CS106B
Winter 2019

Handout 09
January 18, 2019

Assignment 2: Fun with Collections

This assignment is all about the amazing things you can do with collections. It's a two-parter. The first exercise explores a mathematical model of crystal formation, and the second is a devious twist on the classic word game *Hangman*. By the time you've completed this assignment, you'll have a much better handle on the container classes and how to use different types like queues, maps, vectors, and sets to model and solve problems. Plus, you'll have some things we think you'd love to share with your friends and family.

Due Monday, January 28th at the start of class.

This assignment must be completed individually.
Working in pairs is not permitted.

At a high level, this assignment consists of two parts:

- ***Crystals:*** With a combination of `Set`s and `Queue`s, you can simulate the growth of a crystal, along the lines of what happens when snowflakes form or when you're making rock candy. This particular coding assignment is mostly about getting used to the different collection types in C++ and their nuances.
- ***Evil Hangman:*** *Hangman* is a children's game that assumes one of the players is being honest. But if one of the players is dishonest and can perform a billion operations per second, hilarity ensues.

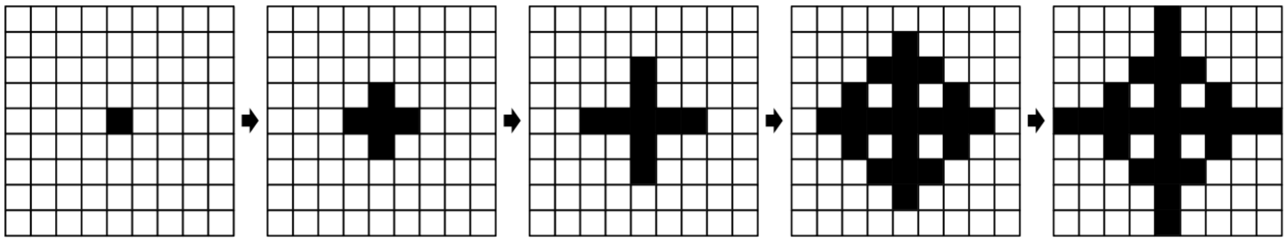
We recommend that you start this assignment early and make slow and steady progress on it throughout the ten days that you have to complete it. For example, we recommend trying to get the Crystals assignment completed within three days of this assignment going out, leaving the remaining time to work on Evil Hangman.

Depending on when you took CS106A, you may have implemented *Hangman* as one of your assignments. Despite the superficial similarity to *Hangman*, the Evil Hangman assignment is *fundamentally different*. You're unlikely to be able to reuse any code from CS106A in the course of solving it, and you should expect to put in more time and effort into coding up Evil Hangman than you did coding up *Hangman*. That being said, chances are that when you've finished Evil Hangman, you're going to be pretty impressed by what you've built!

Part One: Crystals

In the early 1960s, physicist and mathematician Stanisław Ulam devised a simple set of rules that produced patterns similar to how crystals form (either in nature as snowflakes or in the lab in a recrystallization). Ulam published a paper outlining his work in 1962, with a note that his insights were heavily influenced by computer experiments he conducted at the Los Alamos Scientific Laboratory.

Ulam's system works as follows. We imagine that the world is an infinite two-dimensional grid stretching off in all directions, where each cell in the grid is either empty or full. We begin by filling one of the cells somewhere in the grid. This is the first generation of the crystal. From that point forward, each successive generation is defined as follows: retaining all the cells from the previous generation, fill in each location adjacent to *exactly one* of the squares from the previous generation, where we only consider adjacency in the four cardinal directions (up, down, left, and right). The first few iterations of Ulam's crystal are shown below:



If you simulate this process for longer and longer periods of time, you'll see intricate, repeated patterns that appear at different levels of scale. It's quite beautiful to watch this process unfold!

