**Project 1**

Title

**Minesweeper**

Course

**CIS-17C**

Section

**48942**

Due Date

**November 19, 2015**

Author

**Edgar M Trujillo Sanchez**

**Introduction**

Title: Minesweeper

Minesweeper is a single player puzzle game. Its goal is to clear a board, also called minefield, without detonating any mines. The game starts with the player being asked to provide the desired length of the minefield’s sides. Using this information, a square minefield is formed with mines randomly placed. The number of mines found in the minefield is equivalent to the length of the sides. At this point, the user must start choosing locations on the board by providing row and column numbers and either revealing the location’s contents, with the left-click option, or placing a flag on the selected location, with the right-click option. This will be repeated until the user clears the whole board or until a mine is detonated.

The user also has the choice to restart the game or exit the game while playing. To restart the game the user must enter a ‘D’ when being asked to provide a right or left click. This will provide a new minefield with its mines at new random locations. To exit the game the user must enter an ‘E’ when being asked to provide a right or left click. This will exit the game immediately.

The user is thanked for playing before ending the game and whether the player wins, loses, or decides to exit the game.

**Summary**

Project size: 801 lines

Number of variables: 23

STL library components used: Yes

This project includes many of the concepts we have covered in class so far. For example, I used the templated vector class that we developed, through homework assignments, to implement the grid that stored the mines and the mine counts on the locations neighboring the mines. Also, I utilized the STL vector to implement the grid printed to the console screen during playtime. This grid was constantly updated and printed to the screen as the player continued to solve the minefield puzzle. The STL library fill algorithm was used to initialize the STL vector before playtime. Moreover, the STL vector’s assign function was used allocate the necessary memory for the playtime grid after the user had specified the desired size of the minefield.

It was fairly challenging to develop this project. One of the main issues I had was to synchronize all of my loops for the restart game option of the game. I also had a hard time figuring out which of the STL containers was best to implement this game. Another problem I had was that I was using one-dimensional vectors to implement two-dimensional grids. In consequence, I had to utilize an offset when accessing the vectors.

To complete this project it took me about a week and a half and a total of about forty-two hours. The reason I took that long was because I was not familiar with the STL library and the offset issue.

There are some other features that would make this game more enjoyable but I did not have time to add them. One of them is adding a playtime clock which would keep track of how long the player takes to solve the puzzle. Furthermore, after adding the clock functionality, a highscores function could be implemented to keep track of the players that win the game in the fastest times.

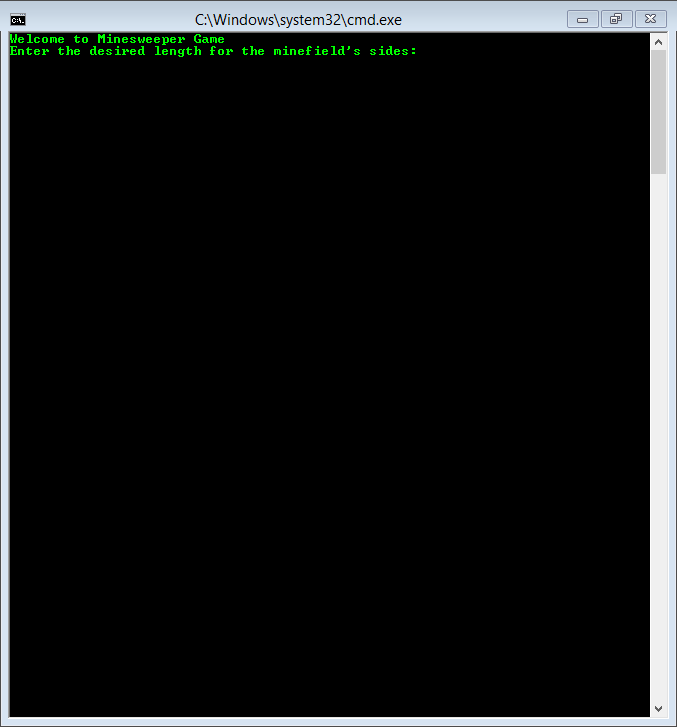
**Description**

This minesweeper game was implemented using a class called Minefield. Minefield’s main components are an integer variable for the height of the minefield, an integer variable for the width of the minefield, a vector of integers created using my own templated vector implementation, and a STL vector of character values. My vector was used to create an instance of the board that stored the locations of the mines and the mine counts on the locations neighboring the mines. In this board, mines were represented by the integer value of 9. The STL vector was used to create an instance of the minefield that the player would see during playtime. In this version of the minefield, mines were represented by the character value ‘X’.

