Assignment 1: More C++!

Here it is – the first programming assignment of the semester! This assignment is designed to get you comfortable designing and building software in C++. It consists of four problems that collectively play around with control structures, string 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. Good luck, and have fun!

This assignment must be completed individually. Working in groups is not permitted.
Due Wednesday, February 20th at the start of class.

This assignment consists of four parts, of which three require writing code. This will be quite a lot to do if you start this assignment the night before it's due, but if you make slow and steady progress on this assignment each day you should be in great shape. Here's a recommended timetable:

- Aim to complete Stack Overflows the day this assignment is released. You don't need to write any code for it; it's just about working the debugger, something you practiced in Project00.
- Aim to complete the three recursion problems within five days of this assignment being released. I recommend that you spend a day each on Consonantize, Fair Play , and Concentrations, leaving the rest as buffer time. Recursion as a concept can take a bit of time to adjust to, and that's perfectly normal. Allocating some extra buffer time here will give you a chance to tinker around and ask for help if you need it.

As always, feel free to reach out to me if you have questions. Feel free to pose questions on Piazza (try to make the as specific as possible), or by email, or to stop by my office (MTWR, 9:20AM – 10:40AM in Math12).

1. **Stack Overflows**
   Whenever a program calls a function, the computer sets aside memory called a stack frame for that function call in a region called the call stack. Whenever a function is called, it creates a new stack frame, and whenever a function returns the memory space for that stack frame is recycled.

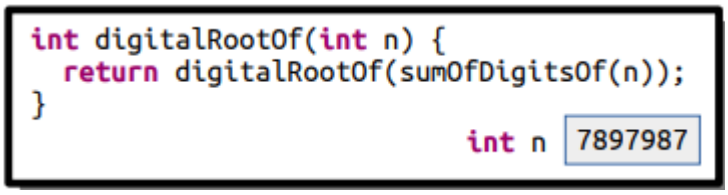
   As we saw in class, whenever a recursive function makes a recursive call, it creates a new stack frame, which in turn might make more stack frames, which in turn might make even more stack frames, etc. For example, when we computed `factorial(5)`, we ended up creating a net of six stack frames: one for 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 you called factorial on a very large number, say, `factorial(78979)`? This would create a *lot* of stack frames; specifically, 78,979 of them (one for `factorial(79879)`, one for `factorial(79879)`, ..., and one for `factorial(0)`). This is such a large number of stack frames that the call stack might not have space to store 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.

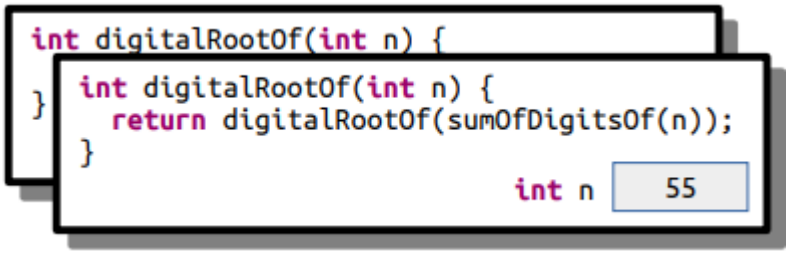
   In the previous example, a stack overflow occurs because we need a large but still finite number of stack frames. However, you often see stack overflows resulting due to programming errors. For example, consider the following (buggy!) implementation of the `digitalRootOf` function from Monday's lecture:

```
1 int digitalRootOf(int n) {
      return digitalRootOf(sumOfDigitsOf(n));
3 }
```

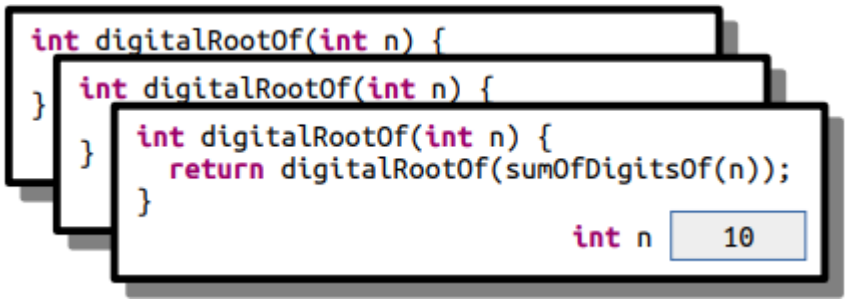
Imagine that you call `digitalRootOf(7897987)`. The initial stack frame looks like this:



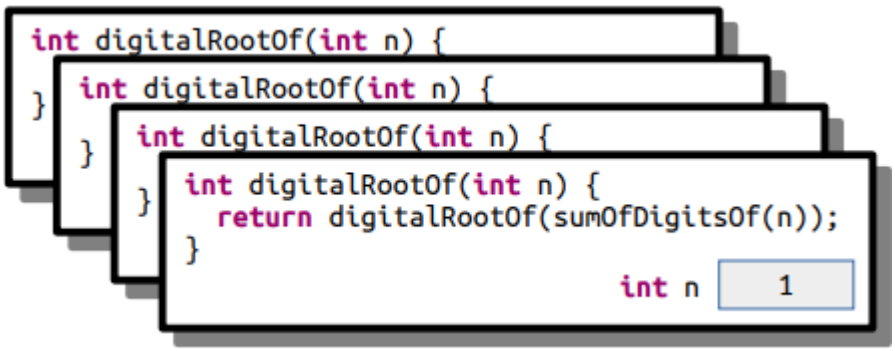
This call now tries to call `digitialRootOf(sumOfDigitsOf(7897987))`. The sum of the digits in the number is `7 + 8 + 9 + 7 + 9 + 8 + 7 = 55`, so this produces a call to `digitalRootOf(55)`:



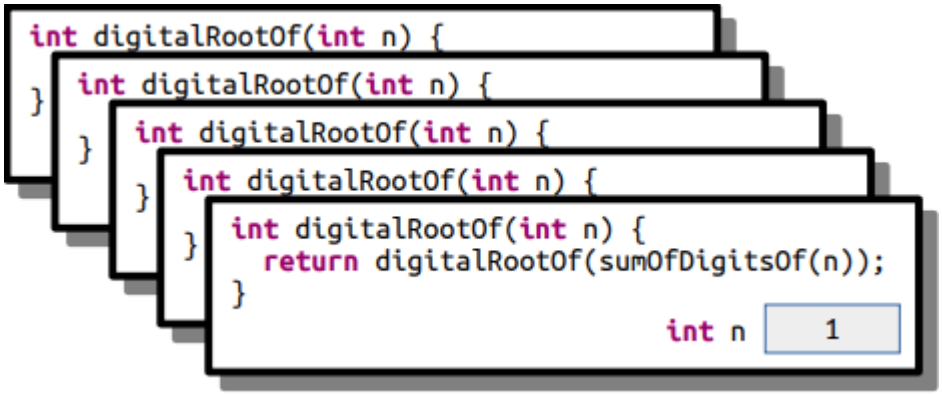
This now calls `digitalRootOf(sumOfDigitsOf(55))`, which leads to calling `digitalRootOf(10)`



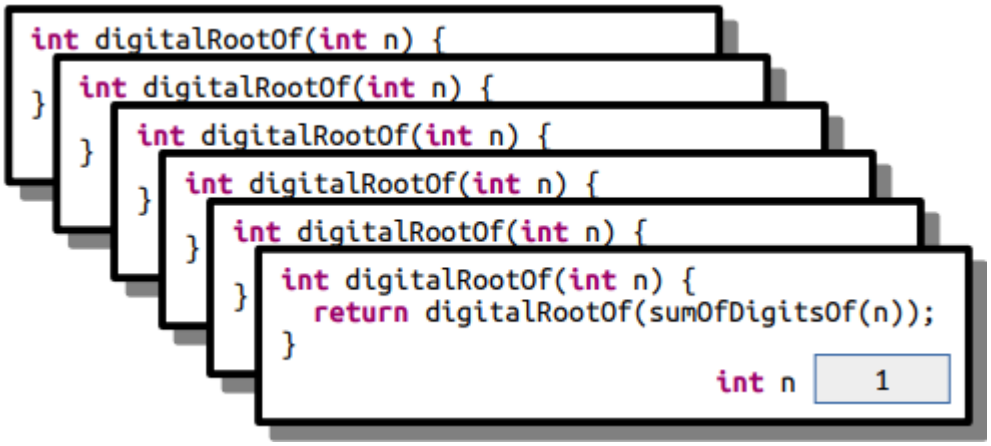
This now calls `digitalRootOf(sumOfDigitsOf(10))`, which calls `digitalRootOf(1)`:



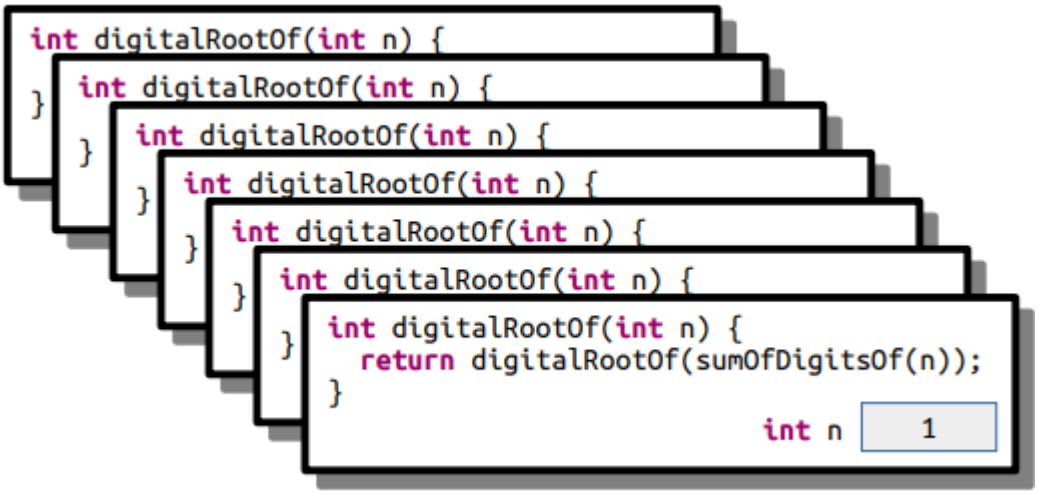
This then calls `digitalRootOf(sumOfDigitsOf(1))`, which calls `digitalRootOf(1)` again:



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



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

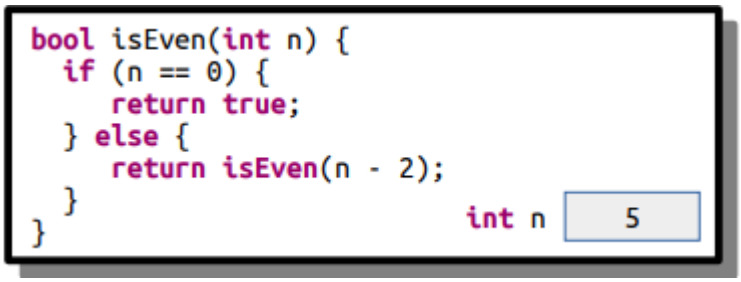


This recursion is off to endless recursion. It's like an infinite loop, but with function calls. This code will trigger a stack overflow because at some point it will exhaust the memory in the call stack. Another place you'll see stack overflows is when you have a recursive function that, for some reason, misses or skips over its base case. For example, let's suppose you want to write a function `isEven` that takes as input a number `n` and returns whether it's an even number. You note that 0 is even and, more generally, a number `n` is even precisely if `n - 2` is even. For example, 2 is even because

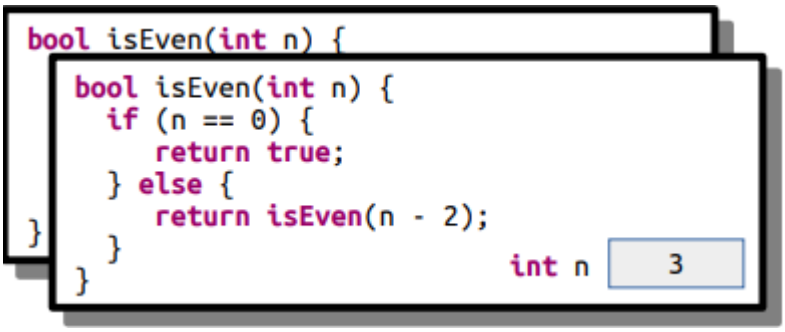
2 { 2 = 0 is even, 4 is even because 4 { 2 = 2 is even, and 6 is even because 6 { 2 = 4 is even, etc. Based on this (correct) insight, you decide to write this (incorrect) recursive function:

```
1 bool isEven(int n) {
      if (n == 0) {
3         return true;
      } else {
5         return isEven(n - 2);
      }
7 }
```

