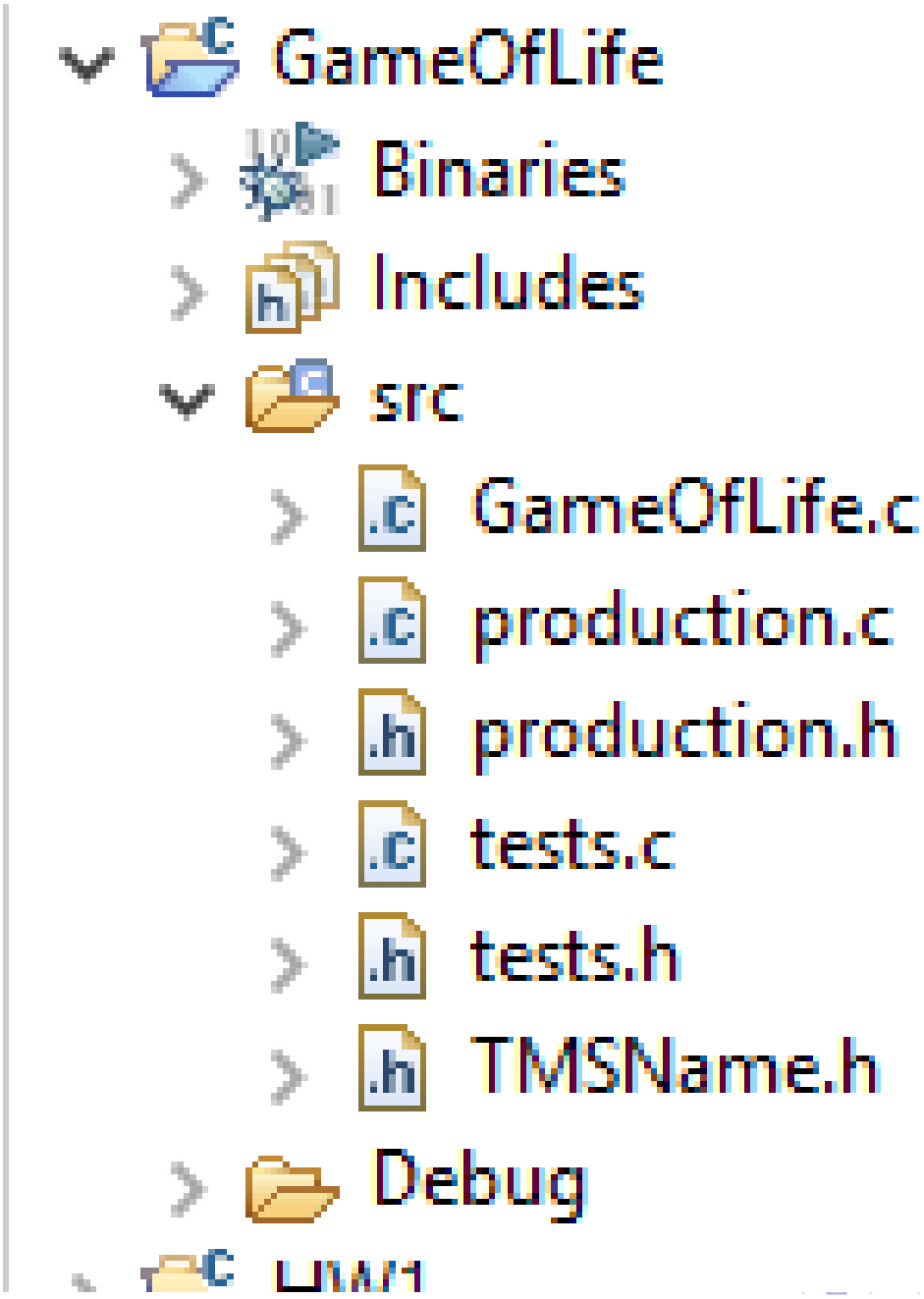
Outline

- Getting Started:
- template
- top down
- sequence diagram
- messages to methods

