Homework

00

C++ Mechanics

Here is our problem for your first homework assignment.

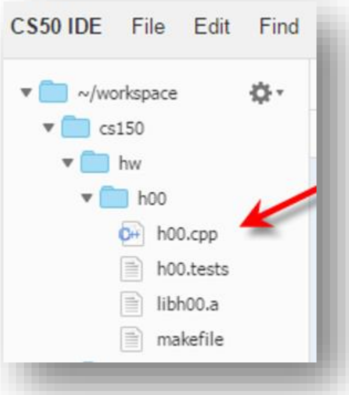
> *A metric ton is* **35,273.92** *ounces. Write a program that will read the weight of a package of breakfast cereal in ounces and output the weight in metric tons as well as the number of boxes needed to yield one metric ton of cereal.* *--Savitch, Absolute C++ 5th Edition, Chapter 2*

This is an **interactive IPO** (*Input, Processing, Output*) programs. You'll type the input at the keyboard in response to a **prompt**, and the output will be displayed on your monitor. If you find this "old fashioned", rest assured that the concepts you learn will remain the same for even the most sophisticated program.

1. The Starter Code

For each assignment, I'll provide you with a set of "starter files" that provide a framework for running and testing simple C++ programs. Make sure that you have completely configured your IDE and GitHub Classroom Repository for C++. Pull down the latest starter code using the following two commands (if you haven't already).

```
cd
get-starters
```



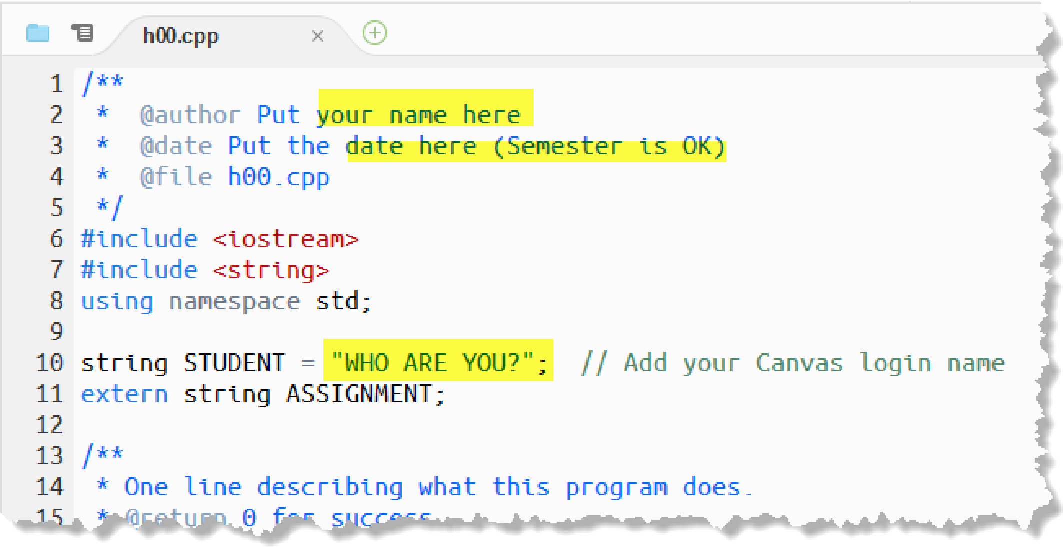
Open the **source code** for this problem by double-clicking **h00.cpp** in the file pane.

2. Identify Yourself

There are three places you'll need to add **identification** to every project you build.

- First, in the **file documentation comment** at the top of the program, add your name after the `@author` tag and the date after the `@date` tag. You can just use the semester for the date if you like, but tell me what section you are in.

- Next, find the **STUDENT** variable and fill in in with your **Canvas login ID**. I use this as a sorting key when processing the assignments; if you do this wrong, you will not receive any credit for an assignment.



```
/**
 *  @author Put your name here
 *  @date Put the date here (Semester is OK)
 *  @file h00.cpp
 */
#include <iostream>
#include <string>
using namespace std;

string STUDENT = "WHO ARE YOU?";  // Add your Canvas login name
extern string ASSIGNMENT;

/**
 * One line describing what this program does.
 * @return 0 for success
```

3. Designing a Solution

When it comes to programming, C++ is like any other language in at least one way: **all programming starts with planning and design**. Before we can write our C++ program, we have to spend time thinking about the inputs, processing and outputs.

