CS 111 - Homework 11

Deadline

11:59 pm on **Friday, December 5, 2025**

Purpose

To practice with **`while`** statements, more interactive input, and a question-controlled loop.

How to submit

You complete **Problems 1-2** 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 3** onward, in your files **`111hw11.cpp`**, **`111hw11-out.txt`**, and **`111hw11-prob6.cpp`**, on the course Canvas site.

(So, NOTE that, THIS time, you again will be creating **TWO `.cpp`** files to turn in, for the remaining problems!)

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

- Each time you submit a version of your **`111hw11.cpp`**, IF that version currently compiles, also submit a copy of the example output from running that latest version in file **`111hw11-out.txt`**.
- Be careful that each submitted **`111hw11-out.txt`** was created by running the compiled version of the **`111hw11.cpp`** file submitted along with it.
- (You are **NOT** submitting a `111hw11-prob6-out.txt` file, for the same reason you were not asked to submit a file `lab12-out.txt` for the Week 12 Lab Exercise!).

Important notes - 14 points

- **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!
- NOW that you will be programming with **MORE side-effects**, note that, for each function that **HAS** 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 for function **`vertical`** are written 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 - 12 points

Problem 1 is correctly answering the "HW 11 - Problem 1 - Short-answer questions on recognizing different C++ statement types" on the course Canvas site.

Problem 2 - 8 points

Problem 2 is correctly answering the "HW 11 - Problem 2 - Short-answer questions focusing on **`cout`**" on the course Canvas site.

Homework Program Setup for Problems 3 onward

For **EACH** of the **TWO** programs involved in this homework:

- **Copy** the contents of the **`111template.cpp`**, posted on the course Canvas site and on the public course web site, into a file within the CS50 IDE (at https:/cs50.dev/) named as specified in Problem 3 and Problem 6.
- See the comment that has `by:` and `last modified:` ?
  - START that comment with: `CS 111 - HW 11`
  - Then put your name after `by:` , and today's date after `last modified:` .
  - For example:

```
/*---
    CS 111 - HW 11
    by: Your Name
    last modified: 2025-12-01
---*/
```

Problem 3 - function `count_blanks`

Problems 3 through 5 will all be in a single file named **`111hw11.cpp`**.

In the "first `main.cpp` template" you pasted into your **`111hw11.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 3
===*/
```

(You can now delete the `"...PUT YOUR SIGNATURES,..."` comment if you wish, or leave it -- your choice!)

This problem's purpose is to provide more practice with the C++ **`while`** statement.

Using the design recipe, design a function **`count_blanks`** that expects a string, and returns how many blanks are in that string.

- Include at least four different tests for **`count_blanks:`**
  - one in which the given string is EMPTY (has length 0)
  - one in which the given string contains NO blanks (it just contains non-blank characters)
  - one in which the given string contains exactly ONE blank (as well as non-blank characters)
  - one in which the given string contains MORE than one blank (as well as non-blank characters)
- For full credit, **`count_blanks`** must also:
  - appropriately use a **`while`** loop
- Remember to include a **signature**, **purpose**, **function header**, **tests**, and then completed **function body** for **`count_blanks`**.
- Be sure to include your tests BOTH in a comment after your purpose statement, AND in `main`, as we have done in class.
- IF you would like, you can also include one or more `cout` statements that include JUST an example call of your function **after** these tests, so that you see the value those call(s) return.

Problem 4 - function `sum_coins`

After your function for Problem 3, type a blank link, and then type the comment:

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

This problem's purpose is to provide still-more practice with the C++ **`while`** statement.

In Homework 10 - Problem 5, you wrote a function **`coin_worth`** that expects a character representing a coin and returns the decimal worth of that coin.

Problem 4 - Step 1

Copy the opening comment with the signature, purpose, and **`bool`** test expressions and the function definition for Homework 10 - Problem 5's function **`coin_worth`**.

- Note: if you did not do Homework 10 or you are not confident in your version of **`coin_worth`**, you can e-mail me and ask for a version of **`coin_worth`**.
  - BUT if I send you this function, you **cannot** then submit that **`coin_worth`** version as part of an

improved Homework 10 submission!

Now that **`coin_worth`** is in your **`111hw11.cpp`** file, it can be used by another function that follows it in this file.

Problem 4 - Onward!

Consider:

- Remember the `string` class method **`at`**? It expects the position of a desired character in that string, and returns the `char` at that position in the string (remembering, also, that it considers the position of the first character in the string to be position **0**, not 1).
- Also remember the `string` class method **`length`**, that expects nothing and returns the length of the calling string.
- Do you see that you could use a **`while`** loop with this **`at`** method to do something with each **`char`** within a string?
  - You could have an **`int`** local variable, representing the next character position in the string, and use it as the argument to **`at`** to get the character at that position.
  - And this loop can continue while this **`int`** local variable's value is still less than the string's length.
- Recall, also, how you "built" a results-string in the Week 13 Lab Exercise in function `repeat_str`. You used another local variable that you kept "adding" string copies to.

NOW consider: what if you had a string whose characters were coin values? For example:

`"qDnNPdQpN"`

- Using the design recipe, write a function **`sum_coins`** that expects a string of coin characters, and returns the sum of the decimal worths of the coin characters in that string. For example,

