Contents

1 Basic Test Results

```
Running...
Opening tar file
Makefile
PitPlugger.c
RadiusReader.c
Draw2DCharArray.c
OK
Tar extracted O.K.
Checking files...
OK
Making sure files are not empty...
OK
Importing files
OK
Compilation check...
Compiling...
gcc -Wall -c Draw2DCharArray.c
gcc -Wall -c DrawArrayDriver.c
gcc -Wall Draw2DCharArray.c DrawArrayDriver.c -o DrawArrayDriver
OK
Compiling...
gcc -Wall Draw2DCharArray.c PitPlugger.c RadiusReader.c -o PitPlugger
OK
Compilation went without errors, BUT you must check to see if you got warnings!!!
Check some inputs:
Running test...
OK
OK

========================
= Checking coding style =
========================
 ** Total Violated Rules      : 0
 ** Total Errors Occurs       : 0
 ** Total Violated Files Count: 0
```

2 Draw2DCharArray.c

```c
/*
 * Draw2DCharArray.c
 *
 *  Created on: Jul 29, 2014
 *      Author: roeia1
 */
#include <stdio.h>
#include "Draw2DCharArray.h"

/**
 * Initialize the given char array to be filled with space (' ') characters.
 */
void initializeArray(char arr[ROWS][COLS])
{
    int currRow;
    for (currRow = 0; currRow < ROWS; currRow++)
    {
        int currCol;
        for (currCol = 0; currCol < COLS; currCol++)
        {
            arr[currRow][currCol] = ' ';
        }
    }
}

/**
 * Draw the char array on the screen.
 * After each row go down a line ('\n').
 * Stop drawing when a nul character ('\0') is met.
 */
void drawArray(char arr[ROWS][COLS])
{
    char flag = 0;
    int currRow;
    for (currRow = 0; currRow < ROWS; currRow++)
    {
        int currCol;
        for (currCol = 0; currCol < COLS; currCol++)
        {
            if (arr[currRow][currCol] == '\0')
            {
                flag = 1;
                break;
            }
            printf("%c", arr[currRow][currCol]);
        }
        if (flag == 1)
        {
            break;
        }
        printf("\n");
    }
}
```

3 Makefile

```makefile
TARGETS = DrawArrayDriver PitPlugger
C = gcc
FLAGS = -Wall
LOCATION = /cs/course/2014/slabc/public/ex2/inputOutput/
PITFILE = pit_radius_1.in
STONEFILE = stone_radius_1.in
EXPECTED = map_pit_1_stone_1.out
RESULT = my_map_pit_stone.out

DrawArrayDriver: DrawArrayDriver.o Draw2DCharArray.o
	$(C) $(FLAGS) Draw2DCharArray.c DrawArrayDriver.c -o DrawArrayDriver

PitPlugger: Draw2DCharArray.c PitPlugger.c RadiusReader.c
	$(C) $(FLAGS) Draw2DCharArray.c PitPlugger.c RadiusReader.c -o PitPlugger

all: $(TARGETS)

tar:
	tar cvf ex2.tar Makefile PitPlugger.c RadiusReader.c Draw2DCharArray.c

PitPlugger.o: PitPlugger.c RadiusReader.h Draw2DCharArray.h
	$(C) $(FLAGS) -c PitPlugger.c

Draw2DCharArray.o: Draw2DCharArray.c Draw2DCharArray.h
	$(C) $(FLAGS) -c Draw2DCharArray.c

DrawArrayDriver.o: Draw2DCharArray.o DrawArrayDriver.c
	$(C) $(FLAGS) -c DrawArrayDriver.c

test1: PitPlugger
	rm $(RESULT) -f
	PitPlugger $(LOCATION)$(PITFILE) $(LOCATION)$(STONEFILE) > $(RESULT)
	diff $(RESULT) $(LOCATION)$(EXPECTED)

clean:
	rm -f *.o

.PHONY: all tar test1 clean
```

4 PitPlugger.c

```c
/*
 * PitPlugger.c
 *
 *  Created on: Jul 30, 2014
 *      Author: roeia1
 */

#include <stdio.h>
#include <stdlib.h>
#include "Draw2DCharArray.h"
#include "RadiusReader.h"

#define TRUE 1
#define FALSE 0
#define NUM_OF_FILES 2
#define NUM_OF_ARGUMETNS NUM_OF_FILES + 1
#define PIT_LOCATION 0
#define STONE_LOCATION 1
#define NUMBER_OF_DOTS 3
#define ROOF_STONES 1
#define FIRST_LEVEL 0
#define MIN(a, b) (((a) < (b)) ? (a) : (b))
#define EMPTY_SIGN ' '
#define STONE_SIGN '-'
#define GROUND_SIGN '*'
#define FLOOR_SIGN '+'

/**
 * Printing a message indicates the status of the pit.