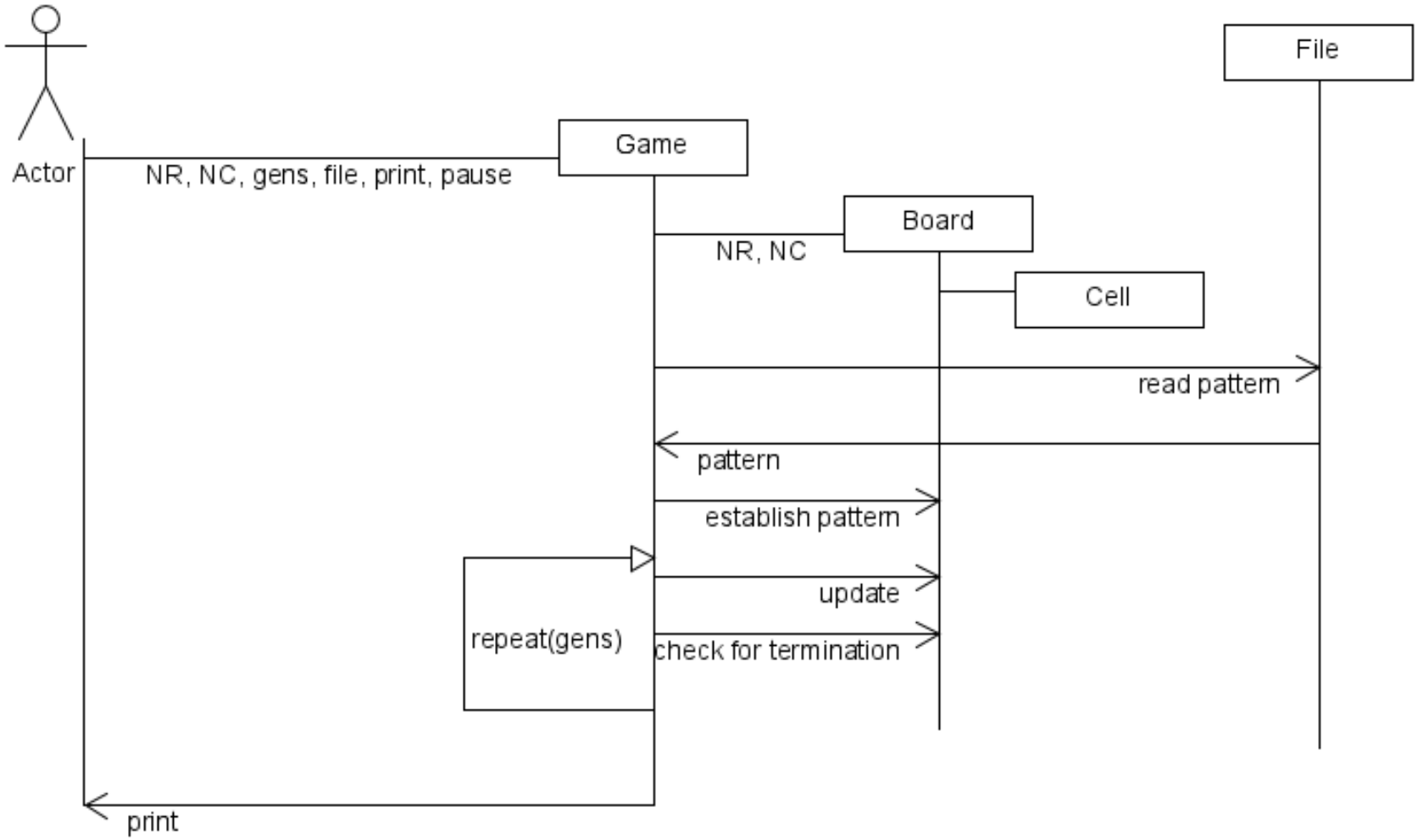
Game of Life is a suitably difficult problem; we'll use it for an example. I

- From the problem statement, we want to extract nouns,
- and consider whether they are relevant to our solution.
- Cells, rows, columns: These are useful for the gameboard.
- There is the idea of a generation, and there is a maximum generation.
- We might want to keep track of previous, current and future game boards.
- There is the idea of empty board: no cell is "alive".
- There is the idea of a steady state: where current board = future board, or
- current board = previous board.
- There is the idea of oscillation: where future board = previous board.

We get started: We know the initial files.



