CHAPTER 5

Using CGI-LIB.PL

OVERVIEW

There are certain tasks that every CGI program must be capable of doing in order to perform Common Gateway Interface (CGI) functions. For example, form variables need to be read into the program, and specially formatted information must be communicated back to the server. Although there are several good libraries of CGI-related routines, all the scripts in this book use **CGI-LIB.PL**: It is small, efficient, Perl 4–compatible, and well supported throughout the CGI and Perl programming community.

CGI-LIB.PL was written by Steven Brenner and is the de facto standard library of CGI programming routines for Perl 4 and above. The library is short and simple, and it does 99 percent of what a CGI programmer needs to accomplish with regard to the Common Gateway Interface specification. An examination of this well-written library reveals many interesting Perl techniques.

The main thing that **CGI-LIB.PL** does is to read all the form variable input into an associative array for picking out the values. It also has the capability of printing standard HTML headers and footers along with the magic "Content-type: text/html\n\n" header. This header is absolutely necessary and printing it is generally the first thing every CGI programmer does in a script. CGI-LIB also has small routines to do general housekeeping, such as printing error messages as HTML output, printing associative arrays, and returning URLs and certain environmental variables related to CGI. Advanced features, such as the ability to support file uploads, have recently been added.

INSTALLATION AND USAGE

As with most of the libraries in this section, **CGI-LIB.PL** is easy to install. Merely insert a line to `require` **cgi-lib.pl,** such as the following:

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

Reading and Parsing Form Variables

The `&ReadParse` function is used to read the form variables whether the form was called via a `GET` or by a `POST` method. If `&ReadParse` is called without a parameter, by default the form variables are read into an associative array called `%in`. You can use your own associative array if you pass it by reference to `&ReadParse`. For example, if you wanted to read the form variables into an associative array called `%form_data`, you would use this convention:

```
&ReadParse(*form_data);
```

> **NOTE** Normally, when a variable is passed to a subroutine in Perl, only a copy of its value is passed. By replacing the $, @, or % symbol with a * in front of normal scalar, array, and associative array variables respectively, you are telling it to copy the location in memory where the variable value exists. This way, when the variable is changed in the subroutine, that change is simultaneously done to the originally passed variable instead of a mere copy of the value of the variable.

> **While many CGI scripts use the default `%in` associative array to represent data, it is also common to see CGI programmers using a different associative array name. For example, many of the scripts discussed in this book use `%form_data` to hold form variable information. Both have their advantages. `%in` is a concise way of saying "input form variables" and is really quick to type. However, `%form_data` is more readable to some CGI authors especially when the script uses several associative arrays.**

NOTE

Uploading Files Using HTML

CGI-LIB.PL allows you to upload files using a new HTML input tag that specifies a filename to upload. Triggering the Web browser to upload a file takes several steps. First, the FORM METHOD must be POST. Second, a part in the FORM tag must say ENCTYPE=multipart/form-data. With this encoding type, whole files that are sent by the browser are broken apart using boundary information sent by the browser. Normally, by default, the encoding type is application/x-www-form-urlencoded, which means that the form variables are posted in roughly the same way as they would have been had they been part of the command line using the GET method. This common form of encoding is discussed in more detail in Chapter 2 where we discuss how to fool the CGI program into thinking that information has been posted to it. In addition, the new INPUT tag lets you specify TYPE=FILE to associate a filename with a particular INPUT FORM variable. When the file gets uploaded, **CGI-LIB.PL** has the capability of parsing the information automatically and downloading it. The following sample HTML code would let you perform a file upload:

```
<HTML><HEAD>
<TITLE>File Upload</TITLE>
</HEAD>
<BODY>
<FORM METHOD=POST ENCTYPE=multipart/form-data ACTION="upload.cgi">
<P>
<INPUT TYPE=FILE NAME="upload_file"><P>
<INPUT TYPE=SUBMIT VALUE="Do The Upload">
</FORM>
</BODY></HTML>
```

In addition to all the changes in the HTML form, you must make changes in the CGI program. Certain **CGI-LIB.PL** variables must be changed to allow the files to be uploaded. The following code snippet from **CGI-LIB.PL** lists the relevant variables that need to be changed:

```
$cgi_lib'maxdata    = 131072;
   # maximum bytes to accept via POST - 2^17
$cgi_lib'writefiles =     0;
   # directory to which to write files, or
   # 0 if files should not be written
$cgi_lib'filepre    = "cgi-lib";
   # Prefix of file names, in directory above
```

> **NOTE**
> Many variables in CGI-LIB.PL are prefixed with "cgi_lib" and an apostrophe. This tells Perl to distinguish these variables in their own package. Basically, the variables become global to the CGI-LIB.PL script, but are not seen or changed in the main CGI script unless you explicitly reference them with the "cgi_lib'" prefix. Because of this prefix, the likelihood that these variables will be mistakenly defined in another part of the CGI script, thus causing a naming conflict, becomes almost null.

