Project 4 – Labyrinth Escape II
CS 251, Fall 2021

Collaboration Policy: By submitting this assignment, you are acknowledging you have read the collaboration policy in the syllabus for projects. This project should be done individually. You may not receive any assistance from anyone outside of the CS 251 teaching staff. Also, please see copyright at the bottom of this description. Do not post your work for this class publicly, even after the course is over.

Late Policy: You are given a total of 5 late days to use at your discretion (for all projects, not each project). You may use the late days in 24-hour chunks (either 1 day at a time or all five at once). You do not need to alert the instructor to ask permission to use late days. You can manage your use of late days on Mimir directly.

Early Bonus: If you submit a finished project early (by Wednesday, October 13th at 11:59 pm), you can receive 10% extra credit. Please note, your submission needs to pass 100% of the tests cases to receive the early bonus and all files must be included and final.

Test cases/Submission Limit: Limited submissions (15 total). Most test cases are hidden.

IDE: You will need to use Codio to develop your code, as we have set up all the Linux tools needed (gdb, GoogleTests, valgrind). To find the starter code, just login at codio.com and you will see Project 5. If you haven't set up your account yet, use these instructions.

What to submit: (1) grid.h; (2) EscapeTheLabyrinth.h; (3) tests.cpp; (4) gdb_log.txt

.pdf starter code

Project Background

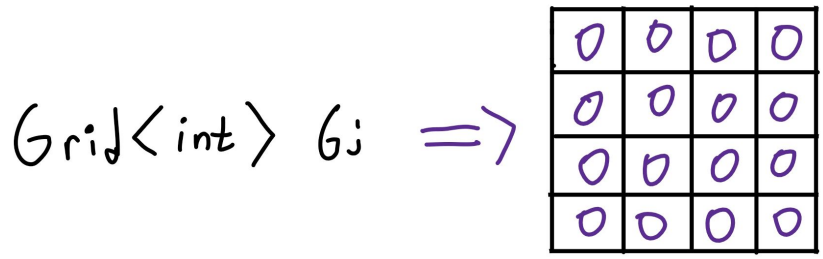
In your last project, you implemented an abstraction using a C-array data structure. Hopefully, that made you familiar with writing classes, writing abstractions, memory management (allocating and deallocating memory on the heap), and an introduction to classes. The next step in your journey is implement an abstraction using a linked structure. This will prepare you for what is next…trees and graphs, which are more complex linked structures. If you have yet to experience the joy of a segmentation fault, it is bound to happen soon. Remember, a segmentation fault is a result of accessing memory you do not have permission to access.

There are a few news things you will be working with for this Project. First, you will be using the Google Test framework. After getting it set up (see Google Tests section at bottom of handout), this should make your life easier for testing. Second, you will be writing a templated class, which is a first. Third, you will be learning the power of a debugger (gdb) when working with linked structures. And along the way, you will get to escape a labyrinth, which I personally found quite thrilling.