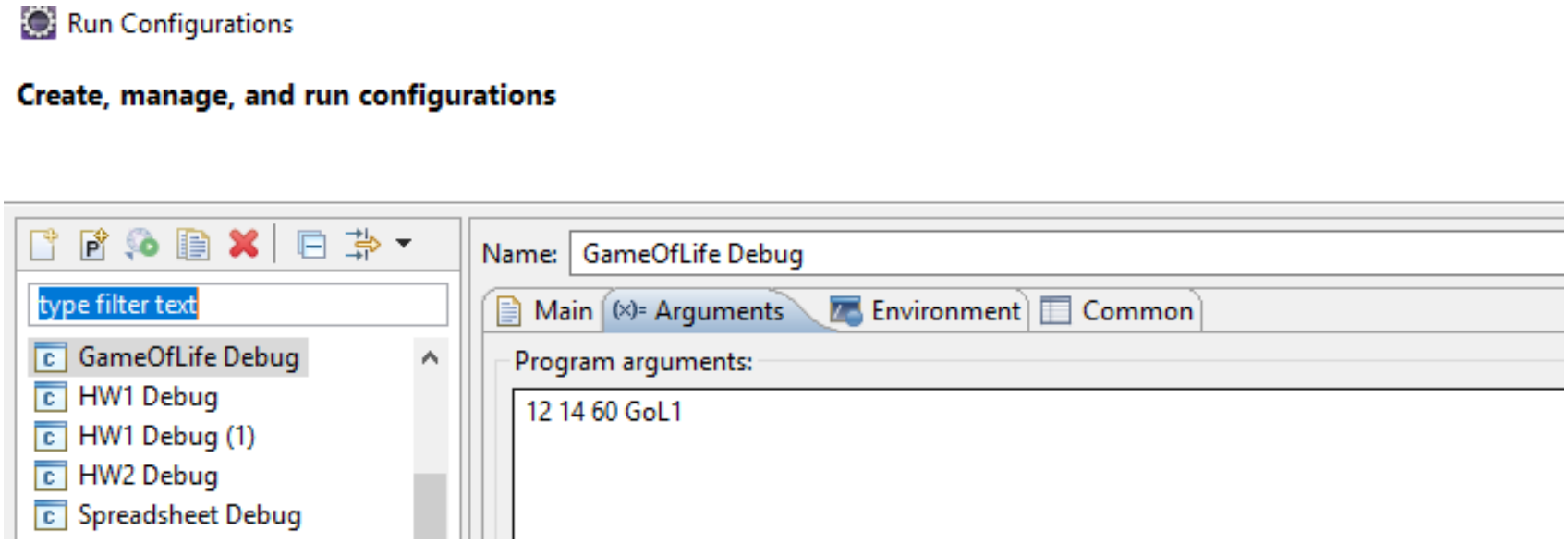
We get started with a sequence diagram.



Each message implies a function invocation.

- We do not have to start implementing functions from the top of the sequence diagram.
- Let's handle "command line" arguments.
- In Eclipse, we provide these run time arguments in the run configuration.

