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.

Here it is – the first programming assignment of the quarter! This assignment is designed to get you comfortable designing and building software in C++. It consists of five problems that collectively play around with control structures, strings processing, recursion, and problem decomposition in C++. By the time you've completed this assignment, you'll be a lot more comfortable working in C++ and will be ready to start building larger projects as the quarter progresses.

Good luck, and have fun!

This assignment must be completed individually. Working in groups is not permitted.
Due Monday, January 23rd at the start of class.

Problem One: 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:

```
Flip: heads
Flip: tails
Flip: heads
Flip: tails
Flip: tails
Flip: heads
Flip: heads
Flip: heads
It took 8 flips to get 3 consecutive heads.
```

In the interest of making grading easier, please try to have your program's output format match our own: list each flip on its own line and print out the final number of flips on the last line.

In order to write this program, you're going to need to be able to generate random numbers. We've provided you a random number generation library that you can read about by checking out the documentation for the Stanford C++ libraries up on the course website. Visit the website, click the link titled "Stanford C++ Library Documentation," then click the link for `"random.h"` for information about what functions are provided to you, what they do, and how to call them. Reading library documentation is an important skill to develop as a programmer, as you'll often have to learn to use new tools along the way!

Problem Two: Combinations and Pascal's Triangle

Let's suppose you have a group of n people. How many ways are there to choose a group of k of them? If you have a group of six people, for example, there are fifteen ways to choose two of them, twenty ways to choose three of them, one way to choose six of them (pick everyone), and one way to choose zero of them (pick no one).

In mathematics, the number of ways to choose k people out of a group of n people is denoted

$$\binom{n}{k}$$

and is read aloud as "n choose k." Numbers of this form are called ***binomial coefficients*** and show up everywhere in computer science. (Take CS109 for more details!)

There's an exact formula for n choose k that's typically defined using the factorial function. Specifically:

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

But what if you didn't know this? It turns out there's a beautiful recursive approach for computing binomial coefficients that uses a structure called ***Pascal's Triangle***, named after the French mathematician Blaise Pascal. Pascal's triangle looks like this:

$$\binom{0}{0}$$

$$\binom{1}{0} \quad \binom{1}{1}$$

$$\binom{2}{0} \quad \binom{2}{1} \quad \binom{2}{2}$$

$$\binom{3}{0} \quad \binom{3}{1} \quad \binom{3}{2} \quad \binom{3}{3}$$

$$\binom{4}{0} \quad \binom{4}{1} \quad \binom{4}{2} \quad \binom{4}{3} \quad \binom{4}{4}$$

…

Notice how the top number in the binomial coefficient increases from top to bottom and the second number in the coefficient increases from left to right.

If you actually evaluate the terms in Pascal's triangle, you get this pattern:

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

...

Notice 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. For example, the highlighted entry, which corresponds to 6 choose 2, is the sum of the two entries, 5 and 10, that appear above it to either side. (Curious why that is? We'll see why in a few weeks!)

Using the fact that every term in Pascal's triangle is the sum of the two terms above it, plus the observation that each term in Pascal's triangle is *n* choose *k* for some *n* and some *k*, write a recursive implementation of a function

```
int nChooseK(int n, int k)
```

that computes *n* choose *k* using no loops, no multiplication, and no calls to a function that computes factorials. You can assume the values of *n* and *k* will be nonnegative and that *k* will be less than or equal to *n*.

Problem Three: Stack Overflows

In C++ (and just about every other programming language), whenever a program calls a function, the computer will set aside some memory for that function call (this space is called a ***stack frame***) in a region of memory called the ***call stack***. Whenever a function is called, it creates a new stack frame, and whenever a function returns the space for that stack frame is recycled.

