Assignment 2
C/C++ Programming I

C1A2 General Information

Limiting the "Scope" of Variables

The scope of an identifier (a name) is defined as the portion of code over which it is accessible. The scope of a variable declared inside a block extends from that declaration to the end of that block, where a "block" is defined as a "curly-brace enclosed sequence of 0 or more statements". Good programming practice dictates that the scopes of non-const variables be as small as possible to prevent their values from being changed by code that should not change them. However, since the values of const variables cannot be changed, if being used in place of macros they should be defined in the same place the macros would have been defined. Otherwise they should be defined as the first things in the function that uses them. Consider the following three examples:

```
1       ...Any C or C++ Function...
2       {
3           int x, y, z;
4           for (x = 0; x < VAL1; ++x)
5           {
6               for (y = 0; y < VAL2; ++y)
7               {
8                   if (x + y > VAL3)
9                   {
10                      z = x - y;
11                  }
12              }
13          }
14      }
15
16      ...Any C or C++ Function...
17      {
18          int x;
19          for (x = 0; x < VAL1; ++x)
20          {
21              int y;
22              for (y = 0; y < VAL2; ++y)
23              {
24                  if (x + y > VAL3)
25                  {
26                      int z = x – y;
27                  }
28              }
29          }
30      }
31
32      ...Any C++ or >=C99 C Function...
33      {
34          for (int x = 0; x < VAL1; ++x)
35          {
36              for (int y = 0; y < VAL2; ++y)
37              {
38                  if (x + y > VAL3)
39                  {
40                      int z = x – y;
41                  }
42              }
43          }
44      }
```

Poor Declaration Placement (C or C++)

All variables are declared on line 3, which is inside the block that starts on line 2 and ends on line 14. Thus, their scope extends from line 3 to line 14 and they are all accessible within that region. Note, however, that variable **y** is only needed from line 6 through line 10 and variable **z** is only needed on line 10 and, thus, their scopes are both wider than necessary. (Also note that loop count variables should always be initialized in a "for" statement's "initial expression", not the original declarations.)

Better Declaration Placement (C or C++)

Variable **x** is declared as in the previous example because it is needed from line 19 through line 26. Its scope extends from line 18 to line 30. However, since variable **y** is only needed from line 22 through line 26 it is declared on line 21, which is inside the block that begins on line 20 and ends on line 29. Thus, its scope only extends from line 21 to line 29. Finally, since variable **z** is only needed on line 26 it is declared there, which is inside the block that begins on line 25 and ends on line 27. Its scope only extends from line 26 to line 27.

Best Declaration Placement (All C++ and C >= C99)

Although variables that are not being used as "for" loop counters should be declared as in the previous example, those that are being used for that purpose should be declared and initialized as shown in this example. This further limits their scope to only within the "for" statement itself. That is, the scope of variable **x** is now from line 34 to line 43 and the scope of variable **y** is now from line 36 to line 42.

Get a Consolidated Assignment 2 Report (optional)

If you would like to receive a consolidated report containing the results of the most recent version of each exercise submitted for this assignment, send an empty email to the assignment checker with the subject line **C1A2_*ID***, where ***ID*** is your 9-character UCSD student ID. Inspect the report carefully since it is what I will be grading. You may resubmit exercises and report requests as many times as you wish before the assignment deadline.

C1A2E0 *(6 points total - 1 point per question – No program required)*

Assume language standards compliance and any necessary support code unless stated otherwise. Testing erroneous or implementation dependent code by running it can be misleading. These are not trick questions and each has only one correct answer. Major applicable course book notes are listed.

1. The data type of the literal *252767* is:
   (Notes 2.1 & 2.2)
   A. implementation dependent
   B. **long**
   C. **int**
   D. not defined in standard C
   E. none of the above

2. The data types of:
   a) unsuffixed floating literals and
   b) unsuffixed integer literals, are:
   (Notes 2.2 and 2.4)
   A. implementation dependent in both cases.
   B. determined by the value of the literals in both cases.
   C. a) **double**
   b) determined by the number of digits
   D. a) **double**
   b) determined by the value and base of the literal
   E. none of the above

3. A mathematical operation where all operands are type **char**:
   (Note 2.10)
   A. is illegal - type **char** is only used for character operations.
   B. is evaluated using type **signed char** arithmetic.
   C. is evaluated using type **char** arithmetic.
   D. is evaluated using type **int** or **unsigned int** arithmetic.
   E. cannot be written portably.

