# Project Maze

*Due: Wednesday, September 21, 2016*

## 1 Introduction

The pipe system at the Marine Life institute is as complex as it is vast. Navigating through it is an especially daunting task for a fish suffering from short term memory loss like Dory. With Bailey's echolocation out of commission, it's up to you to map out the pipe system and help guide Dory to the open ocean exhibit!

## 2 Assignment

This C programming assignment contains two parts: first you will write a program *generator*, which generates mazes; then you will write a program *solver*, which solves those mazes.

To get started, run

```
cs033_install maze
```

to copy the stencil for this project into *~/course/cs033/maze*.

## 3 Stencil

You have been provided with the following stencil files:

`generator.c` - contains main functions for generator part of the assignment that you will need to fill in.
`solver.c` - contains main functions for solver part of the assignment that you will need to fill in.
`common.c` - contains necessary functions for both parts that you will need to complete.

Additionally, you have also given the header files `generator.h`, `solver.h`, and `common.h`.

`common.h` - contains definitions for the Direction Enum and an outline for the maze room struct that you will need to fill in.

## 4 Generator

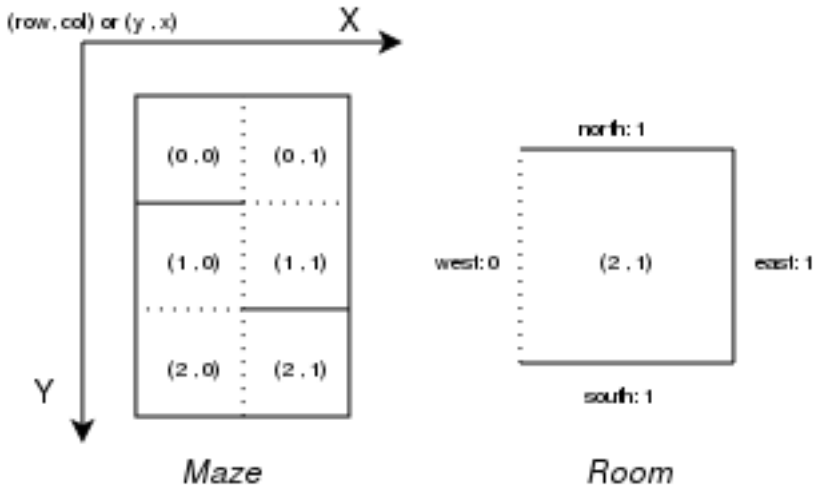
Your generator program should take three arguments:

```
./generator <output maze file> <number of rows> <number of columns>
```

where the first argument is the name of the output file, and the next two arguments are the dimensions of the maze. Your code should be able to handle variable size mazes. Room indices are counted starting from zero at the upper-left corner; thus, the lower-right corner of a $10 \times 25$ maze will have coordinates $(9, 24)$.

In the Life lab, you learned how to index into a one-dimensional array to represent a two-dimensional array. In this project, the representation of your maze is entirely up to you. While the stencil uses a one-dimensional representation, you may choose to use either a one-dimensional or two-dimensional array.

Each room in the maze has four connections, one for each neighboring room. These connections can either be walls or openings. Rooms which are on the edges of the maze should always have a wall on that border - for example, a room on the south border of the maze should always have a wall on its southern end.




## 4.1 Encoding a Maze in Memory

You will need to represent the maze in two different ways: first, as a data structure within your program; second, as text in a file on disk. In this half of the Maze assignment, you will only write mazes from memory to files. In the second half, you will read mazes from files back into memory.

### 4.1.1 Within the Program

Representation of a maze within your program is up to you. However, for each room you will need to keep track of the following:

- the `row` and `column` of the room.
- whether or not the room has been visited.
- for each connection of the room, whether that connection is a wall, opening, or is uninitialized.

You should use a `struct` to represent each room.

### 4.1.2 Within a File

Mazes will be written to files on disk by your generator program. It is imperative that your maze files conform to our file specification.

Each line of the output file will correspond to a row of rooms.

For each room, you need a concise way to write that room's set of connections. Since there are four possible directions, with two options per direction(wall/opening), there are a total of 16 unique combinations of room connections.

Ideally, we would only need to write one character per room. Since there are 16 options, we can store the output as a one-byte hexadecimal number.

So how can you convert your connections (currently stored as **0** for an opening or **1** for a wall) into hexadecimal values? We already have 1 field per connection, and conveniently, hexadecimal numbers are made up of four bits.