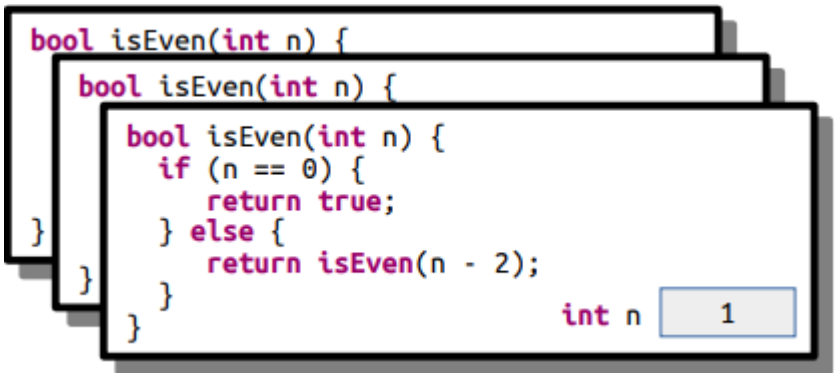
Now, what happens if you call `isEven(5)`? Initially we have:



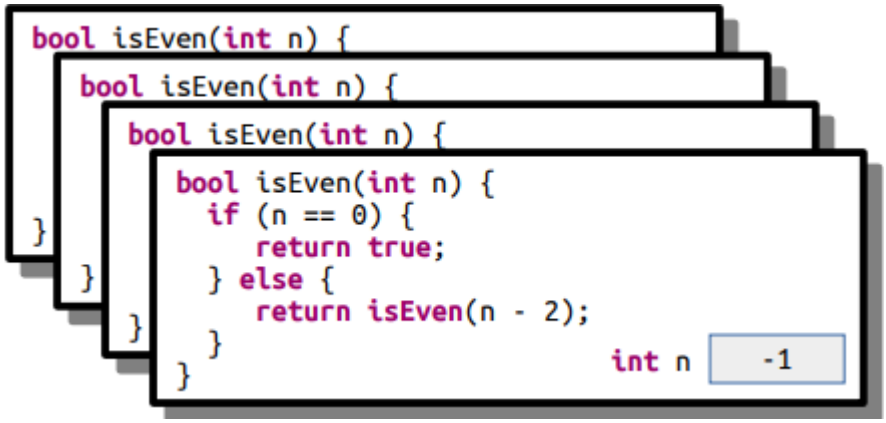
This call will then call `isEven(3)`:



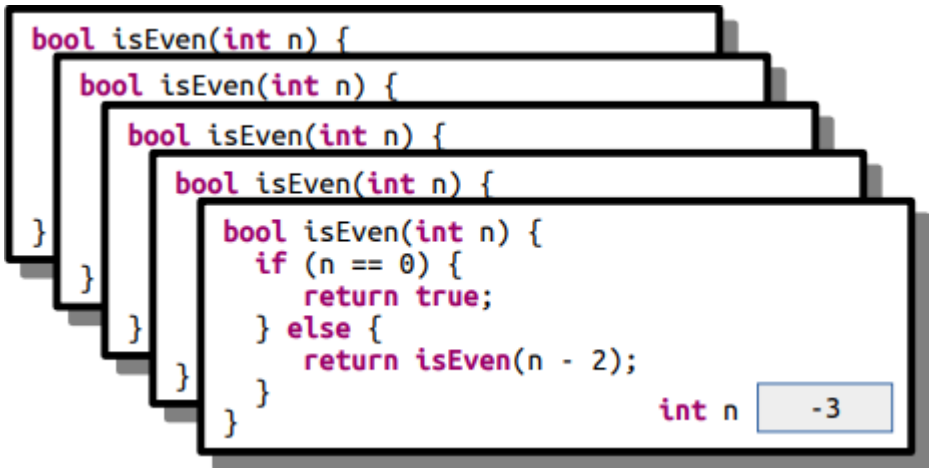
And that call then calls `isEven(1)`:



And it goes wrong. This function now calls `isEven(-1)`, skipping over the base case:



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



And again we're in and endless recursion leading to Stack Overflow.

