# CS370 – ASSIGNMENT #5

## NAME: Kostiantyn Makrasnov

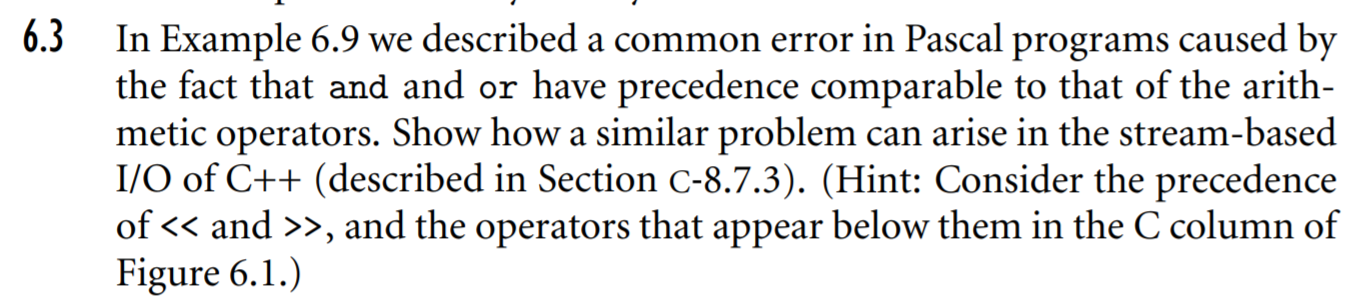
**GRADE:**

|  |  |  |
| --- | --- | --- |
| **CATEGORY** | **POINTS** |  |
| EX05\_01 |  | 20 |
| EX05\_02 |  | 10 |
| EX05\_03 |  | 10 |
| EX05\_04 |  | 15 |
| EX05\_05 |  | 30 |
| EX05\_06 |  | 15 |
|  |  |  |
| **TOTAL** |  | 100 |

## EXERCISES:

General References used throughout:

* Textbook

**EX05\_01 –** Complete Exercise 6.3, page 286

If one was to evaluate a condition and print out a result with C++ stream IO, because the stream << operator takes precedence over comparison operators (!=, ==) the following piece of code would not compile even though if the precedence of equality tests was higher the expression would output a 1 (true).



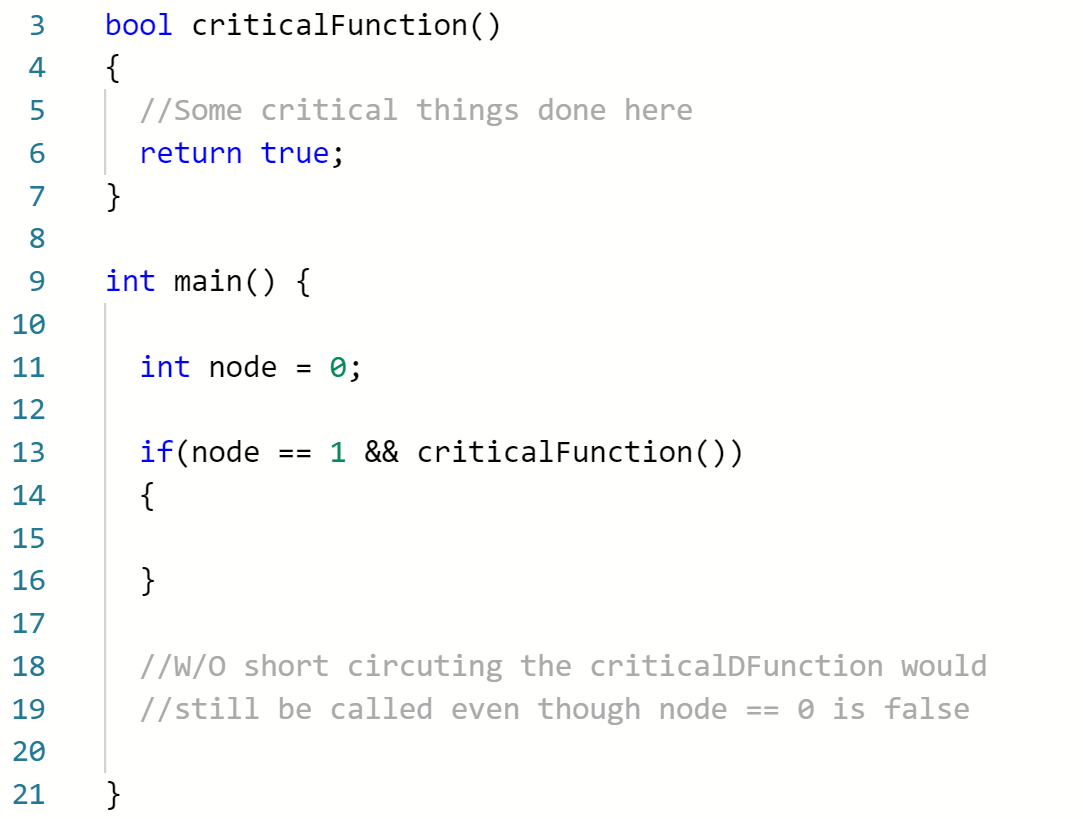
We can check this by forcing the condition to be evaluated first with parenthesis and indeed this statement prints 1.



