CS 111 - Homework 12

Deadline

11:59 pm on **Friday, December 13, 2024**

Purpose

To practice with functions whose arguments include an array argument and its size, file output and file input, and `for` loops and C++ shortcut operators.

How to submit

You complete **Problems 1-3** on the course Canvas site (short-answer questions on various C++-related topics), so that you can see if you are on the right track.

Then, you will submit your work for **Problems 4** onward, in your files **`111hw12.cpp`** and **`111hw12-out.txt`**, along with the two file **`prob6-1.txt`** and **`prob6-2.txt`** that its program creates, on the course Canvas site.

Turn in versions of your files **early** and **often**!

- Each time you submit a version of your **`111hw12.cpp`**, IF that version currently compiles, also submit a copy of the example output from running that latest version in file **`111hw12-out.txt`**.
  - Be careful that each submitted **`111hw12-out.txt`** was created by running the compiled version of the **`111hw12.cpp`** file submitted along with it.
- Likewise, IF that version currently successfully creates the files **`prob6-1.txt`** and **`prob6-2.txt`**, submit copies of those as well.
  - And, be careful that each submitted **`prob6-1.txt`** and **`prob6-2.txt`**,were created by running the compiled version of the **`111hw12.cpp`** file submitted along with it.

Important notes

- **NOTE:** if you are just adding statements **to a `main` function**, the usual design recipe steps are **NOT** required. (They are, of course, required for all **(non-`main`) functions** that you design/define.)
- IF you would like: FEEL FREE to include additional `cout`s of `endl` or spacing or headings between testing calls of different problems if you would like to have more-readable program output!
- Remember that, **for each function WITH side-effects**:
  - You ALSO need to **DESCRIBE those side-effects** in its **purpose** statement, in a "has the side-effects of..." clause, **along with** describing what it "expects..." and what it "returns ...".
  - You ALSO need to **DESCRIBE the expected side-effects** that should be seen as a result of each of its function tests, BOTH along with its `bool` test expressions in the comment after its purpose statement AND when running those tests in its testing `main` function.
  - (See how the tests for function `cheer` are written in the Week 13 Lecture 1 posted examples, and how the tests are written from function `vertical` in the Week 13 Lecture 2 posted examples.)
- Be careful to follow class style standards, including required class indentation.
  - When in doubt, ASK, and/or follow the style you see in the posted class examples.
- You are still expected to follow the Design Recipe for all (non-`main`) **functions** that you design/define.

  - Remember the C++ "graphic design recipe helper" posted on the course Canvas site and on the public course web site, "translating" the design recipe steps into C++ syntax.
  - Remember, you will receive **significant** credit for the signature, purpose, header, and tests/test expressions portions of your functions.
  - Typically you'll get at least half-credit for a correct signature, purpose, header, and tests/test expressions, even if your function body is not correct.
  - (and, you'll **lose at least half-credit** if you omit these or do them poorly, even if your function body is correct).

- Be especially careful to include at least two tests/test expressions for every function, including at least one specific test/test expression for each "kind"/category of data, and (when there *are* boundaries) for boundaries between data. You can lose credit for not doing so.

  And, remember that tests for non-`void` functions should be:

  - written as `bool` expressions within a non-`main` function's opening comment, after its purpose statement, **AND**
  - written within parentheses `( )` within a `cout` in the testing `main` function.

- Please let me know if you have any questions or concerns about the above requirements.

Problem 1 - 9 points

Problem 1 is correctly answering the "HW 12 - Problem 1 - Short-answer questions on array parameters" on the course Canvas site.

Problem 2 - 11 points

Problem 2 is correctly answering the "HW 12 - Problem 2 - Short-answer questions on file input/output" on the course Canvas site.

Problem 3 - 9 points

Problem 3 is correctly answering the "HW 12 - Problem 3 - Short-answer questions on for loops and C++ shortcut operators" on the course Canvas site.

- Note: we will be discussing these in class on Tuesday, December 10, and the Week 15 Readings links (now posted on Canvas and on the public course web site) has links to readings about these, if you want to complete this question before Tuesday December 10.

