1 Instructions

You may work with one partner as a group of two programmers on this lab project if you wish or you may work alone. If you work with a partner, only one member of the group shall submit the lab project to Canvas for grading. To ensure you each receive the same score, it is imperative that you follow the instructions in §5 *Submission Information* (i.e., put both of your ASURITE ID's in the submission archive filename and write both of your names, ASURITE ID's, and email addresses in the AUTHOR1: and AUTHOR2: lines of the header comment block for each source code file). What to submit for grading, and by when, is discussed in §5; read it now.

2 Learning Objectives

After completing this assignment the student shall be able to,

- Complete all of the objectives of the previous lab projects.
- Write **#include preprocessor directives** to include three C++ Standard Library header files: **cmath**, **iomanip**, and **iostream**.
- Define and use a **named constant**.
- Use **cout** to send string literals and **int** and **double** values to the output window.
- Use **endl** to display output on multiple lines of the output window.
- Define and use **int** and **double** variables and constants.
- Assign values to variables using the **assignment operator =** when writing an **assignment statement**.
- Store integer and real numbers read from the keyboard via **cin** into **int** and **double** variables.
- Write **arithmetic expressions** using these five arithmetic operators: + - * / and %.
- Understand the **operator precedence rules** and the difference between **integer division** and **floating point division**.
- Use the **static_cast<>** operator.
- Call C++ standard library **math functions**.
- Use **fixed** and **setprecision** stream manipulators to output real numbers with a specific number of digits after the decimal point.
- Use **indentation** and **spacing** to properly **format code**.

3 Prelab Exercises

We recommend that you complete prelab exercises 1-4 **before** your lab section meets.

3.1 Prelab Exercise 1 – Create main.cpp in your C++ IDE

Create a new C++ project in Repl.it (or whichever C++ IDE you are using). Name your project *lab02*. Add a source code file named *main.cpp* to the project and enter this code:

```
//****************************************************************************
// FILE: main.cpp
//
// DESCRIPTION
// Write a description of the lab project and what you are supposed to learn from
// completing it (see §2 Learning Objectives).
//
// AUTHOR1: your-name, your-asurite-id, your-email-address
// AUTHOR2: your-name, your-asurite-id, your-email-address        note: omit this line if you work alone
//
// COURSE INFO
// CSE100 Principles of Programming with C++, Fall 2020
// Lab Project 2   Day/Time: your-lab-date-and-time   TA: your-lab-ta's-names
//****************************************************************************

//----------------------------------------------------------------------------
// main() is where the program begins executing.
//----------------------------------------------------------------------------
int main()
{
    return 0;
}
```

This is the simplest C++ program one can write which builds and runs and will be the **basic template** for every lab project. This program does nothing other than start up and immediately terminate. When implementing the Lab Project 2 software requirements as discussed in §4.2 *Software Requirements*, you will augment this code.

3.2 Prelab Exercise 2 – Textbook Reading Assignment

Read §1.6 *The Programming Process* in the textbook. In particular, focus on the discussion of **pseudocode** in Item 4.

3.3 Prelab Exercise 3 – Pseudocode and Software Design

Read §4 *Lab Project* of this document, which describes what the lab project program will do. Then, come back here and read the remainder of this section. Earlier in the lectures, we discussed the five software development stages.

1. **Software requirements -** The software requirements document **what** the program is to do. For each of your lab projects, I will give you the software requirements in the *Software Requirements* of the lab project document where I describe the program.

2. **Software design -** *W*e plan **how** we will write the program so that it **meets** the software requirements, i.e., so the program does what it is supposed to do. We will discuss software design more later in this document and in the lectures.

3. **Implementation -** Using the software design from the software design stage, we write the code so that it **implements** the design.

4. **Testing -** Once we have implemented the code, we do not know if it is correct unless we test it. The goal of testing is to find and correct bugs in the code. Be aware that just because a program compiles and can be run, it does not mean the program is correct. In CSE100, we will design and use one or more **test cases** to test that the program meets the software requirements.

5. **Maintenance -** In the real world, once a program reaches a point where it is ready to be released to users, the users will begin to find bugs. We test and fix bugs in the maintenance stage. Additionally, many programs evolve over time by adding new features, culminating in new versions of the program.

