

# Lab 8 - Life is Like a Box of Squares

## Introduction

Have you ever found yourself outside on a *very hot* summer night, dripping with the stickiness of high humidity, only to be extra annoyed by the noise emanating from the pulsing, rhythmic chirping of a tree full of crickets? Well, I have. Then I paused to think ... just how did each of those crickets tune their singing to match the rate of their neighbors? I mean, they can't really talk to one another? They haven't evolved to the point where they can have meetings? Do they have ESP?




Or maybe you've won the **lottery** ➡

(<https://www.nps.gov/grsm/learn/nature/fireflies.htm>) and had the chance to visit the Great Smoky Mountains to see the forests **pulsate with light** ➡ (<https://www.quantamagazine.org/how-do-fireflies-flash-in-sync-studies-suggest-a-new-answer-20220920/>) during the Synchronous Firefly's mating season. Just how does that happen?

Don't like the outdoors? Okay, then maybe you found yourself wondering in the **earlier this year** ➡ ([https://www.wsj.com/articles/silicon-valley-bank-collapse-ceo-management-cb75f147?st=1eng4vlqx2wu4nu&reflink=desktopwebshare\\_permalink](https://www.wsj.com/articles/silicon-valley-bank-collapse-ceo-management-cb75f147?st=1eng4vlqx2wu4nu&reflink=desktopwebshare_permalink)) just why people seem to succumb to the fear of a **financial contagion** ➡ (<https://www.sciencedirect.com/uc.idm.oclc.org/science/article/pii/S1877050919307781>) all at the same time?

The only thing more amazing than *that* those patterns exist is that they are all the result of the same underlying cause: **So-called emergent behavior!** ➡ (<http://archive.sciencewatch.com/inter/aut/2008/08-dec/08decWattsET/>).

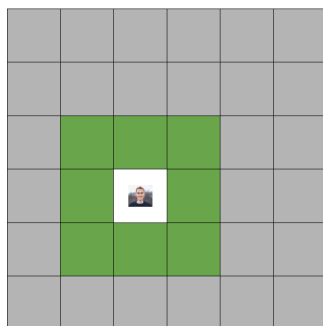
There are countless other examples of cases where a multitude of independent entities, each operating under the same set of *very simple rules*, coordinate (almost miraculously) to produce some **very** ➡ (<https://www.nature.com/scitable/topicpage/biological-complexity-and-integrative-levels-of-organization-468/>), **very** ➡ (<https://mathworld.wolfram.com/UniversalCellularAutomaton.html>) sophisticated behavior. There are scientists and philosophers who even believe that emergence is *the* **unifying theory of everything** ➡ (<https://medium.com/sfi-30-foundations-frontiers/emergence-a-unifying-theme-for-21st-century-science-4324ac0f951e>).

In this lab you will have the chance to build your own Metaverse whose denizens live in square boxes in close proximity to one another, operate independently according to a very small number of simple rules and, when viewed from far away, seem to produce amazingly complex actions. Our Metaverse will be built using the rules of the **Game of Life** 

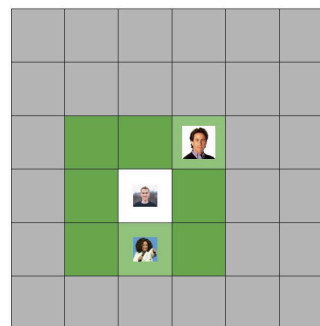
([https://en.wikipedia.org/wiki/Conway%27s\\_Game\\_of\\_Life](https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life)), a "game" proposed by mathematician John Conway in 1970. Each location in the Metaverse



- has a set of as many as eight (8) neighbors, above-left, above, above-right, right, below-right, below, below-left and left (the green locations in the Metaverse below are my neighbors) and
- is either occupied or not.

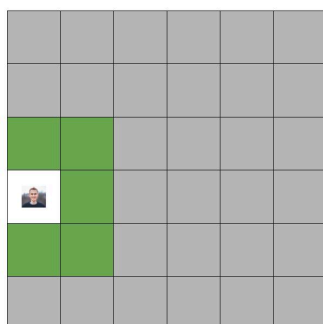


My eight neighbors are shown as the green locations in this Metaverse.