- **Always** design your programs **before** you start writing code.
- What are the **inputs**, **outputs**, **algorithms** and **assumptions**?
- **Write it in English** before you ever start writing in C++.

> The sooner you start writing code, the longer it will take you to finish.

A Preliminary Solution

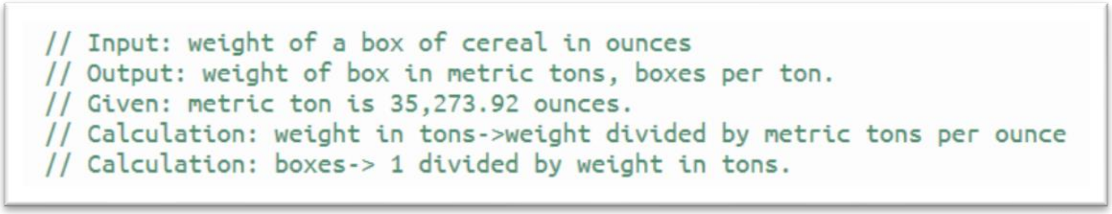
Here is some information that we can discover from the problem description:

- Input: weight of a box of cereal in ounces
- Output: weight of box in metric tons *and* number of boxes in a metric ton.
- Given: metric ton is **35,273.92** ounces
- Calculation: the weight in metric tons is equal to the weight in ounces divided by the number of metric tons per ounce.
- Calculation: the number of boxes per metric ton is equal to one divided by the weight of a single box in metric tons.

You shouldn't jump in and start coding at this point, but you **can** put your design information into a **program comment** before you begin programming.

Internal Documentation

Internal documentation (such as that needed by the programmer to implement the function), should appear in an **implementation comment** inside the function. Place that in the **run()** function, like this:



```
// Input: weight of a box of cereal in ounces
// Output: weight of box in metric tons, boxes per ton.
// Given: metric ton is 35,273.92 ounces.
// Calculation: weight in tons->weight divided by metric tons per ounce
// Calculation: boxes-> 1 divided by weight in tons.
```

> Using single-line comments is easier than using paired comments, because your IDE has a shortcut key to generate them (Shift+/).

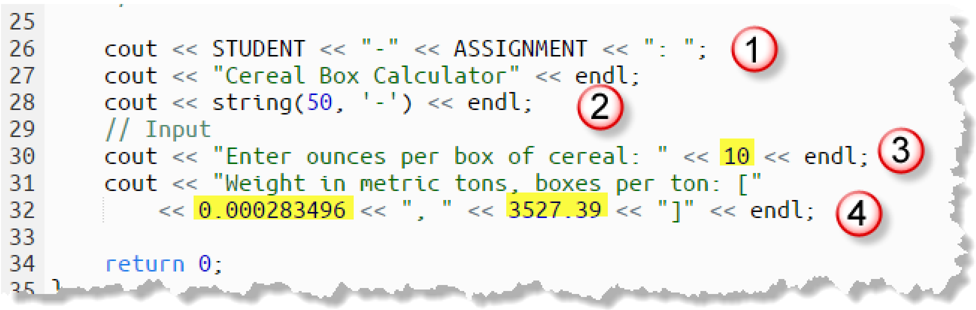
What is the run Function

As you know from Chapter 0 in the course reader, every C++ program begins with a **function** named **main()**. When you scroll though **h00.cpp** you won't find **main()** but a **function** named **run()** instead. In CS 150 we'll use **run()** for our programs to simplify the process of testing. The "real" **main()** function is inside the **library file** **libh00.a**.

4. Mocking Up the Interactions

The easiest way to start coding an IPO program is being by **mocking up the interaction**, using plain output, substituting both input and output values with **literals**.

The C++ **standard library output object** is named **`cout`** (analogous to **`System.out`** in Java). You use it along with the **insertion operator** **`<<`** like this:



```
25
26     cout << STUDENT << "-" << ASSIGNMENT << ": ";
27     cout << "Cereal Box Calculator" << endl;
28     cout << string(50, '-') << endl;
29     // Input
30     cout << "Enter ounces per box of cereal: " << 10 << endl;
31     cout << "Weight in metric tons, boxes per ton: ["
32         << 0.000283496 << ", " << 3527.39 << "]" << endl;
33
34     return 0;
35 }
```

The **arguments** sent to **`cout`** are printed from left to right, each separated from the others by the insertion operator. Place these statements inside the **`run()`** function.