The **software design** is the plan for how to write the code so that it meets the software requirements. If we were architects tasked with building a skyscraper, the software design is somewhat akin to the detailed blueprints that document **how** the building will be constructed by the construction engineers. In software development, there are many methods for documenting the design of a program. One commonly used method is to document the design in pseudocode. The prefix pseudo in this context means that pseudocode is **similar to** or **like** code written in a formal high-level programming language, but pseudocode **is not** C++ code, nor code written in any other real programming language. Rather, pseudocode is a "made-up" language and there is no standard syntax for writing pseudocode. We could simply write English sentences that document the design. This would suffice for reasonably simple programs, but in general, English is not a precise-enough language for writing pseudocode[1], so the software designer will generally write pseudocode that looks more similar to code of a formal programming language.

The primary advantage of designing the program using pseudocode is that since there is no standard syntax for pseudocode (we just make it up), the designer can focus on figuring out how to implement the program without being overly concerned with or getting sidetracked with programming language syntax. One of the most common mistakes that beginning programmers (and professional programmers, for that matter) make when starting a new program, is they will immediately begin trying to write the code, without doing any planning or performing any software design. If you do this, invariably you will be confronted with programming language syntax and syntax errors, and will begin to spend more time fighting the syntax rules of the programming language and less time on implementing correct code. The end result is usually a mess of a program that may not work, or if it works, it may work poorly and almost always, the code will be messy and highly unreadable (remember, readability is a very good thing).

For many of our lab projects, I will give you the design in the form of pseudocode and you job will be to **implement** that design by writing C++ code. Then, you will perform **testing** to ensure that your program meets the software requirements. For some lab projects, I may not give you the pseudocode design, but I **expect** and **strongly encourage** you to write your own pseudocode, designing the program **before** you give any thought to writing the C++ code. In theory, given a well-documented, unambiguous, and clear design, a well-trained monkey who knows C++ could convert that design into working C++ code.

Out of all of the five software development stages, implementation is by far the easiest step and design is generally regarded as the most difficult step, but design *can* be learned in four easy steps: (1) Study the good designs of other programmers; purchase and read software design textbooks; there is a huge amount of free and open-source software (FOSS) available for download from the internet, not every FOSS project contains well-written code conforming to a high-quality design, but many do. (2) Practice designing your own programs using the

[1]English (and probably all spoken languages) is filled with ambiguity, e.g., "Sally saw the dog in the window and she wanted it." Does these sentence convey the information that Sally wanted the dog or the window, or even perhaps that the dog is a she and the dog wanted the window? When executing a program, the CPU cannot tolerate ambiguity at all and therefore, all programming languages must have very precise syntax and semantic rules which must be followed in order to create an unambiguous sequence of instructions which the CPU can execute.

design principles you are learning in Step 1. (3) Implement your design in a programming language; in the process, you will discover the flaws in your design and learn how to better design the software; (4) Repeat steps 1-3 for around 5- to 10-thousand hours and you will become a pretty good software designer.

Here is the software design for the software requirements in §4.2. Your job will be to read, study, and understand this design, and then implement it in C++. After implementing the design, you will perform testing using three **test cases**, see Prelab Exercise 4. Note that the notation used below in the pseudocode is my own invention. Practically every programmer has their own **style** for writing pseudocode (remember I have mentioned the term **programming style** in the lectures, pseudocode style is the same sort of thing), and like many other things—including my C++ programming style—my pseudocode style evolves over time. The pseudocode I write today does not look like the pseudocode I was writing 10 or 20 years and I am always searching for a "better" way to convey the information about my software designs. A few notes on my pseudocode style:

- I use an em-dash for starting a comment, which extends to the end of the line in the same way // comments operate in C++.
- The "reserved words" of my pseudocode language are in **bold**.
- I use a **file** line (see the first line) to indicate that the following pseudocode would be implemented in a file named as specified. Of course, when implementing the pseudocode in C++, you would name the file with a .cpp filename extension.
- I italicize *identifiers_like_this* and use underscores to separate the major words of an identifier.
- I use the left arrow ← to represent assignment. When implementing this pseudocode in C++ code, you would use **=** because **=** is the C++ assignment operator.
- I use **real** in my pseudocode when something is a real number. Recall in C++, that we specify real using the **double** reserved word. I do not use double in pseudocode because it conveys little meaning, unless one understands the etiology of the word.
- I use mathematical notation when I can if it enhances the pseudocode readability.

```
— ————————————————————————————————————————————————————————————————————————————————————————————————————
— This is the pseudocode for Lab Project 2, which shall be implemented in a file named main.cpp.
— ————————————————————————————————————————————————————————————————————————————————————————————————————
file main
    define int constant NUM_STUDENTS ← 3

    — ————————————————————————————————————————————————————————————————————————————————————————————————
    — This is the main function of the program. The first statement which is executed when the program is run is the first
    — statement of main(). The initialization of the constant NUM_STUDENTS occurs before main() starts.
    — ————————————————————————————————————————————————————————————————————————————————————————————————
    function main()
        — Define three integer variables to store the exam scores for Homer, Lisa, and Ralph.
        define int variable exam_homer, exam_lisa, exam_ralph

        — Display a prompt informing the user that we are waiting on them to enter the exam score for Homer. Then, read the score from
        — the keyboard, saving the entered value in the variable exam_homer.
        output "Enter exam score for Homer? "
        exam_homer ← input integer from keyboard

        — Display a prompt informing the user that we are waiting on them to enter the exam score for Lisa. Then, read the score from
         — the keyboard, saving the entered value in the variable exam_lisa.
        output "Enter exam score for Lisa? "
        exam_lisa ← input integer from keyboard

        — Display a prompt informing the user that we are waiting on them to enter the exam score for Ralph. Then, read the score from
        — the keyboard, saving the entered value in the variable exam_ralph.
        output "Enter exam score for Ralph? "
        exam_ralph ← input integer from keyboard

        — Define a real variable named exam_avg to store the exam average we are calculating. Then, calculate the exam average, which
        — is the sum of the three exam scores divided by 3 (but we shall use the named constant NUM_STUDENTS here because one of
        — your lab project objectives is to learn how to define and use a named constant). Do you understand why I put the summation
        — inside parentheses? If not, before you implement the C++ code you should learn why. The final operation is to store the value
        — of the exam average in the variable exam_avg.
```

```
        define real variable exam_avg ← (exam_h + exam_l + exam_r) ÷ NUM_STUDENTS

        — Define a real variable named sq_diff_homer to store the square of the difference of Homer's exam score and the exam average.
        — Then, perform the calculation, and finally, store the calculated value in the defined variable.
        define real variable sq_diff_homer ← (exam_homer - exam_avg)^2
        — Repeat the above operation for Lisa and Ralph.
        define real variable sq_diff_lisa ← (exam_lisa - exam_avg)^2
        define real variable sq_diff_ralph ← (exam_ralph - exam_avg)^2

        — Define a real variable named exam_variance, calculate the exam variance using standard statistical formula, and then store the
        — calculated value in the defined variable. Do you know why the parentheses are there? Are all of the parentheses technically
        — required in order to calculate the correct value or did the instructor just put some of them there in an attempt to make the
        — formula more readable? If you do not know, learn this before your lab section meets. If you are still confused, then when your
        — lab meets, ask your lab instructor to explain it.
        define real variable exam_variance ← (sq_diff_homer + sq_diff_lisa + sq_diff_ralph) ÷ (NUM_STUDENTS - 1)

        — Define a real variable named exam_stddev, calculate the exam standard deviation using standard statistical formula, and then
        — store the calculated value in the defined variable.
        define real  variable exam_stddev ← √exam_variance

        — Output a blank line. On the line below the blank line, output the average exam score. Finally, on the next line, output the exam
        — standard deviation. Note: when implementing this pseudocode in C++, you must configure the cout output stream to display
        — real numbers in fixed notation with two digits after the decimal point. In pseudocode I do not worry about formatting issues.
        output blank line
        output "The average exam score is:     ", exam_avg, "%"
        output "The exam standard deviation is: ", exam_stddev

        — Return 0 from main() to inform the operating system that the program successfully completed.
        return 0
    end function main

end file main
```

3.4 Prelab Exercise 4 – Testing

Please re-read §3.4 of the Lab Project 1 document which discusses testing. One of the most important stages of the software development process is **testing**. Just because a program compiles does not mean it is perfect. It only means there are no **syntax errors** in the code, but there could still be, and most likely will be, **logical errors** or **bugs** in the code that will cause the program to not meet the software requirements.