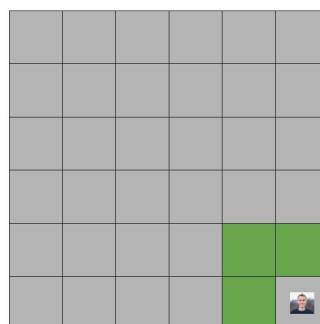


I have two occupied neighbors, Oprah and Jerry.

The Metaverse does not "wrap around". In other words, if you inhabit a location precariously close to an edge or a corner, you will have fewer neighbors than someone inhabiting a location closer to the center of the Metaverse.



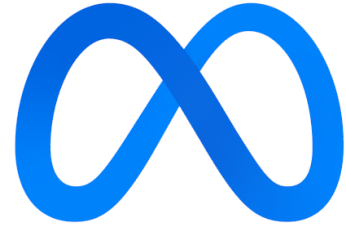
I only have 5 neighbors because I am living on the edge.



I only have three neighbors because, unlike Baby, you can put me in a corner.

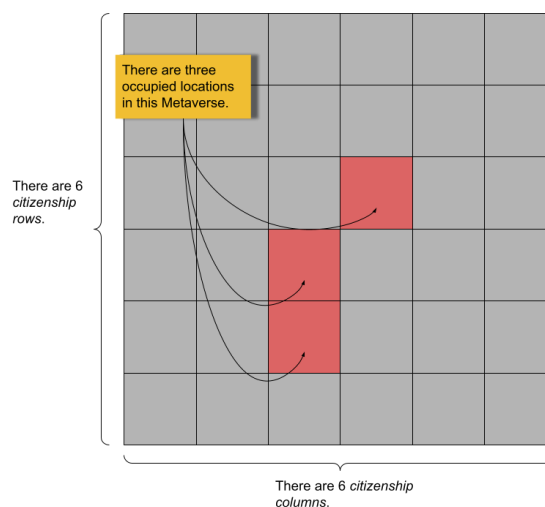
As ruler of your Metaverse you will program the rules that determine citizenship. Time in your Metaverse proceeds in a series of *ticks* that separate *generations*. At each tick, your rules will determine who can remain a citizen, who must leave, and who can become a citizen. For each of the locations in the Metaverse,

- If the location is occupied *and* it has either two or three occupied neighbors, the location will continue to be occupied during the next generation.
- If the location is *not* occupied *and* it has exactly three occupied neighbors, the location will become occupied during the next generation.
- Otherwise, the location is *not* occupied during the next generation.



Does Minecraft count as a metaverse? I am curious to know. Be the first to offer me your opinion, and a fun thing might happen.

Let's say that the 0th generation of our Metaverse looked like:



The 0th generation is our current Metaverse. From the 0th generation, we would want to tick over to the Metaverse's 1st generation and make that the current Metaverse. To do that, we should

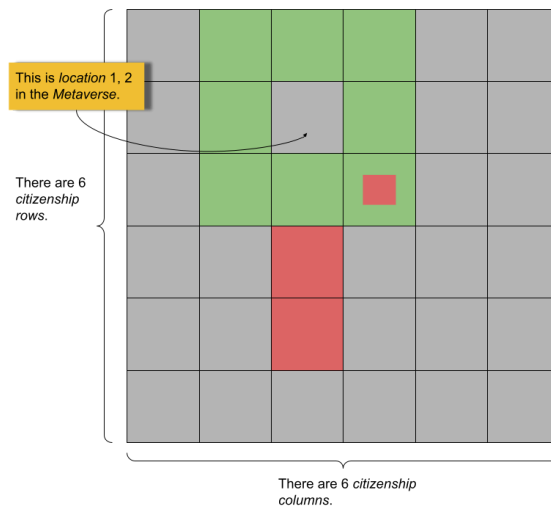
1. Create a blank 1st Generation Metaverse.
2. Populate that blank 1st generation Metaverse based on the population of the 0th generation Metaverse using the three (3) rules above.
3. Set the current Metaverse to the one that we just populated.

Step two (2) seems like the most complicated step. We should probably add some additional pseudocode to understand how that step should proceed.