The question then comes up of how exactly to implement this in code. One way we could do this would be to use a `Grid<bool>`, where each location is `true` if the cell is occupied and `false` otherwise. We could then repeatedly scan over the grid, finding locations that have exactly one neighbor active and including them in the next generation. While this approach would work, it has a couple of problems:

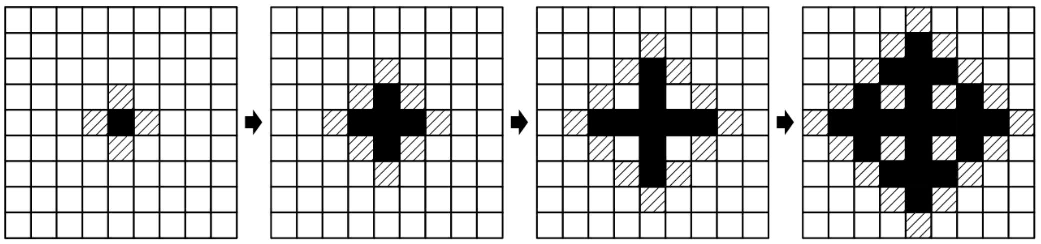
- The `Grid` type has a fixed size. If we make the grid too small, then the pattern might get cut off right as it's getting interesting. If we make the grid too big, most of the space will be wasted.
- It's extremely rare for a grid location to change from false to true. This means that if we keep iterating over the entire grid looking for places to make updates, most of the work we'll be doing will be wasted effort.

We'll address each of these issues in different ways. First, rather than storing the state of the world as a `Grid<bool>`, we'll represent the world using a `HashSet` (basically, a faster version of the `Set` type from lecture) that holds all of the (x, y) coordinates of the cells in the crystal. This allows the crystal to grow without bound in each direction, since `Set` and `HashSet` don't have fixed sizes.

The next question is how to handle the fact that between generations of the crystal growth, only a few spots in the world are "interesting." That is, there are only a few spots on the crystal where new cells might get added in. We could try scanning over all the locations in the crystal looking for new places to add cells in, but that would focus most of the effort on older sites in the crystal that can't possibly spawn any new ones.

To address this, we'll introduce the idea of ***active sites***. We know that, from one generation to the next, the only places where new cells can be added into the crystal are places that are adjacent to a cell in the crystal. But more importantly, those sites have to be adjacent to a location that was added in the immediately preceding generation of the crystal growth. Therefore, as we're growing our crystal, we'll keep track of a separate list of locations indicating all the possible places where the crystal could grow in the next generation. We'll then only check those locations when computing the next generation.

To see how this might work in practice, here's a second trace of the growth of the crystal. The cells in white are empty, the cells in black are part of the crystal, and the hatched cells represent active sites that need to be considered for growth in the next generation. As you can see, some of those hatched cells do indeed get added into the crystal, while others end up being excluded because they have more than one neighbor already incorporated into the crystal:



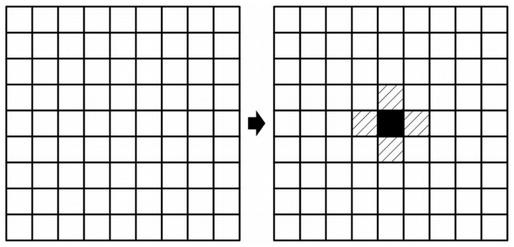
The general rule here is that whenever you add a cell into the crystal, you add all of its empty neighbors into the list of active sites, since those sites might potentially end up becoming a part of the crystal as well. This explains why, in going from the third step to the fourth step, some of the cells inside the empty space of the existing crystal get added into the list of active sites: those cells are empty and they're next to newly-added spots.

