Students:
Section 14.6 is a part of 1 assignment:
**Programming Project 06 - Escape the Labyrinth - Grid Pointer Mazes**

This assignment's due date has passed. Activity will still be recorded, but will not count towards this assignment (unless the due date is changed). See this article for more info.

Includes: 🟩 zyLab

Due: 11/21/2024, 11:59 PM CST

Instructor created

# 14.6 P06 - Escape the Labyrinth - Grid Pointer Mazes

## Project Background

Lab 11 involves implementing a **LetterGrid** class, which is an abstracted two-dimensional array of characters, i.e. it is actually a heap-allocated dynamic array of pointers, where each pointer references a heap-allocated array of characters. This is an abstraction because the low-level details are all contained in the class definition, allowing the programmer to simply use it as a two-dimensional grid. This should familiarize you with writing a class in C++, specifically for an abstracted two-dimensional basic data structure and the related memory management tasks, i.e. allocating and deallocating memory on the heap in the constructors and destructors.

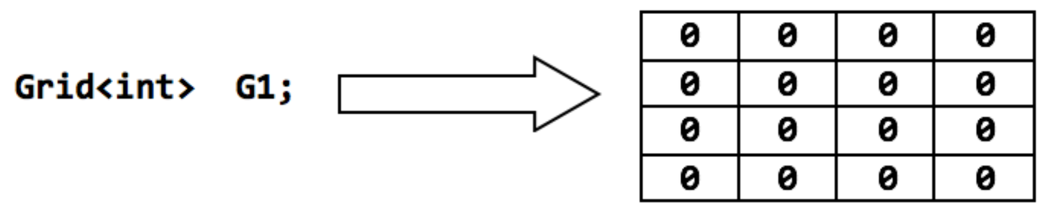
The next step in your journey is to extend the **LetterGrid** implementation to a general-typed **Grid** abstraction, and use it to represent linked structures that can have a prescribed structure or are arbitrarily shaped. Let the segmentation faults fly as they are bound to happen here. Remember, a segmentation fault is a result of accessing memory you do not have permission to access.

This is a unique Programming Project, in that many of the the primary tasks do not involve implementing program functionality. The first task requires implementing a templated **Grid** class (similar to the **LetterGrid**), which allows any type for the grid elements (i.e. whereas the **LetterGrid** class only allows character elements, the **Grid** class allows arbitrarily-typed elements). After that is when it gets fun: unleash the power of the *gdb debugger*, specifically when working with linked structures, to escape a labyrinth with rooms that are connected by pointers. Your experience should be filled with setbacks and rewards, culminating in the thrill of solving the pointer maze!

## Task #1: write grid.h

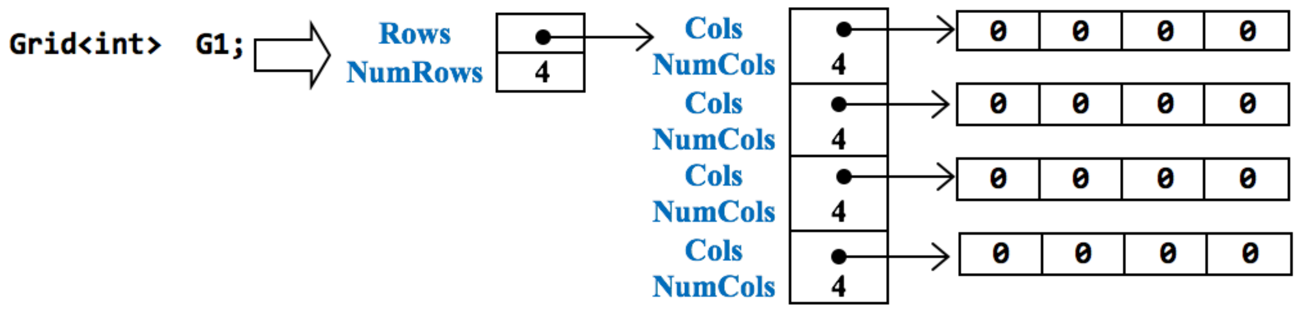
The goal of this task is to define a class **Grid<T>** explicitly designed to support a two-dimensional grid. We need this abstraction in order to create mazes and escape labyrinths later. A grid is defined to have a given number of rows and columns, and the resulting elements are initialized to C++'s natural default value. For example, the figure below shows the default (4x4) integer grid.

