System Administration for the Web:
Week 7 Lecture Notes
Basic Programming and Perl

October 24, 2005

1 What is Programming?

Most computer scientists would define programming as implementing an abstract algorithm using a programming language to produce a set of instructions that a computer can understand.

I think that's a dull and very difficult to understand definition. Programs are like recipes for making food, and, using that analogy, programming is the process of organizing the production of a certain type of food into a set of consistent and reproducible steps.

Some recipes are quite simple and linear, like those for mixing drinks. You take a certain amount of various ingredients and combine them in a certain order. Other recipes are more complicated and usually require making decisions: to flip or not to flip a steak on a grill or whether to take cookies out of an oven.

Regardless of the complexity of a certain computer program, though, programming is almost always accomplished in 4 steps:

1. Planning
2. Prototyping and Coding
3. Testing and Debugging
4. Release

Step 2 is often repeated after Step 3 upon the discovery of programming errors.

2 Why Program?

Simple: *fame*, *fortune*, or *laziness*.
In the context of system administration, programming usually falls under laziness. As you've seen with shell scripts, programs can automate repetitive tasks and save system administrators a ton of work.

3 Perl

The programming language you will be learning is Perl, the Practical Extraction and Reporting Language. Perl was originally created by Larry Wall, a one-time graduate student at UC Berkeley, and is currently maintained by programmers all over the world. Perl was originally designed to manipulate text files such as log files – hence *extraction* and *reporting* – but has evolved into an extremely-versatile language for everything from shell scripts to corporate websites.

3.1 Perl's Strengths

- Excellent text manipulation
- Fast prototyping
- Flexible syntax
- No artificial limits

3.2 Perl's Weaknesses

- Interpreted language
- Flexible syntax

3.3 How is Perl Used?

- Amazon.com
- MovableType
- Slashdot
- Sweden's pension system
- A server near you

4 Data and Data Structures

Data is anything your program must track, be it user-input, dates, values, quantities, colors, etc. *Scalar data* is the basic unit of data in Perl.

Scalar data can be anything from letters, words, numbers, sentences, paragraphs, punctuation marks, or any combination of these categories. To Perl, scalar data is a merely a stream of characters, such as letters, numbers, spaces, and symbols. Additionally, there is a category of characters that require special representation in Perl through the use of *escape-sequences*; common examples of such characters are found in Table 1.

Scalar data in Perl is held in three types of basic containers or data structures: *variables*, *arrays*, and *hashes*.

Escape-Sequence	Meaning
\n	Newline
\t	Tab
\\	Backslash (\)
\"	Double quote (“)

Table 1: Common Perl Escape-Sequences

NOTE: Perl is a very context-sensitive language. This means that it will try to automatically understand how you wish to use a piece of scalar data. For example, the scalar data “45” could be interpreted as either the number 45 or a string composed of the characters 4 and 5. Unlike other programming languages, where you much always specify the context under which you are using a piece of scalar data, Perl will automatically decide for you depending on what you do to the data.

4.1 Variables

Variables are the simplest type of data structure in Perl; they hold a single piece of scalar data. Variable names begin with a dollar sign (`$`) followed by characters that uniquely identify each variable – `$sample_variable`.

Assignment

Assigns a piece of scalar data to a variable

Operator: `=`

Example: `$sample_variable = “sdf343fd”;`

Result: `$sample_variable = “sdf343fd”`

Concatenation

Joins pieces of scalar data together

Operator: `.`

Example: `$sample_variable = “year” . “2005”;`

Result: `$sample_variable = “year2005”`

Chomp

Removes a newline from the end of a piece of scalar data

Operator: `chomp`

Example: `$sample_variable = “This a sentence.\n”;`
`chomp $sample_variable;`

Result: `$sample_variable = “This is a sentence.”`

Mathematical Functions

Performs mathematical operations on numeric scalar data
Note: Addition is the example presented, but other mathematical operators (`*`, `/`, `-`) work too.

Operator: `+`

Example:
```
$sample_variable = “10”;
$another_variable = $sample_variable + 1;
```

Result:
```
$sample_variable = “10”
$another_variable = “11”
```

4.2 Arrays

Arrays are data structures that hold multiple pieces of scalar data, each *indexed* by a number. Arrays are best described as a numbered list of scalar data pieces. Array names begin with an at-sign (`@`) followed by characters that uniquely identify each array – `@sample_array`. Elements of an array are referenced by a dollar sign followed by the name of the array and the index of the element in square brackets – `$sample_array[0]`. Indices numbering begins at zero.

NOTE: Most variable operations can be performed on individual array elements.

Assignment Method #1

Assigns scalar data to an array

Operator: `=`

Example:
```
$sample_array[0] = “data_piece_0”;
$sample_array[1] = “data_piece_1”;
```

Result:
```
@sample_array =
0 -> “data_piece_0”
1 -> “data_piece_1”
```

Assignment Method #2

Assigns scalar data to an array

Operator: `()`

Example:
```
@sample_array = (“data_piece_0”, “data_piece_1”);
```

Result:
```
@sample_array =
0 -> “data_piece_0”
1 -> “data_piece_1”
```

Pop

Removes the last element of an array and returns it

Operator: `pop`

Example:
```
@sample_array = (“data_piece_0”, “data_piece_1”);
$temp = pop @sample_array;
```

Result:
```
@sample_array =
0 -> “data_piece_0”
$temp = “data_piece_1”
```

Push

Adds an element to the end of an array

Operator: `push`

Example:
```
@sample_array = (“data_piece_0”, “data_piece_1”);
push @sample_array, “data_piece_2”;
```

Result:
```
@sample_array =
0 -> “data_piece_0”
1 -> “data_piece_1”
2 -> “data_piece_2”
```

Shift and Unshift

Removes and adds an element to the beginning of an array, respectively
Similar syntax as `pop` and `push`

4.3 Hashes

Hashes are the most flexible basic data structure available in Perl. They are similar to arrays in that they hold multiple pieces of scalar data, but rather than having elements indexed by a number, they are indexed by words or, more appropriately, *keys*. Hash names begin with a percentage sign (%) followed by characters that uniquely identify each hash – `%sample_hash`. Elements of an array are referenced by a dollar sign followed by the name of the array and the key of the element in curly brackets and quotation marks – `$sample_hash{"key_name"}`.

