a range of values to compare if the comparison value is a range of numbers rather than a straight value. An example of a range value is **10-20**, which would mean that the input value has to be between 10 and 20. In this case, **$lowrange** would be set equal to **10** and **$highrange** would be set equal to **20**.

```
  local($lowrange, $highrange);
```

If the value to compare is a range of values signified by having a hyphen, then a range comparison is done with the input value.

```
if ($value_to_compare =~ /-/) {
```

The low and high end of the range are split by the hyphen into the **$lowrange** and **$highrange** variables respectively.

```
    ($lowrange, $highrange) = split(/-/,
$value_to_compare);
```

If the low range does not have a value, it means that the range is open-ended, so the routine assumes that the high range was entered and it only compares against this high-range value. An example of an open-ended range that looks like this is **-10**.

```
    if ($lowrange eq "") {
      if ($input_value <= $highrange) {
        return(1);
      } else {
        return(0);
      }
```

Alternatively, the high range could be missing instead of the low range. In this case, the match is still considered open-ended, but the low range is compared instead of the high range. An example of an open-ended range that looks like this is **5-**.

```
    } elsif ($highrange eq "") {
      if ($input_value >= $lowrange) {
        return(1);
      } else {
        return(0);
      }
```

If both the low and high range have values, then the value is compared against both ranges.

```
    } else {
      if (($input_value >= $lowrange) &&
         ($input_value <= $highrange)) {
        return(1);
      } else {
        return(0);
      }
    }
```

The last case that arises is when the value to compare is not a range. Thus, a straight pattern match compare is done with the values.

```
   } else {
    if (($input_value =~ /$value_to_compare/i) ||
        ($value_to_compare eq "")) {
      return(1);
    } else {
      return(0);
    }
  }
} # End of compare_logic_values
```