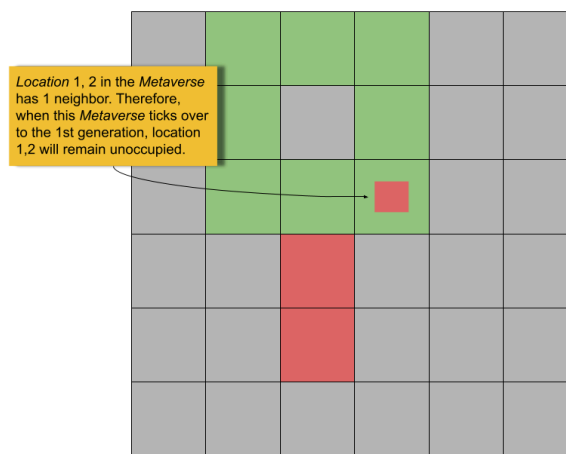
How would we determine the citizenship of the next generation? As we *ticked* to the next generation, we would have to make a calculation about *each* of the Metaverse's locations. For each of the current *Metaverse's* locations, we should

1. Count up the number of neighbors
2. Use that number to determine whether that location is occupied in the next generation
3. If it is occupied in the next generation, mark it as occupied.

As part of examining each of the current Metaverse's locations, we will consider location 1, 2.

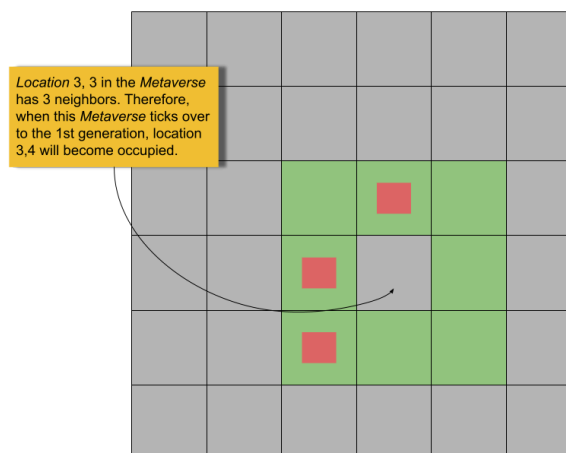


Location 1,2 has exactly 1 neighbor.



Based on the three (3) rules for citizenship, location 1, 2 will *not* be occupied in the next generation. So, we would leave location 1, 2 in the 1st generation Metaverse blank.

On the other hand, when we consider location 3, 3, we see that it has exactly three occupied neighbors!



What does that mean? Well, it means that in the 1st generation Metaverse, that location will be occupied!

After examining each of the locations in the 0th generation we will have assessed the number of neighbors for each location.

0	0	0	0	0	0
0	0	1	1	1	0
0	1	2	1	1	0
1	2	2	3	1	0
0	2	1	2	0	0
0	1	1	1	0	0

Using that information, we can determine which of the locations of the Metaverse will be occupied in the 1st generation. We will now call the 1st generation Metaverse the current Metaverse!


In other words, citizenship in our Metaverse tick after tick seems to depend on a Goldilocks scenario of enough neighbors to support a healthy community but not enough to produce overcrowding!

With just those three rules, given the right initial citizenship conditions, your Metaverse will generate some very interesting demographics!

All that's left is for you to set things in motion!

## Program Design Task

As my dad always says, "**A lie makes it around the world while I am still putting my shoes on one foot at a time.**" <https://www.nytimes.com/2017/04/26/books/famous-misquotations.html>."

*Before* you start writing code, please write your pseudocode. Even before *that*, take some time to think through the description of the conditions under which a location remains (or becomes) occupied in a subsequent generation and the description of the conditions under which a location previously occupied becomes vacant. Also, you might be tempted to use a **brute-force method**

→ [https://en.wikipedia.org/wiki/Brute-force\\_search](https://en.wikipedia.org/wiki/Brute-force_search) for counting the number of occupied neighbors of a given location -- can you do better than that? Can you conceive of a set of nested loops that will walk the neighbors automatically? And, if so, would writing an algorithm like that make it more or less likely that you will produce correct code?

## Program Design Requirements

Your pseudocode or flow chart must describe the entirety of the solution. You may choose to write the flow chart or the pseudocode at any level of detail but, remember, this is a tool for you! Your pseudocode or flow chart *must* include a description of how you plan

1. to calculate the number of neighbors of a given location safely (i.e., without accessing locations of the Metaverse that are either
  1. too large or
  2. too small);
2. to determine the occupancy of a location during a transition between generations; and
3. to read in the initial occupancy of your Metaverse from a configuration file (see below).

## Programming Task

Your task in this lab is to write *metaverse*, an implementation of the Metaverse according to the rules above. Your implementation will model the demographics of a Metaverse for a certain number of generations. During the transition between ticks/generations, *metaverse* will display the state of the universe in advanced, two-dimensional ASCII art. The size of the Metaverse and its initial demographics will be provided by the user in a file (always) named `universe.meta`.

