1. For what types of A and B is the simple assignment statement A = B legal in C++ but not Java?

When A is int and B is: bool, double, float, long;

When A is bool and B is int, char, double, short, float, long;

When A is char and B is int, bool, double, short, float, long;

When A is double and B is double;

When A is short and B is int, bool, char, double, float, long;

When A is float and B is bool, double;

When A is long and B is bool, double, float;

2.

Disadvantages: More things to pay attention to while coding. Use has to remember and map out the binding carefully. The advantage is better memory use if done correctly. The program also becomes safer. Unintended casting and mutation can be avoided.

Supposedly, we can use actual “True” and “False” to represent boolean values, instead of integer coercion. This will ensure the boolean variable does not get adjusted unnoticed.

Evaluation criteria wise: (replace all implicit type conversion with explicit type conversion):

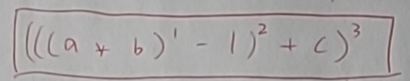
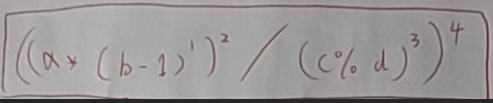
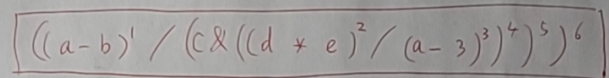
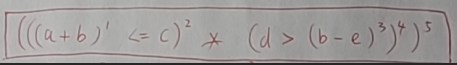
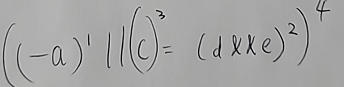
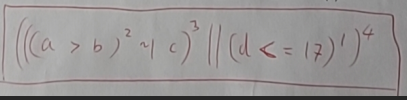
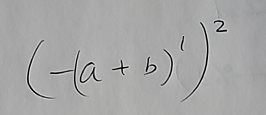
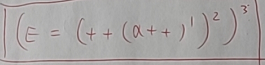
Better readability,

Worse writeability,

Better reliability,

Better cost due to ease to optimisation.

3.

1. 
2. 
3. 
4. 
5. 
6. 
7. 
8. 
9. 

4.

ANS:

(((a\*b)-1)+c)

=((51\*7)-1)+11)

=357-1+11

=367

(2)

((a\*(b-1)) / (c%d))

=((51\*(7-1)) / (13%2))

=(51\*6/1)

=306

(3)

((a-b)/(c&((d\*e)/(a-3))))

=((51-7)/(11&((13\*2)/(51-3))))

=42/(11&0.541667)

=42/0

=infinity

(4)

((a+b)<=11)\*(d>(b-e))

=((51+7)<=11)\*(13>(7-2))

=(58<=11)\*(13>5)

=False \* True

Cannot compute boolean value multiplication

(5)

((-a) || (c ) = (d&&e))

=(-51 || (c = (13&&2) ))

value 2 assigned to c

=(-51 || 2 )

=-51

(6)

(((a>b)~|c)||(d<=17))

=((51>7)~|11) || (13<=17) )

=(True ~| 11) || True

boolean value should not be in a bitwise computation

suppose True == 0

=(0 ~| 11) || 0

=11||0

=11

(7)

((-a)+b)

=-51+7

=-44

(8)

((a+(b\*c))+d)

=(51+(7\*11))+13

=(51+77)+13

=141

(9)

(e = (++(a++)))

=(e = (++(51++)))

=(e = (++(52)))

=(e = 53)

e assigned to 53

5.

CFG:

<expr> -> - <expr>

! <expr>

( <expr> )

<expr> {+|-|\*|/|&&||||~||&} <expr>

<var>

<var\_id> = <expr>

<var> -> {++|--} <var\_id>

<var\_id> {++|--}

<var\_id>

6.

1.add(minus(multi(a,b),1),c)

2.div( multi(a,minus(b,1)), mod(c,d) )

3.div(minus(a,b), div( bitwise\_and(c, multi(d,e) ) , minus(a,3) ) )

4.multi( less\_equal\_than( add(a,b),c), larger\_than(d, minus(b,e) ) )

5.logical\_or(negative(a) , c = logcial\_add(d,e) )

6.logical\_or( XOR(larger\_tahn(a,b), c) , less\_than(d,17) )

7.negative( add(a,b) )

8.add(add(a, multi(b,c)), d)

9.e = increment(increment(a))

No need to use precedence marker/symbol. Every function, except for the parent function, will always be a child to another function.

7.

