Input and Output in Perl

- We already know how to open a file and read its contents a line at a time:

```
open INFILE, "data.txt" or
      die "Could not open file to read: $!";

while (<INFILE>)
{
  # Do some processing on $_ ...
}
```

- To open a file for *writing*, we prefix the filename with a ">":

```
open OUTFILE, ">output.txt" or
      die "Could not open file to write: $!\n";
```

- Note: the `output.txt` file is *replaced* with any new data we send to it, i.e., it is *overwritten*

Perl's Copy

- Here's how to write a standard file copy program in Perl (assumes text files):

```
#!/usr/bin/perl -w

open FROM, "$ARGV[0]" or
      die "Could not open file to copy FROM: $!\n";

open TO, ">$ARGV[1]" or
      die "Could not open file to copy TO: $!\n";

print TO <FROM>;
```

- When opening a file for reading, you can be explicit and state the above open command as:

```
open FROM, "<$ARGV[0]" or
      die "Could not open file to copy FROM: $!\n";
```

STDIN and STDOUT

- The standard filehandles for input and output are opened automatically by Perl
- Sometimes it is useful to control the filehandle `STDIN` and `STDOUT` are associated with
- This next fragment of Perl code connects `STDIN` up to the filehandle `INPUT` if a command line argument is passed to the script, otherwise `STDIN` is opened as normal:

```
open INPUT, $ARGV[0] || '<-' or
            die "Could not open input file: $!\n";
```

- The `'<-'` symbol is a pseudofile representing `STDIN`, whereas `'>-'` represents `STDOUT`

More on Filehandles

- Again, remember that the '>' output specifier opens a file for output, and, if it exists, *overwrites* its contents with the data written

- To open a file while *preserving* its contents, use the append specifier '>>':

```
open LOGFILE, '>>violations.log' or
       die "Unable to append to log file: $!\n";

($sec, $min, $hour, $month_day, $month, $year) = gmtime();
++$month; $year += 1900;
print LOGFILE "Violation at $hour:$min:$sec on $month_day/$month/$year\n";
```

- To open a file for *reading and writing*, use '+<' to preserve the contents before you process the file, or use '>+' to overwrite the file contents:

```
open DATABASE, "+<mytextdbfile.txt" or
      die "Could not open database file: $!\n";
```

Reading and Writing to Pipes

- On Linux (UNIX) systems, we can *pipe* the output from one command into another:

```
ls -lR | more
```

- The `ls -lR` command will execute the long version of the `ls` command, and *recursively* access all subdirectories looking for files to list, while the `more` command takes a file as its standard input, and lets us view the file a page (screen full) at a time

- We can use pipes as `STDIN` and `STDOUT` within Perl scripts:

```
open STDIN, 'ls -lR|' or
      die "Unable to pipe from ls command: $!\n";

open STDOUT, '|lpr -Ppsc' or
      die "Unable to pipe to printer at psc: $!\n";
```

The Current File and Line Number

- Here's a variation of the script from the end of the notes on lists:

```
#!/usr/bin/perl -w

$string = shift;

while (<>)
{
  print "Found: $string at line: $. in file: $ARGV\n" if /$string/;
}
```

- We look for the string passed in as a command line argument, then if we find it, we print out the current line number of the file being processed (which is in the built-in variable `$.`) as well as the current filename (which is in the built-in variable $ARGV)

- Note: `$.` can be referred to as `$INPUT_LINE_NUMBER` if we '`use English;`'

$. Perl Gotcha

- Unfortunately, if we feed the above script a collection of files, `$.` is only reset whenever a filehandle is *explicitly closed* - I ran the following command, knowing that only one file from my collection of `*.perl` files contained the string "`barryp`":

```
$ perl test.pl "barryp" *.perl
```

- and got the following output:

```
Found: barryp at line: 453 in file: io.perl
Found: barryp at line: 455 in file: io.perl
Found: barryp at line: 459 in file: io.perl
Found: barryp at line: 466 in file: io.perl
```

- when what I really *expected* to see was this:

```
Found: barryp at line: 18 in file: io.perl
Found: barryp at line: 20 in file: io.perl
Found: barryp at line: 24 in file: io.perl
Found: barryp at line: 31 in file: io.perl
```

$. Perl Gotcha (cont.)