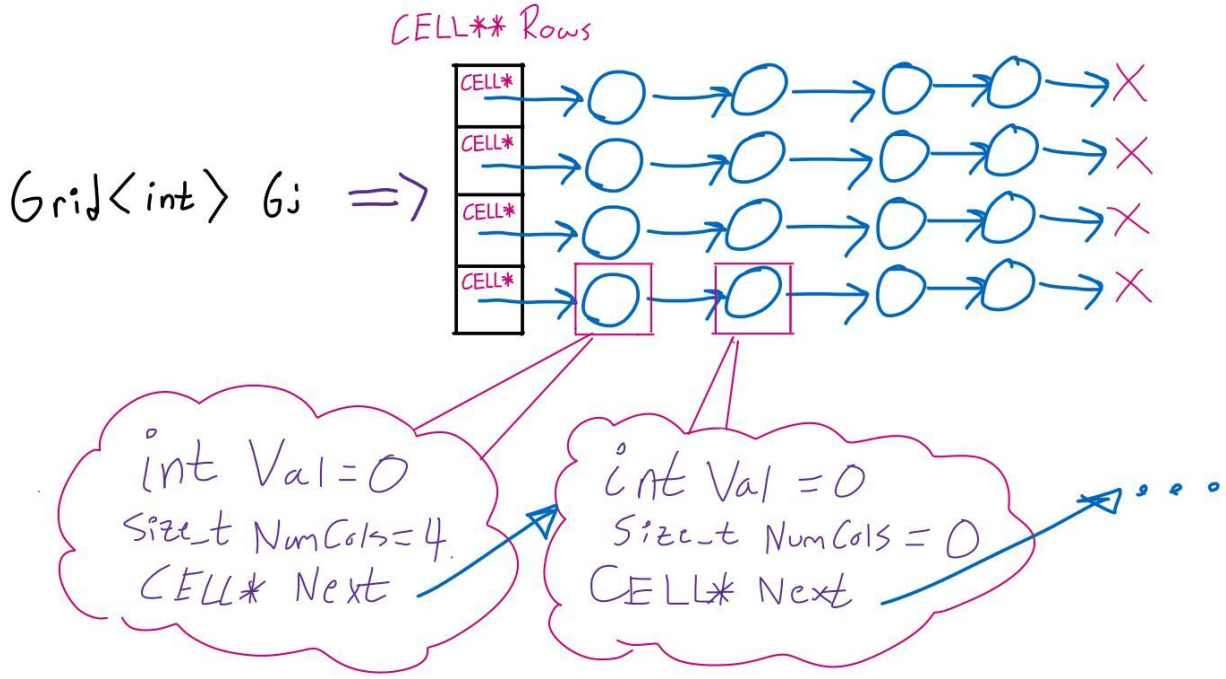
As a warning, Milestones 2+ depend on Milestone 1. The test cases provided for Milestone 2+, as well as the files provided to explore the mazes will all segfault if you run them before properly implementing grid.h.

Milestone #1 – Write grid.h and test

The goal of this part of the project is to define a class **Grid<T>** explicitly designed to support a 2D grid. We need this abstraction in order to create mazes and escape labyrinths later. A grid is defined to have a given number of rows and columns, and the resulting elements are initialized to C++'s natural default value. For example, the default is a 4x4 grid:



Keep in mind this is an abstraction, the actual implementation of the grid is quite different. While this version of grid is not going to be able to dynamically grow (you are welcome), we are going to design the data structure under the hood for potential future plan. A C-style array of linked lists are used.



The milestone consists of implementing a set of member functions for class **Grid<T>**. The first requirement is that you *must* implement Grid as discussed above, and defined in the provided "Grid.h" header file. There are lots of possible implementations, but we require that you use this approach. Here are the relevant declarations from "Grid.h":

```
template<typename T>
class Grid {
private:
  struct CELL {
    CELL* Next;
    T Val;
    size_t NumCols;  // total # of columns (0..NumCols-1)

    CELL(CELL* _Next = nullptr, T _Val = T(), size_t _NumCols = 0) {
      Next = _Next;
      Val = _Val;
      NumCols = _NumCols;
    }
  };

  size_t NumRows;  // total # of rows (0..NumRows-1)
  CELL** Rows;     // C array of linked lists
```

You are free to add additional member variables to improve the implementation, as well as private helper functions. But you cannot change the overall implementation of a grid: it must remain a pointer to an array of linked lists. The linked list is composed of a CELL structure which has a next CELL (which could be nullptr), a value of type T, and the number of columns. You cannot switch to a vector-based implementation, nor other data structures.

By default, a grid is 4x4. Here's the code for the default constructor that creates this 4x4 grid. You'll want to match this up with the diagram shown above.

```
public:
  //
  // default constructor:
  //
  // Called automatically by C++ to construct a 4x4 Grid.  All
  // elements are initialized to the default value of T.
  //
  Grid() {
    // initialize 4 rows
    Rows = new CELL*[4];
    NumRows = 4;

    // allocate the first cell of the linked list with default value:
    for (size_t r = 0; r < NumRows; ++r) {
        Rows[r] = new CELL(nullptr, T(), 4);
        CELL* cur = Rows[r];

        // create the linked list for this row.
        for(size_t c = 1; c < Rows[r]->NumCols; ++c) {
            cur->Next = new CELL(nullptr, T());
            cur = cur->Next;
        }
    }

  }
```