For example, suppose you wrote the code such that when dividing the sum of the exams by *NUM_STUDENTS* during the exam average calculation, you did not properly typecast either the sum to **double** or the value of *NUM_STUDENTS* to **double**. This will cause the division to be performed as **integer division** and it will cause the exam average to be truncated, losing the digits after the decimal point. Then, because the exam average was incorrectly calculated, the variance and standard deviation will also be incorrectly calculated. The end result is that the program will output incorrect data due to this bug.

Testing is an important part of software engineering education—there are entire classes devoted to the subject—and there are many different methods for testing programs. In CSE100 we will focus on what is called **acceptance testing**, which just means that we will simply run the program and test it with different inputs to ensure that the program output meets the software requirements.

A **test case** is a specification that is used to test the program against. The test case specifies **input data** to the program and documents **actual output** (or **behavior** if you wish). The test case is written **before** executing the program; in fact, test cases can be, written as early as during the **software requirements stage** of the software life cycle.

To **perform a test case**, after the program compiles, we run it, and the test case input data is used as input to the program. The program produces some output, and we compare the expected output (which was documented in the test case) to the actual, observed output. If they match, then the test case **passes** and should be documented as such. If the observed output does not match the expected output, then there are two possibilities:

(1) **Test Case Failure Reason 1:** The **test case is incorrect**, i.e., the expected output documented in the test case is incorrect because you made a mistake in determining what the expected output should be; or

(2) **Test Case Failure Reason 2:** There is a **bug** in the program.

For situation (1) you should rewrite the test case to be correct. For situation (2) you should find and correct the bugs in the program. In either case, after fixing either your test case or the program code, you repeat running the program and performing the test cases until the program **passes** all of them.
As an example of a test case, a test case for this lab program could be written and documented like this,

Test Case 1
Description: Tests that the program computes and displays correct exam average and standard deviation.
Input Data: Homer's exam score is 73, Lisa's is 100, and Ralph's is 44.
Expected Output (test case input is shown in bold):
Enter Exam Score for Homer? **73**
Enter Exam Score for Lisa? **100**
Enter Exam Score for Ralph? **44**

The exam average score is:****** 72.33% *note: the asterisks represent space characters*
The exam standard deviation is:* 28.01 *to be printed. Do not print asterisks.*

When we perform the test case, we would run the program and feed it the input data. The program will output results, and we check the results to see if they match the expected output. Once the test case **passes**, we can finish documenting the test case like this,

Test Case 1
Description: Tests that the program computes and displays correct exam average and standard deviation.
Input Data: Homer's exam score is 73, Lisa's is 100, and Ralph's is 44.
Expected Output (test case input is shown in bold):
Enter Exam Score for Homer? **73**
Enter Exam Score for Lisa? **100**
Enter Exam Score for Ralph? **44**

The exam average score is:****** 72.33%
The exam standard deviation is:* 28.01

Actual Output:
Enter Exam Score for Homer? **73**
Enter Exam Score for Lisa? **100**
Enter Exam Score for Ralph? **44**

The exam average score is:****** 72.33%
The exam standard deviation is:* 28.01

Test Case Result: **passed**

If the expected and actual outputs do not match and you cannot find the bug(s) in either the test case calculations or in the program, then document that the test case **failed**, e.g.,

Test Case 1
Description: Tests that the program computes and displays correct exam average and standard deviation.
Input Data: Homer's exam score is 73, Lisa's is 100, and Ralph's is 44.
Expected Output (test case input is shown in bold):
Enter Exam Score for Homer? **73**
Enter Exam Score for Lisa? **100**
Enter Exam Score for Ralph? **44**

The exam average score is:****** 72.33%


```
The exam standard deviation is:* 28.01

Actual Output:
Enter Exam Score for Homer? 73
Enter Exam Score for Lisa? 100
Enter Exam Score for Ralph? 44

The exam average score is:****** 72.00%
The exam standard deviation is:* 27.89

Test Case Result: failed
```

For this lab project, you shall write and complete three test cases, each with different input data, documenting the test cases and the testing results in the header comment block of your program, as shown below. You may use the Test Case 1 from the previous page as one of your three test cases.