Let's look at each of the sections:

1. The first line has **four arguments**: your student ID and the assignment number, stored in two **variables** that have been previously defined, and two **string literals**, one containing a hyphen and the other containing a colon followed by a space. A **string literal** is text inside of **double quotes**.

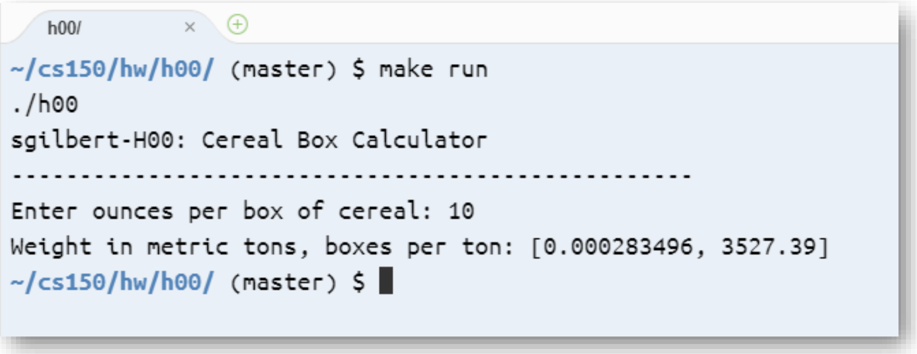
   Each output line ends in a **semicolon**, the C++ **statement terminator**. Each line (except the first), also ends with **`endl`**, pronounced "*end-ell*", which represents the **newline character** on your platform. That means the first two source code lines will produce one line of output.

2. The second section (line 28 in the code shown here), creates a C++ **`string`** object by using a **constructor**. In C++ **`string`** objects are a different type than string **literals**. This **`string`** will consist of 50 hyphens. To use the C++ **`string`** type, the starter code includes the **`<string>`** header.
3. The third section is the input section. I have separated (and highlighted) the numbers in the code that I expect to **receive as input from the user** from the literal text which will not change when the program is run. There are no quotation marks around those numbers.
4. I have done the same thing in the last section of which will be the **program's output.** This style of mockup will make the subsequent development of the program a little easier because you won't have to change the output text and possibly mess up the spacing.

5. Compile, Link & Run

Once you've entered and saved your source code, you're ready to **compile** and **link** it. This is called building the program, and is done by running a program named `make`. Switch to the terminal, make sure you are in the `h00` folder, and then type `make run`.

1. The `make` program reads the `makefile` and **compiles** the `.cpp` file.
2. If everything compiles correctly, the **linker** combines the object-code with the library and produces the executable.
3. If your program built successfully, the `make` program runs it.



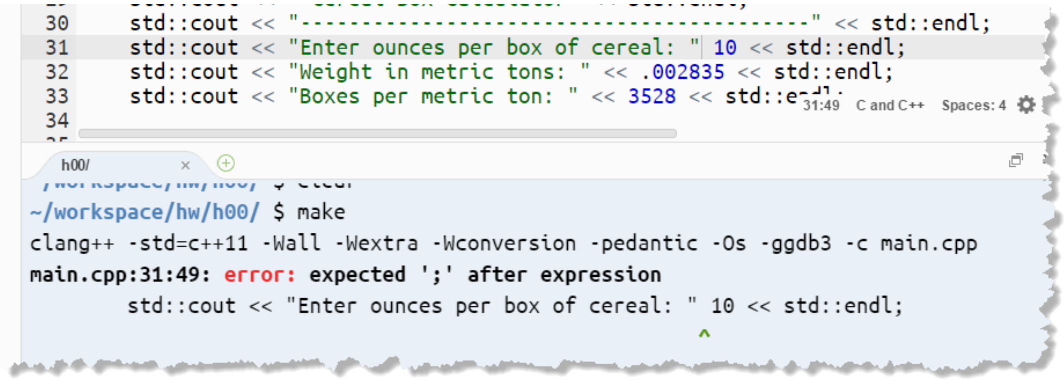
```
~/cs150/hw/h00/ (master) $ make run
./h00
sgilbert-H00: Cereal Box Calculator
--------------------------------------------------
Enter ounces per box of cereal: 10
Weight in metric tons, boxes per ton: [0.000283496, 3527.39]
~/cs150/hw/h00/ (master) $
```

The output appears exactly like the interactions dialog we mocked up.