**EX05\_02 –** Complete Exercise 6.7, page 287

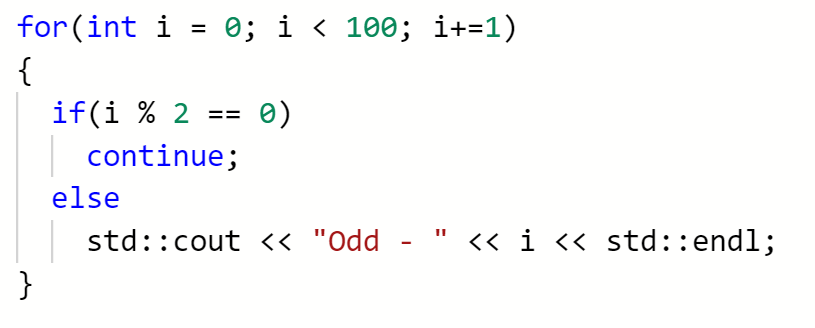
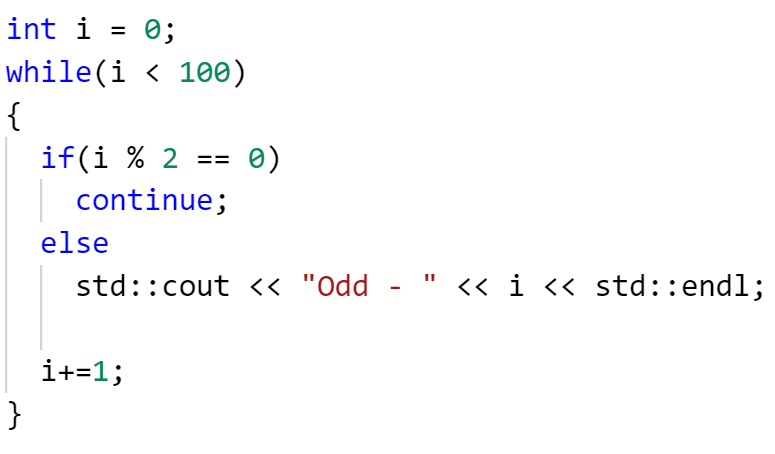
Because ***&i*** returns the address of the variable ***i***, finding the address of an address with ***&(&i)*** makes little sense and is invalid no matter what type of variable ***i*** is because the addresses themselves are constants and therefore do not have any address of their own.

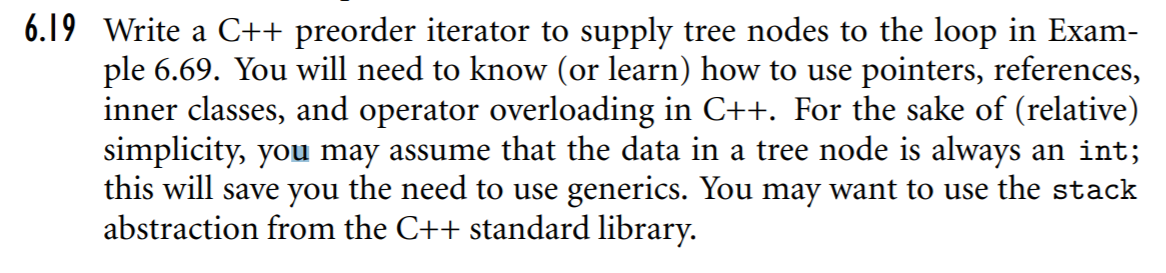
**EX05\_03 –** Complete Exercise 6.12, page 288: Describe a plausible scenario in which a programmer might wish to avoid short-circuit evaluation of a Boolean expression.

