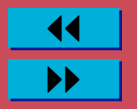
Some More Perl — Useful Examples

Perl is full of some very useful features

Perl makes many task very easy once you understand how these features work.

Some features, especially text processing with Regular Expressions help explain why Perl is so Popular as a CGI and more general purpose language.




Regular Expressions

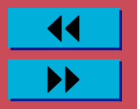
A **regular expression** is a template or pattern to be matched against a string.

In Perl a regular expression is enclose inside two slashes:

```
/regular_expression/
```

The regular expression may contain:

- Ordinary text to be matched to an exact pattern (or sub pattern)
- Special operator characters — characters that have a special meaning and control how we match patterns




Special pattern matching character operators

Perl has a few special pattern matching character within `/../`:

```
\    Quote the next metacharacter
^    Match the beginning of the line
.    Match any character (except newline)
$    Match the end of the line
     (or before newline at the end)
|    Alternation
()   Grouping
[]   Character class
```


The . Operator — Any Character Matches

The simplest and very common pattern matching character operators is the **.**

This simply allows for any single character to match where a **.** is placed in a regular expression.

For example **/b.t/** can match to **bat, bit, but** or anything like **bbt, bct ....**




Alternative Character Matches

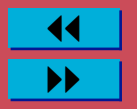
Square brackets **(`[..]`)** allow for any one of the letters listed inside the brackets to be matched at the specified position.

For example `/b[aiu]t/` can only match to **bat, bit** or **but**.

You can specify a **range of values** inside `[..]`.

For example:

```
[012345679] # any single digit
[0-9] # also   any single digit
[a-z] # any single lower case letter
[a-zA-Z] # any single letter
[0-9\-] # 0-9 plus minus character
```


Back
Close

The Caret (Negative Matching) Operator

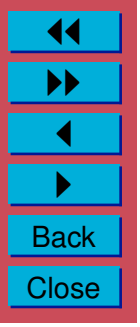
The **caret** (^) can be used to negate matches

For example:

```
[^0-9] # any single non-digit
[^aeiouAEIOU] # any single non-vowel
```

The following **control characters** can also be used:

- \d— Match any **digit** character,
- \s — Match a **space** character,
- \w — Match any **word character**
- \D, \S, \W are the **negations** of \d\s\w — They match any respective **NON**-digit, space or word character.

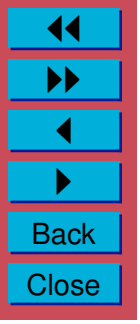


Standard Quantifiers

Quantifiers let us extend the basic pattern matching operators (above) into a much **wider and powerful matching** territory.

The following standard quantifiers are recognized:

```
*       Match 0 or more times
+       Match 1 or more times
?       Match 1 or 0 times
{n}     Match exactly n times
{n,}    Match at least n times
{n,m}   Match at least n
        but not more than m times
```

Some Simple Quantifier Examples

`fa*t` matches to ft, fat, faat, faaat **etc**

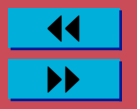
- `a*` allows for matches of zero or more `a`s

`.*` can be used a **wild card** match for any number (zero or more) of any characters.

For Example:

`f.*k` matches to fk, fak, fork, flunk, **etc.**

In fact any strings (However long but at least 2 characters (`fk`)) that start with `f` and end with `k` will match




Some Simple Quantifier Examples (Cont.)

`fa+t` matches to fat, faat, faaat **etc**

- `a+` allows for matches of one or more `a`s

`.+` can be used to match to one or more of any character **i.e.** at least something must be there.

For Example:

`f.+k` matches to fak, fork, flunk, **etc. but not** fk.

In fact any strings (However long but at least 3 characters) that start with `f` and end with `k` with any (one or more) characters in between will match


Some Simple Quantifier Examples (Cont.)

`ba?t` matches to **bt** or **bat**

- `a?` matches to **zero or one a** character.