`$cgi_lib'maxdata` should be set greater than 131072 bytes if you expect larger files to be uploaded. `$cgi_lib'writefiles` needs to be set to a subdirectory on your server where you can create and write to files. `$cgi_lib'filepre` should be set to a standard prefix to append to filenames that get written in the CGI write files directory. The default is `cgi-lib`.

It should be noted that CGI-LIB creates a temporary name for each file upload instead of using the name specified by the user. One reason is that a filename from one operating system may not be compatible with your server's file system naming conventions. More important, there is a possibility that someone could maliciously overwrite an important file if he or she had full control over the naming space.

CGI-LIB.PL creates a temporary name for each file upload instead of using the name specified by the user. There are several reasons CGI-LIB.PL uses a different, computer-generated name. First, the filename from one operating system may not be compatible with the way filenames are stored on your Web server. Secondly, there is a possibility that someone could maliciously overwrite a file that was important to you on your Web server if the user was allowed to truly specify the filename.

NOTE

The chapter in this book regarding the Web Message Board System (BBS) provides a full example of using the file upload parameters. The BBS allows users to upload files as virtual "attachments" to their posted messages on the system.

you are using an operating system other than UNIX, the Perl you are using may require you to use the command "binmode(STDIN);" before you call the ReadParse function to download the file and other form information. The reason for this is that some operating systems do "ascii" conversions when files are read in by default. For example, one dialect of Windows NT Perl actually takes the "\r\n" combinations that delimit a boundary and translates them to "\n". Unfortunately, this really messes up the CGI-LIB.PL parsing routine since it is explicitly looking for "\r\n" combinations. Setting the STDIN file handle to binary mode using Perl's binmode command solves this problem.

WARNING

Printing HTML Information

&PrintHeader returns the standard "Content-type: text/html\n\n" header that every CGI program needs to print in order for the browser to recognize the type of information that is being printed. &HtmlTop prints the HTML head of a document and uses a passed parameter to determine

how the `<TITLE>` and `<H1>` tags should be printed. `&HtmlBot` prints the HTML footer of a document. Both routines save time by printing standard HTML codes that normally are time-consuming to write over and over again for each document. Here is a sample usage of these three functions:

```
print &PrintHeader;
print &HtmlTop("The title of my page";
print &HtmlBot;
```

The above code, would result in the CGI program presenting the following output to the Web server. Note that by the time the Web browser displays this information, the `Content-type: text/html\n\n` message is stripped off so that the user only sees straight HTML code.

```
Content-type: text/html

<HTML>
<HEAD>
<TITLE>The title of my page</TITLE>
</HEAD>
<BODY>
<H1>The title of my page</H1>
</BODY>
</HTML>
```

Splitting Multiple-Valued Form Variables

If your form variable contains multiple values, an associative array will not help you, because only one value can be stored in an associative array for each key. Brenner gets around this problem by storing all the values for a key in one associative array entry, but he separates them with the `\0` character. Passing the value that you believe to contain multiple values to the `&SplitParam` routine splits the multiple-valued key and returns an array list of all the values. The following example would split the string "test\0test1\0test2" into an array list of "test," "test1," and "test2."

```
@split_values = &SplitParam("test\0test1\0test2");
```

Testing Which Form Method Was Used

`&MethGet` and `&MethPost` allow you to test which method was used when your CGI program was called. They return a true value if a particular method was used. If the following snippet of CGI code were called using the POST method, it would print "hello."

```
if (&MethPost) {
    print "Hello\n";
} # End of if &MethPost
```

Retrieving the Current URL

`&MyBaseURL` and `&MyFullURL` return the URL of the current script. `&MyFullURL` returns the complete URL, including form variables used in the GET method. `&MyBaseURL` returns just enough of the URL to include the basic script name. For example, if our script were called as `http://mycompany.com/test.cgi?last_name=smith`, then the script shown next would return the following pieces of information: `http://mycompany.com/test.cgi` and `http://mycompany.com/test.cgi?last_name=smith`.

```
print &MyBaseURL;
print &MyFullURL;
```

Printing CGI Errors

`&CgiError` and `&CgiDie` accept an error message as a parameter, convert the error message to HTML, and print it so that users can see the error on a Web browser. `&CgiDie` behaves exactly like `&CgiError` except that it exits the program with the DIE command.

Most Perl programs use the plain DIE command in case of a system failure. Using `&CgiDie` is much better in the case of CGI programs, because the DIE message frequently never gets sent to the browser, making problems difficult to troubleshoot. The following code gives an example of how you would use `&CgiDie` to trap an error if a file does not open correctly.

```
open(TESTFILE,"<test.file") ||
   &CgiDie("The File: test.file could not be opened.");
```

> **NOTE** For the code above, you could also have used `CgiError` to report the problem. However, opening a file is usually a crucial step in a program so you want the script to end cleanly instead of attempting to go on when the file could not be opened. I almost never use `CgiError`. I use `CgiDie` instead.

