# CHAPTER THIRTEEN

# THE KEYWORD SEARCH LIBRARY

This subroutine library is used to handle keyword search requests for the HTML-based Web store. Its job is to search every HTML product file for occurrences of the keywords specified in the keyword input field and create an HTML page containing links to every one of those pages.

Thus, it creates an intermediate dynamically generated HTML list of product pages.

## Defining the Operating Environment

```
sub html_search
  {
```

The subroutine begins by defining a few important variables coming in as form data:

- **$keywords** will be set equal to the string entered by the customer in the TEXT field form input box named **'keywords'**.
- **$exact_match** will be set either to on or null depending on whether or not the customer clicked the checkbox named **exact_match**. It is not necessary for an administrator to even have this checkbox on the form. If the administrator does not include this option on the form, the script will simply search according to whole-word matching.

- **$case_sensitive** will similarly be set to on or null depending on whether the customer has clicked the checkbox named **case_sensitive**. Again, the administrator may choose to not give the customer the option if she so chooses.

```
$keywords = $form_data{'keywords'};
$exact_match = $form_data{'exact_match'};
$case_sensitive = $form_data{'case_sensitive'};
```

Then, **@keyword_list** is created by splitting the **$keywords** string on every occurrence of a space. Thus, all the keywords may be individually checked.


This script searches according to the "and" methodology. That is, every word (defined as characters separated by white spaces) must appear on the page searched for it to register a hit. The script does not support "or" searching.

```
@keyword_list = split(/\s+/, $keywords);
```

# Displaying the HTML Header

The HTML header for the results page is sent to the browser:

```
print qq!
<HTML>
<HEAD>
<TITLE>Search Results</TITLE></HEAD>
<BODY BGCOLOR = "FFFFFF" TEXT = "000000">
<UL>!;
```

# Performing the Search

Before going further, let us step back and see what the script needs to do in order to perform a keyword search. The routine needs to traverse the directory structure under **$root_web_path** and in doing so, also parse the HTML files to see if they have the keywords we are searching for and, if a match is found, it must determine what the HTML titles are in order to build of a list of successful hits for the client.

As the script goes down a directory looking for entries, if it finds that one of those entries is a directory, then that directory is opened. This directory becomes the new directory to traverse. In order to keep traveling down the directory tree, the script needs to keep track of where it has been. An array called **@dirs** keeps track of thissearch by containing the already open directory names that the script has not yet finished searching. As a directory gets opened for searching, it is appended as a new

element to the end of the **@dirs** array.

The following code sets up the initial variables for the algorithm described above.

- **@dirs** is an array of directories that is used as a placeholder for going back up the directory tree when we run out of files to read in a subdirectory.
- **$cur_dir** is the current directory number as a reference to the element in @dirs for which directory we are currently reading. The directory handles in this program are referred to as the string **DIR** followed by the current directory number indicated by **DIR$cur_dir**.
- **number_of_hits** is the current number of successful hits found while searching the files. The number of hits is equal to the number of files that will be returned as matches for the keyword terms:

```
@dirs = ($sc_root_web_path);
$number_of_hits = 0;
$cur_dir = 0;
```

We initialize the process by opening the directory handle using the reference **DIR$cur_dir** and the path that has been passed to the **@dirs** array. **$end_of_all_files** is a flag that when set to 1, will stop the searching routine since it means that we have finished searching every file in every directory that we can search.

```
  $end_of_all_files = 0;
  opendir("DIR$cur_dir", $dirs[$cur_dir]);
```

The following while loop does not exit until the script is through searching all of the files. Within this top level while loop, there is a second level while loop

that goes on forever unless the **last** command is encountered inside. It is inside this second while loop that the directory tree for HTML documents is traversed:

```
while (!($end_of_all_files))
  {
  while (1)
    {
```

First, the script grabs a reference to the next valid directory or filename into the **$filename** variable. Next, **$fullpath** is set to the current path plus filename:

```
$filename = &GetNextEntry("DIR$cur_dir",
            $dirs[$cur_dir]);
$fullpath = "$dirs[$cur_dir]/$filename";
```

Then, for the entry that was received, the routine goes through multiple cases and does dwhat is appropriate for those cases. Five basic cases are discussed further below.