`b.?t` matches to **bt, bat, bbt, bct etc. but not bunt** or **string lengths higher than four-letters.**

`ba{3}t}` only matches to **baaat**

- as `a{3}` only matches to **3 a** characters

`ba{1,4}` matches to **bat, baat, baaat** and **baaaat**

- as `a{1,4}` matches to between **1 and 4 a** characters

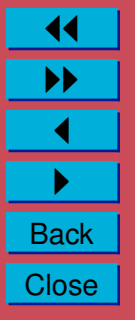
More Examples in Online HTML Notes and Additional Online Perl Notes.




Parenthesis (. .) For Enforcing Precedence and For Memory and Backtracking

In Perl Regular Parenthesis Brackets (. .) have two functions:

- Enforcing Precedence
- For Memory and Backtracking in Current and Further Perl Expressions



Parenthesis (..) For Enforcing Precedence

Parenthesis (..) can be used to delimit special matches and therefore **enforce precedence**.

- Much like you control evaluation of arithmetic and other expressions in any programming language

For example:

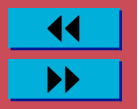
```
(abc)*
```

matches " ",abc, abcabc, abcabcabc,.....

and

```
(a|b)(c|d)
```

matches ac, ad,bc,bd




Parenthesis (..) as Memory and Backtracking

- The brackets `()` can be used to remember (sub)patterns
- Any number of pairs of brackets `()` may be used:
  - Each Pair may be referenced within the Perl Regular Expression by a `\1,\2,\3 ....`
  - Each `\1,\2,\3 ....` refers to a pattern matched at the respective (preceding) `()`

Example :

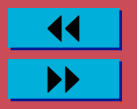
```
dave(.)marshall\1
```

will match something like

```
daveXmarshallX
```

BUT NOT

```
daveXmarshallY
```


Back
Close

Multiple Memories of Matches

You can have more than one memory per Regular Expression

For Example:

```
a(.)b(.)c\2d\1
```

would match `axbycydx`.

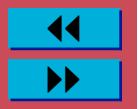
Multiple chars (incl. zero character) can be remembered:

```
a(.*)b\1c
```

matches to `abc` or `aFREDbFREDc`

BUT NOT

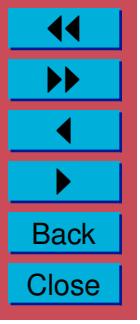
`aXXbXXXc`, for example.



Memory Variables (Read Only)

Not only does Perl remember matches inside a Regular Expression but immediately after matches are remembered

- After a successful match the variable `$1, $2, $3, ...` are set on the same values as `\1,\2,\3, ....`
- These are Special Perl Variables that a Read Only — you cannot assign any values to them in your Perl code.
- Only the last match is guaranteed to have meaningful values loaded into `$1, $2, $3, ...`
- After every Regular Expression Match `$1, $2, $3, ...` will be loaded — Even if you do not use them in your program

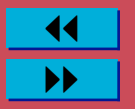


Memory Variables Example

```
$_ = "One Two Three Four Once ....";
/(\w+)\W+(\w+)/; # match first two words

print "1st Word is " . $1" . "\n";
print "2nd Word is " . $2" . "\n";
```

How does this work?

- We check for valid sequences of characters — Not Valid English Words
- `\w` matches any word character
- `w+` matches one or more word characters
- `(\w+)` remembers respective matches
- `\W` matches any **Non**-word character
- `\W+` matches one or more **Non**-word characters
- `\W+` delimits any break between two sets of word characters
- `$1` and `$2` loaded with two sets of respective word characters.



Back
Close

Suppressing Memories

`(?:regular_expression)`

- This groups things like regular `()` but doesn't make back references to internal or external memories like `()` does.



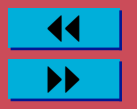

Perl Regular Expression Matching in Practice

We have so far seen how matching can be set up but not how it works in practice.

By default, Perl matches all regular expressions to a special variable: `$_`

- We will see may uses of `$_` in a moment
- This makes for very short hand file access, for example.

However, in many case we may wish to **bind** the matching to our own **Target Variable**




Setting the Target Operator

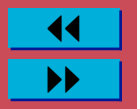
The `|=~|` **operator** lets you match against a specified target

- Rather than the environment variable `$_`.
- **For example**: **In CGI** we frequently need to match against input **name/value pairs**.

