Assignment 3
C/C++ Programming I

C1A3 General Information

--- No General Information for This Assignment---

Get a Consolidated Assignment 3 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 **C1A3_*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.

C1A3E0 *(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. What is output: *printf("%i\n", 6/3 + !2.2 + 3);*
   (Note 3.2)
   A. *7*
   B. *7.2*
   C. It won't compile.
   D. *5*
   E. garbage because ***6/3 + !2.2 + 3*** is type **double** but *%i* specifies type **int**

2. Predict the output from:
```
if (5 < 4)
    if (6 > 5)
        cout.put('4');
else
    cout.put('3');
else
    cout.put('2');
cout.put('1');
```
   (Note 3.15)
   A. *31*
   B. *21*
   C. *321*
   D. *41*
   E. Implementation dependent

3. Predict the output from:
```
switch (2 * 2)
{
    case 8/2:  cout << "value = 4";
    case 29:  cout << "29 ";
    case 'E':  cout << 'E' << " ";
    default:  cout << "break ";
    break; case 2/2:  cout << "Got an 'A' ";
}
```
   (Notes 3.17 & 3.18)
   A. *value = 4*
   B. *value = 429 E break*
   C. *value = 429 E*
   D. *Got an 'A'*
   E. The output is implementation dependent.

4. If the ASCII character set is being used, what gets printed by:
   putchar(putchar('z') - putchar('A'));
   (Notes 3.3 & B.1)
   A. *zA9* only
   B. *Az9* only
   C. either *zA9* or *Az9*
   D. either *zA9* or *Az9* or *9zA* or *9Az*
   E. Possibly something not listed above

5. What gets printed by:
   putchar('A') || putchar('\0x0');
   (Notes 1.5 & 3.3)
   A. *A* and garbage
   B. *A* and *\0x0*
   C. either *A* or *\0x0* but not both
   D. either *A* or *B* but not both
   E. *A* only

6. For **float** x = 5; what is the value and data type of the entire expression on the next line:
   25, sqrt(9.0), ++x, printf("123")
   (Note 3.11)
   A. 3.0 (type **double**)
   B. 3 (type **int**)
   C. 25 (type **int**)
   D. 6 (type **float**)
   E. implementation dependent

Submitting your solution

Using the format below place your answers in a plain text file named **C1A3E0_Quiz.txt** and send it to the assignment checker with the subject line **C1A3E0_*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 "How to Prepare and Submit Assignments" for additional exercise formatting, submission, and assignment checker requirements.

C1A3E1 *(3 points – C Program)*

Exclude any existing source code files that may already be in your IDE project and add a new one, naming it **C1A3E1_main.c**. Write a program in that file to compute and display a table of cubic sums. For the sake of this exercise I have defined a "cubic sum" as the sum of the cubes of all numbers from *0* through some arbitrary value >= *0*. For example, the cubic sum for the number *5* can be calculated as $0^3 + 1^3 + 2^3 + 3^3 + 4^3 + 5^3$ and has a value of *225*. Here is a table of cubic sums for *0* through *5*:

```
   nbr    cubic sum
---------------
     0            0
     1            1
     2            9
     3           36
     4          100
     5          225
```

IMPORTANT:
Your code must use a type **short** (not **unsigned short**) variable to represent the value of the **cubic sum** in the table above, but use type **int** variables for everything else. The results must be correct up to the maximum value type **short** can represent on any and every machine on which your unaltered code is compiled and run. Since compiler manufacturers are allowed to make that maximum as great as they see fit as long as it is at least **32767**, it could conceivably be so great that hundreds of digits would be required to represent it.