Printing Associative Arrays

Brenner includes a convenient routine to print the contents of an associative array. `&PrintVariables` accepts an associative array as input, goes through the keys one by one, and returns the value as HTML code for printing. The `&PrintEnv` subroutine calls the `&PrintVariables` subroutine to print the list of all the current environmental variables. In this example, `&PrintVariables` prints the `%in` associative array:

```
print &PrintVariables(*in);
```

DESIGN DISCUSSION

Brenner uses a variety of interesting Perl techniques to make his library small and efficient. Going through the inner workings of his routines goes a long way toward demonstrating how CGI and Perl work together.

We will review **CGI-LIB.PL** version 2.8, which is the latest version of CGI-LIB as of this writing. (Version 2.8 adds support for file uploads, increasing the size of the library. If you are not using file uploads in your CGI programs, you may want to use an older, more streamlined version of CGI-LIB.) For the purpose of being complete, we include Brenner's comments. Although the comments are very good, we have expanded on them. We rely so much on the library for our scripts that to customize them "like a pro," it helps to understand how CGI-LIB is doing its magic.

Main Script

The following piece of code descriptively sets the `$cgi_lib'version` variable to `$Revision: 2.8 $` for the purposes of reading the code. However, adding the `=~ s/[^.\d]//g` tells Perl to strip out the non-numeric information. Thus, `$cgi_lib'version` is actually equal to `2.8`. This allows a programmer to check for version information after requiring **cgi-lib.pl**. For example, if we knew we were enabling file upload, we might check to see that the version number was 2.8 or greater. Because all the CGI-LIB variables tend to be prefixed with `cgi-lib`, we will refer to the variables without the prefix.

```
($cgi_lib'version = '$Revision: 2.8 $') =~ s/[^.\d]//g;
```

`$maxdata`, `$writefiles`, and `$filepre` all affect file uploading in different ways. `$maxdata` is set to the maximum number of bytes that the CGI program will accept for a file upload. Note that this value is the total value that will be accepted for the entire POST process. If you are uploading multiple files, this value would have to be as large as all the files combined plus some minimal header information being sent by the browser. `$writefiles` should be set to a directory that you wish the uploaded files to go to. `$filepre` should be set to a file prefix to add to the program.

$bufsize is the maximum amount of data to handle in one pass. The `ReadParse` subroutine, discussed next, processes values in chunks of size `$bufsize`. Processing too large a string at one time can cause problems, especially if there is an error somewhere further down the stream of characters that are being interpreted. When you're processing MULTIPART data, each part is separated by an arbitrary boundary. The `$maxbound` variable is set to 100, which means that the divider can be only 100 characters long. `$headerout` is used by the `CgiError` reporting routines to see whether we have already output a "Content-type: text/html\n\n" header. Whenever the header is printed, `$headerout` is incremented by 1. Then the header will not print if the `CgiError` is called more than once to report a problem in the program.

```
# Parameters affecting cgi-lib behavior
# User-configurable parameters affecting file upload.
$cgi_lib'maxdata    = 131072;
    # maximum bytes to accept via POST - 2^17
$cgi_lib'writefiles =      0;
    # directory to which to write files, or
    # 0 if files should not be written
$cgi_lib'filepre    = "cgi-lib";
     # Prefix of file names, in directory above


# Do not change the following parameters unless you have special
# reasons
$cgi_lib'bufsize  =  8192;
# default buffer size when reading multipart
$cgi_lib'maxbound =   100;
# maximum boundary length to be encountered
$cgi_lib'headerout =    0;
# indicates whether the header has been printed
```

> **N O T E** The about variables are used primarily in the ReadParse subroutine discussed below. Understanding how these variables can be configured is the first step to understanding how ReadParse works.

ReadParse Subroutine

The following author's comment block explains how the `ReadParse` subroutine works. `ReadParse` is the heart of CGI-LIB. It takes all the form variables and reads them into an associative array. In addition, `ReadParse` processes file uploads. We will go over this routine in more detail below.

```
# ReadParse
# Reads in GET or POST data, converts it to
# unescaped text, and puts key/value pairs in %in,
# using "\0" to separate multiple selections
# Returns >0 if there was input,
# 0 if there was no input
# undef indicates some failure.


# Now that cgi scripts can be put
# in the normal file space, it is
# useful to combine both the form
```

```
# and the script in one place.  If no
# parameters are given (i.e., ReadParse
# returns FALSE), then a form could be
# output.

# If a reference to a hash is given,
# then the data will be stored in that
# hash, but the data from $in and @in
# will become inaccessible.
# If a variable-glob (e.g., *cgi_input)
# is the first parameter to
# ReadParse,
# information is stored there,
# rather than in $in, @in, and %in.
# Second, third, and fourth
# parameters fill associative arrays
# analogous to %in with data relevant to file uploads.

# If no method is given, the script
# will process both command-line arguments
# of the form: name=value and any text that is in
# $ENV{'QUERY_STRING'}
# This is intended to aid debugging
# and may be changed in future releases
```

By default, the ReadParse function generates four associative arrays. The first array, `%in`, contains a list of the form variables as keys in the associative array and their values. If another associative array is passed by reference, `ReadParse` will use this new array to store the form values. For example, many scripts in this book use %form_data instead of the `default %` in to store HTML form input.