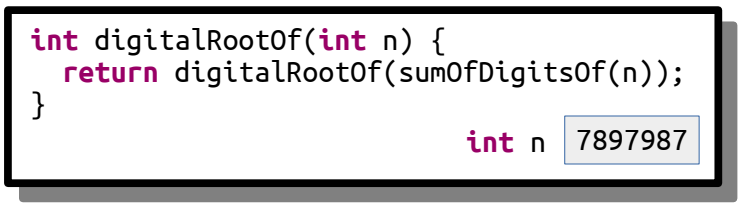
As you saw on Wednesday, whenever a recursive function makes a recursive call, it creates a new stack frame, which in turn might make more stack frames, which in turn might make even more stack frames, etc. For example, when we computed `factorial(5)`, we ended up creating a net of six stack frames: one for each of `factorial(5)`, `factorial(4)`, `factorial(3)`, `factorial(2)`, `factorial(1)`, and `factorial(0)`. They were automatically cleaned up as soon as those functions returned.

But what would happen if, say, you called the factorial function on a very large number, say, something like `factorial(7897987)`? This would end up creating a *lot* of stack frames, specifically, 7897988 of them (one for `factorial(7987987)`, one for `factorial(7987986)`, …, and one for `factorial(0)`). This is such a large number of stack frames that the call stack might not have space to store all of them. When too many stack frames need to be created at the same time, the result is a ***stack overflow*** and the program will crash.

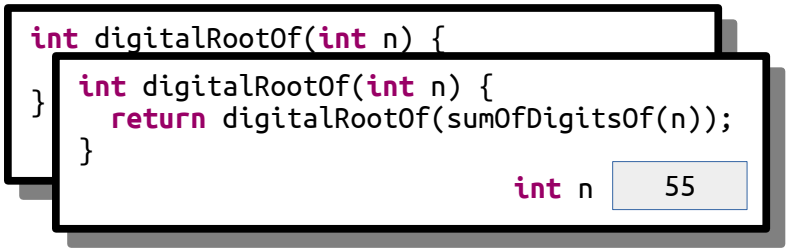
In the previous example, a stack overflow would occur because we needed a large but still finite number of stack frames. However, you often see stack overflows resulting due to an error in a piece of code. For example, consider the following (buggy!) implementation of the `digitalRootOf` function from Friday's lecture:

```
int digitalRootOf(int n) {
    return digitalRootOf(sumOfDigitsOf(n));
}
```

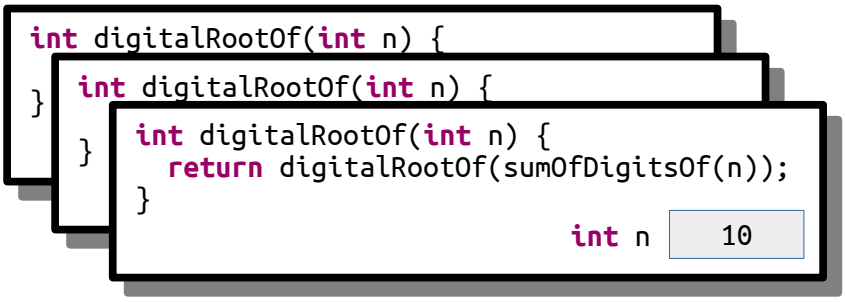
Let's imagine that you try calling `digitalRootOf(7897987)`. The initial stack frame looks like this:



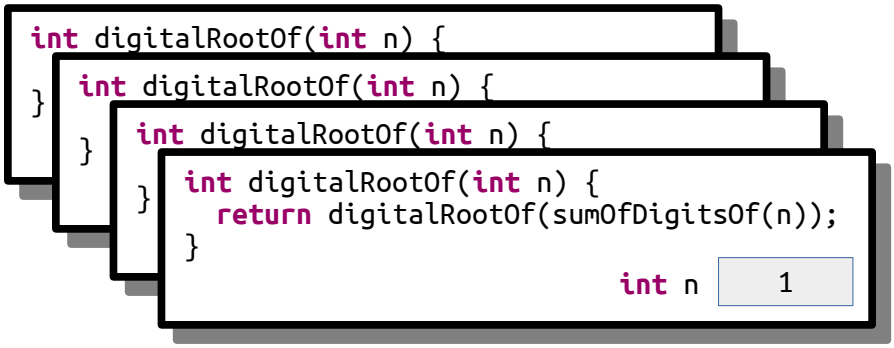
This call now tries to call `digitialRootOf(sumOfDigitsOf(7897987))`. The sum of the digits in the number is 7 + 8 + 9 + 7 + 9 + 8 + 7 = 55, so this fires off a call to `digitalRootOf(55)`, as shown here:



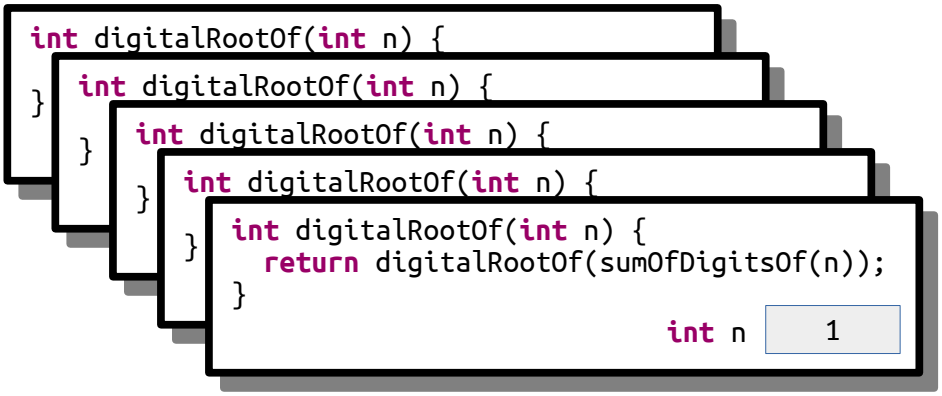
This now calls `digitalRootOf(sumOfDigitsOf(55))`, which ends up calling `digitalRootOf(10)`:



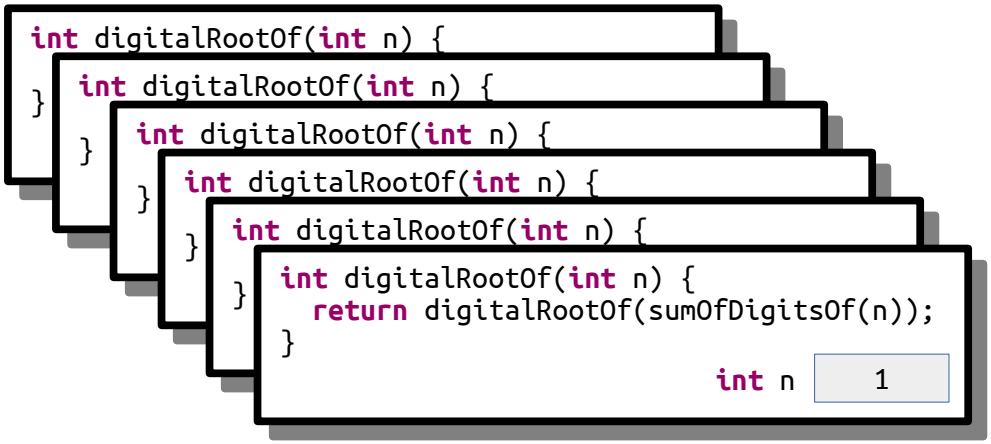
This now calls `digitalRootOf(sumOfDigitsOf(10))`, which ends up calling `digitalRootOf(1)`:



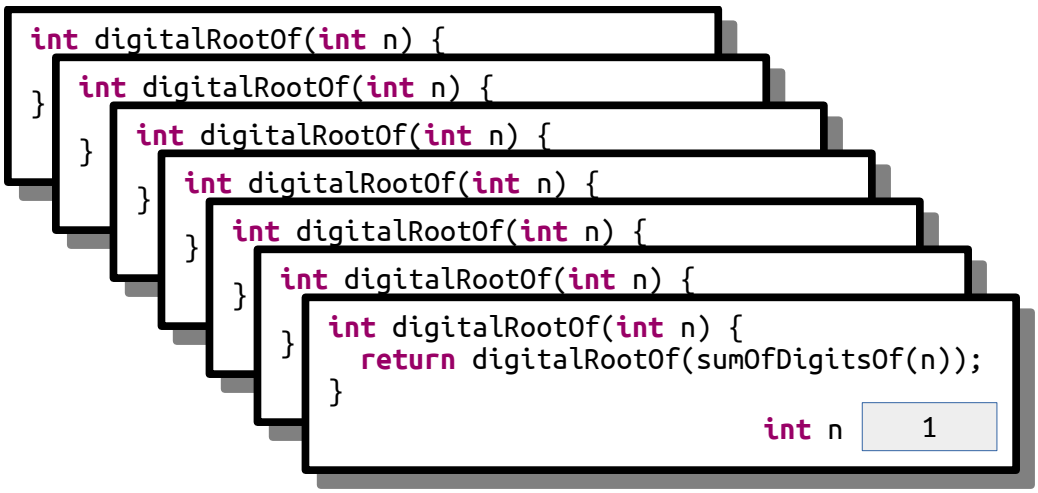
Now things get weird. This function call now tries to call `digitalRootOf(sumOfDigitsOf(1))`, which ends up calling `digitalRootOf(1)` again, as shown here:



This call makes yet another call to `digitalRootOf(1)` for the same reason:



And *that* call makes yet *another* call to digitalRootOf(1):



And, as you can see, this recursion is off to the races making recursive calls over and over and over again. It's like an infinite loop, but with function calls. This code will eventually trigger a stack overflow because at some point it will exhaust the memory in the call stack.

Another reason you'll see stack overflows is in the case where you have a recursive function that, for some reason, misses or skips over its base case. For example, let's suppose you want to write a function

```
bool isEven(int n)
```

that takes as input a number n and returns whether it's an even number. You make the observation that the number 0 is even (trust us, it is – take CS103 for details!) and that, more generally, a number n is even precisely if $n - 2$ is even. For example, 2 is even because $2 - 2 = 0$ is even, 4 is even because $4 - 2 = 2$ is even, 6 is even because $6 - 2 = 4$ is even, etc.

Based on this (correct) insight, you decide to write this (incorrect) recursive function:

```
bool isEven(int n) {
    if (n == 0) {
        return true;
    } else {
        return isEven(n - 2);
    }
}
```

Now, what happens if you call `isEven(5)`? This initially looks like this:

```
bool isEven(int n) {
  if (n == 0) {
     return true;
  } else {
     return isEven(n - 2);
  }
}
```
int n 5

This call will go and call `isEven(3)`, as shown here:

```
bool isEven(int n) {
  bool isEven(int n) {
    if (n == 0) {
       return true;
    } else {
       return isEven(n - 2);
    }
  }
```
int n 3