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, use the 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, there's a recursive function that looks like this:

```
1 void tripStackOverflow(int index) {
      tripStackOverflow(nextTable[index]);
3 }
```

Here, `nextTable` is a (1024-element) array of the numbers 0 through 1023 that have been shuffled. 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 hard to predict by hand, and since the recursion never stops (why?) this code will always trigger a stack overflow.

The starter code includes the option to call this function passing in 137 as an initial value. Run the starter code in debug mode 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 `tripStackOverflow`.

The numbers in `nextTable` have been crafted so that the calls in `tripStackOverflow` form a cycle. Specifically, `tripStackOverflow(137)` calls `tripStackOverflow(x)` for some number `x`, and that calls `tripStackOverflow(y)` for some number `y`, and that calls `tripStackOverflow(z)` for some number `z`, etc. until eventually there's some number `w` where `tripStackOverflow(w)` calls `tripStackOverflow(137)`, starting the cycle anew.

Your task in this part of the assignment is to find the sequence of the numbers in the cycle. For example, if the sequence was

```
tripStackOverflow(137) calls
tripStackOverflow(106), which calls
tripStackOverflow(271), which calls
tripStackOverflow(137), which calls
tripStackOverflow(106), which calls
tripStackOverflow(271), which calls
tripStackOverflow(137), which calls
tripStackOverflow(106), which calls
```

Then you would report the cycle 137, 106, 271, 137.

Update the file comments at the top of `StackOverflow.cpp` to list the cycle you found.

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!

2. Consonantize

Here's a puzzle: can you identify these movie titles, given that all characters except consonants have been deleted?

```
BTNDTHBST MN CTCHMFCN SRSMN
```

Answer 1 *The first is "Beauty and the Beast," the second is "Moana," the third is "Catch Me If You Can," and the last one is an exercise to the reader.* ☺

To form a puzzle string like this, you simply delete all the letters from the original word or phrase except for consonants, then convert the remaining letters to ALL-CAPS. Your task is to write a recursive function `string consonantize(string phrase);` that takes as input a string, then transforms it into an Only Consonants puzzle. For example:

- `consonantize("Donald Knuth")` returns `"DNLDKNTH"`,
- `consonantize("Ada Lovelace")` returns `"DLVLC"`,

- `consonantize("IAEA 1957")` returns `""`,
- `consonantize("For sale: Starting Out With C++, never used.")` returns `"FRSLSTRTNGTWTHCNVRSD"`,
- consonantize("consonantize(consonantize)") returns `"CNSNNTZCNSNNTZ"`, and
- `consonantize("Annie Mae, My Sea Anemone Enemy!")` returns `"NNMMSNMNNM"`.

Some notes on this problem:

- Your solution must be recursive. You may not use loops (`while`, `for`, `do...while`, or `goto`).
- Make sure that you're always returning a value from your recursive function. It's easy to accidentally forget to do this when you're getting started with recursion.
- You can use `toupper` from the `<cctype>` header to convert a single character to upper case (or write your own). It takes in a `char`, then returns the upper-case version of that letter. If you call `toupper` on a non-letter character like `'$'` or `'*'`, it will return the original character unmodified.
- The `isalpha` function from the `<cctype>` header takes in a character and returns whether it's a letter. There is no library function that checks if a letter is a consonant, though.
- Just to make sure you didn't miss this, we are treating the letter `y` as a vowel.

Before turning in your code, do some "check/expect" tests (for the examples given, say) to make sure your code is robust. When making your tests, be strategic: What are some tricky cases you might want to confirm work correctly? Are there any edge cases (extremely small cases, highly unusual cases, etc.) that would be worth testing?

Once you have everything working, leave an `Only Consonants` puzzle of your own choosing in Piazza! All the code you'll need to send me (via email) for this part of the assignment should go in the `consonantize.cpp` file.