The three other associative arrays that `ReadParse` uses as parameters have been added to deal with file upload. `%incfn` associates the client's filename with the form variable name. This value may not be provided by the browser. `%inct` is the content type of the file associated with the form variable name. This value also may not always be provided by the Web browser when uploading a file. It is important to note that because we are not guaranteed to get a filename from the browser, we must generate our own filename. This server-generated filename is stored in the `%insfn` associative array. Normal POST and GET mothods using the URL-encoding mechanism only assign form variables using the % in associate array.

```
sub ReadParse {
  local (*in) = shift if @_;     # CGI input
  local (*incfn,
          # Client's filename (may not be provided)
      *inct,
          # Client's content-type (may not be provided)
      *insfn) = @_;
          # Server's filename (for spooled files)
  local ($len, $type, $meth, $errflag, $cmdflag, $perlwarn);
```

Perl warnings are disabled in various places throughout the code, because environment variables are checked throughout the code. If we are using the Perl -w (warn) flag to execute the program at the command line, the environment variables will typically not be set to the values they would have when the program is run from the Web server. $^W is a special Perl variable that indicates the current warning level checker. $^W is set to 0 to clear it. The $perlwarn variable is used as a placeholder to set it back to $^W.

```
  # Disable warnings as this code
  # deliberately uses local and environment
  # variables which are preset to
  # undef (i.e., not explicitly initialized)
  $perlwarn = $^W;
  $^W = 0;
```

The following code takes some of the environment variables that we will be looking at and converts them to regular Perl variables:

```
 # Get several useful env variables
  $type = $ENV{'CONTENT_TYPE'};
  $len  = $ENV{'CONTENT_LENGTH'};
  $meth = $ENV{'REQUEST_METHOD'};
```

If we exceed the maximum amount of data that we are allowed to receive, then CGI-LIB will exit with an error printed to the Web browser as HTML code as well as to the STDERR error log.

```
  if ($len > $cgi_lib'maxdata) { #'
      &CgiDie("cgi-lib.pl: Request to receive too much data: $len bytes\n");
  }
```

If the method is GET, we know that the form variables are encoded as type application/x-www-form-urlencoded. Basically, this format is the same form that you see when you are calling a CGI script with variables on the URL line. For example, setting the form variable test equal to test 1 and testagain to test 2 would be encoded as test=test+1&testagain=test+2. If the method is POST, we do not necessarily know whether the POSTed data will be in the urlencoded format or the multipart/form-data format, so the last part of the following if statement checks whether the content type is equal to application/x-www-form-urlencoded.

```
if (!defined $meth || $meth eq '' || $meth eq 'GET' ||
  $type eq 'application/x-www-form-urlencoded') {
local ($key, $val, $i);
```

Because the content is in the urlencoded format, we read the form variables in the urlencoded format. If there is a QUERY_STRING environmental variable but no GET method, we can assume that the extra information in the URL is part of command-line parameters to the CGI program. Thus, $cmdflag is set to 1.

If the method is GET, the QUERY_STRING value is assumed to be the form variables themselves. If the method is POST, then STDIN is read into $in for the length of the CONTENT_LENGTH environmental variable. If none of these conditions is satisfied, CGI-LIB dies with an error.

```
# Read in text
if (!defined $meth || $meth eq '') {
  $in = $ENV{'QUERY_STRING'};
  $cmdflag = 1;  # also use command-line options
} elsif($meth eq 'GET' || $meth eq 'HEAD') {
  $in = $ENV{'QUERY_STRING'};
} elsif ($meth eq 'POST') {
    $errflag = (read(STDIN, $in, $len) != $len);
} else {
  &CgiDie("cgi-lib.pl: Unknown request method: $meth\n");
}
```

The following code splits the form variables in the urlencoded format by separating the fields on the basis of an & or ; character:

```
@in = split(/[&;]/,$in);
push(@in, @ARGV) if $cmdflag;
            # add command-line parameters
```

Certain characters cannot be represented directly in the urlencoded format. Spaces are represented as either `%20` or `+` symbols. The following routine first takes the +s and converts them to spaces. Then it splits the key and value pairs on the basis of the `=` character. The last parameter of the split statement ends the split after there are two distinct values. After all, it is possible that the value of a variable contains an `=` character.

```
foreach $i (0 .. $#in) {
  # Convert plus to space
  $in[$i] =~ s/\+/ /g;

  # Split into key and value.
  ($key, $val) = split(/=/,$in[$i],2);
      # splits on the first =.
```