4. Predict the values of *5 / -2* and *5 / -2.0* on pre-C99 and pre-C++11 compilers.
   (Note 2.8)
   A. two possibilities:
   -2 and *-2.5* **or** *-3* and *-2.5*
   B. two possibilities:
   -2 and *-2.5* **or** *-1* and *-2.5*
   C. *-2* and *-2.500000*
   D. A general prediction cannot be made for all implementations
   E. None – a compiler or run time error occurs

5. If **char** is 8 bits and **int** is 16 bits, predict the values of *-7 % 3* and ***sizeof**(-5 % 3)* on pre-C99 and pre-C++11 compilers.
   (Notes 2.8 & 2.12)
   A. two possibilities: *-1* and *2* **or** *2* and *2*
   B. two possibilities: *-1* and *2* **or** *1* and *2*
   C. three possibilities:
   -1 and *2* **or** *2* and *2* **or** *-2.3* and *2*
   D. *-1* and *2* **or** *-1* and *4* depending upon the data type of **sizeof**
   E. Such a prediction cannot be made for all implementations.

6. Which of the following guarantees the correct answer on any machine?
   (Notes 2.10 & 2.11)
   A. **long** value = 300 * 400 * 10**L**;
   B. **long** value = 300 * 400**L** * 10;
   C. **float** value = 300 * 400 * 10;
   D. **double** value = (**double**)(300 * 400 * 10L);
   E. none of the above because the value of each is implementation dependent

Submitting your solution

Using the format below place your answers in a plain text file named **C1A2E0_Quiz.txt** and send it to the Assignment Checker with the subject line **C1A2E0_*ID***, where ***ID*** is your 9-character UCSD student ID.

-- Place an appropriate "Title Block" here --
1. A
2. C
etc.

See the course document titled "Preparing and Submitting Your Assignments" for additional exercise formatting, submission, and Assignment Checker requirements.

C1A2E1 *(5 points – C++ Program)*

Exclude any existing source code files that may already be in your IDE project and add a new one, naming it **C1A2E1_main.cpp**. Write a program in that file to convert a lowercase character to uppercase.

Your program must:

1. prompt the user to enter any character;
2. use `cin.get` to read the character;
3. assume the input character is lowercase (even if it isn't) and attempt to convert it to uppercase;
4. display the results in the following format with single quotes around the original and converted characters, respectively, followed by the underlying decimal value of the converted character. For example, if the user inputs the character **a** the output must be exactly as shown below:
   ```
   The uppercase equivalent of 'a' is 'A' (decimal value = 65)
   ```
5. not test anything; simply apply the same conversion algorithm to any input character without regard for whether it was actually lowercase and display the results;
6. not use `toupper` or any other function to do the conversion (although `toupper` is the best solution in "real life");
7. not name any variable `uppercase` (to avoid standard library conflicts & a bogus assignment checker warning).

Manually re-run your program several times, testing with at least the following 6 input characters:

b z Q 0 } ***a literal space***

Explain what happens and why in the following two situations and place these explanations as comments in your "Title Block":

1. The user enters anything other than a lowercase character;
2. The user precedes the input character with a whitespace.

Submitting your solution

Send your source code file to the Assignment Checker with the subject line **C1A2E1_*ID***, where ***ID*** is your 9-character UCSD student ID.

See the course document titled "Preparing and Submitting Your Assignments" for additional exercise formatting, submission, and Assignment Checker requirements.

Hints:

The "underlying value" of a character simply means the value used to represent that character in the character set being used. For example, in the ASCII character set (Note B.1) the underlying value of the character ***'@'*** is 64 decimal (or if you prefer, 100 octal or 40 hexadecimal).

The most general way to represent the numerical difference between the ASCII uppercase and lowercase character sets is the expression ***'a' - 'A'***. Initialize a constant variable (Note 2.14) to that expression and use it in your code and comments as needed. Note, however, that the standard library function `toupper` provides the most portable solution, although you are not allowed to use it in this exercise. This function and its `tolower` counterpart will do the conversions in a completely portable way without regard for the specific characteristics of whatever character set is being used. For your own knowledge and for future use you should look up these two functions in your compiler's documentation, in one of the books recommended for this course, or online.

C1A2E2 (5 points – C Program)

Exclude any existing source code files that may already be in your IDE project and add a new one, naming it **C1A2E2_main.c**. Write a program in that file to display a triangle of characters on the screen.

Your program must:

1. prompt the user to enter any positive decimal integer value;
2. use nested "for" loops to display that number of lines of characters on the console screen starting in column 1, with each successive line containing one more character than the previous. Each line must end with a "diagonal" character and any preceding characters must be "leader" characters. The first line will only contain the diagonal character. The first leader character will be the least significant digit of the value entered by the user and subsequent leader characters will increase by 1, rolling back around to 0 after 9 has been used. For example, if the user inputs a 6 and the diagonal character is @, the following will be displayed:
```
@
6@
78@
901@
2345@
67890@
```
3. use the exact identifier name DIAGONAL_CHAR for a macro that represents the desired diagonal character. Embedding the actual diagonal character itself (or its actual name such as "*at*", "*wave*", "*pound*", "*hash*", "*dollar*", "*percent*", "*dot*", "*two*", "*five*", etc.) in the body of your code or in any of your comments is an inappropriate use of "magic numbers".