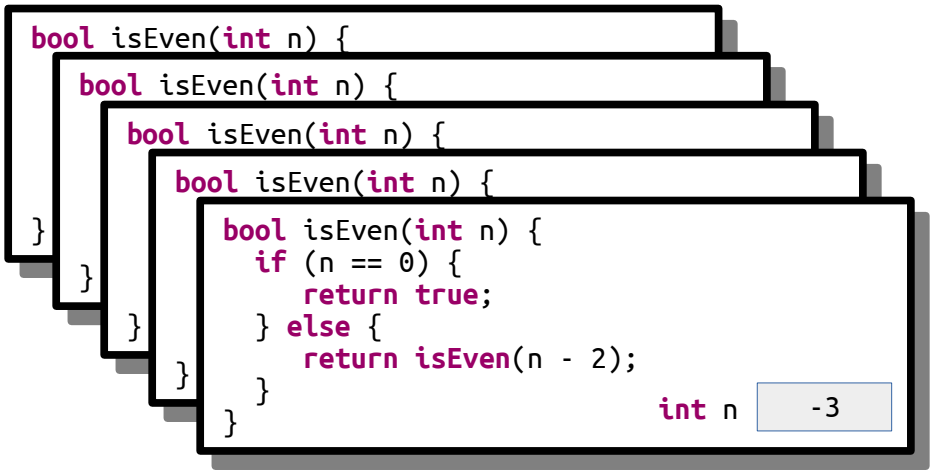
And that call then goes and calls `isEven(1)`:

```
bool isEven(int n) {
  bool isEven(int n) {
    bool isEven(int n) {
      if (n == 0) {
         return true;
      } else {
         return isEven(n - 2);
      }
    }
```
int n 1

And here's where things go wrong. This function now calls `isEven(-1)`, skipping over the base case:

```
bool isEven(int n) {
  bool isEven(int n) {
    bool isEven(int n) {
      bool isEven(int n) {
        if (n == 0) {
           return true;
        } else {
           return isEven(n - 2);
        }
      }
```
int n -1

This call then calls `isEven(-3)`:



And again we're off to the Stack Overflow Races because this is just going to keep getting more and more negative until we run out of call stack space.

It's important to know about stack overflows because as you start writing your own recursive functions you're likely to run into them in practice. When that happens, don't panic! Instead, pull out your debugger. Look at the call stack – it will be really, really full – and move up and down it, looking at the local variables. As you do, look at the arguments to the functions on the call stack. Are they repeating over and over again? Are they getting more and more negative or bigger and bigger? Based on what you find, you can make progress on debugging your code.

In this part of the assignment, we've included a recursive function that looks like this:

```
void triggerStackOverflow(int index) {
    triggerStackOverflow(kGotoTable[index]);
}
```

Here, `kGotoTable` is a giant (1024-element) array of the numbers 0 through 1023 that have been randomly permuted (see next week's section handout for how to do this!) This function looks up its argument in the table, then makes a recursive call using that argument. As a result, the series of recursive calls made is extremely hard to predict by hand, but since the recursion never stops (do you see why?) this code will always trigger a stack overflow.

Our starter code includes the option to call this function passing in 137 as an initial value. Run the provided starter code in debug mode (hey, you learned how to do that in Assignment 0!) and trigger the stack overflow. You'll get an error message that depends on your OS (it could be something like "segmentation fault," "access violation," "stack overflow," or something like that) and the debugger should pop up. Walk up and down the call stack and see if you can see the sequence of values passed in as parameters to `triggerStackOverflow`.

We've specifically crafted the numbers in `kGotoTable` so that the calls in `triggerStackOverflow` form a cycle. Specifically, `triggerStackOverflow(137)` calls `triggerStackOverflow(`x`)` for some number x, and that calls `triggerStackOverflow(`y`)` for some number y, and that calls `triggerStackOverflow(`z`)` for some number z, etc. until eventually there's some number w where `triggerStackOverflow(`w`)` calls `triggerStackOverflow(137)`, starting the cycle anew.

Your task in this assignment is to give us the sequence of the numbers in the cycle, starting with 137 and ending back up at 137. For example, if the sequence was

`triggerStackOverflow(137)` calls
`triggerStackOverflow(106)`, which calls
`triggerStackOverflow(271)`, which calls
`triggerStackOverflow(137)`, which calls
`triggerStackOverflow(106)`, which calls
`triggerStackOverflow(271)`, which calls
`triggerStackOverflow(137)`, which calls
`triggerStackOverflow(106)`, which calls
…

Then you would give us the cycle 137, 106, 271, 137.

