

Comments Regarding Quiz 1

Contents

1	First: A Note about the Quiz	2
2	Operators and Precedence	3
3	Operators as Functions	6
4	Code Comprehension: Unraveling Loops	7

1 First: A Note about the Quiz

- Cumulatively, the quizzes are worth 5% of your final grade
 - With 5 quizzes, each is worth 1% of your final grade
 - So don't worry *too much* about your grade if you didn't do so well
-
- That said, the quizzes are meant to test the reading and the C++ language
 - They will never contain any higher-level concepts, such as good software design
 - (*Those topics are reserved for the homeworks and exams!*)
-
- As I've mentioned many times, the *focus* of this course is learning good problem solving abilities and programming methodology
 - Learning the C++ language is also a goal, but it's relatively minor
 - (*With more work in CS2, Data Structures, and Algorithms, you'll quickly become very proficient with C++*)
 - However, there are some key parts of C++ you *must learn!*

2 Operators and Precedence

- This is one of those things you *must learn*
- Learning which operators are evaluated first is mandatory in order to use them effectively
- The order in which operators are evaluated may seem arbitrary, but the order is actually based on what's commonly done in mathematics

- I'm not going to cover the order in detail, but simply tell you that *you must learn these!*
- There is an appendix in the back of your book that sums up the operators and their precedence very nicely
- The appendix, however, contains a listing of *all* C++ operators, so here's a table with what you need to know...

Operator	What it Represents	Associativity
++	Pre/Post-increment operator	
--	Pre/Post-decrement operator	
*	Multiplication	left to right
/	Division	left to right
%	Modulus	left to right
+	Addition	left to right
-	Subtraction	left to right
<	Less-than	left to right
<=	Less-than-or-equal-to	left to right
>	Greater-than	left to right
>=	Greater-than-or-equal-to	left to right
==	Equal-to	left to right
!=	Not-equal-to	left to right
&&	Logical AND	left to right
	Logical OR	left to right
=	Assignment	right to left
+=	Addition assignment	right to left
-=	Subtraction assignment	right to left
*=	Multiplication assignment	right to left
/=	Division assignment	right to left
%=	Modulus assignment	right to left

- In the above table, operators higher in the table appear have higher precedence than those lower in the table
- *However*, those operators above in the same box (such as + and -) have *equal* precedence
- In any expression, those operators with higher precedence are evaluated first
- If operators with equal precedence appear in the expression, they will be evaluated in the order they appear
- The associativity tells us how expressions such as

$$1 + 2 + 3 + 4 + 5$$

will be evaluated. Since + is *left to right* associative, the above expression will be evaluated as...

$$((((1 + 2) + 3) + 4) + 5)$$

If + were *right to left* associative, it would be evaluated as...

$$(1 + (2 + (3 + (4 + 5))))$$

- **Example** Consider the following expression...

$$n = 1 + 2 * 3;$$

since * has higher precedence than +, $2 * 3$ is evaluated *first*. After that, the addition is performed on *the result of* $2 * 3$. If we wanted to add parentheses to make this explicit, the expression would look like the following...

$$n = (1 + (2 * 3));$$

- **Example** Consider the following expression. . .

$$n = 1 + 2 * 3 / m++ * 5 + 6;$$

where m is another variable. Using the precedence chart above, the expression would be evaluated in the following order. At each step, I display in **red** the operator that is being evaluated.

Order	Expression
1	$m++$
2	$2 * 3$
3	$(2 * 3) / m++$
4	$((2 * 3) / m++) * 5$
5	$1 + (((2 * 3) / m++) * 5)$
6	$(1 + (((2 * 3) / m++) * 5)) + 6$

- If this seems confusing, don't worry. . . it is
- As a good rule of thumb when programming. . .

For lengthy/complicated expressions, use parentheses to make the order of operations explicit!

3 Operators as Functions

- **First** Make sure you understand the precedence and associativity of operators above
- **Next** Make sure you understand that these operators are simply *function*, that return values
- Arithmetic operators, such as +, *, etc. return numeric values
- Relational operators such as ==, <, >=, etc. return *boolean* values
- Taking all of this into account, an expression such as...

