IT441

Network Services Administration

Data Structures:

Arrays

Data Types

- Remember there are *three* basic data types in Perl
  - Numeric
  - String
  - Boolean (Logical)

- 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

- In PERL there are three types of data structures:
    - Scalars
        - ***Single*** values: Number, string, Boolean
        - The most basic structure
    - Arrays - ***Sequences*** of values
    - Hashes - Key-value ***pairs***
- Each structure has it own naming syntax.

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

Lists

- We talked about lists already.
- 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*: `0`, `1`, `2`, `3`,…

Another Data Structure

- As we mentioned, a list cannot be named with a variable
- However, we have a data structure called an array
  - An array, too, is an ordered sequence of values
  - We can give an array a name that starts with a @
- To make and name, an array we assign it to a variable:

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

Accessing Individual Elements

- How do we access an individual element in an array?
  - Just like we did in a list.
- Using a list if we code:

```
print (('now', 'is', 'the', 'time')[2]);
```

  - It will print out **`the`**
- Likewise, if we define an array:

```
@s = ('now', 'is', 'the', 'time');
print @s[2];
```

  - The print statement will also print out **`the`**

Scalar vs. List Context

- What about `print $s[2];` ? What will it print out?
- Why does the statement `print $s[2];` work?
  - Use the prefix for what you ***want*** -- not what you have.
  - This is referred to as *list vs. scalar* context, and it may well become a very important concept later…
- When using an input filehandle in an *assignment* statement, the *type of variable* will make a difference:

```
$scalar_var = <IN1> ;   # Gets next line as scalar value
@array_var  = <IN1> ;   # Gets all (remaining) lines and
                         # puts them in an array
```

Array Functions

- How do we add data to an array?

```
@array = (@array, $scalar); #is one way!
```

- But there is another way!!

```
push @array, $scalar; #will do the same thing!
```

- `push` will append the value in `$scalar` to the top of `@array`
  - We say the end of the array (i.e., highest index) is the "top"
  - And the front (i.e., lowest index) is the "bottom"
- Likewise, `pop` will take the last/top value in an array and do something with it.

```
$scalar = pop @array
```

Array Functions

- **`push()`** and **`pop()`** act on the top of an array (the highest indexed end)
- **`shift()`** and **`unshift()`** act on the bottom of an array and perform the same function.
- We already know what reverse() does…right?
  - Note that **`reverse`** does not change the original array
  - Rather, it is more like create a new array, with the same values, only in the reverse order.
  - You can name the reversed array: **`@rev_arr = reverse(@arr);`**

Array Functions

- You can use **push** , **pop** , **shift** , and **unshift** in order to implement stacking and queuing logic
  - *Stack* items are accessed **LIFO** (*last in, first out*)
    - One example would be a stack of cafeteria trays
    - Your code would act on ***one*** of ends
      - You can **push** onto and **pop** off of the end, or…
      - **unshift** onto and **shift** off of the front.
  - *Queue* items are accessed **FIFO** (*first in, first out*)
    - Standing in line is a familiar example of a queue
    - Your code would have to act on opposite ends
      - **push** onto the end and **shift** from the front, or…
      - **unshift** onto the front and **pop** from the end.

Array Functions

- Another function is `sort()`
  - You may have used it in a previous project…
  - What do you think it does?
- One thing you want to keep in mind is whether you want data sorted *as strings* or *as numbers*
  - By default, `sort()` will sort the values as **strings**…
    - which causes a sequence like `1, 2, 11, 24, 3, 36, 40, 4`…
    - to become `1, 11, 2, 24, 3, 36, 4, 40`
  - A "stringy" ordering considers a shorter string (e.g., "book") to come before a longer one starting with the same ("bookcase")

Array Functions

- You can *tell* the `sort()` function *how* to sort the items
- For example…

```
@unsorted = (1, 2, 11, 24, 3, 36, 40, 4);
print sort { $a cmp $b } @unsorted;
```

- …gives us `1 11 2 24 3 36 4 40` , whereas…

```
@unsorted = (1, 2, 11, 24, 3, 36, 40, 4);
print sort { $a cmp $b } @unsorted;
```

- gives us `1 2 3 4 11 24 36 40` – which, of course, is probably what we *really* want!

The Overall World of PERL

- What is a **namespace**?
  - For starters, consider the two parts of the word:
    - "name" : What we call a thing – a value, a data structure, a function, etc.
    - "space" : An environment or context, such an area of a program
  - Thus, we can think of a "namespace" as a *context* where specific *names* have specific *meanings*.
  - Depending upon your namespace, the same name could have different meanings.
  - A full name for something would consist of:
    - A namespace
    - And a local name

The Overall World of PERL

- Consider the machines on the IT Lab LAN.
  - Local names: it20, it25, itvm26-1a, and so forth. Within the LAN, you can access the machines using just those names.
  - Fully-qualified names:
    - it20.it.cs.umb.edu
    - itvm26-1a.it.cs.umb.edu
  - As such, we could say it.cs.umb.edu is a namespace where those names refer to those machines
- Another example: **`/home/johndoe/it441/ex/ex2/typescript`**
  - *Local* name: **`typescript`** (i.e., the *filename*)
  - Name*space* : **`/home/johndoe/it441/ex/ex2`** (i.e., the *path*)

The Overall World of PERL

- What is a **package**?
  - Packages are Perl files, with a **`.pm`** extension, that are considered a separate namespace.
  - A package, then, is just *a group of related "things"* – scalars, arrays, hashes, and subroutines – *for a specific purpose*.
  - Once a package is included in a **`.pl`** file (invoking **`use`**) and you want to use one of the variables of the package, you may have to use the *scope resolution operator*

```
$package::variable_1
```

The Overall World of PERL

- What is a **module**?
  - Modules are packages which have the capabilities of
    - exporting selective subroutines, scalars, arrays, and hashes of the package
    - to the namespace of the main package itself.
  - Therefore, to the interpreter, these look *as though* the subroutines are part of the main package itself…
  - …so there is no need to use the scope resolution operator while calling them.
- This, of course, is partly why we set up ***CPAN*** in Exercise 2!

CPAN

- Why use PERL?
- What other languages could we use?
  - Ruby, Python, Bash scripting.....
- Other people have already done it:

  http://www.perl.org

  http://www.cpan.org

  http://www.perlmonks.org
- As programmers and IT people are fond of saying...

"Don't reinvent the wheel!"

Special Directives

- You have, perhaps, seen (or used) things like the following →

```
#!/usr/bin/perl -w
use strict;
use warnings;
```

- *Warnings* concern scenarios where part of our code
  - could be problematic at some point in execution...
  - but won't necessarily prevent execution
- Both `-w` and `use warnings;` make the interpreter print a warning message in such cases.
- They behave differently with respect to Perl versions, program scope, flexibility, and other factors

Special Directives

```
#!/usr/bin/perl -w
use strict;
use warnings;
```

- ***use strict;*** is a bit different.
- Here, we are concerned with fostering and maintaining good practice.
- Essentially, it forces you to be diligent when writing code
  - For example, all variables have to be explicitly declared as *lexical* (using ***my***) or as *global* (using ***our***)
  - For new Perl programmers, ***use strict;*** can be like "training wheels" in learning the language
  - For the more experienced, it can guard against coding errors