## Programming Assignment 1 Getting Started with C++

**Time due: 11:00 PM Tuesday, April 7**

The purpose of this assignment is to have you start learning how to use the g++ and either the Visual C++ or Xcode environments, and understand a variety of programming errors.

Here's what you are to do:

1. (optional) Obtain a copy of [Visual C++](https://web.cs.ucla.edu/classes/spring20/cs31/visualcpp.html) and install it. You don't need to do this if you prefer to use Visual C++ on the SEASnet computers (~~in the lab or~~ remotely), or if you prefer using Xcode.

(optional) Obtain a copy of [Xcode](https://web.cs.ucla.edu/classes/spring20/cs31/xcode.html) and install it. You don't need to do this if you prefer using Visual C++.

1. Enter [this C++ program](https://web.cs.ucla.edu/classes/spring20/cs31/Projects/1/spec.html#program) into your development environment. Do not change the program.
2. Build the executable from the program. (Fix any typos you may have made when entering the program.)
3. Execute the program with a variety of reasonable input integers to see if it runs as one would expect from reading the source code. (If the Visual C++ console window disappears when your program finishes executing, before you have a chance to see the output, you probably forgot to do step 4 from the [Visual C++](https://web.cs.ucla.edu/classes/spring20/cs31/visualcpp.html#step4) writeup, or you started execution by selecting the Start Debugging item from the Debug menu or by double-clicking on the .exe file. What you want to do is select the Start Without Debugging item from the Debug menu; if there is no such menu item, [fix it](https://web.cs.ucla.edu/classes/spring20/cs31/visualcpp.html#step7) as directed in step 7.)
4. Using the program as given, without changing it in any way, run it with input integers that cause it to produce incorrect, unusual, or nonsensical output. (Notice we're saying to try input integers, not input like 743.907 or wash your hands.)
5. Starting from the program as given, introduce into the source code at least one error that someone might make that, while not preventing a successful build, causes the program when it runs to produce incorrect results from reasonable input.
6. Again starting from the program as given, introduce at least two distinct types of mistakes that someone might make, each of which would cause the program to fail to compile correctly.

You should create a separate project for each of steps 2, 6, and 7, since you're not allowed to have multiple files in the same project if more than one has a main routine.

In addition to running the programs under Visual C++ or Xcode, run them using g31 as the [g++ with Linux](https://web.cs.ucla.edu/classes/spring20/cs31/linux.html) writeup tells you. (As the [Project Requirements](https://web.cs.ucla.edu/classes/spring20/cs31/requirements.html) document tells you, "run using g31" is shorthand for "run using g31 on cs31.seas.ucla.edu" — that specific command (g31, not g++) on a SEASnet machine reached via that specific name.)

What you will turn in for this assignment is a compressed file in zip format containing exactly four files:

1. A file named **original.cpp** that contains the program as given.
2. A file named **logic\_error.cpp** with the program you produced in step 6.
3. A file named **compile\_error.cpp** with the program you produced in step 7.
4. A file named **report.docx** or **report.doc** (in Microsoft Word format) or **report.txt** (an ordinary text file) that describes the input you provided in step 5 and each of the errors you introduced into the logic\_error.cpp and compile\_error.cpp programs. Briefly discuss any error messages the compiler reported, and incorrect, unusual, or nonsensical results. This report may well end up being much less than a page long.

The zip file itself may be named whatever you like.

Do **not** include anything else in the zip file. (Some Windows users seem not to be aware of [Windows filename extensions](https://web.cs.ucla.edu/classes/spring20/cs31/filenameext.html), so end up putting the wrong files in their zip file.) To create a zip file on a SEASnet machine, you can select the four files you want to turn in, right click, and select "Send To / Compressed (zipped) Folder". Under macOS, copy the files into a new folder, select the folder in Finder, and select File / Compress "folderName"; make sure you copied the files into the folder instead of creating aliases to the files.

We will be using software tools to help us grade your projects, so there are certain requirements you must meet for the tools to work:  **The zip file you turn in for this project must have exactly four files in it, with exactly the names indicated. If you do not follow these requirements, your score on this project will be zero.** "Do you mean that if I do everything right except misspell a file name or include an extra file, I'll get no points whatsoever?" Yes. That seems harsh, but attention to detail is an important skill in this field. A draconian grading policy certainly encourages you to develop this skill.

The only exception to the requirement that the zip file contain exactly four files of the indicated names is that if you create the zip file under macOS, it is acceptable if it contains the additional files that the macOS zip utility sometimes introduces: \_\_MACOSX, .DS\_Store, and names starting with .\_ that contain your file names.

By April 6, there will be links on the class webpage that will enable you to turn in your zip file electronically. Turn in the file by the due time above. Remember that most computing tasks take longer than expected; this applies especially to steps 1, 2, and 3 above. Start this assignment now!

Use this project as an opportunity to learn what happens when you make mistakes. After you've turned in what's required for this project, play around. Introduce a mistake into the program and see what happens. Fix it. Introduce a different mistake and see what happens then. Fix it. Keep doing this so you can see the kinds of problems that might arise when you develop your own programs and what the compilers say for each particular problem (if they even detect them at all). See what happens if you make more than one mistake in the program. Will they all be detected? Will an earlier mistake interfere with the reporting of a later one?

Here is the C++ program:

// Code for Project 1

// Report poll results

#include <iostream>

using namespace std;

int main()

{

int numberSurveyed;

int obey;

int disobey;

cout << "How many people were surveyed? ";

cin >> numberSurveyed;

cout << "How many of them are obeying the stay-at-home and other health orders? ";

cin >> obey;

cout << "How many of them are disobeying the stay-at-home and other health orders? ";

cin >> disobey;

double pctObey = 100.0 \* obey / numberSurveyed;

double pctDisobey = 100.0 \* disobey / numberSurveyed;

cout.setf(ios::fixed);

cout.precision(1);

cout << endl;

cout << pctObey << "% say they are obeying the orders." << endl;

cout << pctDisobey << "% say they are disobeying the orders." << endl;

if (obey > disobey)

cout << "More people are obeying than disobeying the orders." << endl;

else

cout << "More people are disobeying than obeying the orders." << endl;

}

## Programming Assignment 2 Taxing Your Patience

**Time due: 11:00 PM Thursday, April 16**

Waiting to be taken up by the state legislature is a proposal for a major reform of the state income tax code. Your job is to write a program that determines how much a taxpayer would pay under the plan.