Manually re-run your program several times, testing with several different line counts and diagonal characters. To change the diagonal character you must change the value associated with the DIAGONAL_CHAR macro and recompile.

Submitting your solution

Send your source code file to the Assignment Checker with the subject line **C1A2E2_*ID***, where ***ID*** is your 9-character UCSD student ID.

See the course document titled "Preparing and Submitting Your Assignments" for additional exercise formatting, submission, and Assignment Checker requirements.

Hints:

Consider using the **%** operator (note 2.8) to obtain the least significant digit of an integer value.

A "nested loop" is merely a loop of any type that is within the body of another loop of any type. The code in the following example uses two "for" loops to display sequential values from 00 through 99, where the outer loop controls the most significant digit and the inner loop controls the least significant digit. "Magic Numbers" are used only for illustration. Additional code may be added where desired.

```
for (int msd = 0; msd < 10; ++msd)          // the loop that controls the MSD
{
   for (int lsd = 0; lsd < 10; ++lsd)       // the loop that controls the LSD
   {
      cout << msd << lsd << '\n';           // display the MSD and LSD
   }
}
```

For this exercise the outer loop should be used to keep track of the number of lines and the inner loop should be used to keep track of the number of leader characters on a line. A "for" loop should normally be used whenever a variable must be initialized when the loop is first entered, then tested and updated for each iteration. It is inappropriate to use a "while" loop or a "do" loop under these conditions. Be sure to choose meaningful names for your loop count variables noting that names like "i", "j", "k", "outer", "inner", "loop1", "loop2", "counter", etc., are non-informative and totally inappropriate. No more than four variables are necessary to complete this exercise and no "if" statement is necessary. If you use more than four variables or an "if" statement you are unnecessarily complicating the code.

C1A2E3 *(4 points – C++ Program)*

Exclude any existing source code files that may already be in your IDE project and add a new one, naming it **C1A2E3_main.cpp**. Write a program in that file to display a triangle of characters on the screen.

Your program must:

1. prompt the user to enter any positive decimal integer value;
2. use nested "for" loops to display that number of lines of characters on the console screen starting in column 1, with each successive line containing one more character than the previous. Each line must end with a "diagonal" character and any preceding characters must be "leader" characters. The first line will only contain the diagonal character. The first leader character will be the least significant digit of the value entered by the user and subsequent leader characters will increase by 1, rolling back around to 0 after 9 has been used. For example, if the user inputs a 6 and the diagonal character is @, the following will be displayed:
```
@
6@
78@
901@
2345@
67890@
```
3. use the exact identifier name DIAGONAL_CHAR for a **const char** variable that represents the desired diagonal character. Embedding the actual diagonal character itself (or its actual name such as "*at*", "*wave*", "*pound*", "*hash*", "*dollar*", "*percent*", "*dot*", "*two*", "*five*", etc.) in the body of your code or in any of your comments is an inappropriate use of "magic numbers".

Manually re-run your program several times, testing with several different line counts and diagonal characters. To change the diagonal character you must change the value associated with the DIAGONAL_CHAR variable and recompile.

Submitting your solution

Send your source code file to the Assignment Checker with the subject line **C1A2E3_*ID***, where ***ID*** is your 9-character UCSD student ID.

See the course document titled "Preparing and Submitting Your Assignments" for additional exercise formatting, submission, and Assignment Checker requirements.

Hints:
Consider using the **%** operator (note 2.8) to obtain the least significant digit of an integer value.

A "nested loop" is merely a loop of any type that is within the body of another loop of any type. The code in the following example uses two "for" loops to display sequential values from 00 through 99, where the outer loop controls the most significant digit and the inner loop controls the least significant digit. "Magic Numbers" are used only for illustration. Additional code may be added where desired.

```
for (int msd = 0; msd < 10; ++msd)          // the loop that controls the MSD
{
   for (int lsd = 0; lsd < 10; ++lsd)       // the loop that controls the LSD
   {
      cout << msd << lsd << '\n';           // display the MSD and LSD
   }
}
```
For this exercise the outer loop should be used to keep track of the number of lines and the inner loop should be used to keep track of the number of leader characters on a line. A "for" loop should normally

be used whenever a variable must be initialized when the loop is first entered, then tested and updated for each iteration. It is inappropriate to use a "while" loop or a "do" loop under these conditions. Be sure to choose meaningful names for your loop count variables noting that names like "i", "j", "k", "outer", "inner", "loop1", "loop2", "counter", etc., are non-informative and totally inappropriate. No more than four variables other than DIAGONAL_CHAR are necessary to complete this exercise and no "if" statement is necessary. If you use more than four variables or an "if" statement you are unnecessarily complicating the code.