- the highest-order bit represents the **east** connection
- the next-highest bit represents the **west** connection
- the next-lowest bit represents the **south** connection
- the lowest-order bit represents the **north** connection

For example, a room with openings to the north and south and walls to the east and west would be stored as 1100.

## 4.2 Translating Between Representations

Your `generator` must translate from the program representation of the maze to the file representation. To do this, you need a way to convert from a four-bit binary number to one hexadecimal character.

There are a few ways to do this, such as bit-shifting, but a simpler way is to convert by hand to decimal and delegate the hexadecimal conversion to `fprintf()`. Each bit corresponds to a power of two:

| $E$ | $W$ | $S$ | $N$ |
|---|---|---|---|
| 8 | 4 | 2 | 1 |

As per the example before, a room with connections 1100 would be represented with $1(8) + 1(4) + 0(2) + 0(1) = 12$, or `0xC` in hexadecimal.

Once you have a decimal representation of your room, there is a lovely trick you can pull to print out a hex value: instead of giving `fprintf() "%d"`, you can give it `"%x"` and it will print your decimal value as a hex character.

As an example, the following is a sample maze representation in a file, where each hexadecimal number represents a room's connections:

```
597333331395397313333313b
c6339595adccd639633b59639
cd53286a70ac619c5333a639c
c4a59e5396969cc6ad51b53ac
ce5a632bc5a5ac61b4ac5a738
432339ddcc5a5adc5ad6a5958
cd5396accccdc58cc5239c6ac
cccd633accc4aec6a639cc59c
68c43339cccc5949719cc6acc
7a6a7332a6a6a6a63a6a633ae
```

### 4.3 Generation Algorithm

There are a few ways to generate a maze, but the simplest uses the *drunken-walk algorithm*. This algorithm recursively constructs a maze by visiting each room, and then randomly choosing connections for that room by visiting the adjacent rooms.

Starting at room (0,0), drunken-walk randomizes an order in which to visit the adjacent rooms, and then visits each of those rooms.

- If the neighboring room has not yet been visited, then the algorithm recurs on that room.
- If it has already been visited and already has a connection defined in the given direction, then the current room should copy that connection to be consistent.
- Otherwise, a connection should be randomly chosen in that direction and stored in the current room. Note that increasing the probability of choosing a wall increases the difficulty of the maze. To mimic the demo exactly, **always choose to make a wall**.

If implemented correctly, there is guaranteeably a path from any room to any other room in the maze.

Pseudocode:

```
drunken_walk(row, col):
    r = rooms[row][col]
    set r.visited to true
    for each direction dir in random order:
        if (row + row_offset of dir, col + col_offset of dir) is out of bounds:
            store a wall in r at direction dir
        else:
            neighbor = rooms[row + row_offset of dir][col + col_offset of dir]
            if neighbor has not yet been visited:
                store an opening in r at direction dir
                drunken_walk(neighbor.row, neighbor.col)
            else:
                opposite_dir = the opposite direction of dir
                if neighbor has a connection c (opening or wall) in direction opposite_dir:
                    store c in r at direction dir
                else:
                    store a wall in r at direction dir
```

### 4.3.1 Random Number Generation

To ensure that the maze generated by your program is not always the same, you'll need to use random number generation. One way to generate a random integer in C is to use the `rand()` function, which takes no arguments and returns an integer between 0 and `RAND_MAX`[1]. To get a random number between 0 and `n-1`, you can take `rand() % (n)`.[2]

The `rand()` function is actually a *pseudorandom number generator*, meaning that it outputs a consistent sequence of values when given a particular *seed* value. By default, `rand()` has a seed value of 1, so unless you change this your program will generate the same sequence of random numbers each time it is run.

To change the seed value, include the line `srand(time(NULL))`[3] at the beginning of your `main()` function.

To randomize the order of directions through which you will search, declare the directions in some fixed order in an array. You can then shuffle that array in-place with the following algorithm[4]:

```
shuffle_array(A[n]):
    for i from 0 to n-1:
        choose a random number r between i and n-1, inclusive
        switch A[i] and A[r]
```

This procedure produces all possible orderings with equal probability.

[1] `RAND_MAX` is defined in *stdlib.h*. Its value is library-dependent, but is guaranteed to be at least 32767.

[2] This is slightly biased towards lower numbers, but it's good enough for this sort of application.

[3] `time()` returns the number of seconds that have occurred since January 1, 1970. You can find it in the `<time.h>` header file.