3. Fair Play

Consider the following scenarios:

- Ten people want to play pick-up basketball. They select one person to captain team `A` and one to captain team `B`. The two captains then take turns choosing players for their team. One option would be to alternate between captains `A` and `B`, but that would give the team that picked first a noticeable advantage over the other. In what order should the captains pick players?
- The World Chess Championship is a multi-game chess match held every two years between the reigning world champion and a challenger. In chess, white has a slight advantage over black, so both players should have an equal number of turns as white and as black. However, if one player gets too many turns as white early on, they might accumulate a score advantage early on that puts pressure on the other player. In what order should the players play as white and black?
- In old-school NFL rules, if a postseason football game went to overtime, one team would get possession of the ball and be given a chance to score. If they scored, they'd instantly win the game. If they didn't, the other team would get possession and a chance to score. If the other team didn't score, then the first team would get another try, etc. This gives an advantage to whoever gets possession first. What's a better way to decide which team gets a chance to score to make this as fair as possible?

These scenarios all have a core setup in common. There are two parties (call them A and B) who take turns at an activity that confers an advantage to whoever performs it. The goal is to determine the order in which A and B should perform that activity so as to make it as close to fair as possible.

There's a clever recursive technique for addressing this problem that keeps the advantage of going first to a minimum. We're going to consider two kinds of sequences: A–sequences and B–sequences, which each give a slight advantage to players A and B, respectively.

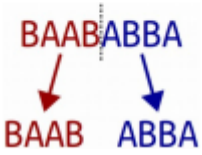
There are different A–sequences and B–sequences of different lengths. Each sequence is given a number called its order. The higher the order of the sequence, the more games are played. For example, here are the A and B sequences of orders 0, 1, 2, 3, and 4:

	Order 0	*Order* 1	*Order* 2	*Order* 3	*Order* 4
A-Sequence	**A**	**AB**	**ABBA**	**ABBABAAB**	**ABBABAABBAABABBA**
B-Sequence	**B**	**BA**	**BAAB**	**BAABABBA**	**BAABABBAABBABAAB**

We can interpret these sequences as instructions of who gets to play when. For example, the A–sequence of order 2, ABBA, can be thought of as saying "A plays first, then B, then B again, then A again." There's a slight advantage to A going first, but it's mitigated because B gets two turns in a row. The B–sequence of order three, BAABABBA, means "B takes a turn, then A, then A again, then B, then A, then B, then B again, then A." If you think about what this looks like in practice, it means that B has a little advantage from going first, but the other alternations ensure that A gets to recoup that disadvantage later on.

Notice the nice pattern here. Take the A–sequence of order 4, **ABBABAABBAABABBA**, and split it in half down the middle, as shown to the right. That gives back the two sequences **ABBABAAB** and **BAABABBA**. If we look in the table shown above, we can see that this first sequence is the A–sequence of order 3, and the second sequence is the B–sequence of order 3. Interesting!

Similarly, look at what happens when you split the B–sequence of order 3, **BAABABBA**, in half. That gives back **BAAB** and **ABBA**. And again, if we look in our table, we see that this first string is the B–sequence of order 2 and the second is the A–sequence of order 2. Neat!



More generally, the pattern goes like this: if you take an A–sequence of order n (where $n > 0$) and split it in half, you get back an A–sequence of order $n - 1$ followed by a B–sequence of order $n - 1$. Similarly, if you take a B–sequence of order n (with $n > 0$) and split it in half, you get a B–sequence of order $n - 1$ followed by an A–sequence of order $n - 1$. This process stops when you need to form the A–sequence or B–sequence of order 0, which are just A and B, respectively.

Using these observations, implement a pair of functions

```
string aSequence(int n);
string bSequence(int n);
```

that take as input a number $n \geq 0$, then return the A–sequence and B–sequence of order n, respectively. As with the `Only Consonants` problem, these functions must be recursive and must not use loops.

If someone calls either function passing in a negative value of n, you should use the `error()` function to report an error. That function has the signature

```
void error(string message);
```

and, when called, immediately jumps out of the function to say that something terrible has happened.

I've included some check/expect values you can use to check your solution, but those tests aren't

exhaustive. As part of this assignment, add at least one custom test case of your own, and ideally more. All the code you'll need to write for this part of the assignment should go in `fairPlay.cpp`.