The parts you need to complete are marked with TODO. Make sure you thoroughly test your grid.h using the Google Test framework (tests.cpp, which must be submitted for grading). With the grid getter (the () operator) , please throw an exception if the user tries to access an invalid row, col (the message is your choice). Some initial tests are provided in the tests.cpp file. Remember, this class does not dynamically expand. It is a fixed size based on the constructor. Also, you will notice that this class is templated. Fun! Therefore, make sure you fully test it on all types. Finally, don't forget to run valgrind to make sure you freed all the memory properly. It is always a good idea to leave the destructor for last.

Milestone #2 – understand how to escape a Labyrinth

This step uses your Grid abstraction. Please make sure it well tested and fully functioning before moving on to this step.

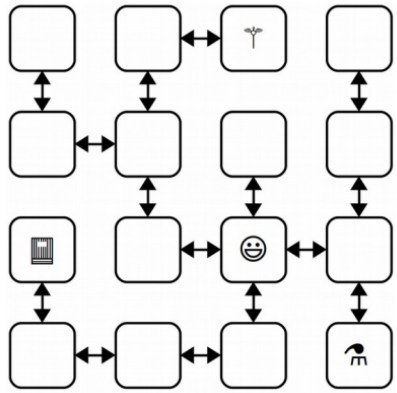
You have been trapped in a labyrinth, and your only hope to escape is to cast the magic spell that will free you from its walls. To do so, you will need to explore the labyrinth to find three magical items:

- The Spellbook (📕), which contains the spell you need to cast to escape.
- The Potion (⚗), containing the arcane compounds that power the spell.
- The Wand (🗡), which concentrates your focus to make the spell work.

Once you have all three items, you can cast the spell to escape to safety. This is, of course, no ordinary maze. ***It's a pointer maze.*** The maze consists of a collection of objects of type MazeCell, where MazeCell is defined here:

```
struct MazeCell {
    string whatsHere; // One of "", "Potion", "Spellbook", and "Wand"
    MazeCell* north;
    MazeCell* south;
    MazeCell* east;
    MazeCell* west;
};
```

For example, check out this 4 × 4 labyrinth:



We've marked your starting position with a smiley face and the positions of of the three items with similarly cute emojis. The MazeCell you begin at would have its north, south, east, and west pointers pointing at the MazeCell objects located one step in each of those directions from you. On the other hand, the MazeCell containing the book would have its north, east, and west pointers set to nullptr, and only its south pointer would point somewhere (specifically, to the cell in the bottom-left corner). Each MazeCell has a variable named whatsHere that indicates what item, if any, is at that position. Empty cells will have whatsHere set to the empty string. The cells containing the Spellbook, Potion, or Wand will have those fields set to "Spellbook", "Potion", and "Wand", respectively, with that exact capitalization.

If you were to find yourself in this labyrinth, you could walk around a bit to find the items you need to cast the escape spell. There are many paths you can take; here's three of them:

ESNWWNNEWSSESWWN

SWWNSEENWNNEWSSEES

WNNEWSSESWWNSEENES

Each path is represented as a sequence of letters (N for north, W for west, etc.) that, when followed from left to right, trace out the directions you'd follow. For example, the first sequence represents going east, then south (getting the Potion), then north, then west, etc. Trace though those paths and make sure you see how they pick up all three items. There are a few more milestones to this project. The next one is a coding problem, and the remaining two are debugger

exercises. We strongly recommend completing them in the order in which they appear, as they build on top of one another.

There isn't a lot of code to write for the rest of the project. However, there is a lot left to do. All the code is contained in these files:

maze.h – a class already written for you. It creates mazes for you. You do not need to write any code in this file.

EscapeTheLabyrinth.h – this is the only file that you will write code in and submit. This is where you will write isPathToFreedom in Milestone #3. This is also where you will put your solutions to your personalized mazes.

ExploreTheRegularLabyrinth.cpp – This is the file you will use to solve the regular maze using gdb in Milestone #4. You won't need to write any code here, just need to run it.