The so-called *Universe File* will have the following format:

```
D,G
CCC ... C
.
.
.
CCC ... C
```

where

- `D` represents the height *and* the width of the Universe (only one number is needed because the Metaverse is always square),
- `G` represents the number of generations to model, and
- `C` represents a location's initial value (either `1` [for occupied] or `0` [for empty])

The first line of the file specifies the *configuration* of the Metaverse and is called the *metaverse configuration line*. Each of the lines of the Universe File after that specify the initial citizenship of the Metaverse. These rows are called *citizenship rows*. Each citizenship row has `D` location values and there are `D + 1` total lines in the file (the `+1` is to account for the metaverse configuration line).

When *metaverse* launches, it will read the Universe File, model the Metaverse for the specified number of generations and then exit.

For instance, if the user provided a Universe File with

```
8,3
00000000
00100000
00100000
00100000
00000000
00000000
00000000
00000000
00000000
```

then *metaverse* would display:

```

+
+
+

+ + +

+
+
+
```

## Programming Requirements:

If you are a Windows-based C++ developer, start with [this skeleton code](https://uc.instructure.com/courses/1657742/files/178591357?wrap=1)

(<https://uc.instructure.com/courses/1657742/files/178591357?wrap=1>). [↓](#)

([https://uc.instructure.com/courses/1657742/files/178591357/download?download\\_frd=1](https://uc.instructure.com/courses/1657742/files/178591357/download?download_frd=1)) . If you are a macOS-based C++ developer, start with [this skeleton code](https://uc.instructure.com/courses/1657742/files/178591517?wrap=1)

(<https://uc.instructure.com/courses/1657742/files/178591517?wrap=1>). [↓](#)

([https://uc.instructure.com/courses/1657742/files/178591517/download?download\\_frd=1](https://uc.instructure.com/courses/1657742/files/178591517/download?download_frd=1)) . Finally, if you are a Linux-based C++ developer, start with [this skeleton code](https://uc.instructure.com/courses/1657742/files/178590977?wrap=1)

(<https://uc.instructure.com/courses/1657742/files/178590977?wrap=1>). [↓](#)

([https://uc.instructure.com/courses/1657742/files/178590977/download?download\\_frd=1](https://uc.instructure.com/courses/1657742/files/178590977/download?download_frd=1)) .

For this lab you are responsible for implementing several helper functions *and* the function that runs the model. In other words, you get to do (almost) everything! Read the next section for a description of what you are given to help you do your job and the section after that for a description of your tasks. I *highly* recommend that you use the given functions to help you implement *metaverse*.


Given:

In any generation, each location in a Metaverse is either occupied or not. That occupational status is binary -- in other words, it would seem reasonable to represent it as a boolean variable. Because the Metaverse's locations form a grid, the ideal way to represent that is with a two-dimensional data structure. In class we have learned two ways of implementing such a structure: an array of arrays or a vector of vectors. For *metaverse* we will prefer a vector because the size of the Metaverse to be modeled can change every time *metaverse* is executed. Given that constraint there is no way that we could tell at the time the compiler builds our code just how big to make our data structure. Therefore, it would not be possible to use arrays. The type of a vector of vectors holding `bool` eans looks like

```
std::vector<std::vector<bool>>>;
```

in C++. But, boy, that's a lot to type! What's more, it does not really give other programmers a sense of the data being stored. Perhaps we are using that two-dimensional array to store a representation of a bingo card -- who knows! In C++ we can give *aliases* to certain types. We can use those aliases to give types more semantically meaningful names. In the *metaverse* application, you will use an alias defined by

```
using metaverse_t = std::vector<std::vector<bool>>>;
```

With that ***alias declaration*** , ([https://en.cppreference.com/w/cpp/language/type\\_alias](https://en.cppreference.com/w/cpp/language/type_alias)), we can write code that understands the `metaverse_t` type. Anything that can be done to/with a two-dimensional array of `bool` eans can be done to/with a `metaverse_t` -- they are synonymous! That's really cool!