Your program must accept as input the taxpayer's name, taxable income, occupation, and number of children. The output will tell how much tax that the taxpayer would pay under the plan.

Here is an example of a dialog with the program (user input is shown in **boldface**):

Name: **Samuel Henryi**

Taxable income: **110000**

Occupation: **engineer**

Number of children: **6**

---

Samuel Henryi would pay $3800.00

According to the plan:

* The first $50000 of taxable income is taxed at 4%.
* The next $70000 of taxable income is taxed at 6% in general. However, if the taxpayer's occupation is engineer or scientist (so spelled, entirely in lower case), the income in this bracket is taxed at only 5%.
* The amount of taxable income that exceeds $120000 is taxed at 9%.
* If the taxable income is under $120000, the tax is reduced by $200 per child; however, the tax is never allowed to go below zero.

As an example, Samuel above would incur $2000 of tax liability for the first $50000 of his income, plus $3000 for the next $60000 of income in the second bracket, for a total of $5000. That tax is reduced by $200 per child, so he would pay $3800.

Here's another example:

Name: **Moe Szyslak**

Taxable income: **29544.69**

Occupation: **bartender**

Number of children: **0**

---

Moe Szyslak would pay $1181.79

You can test your understanding of the tax schedule by experimenting with the [tax proposal calculator](https://web.cs.ucla.edu/classes/spring20/cs31/Projects/2/calc.html) we found at the legislature's website.

Your program must collect the information for one taxpayer in the manner indicated by the examples, and then write to couta line with three hyphens only (no spaces or other characters), followed by exactly one line in a format required below. Our grading tool will judge the correctness of your program by examining only the line following the line with three hyphens (and verifying that there are no additional lines). That one line you write must be in one of the following five forms; the text must be **identical** to what is shown, except that italicized items are replaced as appropriate:

* If an empty string was provided for the name:  
     You must enter a name
* If the taxable income is negative:  
     The taxable income must be nonnegative
* If an empty string was provided for the occupation:  
     You must enter an occupation
* If the number of children is negative:  
     The number of children must be nonnegative
* If the input is valid and none of the preceding situations holds:  
     name would pay $amount

In the last case, name must be the name the user entered, and amount must be the correct answer, shown as a non-negative number with exactly two digits to the right of the decimal point. The lines you write must not start with any spaces. If you are not a good speller or typist, or if English is not your first language, be especially careful about duplicating the messages **exactly**. Here are some foolish mistakes that may cause you to get very few points for correctness on this project, no matter how much time you put into it, because the mistake will cause your program to fail most or all of the test cases we run:

* Not writing to cout a line with exactly three hyphens in all cases.
* Writing any spaces on the line that is supposed to have three hyphens.
* Writing more than one line after the line with three hyphens. Don't, for example, add a gratuitous "Tax plan details subject to change."
* Writing lines to cerr instead of cout.
* Writing lines like these:
* Montgomery Burns woud pay $12345678.90 misspelling
* Montgomery Burns Would pay $12345678.90 wrong capitalization
* Montgomery Burns pays $12345678.90 wrong text
* Montgomery Burns would pay $ 12345678.90 extra space
* Montgomery Burns would pay $12345678.90. extra period
* Montgomery Burns would pay $12345678.900 extra digit
* Montgomery Burns would pay 12345678.90 missing dollar sign
* Emmett Brown would pay $98765 missing decimal point and digits

Your program must gather the name, the taxable income, the occupation, and the number of children in that order. However, if you detect an error in an item, you do not have to request or get the remaining items if you don't want to; just be sure you write to cout the line of three hyphens, the required message, and nothing more after that. If instead you choose to gather all input first before checking for errors, and you detect more than one error, then after writing the line of three hyphens, write only the error message for the earliest erroneous item.

You will not write any loops in this program. This means that each time you run the program, it handles only one taxpayer. It also means that in the case of bad input, you must not keep prompting the user until you get something acceptable; our grading tool will not recognize that you're doing that.

A string with no characters in it is the empty string. A string with at least one character in it is not the empty string, even if the only characters in it are things like spaces or tabs. Although realistically it would be silly to have a taxpayer name consisting of seventeen spaces and nothing more, treat that as you would any other non-empty string: as a valid name. (Since you don't yet know how to check for that kind of situation anyway, we're not requiring you to.)

The one kind of input error that your program does **not** have to deal with, because you don't yet know how to do so, is not finding a number in the input where a number is expected. We promise that our grading tool will not, for example, supply the text not enough when your program requests the income or the number of children. We also promise that our grading tool will supply an integer for the number of children, not a number like 2.5.

The correctness of your program must not depend on undefined program behavior. Your program could not, for example, assume anything about n's value at the point indicated:

int main()

{

int n;

int m = 42 \* n; // n's value is undefined

…

What you will turn in for this assignment is a zip file containing these two files and nothing more:

1. A text file named **tax.cpp** that contains the source code for your C++ program. Your source code should have helpful comments that tell the purpose of the major program segments and explain any tricky code.
2. A file named **report.docx** or **report.doc** (in Microsoft Word format) or **report.txt** (an ordinary text file) that contains:
   1. A brief description of notable obstacles you overcame. (In Project 1, for example, some people had the problem of figuring out how to work with more than one version of a program in Visual C++.)
   2. A list of the test data that could be used to thoroughly test your program, along with the reason for each test. You don't have to include the results of the tests, but you must note which test cases your program does not handle correctly. (This could happen if you didn't have time to write a complete solution, or if you ran out of time while still debugging a supposedly complete solution.) For Project 1, for example, such a list, had it been required, might have started off like this:

More people surveyed than the total obeying and disobeying (1000, 413, 382)

Fewer people surveyed than the total obeying and disobeying (500, 413, 382)

Zero people surveyed (0, 100, 100)

Data giving a non-integer percentage (1000, 413, 382)

More people obeying than disobeying (1000, 413, 382)

Equal number of people obeying and disobeying (1000, 500, 500)

…

By April 15, there will be links on the class webpage that will enable you to turn in your zip file electronically. Turn in the file by the due time above. Give yourself enough time to be sure you can turn something in, because we will not accept excuses like "My network connection at home was down, and I didn't have a way to copy my files and bring them to a SEASnet machine." There's a lot to be said for turning in a preliminary version of your program and report early (You can always overwrite it with a later submission). That way you have something submitted in case there's a problem later. Notice that most of the test data portion of your report can be written from the requirements in this specification, before you even start designing your program.