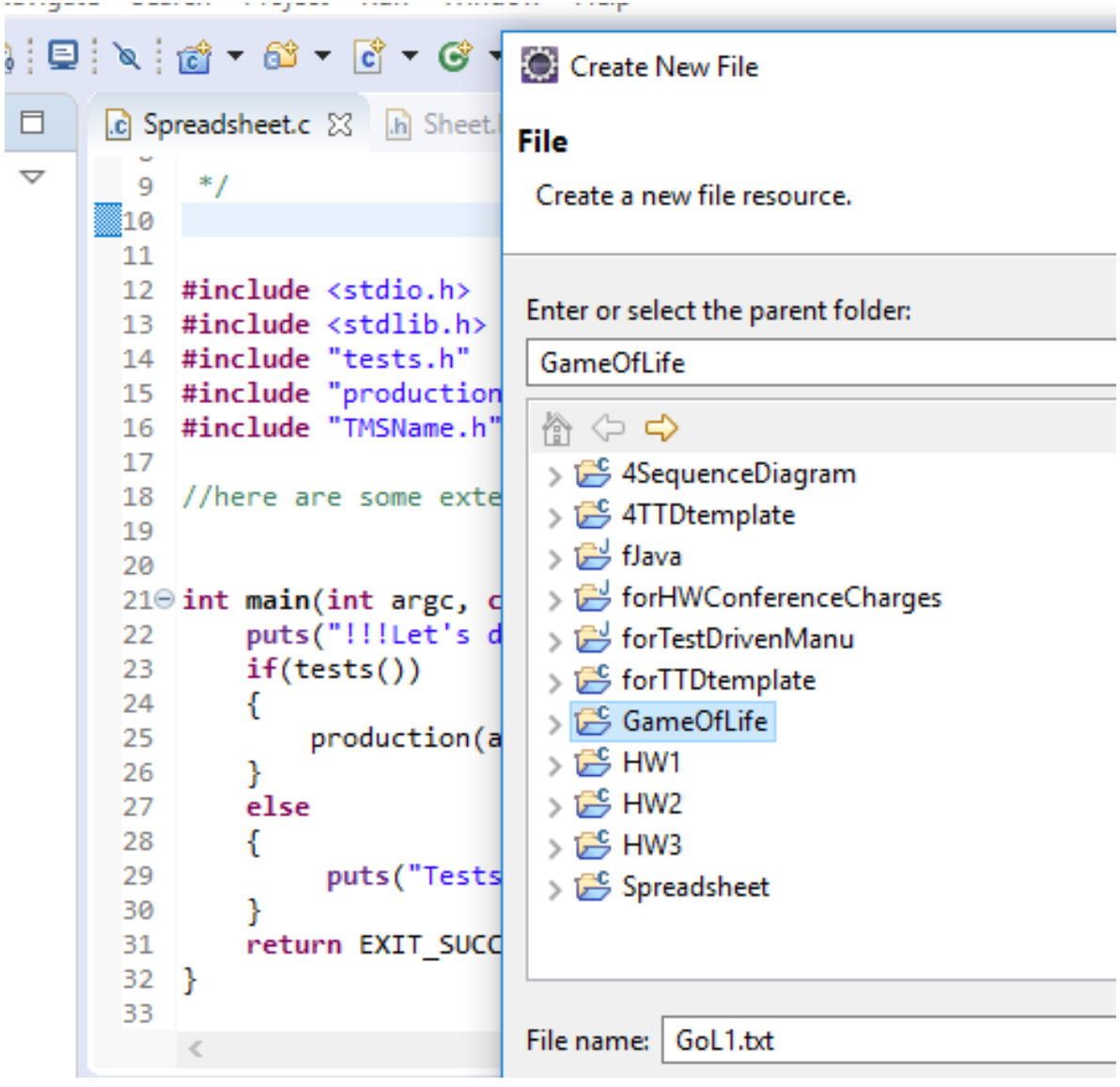
Command line argument values are entered into the run configuration.