[4] This algorithm is known as the Fisher-Yates shuffle, described in 1938 by Ronald A. Fisher and Frank Yates.

### 4.4 Compiling and Running

You have been provided a *Makefile*, a text file that contains scripts for compiling, running, or cleaning up projects (for example).

Makefiles contain *build targets*; if we look at the Makefile included in this project we can see that there are 5 targets: `all`, `generator`, `solver`, `solver_full` and `clean`. Each target has an associated command and optional dependencies. In order to test the first half of the assignment, you will only need `generator` and `clean`.

The make command only builds files that have been modified since the last build and allows you to split up your build process (e.g. splitting up maze generator and maze solver binaries to be built).

To build a particular target, run in the directory containing the Makefile:

```
make <target>
```

from the command line. If no target is specified, it will build the first target (in this case, the target `all`). Right now, as `solver` and `solver_full` are incomplete, you should use `make generator` to build the generator portion of this assignment. You can run `make clean` to remove any previously built targets. (You can also run `make clean all` to remove all existing targets and build new ones at the same time.)

If you want to add extra files to either part of your program, add them to the file list defined by `GEN_OBJS`.

Once you have compiled your program, you can run it with the command:

```
./generator <output maze file> <number of rows> <number of columns>
```

You will learn more about Makefiles in the future lab. You can also reference the C Programming Tools document in the Guides and Documents section of the course website.

### 4.5 Support

In order to test your generator before moving onto the next section, we've provided the following three programs. `cs033_maze_viewer` is designed to only accept mazes with 10 rows and 25 columns. Please only use it with mazes of these dimensions or you will receive errors (even if your code is correct!).

- `cs033_maze_generator_demo <output maze file> <number of rows> <number of columns>`

  This program demonstrates the expected behavior of your `generator` program. It generates a maze in `output_file`.

- `cs033_maze_validator <maze file>`

  This program will check your maze for errors, such as inconsistent or missing walls and inaccessible areas. Please make sure to validate your generated maze before moving on.

- `cs033_maze_viewer <maze file>`

  This script will open your maze in a graphical interface. This script **only works with mazes of size 10x25**.

# 5 Solver

Now that you have a working generator, you will write a maze *solver*, which solves the mazes you generated. Your solver program should solve a maze defined by the given representation, outputting to a file either the path to the exit or the order of its search (see Solution Output, below).

Your solver program should take eight arguments:

```
./solver <input maze file> <number of rows> <number of columns>
         <output solution file> <starting row> <starting column>
         <ending row> <ending column>
```

## 5.1 Translating Between Representations

In your `generator` you translated from the program representation to the file representation. For your `solver` you must translate from the file representation to the program representation.

Before you can perform any operations on the hexadecimal character from a file, you must properly convert it back into an integer. Fortunately, you can combine both of these steps into one by using the `fscanf()` function (described below) with the format string `"%1x"`.

Once you have the hexadecimal number back in decimal form, use bit-level operations to extract the room data. To extract a particular element, use a *bit mask*, or an integer whose binary representation consists entirely of zeroes except for a particular bit (or bits). Some examples include 0x4 (0100) and 0x1 (0001).

To use bit masks, perform a bitwise AND on the mask and the integer to be masked.

For example, the C expression `x & 0x4` returns a copy of `x` with all bits set to 0 except the third (1100 & 0x4 = 0100). Similarly, `x & 0x1` would return a copy of `x` with all bits set to 0 except the first (1100 & 0x1 = 0000, the first bit happens to also be 0 in this case).

## 5.2 Solver Algorithm

Your program should employ a *depth-first search*. Such a search begins at the maze's start room and explores adjacent, accessible rooms recursively.

Beginning with the indicated room, this algorithm repeatedly chooses a path from each room and follows that path until it reaches a dead end, at which point it backtracks and tries a new path. This process continues until all paths have been explored or the destination is found. The following is pseudocode for this algorithm.

```
dfs(row, col):
    if (row, col) are the coordinates of the goal
        return true
    mark the room at [row][col] as visited
    for each direction dir:
        neighbor = rooms[row + row_offset of dir][col + col_offset of dir]
        if the connection in direction dir is open and neighbor is unvisited:
            if dfs(neighbor.row, neighbor.col) returns true
                return true

    /* if the program reaches this point then each neighbor's branch
       has been explored, and none have found the goal. */

    return false
```

### 5.3 Solution Output

Your program should print a list of rooms to the given output file. The room coordinates should be formatted in the following format when printed to the solution file: `<row>, <col>`.

