IT441

Network Services Administration

Data Structures:

Lists

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.

Comment Blocks

```
#!/usr/bin/perl -w
#
#Module Name:  helloWorld.pl
#
#Written by Alfred J Bird, Ph.D., NBCT
#Date Written - 21 September 2011
#Purpose: To print out the string "Hello
 world!"
#
#Date Modified - 25 September, 2011
#Purpose of modification:  To fix spelling
 errors.
#Modified by:  Al Bird
#
print "Hello world! \n";
```

Data Types

- Remember there are three basic data types in Perl
  - Numeric
  - String
  - Boolean (Logical)
- These, of course, fall under the heading of ***scalars***
- I differentiate between data *types* and data *structures*. Not every author or teacher does. Some books use the terms interchangeably so watch out!

Data Structures

- There are three types of structures in Perl for organizing program data
  - ***Scalars*** - A single data value (number, string, etc.)
  - ***Arrays*** - An ordered sequence of data values
  - ***Hashes*** - A set of paired data values, where one (the "key") is used to look up the other (the "value")
- Each structure has it own syntax for variable names...

```
$scalar    @array    %hash
```

- This syntax can be tricky, if you're not careful...

Scalars

- We talked about scalars in the past.
- Scalars are a data type that contain ***one*** element.
  - It can be a *number* such as `1`
  - It can be a *string* such as `"Hello World! \n"`
  - It can be a *boolean* value of **true** or **false**
- It can be stored in a variable with a name such as `$i`
- It is the most primitive of the data structures.

Lists

- Some authors, teachers, and CS pros ***do not*** consider a list a data structure, but some ***do*** so be careful .
- A list is defined as an *ordered set of scalar values*.
- Lists are delimited by parentheses such as

```
()
(1)
("a")
(1, 2, 3, 4, 5)
("a", "b", "c", "d", "e")
('e', 'd', 'c', 'b', 'a')
```

- Remember that a list is ordered!

Using a List

- You have already been using lists without knowing it.
- When you type the following statement

```
print ("Hello ", "world", "! ", "\n");
```

  You are passing a list to the **`print`** function.

- I have just used a new Perl term, *function*.
- A function is a *subroutine* (a free standing piece of code) or an operator that returns a value and/or does something

Another Way to Create a List

- Given a list we created this way:

```
('Hello', 'world.', 'I', 'am', Al')
```

- We can use another method to create it:

```
qw/Hello world I am Al/
```

- As with earlier, similar operators -- we can use *any* nonalphanumeric character as a separator:

```
qw#Hello world I am Al#
qw&Hello world I am Al&
qw{Hello world I am Al}
```

A Third Way to Create a List

- We can create a list by using a *range*.
  - This list `(1, 2, 3, 4, 5, 6)`
  - Is the same as this list `(1..6)`
- But this will not work:
  - `(6..1)` does not give `(6, 5, 4, 3, 2, 1)`
  - because the *left hand side* must be less than the *rhs*
- To get the list `(6, 5, 4, 3, 2, 1)` using a range, we need to type `reverse (1..6)`
- Try these using a `print` statement!

Printing a List

- Remember that a list is ordered!
  - The elements have a location that can be counted
    - The counting starts with **0** (the **1st** element is number **0**)
- How do we *print* a list?
- What is the result of the following statements?

```
print (qw/a b c d e f g/);
```

- How about this statement?

```
print qw/a b c d e f g/;
```

- First, *predict* the results, and then try them and see what happens.

Printing Individual List Elements

- We can refer to individual elements in a list by using a number in square brackets `[]` after the list.
- What is the result of the following statement?

```
print ((qw/a b c d e f g/)[2]);
```

- How about this statement:

```
print (('a', 'b', 'c', 'd', 'e', 'f', 'g') [3]);
```

- First, *predict* the results, and then *try* them and see what happens.
- You can put *a scalar variable* into the brackets

```
$i = 3;
print ((qw/a b c d e f g/)[$i]);
```

A List Slice

- We can refer to a range inside the braces.
- What do we get when we run the following statement:

```
print ((qw/a b c d e f g/)[2..4]);
```

- First, predict the results, and then run the statement.
- What about this statement?

```
print ((qw/a b c d e f g/)[3..1]);
```

Extras

- What do you think will happen if you enter the following code?

```
print (('z', 'x', 'c', 'v', 'b', 'n', 'm')[-1]);
```

- First, make a prediction, and then run the code.
- How about this code?

```
$i=2.9;
print (('z', 'x', 'c', 'v', 'b', 'n', 'm')[$i]);
```

- First, make a prediction, and then run the code.

Another Data Structure

- The problem with a list is that it cannot be named!
- You cannot, for example, do the following:

```
$the_data = qw/a b c d e f g/;
```

- We need to retype the list every time we want to use it.
- To solve this difficulty we have another data structure called an ***array***
  - We can give an array a name that starts with a **@**
  - This means that we can *reference* it and perform more *operations* on it…

Arrays

- An ***array*** is a data structure that has the characteristics of a list but can be *named*!
- To store a scalar literal into a variable we use an assignment statement

```
$a = 1;
```

- To store a list into an *array*, we do the same thing:

```
@a = (1,2,3,4,5);
@l = ('a', 'b', 'c', 'd', 'e', 'f');
@m = qw<az x c v b n m>;
```