The following code takes the `%xx` hexadecimal codes and converts them to ASCII. The regular expression `%([A-Fa-f0-9]{2})` matches on anything that starts with `%` and is followed by a hexadecimal digit (0 through F). The `{2}` in the expression tells the expression to search for only two hexadecimal digits. The parentheses surrounding the hex digits make it so that the variable `$1` in Perl will be automatically set equal to the found digits. The second part of the substitute expression takes the `$1` variable and converts it to an ASCII character using the `pack("c", hex($1))` command.

```
# Convert %XX from hex numbers to alphanumeric
$key =~ s/%([A-Fa-f0-9]{2})/pack("c",hex($1))/ge;
$val =~ s/%([A-Fa-f0-9]{2})/pack("c",hex($1))/ge;
```

Finally, if there is more than one value for the key, then the values become separated by the `\0` character. Then the key is set equal to the whole value in the associative array.

```
# Associate key and value
$in{$key} .= "\0" if (defined($in{$key}));
```

```
          # \0 is the multiple separator
   $in{$key} .= $val;
 }
```

The following routine processes multipart/form-data if the encoding type was sent that way. Parsing multipart/form-data is much more complicated than parsing application/x-www-urlencoded data, because it involves parsing the input buffer for special boundary markers. When the encoding type is multipart, a boundary is specified that separates each part of the form input. An arbitrary boundary marker is selected and used throughout the document. The boundary variable is preceded by two dashes (–) before each boundary specifier. The final boundary marker is also followed by two dashes. Thus, if the boundary marker is TEST, then the following is an example of boundary code:

```
--TEST
(Some Form Data)
--TEST
(More Data)
--TEST
(The last piece of data)
--TEST--
```

The following routines also check to see whether the data being sent is a filename for uploading. If it is, then the file is parsed as multipart data and written to disk.

```
  } elsif ($ENV{'CONTENT_TYPE'} =~ m#^multipart/form-data#) {
    # for efficiency, compile
    # multipart code only if needed
$errflag = !(eval <<'END_MULTIPART');

    local ($buf, $boundary, $head, @heads, $cd, $ct, $fname, $ctype, $blen);
    local ($bpos, $lpos, $left, $amt, $fn, $ser);
    local ($bufsize, $maxbound, $writefiles) =
      ($cgi_lib'bufsize, $cgi_lib'maxbound, $cgi_lib'writefiles);

    # The following lines exist solely
    # to eliminate spurious warning messages
    $buf = '';
```

The following piece of code parses the multipart header to find out what the boundary is. The boundary is designated with a `boundary=[boundary]` message, where `[boundary]` is an arbitrary choice of the Web browser. If the boundary does not exist, the program dies.

```
($boundary) = $type =~ /boundary="([^"]+)"/; #";
        # find boundary
($boundary) = $type =~ /boundary=(\S+)/
  unless $boundary;
&CgiDie ("Boundary not provided") unless $boundary;
$boundary =  "--" . $boundary;
$blen = length ($boundary);
```

The request method must be POST in order for multipart data to be read.

```
if ($ENV{'REQUEST_METHOD'} ne 'POST') {
  &CgiDie("Invalid request method for  multipart/form-data:
$meth\n");
  }
```

`$writefiles` must be set to something for there to be a valid directory to write files to. If a valid directory exists, it is checked to see whether it is a directory, is readable, and is writable. If all these conditions are true, `$writefiles` prefixes the `writefiles` area with the filename prefix for the files that will be written to the Web server if any files are being uploaded.

```
if ($writefiles) {
  local($me);
  stat ($writefiles);
  $writefiles = "/tmp"
    unless  -d _ && -r _ && -w _;
  # ($me) = $0 =~ m#([^/]*)$#;
  $writefiles .= "/$cgi_lib'filepre";
}
```

The following huge block of code parses out the boundaries and the form variables in the boundary. For our purposes, this code will be called only if we are processing a form that uses the new file upload function. If there are form variables between the boundaries, the form variables get