The writeup [Some Things about Strings](https://web.cs.ucla.edu/classes/spring20/cs31/strings.html) tells you what you need to know about strings for this project.

As you develop your program, periodically try it out under another compiler (g31 on cs31.seas.ucla.edu if you're doing your primary development using Visual C++ or Xcode). Sometimes one compiler will warn you about something that another is silent about, so you may be able to find and fix more errors sooner. If running your program under both environments with the same input gives you different results, your program is probably relying on undefined behavior (such as using the value of an uninitialized variable), which we prohibit.

## Programming Assignment 3 You Have Your Orders

**Time due: 11:00 PM Thursday, May 7**

### Introduction

You work for a manufacturer of N95 respirators. Orders for cases of these masks come in from various states and your company is filling them as quickly as it can. The orders and their status of being filled or unfilled are encoded in a string that you will need to process. We will describe the format of the string and what you must do to process it.

This program was written in C++. The program takes in orders of N95 masks come in from various states in the US and fills them as quickly as it can. The orders and their status of being filled or unfilled are encoded in a string the program can process.

First, we define some terms.

A state code is one of the following 54 two-letter codes for U.S. states and some territories, with each letter being in either upper or lower case (so CA Ca cA and ca are all state codes): AL AK AZ AR CA CO CT DE DC FL GA GU HI ID IL IN IA KS KY LA ME MD MA MI MN MP MS MO MT NE NV NH NJ NM NY NC ND OH OK OR PA PR RI SC SD TN TX UT VT VA WA WV WI WY

A digit is one of the ten digit characters 0 through 9.

An order status is a +, indicating that an order has been filled, or a -, indicating that it is has not been filled.

A state order is a state code immediately followed by a sequence of one or more digits immediately followed by an order status. For example, CA132+ and ms6- are state orders; HI 24+ is not, because of the space character between the I and the 2. (The intent is that the digit(s) represent the number of cases of masks in this order for the state.)

An order string is a sequence of zero or more state orders (with no character that is not part of a state order in that string). For example, TX38-CA132+Ms6-nY290-UT006+ is an order string consisting of five state orders; TX38- CA132+ is not an order string, because the space character between the - and the C is not part of any state order. The empty string is an order string consisting of zero state orders.

These are the semantics of an order string: An order string represents a collection of orders, one for each state order in that string. Each state order represents one order of masks for that state; the digits in the state order represent the number of cases of masks for that order. For example, TX38- represents an order from the state TX for 38 cases of masks, and that order has not yet been filled.

### Your task

Your assignment is essentially to take an order string and an order status, and compute the total number of cases of masks for the orders in that order string that have that status. For example, for the order string TX38-CA132+Ms6-nY290-UT006+MS8+CA15+ the total for the status + is 161 (132 for CA plus 6 for UT plus 8 for MA plus another 15 for CA); for the status - it's 334 (38 for TX plus 6 for MS plus 290 for NY).

For this project, you will implement the following two functions, using the exact function names, parameter types, and return types shown in this specification. (The parameter names may be different if you wish.)

bool hasValidSyntax(string orders)

This function returns true if its parameter is an order string (i.e., it meets the definition above), or false otherwise.

int countCases(string orders, char status, int& caseCount)

If the parameter orders is not an order string (i.e., it does not meet the definition above), this function returns 1. Iforders is an order string in which at least one state order specifies zero cases of masks (e.g., GA0+), this function returns 2. If the parameter status is not a + or -, this function returns 3. (If more than one of these situations occur, return one of those occurring situations' return value, your choice which one.) If any of the preceding situations occur, caseCount must be left unchanged. If none of those situations occurs, then the function returns 0 after settingcaseCount to the total number of cases of masks for the state orders in orders that have the status indicated by thestatus parameter.

These are the only two functions you are required to write. (Hint: countCases may well call hasValidSyntax.) Your solution may use functions in addition to these two if you wish. While we won't test those additional functions separately, using them may help you structure your program more readably.

Of course, to test the functions you write, you'll want to write a main routine that calls your functions. During the course of developing your solution, you might change that main routine many times. As long as your main routine compiles correctly when you turn in your solution, it doesn't matter what it does, since we will rename it to something harmless and never call it (because we will supply our own main routine to thoroughly test your functions).

### Programming Guidelines

Your functions must not use any global variables whose values may be changed during execution (so global constants are allowed).

When you turn in your solution, neither of the two required functions, nor any functions you write that they call, may read any input from cin or write any output to cout. (Of course, during development, you may have them write whatever you like to help you debug.) If you want to print things out for debugging purposes, write to cerr instead of cout.  cerr is the standard error destination; items written to it by default go to the screen. When we test your program, we will cause everything written to cerr to be discarded instead — we will never see that output, so you may leave those debugging output statements in your program if you wish.

The correctness of your program must not depend on undefined program behavior. Your program must never access out of range positions in a string. Your program must not, for example, assume anything about n's value at the point indicated, or even whether or not the program crashes:

int main()

{

string s = "Hello";

int n; // n is uninitialized

s.at(5\*n/n) = '!'; // undefined behavior!

…

Be sure that your program builds successfully, and try to ensure that your functions do something reasonable for at least a few test cases under both g++ with Linux and either Visual C++ or clang++. That way, you can get some partial credit for a solution that does not meet the entire specification.

If you wish, you may use this [isValidUppercaseStateCode.txt](https://web.cs.ucla.edu/classes/spring20/cs31/Projects/3/isValidUppercaseStateCode.txt) function as part of your solution. (We can't imagine why you would not want to use it, since it does some of the work of valididating a supposed state code.)

You do not need to know anything about arrays to write this program. You may use arrays if you wish, but the most straightforward solutions to this project actually don't use arrays.