Fq7\_2.py is the RDA for OOP style rewritten expression from problem 6.

see output in fq7\_output.txt

This RDA parse the following syntax: (Too much to fix. I can write RDA)

(1) (a.(multi(a, b)).sub(1)).add(c)

(2) (a.multi((b.sub(1)))).div((c.mod(d)))

(3) (a.sub(b)).div( c.bitwise\_and( (d.multi(e)).div( a.sub(3) ) ) )

(4) ((a.add(b)).smaller\_equal\_than(c)).multi( d.larger\_than( b.sub(e) ) )

(5) ((a.negative(1)).logical\_or(c)).assign( d.logical\_and(e) )

(6) ((a.larger\_than(b)).xor(c)).logical\_or( (d.smaller\_equal\_than(17)) )

(7) (a.add((b))).negative(1)

(8) (a.add((b.multi(c)))).add(d)

(9) e.assign( (a.increment(1)).increment(1) )

EBNF:

<expr> -> <sub\_expr> |

'(' <expr> ')' |

'.' <func\_para>

<sub\_expr> -> <var> <fac> |

<num>

<fac> -> ε |

'.' <func\_para>

<func\_para> -> <func\_name> '(' <expr> ')'

<func\_name> -> "add" | "sub" | "multi" | "div" | "mod" |

"smaller\_than" | "smaller\_equal\_than" | "larger\_than" | "larger\_equal\_than" |

"bitwise\_and" | "bitwise\_or" | "xor" |

"logical\_and" | "logical\_or" |

"negative" | "assign" |

"increment" | "decrement"

<var> -> 'a' | 'b' | 'c' | 'd' | 'e'

<num> \_> '0-9' <num>

Noting: .negative and .assign functions MUST take parameter to ease the rules. Not an uncommon practice, those parameters may come in handy. They will not cause confusions, such as: “.minus(0)” means negative on the variable while “.minus(anything not zero)” means no operations done; the same with .increment() and .decrement()



Code at fq8.py

See output at fq8\_output.txt

Im using OOP function call style expressions in problem6.

Rules that the expression must follow:

A.Certain function-call’s result must follow the datatype of the variable, who calls the function.

B. Certain function-call has certain type of datatype output.

C. As a result of A & B, some functions should not be in the same expression.

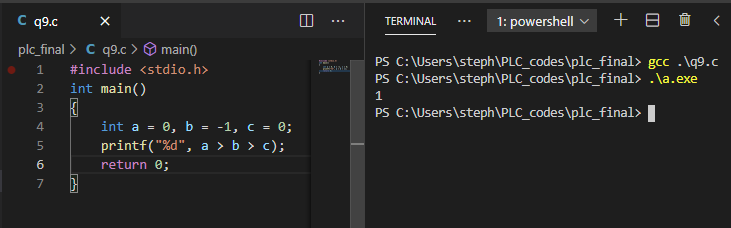
D.Some function only works on certain type of variables.

E.Only one assignment function allowed per line/ statement.(exclude increment and decrement)

F. No zero division allowed

G. Increment, decrement should only be called by unary variable

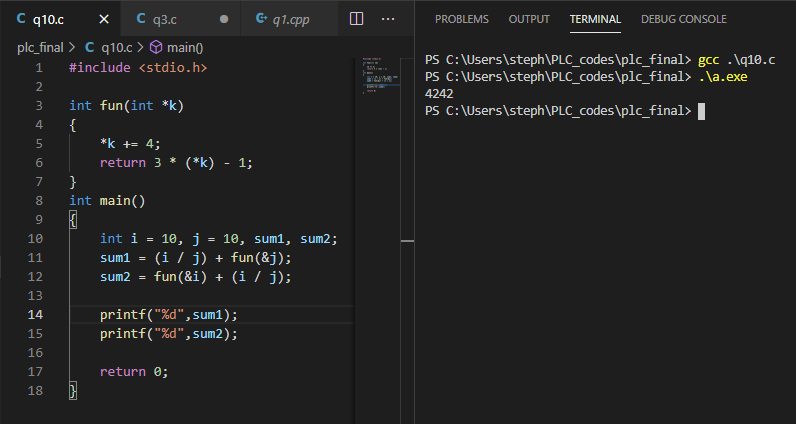
9.



in math logic, "a>b>c" means (a>b and b>c). there are two mathematical comparisons: evaluating a and b, then b and c. for C, a>b is evaluated first, producing either 0 or 1. then, this boolean result in form of integer is compared with c. Comparison between b and c is never done.

for example: int a = 0, b = -1, c = 0; gives a>b>c an result of 1, True in C, while mathematically false.

10.



the final values of sum1 and sum2 are 42 and 42.

at line: sum1 = (i / j) + fun(&j):

"i / j" is 10/10 = 1 for i and j values are 10 and 10 at this point.

fun(&j) takes the address of int j as parameter:

at fun(int \*k) function definition:

where the address of j is viewed as a pointer k;

"\*k" is the value of what the address k(j's address) points to, which denotes the value of j itself;

"\*k += 4" increments j's value by 4. j's value is now 14.

Then j values is multiplied by 3 and minus 1, which is: 14 \* 3 -1 = 41, and returned to main.

sum1 = 10/10 + 41 = 42.

at line: sum2 = fun(&i) + (i / j);

fun(&j) takes the address of int i as parameter:

at fun(int \*k) function definition:

where the address of i is viewed as a pointer k;

"\*k" is the value of what the address k(i's address) points to, which denotes the value of i itself;

"\*k += 4" increments i's value by 4. i's value is now 14.

Then i values is multiplied by 3 and minus 1, which is: 14 \* 3 -1 = 41, and returned to main.

"i / j" is 14/14 = 1 for i and j values are 14 and 14 at this point.

sum2 = 41 + 14/14 = 42.