The Assignment

Your task is to implement the following pair of functions and to provide test coverage for those functions. Your first function is responsible for adding another cell into the crystal at a given location, updating the list of active sites in the process:

```
void crystallizeAt(Crystal& crystal, int x, int y);
```

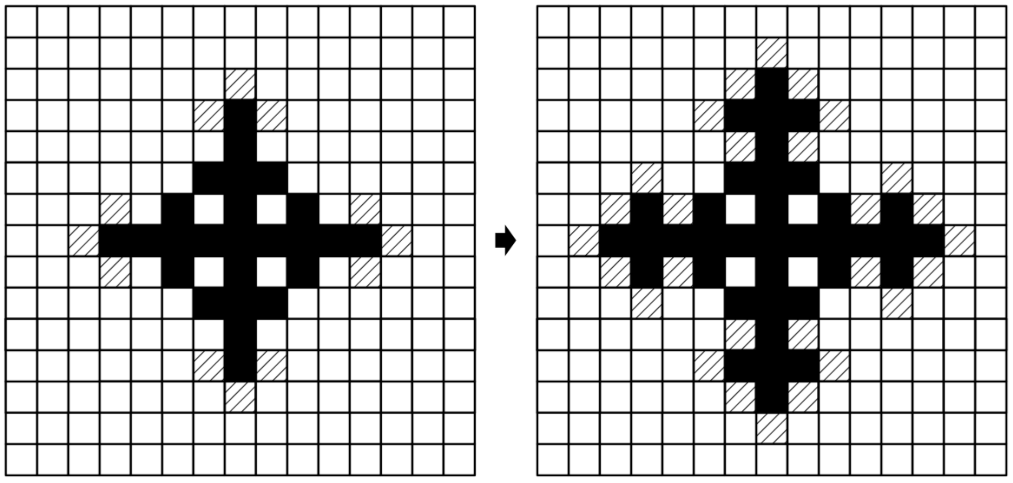
For example, assuming that *c* represents the (empty) crystal on the left side of the diagram below and that (0, 0) is the center square, calling `crystallizeAt(c, 0, 0)` would have the following effect:



The second function is responsible for running one step of the crystal generation process. Here, one "step" means advancing from one generation to the next. It has this signature:

```
void step(Crystal& crystal);
```

For example, here's a before-and-after diagram showing the effect of this function:



The `Crystal` type described here is a `struct` containing two data members. The first is a `HashSet` holding `Point`s, where each `Point` is just an (*x*, *y*) pair. Check the documentation on the CS106B website for more information about it. The second is a `Queue<Point>` holding a list of all the active sites that need to be visited in the next generation.

Why store the list of active sites in a `Queue`, you might ask? Turns out `Queue`s are great for representing to-do lists, since the operations they support are enqueue ("I'll deal with this once I've finished handling everything else I need to do") and dequeue ("let me take care of the task that's been waiting here the longest").

The tricky part here is implementing `crystallizeAt` and `step` so that they keep the `HashSet` and `Queue` in sync with one another. You'll need to guarantee that after each `step`, you've processed everything that used to be in the `Queue` while also priming it with all the locations that need to be considered in the future.

The provided starter files contain both a testing harness along the lines of what you saw in the first programming assignment and a visualizer that will run your code through lots and lots of generations so that you can see what structure emerges from Ulam's crystal model.

Here are a few general pieces of advice for this part of the assignment:

- ***Make sure you understand the relevant data structures***. You don't need to write all that much code for this part of the assignment. However, the code you write will have to make use the `Queue` and `HashSet` types. Before you start crafting your program, take some time to read over what they do and review the relevant parts of the textbook (Chapter 5) and lecture materials, and feel free to hop on Piazza or stop by the LaIR with questions!
- ***Recursion isn't necessary here***. The first programming assignment of the quarter really stressed recursion, and while you're welcome to use it here, this particular problem isn't super well-suited to a recursive solution. Don't worry if you find yourself writing a lot of iterative code – that's perfectly normal.

