

Perl Midterm Exam issues

#6 & #23 – for vs. foreach. The keywords for and foreach are 100% identical. The ONLY difference between the two is that one of them has four extra characters. ANYWHERE you have the word for, you can replace it with the word foreach, and vice-versa, without changing anything else at all. The following four pieces of code are all valid:

```
for ($i=0; $i<4; $i++){           foreach ($i=0; $i<4; $i++){
    print "$i\n";                   print "$i\n";
}                                     }

foreach $thing (@things){         for $thing (@things){
    print "$thing\n";               print "$thing\n";
}                                     }
```

#7 – true & false values - There are exactly four false values in Perl: the number 0, the string '0' (or "0"), the empty string '' (or ""), and the undefined value undef. Anything else at all, including a string consisting of 2 or more zeros only, is a true value.

#9 – arrays in strings - Regardless as to whether or not a combination of letters has been used as an array, Perl will give you an ERROR if you try to use @ in a string without backslashing it. You can't do it. The only exception is when you are actually interpolating an existing array.

#11 – well, first off, many people seem to have trouble with the concept of "one word". Going through that code line by line: First, take whatever is in \$_, split it on the null character, and store the resulting array in @letters. So if, \$_ contained "hello world\n", @letters now contains ('h', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd', '\n'). Second line – Assign \$last to be the member of @letters positioned at the last index of @letters – which means the last member of @letters. In our example, this is the newline character. The final line – first, compare \$last with the newline character; if they are equal, then execute chop;. Remember that if chop is not given any arguments, it defaults to working on \$_. So the last line will remove the final character from our original string (contained in \$_), if and only if the last character of that string is a newline. This is precisely what the function chomp does. So the answer to this question (asking for a one-word piece of code) is chomp;

#14 -- || vs or – first, the issue is not which of these two operators has a higher precedence. It also has nothing at all to do with string vs numerical operations. The issue is that || has a higher precedence over =, while or has a lower precedence than =. In the first piece of code, the || is done first. Perl evaluates the first argument to the || statement. If it's true, Perl returns that value. If not, it evaluates the next one. If that's true, it returns it. If not it evaluates the last argument. So \$a ends up with the first true value of \$b, \$c, or \$d. In the second piece of code, \$a = \$b is done first. This statement will take place no matter what any of the truth values are. Then, if this statement evaluates to false (which would mean that \$b evaluated to false, since = returns the value of it's right-hand argument), \$c is evaluated. If \$c also is false, then \$d is evaluated. To make these two pieces identical, one could use parentheses:

```
$a = $b || $c || $d;           $a = $b or $c or $d;
$a = ($b or $c or $d);         ($a = $b) || $c || $d;
```

#21 – split arguments - the first argument to split is a regular expression. the regular expression can be delimited by any non-alpha-numeric character, just as it can in a pattern match. Further, the same dirty dozen rules apply – only those twelve, plus the delimiters, need be backslashed. Anything else will match itself.

#21 – sort order – by default, `sort` sorts ASCIIbetically, low to high. So 100 comes before 99. (in fact, 1999999 comes before 2). So reversing the order of the sort would not help, as now, 99 would come before 88. One must write one's own numerical sort function. The easiest of which is `{ $a <=> $b }`

#21 – chomping – While it's true that the list wasn't chomped, and therefore the number that was entered last will be printed with a newline character, this does not at all affect the sort order, and is merely a cosmetic difference in output.

#24 – for/foreach aliases – The variable assigned in a foreach-style loop is an ALIAS. This means that whatever happens to that variable does indeed affect the entire loop.

#25 – calling functions in context – `reverse` does not automatically return an array. A function can have different return values based on the context in which it is called. `reverse` is one such function. When called in list context, it does indeed return an array. However, when it is called in scalar context, as with this example, it instead returns a string consisting of all the members of the array passed to it, concatenated and *then* reversed.

#27 – looking for numbers – many people tried to simply compare the input numerically 0. This does not work. the numerical operators `==`, `>`, `>=`, `<`, and `<=` will all convert their operands to numbers before evaluating them. So saying `$input >= 0` will always return true if `$input` contains a string, since strings evaluated in numeric context are 0. Similarly, trying to take the input, and modulus it by the number 1 does absolutely nothing. `%` is also a numerical operator, and will also convert its operands to numbers. There are only two ways of determining whether the input is an integer. The best is using pattern matching – in which case you must insure that ONLY digits are present, using `^` and `$`, not just anywhere in the string. The other method is to convert the input to an integer explicitly using `int`, and then comparing that to the original input. However, in this case, you must be sure to compare it via the string operator `eq`, otherwise, the input will be converted to a number, and your comparison will be vacuously true every time.

#29 – case insensitivity – anytime you see the words “case insensitive”, you should immediately think “Regular Expression”. Testing if two strings are equal via `eq` will not give you case insensitivity at all. One way around this is to convert your input to all lowercase (for example: `$input = "\L$input\E";`) and then checking against the keyword, but this will not work in all cases. (For instance, you wish to only add words to an array or hash if they haven't already been added before, regardless of the case they were in in the two instances).

If you want to compare two strings using case insensitivity, you almost always must use Regular Expressions. Specifically (for this example), your expression should look like this: `$input =~ /^done$/i;` Note that both the `^` and the `$` are necessary. Without them, your Regular Expression would match on words like “doners” or “redone”.

#29 – hashes – I must once again attempt to convince you of the benefits of hash variables in Perl. Any time you feel yourself about to be using 2 or 3 arrays, or looping 2 or 3 times through the same array to count identical members, chances are you want to be using a hash table. The amount of code and time it can save you are really something special, if you take the time to learn them. Further more, the “looping multiple times” routine in this example would cause some problems the way it was written on many exams, in that each word would be printed out the number of times it was entered, each time reporting that it was entered one time less than the last time it was entered. (I do not believe I subtracted points for this, however)