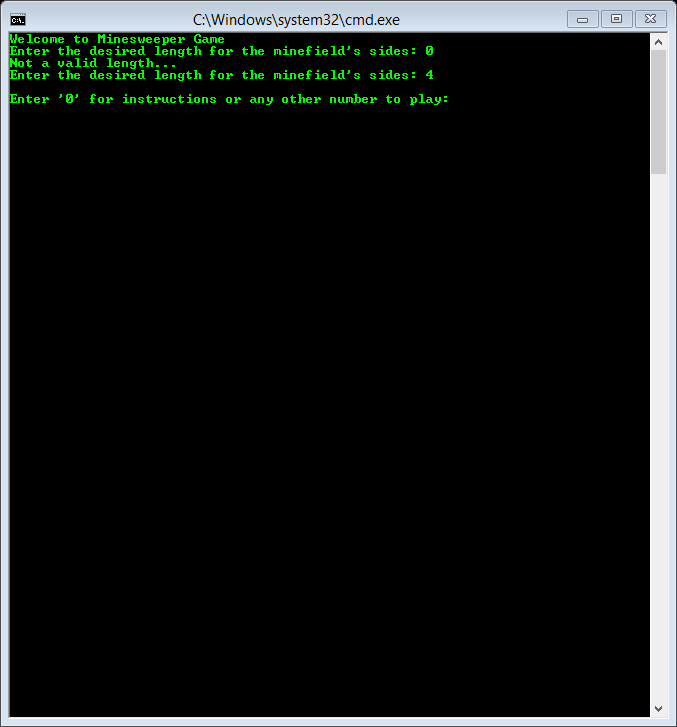
-Sample Inputs/Outputs

All the inputs for this game are given by the user while the game is running.

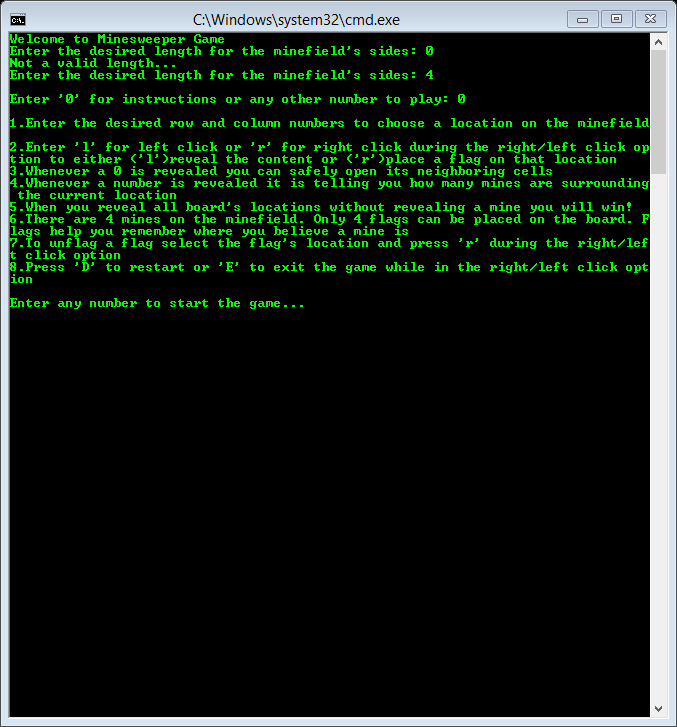
Minesweeper starts by welcoming the user and asking him/her for the desired size of the minefield’s sides:



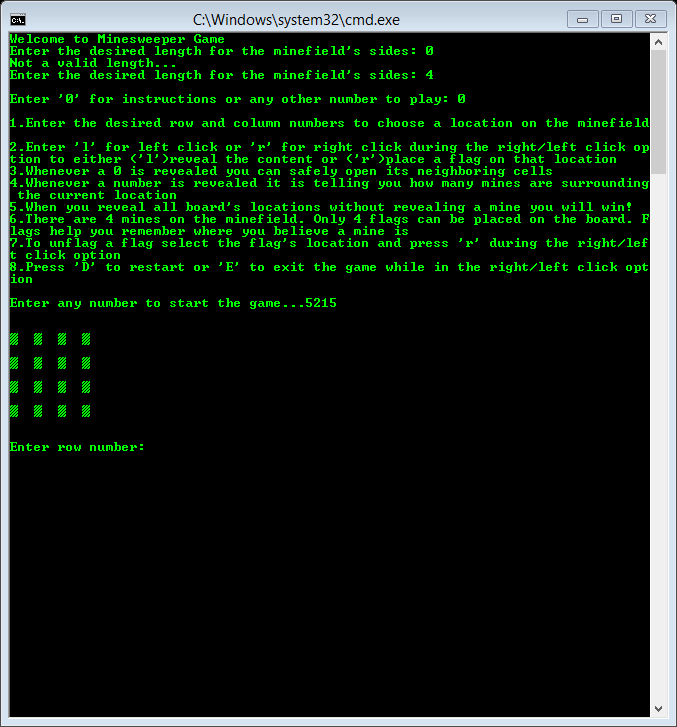
Once the user enters a valid length for the sides, the user is given the choice to see the game’s instructions or to start playing:



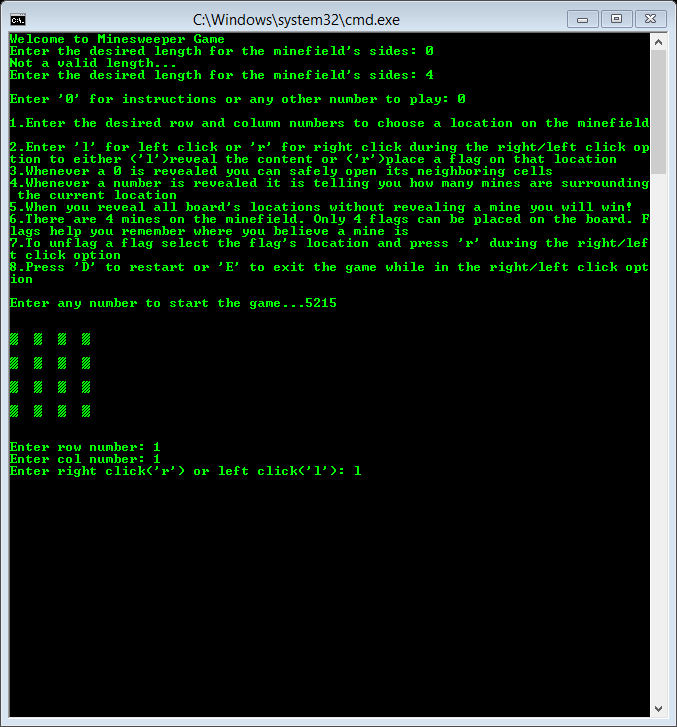
When a ‘0’ is entered, the instructions to how to play the game are displayed. After this, the user must enter any number to start playing:

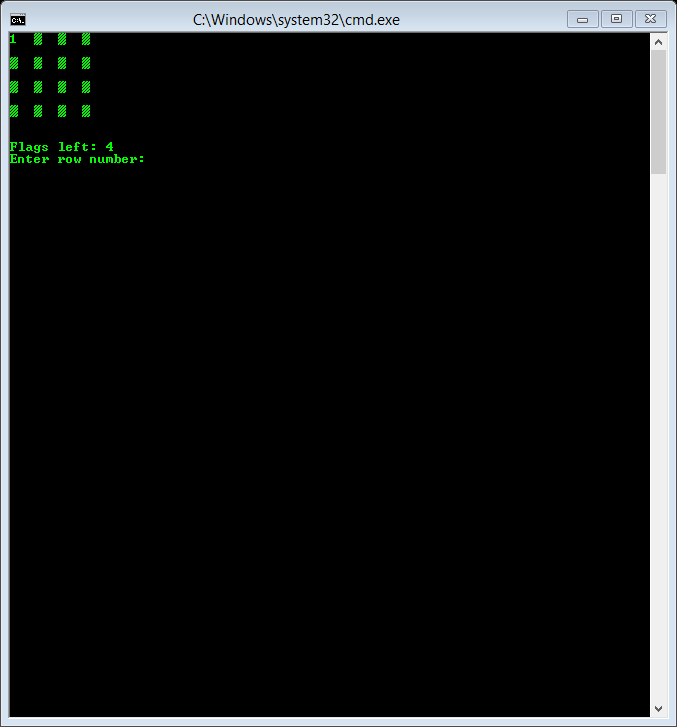


Once the user starts playing, the minefield with its contents hidden is displayed:

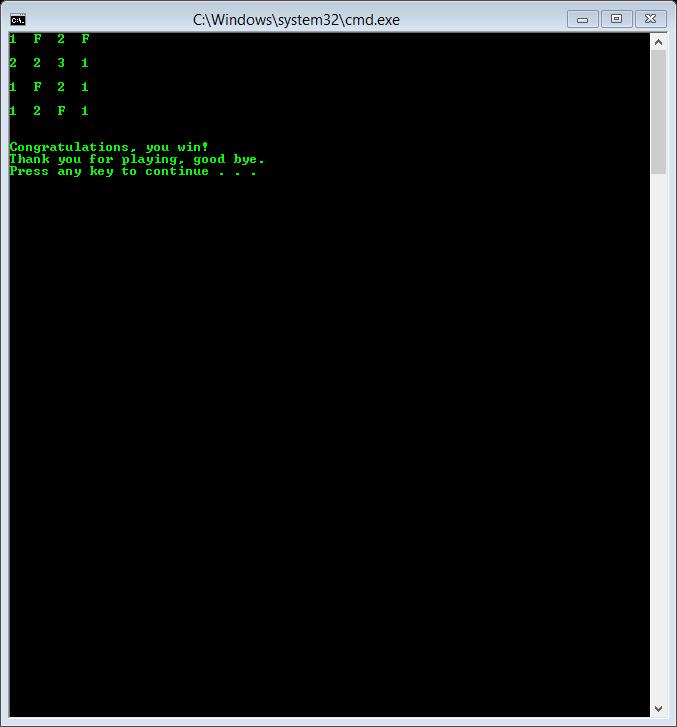


After the user selects the first location and decides whether to reveal its contents or place a flag, the console screen is cleared and only the updated minefield is displayed:

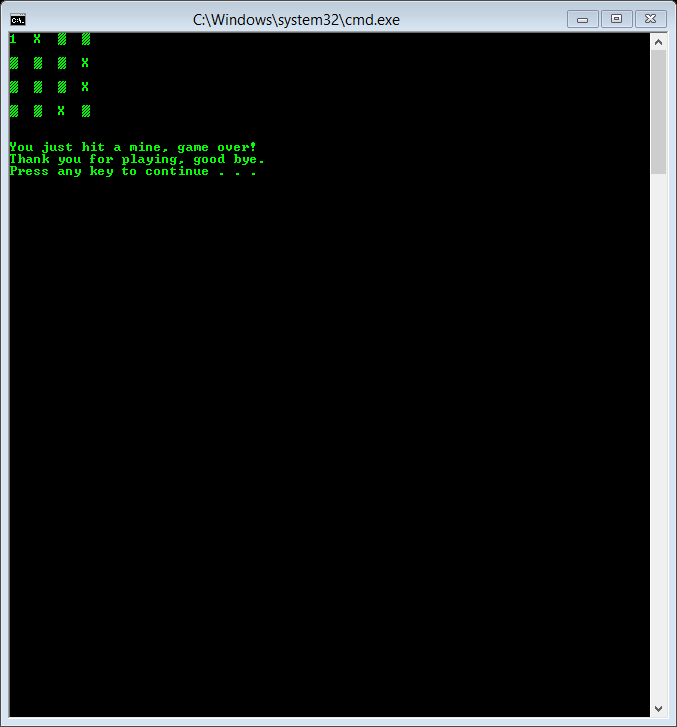




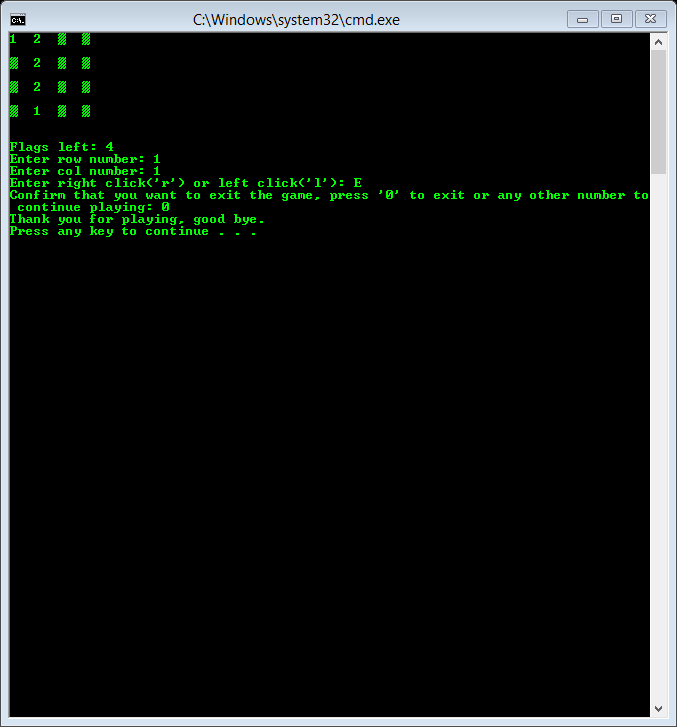
The user must continue choosing other locations and decided whether to reveal the contents of the location or place a flag. Once the user clears the whole minefield he/she wins:



If a mine is detonated at any point, the user loses and the game ends:



To restart or exit the game the user must enter a ‘D’ or an ‘E’ while playing and during the right/left click option:



-Flowchart

if play

if help

END

minefield.endGame()

if control = false

exit game or game over

if control = true

restart game

minefield.playTime(control)

minefield.allocateGrids()

minefield.setShown()

minefield.setMines()

minefield.setCount()

minefield.printPlayGrid()

do-while loop

bool control = true;

minefield.helpPlay()

minefield.instructions()

Help or Play

minefield.setSize(length)

Set size of board

minefield.sidesLength()

user input

int main()

Get size of board

Declare Minefield object

int height=0;

int width=0;

myVector<int> grid; size=0

STL vector<char> playGrid; size=0

START

-Variables

All variables will be separated according to the file they appear on

main.cpp

// Declaring minefield object

Minefield minefield;

//Variable holds length of the sides of the minefield

int length=0;

//Variable will determine when the game ends

bool control=true;

Minefield.h

MyVectorDA<int> grid;

vector<char> playGrid;

int height;

int width;

Minefield.cpp

// Variable holds length of the sides of the minefield

int length=0;

// For loop counters

int i=0;

int j=0;

//ASCII value 178

char a=178;

//counter for mines surrounding grid location

int count=0;

//counter for mines inside for loop

int mineC=0

// Choice for helpPlay function

int choice;

//will hold the user-given row number

int urow=0;

//will hold the user-given col number

int ucol=0;

//will hold user-given right/left click

char uclick=0;

//will keep track of open locations

int totalCtn=0;

//holds total locations

int total=grid.size();

//will keep track of how many flags are placed by user

int flagCtn=0;

//ASCII value 178 for hidden locations on playGrid

char hide=178;

// Confirm that user wants to restart/exit game

int confirm;

// holds converted int value

char temp;

**References**

1. Class Textbook: Introduction to Algorithms

2. Homework Assignments

3. Following websites:

-http://minesweeperonline.com

-http://www.theasciicode.com.ar/

-http://www.cplusplus.com (for STL vector and clear console screen)

-http://stackoverflow.com (converting an int to ASCII character)

-https://www.sgi.com/tech/stl/ (for STL library)

**Program**

/\*

\* Edgar M Trujillo Sanchez

\* Nov 19th, 2015

\* Purpose: Project1 - This program will simulate the game Minesweeper

\* using the STL vector and some of its algorithms, as well as my own

\* vector class.