In addition to the above requirements, your program must:

1. prompt the user to enter an integer value >= 0 and store it in a type **int** variable;
2. compute and display a table like the one illustrated above for all values from *0* through the value entered by the user. Values must be displayed as decimal integers with no exponents or decimal points.
3. align the least significant digits in both columns for all entries in the table. Do not attempt to write code to compute the field widths needed for these columns. Instead, a fixed width of 3 for the 1st column and 10 for the 2nd is fine for the values tested in this exercise unless you start getting misalignments or simply want to make them wider. Separate the fields with at least one space so the numbers won't run together.
4. use a row of hyphens to separate the column titles from the values.
5. not use floating point literals, floating point variables, floating point functions, or floating point type casts;
6. not use an **if** statement or more than 1 looping statement;
7. not use arrays or recursion; recursion occurs when a function is called before a previous call to that function has returned, for example, when a function calls itself.

Manually re-run your program several times, testing with at least the following 3 input values:

1 25 36

If you find that any of the cubic sum values are incorrect determine if the expected values exceed the maximum value supported by type **short** on your machine, in which case they should be incorrect. Even if they are all correct they will eventually exceed the maximum if the user input value is increased sufficiently.

Suggest a possible way, without restricting user input, the program could be modified so that all values would be correct, but don't incorporate your suggestion into the code you will be submitting for grading. Instead, merely place your suggestion as a comment in the file's "Title Block". Note that even using type `unsigned long long` or type `long double` won't eliminate eventual erroneous values.

Submitting your solution

Send your source code file to the assignment checker with the subject line **C1A3E1_*ID***, where ***ID*** is your 9-character UCSD student ID.

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

Hints:

Use 1 type **int** variable to get the user input value, another to represent the value to be cubed (1, 2, 3, 4, 5, etc.), and a type **short** variable to represent the sum of all previous cubes (the cubic sum). Then implement the following algorithm, which is completely independent of the number of digits required to represent the maximum value type **short** can represent:

1. Get the user input value.
2. Initialize both the value to be cubed and the cubic sum to 0.
3. **IF** the value to be cubed is less than or equal to the user input value:
    a. Calculate the cube and add it to the cubic sum.
    b. Display the value that was cubed and the cubic sum.
    c. Increment the value to be cubed.
    d. Repeat from step 3.

    ELSE you're done!

C1A3E2 *(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 **C1A3E2_main.cpp**. Write a program in that file to reverse the digits of an arbitrary user-entered decimal integer value based solely upon its numeric value, not the individual characters entered. If the value is negative the minus sign must be displayed last. Here are some sample input values and the expected reversals:

Input	Reversal
3987	**7893**
-2645	**5462-**
100	**001**
000120	**021**
-0023	**32-**
000	**0**

Your program must:

1. prompt the user to enter any decimal integer value;
2. use `cin >>` to read the entire value at once into a type **int** variable;
3. display the variable's value and the reversed value in the format below, placing double-quotes around both for readability. For example if the user input is `-00000000000000000000000026450` the following would get displayed:
   `"-26450" in reverse is "05462-"`
4. not use any non-const variables that are not type **int** or type **bool**;
5. not use anything involving floating point types (the pow function, `math.h`, type **double**, etc.);
6. not use arrays or recursion; recursion occurs when a function is called before a previous call to that function has returned, for example, when a function calls itself.

A special case to handle a user input of zero is not necessary if your code is optimally designed.

Manually re-run your program several times, testing with at least the following 6 input values:
3 -123 0 1010 -1010 -0007000

Submitting your solution

Send your source code file to the assignment checker with the subject line **C1A3E2_*ID***, where ***ID*** is your 9-character UCSD student ID.

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

Hints:
The recommended, but not required, algorithm below uses a "do" loop to pick off and display the digits of the user input value one at a time moving right-to-left. A special case to handle zero is unnecessary.

1. Prompt the user for input then read it into a variable named **inValue**.
2. Display the required output message up to where the reversed value should start.
3. Use a Boolean variable to remember if the input value was positive or negative.
4. If the input value was negative make **inValue** positive.
5. Modulo-divide **inValue** by 10 to produce its least significant digit (LSD), then display that LSD.
6. Divide **inValue** by 10 to remove its LSD and assign the result back into **inValue**.
7. IF **inValue** is not equal to 0 repeat from step 5.
8. ELSE IF the original user input value was negative display a minus sign.
9. Finish the display.
10. You're done!