Function	Use	Input	Output
<code>initialize_metaverse_from_file</code>	This function will initialize a Metaverse from a Universe File.	<div>1. A file reference: A reference to a <code>std::ifstream</code> for accessing the Universe File</div> <div>2. A Metaverse reference: A reference to a <code>metaverse_t</code> that will be configured according to the</div>	The function will return <code>true</code> if reading a configuration from the given Universe File was successful. On success, the by-reference parameter for the <code>metaverse_t</code> will be updated to match the Metaverse configured in the



		<p>data in the Universe File</p> <p>3. A Generation reference: A reference to an <code>int</code> that will hold the number of generations to model (as read from the Universe File).</p>	<p>Universe File and the by-reference parameter for the number of generations to model will be updated to match the input from the Universe File.</p>
display_metaverse	<p>This function prints out a Metaverse (as encoded in a <code>metaverse_t</code>) to the specified I/O stream in a pretty format.</p>	<p>1. An output stream reference: A reference to a <code>std::ostream</code> that will be used as the destination for the formatted output.</p> <p>2. A Metaverse reference: A <code>const</code> reference to a <code>metaverse_t</code> to format and write to the given output stream.</p>	<p>The function returns nothing but does have the side effect of writing the formatted version of the given Metaverse to the given output stream.</p>

## To Do

Function	Use	Input	Output
model_metaverse	<p>This function will model the evolution of a Metaverse for a given number of generations. At each tick, the function will display the state of the Metaverse.</p>	<p>1. Metaverse reference: A <code>const</code> reference to a <code>metaverse_t</code> specifying the citizenship of the Metaverse at Generation 0.</p> <p>2. Number of generations: The number of generations for</p>	<p>This function will return no value. However, it will have the side effect of printing out the state of the Metaverse between generations.</p>

		which to model the Metaverse.	
<pre>read_metaverse_configuration_line_from_file</pre>	<p>This function will read and parse the configuration line from a Universe File. It will return <code>true</code> and update its <i>out parameters</i> if reading was successful (i.e., when a size and number of generations, separated by a comma, could be read). It will return <code>false</code> otherwise.</p>	<ol style="list-style-type: none"> <li>1. Universe File reference: a reference to a <code>std::ifstream</code> for the Universe File. You can assume that the read head for the file will be positioned at the beginning of the metaverse configuration line that you will read.</li> <li>2. Size number reference: A reference to an <code>int</code>.</li> <li>3. Generation number reference: A reference to an <code>int</code>.</li> </ol>	<p>The function will return <code>true</code> or <code>false</code> depending upon whether a configuration could be read from the Universe File using the given reference. If the read was successful, the Size reference will be updated with the size of the Metaverse and the Generation reference will be updated with the number of generations to model (both as specified in the Universe File).</p>
<pre>citizenship_row_to_metaverse_row</pre>	<p>This function will take a string of characters read from a row of the Universe File and update the initial citizenship of Metaverse accordingly.</p>	<ol style="list-style-type: none"> <li>1. Row string: a <code>const</code> reference to a <code>std::string</code> which contains the citizenship row of the Metaverse at ...</li> <li>2. Row number: an <code>int</code> which specifies the row whose occupancy is defined by the Row string.</li> <li>3. Metaverse reference: A <code>metaverse_t</code> reference to be updated according to the preceding parameters.</li> </ol>	<p>The function will return <code>true</code> and update the given Metaverse according to the Row string <i>as long as the initial occupancy length matches the Metaverse size</i>. It will return <code>false</code>, and <i>not</i> update the Metaverse, if the Row string does not have a valid length.</p>

resize_metaverse	This function will resize a Metaverse according to a given size.	<ol style="list-style-type: none"> <li>1. Size number: an <code>int</code> representing the number of rows and columns in the Metaverse.</li> <li>2. Metaverse reference: A <code>metaverse_t</code> reference to be resized according to the Size number.</li> </ol>	The function will resize the given <code>metaverse_t</code> parameter according to the specified size. The function <i>always</i> returns <code>true</code> .
count_neighbors	Count the number of occupied neighbors for a given location.	<ol style="list-style-type: none"> <li>1. Metaverse constant reference: A <code>metaverse_t</code> <code>const</code> reference.</li> <li>2. Row number: An <code>int</code> for the row of the location to determine neighborhood population.</li> <li>3. Column number: An <code>int</code> for the column of the location to determine neighborhood population.</li> </ol>	The function will return the number of occupied neighbors in the current generation of the Metaverse described by Metaverse referred to by the Metaverse constant reference at the location specified by the Row and Column number parameters.
occupied_in_next_tick	This function will return <code>true</code> if a given location in the Metaverse is occupied in the next generation.	<ol style="list-style-type: none"> <li>1. Boolean: A bool that indicates whether the location is occupied (or not) in the current generation.</li> <li>2. Occupied neighbor count: An integer holding the count of the number of occupied neighbors.</li> </ol>	The function will return <code>true</code> if the (un)occupied location with the specified number of occupied neighbors will be occupied in the next generation.
tick	Build a new Metaverse one tick in the future	<ol style="list-style-type: none"> <li>1. Metaverse constant reference: A</li> </ol>	The function will return a new Metaverse (as a