Figure 14.6.1: Default integer Grid



Keep in mind that this is an abstraction; the actual implementation of the grid is quite different. A **Grid** object is actually a heap-allocated array of pointers, where each pointer references a heap-allocated array of elements of the declared type. For convenience, the version of the **Grid** class used for this project does not require dynamic growth, but it would be possible with this implementation. Thus, the **Grid** class has a more general usage than how it is used in the following tasks. The figure below diagrams the actual implementation for an integer **Grid**, which involves pointers to C-style arrays.

Figure 14.6.2: Implementation diagram for an integer Grid



The task consists of implementing a set of member functions for class **Grid<T>**. It is required to implement **Grid** as described above and as defined in the provided *Grid.h* header file. Here are the relevant declarations from *Grid.h*:

```
template<typename T>
class Grid {
private:
  struct ROW {
    T*   Cols;      // array of column elements
    size_t NumCols;  // total # of columns (0..NumCols-1)
  };

  ROW* Rows;       // array of ROWs
  size_t  NumRows;   // total # of rows (0..NumRows-1)
...
```

You are free to add additional member variables to improve the implementation, as well as private helper functions. However, do not change the overall implementation of a grid: it 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.

By default, a grid is 4x4. Here’s the code for the default constructor that creates a 4x4 grid. 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() {
    Rows = new ROW[4];  // 4 rows
    NumRows = 4;

    // initialize each row to have 4 columns:
    for (size_t r = 0; r < NumRows; ++r) {
      Rows[r].Cols = new T[4];
      Rows[r].NumCols = 4;

      // initialize the elements to their default value:
      for (size_t c = 0; c < Rows[r].NumCols; ++c) {
        Rows[r].Cols[c] = T();  // default value for type T:
      }
    }
  }
```

The parts you need to complete are marked with *TODO* statements in *grid.h*.

Note that the **Grid** class is VERY similar to the **LetterGrid** class implemented in Lab 11, with a few key differences that gives the **Grid** class more general applicability:

1. **Grid** is a templated class, so that elements of the two-dimensional structure can have any type, where the **LetterGrid** class is limited to character elements only.
2. Whereas the **LetterGrid** class utilizes separate setter/getter functions to assign/access cell elements, the **Grid** class combines these two operations by overwriting the parentheses operator; consider the following example:

```
Grid<int> G;   // default 4x4 integer grid, with all elements set
to zero
G(1,2) = 5;    // assign element in row 1, column 2 to the value
5
cout << G(1,2) // access element at row 1, column 2 and print it
to screen
```

Thus, whereas your task can be thought of as *implementing a class from scratch*, it can also be framed as an *extension of the **LetterGrid** class to have more general applicability*.

The remaining tasks depend on a fully functional **Grid** class in *grid.h*. The autograded test cases provided for Task 2+, as well as the files provided to explore the mazes, will produce a segmentation fault if you run them without a properly implemented *grid.h*. Thus, it is imperative that the methods in *grid.h* are fully tested and pass all of the autograded test cases before moving on.

---

---

## Trapped in a Labyrinth - How to Escape

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:

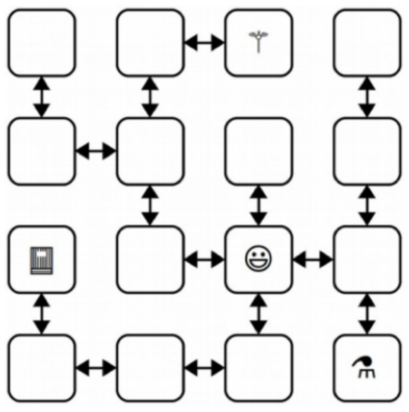
- The Spellbook (📕), which contains the spell that must be cast in order to escape.
- The Potion (⚗), containing the arcane compounds that power the spell.
- The Wand (⚚), which concentrates focus to make the spell work.

Once you have all three items, you can cast the spell and 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:

Figure 14.6.3: A sample 4x4 labyrinth



Your starting position is marked with a smiley face, and the positions of the three items are marked with similarly cute emojis. In this example, the **MazeCell** you begin at has 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 has 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 for casting 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.

## Remaining Tasks & Files

There are a few more tasks to this project. The next task is a coding problem, and the final two tasks are exercises that require you to escape labyrinths using the *gdb debugger*. It is strongly recommend that you complete the tasks in the order, as they build on top of one another.

There isn't a lot of code to write for the rest of the programming project. However, there is a lot left to do. All of 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 requires additional code development. You need to write **isPathToFreedom()** in Task #2. This is also where you put the escape solutions to your personalized mazes.
- **ExploreTheRegularLabyrinth.cpp** – This is the file you use to solve the regular maze using gdb in Task #3. You don't need to write any code here, instead it contains the primary **main()** application for Task #3.
- **ExploreTheTwistyLabyrinth.cpp** – This is the file you use to solve the twisty maze using gdb in Task #4. You don't need to write any code here, instead it contains the primary **main()** application for Task #4.

## Task #2: write isPathToFreedom()

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 the starting location in the maze and a string made purely from the characters 'N', 'S', 'E', and 'W', then returns whether that path leads to an escape from the maze. A path leads to an escape from the maze if BOTH of the following are true:

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'), return false.

**Some notes on this problem:**

- Every **MazeCell**’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.
- 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.

## Task #3: escape your personalized secret Labyrinth

Your next task is to escape from a labyrinth that’s specifically constructed for you. The provided starter code must use **your netID** 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.” The uniqueness of your maze is guaranteed by the uniqueness of your netID. 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, **kYourNetID**, is a spot for your netID. In the starter code, it’s marked with a TODO message. Edit this constant so that it contains your newID. It is required that this constant variable is a perfect match to your netID. Submissions will be manually inspected to ensure **kYourNetID** is a perfect match to each students actual netID. Thus, it is imperative that you are 100% sure what your netID is. To help you out, here are some examples:

- Professor Reckinger's email is **scotreck**@uic.edu, so
  Professor Reckinger's netID is **scotreck**
- Student Sparky's email is **sspark211**@uic.edu, so
  Student Sparky's netID is **sspark211**

The second constant variable is **kPathOutOfRegularMaze**, which will eventually be a string representing a path to freedom. However, in the starter code, **kPathOutOfRegularMaze** is simply a TODO message, so it's not going to let you escape from the labyrinth.

To provide context on how the constant variables **kYourNetID** and **kPathOutOfRegularMaze** are used, here is a sample test case function:

```
bool test_case() {
  Maze m(4,4); // constructs a default Maze
  MazeCell* start = m.mazeFor(kYourNetID); // Maze m is now
unique for input kYourNetID
  return isPathToFreedom(start, kPathOutOfRegularMaze);
  // returns true if kPathOutOfRegularMaze leads to an escape
from maze m from start
  // otherwise, returns false
}
```

This test case generates a personalized labyrinth based on the **kYourNetID** constant and returns a pointer to one of the cells in that maze. It then checks whether the constant **kPathOutOfRegularMaze** is a sequence that leads to an escape from the maze.

Your task is to edit the string **kPathOutOfRegularMaze** with an escape sequence once you find one. To come up with a path out of the labyrinth, you will use the trusted debugging tool gdb!

A reminder tutorial on using gdb, specifically geared for this project, is provided in the next section. Note that you are required to submit your gdb log in a text file, so make sure to save your process.

## Tutorial for using gdb to navigate a pointer maze

Note: the following tutorial uses **kYourNetID = "scotreck" .**

Navigating a regular pointer maze involves running 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
```

The program output should make it clear that you have NOT escaped the labyrinth. Now, run the executable under gdb:

```
>> gdb ExploreTheRegularLabyrinth.exe
```

Then, set a breakpoint at the line labeled in a comment in **ExploreTheRegularLabyrinth.cpp** (likely line 15) and run the program:

```
(gdb) b ExploreTheRegularLabyrinth.cpp:15
(gdb) r
```

The break occurs on the nearest line of code that is not blank, so you might actually see a different line number for the break.