- The Perl default (or null) filehandle *comes to the rescue!*
- The $ARGV filename has associated with it a filehandle called ARGV
- If we check to see if the filehandle ARGV has reached the end-of-file, we can close the filehandle and (as a side effect) reset the $. variable to zero
- We add one line to the script, and then it works as we expect it to:

```
#!/usr/bin/perl -w

$string = shift;
while (<>)
{
  print "Found: $string at line: $. in file: $ARGV\n" if /$string/;

  close ARGV if eof;    #Note: 'eof' is a built-in Perl function.
}
```

Line Ranges

- Here's a script which only prints the first 5 lines of each file processed:

```
#/usr/bin/perl -w

while (<>)
{
      print if 1..5;    # Note: 1..5 is a line range.
}
```

- This script prints to the end of a file after it finds the first blank line:

```
#/usr/bin/perl -w

while (<>)
{
      print if /^$/..eof;
}
```

Here Documents

- Problem: we want to print out a formatted multi line text message
- We can easily do this with a whole bunch of `print` commands, with each line of the output associated with a single print statement:

```
print "psearch.pl: version 1.1, by Paul Barry, November 1999.\n\n"
print "Usage:\n"
print "    perl psearch.pl \"search string\" <list of files>\n\n"
print "    The psearch program looks for a given search string  ...
```

- Note how we have had to concern ourselves with *escaping* special characters such as `"`
- Of course, this is Perl, and *There's More Than One Way To Do It!*

Here Documents (cont.)

- Perl borrows a mechanism from the UNIX world and implements a *'here document'*
- Here documents allows us to more easily handle the above, and (on the next slide) we extend our search program from earlier to include a usage message which is implemented by way of a *here document*

psearch.pl

```
#!/usr/bin/perl -w

$usage = <<USAGE_MSG;

psearch.pl: version 1.1, by Paul Barry, November 1999.

Usage:
    perl psearch.pl "search string" <list of files>

    The psearch.pl program looks for a given search string in the list of
    files provided on the command line.  Here are some examples of its use:

            perl psearch.pl "exit tutorial" tutor.doc
            perl psearch.pl "stdio.h" common.h myproject.h
            perl psearch.pl "ethernet" *.txt
USAGE_MSG

if ($#ARGV < 2) { print $usage; exit 1 }

$string = shift;

while (<>)
{
  print "Found: $string at line: $. in file: $ARGV\n" if /$string/;

  close ARGV if eof;
}
```

Another Here Document Example

- Here's a short script which generates an even shorter web page to STDOUT:

```
#!/usr/bin/perl

print <<END_HTML;
<HTML>
<HEAD>
<TITLE>A Really Short Web Page</TITLE>
</HEAD>
<BODY>
<B>Hello World!</B>
</BODY>
</HTML>

END_HTML
```

Formatted Output

- If you are working with a formatted text file (the `/etc/passwd` file is one such example), Perl provides a simple, yet highly effective, report generation mechanism using the `format` and `write` commands - if this interests you, look up all the details in the `man perlform`
- Perl also supports the `sprintf` statement, which works essentially the same way it does in C:

```
$string = sprintf "%02d:%02d:%02d\n”, $mins, $secs, $frames;
print $string;
```

- can also be written as:

```
print sprintf "%02d:%02d:%02d\n”, $mins, $secs, $frames;
```

- As the `print sprintf` combination is so common, Perl supports the use of `printf` function as well

