Assignment 1: Welcome to C++!

Handout and assignment based on an assignment by Eric Roberts and Julie Zelenski.
Thanks to Sophia Westwood for suggesting Flesch-Kincaid readability as an assignment.

As a prelude to the later assignments in this quarter, your first task is to get acclimated to the C++ programming language. This assignment contains several C++ tasks that give you a feel for many different aspects of the language – functions, primitive types, strings, and even some recursion for color.

Assignment Files Due Tuesday, July 2 at 11:00AM

Email Due Friday, July 5 at 11:00AM

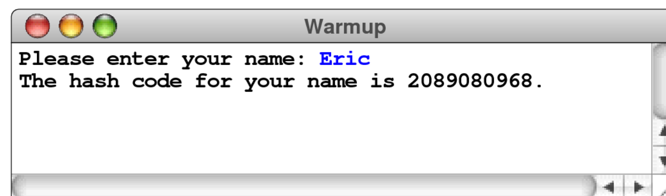
Part 1: Welcome to CS106B!

Your first task is to set up your C++ compiler. If you're using the machines in Stanford's public clusters, you don't need to install the software. If you're using you own machine, you should consult one of the following handouts, depending on the type of machine you have:

- Handout #04M. Downloading Xcode on the Macintosh
- Handout #04P. Downloading Visual Studio for Windows
- Handout #04L. Getting Started on Linux

Once you have the compiler ready, go to the assignments section of the web site and download the starter file for the type of machine you are using. For either platform, the **`Assignment1`** folder contains five separate project folders: one for this warmup problem and one for each of the four problems in Part 2 of the assignment. Open the project file in the folder named **`0-Warmup`**. Your mission in Part 1 of the assignment is simply to get this program running. The source file we give you is a complete C++ program—so complete, in fact, that it comes complete with two bugs. The errors are not difficult to track down (in fact, we'll tell you that one is a missing declaration and the other is a missing **`#include`** statement). This task is designed to give you experience with the way errors are reported by the compiler and what it takes to fix them.

Once you fix the errors, compile and run the program. When the program executes, it will ask for your name. Enter your name and it will print out a "hash code" (a number) generated for that name. We'll talk later in the class about hash codes and what they are used for, but for now just run the program, enter your name, and record the hash code. You'll email us this number. A sample run of the program is shown below:



Once you've gotten your hash code, we want you to e-mail it to your section leader and introduce yourself. You don't yet know your section assignment, but will receive it via email after signups close, so hold on to your e-mail until then. **You won't learn your section leader until after the due date of the assignment files, so the email isn't due until Friday, June 5th 11AM.** You should also **cc** me on

the email(**`adgress@cs.stanford.edu`**) so I can meet you as well! Here's the information to include in your email:

1. Your name and the hash code that was generated for it by the program.
2. Your year and major (if you are an undergrad) or department (if you are a grad student).
3. When you took CS106A (or equivalent course) and how you feel it went for you.
4. What you are *most* looking forward to about CS106B.
5. What you are *least* looking forward to about CS106B.
6. Any suggestions that you think might help you learn and master the course material.

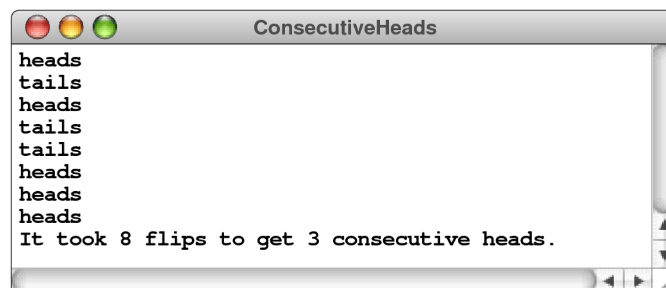
Part Two: Welcome to C++!

Most of the assignments in this course are single programs of a substantial size. To get you started, however, the first assignment is a series of four short problems that are designed to get you used to using C++ and to introduce the idea of functional recursion.