## Case 1: No More Files in Current Directory

In case 1, the file is NULL, but since there are still entries in the **@dirs** variable, the program goes back up the directory tree and continues searching in a previous directory where it left off. Specifically, this involves closing the current directory, subtracting one from the **$cur_dir** variable and then issuing a **next** command to force another iteration through the WHILE(1) loop again:

```
if (!($filename) && $cur_dir > 0)
  {
 closedir("DIR$cur_dir");
 $cur_dir--;
 next;
 }
```

## Case 2: The End of the Search

In case 2, there are no more filenames to search on, but the script has already been through all the previous entries in the **@dirs** array. Thus, the search must end. This is done by closing the current directory handle, setting the **$end_of_all_files** to one, and issuing the **last** command to break completely out of the WHILE(1) loop:

```
if (!($filename))
  {
   closedir("DIR$cur_dir");
   $end_of_all_files = 1;
   last;
   }
```

## Case 3: The File is a Directory

Case 3 discovers that the filename is actually a directory, so the script descends down into the directory if it is both readable and executable. The program checks if the file is a directory using the **-d** flag. It also checks for Readability and Execute rights by using the **-r** and **-x** flags.

Finally, the program goes down the directory tree if the filename is a directory, by incrementing the current directory counter, **$cur_dir**, by one, pushing a new path onto the **@dirs** array, and opening a new directory handle. Finally, the **next** command is used to force the script to go back to the top of the WHILE(1) loop:

```
  if (-d $fullpath)
        {
           if (-r $fullpath && -x $fullpath)
             {
             $cur_dir++;
             $dirs[$cur_dir] = $fullpath;
             opendir("DIR$cur_dir", $dirs[$cur_dir]);
             next;
             }
           else
             {
             next;
             }
           } # End of Case 3 (File is directory
```

## Case 4: The File Is Unwanted

In Case 4, the script checks to see if the file about to be searched is actually unwanted. The program starts by setting the **$unwanted_file** flag to 0. Then, each unwanted file in the **@unwanted_files** array is gone through and checked if the filename and path is unwanted by doing a pattern match against it. If it

is, then the **$unwanted_file** flag is set to 1. Finally, after all the **@unwanted_files** have been checked, if the **$unwanted_file** flag is equal to 1, the **next** command is issued to reiterate through the WHILE(1) loop again:

```
$unwanted_file = 0;
foreach (@sc_unwanted_files)
   {
      if ($fullpath =~ /"$_"/)
     {
        $unwanted_file = 1;
        }
      } # End of foreach unwanted files
if ($unwanted_file)
      {
      next;
      } # End of Case 4 Unwanted File
```

## Case 5: The File Must Be Searched

In this, the last case, the script finds out that the file really is a file that we want to search for keywords in. The **-r** flag is used to check if the file is readable and if it is, the **last** command is issued in order to force a breakout of the WHILE(1) loop. Breaking out of this loop will allow the script to move on and search through the file:

```
      if (-r $fullpath)
        {
       last;
       } # Make sure the file is readable
    } # End of While (1)
```

After the WHILE(1) loop, we check again for the **$end_of_all_files** flag. If it is not set equal to one then the script can continue the file searching:

```
if (!($end_of_all_files))
  {
```

When we search a file, we initially set **@not_found_words** equal to the array of keywords we want to search. This corresponds to the idea that initially all of the words are not found. As we search the file and find keywords later on, those keywords will be deleted from the **@not_found_words** array. When the

**@not_found_words** array has no elements left in it, we know that all the keywords were found in the file and that we have found a hit:

```
@not_found_words = @keyword_list;
```

