Command Line Switches

Introduction

- Options are also called ***switches***
- They can turn ON or turn OFF different behaviors
- The most frequent way to specify a command-line options is on the command line.

How Are the Options Specified? – 1/2

- Perl's options are specified using a dash and then a single character followed by arguments

```
perl –w script.pl
```

How Are the Options Specified? – 2/2

- **We can also specify command-line options inside your script file using the #! line**
- Perl starts parsing the #! switches immediately after the first instance of "perl" on the line.
- If we started our script with this line:

```
#!/bin/perl –w
```

- Then Perl will run with the -w option in effect.

The shebang line – 1/2

- Almost every Perl program starts out like this:

  #!/usr/bin/perl

- This is a UNIX construct, which tells a shell that executes the file directly, what program to pass the rest of the input to.

The shebang line – 2/2

- That is, if we had a program containing

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

as its first line, and executed it as

```
perl -l program.pl
```

both the **-l and -T switches are used, but -l is used first.**

–w switch

- This **turns on warnings** that Perl will then give you if it finds any of a number of problems in your code.

```
$ perl -w test.pl
```

-c switch

- This command-line switch allows us to **check the given file for syntax errors**

perl -c try.pl

-e switch

- This command-line switch **allows us to run code from the command line**

```
$ perl -e 'print "Hello\n"'
Hello

$ perl -e 'print "Number=2*2\n"'
Number=4
```

- Useful for small programs, quick calculations, and in combination with other switches

-n switch – 1/3

- Perl's -n switch **allows us to run a program** (usually specified with -e) **against every line on standard input**.

  $ perl -n -e 'some code' file1

- Then Perl will interpret that as:

```
while (<>)
{
        # your code goes here
}
```

- Each line of the input files will be put, in turn, into $_ so that you can process it.

-n switch – 2/3

$ perl -n -e 'print "$. - $_"' file

- This gets converted to:

```
while (<>)
{
        print "$. - $_"
}
```

- This code prints each line of the file together with the current line number.

-n switch – 3/3

```
$ cat /etc/passwd | perl -e 'while (<>) { if
  (/^(\w+):/) { print "$1\n"; } }'

  root     ...

  $ cat /etc/passwd | perl -n -e 'if
  (/^(\w+):/) { print "$1\n" }'

  root     ...
```

-p switch – 1/5

- This option always prints the contents of $_ each time around the loop.

```
while (<>)
{
    # your script
}
continue
{
    print;
}
```

-p switch – 2/5

- It uses the little-used continue block on a while loop to ensure that the print statement is always called.

- **If both -n and -p are specified on the command line, the -p option will take precedence**.

-

-p switch – 3/5

```
$ perl -p -e '$_ = "$. - $_"'
```

- In this case there is no need for the explicit call to print as -p calls print for us.

-p switch – 4/5

- If we combine the **-i switch**, Perl will **edit our file in place.**

- So, to convert a bunch of files from DOS to UNIX line endings, you can do this:

$ perl -p -i -e 's/\r\n/\n/' file1 file2

i.e. \r\n will be converted into \n

-p switch – 5/5

- To print the first word of each line, use this command line:

  perl -p -e "s/\s*(\w+).*/$1/;" test.dat

Using the -0 Option - 1/6

- The -0 (zero) option will let us **change the record separator**.
- This is useful if our records are separated by something other than a newline.
- Say, let us use the example of input records separated by a dash instead of a newline.
- First, we need to find out the octal value of the dash character.

Using the -0 Option - 2/6

- The easy way to do this is to covert from the decimal value, which will be displayed if we run the following command line.

perl -e "print ord('-');"

- This program will display 45. Converting 45_{10} into octal results in 55_8

Using the -0 Option - 3/6

- Next, we will need an input file to practice with the following data held in a test file, ***test.dat***:

  Jack-John-James-Tom

- A program that reads the above data file using the diamond operators is now developed:
  1. The program will use the dash character as an end-of-line indicator.
  2. We set the record separator to be a dash using the #! switch setting method.
  3. Open a file for input.

Using the -0 Option - 4/6

4. Read all of the records into the *@lines* array.
5. One element in *@lines* will be one record.
6. Close the file.
7. Iterate over the *@lines* array and print each element.