ExploreTheTwistyLabyrinth.cpp – This is the file you will use to solve the twisty maze using gdb in Milestone #5. You won't need to write any code here, just need to run it.

Milestone #3 - write isPathToFreedom

Your first task is to write a function that, given a cell in a maze and a path, checks whether that path is legal and picks up all three items. Specifically, in the file **EscapeTheLabyrinth.h**, implement the function:

```
bool isPathToFreedom(MazeCell *start, const string& moves);
```

This function takes as input your starting location in the maze and a string made purely from the characters 'N', 'S', 'E', and 'W', then returns whether that path lets you escape from the maze. A path lets you escape the maze if

(1) the path is legal, in the sense that it never takes a step that isn't permitted in the current MazeCell, and
(2) the path picks up the Spellbook, Wand, and Potion.

The order in which those items are picked up is irrelevant, and it's okay if the path continues onward after picking all the items up (however, it still must be a valid path). You can assume that **start** is not a null pointer (you do indeed begin somewhere). However, if the input string contains any invalid characters (that is, characters other than 'N', 'S', 'E', or 'W'), you should return false.

To summarize:
1. Implement the isPathToFreedom function in **EscapeTheLabyrinth.h**.
2. Test your code thoroughly by writing tests in tests.cpp (some are provided).

Some notes on this problem:

- Every maze cell's whatsHere field will contain exactly one of the four options indicated earlier ("", "Spellbook", "Wand", and "Potion"). You don't need to worry about the case where the maze contains items other than these. We could have conceivably used an enumerated type to represent this but figured it would be easier to just use strings here.
- Your code should work for a MazeCell from any possible maze, not just the one shown here.
- Although in the previous picture the maze was structured so that if there was a link from one cell to another going north there would always be a link from the second cell back to the first going south (and the same for east/west links), you should not assume this is the case in this function. Then again, chances are you wouldn't need to assume this.
- You shouldn't need to allocate any new MazeCell objects in the course of solving this problem. Feel free to declare variables of type MazeCell*, but don't use the new keyword. After all, your job is to check a path in an existing maze, not to make a new maze.

Milestone #4 – Escape your personalized secret Labyrinth

Your next task is to escape from a labyrinth that's specifically constructed for you. The starter code we've provided will use your name to build you a personalized labyrinth. By "personalized," we mean "no one else in the course is going to have the exact same labyrinth as you." Your job is to figure out a path through that labyrinth that picks up all the three items, allowing you to escape.

Open the file **EscapeTheLabyrinth.h** and you'll see three constants. The first one, kYourName, is a spot for your name. Right now, it's marked with a TODO message. Edit this constant so that it contains your name (full first and last name, some unique to only you).

Scroll down to the second to last test cases in tests.cpp. This test case generates a personalized labyrinth based on the kYourName constant and returns a pointer to one of the cells in that maze. It then checks whether the constant kPathOutOfRegularMaze is a sequence that will let you escape from the maze. Right now, kPathOutOfRegularMaze is a TODO message, so it's not going to let you escape from the labyrinth. You'll need to edit this string with the escape sequence once you find it. To come up with a path out of the labyrinth, you will use a debugger! We will be using gdb. I will show you how to use it now. Please note that you need to turn in your gdb log in a text file, so make sure to save what you do. [This a good resource too](#).

You are going to run gdb on **ExploreTheRegularLabyrinth.cpp.** The makefile is set up to run this file for you. Try it first by typing:

```
>> make build_reg
>> make run_reg
```

It should say that you did not escape. Now, let's explore the maze. After compiling (make build_reg), run gdb on the **ExploreTheRegularLabyrinth executable:**

```
^_^[shanon] >>  gdb ExploreTheRegularLabyrinth.exe
```

```
GNU gdb (Ubuntu 8.2-0ubuntu1~18.04) 8.2
Copyright (C) 2018 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ExploreTheRegularLabyrinth.exe...done.
```