Update the file comments at the top of the `WelcomeToCpp.cpp` file to include the sequence that you found, and, while you're there, why don't you go and update those comments to include your name, your section leader's name, and a description of the assignment?

Some notes on this part of the assignment:

- The topmost entry on the call stack might be corrupted and either not show a value or show the wrong number. Don't worry if that's the case – just move down an entry in the stack.
- Remember that if function *A* calls function *B*, then function *B* will appear higher on the call stack than function *A* because function *B* was called more recently than function *A*. Make sure you don't report the cycle in reverse order!

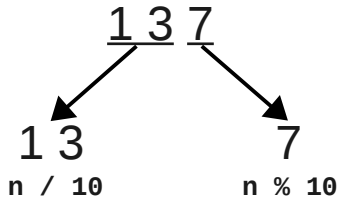
Problem Four: Implementing Numeric Conversions

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, calling `integerToString(1729)` will return the string `"1729"`. The second converts in the opposite direction, so calling `stringToInteger("-137")` will return the integer `-137`.

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, along the lines of what we saw in lecture. Specifically, by using division and modulus by ten, it's possible to split a number into its ones digit (`n % 10`) and all the remaining digits (`n / 10`), as shown here:



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 (the 1 in 137) rather than the last (the 7 in 137). 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 kinda defeats the point of the assignment. ☺

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

- When you get the last digit of a number *n* by writing `n % 10`, you're left with an *integer*, not a *character*. To convert from that numeric value to the character representing that digit, use the following syntax:

  ```
  char ch = char(digit + '0');
  ```

  You can similarly convert from a character representing a digit to the numeric value of that digit by going the other way:

  ```
  int val = ch - '0';
  ```

- If you find yourself needing to convert from a character to a string in C++, use this syntax:

  ```
  string str(1, 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, so calling `intToString(-137)` should return `"-137"` and calling `stringToInt("-137")` should return `-137`.
- Surprisingly, in C++, the behavior of the modulus operator on a negative number isn't consistent from computer to computer. As a result, you should not apply the `%` operator to negative numbers.
- 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 much easier than most other approaches.
- You do not need to worry about the case where the input to either function is invalid and therefore don't need to do any error-checking.

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 Five: The Flesch-Kincaid Grade-Level Test

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 thinks is necessary to understand that text. 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 15.1 would be appropriate for a typical college junior. 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 this formula:

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

Here, the constants C_0, C_1, and C_2 are 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!)

In this last part of the assignment, you'll create a program that lets you compute the Flesch-Kincaid grade level of all sorts of different pieces of text. Specifically, your job is to write a function

```
DocumentInfo statisticsFor(istream& source);
```

that takes as input a stream containing the contents of a file (described later), then returns a `DocumentInfo` object (described later) containing the number of sentences, words, and syllables contained in that file. Our provided starter code will call your function and use it to compute the Flesch-Kincaid score for the file. The rest of this section talks about what, specifically, you'll need to do in order to get this function to work.

So what exactly is this `DocumentInfo` thing you're returning? If you look in the starter code (look at `Headers/src/WelcomeToCpp.h`), you'll see that it's defined like this:

```
struct DocumentInfo {
    int numSentences;
    int numWords;
    int numSyllables;
};
```

This is a *structure*, a type representing a bunch of different objects all packaged together as one. Here, this structure type groups together three `int`s named `numSentences`, `numWords`, and `numSyllables`. The name `DocumentInfo` refers to a type, just like `int` or `string`. You can create variables of type `DocumentInfo` just as you can variables of any other type, like this:

```
DocumentInfo info;
```

One you have a variable of type `DocumentInfo`, you can access the constituent elements of the `struct` by using the dot operator. For example:

```
DocumentInfo info;
info.numSentences = 137;
info.numWords++;
cout << info.numSyllables << endl;
```

As a note, just as regular `int` variables in C++ are initialized to garbage values if you don't specify otherwise, the `int` variables inside of a `DocumentInfo` will get set to garbage if you don't initialize them. Therefore, you might want to initialize your variables like this:

```
DocumentInfo info = { 0, 0, 0 }; // Everything is zeroed out!
```

With that introduction to `struct`s done, let's talk about what exactly you need to get this function to do.

Step One: Tokenize the Input File

The `statisticsFor` function takes in an `istream&`, a reference to a stream. Streams are C++'s way of getting data from an external source (the keyboard, a file, the network, etc.), and in this case lets you read the contents of a file. You'll need to read in the original text and break it apart into a bunch of individual

words and punctuation symbols so that you can 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 how to use the `TokenScanner` class in lecture. However, we've provided a lot of documentation on the `TokenScanner` class on the course website (http://cs106b.stanford.edu) via the "Stanford C++ Library Docs" link. Learning how to read documentation is a critically important skill as a programmer, since you'll often find yourself working with new libraries! Read over the docs for the `TokenScanner` type. Take a look at the sample code at the top of the documentation to get a sense for how to use `TokenScanner` to read all the tokens out of a file. Find a way to get the `TokenScanner` to

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

(There's a note in the documentation that advises against using `TokenScanner`, but you should ignore that. Trust us – `TokenScanner` is a great fit here!)

We've provided a bunch of sample files and our own implementation's reference outputs (check the `Other Files/res` directory). As a first step, see if you can get the function to print out all the tokens in the input file, one line at a time. Don't worry about returning statistics yet – just make sure you're able to say what all the tokens are.

Before you move on to the next step, make sure that you're matching our tokens exactly. Start off by reading some of the shorter files and comparing the tokens you're getting to the tokens we're getting. For example, if you want to look at Obama's recent Farewell Address, look at `Obama-Farewell.txt`. Make sure that you're tokenizing `Obama's` as one token, not as the three tokens `Obama`, `'`, and `s`. Then compare your output against `Obama-Farewell.txt.analyzed` to make sure that you're getting the same tokens.

Step Two: Count Words and Sentences

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 should treat any token that starts with a letter as a word, so `apple` and `ISN'T` would both be considered words. As an approximation of the number of sentences, you can just count up the number of punctuation symbols that typically appear at the ends of sentences (namely, periods, exclamation points, and question marks). This isn't entirely accurate, but it's good enough for our purposes.

As an edge case, if a file doesn't appear to have any words or sentences in it, you should just pretend that it contains a single word and a single sentence (which prevents a division by zero error when evaluating the formula for Flesch-Kincaid readability). Once you've done that, have your program return the number of words and sentences it finds.

Before moving onward, why not take some time to test that your program works correctly? The starter code for this assignment is programmed so that if you try running your program on one of our provided sample files, it will automatically compare your numbers to our own and let you know if they don't match. Make sure that you're getting the same number of words and sentences for each of the files that we've provided. If you are, fantastic! You can move on to the next section. If not, see if you can track down what's causing the difference. A disparity indicates that you have a bug somewhere.

Step Three: Count Syllables in Words

Your last task is to count up how many syllables are in each of the words that you find. Systematically counting syllables in word is almost impossible, since syllable counts varies dramatically based on pronun-

ciation (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; the word `peach` would have one syllable; the word `deduce` would have two syllables, since the final e does not count as a syllable; the word `oboe` would have two syllables (although it ends in an e, that e is preceded by another vowel); the word `why` would have one syllable, since y counts as a vowel; and the word `enqueue` would have two syllables. Notice that under this definition, the word `me` would have zero syllables in it, since the final e by itself doesn't contribute to the total. To address this, ***you should always report that any given word has at least one syllable***, even if this heuristic would normally report otherwise.

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.

