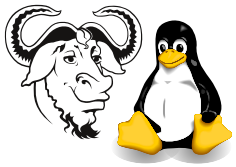

Systems and Network Management

Summary of Perl

Contents

1 Main Topics

Shebang Each Perl program begins with a "shebang":

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

It tells the operating system which interpreter to use to execute the program.

You can add options to this, such as the `-w` above, which switches on additional warnings. I strongly recommend always using this while developing the program.

1.1 Variables, Operators

Do `perldoc perlop` and `perldoc perldata`

Scalars and non-scalars There are two categories of variables: *scalars* and *non-scalars*.

- scalars have a single value, such as `"a string"`, and
- non-scalars have a list of values, such as `( 1, 2, "a string" )`

Non-scalars There are two types of non-scalars: *arrays* and *hashes*.

- *arrays* are much like arrays in Java or C (though much more versatile).
- *hash*es are like arrays that are indexed by strings, a bit like `java.util.Hashtable`, but simpler and more flexible.

`$`, `@` and `%` Scalar variable values always start with a `$`, such as `$var = 1;`

Arrays variable values always start with a `@`, such as `@array = (2, 4, 6);`

Hash variable values always start with a `%`, such as `%hash = ( NL => 'Netherlands', BE => 'Belgium' );`

Note that it is a *value*. For example, in `@array`, there is a scalar value `$array[0]`, and in `%hash`, there is a scalar value `$hash{BE}`.

Variable Interpolation: A variable can be put right into a string like this: `"The value of \$var is $var.\n"`

If you print that string, the value of `$var` will be printed in the string, instead of the four characters `$var`. Notice that, just as in C, the backslash hides the special meaning of special characters such as `$`.

Input, Output: You can read from standard input like this: `my $value = <STDIN>;` Note that this will include the newline character at the end. To remove it, do: `chomp $value;`

You can write to standard output with `print`. `print` takes a list of strings: `print "The product of $a and $b is ", $a * $b, "\n";`

Operators: Perl has all the operators of C, in the same priority as in C. In particular, the dot operator (mentioned below) and the repetition operator '`x`' are special to Perl. See `perldoc perlop` for details; the '`x`' operator is under "Multiplicative Operators."

Note Perl also has special operators for comparing strings:

Comparison	*Numeric*	*String*
equal	`==`	`eq`
not equal	`!=`	`ne`
greater than	`>`	`gt`
less than	`<`	`lt`
greater than or equal to	`>=`	`ge`
less than or equal to	`<=`	`le`

Joining strings: The dot "`.`" operator joins strings together, like '`+`' in Java. Example: `print "circumference = " . 2 * $pi * $radius . "\n";`

`use warnings;` Turns on all compile-time warnings. Let the compiler find the bug, not your customer. You can instead add "`-w`" to the "shebang" as described above.

`use strict;` Turns on compile-time checks for lots of possible error conditions, such as undeclared variables, and other possible typing errors. I strongly recommend using this in all your programs that are longer than half a page.

`my` and `our`: are used to declare local variables and global variables, respectively. Necessary if you put
`use strict;`
in your program.

1.2 Statements

See `perldoc perlsyn`

Most statements can be written almost exactly the same as in C. If you are not sure how to do something in Perl, try writing it as if it were a C program.

`if`, `while`, `for` need braces: You must use braces in a normal `if` statement, unlike in C or Java.

`if` statement: The `if` statement is similar to C or Java, except that there is a keyword "`elsif`":

```
if ( $age > $max ) {
    print "Too old\n";
} elsif ( $age < $min ) {
    print "Too young\n";
} else {
    print "Just right\n";
}
```

`unless` statement: The `unless` statement is just like the `if` statement, except that the block is executed if the condition is *false*:

```
unless ( $destination eq $home ) {
    print "I'm not going home.\n";
}
```

`for` loops: There are two types of `for` loop, one as in C and Java, the other is more useful in Perl:

```
for ( $i = 0; $i < $max; ++$i ) {
   $sum += $array[ i ];
}
```

But this `for` loop is much more useful. Here is an example that adds 1 to each element of an array:

```
foreach $a ( @array ) {
    ++$a;
}
```

Notice that `$a` here is made a reference to each element of the array, so changing `$a` actually changes the array element. You can write "`for`" or "`foreach`", Perl won't mind.