A typical use is with an `if` statement to control the match, for example:

```
$infile = ; # whatever

if ( $infile =~ /.*\.gif/)
  { # file is gif format file
    # well at least it ends in a .gif

   ........
   }
```

Note: We need the `\.` as `.` has a special regular expression (wild card character) meaning.




Substitution

We may frequently need to change a pattern in a string.

The **substitution operator (`s`)** is the simplest form of substitution. It has the form:

```
s/old_regex/new_string/
```

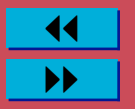
Note: We can (optionally) qualify the substitution with

- A `g` global substitution,
- A `i` ignore case and
- A `e` evaluate right side as expression and others.

We place the qualifiers, if appropriate, on the right hand side of the substitution, For example:

```
s/old_regex/new_string/gie
```

Zero, one or more qualifiers may be used.



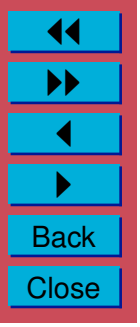

Practical CGI Related Example

To replace the + characters from CGI input with a space we could do:

```
$CGI_in_val =~ s/\+/ /ge;
```

- We use `g` global substitution to replace all occurrences of + in the input string.
- We need `e` to force the evaluations of the expression so that the value can be returned to the input string:
  - That is to say the input string will have its **value changed**, for future use in the Perl code, by this operation

Note: `cgi-lib.pl` actually does this and other cgi character conversion automatically — peek at the `cgi-lib.pl` source code and find such examples



Split() and join()

`Split()` and `join()` are two very useful functions.

`Split()` takes a regular expression and a string:

```
split(reg_ex,string)
```

and looks for all occurrences of the regular expression and the parts of the string that don't match are returned, in sequence, in a list.

Example: To split an input name/value pair CGI input we could do:

```
$cgi_pair = "name=value"; #format of input
($name,$value) = split(/=/,$cgi_pair);
@cgi_list =  = split(/=/,$cgi_pair);
```

The `join()` function takes two lists and glues them together.

Reading Directories

Perl has several functions to operate on functions the `opendir()`, `readdir()` and `closedir()` functions are a common way to achieve this.

```
opendir(DIR_HANDLE,"directory")
```

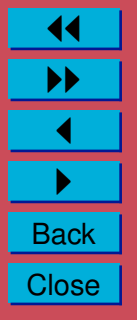
returns a Directory **handle** — just an identifier (no $) — for a given `directory` to be opened for reading.

Note: Exact or relative subpath directories are required.

- UNIX/LINUX/Mac OS X directory paths are denoted by `/`.

`readdir(DIR_HANDLE)` returns a scalar (string) of the **basename** of the file (no sub directories (: or /))

`closedir(DIR_HANDLE)` simply closes the directory.



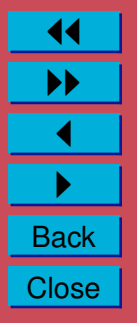
UNIX Read Directory Example

On UNIX we may do:

```
opendir(IDIR,"./Internet")
   || die "NO SUCH Directory: Images";

while ($file = readdir(DIR) )
  {
        print " $file\n";

   }
closedir(DIR);
```

The above reads a sub-directory `Internet` assumed to be located in the same directory from where the Perl script has been run `./`.

Note: The `|| die "..."` is a short hand if type statement that outputs a string, enclosed in `"..."` and then quits the program



Alphabetical Order Directory Example

One further example to alphabetically list files is:

```
opendir(IDIR,"./Internet")
   || die "NO SUCH Directory: Images";

foreach $file ( sort readdir(DIR) )
  {
        print " $file\n";

   }
closedir(DIR);
```





Reading and Writing Files

We have just introduced the concept of a **Directory Handle** for referring to a Directory on disk.

We now introduce a similar concept of **File Handle** for referring to a File on disk from which we can read data and to which we can write data.

Similar ideas of opening and closing the files also exist.

You use the `open()` operator to open a file (for reading):

```
open(FILEHANDLE,"file_on_device");
```

The `file` may be accessed with an absolute or relative path.