Shortcut: use Alt+L to open a terminal in a particular directory.

No! Wait! Something Went Wrong!!!!

Of course it's possible that your code didn't compile and run successfully. There are two errors that can occur at this point:



```
30     std::cout << "----------------------------------------" << std::endl;
31     std::cout << "Enter ounces per box of cereal: " 10 << std::endl;
32     std::cout << "Weight in metric tons: " << .002835 << std::endl;
33     std::cout << "Boxes per metric ton: " << 3528 << std::e
34
```

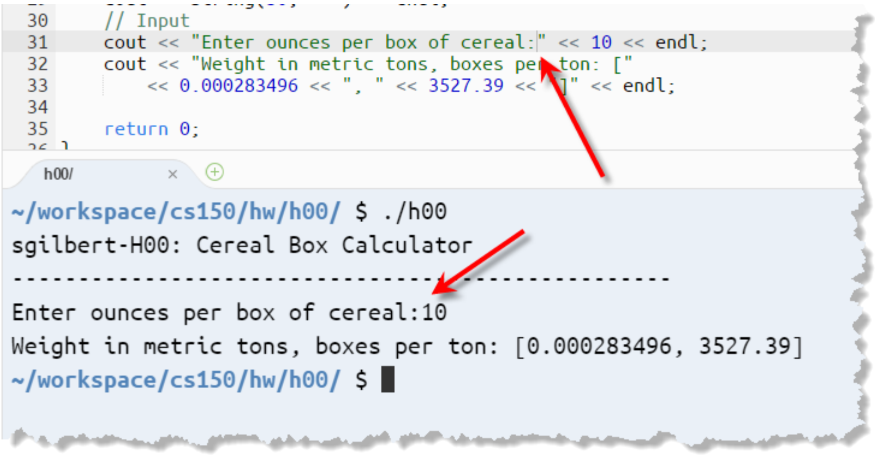
```
~/workspace/hw/h00/ $ make
clang++ -std=c++11 -Wall -Wextra -Wconversion -pedantic -Os -ggdb3 -c main.cpp
main.cpp:31:49: error: expected ';' after expression
        std::cout << "Enter ounces per box of cereal: " 10 << std::endl;
                                                        ^
```

- Syntax or compiler errors occur when you have broken one of the grammar rules of C++. If that occurs, you'll see an (often inscrutable) error message in the output pane instead. Follow these instructions exactly to fix them.

  1. Scroll up to the first error that appears in the output window and make note of the file name and line numbers.
  2. The second line of the error message attempts to show you where the compiler got confused. In this case, it is right before the literal number **10**.
  3. Go to the text editor and **fix the problem** and them save and build again. This is where things get tricky. The compiler **doesn't know** what you **intended** to write, so the solution it recommends is often incorrect. The actual solution in this case is to add the insertion operator (`<<`) that we've forgotten.

  Once you've corrected and saved your source code, build again to see if you've fixed the problem. You **can't go onto the next step** until there are no errors.

- Logic errors or **bugs** occur when your program **doesn't do what it is supposed to do**. If the output of the program looks like this when you run it then you have a **logic error**, because you removed a space that was supposed to appear in the output according to our specification.



```
    // Input
    cout << "Enter ounces per box of cereal:" << 10 << endl;
    cout << "Weight in metric tons, boxes per ton: ["
        << 0.000283496 << ", " << 3527.39 << "]" << endl;

    return 0;
```

```
~/workspace/cs150/hw/h00/ $ ./h00
sgilbert-H00: Cereal Box Calculator
--------------------------------------------------
Enter ounces per box of cereal:10
Weight in metric tons, boxes per ton: [0.000283496, 3527.39]
~/workspace/cs150/hw/h00/ $
```