  Pro tip: it's quite rare to see `Queue`s thrown around in recursive code. The way that a `Queue` processes tasks is quite different from how recursion processes them (recursive calls typically are best modeled by a `Stack` – can you explain why?), and in this particular assignment using recursion is likely going to get you intro trouble.
- You can assume that the location passed in to `crystallizeAt` is a position that hasn't yet crystalized. Using the visual lexicon of the preceding diagrams, `crystallizeAt` will never be called on a black square.
- You can add position (x, y) to a `HashSet<Point>` by writing

  mySet`.add({`*x*`, `*y*`});`
- Because we're using a `HashSet<Point>` rather than a `Grid`, you don't need to do any bounds-checking in this assignment. The crystal can grow infinitely in all directions.
- When you crystallize at a given location, some of the neighbors of that location might already be in the crystal (marked black). You'll need to think about how you want to handle this. Will you add those locations to the `Queue`, and then filter them out later? Or not add them to the `Queue` in the first place? Similarly, when expanding out the crystal, you might find that two cells each want to add the same neighbor to the list of active sites. Will you add both sites, then filter them out when expanding the crystal a second time? Or will you avoid adding the same active site twice?
- You shouldn't find yourself needing to write much code to implement this – our solution checks in at under a hundred lines of code, excluding test cases. If you find yourself writing significantly more than this and you're having trouble getting things working, it might mean that you've missed an easier way to go about solving this problem.

All the code you need to write goes in the `Crystals.cpp` file. As with Assignment 1, we've included a few very basic test cases, which we strongly encourage you to expand as you develop your code. Like Sandpiles from Assignment 1, it can be very difficult to "eyeball" the crystal and check whether it's evolving correctly. Writing tests forces you to articulate "here is what I believe is supposed to happen" and lets you confirm that the system is indeed working as expected.

Part Two: Evil Hangman

It's hard to write computer programs to play games. When we humans sit down to play games, we can draw on past experience, adapt to our opponents' strategies, and learn from our mistakes. Computers, on the other hand, blindly follow a preset algorithm that (hopefully) causes it to act intelligently. Though computers have bested their human masters in some games (including, recently, Go), the programs that do so often draw on hundreds of years of human experience and use extraordinarily complex algorithms and optimizations to outcalculate their opponents.

While there are many viable strategies for building competitive computer game players, there is one approach that has been fairly neglected in modern research – cheating. Why spend all the effort trying to teach a computer the nuances of strategy when you can simply write a program that plays dirty and wins handily all the time? In this assignment, you will build a mischievous program that bends the rules of *Hangman* to trounce its human opponent time and time again. In doing so, you'll cement your skills with the container types and will hone your general programming savvy. Plus, you'll end up with a highly entertaining piece of software, at least from your perspective. 😃

In case you aren't familiar with the game *Hangman*, the rules are as follows:

1. One player chooses a secret word, then writes out a number of dashes equal to the word length.
2. The other player begins guessing letters. Whenever she guesses a letter in the hidden word, the first player reveals each instance of that letter in the word. Otherwise, the guess is wrong.
3. The game ends when all letters in the word have been revealed or when the guesser has made a certain number of incorrect guesses.

Fundamental to the game is the fact the first player accurately represents the word she has chosen. That way, when the other players guess letters, the first player can reveal whether that letter is in the word. But what happens if the player doesn't do this? This gives the player who chooses the hidden word an enormous advantage. For example, suppose that you're the player trying to guess the word, and at some point you end up revealing letters until you arrive at this point with only one guess remaining:

`H E A R -`

There are only three words in the English language that match this pattern: `HEARD`, `HEARS`, and `HEART`. If the player who chose the hidden word is playing fairly, then you have a one-in-three chance of winning this game if you guess `D`, `S`, or `T` as the missing letter. However, if your opponent is cheating and hasn't actually committed to either word, then there is no possible way you can win this game. No matter what letter you guess, your opponent can claim that she had picked a different word, say that your guess is incorrect, and win the game. That is, if you guess that the word is `HEARD`, she can pretend that she committed to `HEARS` or `HEART` the whole time, and there's no way you could know this wasn't the case.