 *
 * This method getting an array representing the stone in each level, and array containing the
 * depth of the pit and the number of stones.
 * It is calculating the sum of the follows:
 * -Levels left open
 * -Stones thrown to pit
 * Printing the information in a detailed message.
 *
 * @param stonesInPit - An array representing stone in each level.
 * @param numOfLines - An array containing the number of lines of each file.
 */
void printResultMessage(unsigned int const* stonesInPit, unsigned int const* pit,
                        unsigned int const* numOfLines)
{
    char isBlocked = FALSE;
    int openPits = 0;
    int stonesUsed = 0;
    int currPit;
    // Checking if the first level of the pit is 0
    if (pit[0] == 0)
    {
        isBlocked = TRUE;
    }
    else
    {
        // Calculating how many stones was used
        for (currPit = 0; currPit < numOfLines[PIT_LOCATION] + ROOF_STONES; currPit++)
        {
            if (stonesInPit[currPit] != 0)
            {
```

```c
                stonesUsed++;
            }
        }
        // Checking if the pit is blocked
        currPit = 0;
        while ((currPit < ROOF_STONES + 1) && (isBlocked == FALSE))
        {
            if (stonesInPit[currPit] != 0)
            {
                isBlocked = TRUE;
            }
            currPit++;
        }
    }
    if (isBlocked == TRUE)
    {
        printf("Hurrah!! You have successfully plugged that pit ;)\n");
    }
    else
    {
        printf("Oy Vey!! The pit is still open, what will we do now? :(\n");
        currPit = ROOF_STONES;
        while ((currPit < numOfLines[PIT_LOCATION] + ROOF_STONES) && (stonesInPit[currPit] == 0) &&
               (pit[currPit-ROOF_STONES] != 0))
        {
            openPits++;
            currPit++;
        }
    }
    printf("This pit is %d levels deep, of which %d levels remain open.\n", \
            numOfLines[PIT_LOCATION], openPits);
    printf("We had %d stones and threw %d of them into the pit.\n\n", numOfLines[STONE_LOCATION],
            stonesUsed);
}

/**
 * Dividing a number by 2 and round up.
 *
 * @param dividend - The number to divide by 2.
 * @return The result of the division.
 */
int divTwoRoundUp(int const dividend)
{
    return (dividend + 1) / 2;
}

/**
 * Inserting a given char number of times to the matrix in a row.
 *
 * @param pitMatrixLine - The line in the matrix to insert the char.
 * @param numOfChars - The number of times to insert the char.
 * @param charToPlace - The char to insert.
 * @param currCell - Pointer to the number representing the cell in the line from there the char
 *                      will be inserted.
 */
void insertChars(char* pitMatrixLine, int const numOfChars, char const charToPlace,
                 int* const currCell)
{
    int currCharNum;
    for (currCharNum = 0; currCharNum < numOfChars; currCharNum++)
    {
        pitMatrixLine[*currCell + currCharNum] = charToPlace;
    }
    *currCell += numOfChars;
}

/**
 * Calculating the pit status after throwing the stones to block it.
```

```
 *
 * For each stone, while the pit isn't blocked, the method checks if the stone radius isn't bigger
 * then the pit level radius till the last level that is still open.
 *
 * Complexity:
 * M - the number of stones.
 * N - the number of levels in the pit.
 * The first loop will run O(N). (will run on all the levels)
 * If M = N or M < N then the second loop will run O(M*N):
 * The internal loop will run O(N) and the outer loop will run O(M).