Set a breakpoint at line 16 of the file (the line after we get our starting point for the maze), press enter. Then, type r and enter to run the executable.

```
(gdb) b ExploreTheRegularLabyrinth.cpp:16
Breakpoint 1 at 0x1dfe: file ExploreTheRegularLabyrinth.cpp, line 16.
(gdb) r
Starting program:
/home/shanon/cs_251_data_structures_spring_2021/project4_testing/
ExploreTheRegularLabyrinth.exe
Breakpoint 1, main () at ExploreTheRegularLabyrinth.cpp:16
16      // Put your break point here!
```

Gdb will break on the nearest line of code (a line that is not blank and has code), so you might actually see a different line number (with the starter code, line 19). We are now at a breakpoint in our code. *TIPS: (a) in GDB you can either write run or r, break or b, etc., each command has an abbreviation; (b) tabs work for autocomplete so you don't need to type out the entire file, etc.* We can explore the program in its current state. Let's look and see the local variables (again, type info or i):

```
(gdb) i locals
m = {grid = {_vptr.Grid = 0x555555765be0 <vtable for Grid<MazeCell*>+16>,
    Rows = 0x555555778e70, NumRows = 4}, numRows = 4, numCols = 4}
name = "Shanon Reckinger"
start = 0x555555779090
```

Cool. You might not see the same local variables when you run your code. Also, your name is stored in a global variable in the starter code, so it will not show up as a local. To see that, you can type ("i variables" instead of "i locals"). Let's derference our start pointer and print out the contents, which is the MazeCell where we've dropped you into the labyrinth.

```
(gdb) p *start
$1 = {whatsHere = "", north = 0x0, south = 0x0, east = 0x5555557790e0,
  west = 0x0}
```

Depending on your maze, you may find yourself in a position where you can move in all four cardinal directions, or you may find that you can only move in some of them. The pointers in directions you can't go are all equal to nullptr, which will show up as 0x0 in gdb.

The pointers that indicate directions you can go will all have memory addresses. You can navigate the maze further by choosing one of those or you could back up to the starting maze cell and explore in other directions. It's really up to you! Here I explored mine a bit, and look, I found the potion:

```
(gdb) p *(start->east)
$2 = {whatsHere = "", north = 0x0, south = 0x555555779220,
  east = 0x555555779130, west = 0x555555779090}
(gdb) p *(start->east->east)
$3 = {whatsHere = "", north = 0x0, south = 0x555555779270,
  east = 0x555555779180, west = 0x5555557790e0}
(gdb) p *(start->east->south)
$4 = {whatsHere = "", north = 0x5555557790e0, south = 0x0, east = 0x0,
  west = 0x5555557791d0}
(gdb) p *(start->east->south->west)
$5 = {whatsHere = "Potion", north = 0x0, south = 0x0,
  east = 0x555555779220, west = 0x0}
```

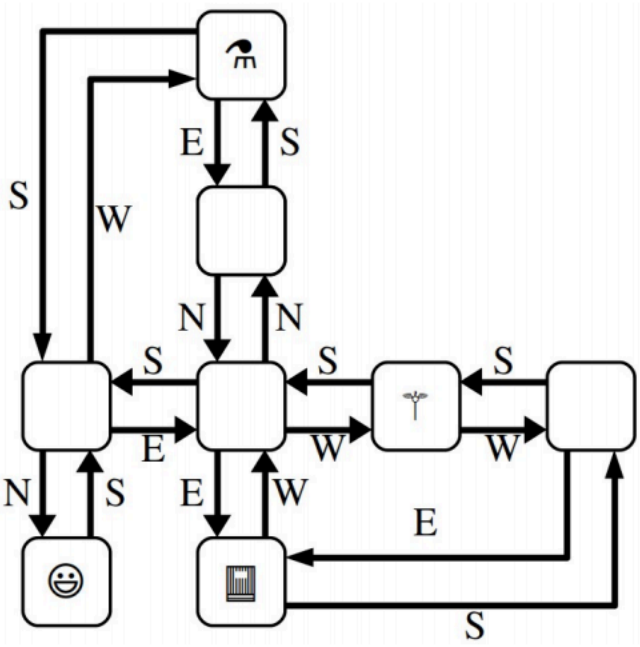
Draw a lot of pictures.
Grab a sheet of paper and map out the maze you're in. There's no guarantee where you begin in the maze – you could be in the upper-left corner, dead center, etc. The items are scattered randomly, and you'll need to seek them out. Once you've mapped out the maze, construct an escape sequence and stash it in the constant **kPathOutOfRegularMaze**, then see if you pass the test. If so, fantastic! You've escaped! If not, you have lots of options. You could step through your isPathToFreedom function to see if one of the letters you entered isn't what you intended and accidentally tries to move in an illegal direction. Or perhaps the issue is that you misdrew your map and you've ended up somewhere without all the items. You could alternatively set the breakpoint at the test case again and walk through things a second time, seeing whether the picture of the maze you drew was incorrect.