There are a number of ways you might write your main routine to test your functions. One way is to interactively accept test strings:

int main()

{

for (;;)

{

cout << "Enter order string: ";

string os;

getline(cin, os);

if (os == "quit")

break;

cout << "hasValidSyntax returns ";

if (hasValidSyntax(os))

cout << "true";

else

cout << "false";

cout << endl;

}

}

While this is flexible, you run the risk of not being able to reproduce all your test cases if you make a change to your code and want to test that you didn't break anything that used to work.

Another way is to hard-code various tests and report which ones the program passes:

int main()

{

if (hasValidSyntax("TX38-CA132+"))

cout << "Passed test 1: hasValidSyntax(\"TX38-CA132+\")" << endl;

if (!hasValidSyntax("MX38-CA132+"))

cout << "Passed test 2: !hasValidSyntax(\"MX38-CA132+\")" << endl;

int cases;

cases = -999; // so we can detect whether countCases sets cases

if (countCases("TX38-CA132+Ms6-nY290-UT006+MS8+CA15+", '+', cases) == 0 && cases == 161)

cout << "Passed test 3: countCases(\"TX38-CA132+Ms6-nY290-UT006+MS8+CA15+", '+', cases)" << endl;

cases = -999; // so we can detect whether countCases leaves cases unchanged

if (countCases("TX38-CA132+", '%', cases) == 3 && cases == -999)

cout << "Passed test 4: countCases(\"TX38-CA132+\", '%', cases)" << endl;

…

This can get rather tedious. Fortunately, the library has a facility to make this easier: assert. If you include the header<cassert>, you can call assert in the following manner:

assert(some boolean expression);

During execution, if the expression is true, nothing happens and execution continues normally; if it is false, a diagnostic message is written telling you the text and location of the failed assertion, and the program is terminated. Using assert, we can write the tests above more easily:

#include <iostream>

#include <string>

#include <cassert>

using namespace std;

bool hasValidSyntax(string orders)

{

… Your code goes here …

}

int countCases(string orders, char status, int& caseCount)

{

… Your code goes here …

}

int main()

{

assert(hasValidSyntax("TX38-CA132+"));

assert(!hasValidSyntax("MX38-CA132+"));

int cases;

cases = -999; // so we can detect whether countCases sets cases

assert(countCases("TX38-CA132+Ms6-nY290-UT006+MS8+CA15+", '+', cases) == 0 && cases == 161);

cases = -999; // so we can detect whether countCases leaves cases unchanged

assert(countCases("TX38-CA132+", '%', cases) == 3 && cases == -999);

…

cout << "All tests succeeded" << endl;

}

The reason for writing one line of output at the end is to ensure that you can distinguish the situation of all tests succeeding from the case where one function you're testing silently crashes the program.

### What to turn in

What you will turn in for this assignment is a zip file containing these two files and nothing more:

1. A text file named **orders.cpp** that contains the source code for your C++ program. Your source code should have helpful comments that tell the purpose of the major program segments and explain any tricky code. The file must be a complete C++ program that can be built and run, so it must contain appropriate #include lines, a main routine, and any additional functions you may have chosen to write.
2. A file named **report.docx** or **report.doc** (in Microsoft Word format) or **report.txt** (an ordinary text file) that contains:
   1. A brief description of notable obstacles you overcame.
   2. A description of the design of your program. You should use [pseudocode](https://web.cs.ucla.edu/classes/spring20/cs31/pseudocode.html) in this description where it clarifies the presentation.
   3. A list of the test data that could be used to thoroughly test your program, along with the reason for each test. You don't have to include the results of the tests. Notice that most of this portion of your report can be written just after reading the requirements in this specification, before you even start designing your program.

By May 6, there will be a link on the class web page that will enable you to turn in your zip file electronically. Turn in the file by the due time above. Give yourself enough time to be sure you can turn something in. There's a lot to be said for turning in a preliminary version of your program and report early (You can always overwrite it with a later submission). That way you have something submitted in case there's a problem later.

## Programming Assignment 4 Array? Hooray!

**Time due: 11:00 PM Thursday, May 14**

### Part 1

Go through the following sections of the class zyBook, doing the Participation Activities and Challenge Activities. We will be looking at whether you have ever successfully completed them; it does not matter how many attempts you make before a successful completion (or how many attempts you make after a successful completion if you want to experiment).

* 5.16 through 5.22
* 6.1
* 6.3 through 6.5

### Part 2

Implement the firstDifference and isContainedIn functions in the following program:

#include <iostream>

#include <string>

#include <cassert>

using namespace std;

int firstDifference(const string a1[], int n1, const string a2[], int n2)

{

// This function considers only the first n1 elements of the array a1 and the

// first n2 elements of the array a2. It returns the position of the first

// corresponding elements of a1 and a2 that are not equal. If the arrays are

// equal up to the point where one or both of them runs out, then return

// whichever value of n1 or n2 is less than or equal to the other. If n1 is

// 0, that means a1 runs out immediately; similarly, for n2 and a2. If n1 or

// n2 is negative, act as if it were 0.

Replace this line with your implementation

}

bool isContainedIn(const string a1[], int n1, const string a2[], int n2)

{

// This function considers only the first n1 elements of the array a1 and the

// first n2 elements of the array a2. If all n1 elements of a1 appear as

// elements in a2, in the same order (though not necessarily consecutively),

// then return true. Return false if a2 does not so contain a1. Of course,

// a sequence of 0 elements is contained in every sequence, even a sequence

// of 0 elements. If n1 or n2 is negative, act as if it were 0.