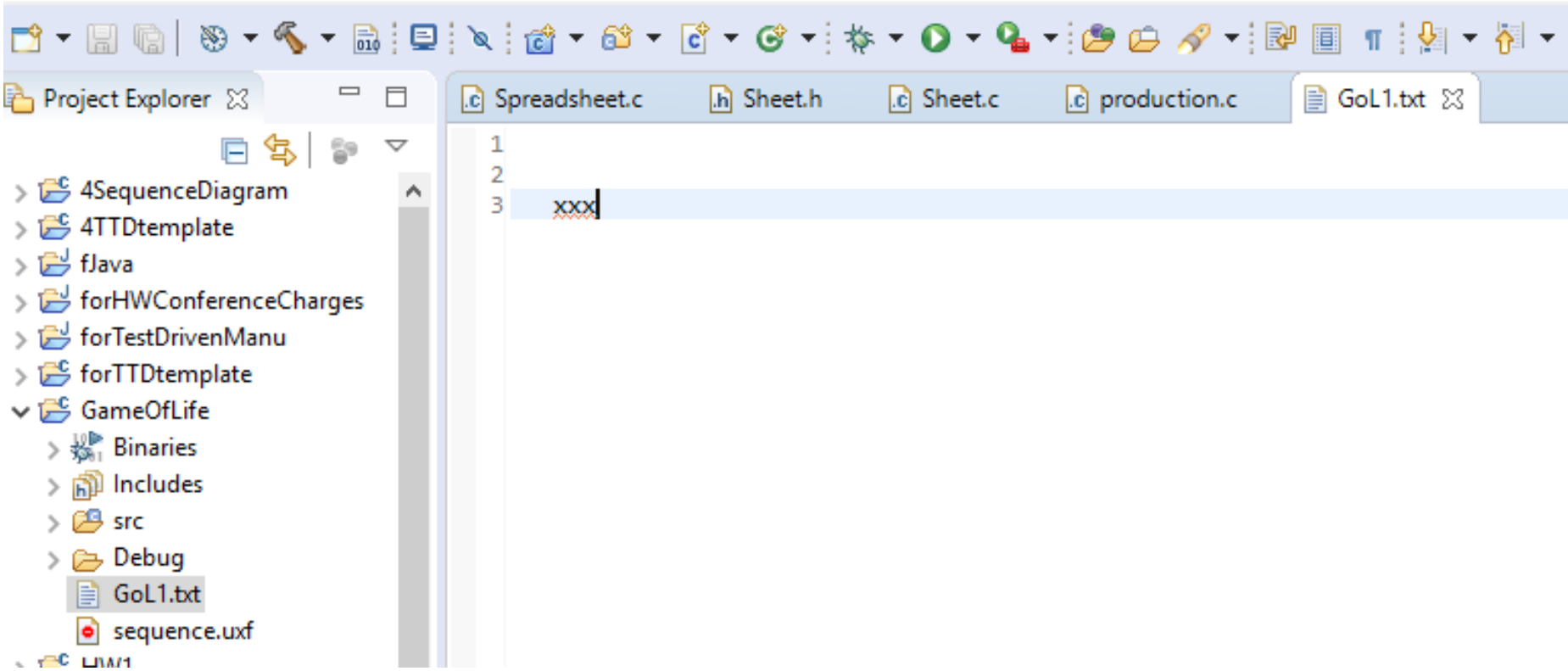
We need a file. I

- There are many ways to make a file.
- Eclipse editor is easy.

We need a file. II



We need a file. III



It's time to think about the order of implementation.

- There are many possible sequences in which we could implement the code.
- By the end, it all functions must be implemented.
- We are at liberty to choose which function to do first.
- It will be convenient to display the content of the boards, as we develop the code.
- Let's implement that function first.

It's time to think about where the function will be.

- The user expects to be able to see the board.
- Therefore, the production.c file has a method to display.
- The board has the data.
- Therefore, the board.c file has a method to display.
- The production.c file method invokes the board.c method.
- Each board will have its own two dimensional array.
- The production.c method will decide which board is being displayed.

We can add a showBoards function. I

```
/*
* tests.h
*
*  Created on: Jul 4, 2019
*      Author: Therese
*/

#ifndef TESTS_H_
#define TESTS_H_


#include "production.h"
bool tests();

bool testDisplayBoard();
```

We can add a showBoards function. II

```
#endif /* TESTS_H_ */

/*
* tests.c
*
*  Created on: Jul 4, 2019
*      Author: Therese
*/

#include "tests.h"
#include "production.h"

bool tests()
{
    bool answer = false;
    bool ok1 =  testDisplayBoard();
    answer = ok1;
```

We can add a showBoards function. III

```
    return answer;
}

bool testDisplayBoard()
{
    bool ok = false;
    //we need to have a board before we can display it.
    //the boards can be represented in arrays, one dimensio
    //we could even have a dimension for the past, current
    //this is a test case
    int nRows = 6;
    int nCols = 8;

    int theBoards[nBOARDS][nRows][nCols];
    //we set one board to a known pattern
    //we'll print out the pattern to the console
    //checking by eye.
```

We can add a showBoards function. IV

```
//set the known pattern:
for(int board = 0; board<nBOARDS; board++)
{
    for(int row=0; row<nRows; row++)
    {
        for(int col=0; col<nCols; col++)
        {
            theBoards[board][row][col] = (row==col); //c
        }
    }
}
int* b0 = &(theBoards[0][0][0]);
int* b1 = &(theBoards[1][0][0]);
int* b2 = &(theBoards[2][0][0]);
//here's the test
puts("Here's board 0");
displayBoard(b0, nRows, nCols);
```

We can add a showBoards function. V

```
        puts("Here's board 1");
        displayBoard(b1, nRows, nCols);
        puts("Here's board 2");
        displayBoard(b2, nRows, nCols);
            printf("Did they look like diagonal matrices?(y or
        fflush(stdout);
        char a = getchar();
        if (a == 'y')
        {
            ok = true;
        }
    return ok;
}
```

This is found in GoL1.zip.