```
sum_coins("qDnNCdQpN") == 0.25 + .10 + .05 + .05 + .01 + .10 + .25 + .00 + .05
sum_coins("Qn") == 0.30
```

- For full credit, **`sum_coins`** must also:
  - appropriately call and use **`coin_worth`**
  - appropriately use a **`while`** loop
- Remember to include a **signature**, **purpose**, **function header**, **tests**, and then completed **function body** for **`sum_coins`**.
- Be sure to include your tests BOTH in a comment after your purpose statement, AND in `main`, as we have done in class.
- IF you would like, you can also include one or more `cout` statements that include JUST an example call of your function **after** these tests, so that you see the value those call(s) return.

Problem 5 - function `starbox`

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

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

For some more loop practice...

...Consider -- what would you see on-screen if you called Week 13 Lab Exercise's function **`starline`** **repeatedly**? That is, if you called it, for example, 4 times, each time with an argument of, say, `10`?

- Remember, **`starline`** expects a desired number of stars/asterisks, has the side-effect of outputting a line of that many asterisks to the screen, and returns the number of asterisks printed to the screen.

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 **`111hw11.cpp`** file, it can be used by another function that follows it in this file.

Problem 5 - Onward!

- Using the design recipe, design a C++ function **`starbox`** that expects a desired number of rows **and** a desired number of asterisks per row, has the side-effect of printing to the screen that many rows of asterisks, each with that many asterisks per row, and returns the **total number** of asterisks printed out.
- For full credit, **`starbox`** must also:
  - appropriately call and use **`starline`**
  - appropriately use a **`while`** loop
- For example, **`starbox(3, 5) == 15`** and has the side-effect of causing the following to be printed to the screen:

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

- And, **`starbox(4, 2) == 8`** and has the side-effect of causing the following to be printed to the screen:

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

- And, since **`starbox`** has side-effects, its purpose statement needs to include a "and has the side-effects of..." clause, and your tests for **`starbox`** after its purpose statement should INCLUDE a description of those side effects as well as including a `bool` expression that should be `true` for each example call.

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

  - give the `bool` expression that should be true, as well as 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, along with the hoped-for `true` result from comparing the actually-returned value to the expected returned value.

  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 star box with <num> rows and

<num> columns, 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 - another interactive front-end for a function

Again: we have mentioned in class that not all **`main`** functions are used just for testing other functions. Sometimes they simply "control" a desired program.

You tried this out in Homework 10 - Problem 8. Now that we have covered loops, we can make an interactive front-end for a function that does more!

Again, this out will be less awkward if it is done in a separate C++ program (with its own **`main`** function).

Copy the contents of the **`111template.cpp`**, posted on the course Canvas site and on the public course web site, into a file within the CS50 IDE (at https:/cs50.dev/) named **`111hw11-prob6.cpp`**.

This program will contain a program whose `main` function JUST serves as a "loopy" interactive front end for previously-designed function(s) (for example, as `lab12.cpp`'s `main` function does).

CHOOSE ONE of the FOLLOWING OPTIONS for this problem. (You can choose to do more than one for the practice, but I will only grade one of them, in the interests of time... `8-)`).

option 6-1 - loopy front-end for `starbox`

The purpose of this option is to write a **`while`** loop that is NOT controlled by a counter-style local variable (that is, it is not a "count-controlled" loop), that just happens to be in a **`main`** function used ***not*** for testing, but for controlling a desired program.

Consider Problem 5's function **`starbox`**, which expects a desired number of rows **and** a desired number of asterisks per row, has the side-effect of printing to the screen that many rows of asterisks, each with that many asterisks per row, and returns the **total number** of asterisks printed out.

What if you would like an interactive front end for **`starbox`** that would allow to be called MORE than once? BUT, instead of knowing how many times in advance it will go, you'd like to have it go as long as the user asks to continue. One way to do this is an approach that could be called a "question-controlled" loop.

