

CS306 – Perl Programming
Spring 2009, Homework Assignment #2
Due: Wednesday, February 4th, 4pm

Guidelines for Submitting Homework

Please follow the same guidelines as for HW1, switching the first part of the filename for each program to “hw2” and the zip file name to lastname-hw2.zip.

Homework #2 (150 total points)

Question 1 (25 points). Why are global variables usually a bad idea in programs? What advantages do lexical variables have over global variables? Do lexical variables prevent you from inadvertently creating problems in higher level scopes (parent scopes)? How about in lower level scopes (child scopes)? (Asked another way, what bad habits do lexical variables succeed in protecting you from doing, and what bad habits can you still get away with even when using lexical variables?)

Notes on programs: Please follow the best practices suggestions provided on the HW1 handout. All of those still apply.

In addition, I expect all programs to contain “use strict;” as the first line of code (after the shebang line and any comments you place at the top of your program). This in turn will force all variables to be limited in scope using “my” to create them as lexical variables. (This is not strictly true [pun intended, and also awful] but is good enough for our purposes).

Program 1: Rewrite the Ticket Line Program (40 points)

Rewrite the Ticket Line Program from HW1, Program 8 to incorporate subroutines. You will add the following three subroutines: `add_person()`, `call_person()`, and `print_line()`. (The fourth case, quitting, is so simple we will continue to implement it at the top level of the program). In your branching logic where you are determining which menu option the user chose, you will now call subroutines instead of handling the task directly at that location.

You must always pass all necessary input data into your subroutines, and return any modified data. That is to say, the following code will not receive credit:

```
# NOT the way to do this assignment!  
my @line = qw /fran joe/;  
my $new_person = "eliza";  
add_person();  
sub add_person {  
    push @line, $new_person;  
}
```

You should pass `@line` and `$new_person` into `add_person()`, and return the modified list representing the line back out of `add_person()`. I would expect to see subroutine calls like this:

```
# assume @line already exists with some elements
```

```
@line = add_person($new_person, @line);  
# note this is safe because perl will evaluate the right side (using  
# the current values of @line) first, before calling the subroutine  
# and writing over @line with the newly modified list being returned
```

Remember to be careful when passing arrays into subroutines. You are limited to one array based on what we have learned thus far, and for safety, it should be the last parameter you pass into the subroutine, if you have other scalars that you also want to pass into the subroutine. (as in the example above).

Program 2: Rewrite School Pride (10 points)

Rewrite the School Pride program (HW1, Program 6) to use a single hash instead of two arrays.

Program 3: Best Sellers (35 points)

A local restaurant is interested in knowing which menu items sell the most, and which menu items make them the most money (of course, these may not be the same items). They have asked you to write a Perl program to help them do this.

The restaurant's register can provide you with a log file of all of the items that were ordered. The log file looks like:

```
chicken sandwich  
2  
9.00  
beer  
3  
3.50  
cheeseburger  
2  
6.00  
soda  
1  
1.00  
bbq pork sandwich  
4  
8.00  
chicken sandwich  
1  
9.00  
turkey club  
2  
7.00  
beer  
2  
4.50  
. . .
```

The records come in sets of three lines. The first line is the item name, the second line is the quantity sold, and the third line is the cost for a single item of that type. In other words, in the first three lines above, two chicken sandwiches were sold for a total of \$18.00.

Your program must be able to process an input list in the format shown above to compute how

many of each item they are selling (a counting task) and also how much money they are making on each item (a running sum task). Both tasks lend themselves particularly well to hashes. You may want hashes like `%count` and `%revenue`.

The output should be a list of the items in alphabetical order, with the quantity sold of each item, and the total money brought in for each item. For the list above, the output would be:

```
bbq pork sandwich: $32.00 (4)
beer: $19.50 (5)
cheeseburger: $12.00 (2)
chicken sandwich: $27.00 (3)
soda: $1.00 (1)
turkey club: $14.00 (2)
```

In other words, they sold the biggest quantity of beer, but they made the most money on bbq pork sandwiches. The formatting of the output is up to you as long as the items are in alphabetical order and the quantity and revenue for each item is listed clearly. You may use something like `printf()` to format the data in a more structured way if you'd like.

Note that different input lists may have different items on them (maybe they only sell french toast at Sunday brunch). Note also that the price of beer changed during the course of the sample data above. (Happy hour ended.) Your program should be able to handle these cases.

Your program also needs to be flexible in terms of where the input list is coming from. It must work if called in any of the following ways:

```
# ./program.pl saleslog.txt
# ./program.pl <saleslog.txt
# cat saleslog.txt | ./program.pl
# ./program.pl <----- should let user enter data on STDIN
```

In class we learned a flexible way to handle all of these cases without having to change the source code at all.