This much runs, but we want to extend it to actually print boards. I

- The version on Canvas, GoL1.zip runs, but does not actually print boards.
- Now that we have a test for print boards,
- we can implement it.
- We're starting with:

```
void displayBoard(int* board, int nRows, int nCols)
{


}
```

- We're adding to get:

This much runs, but we want to extend it to actually print boards. II

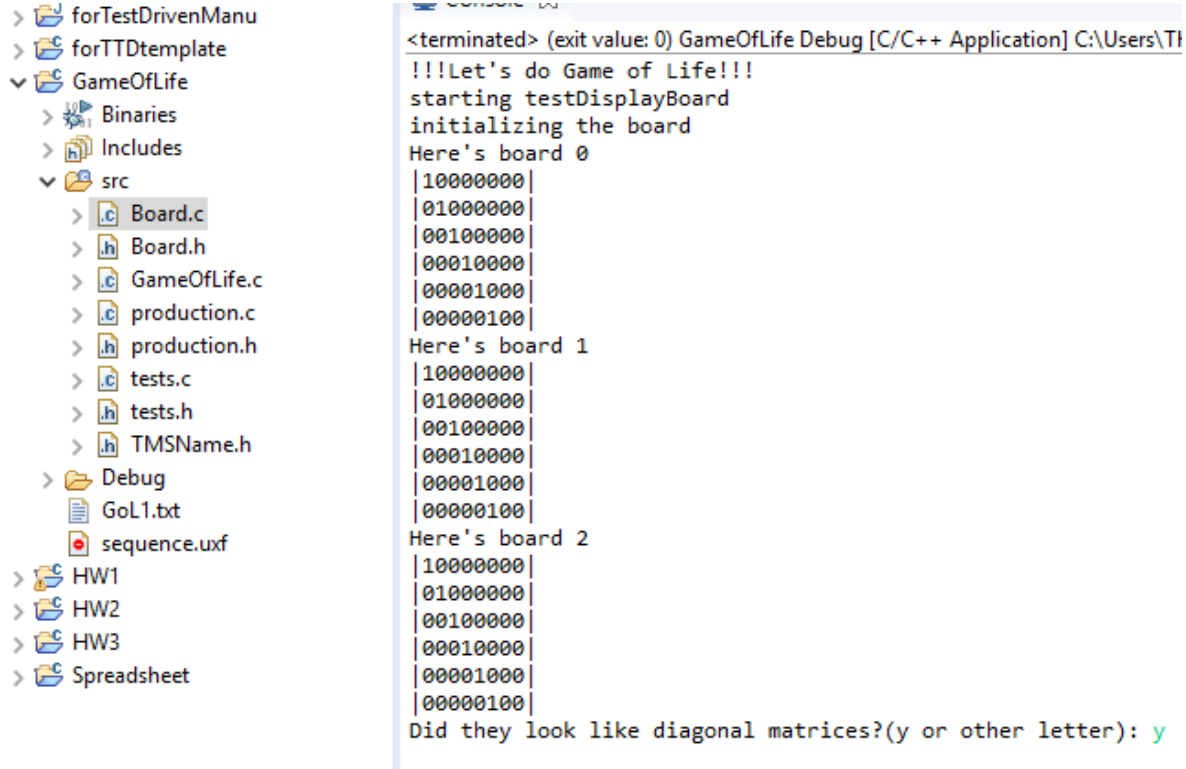
```
#include "Board.h"
void displayBoard(int* board, int nRows, int nCols)
{

    int* cellP = (int*) 0; //initialize the pointer
    int cellContent = 0;
    for (int row = 0; row<nRows; row++)
    {
        printf("|"); //start the row with a vertical ba
        for(int col = 0; col<nCols; col++)
        {
            cellP = board+(row*nCols)+col;
            cellContent = *cellP;
            //content is ‘‘what’s at’’ where the pointe
            printf("%d",cellContent);
```

This much runs, but we want to extend it to actually print boards. III

```
            }
            printf("|\n"); //end the row with a vertical b
            //and go to a new line
        }
    }
```

This much runs, but we want to extend it to actually print boards. IV



This is found in GoL2.zip.

What's next?

- Now that we have some confidence we can see what' in the boards,
- We might choose to see whether we can successfully read a file content into a board.
- We will again use the person to check the console,
- so we can reuse that part.
- We'll also rely on the person to check whether the console output matches the file input.