Here's an example of what the first several rows of your solution file might look like.

```
PRUNED
0, 0
1, 0
1, 1
1, 2
2, 2
...
```

We expect your solver to produce two different modes of output:

- Your program outputs the coordinates of only the final route from beginning to end. Your program should first print the line "`PRUNED`". The program should then print the coordinates of each room on the solution path as described earlier.

  To do this, build a list of rooms as you search, and print out each room in the list when you reach the destination room. You can accomplish this using pointers! Here are two good (and similar) solutions:

  - add a `next` pointer to your room structures. Use these pointers to maintain a linked list of rooms - when you move from room $A$ to room $B$, set room $A$'s pointer to room $B$.
  - define a `struct linked_list` and pass around a pointer to the last node of your list in your recursive `solve` function. When you visit a room, initialize a new list node and append it to the end of the list, using the pointer to the previous node.

  You should feel free to modify the stencil to support either solution.

- Your program outputs the entire path traversed up until the goal is reached. Your program should first print the line "`FULL`". The program should then print each room's coordinates

when first visiting that room, and after each recursive call that returns false. This will print the path from start to finish, including "backtracking" after dead ends.

`Note`: Depending on how your solver algorithm searches the maze, there can be multiple valid FULL solutions.

The choice should be made when your program is compiled. This is done using preprocessor macros. Macros are defined using the `gcc` compiler flag `-D<macro>`, which defines `<macro>` for the preprocessor. For example, to add the macro `PIZZA` to your program, add the flag `-DPIZZA`. In your Makefile, you'll see the flag `-DFULL` in the command for the `solver_full` target, which defines the macro `FULL` for that target.

To write code that will execute only when a specific macro is defined, refer to the example below:

```
#ifdef FULL
printf(<something>);
#else
printf(<something else>);
#endif
```

The above code fragment executes the `printf(<something>)` statement only if the macro `FULL` is defined, and executes the `printf(<something else>)` statement otherwise. You can also use the macro `#ifndef <macro>` to execute code only if `<macro>` is not defined.

Your program should print the entire search if a macro `FULL` is defined and print only the path to the exit otherwise. Rooms should be printed with format `<row>, <col>` on its own line with no parentheses with the upper-left corner of the maze corresponding to coordinate `(0, 0)`. If the start and end rooms happen to be the same, your output should contain the room only once.

### 5.4 Compiling and Running

In order to test the solver part of the assignment, you will need to build the following targets.

- `solver`, which builds your solver program with no macros defined (i.e. it should print pruned output).
- `solver_full`, which builds your solver with the `FULL` macro defined (i.e. it should print its full exploration path).

You can build each target separately using `make <target>`, or run `make all` (or just `make`) to build both (along with `generator`).

Once you have compiled your solver, you can run it with the command

```
./solver <input maze file> <number of rows> <number of columns>
        <output solution file> <starting row> <starting column>
        <ending row> <ending column>
```

or

```
./solver_full <input maze file> <number of rows> <number of columns>
        <output solution file> <starting row> <starting column>
        <ending row> <ending column>
```

### 5.5 Support

s In order to test this part of the assignment on your generated mazes, you are provided with the following programs. `cs033_maze_solution_viewer` is designed to **only accept mazes with 10 rows and 25 columns**. Please only use it with mazes of these dimensions.

- `cs033_maze_solver_demo <input maze file> <number of rows> <number of columns>`
  `<output solution file> <starting row> <starting column>`
  `<ending row> <ending column>`

  This is a demo program for the solver project; it produces pruned output, i.e. just the path from the start coordinate to the end coordinate.

- `cs033_maze_solver_full_demo <input maze file> <number of rows> <number of columns>`
  `<output solution file> <starting row> <starting column>`
  `<ending row> <ending column>`

  This is a demo program for a solver's full output. It produces a list of rooms in the order in which they were explored.

- `cs033_maze_validator <maze file> <solution file>`
  `<starting row> <starting column>`
  `<ending row> <ending column>`

  This program will also validate a solution file (if you provide it one) for a maze. Make sure the end-point coordinates correspond to those given to your solver.

- `cs033_maze_solution_viewer <maze file> <solution file> <speed>`

  This program will visualize your solution path in the provided maze. The `<speed>` argument to `cs033_maze_solution_viewer` must be an integer between 1 and 3, inclusive; higher values make Dory move faster. If either the maze or the solution file is improperly formatted, the visualizer will exit.