Let's illustrate this technique with an example. Suppose that you are playing Hangman and it's your turn to choose a word, which we'll assume is of length four. Rather than committing to a secret word, you instead compile a list of every four-letter word in the English language. For simplicity, let's assume that English only has a few four-letter words, all of which are reprinted here:

`ALLY    BETA    COOL    DEAL    ELSE    FLEW    GOOD    HOPE    IBEX`

Now, suppose that your opponent guesses the letter 'E.' You now need to tell your opponent which letters in the word you've "picked" are E's. Of course, you haven't picked a word, and so you have multiple options about where you reveal the E's. Here's the above word list, with E's highlighted in each word:

ALLY   B**E**TA   COOL   D**E**AL   **E**LS**E**   FL**E**W   GOOD   HOP**E**   IB**E**X

Now, suppose that your opponent guesses the letter 'E.' If you'll notice, every word in your word list falls into one of five "word families:"

- `----`, which contains the words `ALLY`, `COOL`, and `GOOD`.
- `-E--`, containing `BETA` and `DEAL`.
- `--E-`, containing `FLEW` and `IBEX`.
- `E--E`, containing `ELSE`.
- `---E`, containing `HOPE`.

Since the letters you reveal have to correspond to some word in your word list, you can choose to reveal any one of the above five families. There are many ways to pick which family to reveal – perhaps you want to steer your opponent toward a smaller family with more obscure words, or toward a larger family in the hopes of keeping your options open. For this assignment, in the interests of simplicity, we'll adopt the latter approach and always choose the largest of the remaining word families. In this case, it means that you should pick the family `----`. This reduces your word list down to

```
ALLY    COOL    GOOD
```

and, since you didn't reveal any letters, you would tell your opponent that his guess was wrong.

Let's see a few more examples of this strategy. Given this three-word word list, if your opponent guesses the letter O, then you would break your word list down into two families:

- `-OO-`, containing `COOL` and `GOOD`.
- `----`, containing `ALLY`.

The first of these families is larger than the second, and so you choose it, revealing two O's in the word and reducing your list down to

```
COOL    GOOD
```

But what happens if your opponent guesses a letter that doesn't appear anywhere in your word list? For example, what happens if your opponent now guesses 'T'? This isn't a problem. If you try splitting these words apart into word families, you'll find that there's only one family: the family `----` containing both `COOL` and `GOOD`. Since there is only one word family, it's trivially the largest, and by picking it you'd maintain the word list you already had.

There are two possible outcomes of this game. First, your opponent might be smart enough to pare the word list down to one word and then guess what that word is. In this case, you should congratulate her – that's an impressive feat considering the scheming you were up to! Second, and by far the most common case, your opponent will be completely stumped and will run out of guesses. When this happens, you can pick any word you'd like from your list and say it's the word that you had chosen all along. The beauty of this setup is that your opponent will have no way of knowing that you were dodging guesses the whole time – it looks like you simply picked an unusual word and stuck with it the whole way.

We’ve posted a few sample runs of this program in action up on the course website:

- This example guesses each letter in the order in which they appear on a QWERTY keyboard, and still manages to lose.
- This sample gets trounced twice on the same sequence of inputs, being told that the hidden word was different in both cases.
- This one shows a frustrated user typing in lots of invalid inputs, eventually managing to guess the hidden word. Turns out that this algorithm doesn’t help much when there aren’t many words of the given length.
- This run shows someone getting trounced with a three-letter word.

As you can see, it is ***extremely*** difficult to pull off a victory, though it is indeed possible!

The Assignment

Your assignment is to write a computer program which plays a game of Hangman using this “Evil Hangman” algorithm. In particular, your program should do the following:

1. ***Set up the game***. Prompt the user for a word length, prompting as necessary until she enters a number where there's at least one word exactly that long. Then, build a list of all words whose length matches the input length. We’ve provided you with the file `EnglishWords.txt`, which contains a gigantic list of English words; you’ll need to decide how to store that list. Check Chapter 4 of the textbook for more information about how to read data from files.

   Next, prompt the user for a number of guesses, which must be an integer greater than zero. Don't worry about unusually large numbers of guesses – after all, having more than 26 guesses is not going to help your opponent!

   Finally, prompt the user for whether she wants to have a running total of the number of words remaining in the word list. This completely ruins the illusion of a fair game that you'll be cultivating, but it's quite useful for testing.

