Test 2

1. Question 1
   1. The readability is easier to follow when using enums compared to int variables. The objects provide more information to the programmer than when using int variables that could be of random descriptions. The enums will be more reliable because the information for them is in one spot and cannot be changed anywhere else in the code, whereas integer variables may be changed throughout the code into an unexpected value.
2. Question 2
   1. The difference between Perl and in class for dynamic scoping is the fact that in class we discussed that dynamic scoping allocates space in memory for the variables defined in each function. In Perl, when the global variable is “hidden” by being declared in a function, space is not allocated to it, instead the value of the global variable is temporary changed to the new value until the program exits the function’s scope, then it is reverted back to the original value.
3. Question 3
   1. The function that declared an array using heap-dynamic protocol took the longest amount of time followed by the statically declared array, and lastly the regularly declared stack-dynamic array took the least amount of time.
   2. This cannot be done in Java because Java only creates heap-dynamic objects. Whenever an array is made the array is created on the heap, Java doesn’t allow the creation of stack-dynamic objects.
4. Question 4
5. Question 5
   1. Sum1 has a value of 46, and sum2 has a value of 48. The results are different for the two because the function fun is taking a pointer as a parameter and the user is passing in a memory address into the function. Since a memory address was passed into the function and it’s being dereferenced in the function with the pointer, the value at the specified memory address is being changed. This means that the value will be changed even after the function has ended and thus is why the values are different. Sum1 calls the function fun after it has already done calculations to i’s previous value of 10, whereas sum2 is after j’s value has already been changed to 14.
   2. If there were no precedence rules that means we read the expression through C’s associativity rules. Since C reads from left to right that means the expressions will still result in the same value for sum1. Sum2 will have a different value because it will be 41 from fun(&j) + 14 (value of j) / 2 = 55 / 2 = 27.
6. Question 6

|  |  |  |
| --- | --- | --- |
|  | Dynamic | Static |
| Main calls sub1; sub1 calls sub2; sub2 calls sub3 | a, x, w (sub3)  b, z (sub2)  y (sub1) | a, x, w (sub3)  y, z (main) |
| Main calls sub1; sub1 calls sub3; | a, x, w (sub3)  y, z (sub1) | a, x, w (sub3)  y, z (main) |
| Main calls sub2; sub2 calls sub3; sub3 calls sub1 | a, y, z (sub1)  x, w (sub3)  b (sub2) | a, y, z (sub1)  x (main) |
| Main calls sub3; sub3 calls sub1; | a, y, z (sub1)  x, w (sub3) | a, y, z (sub1)  x (main) |
| Main calls sub1; sub1 calls sub3; sub3 calls sub2; | a, b, z (sub2)  x, w (sub3)  y (sub1) | a, b, z (sub2)  x, y (main) |
| Main calls sub3; sub3 calls sub2; sub2 calls sub1 | a, y, z (sub1)  b (sub2)  x, w (sub3) | a, y, z (sub1)  x (main) |

1. Question 7
   1. A > B > C in mathematics is similar to saying a > b && b > c or A is greater than b greater than C. The mathematical logic says that A is greater than B and B is greater than C, but in C based language that is not the same. In C the expression evaluates to A > B equates to either 0 (false) or 1 (true) and then the result of A > B is evaluated with C. Therefore, the expression is A > B = (0 or 1) and then either 0 > C or 1 > C depending on the result of the first expression. An example of this would be if A = 5, B = 4, C = 3, then the expression would be 5 > 4 = 1 and then 1 > 3 which would equal 0 since it’s false.
2. Question 8
   1. ((a \* b)1 – (1 + c)2)3
      1. Rewritten: c - 1 - b \* a
   2. ((((++a)2 \* (b – 1)1 )3 / c)4 % d)5
      1. Rewritten: d % c / (++a) \* b – 1
      2. This expression cannot be rewritten without parenthesis. There is no way to pre-increment a before multiplying it by the result of b-1. Also, if there is no parenthesis to show that it’s a pre-increment to a then it may appear as an error.
   3. ((a – b)1 / (c & ((((d \* e)3 / a)4 – 3)2)5)6)7
      1. Rewritten: (c & 3 - a / d \* e) / a – b
      2. This expression cannot be rewritten without parenthesis. Since the last operation from reading right to left would be to divide the results up to the point by (a – b), it’s not possible to delete by an expression. There is no way to subtract a by b and still be able to create the same result on the right-hand side of the division sign unless parenthesis is used for a – b.
   4. ((-a)1 or ((c = d)2 and e)3)4
      1. Rewritten: (e and d = c) or (-a)
      2. The expression cannot be rewritten without parenthesis. The unary operator – needs to be grouped with a through the parenthesis, if it isn’t then the unary operator may be seen as a subtraction sign resulting in an error. Even if the unary operator isn’t seen as a subtraction sign, the expression would still be read as “a or e and c = d”, the unary operator would not be taken into account until after the expression has been evaluated.
   5. (((a > b)1 xor c)3 or (d <= 17)2)4
      1. Rewritten: (17 >= d) or c xor b < a
      2. The expression cannot be rewritten without parenthesis. The expression is fine up until the point where the value of c xor b < e is being compared by “or” of 17 >= d. If the parenthesis is not there, then the expression would be c xor b < e is compared by “or” to the value of d and then the result of that expression will be checked to see if it’s less than or equal to 17.
3. <expr> -> <expr> or < term > | <expr> xor < term > | <term>

<term> -> <term> and < fact > | <fact>

<fact> -> <fact> = < list > | <fact> += < list > | <fact> \*= < list > | <fact> /= < list > | <list>

<list> -> <list> < < bool > | <list> <= < bool > | <list> >= < bool > | <list> > < bool > | <list> != < bool > | <bool>

<bool> -> <bool> - < next > | <bool> / < next > | <bool> % < next > | <bool> not < next > | <next>

<next> -> <next> + < close > | <next> \* < close > | <next> & < close > | <close>

<close> -> <close>++ | <close>-- | <last>

<last> -> -<last> | ++<last> | --<last> | <var>

<var> -> constant | int\_lit | ( <expr> )