To summarize, here's what you need to do:
1. Edit the constant **kYourName** at the top of **EscapeTheLabyrinth.h** with a string containing your name. Don't skip this step! If you forget to do this, you'll be solving the wrong maze!
2. Set a breakpoint at the indicated line in **ExploreTheRegularLabyrinth.cpp**, run the program in gdb.
3. Map out the maze on a sheet of paper and find where all the items are. Once you're done, stop the running program.

4. Find a path that picks up all three items and edit the constant **kPathOutOfRegularMaze** at the top of **EscapeTheLabyrinth.h** with that path. Run the tests (make build_tests, make run_tests to make sure it is right.
5. Finally, save your gdb log (just copy and paste, similar to what you see here in my example) into a text file called gdb-log.txt. You will need to submit this to show your work.

Milestone #5 – Escape your personalized secret *TWISTY* Labyrinth

Now, let's make things a bit more interesting. In the previous section, you escaped from a labyrinth that nicely obeyed the laws of geometry. The locations you visited formed a nice grid, any time you went north you could then go back south, etc. In this section, we're going to relax these constraints, and you'll need to find your way out of trickier mazes that might look something like this:



This maze is stranger than the previous one you explored. For example, you'll notice that these MazeCells are no longer in a nice rectangular grid where directions of motion correspond to the natural cardinal directions. There's a MazeCell here where moving north and then north again will take you back where you started. In one spot, if you move west, you have to move south to return to where you used to be. In that sense, the names "north," "south," "east," and "west" here no longer have any nice geometric meaning; they're just the names of four possible exits from one MazeCell into another. The one guarantee you do have is that if you move from one MazeCell to a second, there will always be a direct link from the second cell back to the first. It just might be along a direction of travel that has no relation to any of the directions you've taken so far.

The second test case in tests.cpp contains code that generates a twisty labyrinth personalized with the kYourName constant. As before, you'll need to find a sequence of steps that will let you

collect the three items you need to escape. In many regards, the way to complete this section is similar to the way to complete the previous one.

Set a breakpoint in the indicated spot in **ExploreTheTwistyLabyrinth.cpp** and use gdb to explore the maze. Unlike the previous section, though, in this case you can't rely on your intuition for what the geometry of the maze will look like. For example, suppose your starting location allows you to go north. You might find yourself in a cell where you can then move either east or west. One of those directions will take you back where you started, but how would you know which one? This is where memory addresses come in. Internally, each object in C++ has a memory address associated with it. Memory addresses typically are written out in the form @0xsomething, where something is the address of the object. You can think of memory addresses as sort of being like an "ID number" for an object – each object has a unique address, with no two objects having the same address. When you pull up the gdb view of a maze cell, you should see the MazeCell memory address. For example, suppose that you're in a maze and your starting location has address 0x7fffc8003740 (the actual number you see will vary on a case by case), and you can move to the south (which way you can go are personalized to you based on your name, so you may have some other direction to move). If you follow the south pointer, you'll find yourself at some other MazeCell. One of the links out of that cell takes you back where you've started, and it'll be labeled 0x7fffc8003740. Moving in that direction might not be productive – it just takes you back where you came from – so you'd probably want to explore other directions to search the maze.