Next, let's explore the program in its current state by getting information about the local variables (sample output is also shown):

```
(gdb) i locals
m = {_vptr.Maze = 0x55863f0afbf8 , grid = {_vptr.Grid =
0x55863f0afc18 +16>,
    Rows = 0x55863ff96eb0, NumRows = 4}, numRows = 4, numCols =
4, kNumTwistyRooms = 12}
start = 0x55863ff970d0
```

Now, let's dereference the start pointer and print out the contents, which is the **MazeCell** where you have been dropped into the labyrinth:

```
(gdb) p *start
$1 = {whatsHere = "", north = 0x0, south = 0x55863ff97210, east =
0x0, west = 0x0}
```

What you see at this point depends on your personalized maze and the starting location within it. 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 the possible directions. The pointers in directions you can't go are all equal to **nullptr**, which shows up as **0x0** in gdb.

The pointers that indicate directions you can go all have non-null memory addresses. You can navigate the maze further by choosing one of those directions to pursue. Proceeding from here is really up to you! Here, the maze was explored a bit, and look, we found the Spellbook!...

```
(gdb) p *(start->south)
$2 = {whatsHere = "", north = 0x55863ff970d0, south =
0x55863ff97350, east = 0x0, west = 0x0}
```

```
(gdb) p *(start->south->south)
$3 = {whatsHere = "", north = 0x55863ff97210, south = 0x0, east =
0x55863ff973a0, west = 0x0}
(gdb) p *(start->south->south->east)
$4 = {whatsHere = "", north = 0x55863ff97260, south =
0x55863ff974e0, east = 0x55863ff973f0, west = 0x55863ff97350}
(gdb) p *(start->south->south->east->north)
$5 = {whatsHere = "", north = 0x55863ff97120, south =
0x55863ff973a0, east = 0x0, west = 0x0}
(gdb) p *(start->south->south->east->north->north)
$6 = {whatsHere = "Spellbook", north = 0x0, south =
0x55863ff97260, east = 0x55863ff97170, west = 0x0}
```

**Draw a lot of pictures.** Grab a sheet of paper and map out your maze. There's no guarantee where you begin in the maze – you could be in the upper-left corner, near the 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.

You are required to submit a diagram of your maze, called **diagram_regular_[netID].pdf**, where **[netID]** is replaced with your netID. Here is a sample diagram for the maze with **kYourNetID = "scotreck".**

**To summarize Task #3, here's what you need to do:**

1. Edit the constant **kYourNetID** at the top of **EscapeTheLabyrinth.h** with a string containing your exact netID. Don't skip this step! If you forget to do this, you'll be solving the wrong maze!
2. Open the program under gdb, set a breakpoint at the indicated line in **ExploreTheRegularLabyrinth.cpp**, and run the program.
3. Map out the maze on a sheet of paper and find where all the items are. Once you're done, quit gdb (make sure to save your gdb log, since you need to submit it; see #5 below). Take a photo of your map diagram if you drew it on paper, or save it as a .pdf if you drew it on a digital device, and be prepared to include your map diagram when you submit your code to Gradescope.

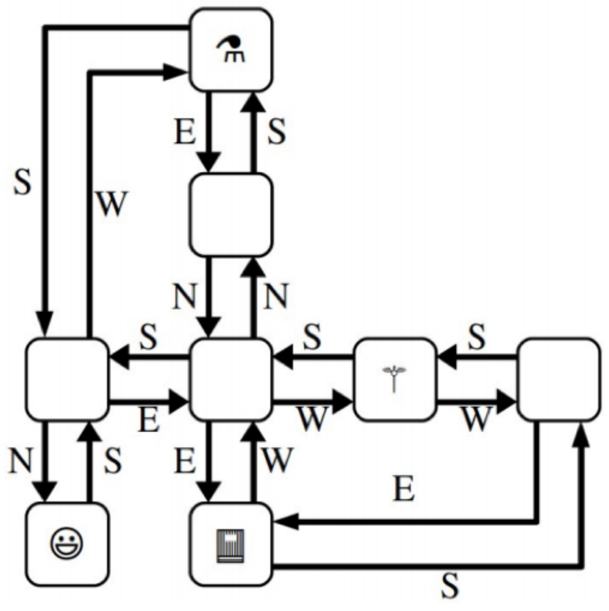
4. Find a path that picks up all three items and edit the constant **kPathOutOfRegularMaze** at the top of EscapeTheLabyrinth.h with that path. Run the program (not using gdb; instead **>> make build_reg, >> make run_reg** ) to check if you have managed to escape.