Problem 1: Rosencrantz and Guildenstern Flip Heads

> *Heads. . . .*
> *Heads. . . .*
> *Heads. . . .*
> *A weaker man might be moved to re-examine his faith, if in nothing else at least in the law of probability.*
>
> —Tom Stoppard, *Rosencrantz and Guildenstern Are Dead,* 1967

Write a program that simulates flipping a coin repeatedly and continues until three *consecutive* heads are tossed. At that point, your program should display the total number of coin flips that were made. The following is one possible sample run of the program:



Problem 2: Combinations and Pascal's Triangle

As mentioned in Chapter 2 of the course reader, the mathematical combinations function $c(n, k)$ is usually defined in terms of factorials, as follows:

$$c(n,k) = \frac{n!}{k!(n-k)!}$$

The values of $c(n, k)$ can also be arranged geometrically to form a triangle in which n increases as you move down the triangle and k increases as you move from left to right. The resulting structure, which is called ***Pascal's Triangle*** after the French mathematician Blaise Pascal, is arranged like this:

$$
\begin{array}{c}
c(0,0) \\
c(1,0) \quad c(1,1) \\
c(2,0) \quad c(2,1) \quad c(2,2) \\
c(3,0) \quad c(3,1) \quad c(3,2) \quad c(3,3) \\
c(4,0) \quad c(4,1) \quad c(4,2) \quad c(4,3) \quad c(4,4)
\end{array}
$$

Pascal's Triangle has the interesting property that every entry is the sum of the two entries above it, except along the left and right edges, where the values are always 1. Consider, for example, the highlighted entry in the following display of Pascal's Triangle:

```
                1
              1   1
            1   2   1
          1   3   3   1
        1   4   6   4   1
      1   5   10  10  5   1
    1   6   15  20  15  6   1
  1   7   21  35  35  21  7   1
```

This entry, which corresponds to $c(6, 2)$, is the sum of the two entries—5 and 10—that appear above it to either side. Using this fact, write a recursive implementation of the `c(n, k)` function that uses no loops, no multiplication, and no calls to `Fact`.

Write a simple test program to demonstrate that your combinations function works. If you want an additional challenge, write a program that uses `c(n, k)` to display the first ten rows of Pascal's Triangle.

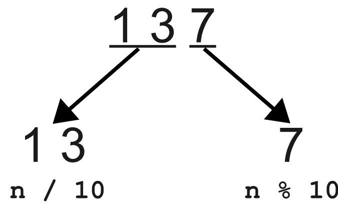
Problem 3: Implementing Numeric Conversions

The `strlib.h` interface exports the following methods for converting between integers and strings:

```
string integerToString(int n);
int stringToInteger(string str);
```

The first function converts an integer into its representation as a string of decimal digits, so that, for example, `integerToString(1729)` should return the string `"1729"`. The second converts in the opposite direction so that calling `stringToInteger("-42")` should return the integer -42.

Your job in this problem is to write the functions `intToString` and `stringToInt` (the names have been shortened to avoid having your implementation conflict with the library version) that do the same thing as their `strlib.h` counterparts but use a recursive implementation. Fortunately, these functions have a natural recursive structure because it is easy to break an integer down into two components using division by 10. This decomposition is discussed on page 42 in the discussion of the `digitSum` function. The integer 137, for example, breaks down into two pieces, as follows:



In other words, you can peel off the last digit of the number **`n`** by using division and modulus by 10. If you use recursion to convert the first part to a **`string`** and then append the character value corresponding to the final digit, you will get the **`string`** representing the integer as a whole.

As a note, it is also possible to split a number into two pieces by separating the *first* digit rather than the last. Although this might seem like a more intuitive way of recursively splitting the number apart, we actually recommend against this approach as it's a bit more complicated to code up.