Homework Program Setup for Problems 4 onward

- **Copy** the contents of the file **`111template.cpp`**, posted on the course Canvas site and on the public course web site, into a file named **`111hw12.cpp`** within the CS50 IDE (at https://cs50.dev/).
- See the comment that has `by:` and `last modified: ?`
  - START that comment with: `CS 111 - HW 12`
  - Then put your name after `by:` , and today's date after `last modified:` .
  - For example:

```
/*---
    CS 111 - HW 12
    by: Your Name
    last modified: 2024-12-09
---*/
```

Problem 4 - function `count_names`

This problem's purpose is to provide practice with a function whose parameters include an array and its size along with another parameter, and does not happen to have side-effects.

In the "first `main.cpp` template" you pasted into your **`111hw12.cpp`**, find the comment:

```
/*--- PUT YOUR SIGNATURES, PURPOSES, TESTS, and FUNCTION DEFINITIONS HERE ---*/
```

AFTER this comment -- but **BEFORE** the function header for the function named `main` -- type a blank link, and then type the comment:

```
/*===
    Problem 4
===*/
```

Using the design recipe, design a function **`count_names`** that expects a name of interest, an array of names, and how many names are in that array, and returns how many times the given name of interest appears in the given array.

- Since **`count_names`** has an array argument, you'll also need to include declarations for the array arguments in your tests, as we did for Week 14's in-class example functions `sum_array` and `get_smallest`.

  That is, for EACH of its tests after the function's purpose statement:

  - give the declaration for an example argument array, and then a `bool` expression including an example call compared to what it should return

- Include at least three tests for `count_names`:
  - one in which the given name of interest does NOT happen to appear in the given array of names
  - one in which the given name of interest appears exactly ONCE in the given array of names
  - one in which the given name of interest appears MORE than once in the give array of names

Problem 5 - function `bar_chart`

The purpose of this problem is to write yet-another function whose parameters include an array and its size, but this time it happens to have side-effects.

After your function for Problem 4, type a blank link, and then type the comment:

```
/*===
    Problem 5
===*/
```

You are again going to make use of Week 13 Lab Exercise function **`starline`** in another problem's function.

Problem 5 - Step 1

Copy the opening comment with the signature, purpose, `bool` test expressions and side-effect descriptions, and the function definition for Week 13 Lab Exercise's function **`starline`**.

- Note: if you did not do the Week 13 Lab Exercise or you are not confident in your version of **`starline`**, you can e-mail me and ask for a version of **`starline`**.

Now that **`starline`** is in your **`111hw12.cpp`** file, it can be used by another function that follows it in this file.

Problem 5 - Onward!

Recall that **`starline`** expects the number of asterisks/stars to output, and it returns that number, but also has the side-effect of printing that many asterisks on one line to the screen (if <=0, no stars are output).

You could use this to create a kind of horizontal bar chart, calling **`starline`** for each of a set of values. And what is an array but a set of values?

Using the design recipe, design a C++ function **`bar_chart`** that expects an array of integers and its size, has the side-effect of printing to the screen a horizontal bar chart with the help of **`starline`**, printing a line of *'s the length of each array value, and returns the total number of asterisks printed in the entire chart.

- For full credit, this function must appropriately call and use **`starline`**

For example, for:

```
const int NUM_MSRS = 7;
int measures[NUM_MSRS] = {3, 1, 6, 2, 8, 4, 5};

bar_chart(measures, NUM_MSRS) == 29
```

...and has the side-effect of causing the following to be printed to the screen:

```
***
*
******
**
********
****
*****
```

- And, since **`bar_chart`** has side-effects, your tests for **`bar_chart`** after its purpose statement should INCLUDE a description of the expected side effects for each testing call.

  Since **`bar_chart`** has an array argument, you'll also need to include declarations for the array arguments in your tests, as we did for Week 14's in-class functions **`sum_array`** and **`get_smallest`**.

  So, for EACH of its tests after the function's purpose statement:

  – give the declaration for an example argument array, an example call, and what should be printed to the screen for that example call.

  And the running versions of those tests in your `main` function should print out a DESCRIPTION of what side-effects should be seen.

  That is, for EACH of its tests to be run in `main`:

  – it should first print a message saying that what follows should be a bar chart with rows of length <num>, <num> ... and <num>, followed by `true`,

  – and *then* put that example/test in its own separate `cout` statement, such that the result of that test will be printed on its own line.

