

IT441

Network Services Administration

Perl: *File Handles*

Comment Blocks

- Perl normally treats lines beginning with a # as a comment.
- Get in the habit of including comments with your code.
- Put a comment block at the beginning of your code which includes...
 - your name,
 - the name of the module,
 - date written, and
 - the purpose of the code.

Alternative String Delimiters

q// : single quoted string

qq// : double quoted string

- In **qq//** , the **//** can be replaced with any other non-alphanumeric character provided you use the same character on both ends of the string
- More generally, **delimiters** – the symbols marking boundaries to parts of something -- can be tricky.
- Make sure you are using them correctly!

Operators on Strings and Numbers

```
$a = "123"
```

```
$b = "456"
```

- What do we get if we write this line of code?

```
print $a + $b;
```

- How about this line of code?

```
print $a . $b;
```

Math Operators

**

Exponentiation

-

Unitary Negation

*

Multiplication

/

Division

%

Modulo (Remainder)

+

Addition

-

Subtraction

- These can be combined with **=**, creating a special assignment operator to
 - Perform the operation with
 - the current value of the left-hand side variable
 - and the expression on the right-hand side
 - and store the result back into the variable

Example: $\$x += y \rightarrow \$x = \$x + y$

What is an Algorithm

- In mathematics and computer science, an algorithm is
 - an effective method
 - expressed as a finite list
 - of well-defined instructions
 - ❖ for calculating a function.
- Algorithms are used for calculation, data processing, and automated reasoning.
- In simple words, an algorithm is a step-by-step procedure for calculations.

File Handles

- To use a file, we need to attach a filehandle to it.
- More generally, we can think of a handle as a way for a program to access some external resource
- It mediates/manages interactions between the program and the resource.
- Thus, a filehandle allows us to read from and write to files.

Examples include:

- File descriptors in Linux, such as 0 (standard input), 1 (standard output), and 2 (standard error)
- In object-oriented programming, a "file object"

File Handles

- In Perl, it will be a variable, which we set up using the `open` statement with three parameters:
 - The **filehandle**, such as `OUT1`
 - The **mode**: read (`<`), write (`>`), or append (`>>`)
 - The file **path**: either *relative* (to *cwd*) or *absolute*
- A simple example:

```
open ( OUT1 , > , 'test out.txt' );
```
- If we open it, we want to **close** it after we are done:

```
close ( OUT1 );
```
- By convention, a *filehandle* is coded in **ALL CAPS**

When File Handles Fail...

- We want to know for sure that we were successful opening the file, so we include a test:

```
open ( OUT1 , '>' , 'test out.txt' ) or die $!;
```

- **\$!** is a *special variable* in Perl. It would be an error message, from the system (e.g., if the opening failed).
- What are some reasons we might **fail** to open a file, in one mode or another?
- See **goodopen.pl** and **badopen.pl** in the textbook
- Also, review the **die** function and its usage

Using a filehandle

- When we use the print function, we are always printing to a handle, such as standard output:

- Therefore, `print "Hello World!\n";`

- is really `print STDOUT "Hello World!\n";`

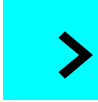
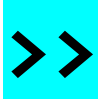
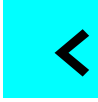
- We could, in fact, print to some *other* destination

```
print OUT1 "Hello World!\n";
```

- To read from a *filehandle*, you wrap it in *angle brackets*.

```
chomp ($in = <IN1>);
```

I/O Redirectors and Perl File-Opening Modes

- Remember what the redirectors do:
 -  **>** : redirect output to ... , which overwrites the file, if it exists
 -  **>>** : append output to ... or create the file, if it doesn't exist
 -  **<** : take input from ...
- You will also use these (in string form, surrounded by quotes) to indicate **mode** (read, write, or append) when opening a file

A Practical Example

```
#!/usr/bin/perl
```

```
open (IN1, '<', 'input.txt') or die $!;  
open (OUT1, '>>', 'out_skip.txt') or die $!;  
while (<IN1>) { # loops until end-of-file  
    chomp $_;  
    print OUT1 "Hello $_!\n\n";  
}  
close (IN1);  
close (OUT1);
```

The Diamond Operator: <>

- We have used the (empty) diamond operator <> before, in the context of getting user input.
- For standard input specifically, we would use <STDIN>
- However, <> does more...
- In Perl, there is a special array called @ARGV which is intended to hold command line arguments to a script.
 - If you provide no arguments, @ARGV is empty
 - If, however, you include arguments, they are stored in @ARGV as whitespace-delimited tokens.

The Diamond Operator: <>

- If **@ARGV** is non-empty, each token in the array will be treated as a filename, then <> will read in
 - the first line of the first file, until
 - the last line of the last file
- If **@ARGV** is empty (no command line arguments), then <> will read in from standard input, line-by-line, until it gets the eof signal – **Ctrl+D**
- Depending on how and when you use <>, it will behave differently...

Using Pipes in Perl

- You can make Linux-style pipes work with a Perl program.
 - Execute another script, and pipe its contents into an input file handle for this one

```
open ('PROG' , '-|' , './other_script.pl file.txt')
```

This is the same as :

```
./other_script.pl file.txt | ./this_script.pl
```

- Start up another script, and send output from this one into the other, as the other's input.

```
open ('PROG' , '|-' , './other_script.pl file.txt')
```

This is the same as :

```
./this_script.pl | ./other_script.pl file.txt
```

File Tests as Conditions

- Files may differ in size, type, permissions, etc.
- To that end, you will often want to check the file before deciding how to proceed. This code checks to see if student has a **.forward** file...

```
if (-e "/home/$student_id/.forward") {  
    open ('IN' , '<' , "home/$student_id/.forward") ;  
    $email = <IN>;  
    close (IN) ;  
}  
else { $email = "$student_id@cs.umb.edu" ; }
```


File Tests as Conditions

- From the textbook:

Table 8-1. *File Test Operators*

Test	Meaning
-e	True if the file exists
-f	True if the file is a plain file—not a directory
-d	True if the file is a directory
-z	True if the file has zero size
-s	True if the file has nonzero size—returns size of file in bytes
-r	True if the file is readable by you
-w	True if the file is writable by you
-x	True if the file is executable by you
-o	True if the file is owned by you