   Please prompt the user for this information in the order specified above. Part of our grading is done automatically, and our autograders will try to enter this information in this order.

2. ***Play the game using the Evil Hangman algorithm***. Specifically, you should do the following:
   1. Print out how many guesses the user has remaining, along with any letters the player has guessed and the current blanked-out version of the word. If the user chose earlier to see the number of words remaining, print that out too.
   2. Prompt the user for a single letter guess, reprompting until the user enters a letter that she hasn't guessed yet. Make sure that the input is exactly one character long and that it's a letter.
   3. Partition the words in the dictionary into their respective word families.
   4. Find the most common “word family” in the remaining words, remove all words from the word list that aren't in that family, and report the positions of the letter guessed (if any) to the user. If the word family doesn't contain any copies of the letter, subtract a guess.
   5. Repeat until the game ends.

3. ***Report the final result***. The game ends when the player is down to zero guesses or when the player has revealed all the blanks in the word. When that happens:
   1. If the player ran out of guesses, pick a word from the remaining word list (choose it however you’d like) and print a message that this was the word the computer initially “chose.”
   2. If the player revealed all the blanks, print out that resulting word and congratulate them!

4. ***Ask to play again***. Prompt the user and ask whether they want to play another game.

Testing Your Program

In the last assignment, we asked you to implement functions with clearly-defined interfaces. Here, you're writing a full program that interacts with the user, processes input, and the like, so you'll want to adopt a different testing approach.

In addition to writing your main program, you will need to write ***unit tests*** to go along with it. For example, you might have a helper function in your program that, given a collection of words, splits those words into word families. You could write test cases that specifically check to make sure that function works properly. Rather than hitting that function with the full dictionary, you could craft a smaller set of words and write a test to specifically check that those words get split apart properly.

As part of this assignment, you should write unit tests for at least ***three*** of the helper functions in your program. Don't put this off until the end – testing as you go can save you hours of frustrating debugging.

Advice, Tips, and Tricks

The starter code for this part of the assignment is essentially blank, and you'll build it from scratch. There is no "right way" to write this program, but some design decisions are better than others. Here's some things to think about as you code this up:

- ***Choose your data structures carefully.*** It's up to you to think about how you want to partition words into word families. Think about what data structures would be best for tracking word families and the master word list. Would a `Vector` work? How about a `Map`? A `Stack` or `Queue`?
- ***Decompose the problem into smaller pieces***. You'll need to write a decent amount of code to solve this problem, but that code nicely splits apart into a bunch of smaller pieces, things like "group words by their word family" or "read a letter from the user." Try to keep your functions short if at all possible, and follow good principles of top-down design. This has the nice side-effect of making your code much easier to test as you go.
- ***Use `Map`'s autoinsertion feature.*** If you look up a key/value pair in a C++ `Map` and the key doesn't exist, the `Map` will automatically insert a new key/value pair for you, using an intelligent default for the value. For example, if `myMap` is a `Map<string, Vector<string>>`, then writing

  ```
  myMap[myKey].add(myValue);
  ```

  will add `myValue` to the `Vector` associated with the key `myKey`, creating a fresh new `Vector` if `myMap` doesn't already have anything associated with `myKey`. Use this to your advantage – you can *dramatically* reduce the amount of code you need to write by using this feature of the `Map` type!
- ***Be careful how you pass arguments to functions***. In this part of assignment, you're likely going to be passing large objects around between functions. Remember to pass large objects either by reference (if you need to modify them) or `const` reference (if you don't) rather than by value.
- ***Letter position matters just as much as letter frequency.*** When computing word families, it's not enough to count the number of times a particular letter appears in a word; you also have to consider their positions. For example, `DEER` and `HERE` are in two different families even though they both have two `E`s in them.
- ***Break ties however you'd like***. In splitting words into word families, you may find that several word families have the same number of words in them. When that happens, you can break ties arbitrarily. This might result in your program missing out on good plays, and that's okay with us. If you want to be more intelligent with how you break ties, go for it, but don't worry if you don't.
- ***Watch out for gaps in the dictionary.*** When the user specifies a word length, you will need to check that there are indeed words of that length in the dictionary. You might initially assume that if the requested word length is less than the length of the longest word in the dictionary, there must be some word of that length. Unfortunately, the dictionary contains "length gaps," lengths where there's no words of that length even though there are words of a longer length.