Problem 6 - function `array_to_file`

The purpose of this problem is to write another function whose parameters include an array and its size, and which also happens to involve writing to a file.

(And, its writing to a file happens from a non-`main` function -- just to make it clear that file i/o does not happen ONLY from `main` functions!)

In your file **`111hw12.cpp`**:

- Near the top of your file **`111hw12.cpp`**, add a **`#include`** for the **`fstream`** standard library, so you can use it here:

```
#include <cstdlib>
#include <iostream>
#include <string>
#include <cmath>
#include <fstream>
using namespace std;
```

- Then, **after** your function for Problem 5, type a blank link, and then type the comment:

```
/*===
    Problem 6
===*/
```

- Using the design recipe, design a function **`array_to_file`** that:
  - expects an array of strings, its size, and the *name* of a file,
  - has the side-effect of writing the contents of that given array to a file with that given file name, one array value per line,
  - and **returns** the number of lines written to that file.
- In the part of your `main` that is testing **`array_to_file`**, you should:
  - call **`array_to_file`** at least **TWICE**,
  - one time writing an array to the file **`prob6-1.txt`**, and
  - one time writing a **different-sized array** to the file **`prob6-2.txt`**, and
  - each time including **`cout`** statements DESCRIBING what *should* be written to each of the files as its side-effect, followed by putting the example/test call in its own separate **`cout`** statement, such that the result of that test will be printed on its own line.

Problem 7 - function `vertical_file`

The purpose of this problem is to practice reading everything from a file and doing something with it, making use of a **`while`** loop **not** *controlled* by a counter variable.

(And, its reading from a file happens from a non-`main` function -- again, just to make it clear that file i/o does not happen ONLY from `main` functions!)

Problem 7 - fun fact: how to read every line from a file using a `while` loop

If you have:

- **declared** and **opened** an input file stream, for example **`my_fin`**,
- and you want to read every *line* from the file that stream is opened on into a `string` variable, for example

next_line,

- you can do so -- using a sufficiently-new C++ compiler -- using:

```
    while (getline(my_fin, next_line))
    {
        desired actions for each line probably using just-read next_line;
        ...
    }

    my_fin.close();
```

  - (this works because, for a sufficiently-new C++ compiler, function **getline** returns **true** if reading a line succeeds, and returns **false** if reading a line fails -- so this loop repeats until all the lines of a file have been read.)

After your function for Problem 6, type a blank link, and then type the comment:

```
/*===
    Problem 7
===*/
```

Remember function **vertical**, from Week 13 Lecture 2?

Problem 7 - Step 1

Copy the opening comment with the signature, purpose, `bool` test expressions and side-effect descriptions, and the function definition for Week 13 Lecture 2's function **vertical**.

Now that **vertical** is in your **111hw12.cpp** file, it can be used by another function that follows it in this file.

Problem 7 - Onward!

Then, using the design recipe, write a function **vertical_file** that expects the name of a file, has the side-effects of reading each line in that file, calling function **vertical** for the line read, and then printing a newline character to the screen, and returns the number of lines read from that file.

(Hint:

- Even though we don't need a count variable to *control* this loop, we *can* still have a count variable to *count* how many times we end up calling **vertical**.)

For example, if you had a little file `lookity.txt` that contains:

```
oh
to demo
whee!
```

then:

```
vertical_file("lookity.txt") == 3
```

and has the side effect of printing to the screen

```
o
h

t
o

d
e
```

```
m
o

w
h
e
e
!
```

- Do you see that Problem 6's output files **`prob6-1.txt`** and **`prob6-2.txt`** would work just fine as inputs to this function? AND they should be here in the right folder, able to be reasonably called.
- Your tests for **`vertical_file`** can call **`vertical_file`** on these two files **`prob6-1.txt`** and **`prob6-2.txt`**, then -- and, for each call, the `main` function should include `cout` statements DESCRIBING what *should* be printed to the screen as its side-effect.

Make sure you submit your files **`prob6-1.txt`** and **`prob6-2.txt`** as well as your **`111hw12.cpp`** and **`111hw12-out.txt`**.