Special variable: `$_`: this special variable appears as the default argument of many built-in functions, including `print`, so this `foreach` loop prints all elements of `array`:

```
foreach ( @array ) {
    print;
}
```

`while` **loops:** are rather like in C or Java.

```
while ( $i < $max ) {
    ++$i;
}
```

Reading each line from input files: We often use a `while` loop to read each line from each of the files listed on the command line:

```
while ( <> ) {
    print $_;
}
```

What this does is:

- If there are command line parameters to this script, then it assumes that they are file names, and opens each in turn, and loops once for each line in the file, setting `$_` to that line
- Otherwise, it reads standard input, setting `$_` to each line.

Note that you could achieve the same result as above with:

```
print <>;
```

What `while ( <> )` Does: The loop:

```
while ( <> ) {
    ⟨statements...⟩
}
```

does something like this:

```
if there are no command line arguments,
    while there are lines to read from standard input
        read next line into $_
        execute ⟨statements...⟩
else
    for each command line argument
        open the file
        while there are lines to read
            read next line from the file into $_
            execute ⟨statements...⟩
        close the file
```

Reading from standard input only: is very similar to using `<>`. This example prints each line of standard input:

```
while ( <STDIN> ) {
    print $_;
}
```

`chomp` Remove newline from end of a string. Normally when you read a line from a file, the newline is on the end of the string. If you don't want it, `chomp` it:

```
while ( <> ) {
    chomp $_; # or just chomp; since $_ is default argument.
    ...       # process the line $_
}
```

`next` and `last` `next` is like `continue` in C; `last` is like `break` in C.

1.3 Backwards Statements

See `perlop perlsyn`

A common way of writing statements in Perl is to put an `if`, `while` or `foreach` modifier after a simple statement. In other words, you can put a simple statement (i.e., with no braces), and put one of these aftewards:

```
if EXPR
unless EXPR
while EXPR
until EXPR
foreach EXPR
```

Note that `unless` statement

⟨*statement*⟩ `unless` ⟨*condition*⟩`;`

corresponds to:

⟨*statement*⟩ `if ! (` ⟨*condition*⟩ `);`

Here are some examples:

```
print $1 if /(\d{9})/;
```

is equivalent to:

```
if ( /(\d{9})/ )
{
    print $1;
}
```

Another example:

```
# print unless this is a blank line:
print unless /^\s*$/;
```

is equivalent to

```
if ( ! /^\s*$/ ) {
    print;
}
```

1.4 Array operations

The documentation for these is in the very loo–oong document `perlfunc`, and is best read with `perldoc -f` ⟨*Function*⟩

`push` add a value at the end of an array, e.g.,

```
my @array = ( 1, 2, 3 );
push @array, 4;
# now @array contains ( 1, 2, 3, 4 )
```

Do `perldoc -f push`

`pop` remove and return value from end of an array

```
my @array = ( 1, 2, 3 );
my $element = pop @array;
# now @array contains ( 1, 2 ) and $element contains 3
```

Do `perldoc -f pop`

`shift` remove and return value from the beginning of an array, e.g.,

```
my @array = ( 1, 2, 3 );
my $element = shift @array;
# now @array contains ( 2, 3 ) and $element contains 1
```

Do `perldoc -f shift`

`unshift` add value to the beginning of an array, e.g.,

```
my @array = ( 1, 2, 3 );
unshift @array, 4;
# now @array contains ( 4, 1, 2, 3 )
```

Do `perldoc -f unshift`

1.5 `split` and `join`

Do `perldoc -f split` and `perldoc -f join`.