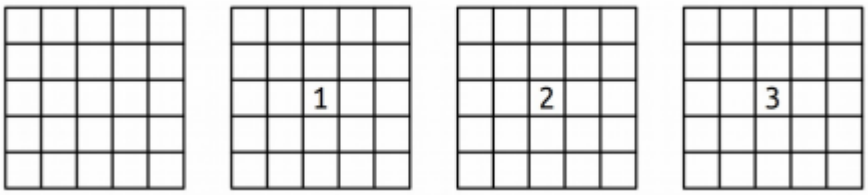
Ther's another interesting property of these sequences. Imagine you're in an open field. You read the characters of an `A`–sequence from left to right. Whenever you read an A, you take a step forward, place down a marker, then rotate 60°. Every time you read a `B`, you turn around without moving. Repeating this process gives an intricate and complex result. Can you figure it out?

Some notes on this problem:

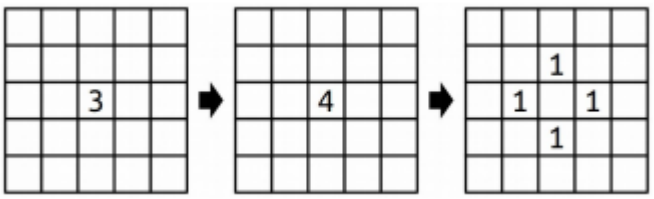
- As with most recursive functions, you probably won't need to write much total code here. If you've written thirty or more lines of code and are struggling to get things working, you may be missing a slightly easier solution route and might want to check in with me for advice.
- An `A`–sequence of order `n` has length `2n`, which means that the lengths of these sequences grow extremely rapidly as a function of `n`. For reference, an order-30 `A`–sequence will be over a billion characters long, and an order-300 `A`–sequence has more characters than there are atoms in the observable universe. Don't worry about generating sequences of those sorts of orders; stick with low-order sequences, say, with $n \leq 20$.
- If your code isn't working, step through it in the debugger! You saw how to step through recursive code in Part One of this project and in Project00.
- Testing is key here. The tests I've provided aren't sufficient to cover all the cases that might come up. You're required to add at least one test case, and really do take that seriously. If you're trying to test something recursive, what sorts of cases might be good to check?

4. Concentrations

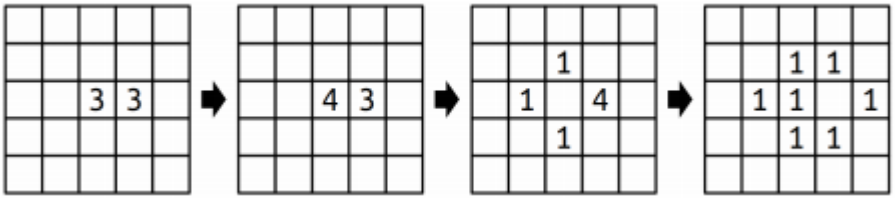
One of the most beautiful ideas in mathematics is that simple rules give rise to complex behavior. You can have straightforward principles that govern how a particular system operates, and yet discover that repeated applications of those rules gives intricate and surprising patterns. The best way to appreciate this is to see it for yourself. This problem concerns a mathematical model of a concentration. A concentration as follows. We begin with a two-dimensional grid subdivided into units called cells. Each cell is initially empty. We then pick a cell and start growing its concentration. Each cell is allowed to hold between 0 and 3 units. So, for example, if we had a 5×5 grid and started adding units into the center square, the first few steps would look like this:



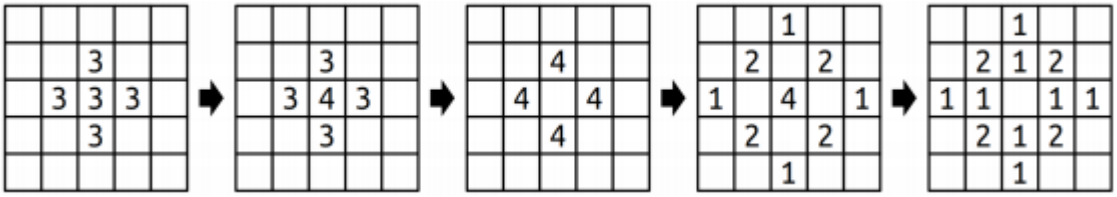
What happens when we drop a fourth grain of sand in the center? Since each cell can hold between 0 and 3 grains of sand, we can't have four grains of sand there. In this case, the concentration diffuses, sending units to each of the four neighbors in the cardinal directions (up, down, left, and right), ending up empty. This is shown here:



