Test 2
Programming Language Concepts
4/25/2020

Christopher Fulton
M/W 5:30 PM

ALL CODE IS LOCATED AT:

https://github.com/cfulton5/PLC-Test-2

1. **C++ file found in the GitHub link above labeled as Question1.cpp**

As far as readability goes for the programs, using enumeration vastly improves it. Using enumeration keeps the (using this example) days of the week bound within a days enumeration. This is superior to integer variables, as they are located in a listed format within the main function. For reliability, enumeration is good when working with user-defined data types, but the single-memory-location for integer variables is a superior data-reliable method.

2. **Perl file found in the GitHub link above labeled as Question2.pl**

Dynamic scoping allows for dynamic and changing variables within a program that alter something from another part of the program. Static scoping is simply applying a variable to its own environment, unchanged by the rest of the program. In this Perl code, the dynamic variable is changed from a sub program and called later. The static variable is also changed within a sub program and called later. Only the dynamic variable displayed a change during the call.

3. **C++ file found in the GitHub link above labeled as Question3.cpp**

The clock times for the C++ program show that the quickest to call is typically the Stack array, as it is located directly within the program's memory stack. This is followed by the heap array, as it has pointers directly to memory, speeding the process. Lastly, the slowest is the static array since the memory used is within the rest of the program memory.

In Java, there are no pointers, so a heap array made of pointers is impossible as the concept is missing from the language- instead using references.

4. **Java files found in the GitHub link above labeled as rda.java and front.java**

Rda.java is the primary function that would deal with precedence, as it parses through an equation to find the first and highest priority function to execute it first. Front.java would help with associativity to make sure the correct expression is executed within its parenthesis and its perameters.

5.

Sum1 = 46 : 10/2 + (3*(10+4)-1) = 46

Sum2 = 48 : (3*(10+4)-2) = 41 since j is referenced into the fun() function, j is modified by +4; 41 + (14/2) = 48

If there were no precedent rules, the outcome would remain the same since there are parenthesis separating the relevant operations. However, if there were no parenthesis, the only change without precedence would be fun(&j) + j / 2 , where fun(&j) adds j before dividing by 2.

6.

Dynamic

Static

a

x	sub3
y	sub1
z	sub2
a	sub3
b	sub2
w	sub3

x	main
y	main
z	main
a	sub1
b	sub2
w	sub3

b

x	sub3
y	main
z	main
a	sub3
b	-
w	sub3

x	main
y	main
z	main
a	sub3
b	-
w	sub3

c

x	sub3
y	sub1
z	sub1
a	sub1
b	sub2
w	sub3

x	main
y	main
z	main
a	sub2
b	sub2
w	sub3

d

x	sub3
y	sub1
z	sub1
a	sub1
b	-
w	sub3

x	main
y	main
z	main
a	sub3
b	-
w	sub3

e

x	sub3
y	sub1
z	sub2
a	sub2
b	sub2
w	sub3

x	main
y	main
z	main
a	sub1
b	sub2
w	sub3

f

x	sub3
y	sub1
z	sub1
a	sub1
b	sub2
w	sub3

x	main
y	main
z	main
a	sub3
b	sub2
w	sub3

7.
In mathematics, a > b > c would evaluate simply to a being larger than b, which is larger than c. In C, however, the “>” is a relational operator. This means that while one would expect a > b > c to evaluate as a > b && b > c , it instead evaluates as a > b first, yielding a 1 for true or 0 for false, then evaluates that 0 or 1 being > c.

8. a. i. $\{a*b\}_1-1+c$
$\{a*b\}_1-\{1+c\}_2$
$\{\{a*b\}_1-\{1+c\}_2\}_3$
ii. (c+1)-b*a
iii. The rewritten statement cannot be written without parenthesis; the c+1 section would both need to be subtracted in order to equate to the same answer. Reordering cannot avoid parenthesis.

b. i. $++a*(\{b-1\}_1)/c\%d$
$\{++a\}_2*(\{b-1\}_1)/c\%d$
$\{\{++a\}_2*(\{b-1\}_1)\}_3/c\%d$
$\{\{\{++a\}_2*(\{b-1\}_1)\}_3/c\}_4\%d$
$\{\{\{\{++a\}_2*(\{b-1\}_1)\}_3/c\}_4\%d\}_5$
ii. d%c/a++*1-b

c. i. $(\{a-b\}_1)/c\&(d*e/a-3)$
$(\{a-b\}_1)/c\&(\{d*e\}_2/a-3)$
$(\{a-b\}_1)/c\&(\{\{d*e\}_2/a\}_3-3)$
$(\{a-b\}_1)/c\&(\{\{\{d*e\}_2/a\}_3-3\}_4)$
$\{(\{a-b\}_1)/c\}_5\&(\{\{\{d*e\}_2/a\}_3-3\}_4)$
$\{\{(\{a-b\}_1)/c\}_5\&(\{\{\{d*e\}_2/a\}_3-3\}_4)\}_6$
ii. (3-a/e*d)&c/b-a
iii. The re-written equation needs parenthesis, as the binary operator “&” would otherwise be operating with an element of a sub-equation.

d. i. $\{-a\}_1$ or c = d and e
$\{-a\}_1$ or $\{c = d\}_2$ and e
$\{-a\}_1$ or $\{\{c = d\}_2$ and $e\}_3$
$\{\{-a\}_1$ or $\{\{c = d\}_2$ and $e\}_3\}_4$
ii. a- or e and d=c

e. i. $\{a>b\}_1$ xor c or d<=17
$\{a>b\}_1$ xor c or $\{d<=17\}_2$
$\{\{a>b\}_1$ xor $c\}_3$ or $\{d<=17\}_2$
$\{\{\{a>b\}_1$ xor $c\}_3$ or $\{d<=17\}_2\}_4$
ii. (d<=17) or c xor b<a
iii. The rewritten equation needs parenthesis or the “or” statement would execute with only one element of (d<=17).

9.

```
<expr> => <expr> or <orExpr>
          | <expr> xor <orExpr>
          | <orExpr>

<orExpr> => <orExpr> and <andExpr>
          | <andExpr>

<andExpr> => <andExpr> = eqExpr
          | <andExpr> += eqExpr
          | <andExpr> *= eqExpr
          | <andExpr> /= eqExpr
          |eqExpr

<eqExpr> => <eqExpr> < <ineqExpr>
          |<eqExpr> <= <ineqExpr>
          |<eqExpr> >= <ineqExpr>
          |<eqExpr> > <ineqExpr>
          |<eqExpr> != <ineqExpr>
          |<ineqExpr>
<ineqExpr> => <ineqExpr> - <math1Expr>
              |<ineqExpr> / <math1Expr>
              |<ineqExpr> % <math1Expr>
              |<ineqExpr> not <math1Expr>
              |<math1Expr>
<math1Expr> => <math1Expr> + <math2Expr>
                |<math1Expr> * <math2Expr>
                |<math1Expr> & <math2Expr>
                |<math2Expr>
<math2Expr> => -<modifier>
                |++<modifier>
                |--<modifier>
                |<modifier>++
                |<modifier>--
                |<modifier>
<modifier> => (<expr>)
              |<id>
              |<number>

<id> => <string>

<number> => <integer>
```