Using the -0 Option - 5/6

```
#filename: use_O_option.pl
#!/usr/bin/perl -0055
open(FILE, "test.dat");
@lines = <FILE>;
close(FILE);
foreach (@lines)
{
        print("$_\n");
}
```

Using the -0 Option - 6/6

- Run the file

  perl use_O_option.pl

- This program will display:

  Jack-
  John-
  James-
  Tom-

- Notice that the end-of-line indicator is left as part of the record.

Using the -i Option – 1/3

- The -i option lets us **modify files in-place**.
- This means that **Perl will automatically rename the input file and open the output file using the original name.**
- We can force Perl to create a backup file by specifying a file extension for the backup file immediately after the -i. Example, **-i.bak.**
- **If no extension is specified, no backup file will be kept**.

Using the -i Option – 2/3

- One of the more popular uses for the -i option is to **change sequences of characters**.

- However, using command-line options you can do it like this:

  perl -p -i.bak -e "s/harry/tom/g;" test.dat

- This command-line will change all occurrences of "harry" to "tom" in the test.dat file

Using the -i Option – 3/3

perl –p –i –e “s/-/:/g” test.dat

Output

Jack:John:James:Tom

Using the -s Option - 1/3

- The -s option lets you **create own custom switches.**
- **Custom switches are placed after the script name but before any filename arguments**.
- **Any custom switches are removed from the @ARGV array.**

Using the -s Option - 2/3

- **Then a scalar variable is named after the switch is created and initialized to 1.**

- For example, let's say that we want to use a switch called

  -useTR in a script :

Using the -s Option - 3/3

```
if ($useTR)
{
        print "useTR=$useTR\n";
}
```

- Execute this program using this following command line:

  perl -s use_S_option.pl -useTR

- and it would display:

  useTR=1

–T switch

- This option puts Perl into "taint mode."

- In this mode, Perl inherently distrusts any data that it receives from outside the program's source -- for example, data passed in on the command line, read from a file, or taken from CGI parameters.

$ perl -c test.pl

Command Line Arguments With @ARGV

- Perl command line arguments stored in the special array called **@ARGV**
- Use **$ARGV[n]** to display argument.
- Use **$#ARGV** to get total number of passed argument to a perl script.

perl args.pl one two three

Command Line Arguments With @ARGV (cont..)

- We can print one, two, three command line arguments with print command:

```
print "$ARGV[$0]\n";
print "$ARGV[$1]\n";
print "$ARGV[$2]\n";
```

- Or just use a loop to display all command line arguments

Command Line Arguments With @ARGV (cont..)

```
#!/usr/bin/perl -w
if ($#ARGV != 2 )
{
    print "usage: mycal number1 op number2\neg: mycal 5
    + 3  OR mycal 5 - 2\n";
    exit;
}
$n1=$ARGV[0];
$op=$ARGV[1];
$n2=$ARGV[2];
$ans=0;
```

Command Line Arguments With @ARGV (cont..)

```
if ( $op eq "+" )  {
    $ans = $n1 + $n2;
}
elsif ( $op eq "-") {
    $ans = $n1 - $n2;
}
elsif ( $op eq "/") {
    $ans = $n1 / $n2;
}
```

Command Line Arguments With @ARGV (cont..)

```
elsif ( $op eq "*") {
   $ans = $n1 * $n2;
}
else  {
   print "Error: op must be +, -, *, / only\n";
   exit;
}
print "$ans\n";
```

Command Line Arguments With @ARGV (cont..)

- Save and run script as follows

```
$ chmod +x mycal.pl
$ ./mycal.pl
$ ./mycal.pl 5 + 3
$ ./mycal.pl 5 – 3
$ ./mycal.pl 5 \* 3
```

- Note: * need to be escaped under UNIX shell.

perl –M /-m

- Perl's -M switch **allows us to use a module from the command line**.

- It's also a convenient shortcut with -e if we need to include a module:

```
$ perl -e 'use Data::Dumper; print Dumper( 1 );'
$VAR1 = 1;
$ perl –MData::Dumper -e 'print Dumper( 1 );'
$VAR1 = 1;
```

-m examples

```
# What version of CGI do I have?
$ perl -MCGI -le'print $CGI::VERSION'
2.89


# Some modules are meant for the command line
$ perl -MCPAN -e'install "Module::Name"'


# Text::Autoformat exports autoformat() by default
$ perl -MText::Autoformat -e'autoformat'
```