`x >= y >= z`

will be interpreted as follows

1. According to the associativity of >= in C++, the expression will be evaluated as

`((x >= y) >= z)`

2. Suppose `x = 0`, `y = 10`, `z = -10`. Then the above expression is equivalent to

`((0 >= 10) >= -10)`

3. Clearly, $0 \geq 10$ is *false*. Since >= is a *function* that returns *boolean* values, the first sub-expression to be evaluated (`0 >= 10`) will return **false**

4. Thus, the expression becomes

`(false >= -10)`

5. Remember *automatic casting*? Here, C++ notices that you're comparing a boolean value (**false**) with an integer (-10).

6. C++ can automatically cast a boolean value to an integer (**false** will be cast as 0), and since `int` is more expressive than `bool`, *there will be no warning generated during compile!*

7. As such, the above expression is equivalent to

`(0 >= -10)`

8. Which is *true!*

9. However for the numbers we provided, this is probably not what we expected or wanted

4 Code Comprehension: Unraveling Loops

- Concerning the **code comprehension** question, many people had problems
- And it's no wonder... variables names like **n** and **m** make things difficult to understand/follow!
- This is why we covered *good C++ programming style*, which specifically included a *naming convention* for variables
- Had I completely employed such a convention, the code would have looked something like the following...

```
// This integer is retrieved from the user. Afterwards, we calculate
// the factorial of this number.
int inputNumber;

// This variable is where we will calculate and store the factorial
// of n (n!). Note that we initialize it to 1... if we initialized
// it to 0 (zero), then all multiplications would result in 0.
int factorial = 1;

cout << "Enter a positive integer: ";
cin >> inputNumber;

// Calculate the product of all integers from 1 to inputNumber.
// Note that for each integer (count) from 1 to inputNumber, we
// simply multiply factorial by the integer (count)
for (int count = 1; count <= inputNumber; count++)
{
    factorial *= count;
}

// Output the factorial
cout << factorial << endl;
```

- So if we only write such code, why worry about comprehending ugly code?
- **Answer** Because you'll frequently have to look at such code and figure it out

- Most people seemed to have difficulty understanding what was going on during the loop
- Few could figure out what the output was for inputs 3 and 4
- Understanding loops can be difficult to do, so I'm going to cover one method of unraveling a loop and trying to figure out what the loop does
- With this method, we simply build a table with each row representing one pass through the loop
- We keep track of all the variables used in the loop, and if any variables change, we update those with each row (each pass through the loop)
- So, for the loop in the following code...

```
int n;
int m = 1;

cin >> n;

for (int count = 1; count <= n; count++)
{
    m *= count;
}
```

- We first notice that `m = 1` and `n` is input from the user
- Next, we unravel the loop. Each row represents the value of each variable *after the iteration is executed*

Iteration	count	m	n
1	1	(1)*1	n
2	2	(1 * 1)*2	n
3	3	(1 * 1 * 2)*3	n
4	4	(1 * 1 * 2 * 3)*4	n
⋮	⋮	⋮	⋮
n	n	(1 * 1 * 2 * 3 * 4 * ...)*n	n

- Note the following
 - `n` doesn't change through the loop (this can easily be seen since `n` isn't updated/changed throughout the loop)
 - `count` begins at 1 and is incremented after each execution. Thus, `count` is essentially *counting through the positive integers starting at 1*
 - With each execution, `m *= count` (remember, this is equivalent to `m = m * count`), is executed. To show how this "builds up", I've represented `m` as the list of multiplications
 - For `m`, the expression in parentheses represents the previous value of `m` (in the row above)
 - The red multiplications represents the multiplication performed in that specific iteration
- Notice that the goal of unraveling the loop like we did above is to eventually figure out what the row for the final iteration is
- In this case, the final iteration is the n^{th} iteration
- Once we can represent the values for all other variables after the final execution, we can understand what the result of the loop is