 * If M > N then the second loop will run O(N*N):
 * The internal loop will run O(N) and the outer loop will run O(N) too because the maximum stones
 * that will take to plug an N size pit would be N stones.
 * The memory usage will be O(MAX_DEPTH) because this is the maximum size in this program that
 * being stored, an array in this size.
 *
 * @param pit - An array representing the radius of each level in the pit.
 * @param stones - An array representing the radius of each stone.
 * @param stonesInPit - The array being created (the pit status), the stone radius in each level
 *                        in the pit.
 * @param numOfLines - An array containing the number of lines of each file.
 */
void stonesToPit(unsigned int const* pit, unsigned int const* stones, unsigned int* stonesInPit,
                 unsigned int const numOfLines[NUM_OF_FILES])
{
    char isBlocked = FALSE;
    int currPit = 0;
    int currStone = 0;
    int currBlock;
    // Searching the first block of the pit
    while (currPit < numOfLines[PIT_LOCATION] && pit[currPit] != 0)
    {
        currPit++;
    }
    currBlock = currPit;
    if (currBlock == FIRST_LEVEL)
    {
        isBlocked = TRUE;
    }
    while ((currStone < numOfLines[STONE_LOCATION]) && (isBlocked == FALSE))
    {
        currPit = 0;
        while ((currPit < currBlock) && (stones[currStone] <= pit[currPit]))
        {
            currPit++;
        }
        stonesInPit[currPit] = stones[currStone];
        if (currPit < ROOF_STONES + 1)
        {
            isBlocked = TRUE;
        }
        currBlock = currPit - 1;
        currStone++;
    }
}

/**
 * Initialize an array with 0.
 *
 * @param array - The array being initialized.
 * @param arraySize - The size of the array.
 */
void initArray(unsigned int* array, unsigned int const arraySize)
{
    int currCell;
    for (currCell = 0; currCell < arraySize; currCell++)
    {
        array[currCell] = 0;
```

```
    }
}

/**
 * Creating a matrix of chars representing the pit status.
 *
 * @param pitMatrix - The matrix being created representing the result, the pit status.
 * @param pit - An array representing the radius of each level in the pit.
 * @param stonesInPit - An array representing the result (the pit status), the stone radius in each
 *                          level in the pit.
 * @param numOfLines - An array containing the number of lines of each file.
 */
void createPitMatrix(char pitMatrix[ROWS][COLS], unsigned int const* pit,
                     unsigned int const* stonesInPit, unsigned int const numOfLines[NUM_OF_FILES])
{
    int currLine = 0;
    int pitsToPrint = MIN(ROWS, numOfLines[PIT_LOCATION]);
    int currCell;
    int stoneSize;
    int currStone;
    for (currStone = 0; currStone < ROOF_STONES; currStone++)
    {
        stoneSize = stonesInPit[currStone];
        if (stoneSize != 0)
        {
            if (stoneSize > COLS)
            {
                stoneSize = COLS;
            }
            currCell = 0;
            insertChars(pitMatrix[currLine], (COLS-stoneSize) / 2, EMPTY_SIGN, &currCell);
            insertChars(pitMatrix[currLine], stoneSize, STONE_SIGN, &currCell);
            insertChars(pitMatrix[currLine], divTwoRoundUp(COLS - stoneSize), EMPTY_SIGN,
                        &currCell);
            currLine++;
        }
    }
    char pitBiggerThenCols;
    int currPit;
    for (currPit = 0; currPit < pitsToPrint; currLine++, currPit++, currStone++)
    {
        int pitSize = pit[currPit];
        stoneSize = stonesInPit[currStone];
        pitBiggerThenCols = 0;
        currCell = 0;
        if (pitSize > COLS)
        {
            pitSize = COLS;
            pitBiggerThenCols = 1;
        }
        if (stoneSize > COLS)
        {
            stoneSize = COLS;
        }
        insertChars(pitMatrix[currLine], (COLS - pitSize) / 2, GROUND_SIGN, &currCell);
        insertChars(pitMatrix[currLine], (COLS - stoneSize) / 2 - currCell, EMPTY_SIGN, &currCell);
        insertChars(pitMatrix[currLine], stoneSize, STONE_SIGN, &currCell);
        insertChars(pitMatrix[currLine],
                    divTwoRoundUp(COLS - stoneSize) - divTwoRoundUp(COLS - pitSize), EMPTY_SIGN,
                    &currCell);
        insertChars(pitMatrix[currLine], divTwoRoundUp(COLS - pitSize), GROUND_SIGN, &currCell);
        if (pitBiggerThenCols == 1)
        {
            currCell = 0;
            insertChars(pitMatrix[currLine], NUMBER_OF_DOTS, '.', &currCell);
            currCell = COLS - NUMBER_OF_DOTS;
            insertChars(pitMatrix[currLine], NUMBER_OF_DOTS, '.', &currCell);
        }
```

```c
    }
    if (pitsToPrint < ROWS)
    {
        currCell = 0;
        insertChars(pitMatrix[currLine], COLS, FLOOR_SIGN, &currCell);
    }
}

/**
 * Getting the data from the files.