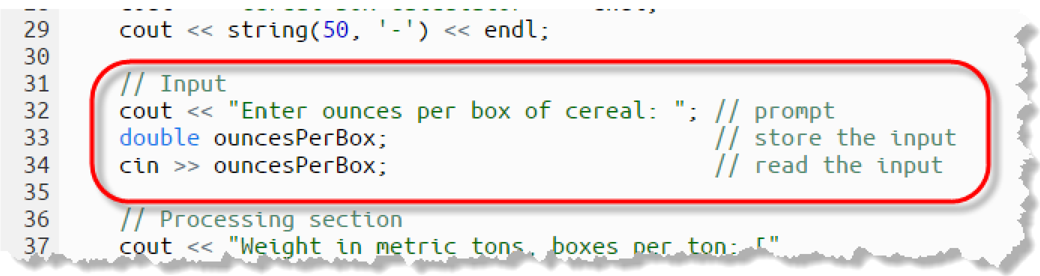
6. Input, Processing and Output

Our program now looks like the mock-up so we can turn to **input and processing**.

- Create **variables** to hold the **input** and the **results** of our calculations.
- Use **`cin`** (*see-in*) with the **extraction** operator **`>>`** to read the input.
- Write **expressions** to calculate the output, storing that in variables.
- **Display** the output and then **test** to see that the output is correct.

6.1 Reading Input

Here's the input section as I've completed it.



```
cout << string(50, '-') << endl;

// Input
cout << "Enter ounces per box of cereal: "; // prompt
double ouncesPerBox;                        // store the input
cin >> ouncesPerBox;                        // read the input

// Processing section
cout << "Weight in metric tons, boxes per ton: ["
```

We need one input value: the number of ounces per box. Cereal boxes often contain a portion of an ounce, so we'll use the data-type called **`double`**. I prefer **camelCase** style for variable names; you may use all lowercase with underscores, which is fine.

To **read input** from the user, we first **prompt for the information** that we expect to be typed and then read the input from the standard input stream **`cin`** ; the **`cin`** object reads from **standard input**, which is your terminal by default.

Prompting Notes

The "mockup" data previously appearing on the prompt line has been removed, as well as the newline (**`endl`**) appearing after the prompt. When you display a prompt for input, you generally omit the **`endl`** at the end of the line.

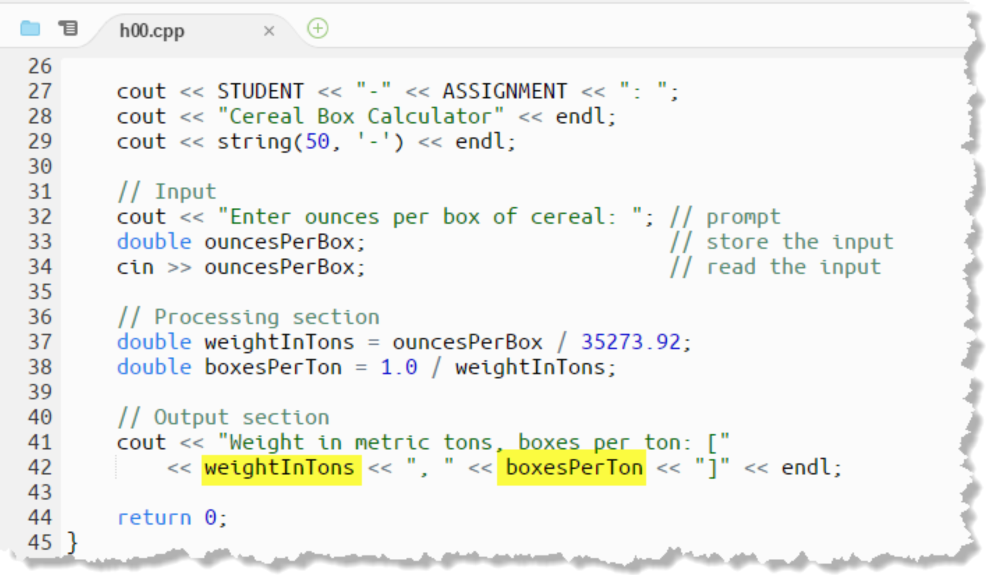
In addition, make sure that the prompt **ends in a single space**, so that it displays the prompt but leaves the console cursor–the blinking vertical bar or square that marks the current input position at the end of the line–waiting for the user's response.

Converting Input Data

The final statement in the marked section reads a sequence of characters typed by the user at the keyboard, and stores the results in the variable **`ouncesPerBox`**. Because this variable was declared as a **`double`**, the **`>>`** operator **automatically converts** the characters typed by the user into a floating-point value.