	from a given Metaverse.	<code>metaverse_t</code> <code>const</code> <code>metaverse_t</code> ) that reference.	describes the next generation of the Metaverse described by the Metaverse reference parameter.
--	-------------------------	----------------------------------------------------------------------------------------	------------------------------------------------------------------------------------------------

Documenting functions is a very important part of a professional programmer's job. Every programming project has their own preferred format for writing comments that describe functions and their inputs/outputs. For the *metaverse* application, the comments for each of the functions you define must conform with the following format:

```
/*
 * <function name>
 *
 * <short description of what the function does>
 *
 * input: <short description of all input parameters>
 * output: <short description of all output parameters
 *         and the return value>
 */
```

Note: You will replace everything *including* and between the `<` `>` s in the style template.

## Unit 5A Testing

Remember when we were building our search engine (BearING) how we had unit tests? Those are the little functions that test functions that we have written (wow, it really is functions everywhere!). While we ultimately want to test whether our entire program works the way we expect, it helps us to have some tests that will show us the smaller components are working as we expect. That way, when we ultimately go to use those functions to build the bigger program, we can have confidence that the building blocks are solid.

In this lab, I did not give you any unit tests. However, I highly recommend that you write your own. They will help you keep track of the progress that you are making toward completion of the lab as you implement each of the required functions.

Just how can you write your own unit tests? How can you execute them once you've written them? Let's assume that we want to write a unit test to determine whether our `read_metaverse_configuration_line_from_file` function works as anticipated. In the skeleton, you are given two different sample metaverse configuration files (`universe.meta` and `universe2.meta`). Open those files in a text editor and see what the configuration line looks like.

I'll wait.

Good! `universe.meta` specifies a universe whose size is 8 with a simulation duration of 8 generations. `universe2.meta` specifies a universe whose size is 6 with a simulation duration of 8 generations. The parameters of `read_metaverse_configuration_line_from_file` are 1) a `std::ifstream` which references a metaverse configuration file, 2) an `int` eger where the function will store the universe size (by reference, of course!) and 3) an `int` eger where the function will

store the number of generations to model (again, by reference!). The function will return a `bool` signalling whether the configuration could be successfully read.

In order to have what we need to invoke the `read_metaverse_configuration_line_from_file` and see how it operates, we will need four variables: one for each of the parameters and one for its return value. We will call `read_metaverse_configuration_line_from_file` and then see whether it returns the proper result and updates the by-reference argument variables correctly.

Our declarations of the necessary variables will look like this:

```
std::ifstream metaverse_file{"universe.meta"};
bool success_or_failure{false};
int generations{0};
int size{0};
```

Here we are associating the `metaverse_file` `std::ifstream`-typed variable with the `universe.meta` file. Remember, that file specifies a universe size of 8 and a modeling duration of 8 generations. If we call

```
success_or_failure = read_metaverse_configuration_line_from_file(metaverse_file, size, generations);
```

then we expect that `size` will be equal to 8 and `generations` will be equal 8 and `success_or_failure` will be equal to `true`. How do we make sure that is the case? Well, we can just print them out to the screen:

```
std::cout << "read_metaverse_configuration_line_from_file result : " << success_or_failure << "\n";
std::cout << "read_metaverse_configuration_line_from_file size : " << size << "\n";
std::cout << "read_metaverse_configuration_line_from_file generations: " << generations << "\n";
```

Awesome!