Opening a File for Reading and Appending

To open a file for writing you must use the "`>`" symbol in the `open()` operator:

```
open(FILEHANDLE,">outfile");
```

Write always starts writing to file at the **start of the file**.

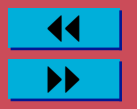
- If the file already exists and contains data.
- The file will be opened and the **data overwritten**.

To open a file for **appending** you must use the "`>>`" symbol in the `open()` operator:

```
open(FILEHANDLE,">>appendfile");
```

The `close()` operator closes a file handle:

```
close(FILEHANDLE);
```

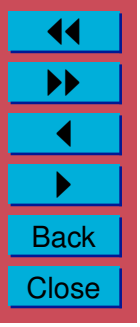
Three Special File Handles

in Perl, hree special file handles are always open `STDIN, STDOUT` and `STDERR`.

`STDIN` reads from **standard input** which is usually the keyboard in normal Perl script or input from a Browser in a CGI script.

`cgi-lib.pl` reads from this automatically.

`STDOUT` (**Standard Output**) and `STDERR` (**Standard Error**)by default write to a console or a browser in CGI.



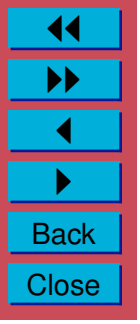
File Reading Example

To read from a file

- You simply use the `<FILEHANDLE>` in a statement (typically `while`)
- This tells Perl to read one line at a time from a file references by `FILEHANDLE` — specified in `open()`
- Each line is read and stored it in a special Perl variable `$_` — which we mentioned a few slides back
- When no more lines can be read `FALSE` is returned — which terminates the `while` statement.

For example:

```
open(FILE,"myfile")
   || die "cannot open file";
while(<FILE>)
{ print $_; # echo line read
}
close(FILE);
```

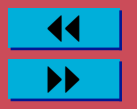
Writing to a File

To **write** to a file you use the `print` command and simply refer to the `FILEHANDLE` before you format and output the string, I.e.:

```
print FILEHANDLE "Output String\n";
```

Example: To read from one file `infile` and copy line by line to another `outfile` we could do:

```
open(IN,"infile")
   || die "cannot open input file";
open(OUT,"outfile")
   || die "cannot open output file";
while(<IN>)
{ print OUT $_; # echo line read
}
close(IN);
close(OUT);
```

Perl Subroutines or functions

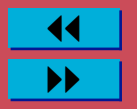
We have actually been using Perl (`cgi-lib.pl`) subroutines or functions from the start.

We will know look at how we develop our own.

Subroutines are useful in breaking up or programs into smaller more manageable pieces that make program development and maintenance much easier.

A subroutine in Perl is defined as follows:

```
sub subname {
     statement_1;
     statement_2;
     ........
     statement_n;
}
```

Back
Close

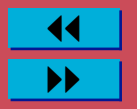
Defining Subroutines

Subroutine names can be any usual name as used for scalars, arrays **etc.**

Subroutines are always referred to by a `&` in Perl.

You can supply arguments to a subroutine:

- They are passed in as a list and referred to by the `$_` list within the subroutine body.
- Another use of `$_`
- Syntax different from Java/C etc.




Simple Perl Subroutine Example: `sum`

Example: To add two numbers together, a function `sum`.
In Perl we may call a function `sum` as follows

```
$a = 3;
$b = 4;
$c = &sum($a,$b);
```

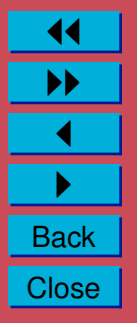
Note: The `&` prefix when we call `sum`

The subroutine may be defined as follows:

```
sub sum {
  $_[0] + $_[1];
}
```

Note: The value of the expression is `$_[0] + $_[1]` is the **returned** value

- The final expression (last function statement) is the value returned by any function unless you explicitly call a `return` operation in the function.



Some Example Perl Scripts

Let us conclude our brief study of Perl by looking at some example CGI scripts.