In addition to searching for the keyword, we will attempt to parse out the name of the title of the HTML file. The **$are_we_in_head** flag is initially set to zero. If it is zero, we know that we are in the header of the HTML file still. Upon reaching a </HEAD> or </TITLE> flag, the script knows that it is done reading the header. The header is read into the **$headline** variable.

```
$are_we_in_head = 0;
     open(SEARCHFILE, $fullpath);
     $headline = "";
     while(<SEARCHFILE>)
       {
       $line = $_;
       $headline .= $line if ($are_we_in_head == 0);
       $are_we_in_head = 1
         if (($line =~ m!</head>!i) || ($line =~
            </title>!i));
```

The **&FindKeywords** subroutine performs the actual searching of the keywords in each line as it is read in from the file. When the **&FindKeywords** subroutine finds a match, it deletes the keyword from the **@not_found_words** array:

```
&FindKeywords($exact_match, $case_sensitive,
                        $line, *not_found_words);
       } # End of SEARCHFILE
     close (SEARCHFILE);
```

# Displaying the Search Hits

If the **@not_found_words** array is less than 1, the script knows that all the keywords were found, so it prints out the matched files. Part of the routine that prints out the match consists of parsing the title of the document out of the HTML code stored in **$headline**:

```
if (@not_found_words < 1)
     {
```

The first thing the routine does is replace all newlines with spaces in **$headline**. Then, it sets up a match against the regular expression <title>(.*)</title>. This expression matches for zero or more characters between the <TITLE> HTML tags. In Perl, the successful match will make the variable **$1** equal to the characters between the <TITLE> tags. The **i** at the end of the match expression indicates that the match is done without regard to case. If the title turns out not to exist in this document, the **$title** variable is set to **"No Title Given"**.


We use a special form of the match operator below. Most of the time we use **/**'s to indicate the endpoints of a search. Here, we use the m (match) operator followed by a different character to use as our matching operator. In this case, we use the exclamation point (**!**) to delimit the search. The reason we do this is because we are including **/**'s inside the actual expression to search, and escaping them with the backslash (****) would look messy.

```
    $headline =~ s/\n/ /g;
    $headline =~ m!<title>(.*)</title>!i;
    $title = $1;

  if ($title eq "")
   {
   $title = "No Title Given";
   }
```

The program then strips out the **$root_web_path** because it contains information we do not want to pass to the user about the internal directory structure of the Web server. Finally, the script prints out the HTML code related to the hit that we have found and increments the hit counter.

```
    $fullpath =~ s!$sc_root_web_path/!!;
    &PrintBodyHTML($fullpath, $title);
    $number_of_hits++;

    } # If there are no not_found_words
  } # If Not The End of all Files
} # End of While Not At The End Of All Files
```

If there were no results found, the HTML for "getting no hits" is printed out.

```
if ($number_of_hits == 0)
  {
  &PrintNoHitsBodyHTML;
  }
```

```
  print qq!</UL></BODY></HTML>!;
  } # end of subroutine
```

# FindKeywords Subroutine

The **FindKeywords** subroutine is the core routine of the entire search engine and is called with the following syntax:

```
&FindKeywords("on", "on", $line, *not_found_words);
```

As you can see, the subroutine accepts a line of a file and the keywords to search for in that line. If a keyword is found, the routine splices it out of the keyword array (**@not_found_words**). Thus, when the **@not_found_words** array no longer has any elements in it, the script knows that all the keywords have been found in the file:

```
sub FindKeywords
  {
```

There are four parameters. The first, **$exact_match**, is equal to **on** if the type of pattern match we are doing is based on an exact one to one match of each letter in the keyword to each letter in a word contained in the HTML document. Likewise, the **$case_sensitive** variable is on if the customer requested case sensitivity in the search. The third parameter, **$line**, is a line in the HTML file that is currently being searched for the keywords. The fourth and final parameter, ***not_found_words**, is a reference to the array **@not_found_words**, which contain a list of all the keywords not found so far. As keywords get found in the searched file, this array has its words removed. Thus, when the array is empty, we know the file contained all the keywords. In other words, there are no "not found words" if the search is successful:

```
  local($exact_match, $case_sensitive,
$line, *not_found_words) = @_;
  local($x, $match_word);
```