processed as usual. If there is a file, then the file gets processed block by
block; the default size of each block is 8192 bytes. Most of the code is
meant for processing out the filename as well as creating the new server
filename in the $writefiles directory.

```perl
    # read in the data and split into parts:
    # put headers in @in and data in %in
    # General algorithm:
    #   There are two dividers:
    #   the border and the '\r\n\r\n'
    #   between header and body.
    #   Iterate between searching for these
    #   Retain a buffer of size(bufsize+maxbound);
    #   the latter part is to ensure that dividers
    #   don't get lost by wrapping between two bufs
    #   Look for a divider in the current batch.
    #   If not found, then save all of bufsize, move
    #   the maxbound extra buffer to the front of
    #   the buffer, and read in a new bufsize bytes.
    #   If a divider is found, save everything up to
    #   the divider.  Then empty the buffer of
    #   everything up to the end of the divider.
    #   Refill buffer to bufsize+maxbound
    #   Note slightly odd organization.  Code before
    #   BODY: really goes with code following HEAD:,
    #   but is put first to 'pre-fill' buffers.  BODY:
    #   is placed before HEAD: because we first need to
    #   discard any 'preface,' which would be analogous
    #   to a body without a preceding head.

    $left = $len;
  PART:
# find each part of the multi-part while reading data
  while (1) {
    last PART if $errflag;

    $amt = ($left > $bufsize+$maxbound-length($buf)
        ?  $bufsize+$maxbound-length($buf): $left);
    $errflag = (read(STDIN, $buf, $amt, length($buf)) != $amt);
    $left -= $amt;

    $in{$name} .= "\0" if defined $in{$name};
    $in{$name} .= $fn if $fn;
```

```
    $name=~/([-\w]+)/;
  # This allows $insfn{$name} to be untainted
    if (defined $1) {
       $insfn{$1} .= "\0" if defined $insfn{$1};
       $insfn{$1} .= $fn if $fn;
    }

  BODY:
   while (($bpos = index($buf, $boundary)) == -1) {
      if ($name) {
  # if no $name, then it's the prologue -- discard
         if ($fn) { print FILE substr($buf, 0, $bufsize); }
         else     { $in{$name} .= substr($buf, 0, $bufsize); }
      }
      $buf = substr($buf, $bufsize);
      $amt = ($left > $bufsize ? $bufsize : $left);
#$maxbound==length($buf);
      $errflag = (read(STDIN, $buf, $amt, $maxbound) != $amt);
      $left -= $amt;
   }
   if (defined $name) {
  # if no $name, then it's the prologue -- discard
      if ($fn) { print FILE substr($buf, 0, $bpos-2); }
      else { $in {$name} .= substr($buf, 0, $bpos-2); }
  # kill last \r\n
   }
   close (FILE);
   last PART
      if substr($buf, $bpos + $blen, 4) eq "--\r\n";
   substr($buf, 0, $bpos+$blen+2) = '';
   $amt = ($left > $bufsize+$maxbound-length($buf)
       ? $bufsize+$maxbound-length($buf) : $left);
   $errflag = (read(STDIN, $buf, $amt, length($buf)) != $amt);
   $left -= $amt;

   undef $head;  undef $fn;
  HEAD:
   while (($lpos = index($buf, "\r\n\r\n")) == -1) {
      $head .= substr($buf, 0, $bufsize);
      $buf = substr($buf, $bufsize);
      $amt = ($left > $bufsize ? $bufsize : $left);
#$maxbound==length($buf);
      $errflag = (read(STDIN, $buf, $amt, $maxbound) != $amt);
      $left -= $amt;
   }
   $head .= substr($buf, 0, $lpos+2);
   push (@in, $head);
   @heads = split("\r\n", $head);
```

```
    ($cd) = grep (/^\s*Content-Disposition:/i, @heads);
    ($ct) = grep (/^\s*Content-Type:/i, @heads);


    ($name) = $cd =~ /\bname="([^"]+)"/i; #";
    ($name) = $cd =~ /\bname=([^\s:;]+)/i unless defined $name;


    ($fname) = $cd =~ /\bfilename="([^"]*)"/i; #";
            # filename can be null-str
    ($fname) = $cd =~ /\bfilename=([^\s:;]+)/i unless defined
$fname;
    $incfn{$name} .= (defined $in{$name} ? "\0" : "") . $fname;


    ($ctype) = $ct =~ /^\s*Content-type:\s*"([^"]+)"/i;  #";
    ($ctype) = $ct =~ /^\s*Content-Type:\s*([^\s:;]+)/i unless
defined $ctype;
    $inct{$name} .= (defined $in{$name} ? "\0" : "") . $ctype;


    if ($writefiles && defined $fname) {
       $ser++;
    $fn = $writefiles . ".$$.$ser";
    open (FILE, ">$fn") || &CgiDie("Couldn't open $fn\n");
     }
     substr($buf, 0, $lpos+4) = '';
     undef $fname;
     undef $ctype;
   }


1;
END_MULTIPART
```

If none of the preceding conditions was satisfied, then the program dies because there was a problem with figuring out the context in which this CGI program was called.

```
  &CgiDie($@) if $errflag;
  } else {
    &CgiDie("cgi-lib.pl: Unknown Content-type:
$ENV{'CONTENT_TYPE'}\n");
  }
```

If everything was successful the lilbrary code ends up at the end of the ReadParse routine. As part of the cleanup, the Perl $^W warn variable is set to whatever value it was when this routine was entered. Finally, if every-

thing went OK, the number of elements in the `@in` array is returned. If everything was not OK, `undef` is returned.

```
  $^W = $perlwarn;

  return ($errflag ? undef :  scalar(@in));
}
```