- ***Don't explicitly enumerate word families.*** If you are working with a word of length *n*, then there are 2^n possible word families for each letter. However, most of these families don't actually appear in English. For example, no words contain three consecutive U's, and no word matches the pattern `E-EE-EE--E`. Rather than explicitly generating every word family whenever the user enters a guess, see if you can generate word families only for words that actually appear in the word list.
- ***Don’t modify collections when iterating over them***. In C++, if you iterate over an object like a `Map`, `Set`, or `Lexicon` and make changes to it during the iteration, you can break the underlying collection. If you want to start with one collection and remove some number of elements from it, we recommend doing so by building a brand-new collection, adding into it the elements you want to keep, and then assigning the new collection the old collection when you’re done.
- ***Recursion isn’t necessary.*** The first programming assignment of the quarter stressed recursion, and while you’re welcome to use it here, it’s not required. Don’t worry if you find yourself writing a lot of iterative code – that’s perfectly normal.
- ***Consider making a “game structure.”*** You may find yourself needing to pass a bunch of information around different functions, like the remaining words, the number of guesses, what’s been guessed so far, etc. This might result functions that take in a *lot* of parameters. As an alternative, consider defining your own custom `struct` to hold all of the information that you need, then pass that `struct` around through your code. For example, if you find yourself always needing to pass around the number of remaining guesses and what the word looks like so far, consider making a `struct` with those values as fields, then passing that `struct` as a parameter through your functions. This dramatically reduces the amount of time you’ll spend typing out parameter names.

All the code you need to write goes in `EvilHangman.cpp`.

Part Three: (Optional) Extensions

If you'd like to run wild with these assignments, go for it! Here are some suggestions.

- ***Crystals:*** Ulam's original 1962 paper outlining the process you just implemented contains suggestions for a few other variations on the idea. For example, what happens if you restrict the cells that you add in so that no two cells can be diagonally adjacent? There are many other variations on this idea that you can explore, and they give rise to all sorts of beautiful patterns. Explore and see what you find!

  The crystal pattern used here grows *beautifully* when you switch from working with a grid of squares and instead use a grid made of hexagons. How would you represent hexagonal coordinates? Pull up the documentation of our graphics package, and take a look at how our graphical program works. Could you draw hexagonal snowflake-style crystals?

- ***Evil Hangman:*** The strategy outlined in this assignment is an example of a *greedy algorithm*. At each step in the game, the program chooses the family of words that keeps the most words remaining even if it's not the best way to ensure a victory. For example, if the user has a single guess left and there are two options available to the program, one where it reveals a letter and keeps two words around and one where it doesn't reveal the letter and drops the number of guesses down to zero, the program really should choose that second option because it forces a victory. Consider making this program more intelligent in how it plays – though do keep in mind that you need to keep it fast or otherwise the user will suspect something's up!

  Alternatively, imagine you knew you were playing against an Evil Hangman opponent. What's the best sequence of letters to punch in? And how many guesses will you need to win? For example, is it ever possible to win with eight guesses using five letters?

Submission Instructions

Before you submit this assignment, make sure that you're doing the following:

- ***Crystals:*** Have you thoroughly tested your code? Make sure you've added in additional test cases beyond what's provided; those cases are not sufficient to guarantee everything works properly.
- ***Evil Hangman:*** Have you written unit tests for at least three helper functions? Have you tried providing bad inputs whenever you ask the user for information to make sure that your program behaves properly?

Once you're sure you've done everything correctly, including going through the Assignment Submission Checklist, submit `Crystals.cpp` and `EvilHangman.cpp` files online at https://paperless.stanford.edu/. And that's it! You're done!

Good luck, and have fun!