Now that we have code for unit test, let's put it in a separate function so that it can be used over and over again:

```
void unit_test_read_metaverse_configuration_line_from_file() {
    std::ifstream metaverse_file{"universe.meta"};
    bool success_or_failure{false};
    int generations{0};
    int size{0};
    success_or_failure = read_metaverse_configuration_line_from_file(metaverse_file, size, generations);
    std::cout << "read_metaverse_configuration_line_from_file result : " << success_or_failure << "\n";
    std::cout << "read_metaverse_configuration_line_from_file size : " << size << "\n";
```

```
std::cout << "read_metaverse_configuration_line_from_file generations: " << generations <<
"\n";
}
```

Great! So, now we just need to figure out how to run our unit test. Where to put it? Yes, that's right! Because our program's execution starts in the `main` function, we should put an invocation of `unit_test_read_metaverse_configuration_line_from_file` in there! Where is the `main` function in this lab? It's in the `main.cpp` file! But wait ... there's code already there! And we want to keep that code because it's important for the lab. What to do? The best solution is to simply comment out the existing code and add a call to the test function!

Once you have done that, you can run your program and the unit test will be the only code that runs! If your `read_metaverse_configuration_line_from_file` function works correctly, you will see output that looks something like

```
read_metaverse_configuration_line_from_file result      : 1
read_metaverse_configuration_line_from_file size       : 8
read_metaverse_configuration_line_from_file generations: 8
```

How neat! And to make sure that everything is really great, you could change your unit test to read from `universe2.meta` and make sure the results match what you would expect:

```
read_metaverse_configuration_line_from_file result      : 1
read_metaverse_configuration_line_from_file size       : 6
read_metaverse_configuration_line_from_file generations: 8
```

Now that you got the hang of unit tests, don't stop there! What if you wanted to test your implementation of, say, `count_neighbors`? Well, we'll just make another unit test function that calls `count_neighbors` and checks the results. Hmm, that function requires a valid metaverse in order to work. No problem, we can just build a metaverse of our own by hand! Let's say that we want to build a metaverse that we can use for testing that has a size of 8. A `metaverse_t` is just a two dimensional `std::vector` of `bool`s and we know how to make one of those. Here we go:


```
metaverse_t testing_metaverse(8, std::vector<bool>(8, false));
```

It would be really great if we were able to test to make sure that we created the metaverse that we envisioned. You can use the `display_metaverse` function to do just that! (**Important:** You will need to ask the preprocessor to

```
#include "display.h"
```

before you are able to use `display_metaverse` in the `gol.cpp` file. You can insert that line before all the other preprocessor directives!).

Before we get too carried away, remember that we are going to have to write each set of test code in its own function so that we can call it from `main`.

You are getting the hang of it now! As you implement the functions in the lab, consider writing unit tests to test your code. That way, when the **final countdown** 

([https://en.wikipedia.org/wiki/The\\_Final\\_Countdown\\_\(song\)](https://en.wikipedia.org/wiki/The_Final_Countdown_(song))) starts, you will be able to write `model_metaverse` using your building blocks with the confidence that they are correct!





One final note: When you are done unit testing and ready to test your `model_metaverse` function, be sure to remove all the calls to your unit tests from `main` and put back the code that we commented out!

## Critical Thinking Task

In myriad ways, people living in one country can affect people in another country, often in a very direct way. That is the promise and peril of modern society. The mobility and interconnectedness that leads to the quick dissemination of new knowledge are the same forces that led to the rapid spread of COVID-19 from country to country. Just as quickly as it was a problem, it became a solution: these forces helped scientists work collaboratively throughout the world to find a vaccine so quickly.

The positive benefits of our connections, however, do not happen by accident. The ability for two people to communicate requires an agreement on the format of messages. Scientific collaboration on studies of raw data requires storage of that data in a specific format. The ability to transfer physical goods from Point A to Point B requires shared understanding about packages' size, shape and weight. The proliferation of academic knowledge requires a shared language and methodology.

These agreements are usually defined by national and international organizations formed by governments or private businesses. The standards' bodies are incredibly important in the field of computer science and telecommunications. For instance, the W3C specifies the valid syntax for a web page; the ISO C++ committee defines the syntax and semantics of the C++ programming language; and ANSI specifies the ASCII codes for a limited subset of characters that can be represented in a computer.