The only situation I can think of where short circuit evaluation would be undesirable is when an assignment or some other operation that follows a condition needs to be executed for a proper output of the rest of the program. With short circuiting the first condition may prevent the execution of the rest of the statements that follow in some cases causing critical procedures to never execute. In the below example criticalFunction() will never get called because in C++ the first condition will short circuit the rest of the expression inside the if statement.

**EX05\_04 –** Complete Exercise 6.16, page 288: The equivalence of for and while loops, as mentioned in Example 6.64, is not precise. Give an example in which it breaks down. Hint: think about the continue statement,

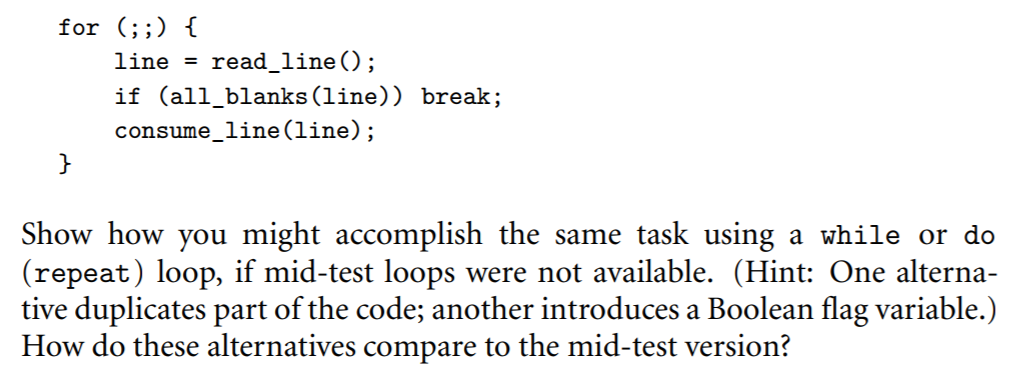
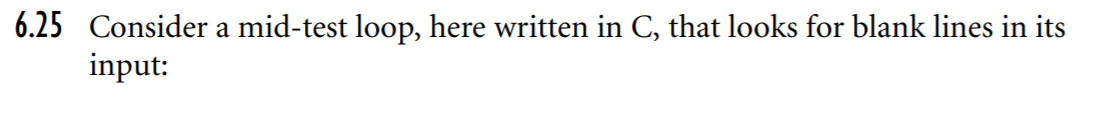
In the case when a continue statement is present within the for loop and is not itself the last statement inside the combination loop the equivalence of **for** and **while** loops breaks down. This is because the iterator statement for the while loop is placed at the very end of the while loop, a continue statement anywhere in the body of the while loop would skip the iterator making the while loop to execute a larger number of times than the for loop. An example is given below, the two loops are not equivalent since the while loop results in an infinite execution.



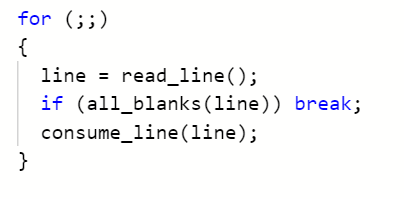
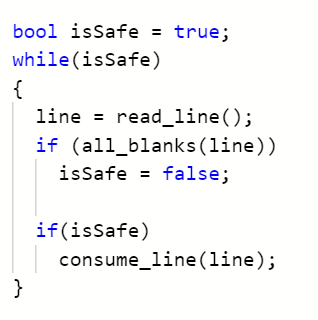
**EX05\_05 –** Complete Exercise 6.19, page 288

Code Provided with document in directory 🡪 .\EX05\_05\...

**EX05\_06 –** Complete Exercise 6.25, page 289-290



**Original**



The alternative seems to be much less readable compared to the mid test version of the code because the blank line condition is checked multiple times. It is also taking a longer amount of lines to do the same work. We also have to spend additional memory and time to store and assign the flag variable that is used in the non-mid test version to indicate when the while loop has to exit.

**Alternative**