PrintHeader Subroutine

The following code block is a simple routine that prints the "Content-type: text/html\n\n" header that all CGI programs must output before they start printing HTML code.

```
# PrintHeader
# Returns the magic line which tells WWW
# that we're an HTML document

sub PrintHeader {
  return "Content-type: text/html\n\n";
}
```

HtmlTop Subroutine

The `HtmlTop` subroutine is useful for quickly generating an HTML header and returning it in a scalar string variable for printing. The routine expects a string to be passed to it in order to generate the `<TITLE>` tags for the HTML header as well as the `<H1>` tags for the header. We typically do not use this routine—our `<H1>` tags tend to be different from our `<TITLE>` tags—but it is helpful for quickly generating an HTML header when we're testing and creating the CGI script initially. Note that the Perl `HERE` document method is used to block the HTML text between the `END_OF_TEXT` tags in the following code:

```
# HtmlTop
# Returns the <head> of a document
# and the beginning of the body
```

```
# with the title and a body <h1>
# header as specified by the parameter

sub HtmlTop
{
  local ($title) = @_;

  return <<END_OF_TEXT;
<html>
<head>
<title>$title</title>
</head>
<body>
<h1>$title</h1>
END_OF_TEXT
}
```

HtmlBot subroutine

HtmlBot returns the footer of the HTML document, which typically contains the closure for the <BODY> tag and the <HTML> tags.

```
# HtmlBot
# Returns the </body>, </html> codes
# for the bottom of every HTML page

sub HtmlBot
{
  return "</body>\n</html>\n";
}
```

SplitParam Subroutine

Some form variables can have multiple values returned for the same key name. When there are multiple values for a single key in the associative array returned by ReadParse, the values are separated by the \0 character. The SplitParam function takes the value and splits it, using the \0 character, into an array of all the values. This array is then returned as the value of the function. The wantarray function in Perl lets the function return an array if the function is called in the context of returning an array, or

return just the first element of the array if the function is called in the context of returning a scalar. For example, `$test = &SplitParam($test-values)` would return the first element in `$test2`, and `@test = &SplitParam($testvalues)` would return the whole array of values.

```
# SplitParam
# Splits a multi-valued parameter
# into a list of the constituent parameters

sub SplitParam
{
  local ($param) = @_;
  local (@params) = split ("\0", $param);
  return (wantarray ? @params : $params[0]);
}
```

MethGet and MethPost Subroutines

The `MethGet` and `MethPost` subroutines return `True` or `False` depending on whether the CGI method currently being called by the browser is `GET` or `POST`. If the environment variable `REQUEST_METHOD` is `GET`, `MethGet` returns `True` and `MethPost` returns `False`. If the environment variable `REQUEST_METHOD` is `POST`, `MethPost` returns `True` and `MethGet` returns `False`. The environment variable is first checked to see whether it is defined. Because the `&&` (AND) operator is used, if the defined function returns `False`, the rest of the statement after the `&&` never gets processed and `False` is returned to the function for the whole statement. In Perl, the `&&` can be thought of as a short-circuit operator. If the first value is `False`, the second value after the `&&` doesn't matter, because the whole statement will evaluate to `False` anyway. Thus, if the first value is `False`, Perl will "short-circuit" the statement and avoid evaluating the second part, because it will make no difference to the outcome.

```
# MethGet
# Return true if this cgi call was
# using the GET request, false otherwise

sub MethGet {
  return (defined $ENV{'REQUEST_METHOD'} && $ENV{'REQUEST_METHOD'} eq
```

```
"GET");
}


# MethPost
# Return true if this cgi call was
# using the POST request, false otherwise

sub MethPost {
  return (defined $ENV{'REQUEST_METHOD'} && $ENV{'REQUEST_METHOD'} eq
"POST");
}
```

MyBaseUrl Subroutine

MyBaseUrl returns the URL that the browser used to call the script minus any command-line options such as PATH_INFO and QUERY_INFO variables. The Perl warning generator is turned off during the calculation.

```
# MyBaseUrl
# Returns the base URL to the script
# (i.e., no extra path or query string)
sub MyBaseUrl {
  local ($ret, $perlwarn);
  $perlwarn = $^W; $^W = 0;
  $ret = 'http://' . $ENV{'SERVER_NAME'} .
         ($ENV{'SERVER_PORT'} != 80 ? ":$ENV{'SERVER_PORT'}" : '') .
         $ENV{'SCRIPT_NAME'};
  $^W = $perlwarn;
  return $ret;
}
```