**Originally known as the European Computer Manufacturers Association, Ecma International - European association for standardizing information and communication systems**  (<https://www.ecma-international.org/about-ecma/history/>) is an international standards organization that is **"dedicated to the standardization of information and communication systems."**  (<https://www.ecma-international.org/>) **What is now called Ecma International was founded in 1960 and included the three major European computer manufacturers at the time: Compagnie des Machines Bull, IBM World Trade Europe Corporation and International Computers and Tabulators Limited.**  (<https://www.ecma-international.org/about-ecma/history/>) Ecma was an early proponent of Near-Field Communication protocols -- the technology that makes it possible to pay by tapping your credit card -- and established an early standard for it in **2003**  ([https://en.wikipedia.org/wiki/Near-field\\_communication](https://en.wikipedia.org/wiki/Near-field_communication)). **They are also**



**actively involved in standardizing the formats that make it possible to share documents between Microsoft Office programs and other productivity tools** [↗\(https://www.ecma-international.org/technical-committees/tc45/\)](https://www.ecma-international.org/technical-committees/tc45/). Ecma is arguably most active in standardizing the syntax and semantics of programming languages. Committees at Ecma are responsible for standardizing **Dart** [↗\(https://www.ecma-international.org/technical-committees/tc52/\)](https://www.ecma-international.org/technical-committees/tc52/), **C#** [↗\(https://www.ecma-international.org/task-groups/tc49-tg2/\)](https://www.ecma-international.org/task-groups/tc49-tg2/) and, most famously, **JavaScript** [↗\(https://tc39.es/\)](https://tc39.es/).

Your critical thinking task is to identify an organization that standardizes and promotes agreements used in computer science and telecommunications and describe their work.

## Critical Thinking Requirement:

In 500 words or less, identify an organization that standardizes and promotes agreements used in computer science and telecommunications and describe their work. All references to external resources must be properly documented and formatted. The choice of formatting for external references is up to you, but you may find it helpful to consult the Purdue OWL for **information** [↗\(https://owl.purdue.edu/owl/research\\_and\\_citation/apa\\_style/apa\\_style\\_introduction.html\)](https://owl.purdue.edu/owl/research_and_citation/apa_style/apa_style_introduction.html). The Purdue OWL also has extensive information on ways to **avoid plagiarism** [↗\(https://owl.purdue.edu/owl/avoiding\\_plagiarism/index.html\)](https://owl.purdue.edu/owl/avoiding_plagiarism/index.html).

## Question Task:

For this lab you deployed your knowledge of two-dimensional vectors. You worked hard to think in terms of types as you worked with the `metaverse_t` data structure. What made it easier or harder to determine whether indexing the instance of a `metaverse_t` retrieved a vector of `bool` eans (i.e., a row of locations) or a single `bool` ean (i.e., a single location)? Mastering the skill of thinking in types will help you as you grow as a developer and computer scientist because it will aid your ability to deploy abstraction. Moreover, this lab represents the first time that you have had a chance to work with data structures. What did you find confusing? Was there anything that you need clarified?

## Question Requirement:

Submit a file named `question.pdf` which contains your question. Take particular notice of the fact that the question does not have to be subtle, or deep or philosophical for you to receive full points. The goal of the question is for me to be able to get a sense whether there are common questions about the material which would indicate where I failed to teach the material appropriately.

## Deliverables

1. The pseudocode describing the algorithm of your *metaverse* program in PDF format (named `design.pdf`).
2. The C++ source code for your *metaverse* application (named `metaverse.cpp`).



3. A written response that describes the work of an organization that standardizes and promotes agreements used in computer science and telecommunications in PDF format (named `protocol.pdf`).
4. A written question about the material covered so far in this class in PDF format (name the file `question.pdf`).

## Rubric

<b><u>Points</u></b>	<b><u>Task</u></b>	<b><u>Criteria</u></b>
15	Design	Pseudocode uses algorithmic thinking to express a high-level description of how to calculate the occupancy of each of the locations of the Metaverse, how to calculate the number of occupied neighbors for a location in the Metaverse (including how to make sure not to access unsafely access elements outside the world) and how to read the Universe File.
15	Critical Thinking	In 500 words or less, your response to the Critical Thinking Task identifies an organization that standardizes and promotes agreements used in computer science and telecommunications and describes their work. All references to external resources must be properly documented and formatted.
10	Programming	Each of the functions that you defined are properly documented with comments that match the format given in the Programming Requirements section.
20	Programming	<i>metaverse</i> application properly models the expansion of the Metaverses given as samples with the skeleton code for this lab.
25	Programming	<i>metaverse</i> application properly models the expansion of the Metaverses <i>not</i> given as part of the skeleton code for this lab.
10	Programming	Code for each of the helper functions is written to match the pseudocode and all variables are given appropriate/meaningful names.
5	Question	Your question shows an engagement with the material.