`split` splits a string into an array:

```
my $pwline = "nicku:x:500:500:Nick Urbanik:/home/nicku:/bin/bash";
my ( $userid, $pw, $userid_number, $group_id_number,
     $name, $home_dir, $shell ) = split /:/, $pwline;
```

Another application is reading two or more values on the same input line:

```
my ( $a, $b ) = split ' ', <STDIN>;
```

`join` is the opposite of `split` and joins an array into a string:

```
my $pwline = join ':', @pwfields;
```

1.6 Executing External Programs

Perl provides many ways of doing this, but we just used the `system` built-in function. In the laboratory in creating user accounts, I have written solutions that pass an array to `system`:

`system`: An example

```
my @cmd = (
           'useradd',
           '-c', "\"$name\"",
           '-p', $hashed_passwd,
           $id
          );
print "@cmd\n";
system @cmd;
```

This also works:

```
system "useradd -c \"$name\" -p \"$hashed_passwd\" $id";
```

The difference is that the second form is usually passed to a command shell (such as `/bin/sh` or `CMD.EXE`) to execute, whereas the first form is executed directly.

Was the command successful? You can tell if the command was successful by checking that the return value was zero:

```
if ( system( "useradd -c \"$name\" -p \"$hashed_passwd\" $id" ) != 0 ) {
    print "useradd failed";
    exit;
}
```

This is usually written in Perl more simply using the built in function `die`, and the `or` operator:

```
system( "useradd -c \"$name\" -p \"$hashed_passwd\" $id" ) == 0
   or die "useradd failed";
```

backticks: Perl provides command substitution, just like in shell programming, where the output of the program replaces the code that calls it:

```
print `ls -l`;
```

Note that you can write `qx{...}` instead:

```
print qx{df -h /};
```

1.7 Subroutines

Do `perldoc perlsub`

Subroutines calls pass their parameters to the subroutine in an list named `@_`. It is best to show with an example:

```
#! /usr/bin/perl -w
use strict;
sub product
{
    my ( $a, $b ) = @_;
    return $a * $b;
}
print "enter two numbers on one line: a b ";
my ( $x, $y ) = split ' ', <STDIN>;
print "The product of $x and $y is ", product( $x, $y ), "\n";
```

Note the following:

parameters: parameters are passed in one list `@_`. If you are passing one parameter, then the builtin function `shift` will conveniently remove the first item from this list, e.g.,

```
sub square
{
    my $number = shift;
    return $number * $number;
}
```

There is another trivial example with two parameters above.

return The `return` builtin function works as in C. The conventional way to return the *false* value is to call `return` with an empty list:

```
return;
```

automatic variables: Use `my` to define variables that are used only within the subroutine.

2 Regular Expressions

Do `perldoc perlrequick`, `perldoc perlre`

We spent most time in the laboratory and in the lectures studying and using regular expressions. Regular expressions are an important part of Perl. Regular expressions just been incorporated into Java 1.4, and are based directly on Perl regular expressions. Regular expressions are also used in many other programming languages, text editors, programs... even Microsoft Word. They will be an important part of the exam.

You should be familiar with character classes, matching the beginning and end of a line, and selecting part of a match with `$1`.... At an absolute minimum, you *must* be familiar with the application of:

`\`	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
`*`	Match 0 or more times
`+`	Match 1 or more times
`?`	Match 0 or 1 times

2.1 Searching with Regular Expressions

This example shows the basic syntax:

```
$string =~ /pattern/;
```

searches `$string` for the regular expression `pattern`, giving true if it is found, false otherwise. We often put this into an `if` statement to see if it matches.

To test that a pattern does *not* match, you can use the `!~` operator:

```
if ( $string !~ /pattern/ ) {
    print "pattern was not found in $string\n";
}
```

See the solutions to the tutorial questions for examples of the use of regular expressions, particularly with using parentheses to select part of the match, and using `$1`, `$2`, ... to capture the match in the parentheses.

3 Perl Regular Expression Symbols: extracted from `perlre` manual page

[Note: this table will be provided in the exam].

`\`	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
`*`	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
`\w`	Match a "word" character (alphanumeric plus "_")
`\W`	Match a non-"word" character
`\s`	Match a whitespace character
`\S`	Match a non-whitespace character
`\d`	Match a digit character
`\D`	Match a non-digit character
`(?:pattern)`	This is for clustering, not capturing; it groups subexpressions like "`()`", but doesn't make back references as "`()`" does.

Regular Expression Modifiers

`i` Do case-insensitive pattern matching.

`g` Match globally, i.e., find all occurrences.

`x` Extend your pattern's legibility by permitting whitespace and comments.