6.2 Processing and Output

Next, **create variables to hold the output values**. We have two outputs, so I'll create two variables: **`weightInMetricTons`** and **`boxesPerMetricTon`**. Initialize those variables and then replace the "mockup" values with the new variables.



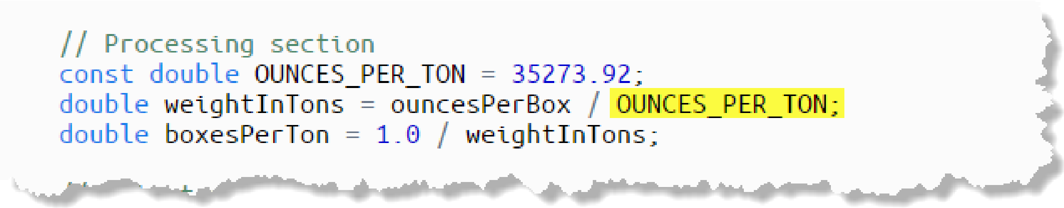
```
    cout << STUDENT << "-" << ASSIGNMENT << ": ";
    cout << "Cereal Box Calculator" << endl;
    cout << string(50, '-') << endl;

    // Input
    cout << "Enter ounces per box of cereal: "; // prompt
    double ouncesPerBox;                        // store the input
    cin >> ouncesPerBox;                        // read the input

    // Processing section
    double weightInTons = ouncesPerBox / 35273.92;
    double boxesPerTon = 1.0 / weightInTons;

    // Output section
    cout << "Weight in metric tons, boxes per ton: ["
        << weightInTons << ", " << boxesPerTon << "]" << endl;

    return 0;
}
```

Initialization Note

In C++, variables are not initially given a value; instead, they use whatever random value happens to be in memory at that time. (This is different than Java which prohibits assigning to uninitialized variables). Instead of first creating the variables and then assigning a value, create variables **only when you can calculate an initial value**.

A Small Improvement: Constants

Our program looks fine, but has one small flaw. We were given the number of ounces in a metric ton using the literal value `35,273.92`. In our calculations, though, you **should not** to use such "**magic numbers**"; they are too easy to mistype and they make code more confusing. Instead, **store all "given" values in named constants**.



```
// Processing section
const double OUNCES_PER_TON = 35273.92;
double weightInTons = ouncesPerBox / OUNCES_PER_TON;
double boxesPerTon = 1.0 / weightInTons;
```

Constants can appear inside your function (as I've done here), or, if you intend to use them throughout your program, you can enter them before the `run()` function.

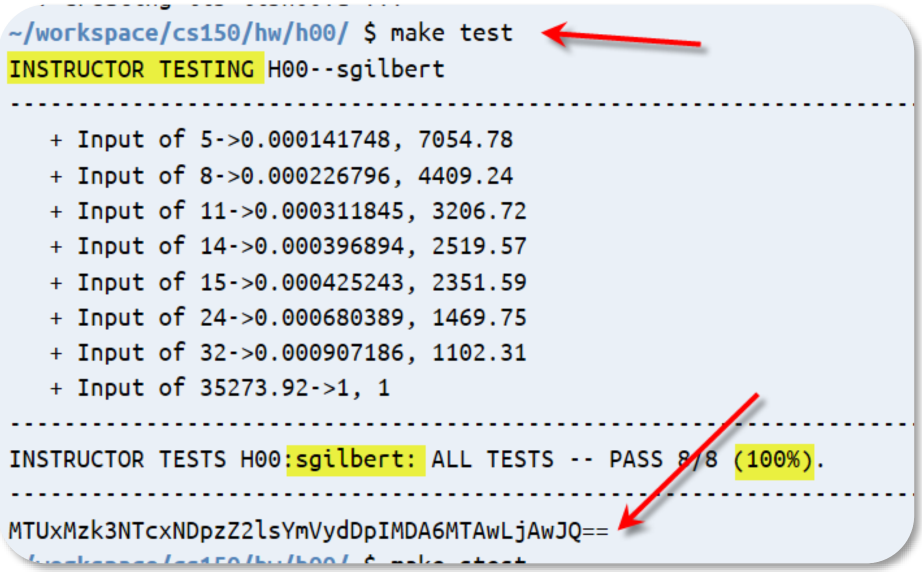
7. Testing Your Program

How do you know if your program is correct? Simple; you need to **test** it. There are two kinds of tests you can do in your homework problems: **instructor tests** (which I've written), and **student tests**, which you can write yourself.