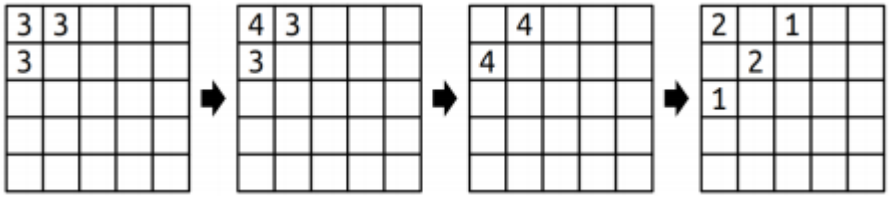
That might be the end of the story, or it might just be the beginning. Imagine, for example, that there are two adjacent cells, each of which contains three units. What happens when a unit is dropped into one of those full cells? That initial cell diffuses, sending a unit in each direction. One of those units lands inside another cell holding three units, boosting it up to four units, causing that cell to diffuse. This is illustrated below:



A single unit can cause a large cascade. For example, here's what happens if five full cells in a plus shape are disturbed by one unit:



There's one additional case to consider, and that's what happens at the border of the world. If a unit falls outside of the the boundaries of the grid, it just falls out and is never seen again. For example, here's what happens if a cell in the corner were diffuse:



If you start with an empty world and repeatedly drop units into the center square, the resulting distribution patterns are surprising. But that's something that you'll need to see for yourself. 😊

Before moving on, take a few minutes to answer the following question: what will the world to the right look like if a single unit is dropped onto the center square? The chain reaction here is quite interesting – work this out with a pencil and paper!

3	3	2
2	3	3
	2	3

Your job is to implement a ***recursive*** function.

```
void dropUnits(Grid<int>& world, int row, int col);
```

that takes as input a row and a column index, along with a `Grid<int>` representing the number of grains of sand in each cell (more on this later on), then adds a single grain of sand to the specified location. As a reminder, here's what happens when you drop sand on a cell:

- If the cell ends up with three or fewer units in it, you're done.
- Otherwise, empty that cell, then drop one unit on each of the four neighbors.

You might have noticed that the first argument is a `Grid<int>`, a type we haven't encountered before. This is a type representing a fixed-size, two-dimensional grid of integers, like the grids that we showed on the previous pages. You can access and manipulate the individual cells using statements like these:

```
world[row][col] += 137;
if (world[row][col] == 0) { ... }
```

Row and column indices start at zero, and row 0, column 0 is in the upper-left corner of the screen. The rows increase from top to bottom and columns increase from left to right.

You may be given coordinates in `dropUnits` that are outside the boundaries of the grid, either because whoever is giving you the coordinates just wants to drop sand on the floor or because units diffused off the edge. If that happens, dropUnits should have no effect and should leave the grid unmodified. You can test whether a coordinate is in bound by using the grid's inBounds() member function:

```
if (world.inBounds(row, col)) {...}
```

Make sure not to read or write to a grid in an out-of-bounds location. Doing so may cause the program to crash with an message.

Some notes on this problem:

- As before, your solution should be purely recursive and should not involve any loops.
- Avoid helper functions here. It's actually easiest to write this as a single function that drops a grain of sand and handles diffusion rather than splitting those functions apart.
- When a cell diffuses, make sure to empty that cell out before you start dropping units on other squares. Otherwise, weird things can happen when a unit falls back into that cell.
- It actually doesn't matter what order you visit the neighbors of a diffusing cell when that cell diffuses. The result is always the same. (This isn't supposed to be obvious, by the way; it's something that requires a creative mathematical proof.)

As before, you are required to add some of your own test cases. Take this seriously! It can be hard to tell whether you've gotten your solution working purely by looking at the output because the system evolves in such amazing and complex ways. Writing your own targeted, focused tests is a great way to make sure that what you think is going to happen matches what actually happens.

All the code you need to write for this part of the assignment goes in the `concentrations.cpp` file.