For Test Cases 2 and 3: (1) Make up some values for the three exam scores, each score shall be in [0, 100]; (2) Use your calculator or Excel to calculate what the exam average and standard deviation is for these three scores; (3) Document the test cases in the source code file header comment block, specifying what the expected program output should be; (4) After the code is implemented and you are ready to perform acceptance testing, perform the test cases following the procedure described on p. 4; (5) If the actual output does not match the expected output, determine which of Test Case Failure Reason 1 or Test Case Failure Reason 1 is causing the test case to fail; (6) Fix the problem as described on p. 4; (7) When the test case finally passes, document that it passed in the Test Case Result section of the test case; if you cannot get the test case to pass, then document that the test case failed in the Test Case Results

```
//**************************************************************************
// FILE: main.cpp
//
// DESCRIPTION
// Write a description of the lab project and what you are supposed to learn
// from completing it (see §2 Learning Objectives).
//
// AUTHOR1: your-name, your-asurite-id, your-email-address
// AUTHOR2: your-name, your-asurite-id, your-email-address
//
// COURSE INFO
// CSE100 Principles of Programming with C++, Fall 2020
// Lab Project 2   Day/Time: your-lab-date-and-time   TA: your-lab-ta's-names
//==========================================================================
//                        TESTING RESULTS
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// Test Case 1
// Description: Tests that the program computes and displays correct exam aver-
// age and standard deviation.
// Input Data: Homer's exam score is 73, Lisa's is 100, and Ralph's is 44.
//
// Expected Output:
// ------------------------------------------------------------
// Enter Exam Score for Homer? 73
// Enter Exam Score for Lisa? 100
// Enter Exam Score for Ralph? 44
//
// The exam average score is:****** 72.33%
// The exam standard deviation is:* 28.01
//
// Actual Output:
// ------------------------------------------------------------
// After performing the test case, copy and paste the actual
// output from your program to here
//
```

```
// Test Case Result: Write passed or failed
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// Test Case 2
// Document test case 2 here
//~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
// Test Case 3
// Document test case 3 here
//*****************************************************************************
```

4 Lab Project

4.1 Background

Suppose a small class contains three students: Homer, Lisa, and Ralph. Each student takes an exam and the exam score is an integer in the range [0, 100]. The exam average can be computed using the formula,

$$exam_{avg} = \frac{(exam_H + exam_L + exam_R)}{\text{NUM\_STUDENTS}}$$

where $exam_H$ is Homer's exam score, $exam_L$ is Lisa's exam score, $exam_R$ is Ralph's exam score, and NUM_STUDENTS is an integer constant which is equivalent to 3. In statistics, the *variance* of this group of three scores is defined to be,

$$exam_{variance} = \frac{(exam_H - exam_{avg})^2 + (exam_L - exam_{avg})^2 + (exam_R - exam_{avg})^2}{\text{NUM\_STUDENTS} - 1}$$

And the standard deviation[2] is,

$$exam_{\text{stddev}} = \sqrt{(exam_{variance})}$$

4.2 Software Requirements

Here are the software requirements for your lab project:

1. When the program starts, it shall display the following prompt asking the user to enter the exam score for Homer. Note that * represents the space character, which shall follow the ? character. (Of course, there are implicit space characters separating the words of the sentence, but we do not usually make such spaces explicit when documenting a program.)

Enter exam score for Homer?*

2. When displaying the prompt string in SWR1, the name of the student shall be displayed.
3. The program shall read the exam score for Homer from the keyboard.
4. After the score for Homer is read in Step 3, SWR's 1–3 shall be repeated for students Lisa and Ralph. After the score for Ralph is entered, the user's view shall be as follows where **user input** is shown in bold and the asterisks represent spaces.

Enter exam score for Homer?***73**
Enter exam score for Lisa?***100**
Enter exam score for Ralph?***44**

5. After reading the score for Ralph, the program shall output a blank line.
6. The program shall calculate the exam average using the formula in §4.1 *Background*.
7. The program shall calculate the exam sample variance (dividing by the number of students minus 1) using the formula in §4.1.
8. The program shall calculate the exam sample standard deviation using the formula in §4.1.
9. After SWR's 4–8 are performed, the program shall display the exam average as a real number, with two digits after the decimal point, right-justified in a field of width 6, conforming to this format where the asterisks are space characters.