Instructor Tests

To run the instructor tests, type `make test` in the terminal. Make sure, of course, that you are in the correct directory (`~/workspace/cs150/hw/h00`). The `run()` function is called with different input values, and checked to see if you have the correct output.

In the instructor tests for this assignment, I've calculated several sizes of boxes from `5` to `32` ounces using Google Docs, and adjusted the decimal places so that they match the default used by C++. I also set the last entry to `35273.92` (the number of ounces in a metric ton), just to check that you got `1` as the output each calculation.



```
~/workspace/cs150/hw/h00/ $ make test
INSTRUCTOR TESTING H00--sgilbert
--------------------------------------------------------------------
   + Input of 5->0.000141748, 7054.78
   + Input of 8->0.000226796, 4409.24
   + Input of 11->0.000311845, 3206.72
   + Input of 14->0.000396894, 2519.57
   + Input of 15->0.000425243, 2351.59
   + Input of 24->0.000680389, 1469.75
   + Input of 32->0.000907186, 1102.31
   + Input of 35273.92->1, 1
--------------------------------------------------------------------
INSTRUCTOR TESTS H00:sgilbert: ALL TESTS -- PASS 8/8 (100%).
--------------------------------------------------------------------
MTUxMzk3NTcxNDpzZ2lsYmVydDpIMDA6MTAwLjAwJQ==
```

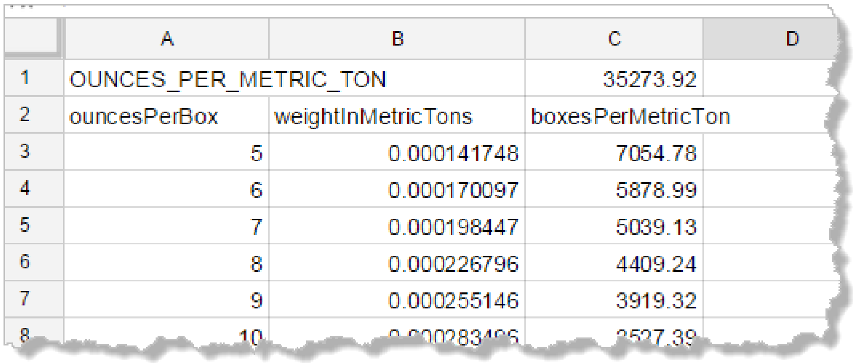
At the bottom of the run is the score. Make sure that **your ID** correctly appears, and that the assignment displayed is also correct. The final line is the completion code.

Submitting

To submit for credit, type `make submit` from the console. You'll receive a confirmation if your submission is accepted. The **CS 150 Homework Console** allow you to check your past scores and see about future deadlines. If you have difficulties, check the FAQ on Piazza or Canvas. (You should get this part to work before leaving the first class.)

Optional: Student Testing

If you want, you can also run your own tests. To do this, you need to supply several input values, and then figure out **exactly what the expected output should be**. The easiest way to do that is to use Excel or Google Docs like this:



	A	B	C	D
1	OUNCES_PER_METRIC_TON		35273.92	
2	ouncesPerBox	weightInMetricTons	boxesPerMetricTon	
3	5	0.000141748	7054.78	
4	6	0.000170097	5878.99	
5	7	0.000198447	5039.13	
6	8	0.000226796	4409.24	
7	9	0.000255146	3919.32	
8	10	0.000283496	3527.39	

Since our program hasn't formatted any of the output, so you might have to adjust the number of decimal places for each portion.

Adding the Tests

The **CS 150 framework** has a simple student testing scheme for IPO programs. Here's how it works.

- For each new input you want to test, **add a new line** to the file `h00.tests` (which you'll find in the folder with your starter code). If there are multiple inputs, then separate them with a space or a newline (`\n`).
- Add a vertical bar (`|`) to **separate the input from the expected output**, and then type the output that you want to check.
- The values being checked appear between square brackets (`[ ]`) in your program. If you have multiple outputs, they must all appear between a single set of brackets. Do not put the square brackets in your test file, however.
- To run the student tests, type `make stest`

When run like this, instead of reading the input from the keyboard, input will be read from the text file, and each line of input will be compared to the expected output.