We begin with a new test.

```
bool tests()
{
    bool answer = false;
    bool ok1 =  testDisplayBoard();
    bool ok2 = testFileInput();
    answer = ok1 && ok2;
    return answer;
}
```

We add the new test to our sequence of tests.

We create the function prototype for the new test.

```
bool testFileInput();
```

We put it in tests.h.

We create the function's stub.

```
bool testFileInput()
{
    bool ok = false;
    return ok;
}
```

We put it in tests.c.

We fill in the function with the first test case. I

```
bool testFileInput()
{
    puts("starting testFileInput");
    fflush(stdout);
    bool ok = false;
    //test case
    //we know our first file is named GoL1.txt.
    //we know our purpose for opening is read

    //we will prepare our target array as in testDisplayBoa
    int nRows = 6;//these are big enough for GoL1.txt's dat
    int nCols = 8;
    char theBoards[nBOARDS][nRows][nCols];
    for(int sheet = 0; sheet<nBOARDS; sheet++)
    {
        for(int row= 0; row<nRows; row++)
```

We fill in the function with the first test case. II

```
        {
            for(int col = 0; col<nCols; col++)
            {
                theBoards[sheet][row][col]= ' ';//initializ
            }
        }
    }


    FILE* fp = fopen("GoL1.txt", "r");
    puts("Attempting to open file");
    fflush(stdout);
    if(fp==NULL)
    {
        puts("Could not find that file.");
    }
    else
```

We fill in the function with the first test case. III

```
{
    puts("found the file.");
    fflush(stdout);
    //now we want to read the lines of the file
    //and set the values into the array.
    //we will want to center the pattern
    //so we will want to know how long the longest line
    //how many lines there are in the file
    int howManyLines = 0;
    int maxCharsFound = 0;
    int charsFound = 0;
    char ch;
    int row = 0;
    int col = 0;
    while((ch = fgetc(fp)) != EOF)
    {
        charsFound = 0;//this is on a per line basis
```

We fill in the function with the first test case. IV

```
        while((ch != '\n') && (ch != '\r') && (ch != E
        {
            theBoards[0][row][col]= ch; //put the chara
            charsFound++; //update the number of chars
            ch = fgetc(fp); //get a new character
            col++;
        }
        //we have reached a new line
        row++;
        col=0;
        howManyLines++;
        maxCharsFound = (charsFound> maxCharsFound)? ch
    }
    fclose(fp);
    puts("closing the file");
    printf("The file has %d lines.\n", howManyLines);
```

We fill in the function with the first test case. V

```
printf("The longest line has %d characters.\n", max
//now we know how many lines, and
//the length of the longest line
int offsetV = (int)floor((nRows-howManyLines)/2);
int offsetH = (int)floor((nCols-maxCharsFound)/2);
printf("The vertical offset is %d.\n", offsetV);
printf("The horizontal offset is %d.\n", offsetH);
fflush(stdout);
if((offsetV <0) || (offsetH <0))
{
    puts("The file is too big for the array.");
    fflush(stdout);
}
else
{
    int wrow;
    int wcol;
```

We fill in the function with the first test case. VI

```
        for(int rrow = 0; rrow< howManyLines; rrow++)
        {
            for(int rcol = 0; rcol<maxCharsFound; rcol+
            {
                wrow = rrow+offsetV;
                wcol = rcol+offsetH;
                theBoards[1][wrow][wcol] = theBoards[0]

            }
        }
    }
    //display the results
    char* b0 = &(theBoards[0][0][0]);
    char* b1 = &(theBoards[1][0][0]);
    //here's the test
    puts("Here's board from file");
    displayCharBoard(b0, nRows, nCols);
```

We fill in the function with the first test case. VII

```
            puts("Here's board as centered");
            displayCharBoard(b1, nRows, nCols);
            fflush(stdin);
            printf("Did the board look like the file?(y or othe
            fflush(stdout);
            char a = getchar();
            if (a == 'y')
            {
                ok = true;
            }
        }
        return ok;

}
```

This code is in GoL3.zip.

The output looks like this:

```
The file has 3 lines.
The longest line has 6 characters.
The vertical offset is 1.
The horizontal offset is 1.
Here's board from file
|oooooo  |
|oooooo  |
|oooxxx  |
|        |
|        |
|        |
Here's board as centered
|        |
| oooooo |
| oooooo |
| oooxxx |
|        |
|        |
Did the board look like the file?(y or other letter): y
```