5. Finally, save your gdb log (just copy and paste, similar to what you see in the tutorial above) into a text file called **gdblog_regular_[netID].txt**. To show your work, be prepared to include **gdblog_regular_[netID].txt** when you submit your code to Gradescope.

## Task #4: escape your personalized secret *TWISTY* Labyrinth

As your final task, 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 a trickier maze that might look something like this:

Figure 14.6.4: A sample twisty labyrinth



This maze is stranger than the previous one you explored. For example, notice that these **MazeCell**s are no longer in a nice rectangular grid where directions of motion correspond to the natural cardinal directions. There is 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.

As before, you need to find a sequence of steps that lets you collect the three items you need to escape. In many regards, the way to complete this section is similar to the way you completed 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 **0x***something*, where *something* is the address of the object. In this sense, the memory address is an “ID number” for the 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 case-by-case), and you can move to the south (which way(s) you can actually go are personalized to you based on your netID, 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 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.

The third constant variable in EscapeTheLabyrinth.h is **kPathOutOfTwistyMaze**, which will eventually be a string representing a path to freedom for the twisty maze. However, in the starter code, **kPathOutOfTwistyMaze** is simply a TODO message, so it’s not going to let you escape from the labyrinth.

To provide context on how the constant variables **kYourNetID** and **kPathOutOfTwistyMaze** are used, here is a sample test case function (VERY similar to the test case for the regular maze above):

```
bool test_case() {
  Maze m(4,4); // constructs a default Maze
  MazeCell* start = m.twistyMazeFor(kYourNetID); // Twisty Maze m
unique for kYourNetID
  return isPathToFreedom(start, kPathOutOfTwistyMaze);
  // returns true if kPathOutOfTwistyMaze leads to an escape from
maze m from start
  // otherwise, returns false
}
```

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, it is recommended that you diagram it on a sheet of paper (or tablet) as follows. For each MazeCell, draw a square labeled with the memory address, or, at least the last 4-5 characters of that memory address as that is usually sufficient to identify which object you're looking at. As you explore, add arrows between the squares labeled with which direction those arrows correspond to OR have the arrows emanate from the side of the square associated with the pointer name (i.e. north, east, south, or west). What you have should look like the picture above, except that each square 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!

NOTE: as with any program, memory locations (i.e. pointers) will change between different runs of the program. Thus, you must complete a full scouting of your maze within a single program run using gdb in order to fully navigate all cells and accurately record all the pointer memory address locations. If you need to restart the program (e.g. leaving the terminal inactive for a time causes it to sleep, which automatically terminates the program), the structure of the maze does not change, but the memory addresses for the cells and, thus, the pointer values do change; in this case, you will need to update all memory address labels on your diagram (but you do not need to redraw the twisty maze from scratch).

You are required to submit a diagram of your twisty maze, called **diagram_twisty_[netID].pdf**, where **[netID]** is replaced with your netID. Here is a **sample diagram** for the twisty maze with **kYourNetID = "scotreck".**

**To recap Task #4, here's what you need to do:**

1. Double-check that the constant **kYourNetID** at the top of EscapeTheLabyrinth.h matches your exact netID. Don't skip this step! If you forget to do this, you'll be solving the wrong maze!
2. Open the program under gdb, set a breakpoint at the indicated line in ExploreTheTwistyLabyrinth.cpp, and run the program.
3. Map out the twisty maze on a sheet of paper and find where all the items are. Once you're done, quit gdb (make sure to save your gdb log, since you need to submit it; see #5 below). Take a photo of your map diagram if you drew it on paper, or save it as a .pdf if you drew it on a digital device, and be prepared to include your map diagram when

you submit your code to Gradescope.

4. Find an escape sequence and edit the constant **kPathOutOfTwistyMaze** at the top of EscapeTheLabyrinth.h with that path. Run the program (not using gdb; instead **>> make build_twisty, >> make run_twisty** ) and see if you've managed to escape!
5. Finally, save your gdb log (just copy and paste, similar to tutorial above) into a text file called **gdblog_twisty_[netID].txt**. To show your work, be prepared to include **gdblog_twisty_[netID].txt** when you submit your code to Gradescope.