Replace this line with your implementation

}

int main()

{

// Here are some tests that demonstrate what these functions do. You may add

// more if you wish.

string colors[6] = { "pink", "blue", "white", "red", "black", "blue" };

string hues[5] = { "pink", "blue", "white", "black", "green" };

string tint[5] = { "yellow", "blue", "white", "black", "green" };

assert(firstDifference(colors, 6, hues, 5) == 3);

assert(firstDifference(colors, 4, hues, 5) == 3);

assert(firstDifference(colors, 2, hues, 5) == 2); // equal up to where a1 runs out

assert(firstDifference(colors, 2, hues, 2) == 2); // equal up to where both run out

assert(firstDifference(colors, 5, hues, 1) == 1); // equal up to where a2 runs out

assert(firstDifference(colors, 5, hues, 0) == 0); // equal up to where a2 runs out

assert(firstDifference(colors, 5, tint, 0) == 0); // equal up to where a2 runs out

string little1[4] = { "blue", "white", "black", "blue" };

assert(isContainedIn(little1, 4, colors, 6));

string little2[2] = { "red", "white" };

assert( ! isContainedIn(little2, 2, colors, 6));

string little3[3] = { "blue", "blue", "red" };

assert( ! isContainedIn(little3, 3, colors, 6));

assert(isContainedIn(hues, 3, colors, 3));

assert( ! isContainedIn(hues, 3, colors, 2));

assert(isContainedIn(hues, 0, tint, 5));

assert(isContainedIn(hues, 0, tint, 0));

cout << "All tests succeeded" << endl;

}

The source file named array.cpp that you turn in will contain the entire program above. You can actually have the main routine do whatever you want, because we will rename it to something harmless, never call it, and append our own main routine to your file. Our main routine will thoroughly test your functions. You'll probably want your main routine to do the same.

The program you turn in must build successfully, and during execution, no function (other than main) may read anything from cin or write anything to cout. If you want to print things out for debugging purposes, write to cerr instead of cout. When we test your program, we will cause everything written to cerr to be discarded instead — we will never see that output, so you may leave those debugging output statements in your program if you wish.

Although our test routine will call your functions passing values for n1 that are less than or equal to the number of elements in the array passed as a1, we will never pass a value for n1 than is greater than the number of elements in the array passed as a1 (and a similar promise goes for n2 and a2). This is because it is impossible for your function implementation to detect that we would be lying about how many elements it can examine without doing anything with undefined behavior. For example, in this situation, the function firstDifference can't possibly know that the caller is lying and telling the function that it can safely access more elements than the array was declared to have:

string animals[4] = { "rat", "pig", "goat", "duck" };

int i = firstDifference(animals, 100, animals, 4); // Bad call of function,

// but your firstDifference implementation doesn't have to check for

// this, because it is impossible to do so.

To make your life easier, whenever this specification talks about strings being equal or not equal, the case of letters matters. This means that you can simply use comparison operators like == or != to compare strings.

### Part 3

Implement the hasValidSyntax and countCases functions in the following program:

#include <iostream>

#include <cstring> // Notice this is NOT <string>; we need it because

// isValidUppercaseStateCode uses std::strstr

#include <cctype>

#include <cassert>

using namespace std;

bool isValidUppercaseStateCode(const char stateCode[]);

bool hasValidSyntax(const char orders[])

{

Replace this line with your implementation

}

int countCases(const char orders[], char status, int& caseCount)

{

Replace this line with your implementation

}

bool isValidUppercaseStateCode(const char stateCode[])

{

// In a declaration of an array with initialization, you can omit

// the number of elements and the compiler will count how many items

// are in the initializer and use that. For a C string, the count is

// the number of characters in the initialization string plus one more

// for the zero byte.

const char codes[] =

"AL.AK.AZ.AR.CA.CO.CT.DE.DC.FL.GA.GU.HI.ID.IL.IN.IA.KS."

"KY.LA.ME.MD.MA.MI.MN.MS.MO.MP.MT.NE.NV.NH.NJ.NM.NY.NC."

"ND.OH.OK.OR.PA.PR.RI.SC.SD.TN.TX.UT.VT.VA.WA.WV.WI.WY";

return (isupper(stateCode[0]) && isupper(stateCode[1]) &&

stateCode[2] == '\0' && strstr(codes, stateCode) != nullptr);

}

int main()

{

assert(hasValidSyntax("TX38-CA132+"));

assert(!hasValidSyntax("MX38-CA132+"));

int cases;

cases = -999; // so we can detect whether countCases sets cases

assert(countCases("TX38-CA132+Ms6-nY290-UT006+MS8+CA15+", '+', cases) == 0 && cases == 161);

cases = -999; // so we can detect whether countCases leaves cases unchanged

assert(countCases("TX38-CA132+", '%', cases) == 3 && cases == -999);

cout << "All tests succeeded" << endl;

}

The hasValidSyntax and countCases functions must behave as required for Project 3, except that instead of accepting and working with C++ strings, they must accept and work with C strings. You are encouraged to adapt [this correct solution to Project 3](https://web.cs.ucla.edu/classes/spring20/cs31/Projects/4/p3orders.cpp) (which is itself adapted from the solution we posted for Project 3). You'll find that the structure and most of the code you need for this part will be identical to what's in that solution.

Your program must **not** include the header <string> and must **not** declare any variables of type string; you must **not** use any C++ strings. The source file named p4orders.cpp that you turn in will contain the entire program above. You can actually have the main routine do whatever you want, because we will rename it to something harmless, never call it, and append our own main routine to your file.

### What to turn in

You won't turn anything in through the CS 31 web site for Part 1; the zyBook system notes your successful completion of the PAs and CAs. For Parts 2 and 3, turn in a zip file containing these two files and nothing more:

1. A text file named **array.cpp** that contains the source code for your C++ program for Part 2.
2. A text file named **p4orders.cpp** that contains the source code for your C++ program for Part 3.

How nice! You don't have to turn in a report file.

By Wednesday, May 13, there will be links on the class webpage that will enable you to turn in your zip file electronically. Turn in the file by the due time above.

## Programming Assignment 5 Type O

**Time due: 11:00 PM Saturday, May 23**

### Part 1

Go through the following sections of the class zyBook, doing the Participation Activities and Challenge Activities. We will be looking at whether you have ever successfully completed them; it does not matter how many attempts you make before a successful completion (or how many attempts you make after a successful completion if you want to experiment).

* 7.2 and 7.3
* 8.1 and 8.2

Also, if you did not complete part 1 of Project 4, this would be the time to do so if you want credit for it.