  This program will also check your input files for correctness, and will print out a list of errors if problems are found.

## 6 Input and Output

In this assignment, you will need to work with files to represent and solve your mazes. The C `<stdio.h>` library contains several definitions that enable you to easily write to or read from files. Included in these definitions is a `FILE` struct, which represents a file within a C program.

### 6.1 Opening a File

To use files (reading, writing, etc), you need to first open it. The `fopen()` function opens a file, returning a pointer to a `FILE` struct that corresponds to the desired file.

```
FILE *fopen(char *filename, char *mode)
```

The desired file is indicated by `filename`. The `mode` argument refers to how the file will be used; if you intend to write to the file, this value should be `"w"`, and if you intend to read from the file, it should be `"r"`.

If the desired file does not exist it will be created. If an error occurs, `NULL` is returned.

### 6.2 Writing to a File

In fact, you can write data to any file (not just standard error) using the `fprintf()` function. This function works in very much the same way as `printf()`.

```
int fprintf(FILE *stream, char *format, ...)
```

The only difference is that `fprintf()` takes an additional argument: the `FILE *` that you obtained with `fopen()`.

**Remember**: To print a hexadecimal number, pass that number as an argument to `fprintf()`, using `"%x"` as your format string.

### 6.3 Reading From a File

For this assignment, you will only need to read from a file in your solver program. `<stdio.h>` defines many functions that you can use to read from files. The function that you will likely find most useful for this part of the project is `fscanf()`.

```
int fscanf(FILE *stream, char *format, ...)
```

`fscanf()` is to `scanf()` as `fprintf()` is to `printf()`: it does the same thing as its counterpart function, but with a `FILE *` as the source of the input or output operation. To receive a hexadecimal number from a hexadecimal character, use the format string `"%1x"`, which will scan a single character into an `unsigned int` variable.

### 6.4 Closing a File

After your program has finished writing to or reading from a file, it should close that file. Do this with the function `fclose()`.

```
int fclose(FILE *fp)
```

This function returns 0 if no error occurred and EOF (a negative value) otherwise.

# 7 Error Checking

When using a library call, it is important to check the value the function returns. If the function returns a value associated with an error, it is important to report the error and handle it accordingly. We expect you to **check for errors throughout your code**, and upon finding any, to **print the appropriate error message** to `stderr` using `fprintf`.

In this program, we should stop program execution, report the error to the user and exit with the program returning an error (`return 1`). Check if an error has occured by checking the return value. If so, write out an appropriate error message to standard error (`stderr`), and exit cleanly. Here's an example:

```
fprintf(stderr, "[Error message goes here.]\n");
```

Most functions will return a certain value to denote an error. Find out what those values are with `man <function>`.
With regard to your `main()` function, it should return 0 if it completed execution normally, or 1 if it exited on encountering an error.

# 8 Grading

Your grade for this project will be calculated as follows:

| | |
|---|---|
| Generator | 35% |
| Solver | 35% |
| Error Handling | 10% |
| Style | 20% |
| Total | 100% |

Both generator and solver will be graded based on Code Correctness (15 pts) and Functionality (20 pts).

- **Code Correctness**: no part of your code relies on undefined behavior, uninitialized values, or out-of-scope memory; your program compiles without errors or warnings.

  *Only* `cs033_maze_validator` *will be used during grading.* (i.e It's ok if your maze doesn't work in the visualizer.)

- **Functionality**: your code produces correct output, and does not crash for any reason. It does not terminate due to a segmentation fault or floating point exception.

- **Error Handling**: your code performs error checking on its function inputs and outputs, and exits gracefully in all situations.

- **Style**: your code should look nice! Use appropriate whitespace and indentation and well-named variables and functions. Your code should be reasonably factored, and functions should not be too long.

Your programs should perform error checking on their input, with one exception: if your solver program successfully opens a maze file, you may assume that the file contents form a correctly-formatted maze. Your program should not crash for any reason; before you hand in your project, make sure that your program does not terminate due to a segmentation fault or floating point exception.

Consult the C Style document (which is on the website) for some pointers on C coding style.

# 9 Handing In

To hand in your project, run the command

```
cs033_handin maze
```

from your project working directory. You should hand in `common.c` `generator.c` and `solver.c` (along with any support code you may have written), a Makefile, and a README. Your README should describe the organization of your programs and any unresolved bugs.

If you wish to change your handin, you can do so by re-running the script. Only your most recent handin will be graded.