If the exact match and case sensitivity are on, then the program matches all the words in the array by surrounding the keywords with **\b**. This means that the keyword has to be surrounded by word boundaries in order to be a valid

match. Thus, the keyword *the* would not match a word like *there* , since *the* is only part of a larger word:

```
if ($case_sensitive eq "on")
  {
  if ($exact_match eq "on")
    {
    for ($x = @not_found_words; $x > 0; $x--)
      {
      $match_word = $not_found_words[$x - 1];
      if ($line =~ /\b$match_word\b/)
        {
```

The **splice** routine used below cuts out the words if they satisfy the search. The **splice** command is a Perl routine that accepts the original array, the element in the array to splice, the number of elements to splice, and a list or array to splice into the original array. Since we are leaving off the fourth parameter of the splice, the routine by default splices "nothing" into the array as the element number. This deletes the element in one convenient little routine:

```
        splice(@not_found_words,$x - 1, 1);
        }
      } # End of for ($x = @not_found_words...
    }
```

If the exact match is not on, then the program will match simply on the basis of the letters in the keyword existing anywhere on the line regardless of if that keyword is part of a larger word or not:

```
else
  {
  for ($x = @not_found_words; $x > 0; $x--)
    {
    $match_word = $not_found_words[$x - 1];
    if ($line =~ /$match_word/)
    {
      splice(@not_found_words,$x - 1, 1);
      } # End of If
    } # End of for ($x = @not_found_words...
  } # End of ELSE
}
```

Next, handle the case-insensitive situation with the exact same routines but performing searches with case-insensitive as indicated by the **i** given after the slashes defining the search term:

```
  else
  {
  if ($exact_match eq "on")
    {
    for ($x = @not_found_words; $x > 0; $x--)
      {
      $match_word = $not_found_words[$x - 1];
      if ($line =~ /\b$match_word\b/i)
      {
        splice(@not_found_words,$x - 1, 1);
        } # End of If
      } # End of for ($x = @not_found_words...
    }
  else
    {
    for ($x = @not_found_words; $x > 0; $x--)
      {
      $match_word = $not_found_words[$x - 1];
      if ($line =~ /$match_word/i)
        {
        splice(@not_found_words,$x - 1, 1);
        } # End of If
      } # End of For Loop
    } # End of ELSE
  }
} # End of FindKeywords
```

# GetNextEntry Subroutine

The **GetNextEntry** subroutine reads the directory handle for the next entry in the directory. The routine accepts the current directory handle and the current directory path as parameters and is called with the following syntax:

```
&GetNextEntry(DIRECTORY_HANDLE, "directory_name");
```

The code is explained below:

```
sub GetNextEntry
  {
local($dirhandle, $directory) = @_;
```

If the next entry is a file, the program checks to see if the file has a **.htm** or **.html** extension. This is accomplished by using the regular expression **/htm.?/i**. The **.?** matches any character once after the **htm**. The **i** after the search terms, tells the program to treat upper- and lower-case characters equally:

```
while ($filename = readdir($dirhandle))
  {
  if (($filename =~ /htm.?/i) ||
   (!($filename =~ /^\.\.?$/) && -d
   "$directory/$filename"))
    {
```

If the program satisfies one of these two conditions, the while loop that reads in subsequent directory entries is exited with the **last** command and the found filename/directory name is returned from the subroutine:

```
 last;
 } # End of IF Filename is html document or a directory
} # End of while still stuff to read
```

The filename will be valid if it is a directory or an HTML file

```
 $filename;
 } # End of GetNextEntry
1;
```