### Part 2

Implement the scoreTypo function in the following program:

#include <iostream>

#include <string>

#include <cassert>

using namespace std;

int scoreTypo(const string dictionary[], int n, string word)

{

Replace this line with your implementation

}

int main()

{

// Here are some tests. You may add more if you wish.

string dict1[6] = { "february", "pointer", "country", "forty", "conversation", "minuscule" };

assert(scoreTypo(dict1, 0, "forty") == -1);

assert(scoreTypo(dict1, 6, "forty") == 0);

assert(scoreTypo(dict1, 6, "fourty") == 1);

assert(scoreTypo(dict1, 6, "febuary") == 1);

assert(scoreTypo(dict1, 6, "miniscule") == 1);

assert(scoreTypo(dict1, 6, "poitner") == 1);

assert(scoreTypo(dict1, 6, "conservation") == 2);

cout << "All tests succeeded" << endl;

}

The scoreTypo function considers only the first n elements of the array passed to dictionary. You may assume that word and that every string in dictionary consists of at least one lower case letter and contains no character that is not a lower case letter. (So we promise that we will not pass strings like "", "eBay", "half-wit", "be at", or "hen3ry" to this function.) If nis not positive, scoreTypo returns −1; otherwise, it returns the smallest typo score among all the typo scores between wordand each string in dictionary.

Let's define a simple typo of a string as one of the following:

* adding exactly one character to the string (e.g., adding one character to forty to produce fo**u**rty)
* removing exactly one character from the string (e.g., removing a character from feb**r**uary to produce febuary)
* replacing exactly one character in the string (e.g., replacing a character in min**u**scule to produce min**i**scule)
* swapping two adjacent characters in the string (e.g., swapping two adjacent characters of poi**nt**er to produce poi**tn**er)

The typo score between two strings is defined to be

* 0 if the strings are identical
* 1 if exactly one simple typo would transform one string to the other
* 2 if neither of the above two conditions hold

For example, the typo score between country and fourty is 2, while the typo score between forty and fourty is 1. In the example program above, 1 is the smallest typo score among all the typo scores between fourty and each word in dict1.

The source file named typo.cpp that you turn in will contain the entire program above. You can actually have the main routine do whatever you want, because we will rename it to something harmless, never call it, and append our own main routine to your file. Our main routine will thoroughly test your function. You'll probably want your main routine to do the same. If you wish, you may add helper functions that your implementation of scoreTypo calls.

The program you turn in must build successfully, and during execution, no function (other than main) may read anything from cin or write anything to cout. If you want to print things out for debugging purposes, write to cerr instead of cout. When we test your program, we will cause everything written to cerr to be discarded instead — we will never see that output, so you may leave those debugging output statements in your program if you wish.

Although our test routine will call your function passing values for n that are less than or equal to the number of elements in the array passed as dictionary, we will never pass a value for n than is greater than the number of elements in the array passed as dictionary.

### What to turn in

You won't turn anything in through the CS 31 web site for Part 1; the zyBook system notes your successful completion of the PAs and CAs. For Part 2, turn in a zip file containing this one file and nothing more:

1. A text file named **typo.cpp** that contains the source code for your C++ program for Part 2.

By Friday, May 22, there will be links on the class webpage that will enable you to turn in your zip file electronically. Turn in the file by the due time above.

## Project 6 Get the Point?

**Time due: 11:00 PM Wednesday, May 27**

This project is designed to help you master pointers. To that end, you'll get the most out of it by working through the problems by hand. Only after that should you resort to running the programs (and stepping through them with the debugger) to check your understanding. Remember, on the final exam you'll have to be able to do problems like this by hand.

This "project" is more like a homework. There are five problems. In problems that ask you to change code, make the few changes necessary to fix the code without changing its overall approach. For example, don't fix the program in problem 1a by changing it to

int main()

{

cout << " 30 20 10" << endl;

