## Lecture Review and Lab 2 Assignment

## Magic Numbers - Unnamed numerical constants

The term **magic number** or **magic constant** also refers to the programming practice of using numbers directly in source code. This has been referred to as breaking one of the oldest rules of programming, dating back to the early day of programming in the 1950’s. The use of unnamed magic numbers in code obscures the developers' intent in choosing that number, increases opportunities for subtle errors (e.g. is every digit correct in 3.1415926535897 and is this equal to 3.1415?) and makes it more difficult for the program to be adapted and extended in the future. Replacing all significant magic numbers with named constants makes programs easier to read, understand and maintain.

**Exercise 1**: Write two versions of a 10 line or less program, the first program that uses magic numbers in the condition of an “if” statement, the second program that replaces it with Named Numeric Constants.

## Boolean Conditions – Easy to understand

A Boolean variable contain only two possible values, True and False. In conditions, there is no further need to evaluate a Boolean variable, it already contains a True or False value. The computer does **not** further process the Boolean variable, thus use of Booleans speeds up execution of programs. Use of Boolean variables in a programming language is a good practice making the program easier to read, understand and maintain.

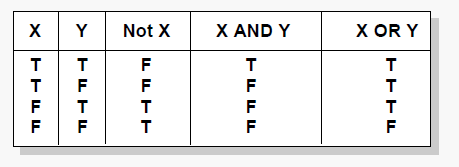
**Exercise 2**: Write two versions of a 15 line or less program, a first program that uses non Boolean relational conditions in an “While” loop statement, the second program that replaces relational term with a single Boolean variable.

**Equivalent Conditions -**

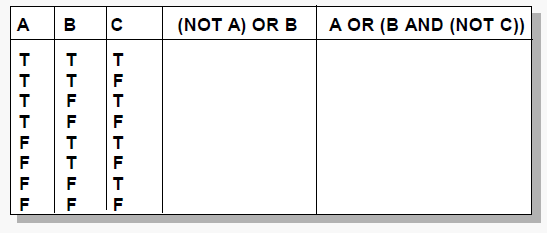
Boolean Truth Tables for &&(and) and ||(or):

## 

A Boolean truth table including the Not logical operation:



**Exercise 3**: Complete the right two columns in the table below.



**Exercise 4**: Write two versions of a 15 line or less program, one that uses one version of a condition conditions in an “if else” statement, the second that replaces it with equivalent but shorter condition.

## Short Circuiting – Minimal Evaluation of Conditions

Short Circuiting the condition in an ‘if’ statement is when it is not necessary to process each term in a compound condition. The expressions in a condition are evaluated normally left to right.

Starting with the first expression, the computer will evaluate each expression in a condition. When it encounters a final expression that is sufficient to determine an outcome for the condition, it will STOP evaluating any more expressions in the condition. This minimal evaluation save time and helps programs run faster.

Despite these speed benefits, minimal evaluation may cause problems for programmers who do not realize (or forget) it is happening. For example, in the code

if ( expA && afunc(b)) { doSomething(); }

if afunc(b) is supposed to perform some ‘*required’* operation regardless of whether doSomething() is executed, such as allocating system resources, and expA evaluates as false, then afunc(b) will not execute, which could cause problems.

**Exercise 4**: Write a 25 line or less program, has a compound conditions in an “if” statement. Use 4 expressions in the conditions. Place the expressions in the best order of execution. Explain why they are in that order.

Note: Expressions in a compound condition should be arranged in order of expected frequency

## Dangling Statements – Defaults bread errors

Sometimes it is unclear which statements belong to an “if” statement. Use of {} removes any doubts.

**Exercise 5**: Write two versions of a 10 line or less program, one that uses non Boolean conditions in an “if else” statement, the second that replaces it with Booleans

**Exercise 6**: What is de Morgan’s Law. Give an example when you would use it. What other classes might you take to help you with conditions in Decision structures and Repetition Structures?