The examples are taken form the Laura Lemay book:

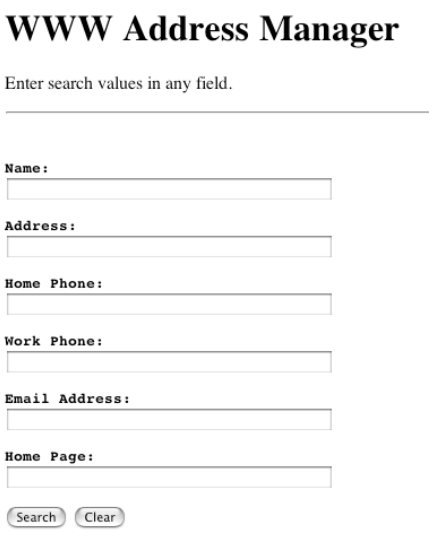
"Teach yourself web publishing in HTML" Series of books.



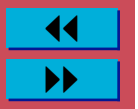

An Address Book Search Engine

This is a more complex and larger script:

- It illustrates how information may be queried from a information stored in a type database — for now we keep things simple.
- The data base is just a text file and we can only read from the file.



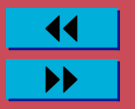
To see this form in action click here.




Address Book Example — The HTML/Browser/Client Side

The HTML Form Front-End is composed via:

```
<HTML>
<HEAD>
<TITLE>Address Book Search Forms</TITLE>
</HEAD>
<BODY>
<H1>WWW Address Manager</H1>
<P>Enter search values in any field.
<PRE><HR>
<FORM METHOD=POST
ACTION="http://www.cs.cf.ac.uk/user/Dave.Marshall/cgi-bin/address.pl">
<P><B>Name:</B>
<INPUT TYPE="text" NAME="Name" SIZE=40>
<P><B>Address:</B>
<INPUT TYPE="text" NAME="Address" SIZE=40>
<P><B>Home Phone:</B>
<INPUT TYPE="text" NAME="Hphone" SIZE=40>
<P><B>Work Phone:</B>
<INPUT TYPE="text" NAME="Wphone" SIZE=40>
<P><B>Email Address:</B>
<INPUT TYPE="text" NAME="Email" SIZE=40>
<P><B>Home Page: </B>
<INPUT TYPE="text" NAME="WWW" SIZE=40>
</PRE>
<INPUT TYPE="submit" VALUE="Search">
<INPUT TYPE="reset" VALUE="Clear">

<HR>
</FORM>
</BODY>
</HTML>
```


Back
Close

Address Book Example — The Perl/CGI/Server Side

The Perl CGI script is as follows:

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

# grab values passed from form:
&ReadParse(*in);

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

# print the top part of the response
print "<HTML><HEAD><TITLE>Addresss Book Search Results</TITLE></HEAD>\n";
print "<BODY><H1>Addresss Book Search Results</H1>\n";

# read and parse data file
$data="address.data";

open(DATA,"$data") || die "Can't open $data: $!\n</BODY></HTML>\n";
while(<DATA>) {
chop; # delete trailing \n
if (/^\s*$/) {
# break between records
if ($match) {
# if anything matched, print the whole record
&printrecord($record);
$nrecords_matched++;
}
undef $match;
undef $record;
next;
}
# tag: value
($tag,$val) = split(/:/,$_,2);
if ($tag =~ /^Name/i) {
```





Back
Close

```
$match++ if( $in{'Name'} && $val =~ /\b$in{'Name'}\b/i) ;
$record = $val;
next;
}
if ($tag =~ /^Address/i) {
$match++ if( $in{'Address'} && $val =~ /\b$in{'Address'}\b/i) ;
$record .= "\n<BR>$val" if ($val);
next;
}
if ($tag =~ /^Home\s*Pho/i) {
$match++ if( $in{'Hphone'} && $val =~ /\b$in{'Hphone'}\b/i) ;
$record .= "\n<BR>Home: $val" if ($val);
next;
}
if ($tag =~ /^Work/i) {
$match++ if( $in{'Wphone'} && $val =~ /\b$in{'Wphone'}\b/i) ;
$record .= "\n<BR>Work: $val" if ($val);
next;
}
if ($tag =~ /^Email/i) {
$match++ if( $in{'Email'} && $val =~ /\b$in{'Email'}\b/i) ;
$record .= "\n<BR><A HREF=\"mailto:$val\">$val</A>" if ($val);
next;
}
if ($tag =~ /Page/i) {
$match++ if( $in{'WWW'} && $val =~ /\b$in{'WWW'}\b/i) ;
$record .= "\n<BR><A HREF=$val>$val</A>" if ($val);
next;
}
# anything else
$record .= $_;
}
close DATA;

if (! defined $nrecords_matched)
{ print "<H2>No Matches</H2>\n"; }
```





Back
Close