Your solution should operate recursively and should use no iterative constructs such as **`for`** or **`while`**. Also, you shouldn't call the provided **`integerToString`** function or any other library function that does numeric conversions, since that defeats the point of the assignment. ☺

As you work through this problem, you should keep the following points in mind:

- The value that you get when you compute **`n % 10`** is an integer, and not a character. To convert this integer to its character equivalent you have to add the ASCII code for the character **`'0'`** and then cast that value to a **`char`** (for example, **`char(ch + '0')`**) If you then need to convert that character to a one-character string, you can concatenate it with **`string()`**, as shown here:

  ```
  string() + ch
  ```

  If you are coming from Java, be aware that the Java trick of writing

  ```
  "" + ch
  ```

  does ***not*** work in C++ and will result in very strange behavior – it might return garbage, or just outright crash the program!

- You should think carefully about what the simple cases need to be. In particular, you should make sure that calling **`intToString(0)`** returns **`"0"`** and not the empty string. This fact may require you to add special code to handle this case.
- Your implementation should allow **`n`** to be negative, as illustrated by the earlier example in which **`stringToInt("-42")`** returns -42. Again, implementing these functions for negative numbers will probably require adding special-case code.
- It's possible to split apart numbers and strings in many ways. You are free to split them however you'd like. However, peeling off the last digit as we've suggested is easier than most other approaches.

We strongly encourage you to test your solution on a variety of inputs. There are many different cases that you need to cover, and it's easy to accidentally miss one or two of them.

Problem 4: The Flesch-Kincaid Grade-Level Test

(We will finish covering the material necessary to complete this part of the assignment on Monday.)

Many word processing programs (such as Microsoft Word) employ heuristics that give an estimate for how difficult it is to read a particular passage of text. This makes it possible for authors to evaluate

whether their writings are too complex for their target audience. It also makes it possible for high school English teachers to save lots of time grading essays. ☺

One automated test for determining the complexity of a piece of text is the *Flesch-Kincaid grade level test*, which assigns a number to a piece of text indicating what grade level the computer things is necessary to understand that text. This test makes no attempt to actually understand the meaning of the text, and instead focuses purely on the complexity of the sentences and words used within that text. Specifically, the test counts up the total number of words and sentences within the text, along with the total number of syllables within each of those words. Given these numbers, the grade level score is then computed using the following formula:

$$Grade = C_0 + C_1\left(\frac{num\ words}{num\ sentences}\right) + C_2\left(\frac{num\ syllables}{num\ words}\right)$$

Where C_0, C_1, and C_2 are constants chosen as follows:

$C_0 = -15.59$ $C_1 = 0.39$ $C_2 = 11.8$

(I honestly have no idea where these values came from, but they're the standard values used whenever this test is performed!) The resulting number gives an estimate of the grade level necessary to understand the text. For example, something with grade level 5.15 could be read by a typical fifth-grader, while something with grade level 15 would be appropriate for a typical college junior.

Your job is to write a program that, given a piece of text, computes the Flesch-Kincaid grade level score from that piece of text. When your program starts up, it should prompt the user for a name of a file containing the text to be read. Your program should continuously reprompt the user until they enter the name of a valid file.

Once you've opened the file, you will need a way to take the original text and break it up into individual words and punctuation symbols. This will let you count up how many words and sentences there are. For this purpose, we provide you a **`TokenScanner`** class that lets you read a file one "piece" at a time, where a "piece" is either a punctuation symbol or word. To start off this assignment, see if you can write a program that will open a file and read it one piece at a time using the **`TokenScanner`**.

We did not cover all the functionality of the **`TokenScanner`** class in lecture. We recommend taking the time to look over the documentation for **`TokenScanner`**, which is available on the course website (under "Stanford C++ Library Docs"). In particular, we recommend finding a way to get the **`TokenScanner`** to do the following:

- Skip over whitespace tokens, so you don't have to handle them later on;
- Read directly from a file, which dramatically simplifies the logic; and
- Tokenize strings like **`isn't`** as a single token **`isn't`** rather than the three tokens **`isn`**, **`'`**, and **`t`**.

As a test, try getting your program to print out all the tokens in the file, one line at a time. We provide a reference solution so that you can compare your output against ours.