Binary Files

- To open a file in *binary mode*, use the `binmode` function:

```
$ep = "Packet.EtherPeek";
open ETHEREEK, $ep or die "Could not open the binary file: $!\n";
binmode ETHERPEEK;
```

- We can now *read* from the file using the `read` function:

```
$bytes = read ETHERPEEK, $buffer, 256;
```

- will attempt to read `256` bytes from the `ETHERPEEK` filehandle into a scalar variable called `$buffer` - the actual number of bytes read are put into `$bytes`

- To *write* to a binary file, simply use the `print` function

- Random access within binary files is accomplished by the `seek` and `tell` functions(which are just like those in C) - see `man perlfunc` for more details

Working with Binary Data

- Perl provides two useful functions, `pack` and `unpack`, that can be used when working with data from binary files

- Let's assume that the `256` bytes read from the `ETHERPEEK` filehandle has the following format:

```
Bytes: 1-10 is a version string
Byte:  11 is an unsigned version number
Bytes: 12-15 contain the current packet number
Bytes: 16-256 contain the packet data
```

- We can use `unpack` to extract the data from the `256` bytes using a *template*:

```
($ver, $ver_num, $pack_num, $data) = unpack "A10INa240", $bytes
```

- where "`A10`" is an ASCII string 10 bytes long, "`I`" is an unsigned integer, "`N`" is a four-byte integer, and "`a240`" is an ASCII string which is 240 bytes long that can contain nulls

An Interesting `unpack` Example

- The following code computes a 16-bit checksum on any file whose name you provide on the command-line:

```
#!/usr/bin/perl -w

$checksum = 0;

while (<>)
{
  $checksum += unpack("%16C*", $_);
}

$checksum %= (2 ** 16) - 1;

print "The checksum for $ARGV is: $checksum\n";
```

- Refer to `man perlfunc` for more information on `unpack`, templates, and the `pack` function (which can undo the work of unpack)

Files and Directories

- We can test files - before working with them - for certain characteristics, using Perl's "dash" operators (this is just some of them):

```
-r  can we read from a file?
-w  can we write to a file?
-x  is the file executable?
-o  do we own the file?
-e  does the file exist?
-z  is the file zero bytes long?
-s  what size is the file?
-d  is the file a directory?
-t  is the file a terminal?
-T  is the file a text file?
-B  is the file a binary file?
```

Opening File Example

- This is a little bit *too cautious*, but it does show the "dash operators" in action:

```
#!/usr/bin/perl -w

$filename = "EtherPeek.Listing";

-e $filename or
      die "$filename does not exist\n";

-r $filename or
      die "$filename cannot be read from\n";

-s $filename or
      die "$filename is empty (no contents)\n";

-T $filename or
      die "$filename is binary\n";

open LISTING, $filename or
      die "Some other problem opening $filename: $!\n";
```

Directories

- Perl's support for working with directories and the files residing therein mimics that provided by UNIX
- Functions such as `rename` and `chdir` work as you'd expect them to, and the `unlink` function deletes a file from the underlying file system
- When working with *files* in Perl, we use *filehandles* (as we has already seen)
- Working with *directories* requires the use of *directory handles*
- With filehandles we use `open`, `read`, and `close`
- With directory handles we use the `opendir`, `readdir`, and `closedir` functions

pod2html.pl

- This example shows a potential use for the three directory handle functions:

```
!/usr/bin/perl -w

sub pod2htmlDoIt
{
  my $file = shift;

  print "Processing $file ... ";
  $cmd = "man2html $file ";
  chop ($file);  # remove the 1 char from eol.
  chop ($file);  # remove the . char from eol.
  $cmd = $cmd . " > $file.html";
  system $cmd;
  print "\n";
}

opendir PERLDOCS, "/usr/barryp/perldocs";

@files = readdir PERLDOCS;

closedir PERLDOCS;

foreach $currentFile (@files)
{
  pod2htmlDoIt ($currentFile) if $currentFile =~ /^perl/;
}
```