```
print "</BODY></HTML>\n";
exit;

sub printrecord {
local($buf) = @_;
print "<P>\n$buf<P>\n";
}
```

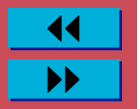




What is going on here?

This Perl Script essentially does the following:

- We use the `cgi-lib.pl ReadParse` subroutine to read the CGI input.
- We extract out the associated name value pairs.
- The data is read in from a file **address.data.txt**
- The data is searched using Perl regular expressions for given **Names, Addresses etc.** and matches stored and printed out.
- The subroutine `printrecord` prints out the matched record.




Creating a Guest Book

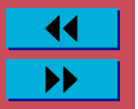
The Guest book is a more complicated example:

- An extension of our the address book example.
- In a guestbook readers can post comments about you WWW pages.
- We write to a file in this example

Guestbooks are quite common on WWW sites.

The HTML Form is as follows:

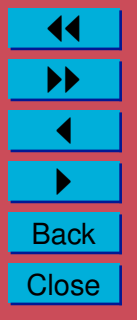
To see this form in action click here.



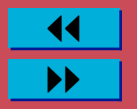

GuestBook Example — The HTML/Browser/Client Side

The HTML Form Front-End is composed via:

```
<HTML>
<HEAD>
<TITLE>Comments!</TITLE>
</HEAD>
</BODY>

<!--GUESTBOOK-->
<H1>Comments!</H2>
<P>Here are comments people have left
about my pages.  Post your own
using the form at the end of the page.
<P>Comments list started on
<!--STARTDATE--> Apr 4 2003
Last post on
<!--LASTDATE-->
Thu Aug 24 09:25:46 PDT 2003
<HR><B>Susan M.
 <A HREF=mailto:sdsm>sus@monitor.com
</A></B>
Tue Apr 10 05:57:09 EDT 2003
<P>This is the worst home page
I have ever seen on the net.  Please
```

```
stop writing.
<HR><B>Tom Blanc
 <A HREF=mailto:le7may@lne.com>
tlb666@netcom.com</A></B>  Wed Apr 11
21:58:50 EDT 2003
<P>Dude.  Get some professional help.
Now.


<!--POINTER-->
<!--everything after this is standard
and unchanging. -->
<HR>
Post a response:
<BR>
<FORM METHOD=POST
ACTION="http://www.cs.cf.ac.uk/user/Dave.Marshall/
cgi-bin/guestbook.pl/guest.html">


Name: <INPUT TYPE="text" NAME="name"
SIZE=25 MAXLENGTH=25>
<BR>
Email address: <INPUT TYPE="text"
NAME="address" SIZE=30 MAXLENGTH=30>
<BR>
```

Back
Close

```
Text:
<BR>
<TEXTAREA ROWS=15 COLS=60 NAME="body">
</TEXTAREA>
<BR>
<INPUT TYPE=submit VALUE="POST">
<INPUT TYPE=reset VALUE="CLEAR">
</FORM>
<HR>
</BODY>
</HTML>
```



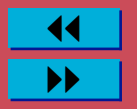


Back
Close

GuestBook Example — The Perl/CGI/Server Side

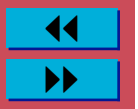
The Perl CGI is as follows:

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

# grab values passed from form:
&ReadParse(*in);

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

# print the top part of the response
print "<HTML><HEAD>\n";
print "<TITLE>Post Results</TITLE>\n";
print "</HEAD><BODY>\n";

# change to your favorite date format:
$date = `date`;
chop($date); # trim \n

# Grab the HTML file and make a file name for the temp file.
$file = "$ENV{'PWD'}" . "$ENV{'PATH_INFO'}";
$tmp = $file . ".tmp";
$tmp =~ s/\//@/g; # make a unique tmp file name from the path
$tmp = "/tmp/$tmp";

# if any fields are blank, then skip the post and inform user:
if ( !$in{'name'} || !$in{'address'} || !$in{'body'}) {
    &err("You haven't filled in all the fields. Back up and try again.");
}

# reformat the body of the post.  we want to preserve paragraph breaks.
$text = $in{'body'};
$text =~ s/\n\r/\n/g;
$text =~ s/\r\r/<P>/g;
$text =~ s/\n\n/<P>/g;
```