Now that you have the individual tokens, try updating your program so that you can count the total number of words and sentences in the file. To determine what counts as a word, you can assume that any token that starts with a letter counts as a word, so **`apple`** and **`A-1`** would both be considered words. As an approximation of the number of sentences, you can just count up the number of punctuation symbols that appear at the ends of sentences (namely, periods, exclamation points, and question marks). This isn't entirely accurate, but it's a good enough approximation. As a minor edge case, if a file doesn't appear to have any words or sentences in it, you can just pretend that it contains a single word and a single sentence (which prevents a division by zero error when evaluating the above for-

mula). Then, update your program to print out a readout of the total number of words and sentences it thinks exists in the file.

Before moving onward, why not take some time to test that your program works correctly? We have provided our own reference implementation of this program, which you can use to see how many words and sentences are in a given file. Try comparing our output to your output on some short sample files. Are you getting the same output as us? If not, take a minute to track down what you're doing differently. If you're being more intelligent than our program, you might get more accurate answers than our reference solution. However, a disparity here might also mean that you have a bug somewhere.

Finally, count up how many syllables are in each of the words that you find. Getting an exact syllable count for each word is almost impossible, since this varies dramatically based on pronunciation (for example, the word "are" is just one syllable, while "area" is three). To approximate the number of syllables in a word, you should count up the number of vowels in the word (including 'y'), *except* for

- Vowels that have vowels directly before them, and
- The letter *e*, if it appears by itself at the end of a word.

For example, the word "program" would be counted as having two syllables, one for 'o' and one for 'a'; the word "peach" would have one syllable since "ea" appear next to one another; and the word "deduce" would have two syllables, since the final 'e' does not contribute to the total. Notice that under this definition, the word "me" would have zero syllables in it, since the final "e" doesn't contribute to the total. To address this, you should assume that all words have at least one syllable in them. This approximation of syllable counts isn't exactly correct. In fact, it incorrectly says that there are just two syllables in the word "syllable." However, for our purposes, this is totally fine.

Before moving on, we suggest trying out your program on a few sample inputs and checking it against our reference program. Counting syllables is probably the trickiest part of this assignment, so play around with our sample inputs and check that you're counting correctly.

Once you've counted up the total number of words, sentences, and syllables, you can compute the Flesch-Kincaid Grade Level score for individual files. Try running your program on some of the files we've provided. Do any of the results surprise you? Try finding a piece of text from one of your favorite books (say, one of the *Harry Potter* titles, *Fifty Shades of Grey*, or perhaps *Programming Abstractions in C++*). How do those scores compare to our sample files? If you find anything interesting, let us know about it in your submission!

If you'd like some extra credit, try improving upon our way of counting words, syllables, or sentences. You could also try implementing a different algorithm for computing readability. Play around and see if you find anything interesting!

Possible Extensions

Interested in exploring onward a bit? Here are some suggestions on how to get started.

- **Consecutive Heads:** Could you adjust the program so that you count how many flips are necessary to find some arbitrary sequence of heads and tails? For example, how many flips, on average, are necessary to get the sequence H T H T T? How does this compare to the number of flips required to get H H H H H? Why is that?
- **Pascal's Triangle:** Pascal's triangle has a huge number of interesting mathematical properties. Vi Hart describes one of them in this video: http://www.youtube.com/watch?v=Yhlv5Aeuo_k. Could you update your program to display Pascal's triangle and show off some of its mysterious properties?

- **Numeric Conversions:** Could you update your program to convert Roman numerals into integers? Or perhaps numbers written in a different base, like hexadecimal (base-16) or binary (base-2) into integers?
- **Flesch-Kincaid Readability:** Can you make the program better at counting syllables? Or perhaps try to be more intelligent about how sentences are handled? Could you try measuring some *other* property of a piece of text in order to determine its complexity?

  There was an interesting article in a recent issue of the New Yorker discussing how a researcher measured the complexity of various pieces of rap music over time (you can read the article at http://www.newyorker.com/talk/2013/04/01/130401ta_talk_wilkinson). Can you verify their results? Or could you perform a similar analysis on a different data set?