It's going to be hard to escape from your maze unless you draw lots of pictures to map out your surroundings. To trace out the maze that you'll be exploring, we recommend diagramming it on a sheet of paper as follows. For each MazeCell, draw a circle labeled with the memory address, or, at least the last 4-5 characters of that memory address. (Usually, that's sufficient to identify which object you're looking at). As you explore, add arrows between those circles labeled with which direction those arrows correspond to. What you have should look like the picture above, except that each circle will be annotated with a memory address. It'll take some time and patience, but with not too much effort you should be able to scout out the full maze. Then, as before, find an escape sequence from the maze!

To recap, here's what you need to do:
1. Set a breakpoint at the indicated line in **ExploreTheTwistyLabyrinth.cpp**, run the gdb on the appropriate executable.
2. Map out the twisty maze on a sheet of paper and find where all the items are and how the cells link to each other. Once you're done, stop the running program.
3. Find an escape sequence, and edit the constant **kPathOutOfTwistyMaze** at the top of of **EscapeTheLabyrinth.h** with that constant. Run the tests again and see if you've managed to escape!
4. Finally, save your gdb log (just copy and paste, similar to what you see here in my example) into a text file called **gdb-log.txt**. You will need to submit this to show your work.

Some notes on this problem:

- The memory addresses of objects are not guaranteed to be consistent across runs of the program. This means that if you map out your maze, stop the running program, and then

start the program back up again, you are not guaranteed that the addresses of the MazeCells in the maze will be the same. The shape of the maze is guaranteed to be the same, though. If you do close your program and then need to explore the maze again, you may need to relabel your circles as you go, but you won't be drawing a different set of circles or changing where the arrows link.

- You are guaranteed that if you follow a link from one MazeCell to another, there will always be a link from that second MazeCell back to the first, though the particular directions of those links might be completely arbitrary. That is, you'll never get "trapped" somewhere where you can move one direction but not back where you started.
- Attention to detail is key here – different MazeCell objects will always have different addresses, but those addresses might be really similar to one another. Make sure that as you're drawing out your diagram of the maze, you don't include duplicate copies of the same MazeCell.
- The maze you're exploring might contain loops or cases where there are multiple distinct paths between different locations. Keep this in mind as you're exploring or you might find yourself going in circles!
- Remember that you don't necessarily need to map out the whole maze. You only need to explore enough of it to find the three items and form a path that picks all of them up.

At this point, you should have a solid command of how to use gdb to analyze linked structures. You know how to recognize a null pointer, how to manually follow links between objects, and how to reconstruct the full shape of the linked structure even when there's bizarre and unpredicable cross-links between them. We hope you find these skills useful as you continue to write code that works on linked lists and other linked structures (so many)!

Using gdb

This is a really nice tutorial on how to use gdb, including a video walkthrough:
https://web.stanford.edu/class/archive/cs/cs107/cs107.1194/resources/gdb

Requirements

1. You may not change the API provided in grid.h. You must keep all private and public member functions and variables as they are given to you. If you make changes to these, your code will not pass the test cases. You may not add any private member variables but you may add private member helper functions.
2. In grid.h, no additional C++ containers are allowed. No additional header files are allowed to be included. The implementation of a grid must follow the diagram shown in this handout, and as defined in the provided "grid.h" file. In other words, a grid must remain a pointer to an array of ROW structures, where each ROW contains a pointer to an array of elements of type T. You cannot switch to a vector-based implementation, nor other data structures.
3. Do not change maze.h in any way. In the test cases, we don't use your maze.h. Anything you do in maze.h will be ignored.
4. You must have a clean valgrind report when your code is run (tests on grid.h and running with the personalized *regular* maze). Check out the makefile to run valgrind on your

code. All memory allocated (call new) must be freed (call delete). The test cases run valgrind on your code. (NOTE: when testing the twisty maze, you will NOT have a clean valgrind report and it is not required that memory for the twisty maze is freed.)

5. Your tests.cpp should have 10s-100s of assertions for each public member function. You need to put in significant effort into testing. Check out the Mimir rubric for how this will be graded.
6. No <added> global or static variables.

Citations/Resources

Assignment is credited to Keith Schwarz at Stanford University (Labyrinth). Additionally, Grid.h is based on work by Joe Hummel at University of Illinois at Chicago and variations made my Kai Bonsol.