As a final step, check your numbers against our own. Again, if you try analyzing one of the files bundled with the project, our starter code will automatically compare your output against ours. If you’re matching our outputs for all the files, fantastic! You’ve probably got it working. If you aren’t, then you need to debug the program to see what’s up. We’ve included in the starter project annotated versions of the sample files that include our own counts for the number of syllables in each word. Do your program’s counts match our own? If not, see if you can figure out why. You might find the menu option to type in text and run your `statisticsFor` function useful here, since that way you can try testing out smaller words and phrases to see what comes back.

Once you've counted up the total number of words, sentences, and syllables, you’re all set, and all that’s left to do is play around with your program and see what you find! 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, *The Grapes of Wrath*, *Divergent*, or *Lean In*). How do those scores compare to our sample files? If you find anything interesting, let us know about it in your submission!

Some general notes and advice on this part of the assignment:

- Chapter 4 of the textbook goes into detail about stream classes and provides examples about how to read from a file. However, for this part of the assignment, the `TokenScanner` class is powerful enough to do all the file reading for you. Read the docs and see if you can find a way to have `TokenScanner` read directly from the stream. If you do so correctly, you should never need to do any explicit reading from the file; `TokenScanner` will do it for you.
- The `TokenScanner` type is a “one-pass” scanner. Once you’ve read all the tokens out of a file, you can’t “go back” and reread them again. If you create multiple `TokenScanner`s to read the same file, you still only get one pass over the file. This means that you can’t, for example, make two passes over the file, once counting sentences and once counting words and syllables.
- Watch out for capitalization when counting syllables. The strings `Unite`, `UNITE`, `unite`, `unItE`, and `UnitE` all should report two syllables. And don’t forget that `y` counts as a vowel. The word `unity` should have three syllables in it.
- If a file has no sentences, you should report that it has one sentence. If a file has no words, you should report that it has one word. However, if the file has no syllables (because there are no words), you should report that it has no syllables.
- If you try reading off the end of a string or before the beginning of a string, you trigger what C++ calls *undefined behavior*. This means that you can’t predict what will happen. Your program might crash and pull up a stack trace, or it might silently fail and return garbage data. Watch out for this when counting syllables.

(Optional) Problem Five: Extensions!

You are welcome to add extensions to your programs beyond what's required for the assignment, and if you do, we're happy to give extra credit! If you do, please submit two sets of source files – a set of originals that meet the specifications we've set out here, plus a set of modified files with whatever changes you'd like.

Here are some ideas to help get you 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?
- ***Combinations and Pascal's Triangle:*** There are a bunch of amazing properties buried in Pascal's triangle. For example, what happens if you print out Pascal's triangle, but first mod every number by two (that is, even numbers print as 0 and odd numbers print as 1). Notice anything interesting?
- ***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? Alternatively, could you get your program to convert from an English description of a number (say, "one hundred thirty seven") to a number or vice-versa? (That last one is a common job interview question!)
- ***Flesch-Kincaid Readability:*** Can you make the program better at counting syllables? Could you 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 are other readability indices like the *Dale-Chall readability score* that compute difficulty in different ways. Try implementing one of those and comparing the results. Which one works better?

  Alternatively, could you use your program to learn something about the human condition? In the past, we've had students use this program to look at the disparity between the reading level of US patents and the average education level of a jury after *voir dire*, look at the evolution of the English language by comparing older texts (Shakespeare, the US Constitution) to newer ones (Mark Zuckerberg's Facebook posts), observe changes in State of the Union addresses over the years, and even comment on the complexity of lyrics in rap music over time. If you find anything interesting, we'd love to see what you come up with!

Submission Instructions

To submit this assignment, submit the file `WelcomeToCpp.cpp` at http://paperless.stanford.edu. If you added in any extensions, feel free to submit any relevant files as well. You don't need to submit the `WelcomeToCppMain.cpp` file, though, since that's a part of the starter code and we've got plenty of copies of it. ☺

And that's it! You're done!

Good luck, and have fun!