NOTE: Most variable operations can be performed on individual hash elements.

Assignment Method #1

Assigns scalar data to a hash

Operator: `=`

Example:
```
$sample_hash{"name"} = "stephen";
$sample_hash{"location"} = "berkeley";
```

Result:
```
%sample_hash =
"name" -> "stephen"
"location" -> "berkeley"
```

Assignment Method #2

Assigns scalar data to a hash

Operator: `=, =>`

Example:
```
%sample_hash = (
"name" => "stephen",
"location" => "berkeley",
);
```

Result:
```
%sample_hash =
"name" -> "stephen"
"location" -> "berkeley"
```

Keys

Returns an array of the keys of a hash (in no particular order)

Operator: `keys`

Example:
```
%sample_hash = (
“name” => "stephen",
“location” => “berkeley”,
);
@my_array = keys %sample_hash;
```

Result:
```
@my_array =
0 -> “location”
1 -> “name”
```

4.4 Advanced Data Structures

It is possible to *nest* one type of data structure inside of another data structure or even inside the same type of data structure; that is, it is possible to create arrays of hashes or hashes of hashes. However, the syntax and theory for doing so is beyond the scope of this Perl introduction. Please refer to the command `perldoc perlreftut` or any other documentation on Perl *references*.

5 Control Structures and Comparison Operators

Sometimes it is necessary to make a decision in a program or to perform a set of actions multiple times. *Control structures* provide the facility to do both of these things.

Control structures are blocks of Perl code enclosed in curly brackets whose execution is dependent on some kind of *test(s)*. These tests are usually performed using *comparison operators* to make some sort of comparison, but they can also be the value of some variable. The most commonly used control structures in Perl are `if`, `while`, and `foreach`.

5.1 The if Control Structure

The basic structure of an `if` control structure is:

```
if (TEST) {
     CODE_IF_TRUE
}
```

where `TEST` is some sort of comparison or value, that, if true, results in the execution of `CODE_IF_TRUE`, which can be multiple Perl operations.

5.2 Comparison Operators

Perl provides a group of comparison operators that can be used in test conditions for control structures. Comparison operators are divided into two categories, numeric and string operators (Table 2). It is important that you use the correct type of comparison operator; using the improper type of comparison operator will result in unexpected program behavior.

Comparison	Numeric	String
Equal	==	eq
Not equal	!=	ne
Less than	<	lt
Greater than	>	gt
Less than or equal to	<=	le
Greater than or equal to	>=	ge

Table 2: Numeric and String Comparison Operators

I mentioned earlier that test conditions could also be values. In Perl, any value other than `0` or *null* (an undefined variable) is true. Consequently, you can provide a variable name as a test condition, and, depending upon its value or whether it's defined, the test will return true or false.

5.3 The while Control Structure

The basic structure of a `while` control structure is:

```
while (TEST) {
     CODE_IF_TRUE
}
```

where `TEST` is some sort of comparison or value, that, if true, results in the execution of `CODE_IF_TRUE`, which can be multiple Perl operations. `CODE_IF_TRUE` will be repeated until `TEST` is false, so it is important that `CODE_IF_TRUE` does something to modify the test condition so that the program does not execute `CODE_IF_TRUE` forever – creating an infinite loop.

5.4 The foreach Control Structure

The basic structure of a `foreach` control structure is:

```
foreach $TEMP_VAR (@ARRAY) {
     CODE_PER_ARRAY_ELEMENT
}
```

where `$TEMP_VAR` is the name of a temporary variable that will hold an element of `@ARRAY`. foreach will copy an element of `@ARRAY` into `$TEMP_VAR`, and execute `CODE_PER_ARRAY_ELEMENT` for each – hence the name – element of array. `foreach` control structures are the best way to perform some action on each element of an array.

6 Basic Input and Output

6.1 Input

You can get input from a user by using the assigning the value `<STDIN>` to a variable. For example,

```
$the_user_input = <STDIN>;
```

which will read the input from a user until the Return key is pressed.

NOTE: The value assigned to such a variable will also contain the newline character, since the input is terminated by a Return. Consequently, it is advisable to `chomp` the variable (Section 4.1).

6.2 Output

You can present output to a user by using the `print` command. For example,

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

which will display the string "Hello World!" followed by a newline.

7 A Simple Perl Program

```
#!/usr/bin/perl
print "I'm thinking of a number between 1 and 100.\n";
$guess = -1;
$number = int rand(100) + 1;
while ($guess != $number) {
     print "What is your guess? ";
     $guess = <STDIN>;
     chomp $guess;
     if ($guess < $number) { print "Too low!\n"; }
     if ($guess > $number) { print "Too high!\n"; }
}
print "You got it right!\n";
```