Program 4: The Car Hash (10 points)

In Program 2 above, we used hashes as a way to store the same piece of information (nicknames) for many different items (schools) – a `%nicknames` hash. In Program 3, we used them as a handy data structure to keep a count or sum of data for various items - `%count` and `%revenue` hashes. Yet another way to use hashes is as a way to keep a lot of pieces of information about a single item (sort of the opposite of what we did in Program 2).

Write a program which keeps track of information about a car. A car has many pieces of information associated with it; some examples are the make, the model, the number of doors, the color and the miles per gallon it gets. Write a program that will prompt the user for each of these values and build a `%car` hash.

After gathering the data, print out the contents of the car hash to the screen. Instead of looping over the hash with a for loop (which you are comfortable with now, since it was required in Program 4), I will show you a way to quickly print the contents of the hash in a way that is easily

readable in a pinch.

Recall that although we might wish to do this:

```
print "%car";
```

unfortunately, nothing magical will happen – we'll just get the 5-character string “%hash” printed on the screen. But, there is a trick, and the secret lies within the [Data::Dumper](#) module.

Perl ships with a LOT of modules. By default, in a Perl program, you only have access to a small subset of the functionality that Perl ships with, sometimes called the core. These are the most basic functions and features of Perl, available automatically in even the simplest program. In order to access some of the advanced functionality, you need to load a module. While we will talk about modules in depth later in the semester, for now I will just show you how to use the [Data::Dumper](#) module.

```
#!/usr/bin/perl
use strict;
use Data::Dumper;
my %orange = (shape => "round", color => "orange",
              foodtype => "fruit", taste => "sweet");
print Dumper(\%orange);
```

Type in this program and see what happens.

Magic bit #1: Where did the Dumper() function come from? You get it for free when you say `use Data::Dumper`. You should be thinking “But isn't it bad form for someone else's module to pollute my namespace with a function like that? What if I had my own function named Dumper()?!” Yep. It's probably bad form. But [Data::Dumper](#) is so darn useful that most people like that it does this anyhow. We'll debate that more later this semester.

Magic bit #2: What's that `\` doing in front of the `%orange`? It makes Dumper() work better. That's all for now. :-)

Now that you see the usefulness of [Data::Dumper](#), get in the habit of using it when you are working with hashes. It's a heck of a lot easier than writing a for loop everywhere you want to see the contents of your hashes while debugging.

[A final note about this program – if you're getting the notion that the `%car` hash feels a little like the beginnings of an instance of a Car class might in Java, that's a good sign.]

Program 5: File Foolishness (30 points)

Write a program that, when called like this:

```
./program.pl input.txt odd.txt even.txt
```

reads in the input file `input.txt`, and writes all of the odd lines to the file `odd.txt` and all of the even lines to the file `even.txt`. That is to say, lines 1,3,5,7,etc... of the original file go to `odd.txt` and 2,4,6,8,etc... go to `even.txt`. The user will pass in the three filenames as arguments to the

program (@ARGV).

In addition, have your program write status to STDERR after every 1,000 lines processed. For example “1000 lines processed” might appear on STDERR, and then a little later, “2000 lines processed”, and so on.

Concentrate on robust coding in this program. By that I mean, you should be checking if there are any errors in opening the files for reading or writing and exiting gracefully (by gracefully I mean explaining to the user about the problem). You should also be prepared to handle a very large input file, so make sure you are writing your program in a way that is friendly to memory usage.

BONUS

Bonus 1. Flexible Subroutines (15 points, no partial credit)

[This is a written answer bonus question. If you choose to answer it, add it at the end of your answers.txt labelled “Bonus 1”.]

You are working on a subroutine that take drink orders at your coffee shop. The crucial pieces of information that you need to know are the type of drink, a flavor, and the size of the drink. Your shop's signature drink is the Giganto Cinnamon Mocha, which is what you recommend to every customer who doesn't already know what they want.

The problem is that you can never remember what order the information needs to be fed into the subroutine, and also, it's tedious to keep passing in Giganto Cinnamon Mochas to the subroutine so often, since that's about 75% of all the drinks you sell. It'd be better if you could just tell the subroutine, “make the usual”.

Can you write this subroutine such that:

- I can pass in the parameters in any order
- If I omit one or more of the drink type / flavor / size parameters, it chooses a default, so if I omit all of those, it should make a Giganto Cinnamon Mocha, and if I only provide a size of “Teeny”, it should make a Teeny Cinnamon Mocha, and so on.