**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 **MazeCell**s 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 will need to relabel your squares as you go, but you won't be drawing a different set of squares or changing where the arrows link.
- It is 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 are completely arbitrary in a twisty maze. 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 explore 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.

Upon completion of this project, you should have a solid command of how to use gdb to analyze linked structures, 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 are bizarre and unpredictable cross-links between them. We hope you find these skills useful as you continue to write code that works on linked structures going forward in your academic and professional endeavors!

## Requirements

- Do not change the Application Programming Interface (API) provided in grid.h. 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.
- 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. 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.
- **VERY IMPORTANT**: you must set **kYourNetID** to match your exact netID before finding an escape path for Tasks #3 and #4. Submissions that solve a maze generated with **kYourNetID** not a perfect match to the student's actual netID will incur at 25 point penalty if it is a minor difference (e.g. one character off, upper case letter use instead of lower case, etc.). Major differences OR failure to update **kYourNetID** at all (i.e. keeping the TODO statement where your netID should go) will result in a 50 point penalty. If the **kYourNetID** matches another student's netID, then the submission will receive a total score of zero for the project as that would be an academic integrity issue.
- 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 (i.e. calls to **new**) must be freed (i.e. calls to **delete**).  The test cases run valgrind on your code.  (NOTE: when testing the twisty maze, you will NOT have a clean valgrind report as it is designed to NOT have a systematic way to traverse the maze for memory clean up, so it is NOT required that memory for the twisty maze is freed.) No <added> global or static variables.
- Coding style issues are manually graded using deductions, worth up to 25% of the total project score. Style points are graded for following the course standards and best practices laid out in the syllabus, including a header comments, meaningful identifier names, effective comments, code layout, functional decomposition, and appropriate data and control structures.
- Programming projects must be completed and submitted individually. Sharing of code between students in any fashion is not allowed. Use of any support outside of course-approved resources is not allowed, and is considered academic misconduct. Examples of what is allowed: referencing the zyBook, getting support from a TA, general discussions about concepts on piazza, asking questions during lecture, etc. Examples of what is NOT allowed: asking a friend or family member for help, using a "tutor" or posting/checking "tutoring" websites (e.g. Chegg), copy/pasting portions of the project description to an AI chatbot, etc. Check the syllabus for Academic Integrity policies.

Violations of this requirement will receive a manually graded deduction, and may be reported to the Dean of Students office.

## Tips for Success on this Programming Project

- **Unlike any other programming project** - The first two tasks involve developing code for a **Grid** class and then writing a function to test a path through a maze built on a **Grid** object. These first two tasks are geared toward practicing class implementation and usage, and would fit right in with other programming projects in this course. However, the final two tasks are unique, in that there is *no code development*; instead, you will use the gdb debugger to navigate a pointer maze to find a path that allows you to escape the maze. It is thrilling, challenging, and rewarding.
- **NOT all tasks/functions are created equal**- As described above, the Programming Tasks for this project are diverse in their nature. There is a great deal of starter code provided. The level of scaffolding is designed to lead you through the tasks that leaves room for creatively applying course concepts and tools. A natural consequence is that..
  - *some Tasks are straightforward and some Tasks are challenging;*
  - *some Tasks ask you to do one thing specifically and some Tasks are purposefully open-end with multiple components;*
  - *some Tasks test your ability to follow instructions verbatim and some Tasks require you to practice problem-solving and critical-thinking skills;*
  - *some Tasks may only take you a few minutes and some Tasks may require an initial attempt followed by a break to do something else only to come back multiple times to tackle the challenge;*

## Submission & Grading

Develop your code in the IDE below. The provided starter code is substantial. Thus, make sure to read all instructions carefully and only make changes to the specific files referenced in the project description.

Use the terminal to test your code interactively as you develop your program. Do NOT use the "Run" button; instead, compile your code using the make utility (e.g. "**make build_reg**" or "**make build_twisty**") and run the resulting executable (e.g. "**make run_reg**" or "**make run_twisty**") in the terminal to test your code interactively as you develop your program. There are also valgrind targets in the makefile (e.g. "**make valgrind_reg**" or "**make valgrind_twisty**") to check for memory leaks or errors. There is also a provided test case suite for you to test your grid.h implementation from the terminal, which can be run using "**make build_tests**" and "**make run_tests**" or "**make valgrind_tests**". Use the "Submit for grading" button to test your code against the suite of autograded test cases, which also submits your program for functionality grading.