C1A3E3 *(6 points – C++ Program)*

Exclude any existing source code files that may already be in your IDE project and add a new one, naming it **C1A3E3_main.cpp**. Write a program in that file to convert an arbitrary user-entered decimal integer value into words based solely upon its numeric value, not the individual characters entered. If the value is negative the word `minus` must be first. Here are some sample input values and the expected words:

Input	Words
00000	**zero**
593	**five nine three**
-593	**minus five nine three**
-500	**minus five zero zero**
-000500	**minus five zero zero**

Your program must:

1. prompt the user to enter any decimal integer value;
2. use `cin >>` to read the entire value at once into a type **int** variable;
3. display the variable's value and the equivalent words in the format below, placing double-quotes around both for readability. For example if the user input is `-00000000000000000000000026450` the following would get displayed:
   "-26450" in words is "minus two six four five zero"
4. not use any non-const variables that are not type **int** or type **bool**;
5. not use anything involving floating point types (the pow function, `math.h`, type **double**, etc.);
6. not use arrays or recursion; recursion occurs when a function is called before a previous call to that same function has returned, for example, when a function calls itself.
7. not use any nested loops – they are totally unnecessary.

Manually re-run your program several times, testing with at least the following 6 input values:
3 -123 0 1010 -1010 -0007000

Submitting your solution

Send your source code file to the assignment checker with the subject line **C1A3E3_*ID***, where ***ID*** is your 9-character UCSD student ID.

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

Hints:
Detailed hints are on the next page…

Hints for Exercise 3:

The optional algorithm below displays a user decimal integer input value in words, one-at-a-time moving left-to-right. Since it uses integer division (both standard and modulo), which is not portable to older compilers if either operand is negative (Note 2.8), the input value is tested and made positive if necessary. There are no nested loops, part A is completed before part B begins, and part B is completed before part C begins. Only one instance of the code for each part is necessary:

Part A:

A1. Prompt the user, get his/her input, and output the display message up to the point where the first word of the value is needed.

A2. If the user input value is negative change it to positive and display the word "minus", followed by a space.

Part B ("for" loop is used):

Find a power of 10 divisor that will produce the most significant digit (MSD) of the positive input value as follows:

B1. Assign 1 to a divisor variable and the positive input value to a dividend variable.

B2. **If** the value of the dividend is greater than 9:

a. Multiply the divisor by 10; the product becomes the new divisor.

b. Divide the dividend by 10; the quotient becomes the new dividend.

c. Repeat from step B2.

Else Proceed to Part C below.

Part C ("do" loop is used):

The starting value for the divisor used in this part will be the value computed for it in Part B above. Part C will pick off the digits of the positive input value left-to-right and display them as words as follows:

C1. Assign the positive input value to a dividend variable.

C2. Divide the dividend by the divisor, which yields the MSD. Display it as a word using an 10-case switch statement (see below).

C3. Multiply the MSD by the divisor and reduce the dividend's value by that amount. (This removes the dividend's MSD.)

C4. Divide the divisor by 10; the result becomes the new divisor.

C5. **If** the new divisor is not equal to 0, repeat from step C2.

Else You are finished displaying the number in words!

<u>About the recommended "switch" statement...</u>

While the use of "magic numbers" is usually a bad idea, in some situations they are appropriate such as for the "cases" used in the "switch statement" recommended for this exercise. Specifically, each case represents a unique numeric value ranging from 0 through 9. There is no underlying meaning to these values other than the values themselves, their purpose is obvious and unmistakable, there is no possibility that they might ever need to be changed, and there is no identifier (name) that would make their meaning any clearer. Thus, the literal values should be specified directly, as follows:

```
switch (…)
{
   case 0:   …
   case 1:   …
   case 2:   …
   etc.
}
```