}

1. The subparts to this problem involve errors in the use of pointers.
   1. This program is supposed to write **30 20 10**, but it doesn't. Find all of the bugs and show a fixed version of the program:
   2. int main()
   3. {
   4. int arr[3] = { 5, 10, 15 };
   5. int\* ptr = arr;
   6. \*ptr = 10; // set arr[0] to 10
   7. \*ptr + 1 = 20; // set arr[1] to 20
   8. ptr += 2;
   9. ptr[0] = 30; // set arr[2] to 30
   10. while (ptr >= arr)
   11. {
   12. ptr--;
   13. cout << ' ' << \*ptr; // print values
   14. }
   15. cout << endl;
   16. }
   17. The findDisorder function is supposed to find the first item in an array that is less than the element preceding it, and set the p parameter to point to that item, so the caller can know the location of that item. Explain why this function won't do that, and show how to fix it. Your fix must be to the function only; you must not change the the main routine below in any way, yet as a result of your fixing the function, the main routine below must work correctly.
   18. void findDisorder(int arr[], int n, int\* p)
   19. {
   20. for (int k = 1; k < n; k++)
   21. {
   22. if (arr[k] < arr[k-1])
   23. {
   24. p = arr + k;
   25. return;
   26. }
   27. }
   28. p = nullptr;
   29. }
   31. int main()
   32. {
   33. int nums[6] = { 10, 20, 20, 40, 30, 50 };
   34. int\* ptr;
   35. findDisorder(nums, 6, ptr);
   36. if (ptr == nullptr)
   37. cout << "The array is ordered" << endl;
   38. else
   39. {
   40. cout << "The disorder is at address " << ptr << endl;
   41. cout << "It's at position " << ptr - nums << endl;
   42. cout << "The item's value is " << \*ptr << endl;
   43. }
   44. }
   45. The hypotenuse function is correct, but the main function has a problem. Explain why it may not work, and show a way to fix it. Your fix must be to the main function only; you must not change the hypotenuse function in any way.
   46. #include <iostream>
   47. #include <cmath>
   48. using namespace std;
   49. void hypotenuse(double leg1, double leg2, double\* resultPtr)
   50. {
   51. \*resultPtr = sqrt(leg1\*leg1 + leg2\*leg2);
   52. }
   53. int main()
   54. {
   55. double\* p;
   56. hypotenuse(1.5, 2.0, p);
   57. cout << "The hypotenuse is " << \*p << endl;
   58. }
   59. The match function is supposed to return true if and only if its two C string arguments have exactly same text. Explain what the problems with the implementation of the function are, and show a way to fix them.
   60. // return true if two C strings are equal
   61. bool match(const char str1[], const char str2[])
   62. {
   63. while (str1 != 0 && str2 != 0) // zero bytes at ends
   64. {
   65. if (str1 != str2) // compare corresponding characters
   66. return false;
   67. str1++; // advance to the next character
   68. str2++;
   69. }
   70. return str1 == str2; // both ended at same time?
   71. }
   72. int main()
   73. {
   74. char a[10] = "pointy";
   75. char b[10] = "pointless";
   76. if (match(a,b))
   77. cout << "They're the same!\n";
   78. }
   79. This program is supposed to write 1 4 9 16 25 36 49 64 81 100 , but it probably does not. What is the program doing that is incorrect? (We're not asking you explain why the incorrect action leads to the particular outcome it does, and we're not asking you to propose a fix to the problem.)
   80. #include <iostream>
   81. using namespace std;
   82. int\* computeSquares(int& n)
   83. {
   84. int arr[10];
   85. n = 10;
   86. for (int k = 0; k < n; k++)
   87. arr[k] = (k+1) \* (k+1);
   88. return arr;
   89. }
   90. void f()
   91. {
   92. int junk[100];
   93. for (int k = 0; k < 100; k++)
   94. junk[k] = 123400000 + k;
   95. }
   96. int main()
   97. {
   98. int m;
   99. int\* ptr = computeSquares(m);
   100. f();
   101. for (int i = 0; i < m; i++)
   102. cout << ptr[i] << ' ';
   103. }
2. For each of the following parts, write a single C++ statement that performs the indicated task. For each part, assume that all previous statements have been executed (e.g., when doing part e, assume the statements you wrote for parts a through d have been executed). For each part, do not use any variable names or string literals not mentioned in that part (e.g., if the part doesn't mention fp or "tuna", do not use fp or "tuna" in your answer).
   1. Declare a pointer variable named fp that can point to a variable of type string.
   2. Declare fish to be a 5-element array of strings.
   3. Make the fp variable point to the last element of fish.
   4. Make the string pointed to by fp equal to "yellowtail", using the \* operator.
   5. Without using the fp pointer, and without using square brackets, set the fourth element (i.e., the one at position 3) of the fish array to have the value "salmon".
   6. Move the fp pointer back by three strings.
   7. Using square brackets, but without using the name fish, set the third element (i.e., the one at position 2) of the fish array to have the value "basa". (You may use fp.)
   8. Without using the \* operator or the name fish, but using square brackets, set the string pointed to by fp to have the value "sole".
   9. Using the == operator in the initialization expression, declare a bool variable named d and initialize it with an expression that evaluates to true if fp points to the string at the start of the fish array, and to false otherwise.
   10. Using the \* operator in the initialization expression, but no square brackets, declare a bool variable named b and initialize it to true if the string pointed to by fp is equal to the string immediately following the string pointed to by fp, and false otherwise.
   11. Rewrite the following function so that it returns the same result, but does not increment the variable ptr. Your new program must not use any square brackets, but must use an integer variable to visit each double in the array. You may eliminate any unneeded variable.
   12. double computeAverage(const double\* scores, int nScores)
   13. {
   14. const double\* ptr = scores;
   15. double tot = 0;
   16. while (ptr != scores + nScores)
   17. {
   18. tot += \*ptr;
   19. ptr++;
   20. }
   21. return tot/nScores;
   22. }
   23. Rewrite the following function so that it does not use any square brackets (not even in the parameter declarations) but does use the integer variable k. Do not use any of the <cstring> functions such as strlen, strcpy, etc.
   24. // This function searches through str for the character chr.
   25. // If the chr is found, it returns a pointer into str where
   26. // the character was first found, otherwise nullptr (not found).
   27. const char\* findTheChar(const char str[], char chr)
   28. {
   29. for (int k = 0; str[k] != 0; k++)
   30. if (str[k] == chr)
   31. return &str[k];
   32. return nullptr;
   33. }
   34. Now rewrite the function shown in part b so that it uses neither square brackets nor any integer variables. Your new function must not use any local variables other than the parameters. Do not use any of the <cstring>functions such as strlen, strcpy, etc.
3. What does the following program print and why? Be sure to explain why each line of output prints the way it does to get full credit.
4. #include <iostream>
5. using namespace std;
6. int\* minimart(int\* a, int\* b)
7. {
8. if (\*a < \*b)
9. return a;
10. else
11. return b;
12. }
13. void swap1(int\* a, int \*b)
14. {
15. int\* temp = a;
16. a = b;
17. b = temp;
18. }
19. void swap2(int\* a, int \*b)
20. {
21. int temp = \*a;
22. \*a = \*b;
23. \*b = temp;
24. }
25. int main()
26. {
27. int array[6] = { 5, 3, 4, 17, 22, 19 };
28. int\* ptr = minimart(array, &array[2]);
29. ptr[1] = 9;
30. ptr += 2;
31. \*ptr = -1;
32. \*(array+1) = 79;
33. cout << "diff=" << &array[5] - ptr << endl;
34. swap1(&array[0], &array[1]);
35. swap2(array, &array[2]);
36. for (int i = 0; i < 6; i++)
37. cout << array[i] << endl;
38. }
39. Write a function named deleteG that accepts one character pointer as a parameter and returns no value. The parameter is a C string. This function must remove all of the upper and lower case 'g' letters from the string. The resulting string must be a valid C string.

Your function must declare no more than one local variable in addition to the parameter; that additional variable must be of a pointer type. Your function must not use any square brackets. Do not use any of the <cstring> functions such as strlen, strcpy, etc.

int main()

{

char msg[100] = "I recall the glass gate next to Gus in Lagos, near the gold bridge.";

deleteG(msg);

cout << msg; // prints I recall the lass ate next to us in Laos, near the old bride.

}

Prepare your solutions to these homework problems as a single Word document named **hw.docx** or **hw.doc**, or a plain text file named **hw.txt**. Put that file in a zip file. By Tuesday, May 26, there will be a link on the class webpage that will enable you to turn in your zip file.

**Programming Assignment 7  
Spider Pit**

**Time due: 11:00 PM Thursday, June 4**

While doing field work in the rainforest, you slipped and fell into a deep pit infested with giant venomous spiders, whose bite will kill you. Fortunately, you are carrying a bag full of rocks. However, the only way to throw a rock hard and accurately enough to kill a spider is from above the spider, so you have to jump over it when throwing your rock.