When you are ready to formally submit, download the following files from the zyLab IDE file tree: **grid.h** and **EscapeTheLabyrinth.h**. Then, upload these files to the Project 06 Gradescope submission form; look for *Project 06*. Additionally, be prepared to also *submit two gdb log files* (one for the regular maze, **gdblog_regular_[netID].txt** and one for the twisty maze, **gdblog_twisty_[netID].txt**) and *two diagrams mapping out your mazes* (one for the regular maze, **diagram_regular_[netID].pdf,** and one for the twisty maze, **diagram_twisty_[netID].pdf**). If you fail to submit to Gradescope, you have not formally submitted anything, and the resulting score is a zero for the project.

The standard deadline for the project is **Thursday, November, 21st**. As detailed in the syllabus, early submissions receive +1 extra credit point/day early, up to a maximum of +3 points. Also, students are allocated a certain number of late days throughout the term, where no late penalty is applied. Students can use their late days whenever they choose, simply by submitting their work to Gradescope after the deadline. However, no more than two late days are allowed for each project.

The grading breakdown is as follows:

- **84 points for functionality** - autograder points with manual deductions for any style issues (check syllabus for how style is graded) and program requirements not adhered to. Style is checked for the code you write in grid.h and EscapeTheLabyrinth.h.
  - **24 points** - grid.h implementation
  - **20 points** - isPathToFreedom()
  - **15 points** - did you escape your regular maze?
  - **15 points** - did you escape your twisty maze?
  - **10 points** - valgrind memory leak and error check
- **6 points for gdb logs** - consistent with mapping out your personalized mazes
- **10 points for diagrams** - neatly map out your personalized mazes, clearly identifying memory addresses, pointers, and items
- **early submission extra credit** - +1 point/day early with a maximum of +3 extra credit points.

## Citation/Inspiration

The idea and much of the source code for the Labyrinth portion of this programming project is greatly inspired and/or directly borrowed from Keith Schwarz at Stanford University. Additionally, much of the grid.h work is greatly inspired by efforts from Joe Hummel and Shanon Reckinger at the University of Illinois Chicago.

## Copyright Statement

LAB ACTIVITY | 14.6.1: Escape the Labyrinth - Grid class, pointer mazes, and exploration using gdb

84 / 84

Run History Tutorial

EscapeTheLabyrinth.h

```
#include <utility>
#include <random>
#include <set>
#include "grid.h"
#include "maze.h"
using namespace std;

// My netID
const string kYourNetID = "sgang29";

// Paths to get out of the mazes
const string kPathOutOfRegularMaze = "ESWSSNNENEESSWWS";
const string kPathOutOfTwistyMaze = "WWEESNWWWENWW";

bool isPathToFreedom(MazeCell *start, const string& moves) {
    // booleans to hold if you have the item or not
    bool spellbook = false;
    bool wand = false;
    bool potion = false;
```

```
20
```

DESKTOP CONSOLE

Submit for grading

Coding trail of your work What is this?

```
11/14 R0,0,32,0,34,0,0,0,0,39,44,0,0,0,44,44,54,69 F
84 S84 U84 min:126
```

Latest submission - 12:23 AM CST on 11/17/24

Submission passed all tests ✓

Total score: 84 / 84

✓ Only show failing tests *(20 tests hidden)*

**Open submission's code**

5 previous submissions

| | | |
|---|---|---|
| 11:29 PM on 11/16/24 | 84 / 84 | **View** |
| 12:04 AM on 11/15/24 | 84 / 84 | **View** |
| 10:03 PM on 11/14/24 | 69 / 84 | **View** |
| 9:07 PM on 11/14/24 | 54 / 84 | **View** |
| 9:07 PM on 11/14/24 | 44 / 84 | **View** |

Trouble with lab?

Feedback?

Activity summary for assignment: Programming Project 06 - Escape the Labyrinth - Grid Pointer Mazes

84 / 84 points

Due: 11/21/2024, 11:59 PM CST

This assignment's due date has passed. Activity will still be recorded, but will not count towards this assignment (unless the due date is changed). See this article for more info.

Completion details