

Assignment 2

C/C++ Programming I

C1A2 General Information

Assignment 2 consists of FOUR (4) exercises:

C1A2E0 C1A2E1 C1A2E2 C1A2E3

All requirements are in this document.

Related examples are in a separate file.

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-body email to the assignment checker with the subject line **C1A2_174273_U09845800** and no attachments.

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.

C1A2 General Information, continued

Using Constant Variables Instead of Macros in C++

Although macros are supported in both C and C++, *constant variables* (or inline functions if necessary) are preferred in C++ except in the few situations where only a macro can do the job.

The C and C++ keyword `const` and the C++-only keyword `constexpr` are known as *type qualifiers*. Using either of them to declare a variable makes it a constant variable. A constant variable must be explicitly initialized when declared and its value cannot be changed thereafter. That is, it is "read only".

In C++, a constant variable that is initialized with a *constant expression* can be used in contexts where a constant expression is required, such as for dimensioning arrays, specifying `case` values in `switch` statements, specifying bit-field widths, and initializing enumerators. A constant expression is an expression that can be fully evaluated at compile time rather than having to wait for the program to run. In C, a constant variable can never be used where a constant expression is required even if its initializer is a constant expression.

`const` or `constexpr`

In C++, the most common practice is to use the `constexpr` qualifier when the variable must be useable in situations where a constant expression is required, such as those mentioned above. If you then mistakenly try to initialize the variable with a non-constant expression, the compiler will generate an error so you can fix the problem. It is also a common practice to specify `constexpr` any time the initializer is a constant expression even if the variable will not be used in a situation that requires one.

Examples:

```
#define NEW_STUDENTS 15           // Avoid an inappropriate magic number in C
constexpr int NEW_STUDENTS = 15; // Avoid an inappropriate magic number in C++

constexpr double SQRT = sqrt(20); // Error - initializer is not a constant expression
const double SQRT = sqrt(20);     // Okay
```

C1A2 General Information, continued

Limiting the “Scope” of Variables

The scope of an identifier (a name) is defined as the section of code over which it is accessible. The scope of a variable declared inside a function extends from that declaration to the end of the block in which it is declared. 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. Consider the examples below. Constant variables that are being used in C++ instead of macros are exceptions. For readability these are often defined in the same place the equivalent macros would have been defined if it were C code. Otherwise, they should be defined first in the function or block that uses them.

```
1  type 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 type 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 type 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

All variables are declared on line 3, which is inside the block that starts on line 2 and ends on line 14. Thus, their scopes extend from line 3 to line 14 and all are accessible anywhere within that region. However, since variable **y** is only needed from line 6 through line 10 and variable **z** is only needed on line 10, their scopes are both wider than necessary. Regardless of scope, good practice dictates that whenever appropriate a “for” statement’s loop count variable be initialized in its “initial expression” rather than in the variable’s original declaration.

Better Declaration Placement

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

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 if appropriate. This limits their scope to the “for” statement only. 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.

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

Assume language standards compliance and any necessary standard library support unless stated otherwise. These are not trick questions and there is only one correct answer. Basing an answer on actual testing is risky. Place your answers in a plain text "quiz file" named **C1A2E0_Quiz.txt** formatted as:

a "Non-Code" Title Block, an empty line, then the answers:

1. A
2. C
- etc.

1. Which of the following is wrong regarding the standard macro named `CHAR_BIT`? (Note 2.3)
 - A. You must never define it yourself.
 - B. It is available in header files `limits.h` and `climits`.
 - C. It represents the number of bits in a byte (in data type **char**).
 - D. Program code should use it instead of a literal 8 to represent the number of bits in type **char**.
 - E. Its value is always 8
2. If your program needs both integer and floating variables and if any type from each category will work, which are usually preferred, respectively? (Notes 2.1 & 2.4)
 - A. **char** and **float**
 - B. **int** and **float**
 - C. **unsigned int** and **double**
 - D. **int** and **double**
 - E. **long long int** and **long double**
3. The data types of the literals 2 and 2.0, respectively, are: (Notes 2.1, 2.2, & 2.4)
 - A. **integer** and **long double**
 - B. **int** and **double**
 - C. **int** and **floating**
 - D. **int** and **float**
 - E. implementation dependent
4. The data type of an expression containing only arithmetic operators (+ - * / %) and type **char** and type **short** operands is: (Note 2.10)
 - A. **char**
 - B. **short**
 - C. **char**, **unsigned char**, or **short**
 - D. **int** or **unsigned int**
 - E. **int** or **long**
5. What is the biggest problem with macro SUM below, which is intended to represent the sum of 5 and 3?
`#define SUM 5+3`
(Note 2.13)
 - A. Its value might not be 8
 - B. It contains magic numbers.
 - C. There is no space around the + operator.
 - D. Macros are not supported in C++.
 - E. There is no major problem.
6. What is wrong with:
`printf("%u", sizeof(double));`
(Note 2.12)
 - A. There is nothing wrong.
 - B. **sizeof(double)** should be **sizeof(unsigned int)** to match the %u.
 - C. The data type produced by **sizeof** might not be compatible with %u.
 - D. printf cannot reliably display values produced by **sizeof**.
 - E. The value produced by **sizeof** might be negative.

Submitting your solution

Send an empty-body email to the assignment checker with the subject line **C1A2E0_174273_U09845800** and with your quiz file attached.

See the course document titled "How to Prepare and Submit 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 (ask) 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 an empty-body email to the assignment checker with the subject line **C1A2E1_174273_U09845800** and with your source code file attached.

See the course document titled "How to Prepare and Submit 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.

1. Define two macros named **LEADER_CHAR** and **DIAGONAL_CHAR**. The replacement list of each must be an arbitrary character literal. Referring to those characters by name (such as "circumflex" and "dollar"), by numeric value (such as 94 and 36), or by literal (such as '^' and '\$') anywhere else in the program, including in comments, is an inappropriate use of "magic numbers".
2. Prompt (ask) the user to enter any positive decimal integer value.
3. Use a *nested loop* 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 the diagonal character and any preceding characters must be leader characters. The first line will only contain the diagonal character. For example, if the user inputs a **4** and the leader character is **^** and the diagonal character is **\$**, the following will be displayed:

```
$  
^$  
^^$  
^^^$
```

4. DO NOT use more than two loop statements.

Manually re-run your program several times, testing with several different line counts, leader characters, and diagonal characters. To change the leader and diagonal characters you must change the character literals associated with the **LEADER_CHAR** and **DIAGONAL_CHAR** macros and recompile.

Submitting your solution

Send an empty-body email to the assignment checker with the subject line **C1A2E2_174273_U09845800** and with your source code file attached.

See the course document titled "How to Prepare and Submit Assignments" for additional exercise formatting, submission, and assignment checker requirements.

Hints:

1. You are unnecessarily complicating your code if you use any "if" statements or more than three variables.
2. A "nested loop" is merely a loop of any kind that is within the body of another loop of any kind, as in the example below. In this exercise the outer loop would be used to keep track of the number of lines and the inner loop would be used to keep track of the number of leader characters on each line. Use a "for" loop if a variable must be initialized when the loop is first entered, tested before each iteration, and updated after each iteration. Choose meaningful names for loop count variables. Names like **i**, **j**, **k**, **outer**, **inner**, **loop1**, **loop2**, **counter**, etc., are not informative and are usually inappropriate.

```
for (...)  
{  
    for (...)  
    {  
        ...  
    }  
}
```

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.

1. Declare two `constexpr`-qualified type `char` variables named `LEADER_CHAR` and `DIAGONAL_CHAR` in two separate statements on two separate lines. As part of each statement, initialize the variable to an arbitrary character literal. Referring to those characters by name (such as "circumflex" and "dollar"), by numeric value (such as 94 and 36), or by literal (such as '^' and '\$') anywhere else in the program, including in comments, is an inappropriate use of "magic numbers".

2. Prompt (ask) the user to enter any positive decimal integer value.

3. Use a *nested loop* 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 the diagonal character and any preceding characters must be leader characters. The first line will only contain the diagonal character. For example, if the user inputs a **4** and the leader character is **^** and the diagonal character is **\$**, the following will be displayed:

```
$
^$
^^$
^^^$
```

4. DO NOT use more than two loop statements.

Manually re-run your program several times, testing with several different line counts, leader characters, and diagonal characters. To change the leader and diagonal characters you must change the character literals assigned to the `LEADER_CHAR` and `DIAGONAL_CHAR` variables and recompile.

Submitting your solution

Send an empty-body email to the assignment checker with the subject line **C1A2E3_174273_U09845800** and with your source code file attached.

See the course document titled "How to Prepare and Submit Assignments" for additional exercise formatting, submission, and assignment checker requirements.

Hints:

1. You are unnecessarily complicating your code if you use any "if" statements or more than five variables.
2. A "nested loop" is merely a loop of any kind that is within the body of another loop of any kind, as in the example below. In this exercise the outer loop would be used to keep track of the number of lines and the inner loop would be used to keep track of the number of leader characters on each line. Use a "for" loop if a variable must be initialized when the loop is first entered, tested before each iteration, and updated after each iteration. Choose meaningful names for loop count variables. Names like `i`, `j`, `k`, `outer`, `inner`, `loop1`, `loop2`, `counter`, etc., are not informative and are usually inappropriate.

```
for (...)
{
    for (...)
    {
        ...
    }
}
```