The average exam score is:******ddd.dd%

[2] In statistics, there is a difference between the **population standard deviation** and the **sample standard deviation**. To calculate the population standard deviation, we would divide by *NUM_STUDENTS* when calculating the exam variance. To calculate the sample standard deviation, we would divide by *NUM_STUDENTS* - 1. I forgot almost everything I learned about statistics in college, so I do not remember the difference between the two, but for this lab project, we are calculating the sample standard deviation.

where *d* is a decimal digit 0-9. There shall be six spaces between the : character and the exam average. Note that when the exam average is less than 100%, the leftmost *d* digit will not be displayed and there will be seven spaces between the **:** character and the value. Note the % symbol following the average.

10. After SWR9 is performed, the program shall display the exam sample standard deviation as a real number, with two digits after the decimal point, right-justified in a field of width 6, conforming to this format where the asterisk is a space character.

The exam standard deviation is:**ddd.dd*

where *d* is a decimal digit 0-9. There shall be one space between the : character and the value of the standard deviation. Note that when the standard deviation is less than 100, the leftmost *d* digit will not be displayed and there will be two spaces between the : character and the value.

11. Following the implementation of SWR10, the program shall end.
12. Note: the output from the program shall be as shown in the documentation of Test Case 1 on p. 5, given that particular set of input data.

4.3 Miscellaneous Implementation Notes

1. Be sure the output from your program matches that as described in the software requirements.

2. To display real numbers (i.e., doubles) with 2 digits after the decimal point, you must,

   a. Write a preprocessor directive #include <iomanip> at the top of the source code file with the other #include directives.
   b. Write cout << fixed << setprecision(2); before SWR9 is implemented.

3. Since each of the exam scores is an integer and *NUM_STUDENTS* is an integer, then when we divide the sum of the exam scores by *NUM_STUDENTS*, we would be dividing an integer by an integer, so the result (the quotient) would also be an integer and we would lose any digits after the decimal point. For example, if the three exam scores were 55, 76, and 21, the calculated average would be (55 + 76 + 21) / 3 = 152 / 3 = 50 (the correct average should be 50.667). Consequently, when calculating the exam average, use the static_cast<double> operator to typecast the value of *NUM_STUDENTS* from int to double,

   exam_avg = (exam_homer + exam_lisa + exam_ralph) / static_cast<double>(NUM_STUDENTS);

4. When calculating the squares of the differences, you can either use the *pow*() function from the C++ Standard Library, e.g.,

   double sq_diff_homer = pow(exam_homer - exam_avg, 2);

   or you can calculate the square using multiplication,

   double sq_diff_homer = (exam_homer - exam_avg) * (exam_homer - exam_avg);

   If you choose to use *pow*(), the declaration of the *pow*() function is in a C++ Standard Library header file named *cmath*, which must be included with a #include directive.

5. Note that when calculating the variance, we shall divide the sum by *NUM_STUDENTS* - 1, but unlike Item 3 (calculating the exam average), we do not need to typecast *NUM_STUDENTS* - 1 to **double** because the numerator will already be a **double** due to the fact that variable *exam_avg* was defined as a **double**.

6. The C++ Standard Library function for calculating the square root of a number is *sqrt*() and it is also defined in the *cmath* header file. To call *sqrt*() write,

   exam_stddev = sqrt(exam_variance);

7. **Important note for Microsoft Visual Studio users** (If you are **not** using MSVS as your C++ IDE, then **ignore** this item). Because of the way Microsoft has written their version of the C++ Standard Library, if you attempt to pass an **int** to *sqrt*(), the compiler will report a syntax error for reasons that will become apparent later in the course. In this case, you would need to typecast the **int** value to a **double** value, e.g.,

   exam_stddev = sqrt(static_cast<double>(exam_variance));

4.4 Additional Programming Requirements

1. As discussed in §3.1 *Prelab Exercise 1 – Create main.cpp in your C++ IDE* and §3.4 *Prelab Exercise 4 – Testing*, write a **header comment block** at the top of your source code containing the information described in those sections. If you work with a partner, be certain to write both the AUTHOR1: and AUTHOR2: lines and provide the requested information for each partner. *If you fail to do this, the partner who does not submit the zip archive to Canvas for grading will receive no points on this lab project.*

2. As described in §3.4 Prelab Exercise 4 – Testing, be certain that you document all three test cases in the header comment block.