MyFullUrl Subroutine

MyFullUrl returns the same thing as MyBaseUrl except that it also returns the extra path information or query information if available. It also turns off Perl warnings for the duration of the calculation.

```
# MyFullUrl
# Returns the full URL to the script
# (i.e., with extra path or query string)
```

```
sub MyFullUrl {
  local ($ret, $perlwarn);
  $perlwarn = $^W; $^W = 0;
  $ret = 'http://' . $ENV{'SERVER_NAME'} .
         ($ENV{'SERVER_PORT'} != 80 ? ":$ENV{'SERVER_PORT'}" : '') .
         $ENV{'SCRIPT_NAME'} . $ENV{'PATH_INFO'} .
         (length ($ENV{'QUERY_STRING'}) ? "?$ENV{'QUERY_STRING'}" : '');
  $^W = $perlwarn;
  return $ret;
}
```

My URL Subroutine

It is recommended that you not use MyURL in your programs, because it is now superseded by the preceding two routines. You will notice that the MyURL function simply calls MyBaseUrl.

```
# MyURL
# Returns the base URL to the script
# (i.e., no extra path or query string)
# This is obsolete and will be
# removed in later versions
sub MyURL  {
  return &MyBaseUrl;
}
```

CgiError Subroutine

CgiError takes a list of error messages and prints a buffer of HTML code that would display the messages one after the other. If no message is given, an error message indicates that a problem occurred in the full URL. If $header_out is 0, then the "Content-type: text/html\n\n" header is printed. The first message in the list is printed as the title and header of the HTML code. All subsequent messages are printed as a list of messages below the header of the HTML code.

```
# CgiError
# Prints out an error message which
# contains appropriate headers,
# markup, et cetera.
```

```
# Parameters:
#  If no parameters, gives a generic error message
#  Otherwise, the first parameter will be the
#  title and the rest will be given as different
#  paragraphs of the body

sub CgiError {
  local (@msg) = @_;
  local ($i,$name);

  if (!@msg) {
    $name = &MyFullUrl;
    @msg = ("Error: script $name encountered fatal error\n");
  };

  if (!$cgi_lib'headerout) { #')
    print &PrintHeader;
    print "<html>\n<head>\n<title>$msg[0]</title>\n</head>\n<body>\n";
  }
  print "<h1>$msg[0]</h1>\n";
  foreach $i (1 .. $#msg) {
    print "<p>$msg[$i]</p>\n";
  }

  $cgi_lib'headerout++;
}
```

CgiDie Subroutine

CgiDie calls CgiError and then calls the Perl DIE routine so that the error
can be printed to STDERR and the program exits.

```
# CgiDie
# Identical to CgiError, but also
# quits with the passed error message.

sub CgiDie {
  local (@msg) = @_;
  &CgiError (@msg);
  die @msg;
}
```

PrintVariables Subroutine

The `PrintVariables` subroutine takes an associative array and returns a buffer of nicely formatted HTML output. An interesting technique for passing parameters is used here. Normally, `@_` is the array or list of parameters. When `@_` is referred to in a numerical expression, its length is returned. If the length of the parameters is 1, we assume that a reference to the associative array was passed. If the length is greater than 1, we assume that the associative array was passed by value. Notice that the values are set differently in the following code for `in` if it is passed by reference or by value. Except for that, the program uses the standard Perl FOREACH statement to walk through the keys of the associative array and parse the value for each key.

```
# PrintVariables
# Nicely formats variables.  Three calling options:
# A non-null associative array - prints the
#   items in that array
# A type-glob - prints the items in the
#   associated assoc array
# nothing - defaults to use %in
# Typical use: &PrintVariables()

sub PrintVariables {
  local (*in) = @_ if @_ == 1;
  local (%in) = @_ if @_ > 1;
  local ($out, $key, $output);

  $output =  "\n<dl compact>\n";
  foreach $key (sort keys(%in)) {
    foreach (split("\0", $in{$key})) {
      ($out = $_) =~ s/\n/<br>\n/g;
      $output .=  "<dt><b>$key</b>\n <dd>:<i>$out</i>:<br>\n";
    }
  }
  $output .=  "</dl>\n";

  return $output;
}
```

PrintEnv Subroutine

`PrintEnv` calls the `PrintVariables` routine and passes the environment associative array by reference.

```
# PrintEnv
# Nicely formats all environment
# variables and returns HTML string
sub PrintEnv {
  &PrintVariables(*ENV);
}
```

End of the Main Script

The following code block sets a couple of variables equal to themselves in order to avoid a warning message about not using the variables. We know that these variables are used in the preceding subroutines, so we do not want a warning message to print if we run the program with the warn parameter turned on. Finally, as with all libraries, the last line is 1; in order to return a TRUE value to the REQUIRE subroutine.

```
# The following lines exist
# only to avoid warning messages
$cgi_lib'writefiles =  $cgi_lib'writefiles;
$cgi_lib'bufsize    =  $cgi_lib'bufsize ;
$cgi_lib'maxbound   =  $cgi_lib'maxbound;
$cgi_lib'version    =  $cgi_lib'version;


1; #return true
```