Back
Close

```
$text =~ s/\n/ /g;
$text =~ s/<P><P>/<P>/g;

# get an exclusive open on the tmp file, so
# two posts at the same time don't clobber each other.
for($count = 0; -f "$tmp"; $count++) {
# oh no. someone else is trying to update the message file.  so we wait.
sleep(1);
&err("Tmp file in use, giving up!") if ($count > 4);  # but not for long
}
open(TMP,">$tmp") || &err("Can't open tmp file.");
# open the HTML file
open(FILE,"<$file") ||
&err("Can't open file $file: $!");

# an HTMLBBS file.  look through it for the HTML comments
# that denote stuff we want to change:
while(<FILE>) {
if (/<!--LASTDATE-->/)  { print TMP "<!--LASTDATE-->  $date \n"; }
elsif (/<!--GUESTBOOK-->/) {
print TMP "<!--GUESTBOOK-->\n";
$guestbook++;
}
elsif (/<!--POINTER-->/) {
# add this post
print TMP "<HR>";
print TMP "<B>$in{'name'}  \n";
print TMP " <A HREF=mailto:$in{'address'}>
                  $in{'address'}</A></B>  $date\n";
print TMP "<P> $text\n<!--POINTER-->\n";
}
else { print TMP $_; }  # copy lines
}
if (! defined $guestbook)
{ &err("not a Guestbook file!"); }
```

Back
Close

```
# move the new file over the old:
open(TMP,"<$tmp") || &err("Can't open tmp file.");
# open the HTML file
open(FILE,">$file") ||
&err("Can't open file $file: $!");

while(<TMP>) {
print FILE $_;
}
close(FILE);
close(TMP);
unlink "$tmp";

# print the rest of the response HTML
print "<H1>Thank you!</H1>";
print "<P>Your comment has been added to the ";
print "<A HREF=\"/User/Dave.Marshall/guest.html\">guestbook</A>\n";
print "</BODY></HTML>\n";
1;

# if we got an error, print message, close & clean up
sub err {
local($msg) = @_;
print "$msg\n";
close FILE;
close TMP;
unlink "$tmp";
print "</BODY></HTML>\n";
exit;
}
```





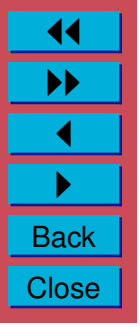
What is going on here?

This Perl Script essentially does the following:

- We use the `cgi-lib.pl ReadParse` subroutine to read the CGI input.
- We extract out the associated name value pairs.
- Note that the form `ACTION` passes in a file to read — actually the HTML source that generated the form in the first place.
  - On Unix because HTML and CGI scripts are placed in different directories you may either have to copy or make a UNIX symbolic link to the file:
    To make a UNIX link:
    * Telnet to UNIX account.
    * `cd` to your `public_html/cgi-bin` directory.
    * Type from the command line:

      ```
      ln -s ../public_html/file.html
      ```
      where `file.html` is the file you need to link to.

      Linking is better than copying since if you change any HTML in the `file.html` the link refers to the updates file. A copied file is out of date and therefore **incorrect**.
- Data is actually written to this file as "guests" leave their comments.
- We therefore have a recursive dynamic HTML file/CGI script interaction. HTML form calls script save to HTML file **etc.**



A Web Page Counter

Web page counters can be used to indicate how many time your Web page or pages have been visited.

The URL:

http://www.htmlgoodies.com/beyond/countcgi.html

contains Perl scripts, instructions and explanations as to how to achieve this.