 *
 * @param argc - The number of arguments received in the main.
 * @param argv - The array of the arguments received in the main.
 * @param filesData - An array that will contain the data from the files.
 * @param numOfLines - An array containing the number of lines of each file.
 */
int getDataFromFiles(int const argc, char* argv[],
                     unsigned int filesData[NUM_OF_FILES][MAX_DEPTH],
                     unsigned int numOfLines[NUM_OF_FILES])
{
    if (argc != NUM_OF_ARGUMETNS)
    {
        printf("Usage: PitPlugger <Pit Radius input file> <Stone Radius input file>\n");
        return EXIT_FAILURE;
    }
    int currFileNum;
    for (currFileNum = 0; currFileNum < NUM_OF_FILES; currFileNum++)
    {
        FILE* currFile = fopen(argv[currFileNum + 1], "r");
        if (currFile == NULL)
        {
            printf("Unable to open file %s.\n", argv[currFileNum]);
            return EXIT_FAILURE;
        }
        numOfLines[currFileNum] = readVector(currFile, filesData[currFileNum]);
        fclose(currFile);
    }
    return EXIT_SUCCESS;
}

int main(int argc, char* argv[])
{
    unsigned int filesData[NUM_OF_FILES][MAX_DEPTH];
    unsigned int numOfLines[NUM_OF_FILES];
    if (getDataFromFiles(argc, argv, filesData, numOfLines) == EXIT_FAILURE)
    {
        return EXIT_FAILURE;
    }
    unsigned int stonesInPit[numOfLines[PIT_LOCATION] + ROOF_STONES];
    initArray(stonesInPit, numOfLines[PIT_LOCATION] + ROOF_STONES);
    stonesToPit(filesData[PIT_LOCATION], filesData[STONE_LOCATION], stonesInPit, numOfLines);
    char pitMatrix[ROWS][COLS];
    printResultMessage(stonesInPit, filesData[PIT_LOCATION], numOfLines);
    createPitMatrix(pitMatrix, filesData[PIT_LOCATION], stonesInPit, numOfLines);
    drawArray(pitMatrix);
    return EXIT_SUCCESS;
}
```

5 RadiusReader.c

```c
/*
 * RadiusReader.c
 *
 *  Created on: Jul 29, 2014
 *      Author: roeia1
 */
#include <stdio.h>
#include "RadiusReader.h"

/**
 * Read a vector of unsigned ints from a given FILE
 * each line contains a single unsigned integer
 * @return number of lines read
 */
unsigned int readVector(FILE* file, unsigned int vec[MAX_DEPTH])
{
    int currLine = 0;
    while ((readSingleUInt(file, &vec[currLine]) == SUCC_CODE) && (currLine < MAX_DEPTH))
    {
        currLine++;
    }
    return currLine;
}

/**
 * Read a single unsigned int from a given FILE
 * each line contains a single unsigned integer
 * @return SUCC_CODE iff successful, FAIL_CODE otherwise
 */
int readSingleUInt(FILE* file, unsigned int* val)
{
    if (fscanf(file, "%u", val) == 1)
    {
        return SUCC_CODE;
    }
    return FAIL_CODE;
}
```