3. Always write your code in a way to enhance readability. This includes writing comments to explain what the code is doing, properly indenting the statements inside *main*(), and using blank lines to separate the various parts of the program.

5 Submission Information

- For help information about how to use Canvas, please visit the Canvas Student Guide.
- You are permitted to work in a group of 2 students. In Canvas, we have created 125 groups, each limited to a maximum of two students. You may work alone or if you wish to work on a team of two, then read this help documentation about how to join a Canvas group in this page of the Student Guide, and then the two team members can each add themselves to an empty group.
- When you are ready to submit your Lab 1 project, begin by using the file explorer program of your operating system to create a new empty folder named lab02-asurite where asurite is your ASURITE user id that you use to log in to MyASU (e.g., mine is kburger2 so my folder would be named lab02-kburger2). If you worked with a partner in a team of two on the project, put both of your ASURITE id's in the name of the new folder, e.g., lab02-asurite1-asurite2.
- If you named your Repl.it project *lab02*, then when you download your project, Repl.it will download a zip archive named *lab02.zip* containing *main.cpp*. Extract *main.cpp* from the downloaded zip archive and copy *main.cpp* to the empty lab02-*asurite* or lab02-*asurite1*-*asurite2* folder. Then, compress the folder creating a zip archive named lab02-*asurite*.zip or lab02-*asurite1*-*asurite2*.zip.
- Submit the lab02-*asurite*.zip or lab02-*asurite1*-*asurite2*.zip archive to Canvas by the deadline using the *L2: Lab Project 2* submission page.
- If you worked with a partner, the first student who adds him- or herself to the group will be designated as the *group leader*. If you worked in a team, the group leader shall upload the solution zip archive to Canvas. If you worked along, then obviously, you shall submit it.
- Once you submit a file in Canvas for grading, you can download the zip archive and verify that it is not corrupted and that it is named correctly and contains the correct *main.cpp* file. Do this as soon as you submit!
- If your program does not compile due to syntax errors or if it does not run correctly because one or more of the software requirements are violated, then upload what you have completed for grading anyway because you will generally receive partial credit points for effort.
- The submission deadline is **11:59pm Sat 13 Feb**. We shall permit lab submissions, up to 48 hours after the deadline, or in other words, you must submit before the 48-hour late submission deadline of **11:59pm Mon 15 Feb**. The penalty for a submission after the Sat deadline but prior to 11:59pm Sun 14 Feb will result in a 10% penalty deduction (-0.5 pts). The penalty for a submission after 11:59pm Sun 14 Feb but prior to the 48 hour late submission deadline of 11:59pm Mon 15 Feb shall be a 20% deduction (that is, -1 pt).
- The Canvas submission link will become unavailable exactly at 11:59pm on Mon 15 Feb and you will not be able to submit your project for grading after that time. We do not accept emailed submissions or submissions that are more than 48 hours later. Do not ask.
- Consult the online syllabus for the academic integrity policies.

6 Grading Rubric

1. **Testing.** Test the student's program using these three test cases.

Content intentionally omitted

2. Assigning Points (0 to 5 pts)

a. Submitted program does not compile due to syntax errors. Almost none of the required code was implemented. Assign **1 pt**.
b. Submitted program does not compile. Only around 25-50% of the code is implemented or correctly implemented. Assign **2 pts**.
c. Submitted program does not compile. More than around 50% of the code is implemented or correctly implemented. Assign **3 pts**.
d. Submitted program compiles. Run the program against the three test cases. If one or more test cases fails and only less than around 50% of the code is correctly implemented. Assign **3 pts**.
e. Submitted program compiles. Run the program against the three test cases. If one or more test cases fails and more than around 50% of the code is correctly implemented. Assign **4 pts**.
f. Submitted program compiles. Run the program against the three test cases. All test cases pass. Assign **5 pts**.

3. The Lab Project 1 Deadline was **11:59pm on Sat 13 Feb**.
    a. Canvas will automatically deduct 10% (-0.5 pts) for a submission between **11:59pm Sat 13 Feb** and **11:59pm Sun 14 Feb**.
    b. Canvas will automatically deduct 20% (1 pt) for a submission between **11:59pm Sun 14 Feb** and **11:59pm Mon 15 Feb**.
    c. After Mon 15 Feb, submissions are not accepted for any reason.