That's the scenario for a new video game under development. Your assignment is to complete the prototype that uses character-based graphics.

If you execute [this Windows program](https://web.cs.ucla.edu/classes/spring20/cs31/Projects/7/spiders.exe) or [this Mac program](https://web.cs.ucla.edu/classes/spring20/cs31/Projects/7/spidersmac.zip) or [this Linux program](https://web.cs.ucla.edu/classes/spring20/cs31/Projects/7/spiders.linux), you will see the player (indicated by @) in a rectangular pit filled with spiders (usually indicated by S). At each turn, the user will select an action for the player to take. The player will take the action, and then each spider will move one step in a random direction. If a spider lands on the grid point occupied by the player, the player dies.

This smaller [Windows version](https://web.cs.ucla.edu/classes/spring20/cs31/Projects/7/minispiders.exe) or [Mac version](https://web.cs.ucla.edu/classes/spring20/cs31/Projects/7/minispidersmac.zip) or [Linux version](https://web.cs.ucla.edu/classes/spring20/cs31/Projects/7/minispiders.linux) of the game may help you see the operation of the game more clearly.

At each turn the player may take one of these actions:

1. Stand. In this case, the player does not move.
2. Move one step up, down, left, or right, to an adjacent vacant grid point. If the player attempts to move into the wall of the pit (e.g., down, when on the bottom row), the result is the same as standing.
3. Move by jumping up, down, left, or right, over an adjacent spider, landing at the grid point on the other side of the spider. This kills the spider. If more than one spider is on the grid point that is jumped over, only one spider there dies. If the grid point on the other side of the adjacent spider has a spider in it, the player dies. If the adjacent spider is on a grid point next to the wall of the pit, the player kills the spider, but hits the wall and falls onto the spot where the now-dead spider is; the fall doesn't hurt the player, but if there's at least one other spider on that spot, the player dies.

The game allows the user to select the player's action: u/d/l/r for movement, and just hitting enter for standing. The user may also type q to prematurely quit the game.

When it's the spiders' turn, each spider picks a random direction (up, down, left, or right) with equal probability. The spider moves one step in that direction if it can; if the spider attempts to move into the wall of the pit, however, (e.g., down, when on the bottom row), it does not move. More than one spider may occupy the same grid point; in that case, instead of S, the display will show a digit character indicating the number of spiders at that point (where 9 indicates 9 or more). If after the spiders move, a spider occupies the same grid point as the player, the player dies.

Your assignment is to complete [this C++ program skeleton](https://web.cs.ucla.edu/classes/spring20/cs31/Projects/7/spiders-skeleton.cpp) to produce a program that implements the described behavior. (We've indicated where you have work to do by comments containing the text TODO; remove those comments as you finish each thing you have to do.) The program skeleton you are to flesh out defines four classes that represent the four kinds of objects this program works with: Game, Pit, Spider, and Player. Details of the interface to these classes are in the program skeleton, but here are the essential responsibilities of each class:

Game

* To create a Game, you specify a number of rows and columns and the number of spiders to start with. The Game object creates an appropriately sized Pit and populates it with the Player and the Spiders.
* A game may be played.

Pit

* When a Pit object of a particular size is created, it has no positions occupied by spiders or the player. In the Pit coordinate system, row 1, column 1 is the upper-left-most position that can be occupied by a spider or the player. (If a Pit were created with 9 rows and 10 columns, then the lower-right-most position that could be occupied would be row 9, column 10.)
* You may tell a Pit object to create or destroy a Spider at a particular position.
* You may tell a Pit object to create a Player at a particular position.
* You may tell a Pit object to have all the spiders in it make their move.
* You may ask a Pit object its size, how many spiders are at a particular position, and how many spiders altogether are in the Pit.
* You may ask a Pit object for access to its player.
* A Pit object may be displayed on the screen, showing the locations of the spiders and player, along with other status information.

Player

* A Player is created at some position (using the Pit coordinate system, where row 1, column 1 is the upper-left-most position, etc.).
* You may tell a Player to stand or to move in a particular direction.
* You may tell a Player it has died.
* You may ask a Player for its position, alive/dead status, and age. (The age is the count of how many turns the player has survived.)

Spider

* A Spider is created at some position (using the Pit coordinate system, where row 1, column 1 is the upper-left-most position, etc.).
* You may tell a Spider to move.
* You may ask a Spider object for its position.

The skeleton program you are to complete has all of the class definitions and implementations in one source file, which is awkward. Since we haven't yet learned about separate compilation, we'll have to live with it.

Complete the implementation in accordance with the description of the game. You are allowed to make whatever changes you want to the *private* parts of the classes: You may add or remove private data members or private member functions, or change their types. You must *not* make any deletions, additions, or changes to the *public* interface of any of these classes — we're depending on them staying the same so that we can test your programs. You can, of course, make changes to the *implementations* of public member functions, since the callers of the function wouldn't have to change any of the code they write to call the function. You must **not** declare any public data members, nor use any global variables whose values may change during execution (so global constants are OK). You may add additional functions that are not members of any class. The word friend must not appear in your program.

Any member functions you implement must never put an object into an invalid state, one that will cause a problem later on. (For example, bad things could come from placing a spider outside the pit.) Any function that has a reasonable way of indicating failure through its return value should do so. Constructors pose a special difficulty because they can't return a value. If a constructor can't do its job, we have it write an error message and exit the program with failure by calling exit(1);. (We haven't learned about throwing an exception to signal constructor failure.)

What you will turn in for this assignment is a zip file containing this one file and nothing more:

1. A text file named **spiders.cpp** that contains the source code for the completed C++ program. This program must build successfully using both g31 and either Visual C++ or clang++, and its correctness must not depend on undefined program behavior. Your program must not leak memory: Any object dynamically allocated during the execution of your program must be deleted (once only, of course) by the time your main routine returns normally.

Notice that you do not have to turn in a report describing the design of the program and your test cases.

By Wednesday, June 3, there will be a link on the class web page that will enable you to turn in your zip file electronically. Turn in the file by the due time above.