(That is, instead of being controlled by the value of a counter variable, it is controlled by a variable set by a user's **answers** to **questions**.)

In the **`111template.cpp`** you pasted into your **`111hw11-prob6.cpp`** file, find the comment:

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

- **AFTER** this comment -- but **BEFORE** the comment and function header for the function named **`main`** -- paste in COPIES of your signature, purpose, opening-comment tests, and function definition for **`starline`** and **`starbox`**.
  - (In this case, do **NOT** copy over the tests from its testing `main` in `111hw11.cpp` -- but **DO** copy over the tests in its opening comment, after its purpose statement.)
  - (You can now delete the `"...PUT YOUR SIGNATURES,..."` comment if you wish, or leave it -- your choice!)
- Then, in its `main` function, add code that does the following -- (this is **pseudocode** for a question-controlled loop):
  - Declare local variables to hold a user's answer, and an entered desired number of rows, and an entered

desired number of asterisks per row. (Carefully choose appropriate data types for these!)

- Ask the user to enter **`y`** if they would like a box of stars.
- Read in their answer using **`cin`**
- while their answer is **`"y"`**,
    - ask the user to enter their desired number of rows
    - read in the number they enter using **`cin`**
    - ask the user to enter their desired number of asterisks per row
    - read in the number they enter using **`cin`**
    - call **`starbox`** appropriately, with the now-set local variables as its arguments, such that its side-effects will be seen, but its return-value ignored
    - ask the user to enter **`y`** if they would like another box of stars
    - read in their answer using **`cin`**

option 6-2 - loopy front-end for `compute_it`

The purpose of this option is to write a **`while`** loop that is NOT controlled by a counter-style local variable (that is, it is not a "count-controlled" loop), that just happens to be in a **`main`** function used ***not*** for testing, but for controlling a desired program.

Consider Homework 10 - Problem 6's function **`compute_it`**, which expects an operator expressed as a **`char`** expression and two numbers, and returns the result of performing the computation with the operator corresponding to that **`char`** expression to those two numbers.

What if you would like an interactive front end for **`compute_it`** that would allow to be called MORE than once? BUT, instead of knowing how many times in advance it will go, you'd like to have it go as long as the user asks to continue. One way to do this is an approach that could be called a "question-controlled" loop.

(That is, instead of being controlled by the value of a counter variable, it is controlled by a variable set by a user's **answers** to **questions**.)

In the **`111template.cpp`** you pasted into your **`111hw11-prob6.cpp`** file, find the comment:

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

- **AFTER** this comment -- but **BEFORE** the comment and function header for the function named `main` -- paste in COPIES of your signature, purpose, opening-comment tests, and function definition for **`compute_it`**.
    - (In this case, do **NOT** copy over the tests from its testing `main` in `111hw10.cpp` -- but **DO** copy over the tests in its opening comment, after its purpose statement.)
    - (You can now delete the `"...PUT YOUR SIGNATURES,..."` comment if you wish, or leave it -- your choice!)
- Then, in its **`main`** function, add code that does the following -- (this is **pseudocode** for a question-controlled loop):
    - Declare local variables to hold a user's answer, and an entered operator-character, and two desired numbers. (Carefully choose appropriate data types for these!)
    - Ask the user to enter **`y`** if they would like a computation

- Read in their answer using **`cin`**
- while their answer is **`"y"`**,
    - ask the user to enter their desired operator-character
    - read in the character they enter using **`cin`**
    - ask the user to enter the first number for their desired computation
    - read in this first number they enter using **`cin`**
    - ask the user to enter the second number for their desired computation
    - read in this second number they enter using **`cin`**
    - call **`compute_it`** appropriately, with the now-set local variables as its arguments, such that its result will be printed to the screen within an appropriate message.
    - ask the user to enter **`y`** if they would like another computation
    - read in their answer using **`cin`**

Remember to submit your files-in-progress **`111hw11.cpp`**, **`111hw11-out.txt`**, and **`111hw11-prob6.cpp`** early and often!