\*/

#include <iostream>

#include<ctime>

#include "Minefield.h"

using namespace std;

int main()

{

// Initialize the random number generator

srand(static\_cast<unsigned int>(time(0)));

// Welcomes user to game

cout<<"Welcome to Minesweeper Game"<<endl;

// Declaring minefield object

Minefield minefield;

//// Getting length of minefield's sides

//Variable holds length of the sides of the minefield

int length=0;

length=minefield.sidesLength();

// Set size of minefield

minefield.setSize(length);

cout<<endl;

// Decide if help or play

minefield.helpPlay();

// Loop will continue looping until user wins or user decides to exit game

bool control=true;

do

{

// Allocate grids with user-given size

minefield.allocateGrids();

// Set playGrid will all locations hidden

minefield.setShown();

// Set mines on grid

minefield.setMines();

// Set count after placing mines

minefield.setCount();

// Print grid

//minefield.printGrid();

// Print playGrid

minefield.printPlayGrid();

// Start playing game

minefield.playTime(control);

//end-of-line statement for when restarting the game

if(control==true)

cout<<endl;

}while(control==true);

// End of game

minefield.endGame();

return 0;

}

/////// Minefield class

#ifndef MINEFIELD\_H

#define MINEFIELD\_H

#include <iostream>

#include <vector>

#include "MyVectorDA.h"

using namespace std;

class Minefield

{

private:

MyVectorDA<int> grid;

vector<char> playGrid;

int height;

int width;

public:

// Default constructor

Minefield();

// Destructor declaration

~Minefield();

// Function that gets the length for the minefield's sides

int sidesLength();

// Function that sets the length of height and width of minefield

void setSize(int);

// Function that allocates the necessary memory for both grid and playGrid according to height and width

void allocateGrids();

// Function that prints to screen the contents of grid

void printGrid();

// Function that prints to screen the contents of playGrid

void printPlayGrid();

// Function that fills up playGrid with ASCII value 178, all locations hidden

void setShown();

// Function that cycles through all grid locations and stores the amount of mines surrounding each

//grid location using the CountNeighbors function

void setCount();

// Function that cycles through all grid locations and counts how many mines surround given location

int countNeighbors(int, int);

// Function that sets "heigth" many mines on grid

void setMines();

// Function that displays rules of how to play game

void instructions();

// Function asks the user whether it needs help of how to play or wants to play

void helpPlay();

// Function that gets the desired location and right/left click from user, restarts/exits game

//if user chooses to. It also either reveals location's content, places a flag, removes a

//flag, ends the game if a mine is uncovered, or ends the game when player wins

void playTime(bool &);

// Function that converts an int value into a char value and returns it

char revealValue(int);

// Function that clears the console screen and prints the contents of

//playGrid with all the mines when user loses

void losePlayGrid();

// Function that clears the console screen and prints the contents of

//playGrid with all the flags when user wins

void winPlayGrid();

// Function that clears the console screen

void clearScreen();

// Function that thanks the user for playing the game

void endGame();

};

#endif

///////Minefield.cpp, implementation file of class

//System Libraries

#include <iostream>

#include <string>

#include <ctime>

#include "Minefield.h"

#include "MyVectorDA.h"

using namespace std;

// Default constructor

Minefield::Minefield()

{

height=0;

width=0;

}

// Destructor

Minefield::~Minefield()

{

height=0;

width=0;

}

// SidesLength function. It gets the length for the minefield's sides

int Minefield::sidesLength()

{

// Variable holds length of the sides of the minefield

int length=0;

//Will continue looping until length is bigger than 0

while(true)

{

cout<<"Enter the desired length for the minefield's sides: ";

cin>>length;

//if user gives a length of less than one

if(length<1)

cout<<"Not a valid length..."<<endl;

//if length is valid break from loop and continue executing program

else

break;

}

return length;

}

// SetSize function. It sets the length of height and width of minefield

void Minefield::setSize(int size)

{

height=size;

width=size;

}

// AllocateGrids function. It allocates the necessary memory for both grid and

//playGrid according to height and width

void Minefield::allocateGrids()

{

grid.allocate(height\*width);

playGrid.assign(height\*width,'0');

}

// PrintGrid function. It prints to screen the contents of grid

void Minefield::printGrid()

{

// Print contents of grid

for(int i = 0; i<height; i++)

{

for(int j=0; j<width; j++)

cout << grid.getElementAt(i\*height+j) << " ";

cout<<endl<<endl;

}

cout<<endl;

}

// PrintPlayGrid function. It prints to screen the contents of playGrid

void Minefield::printPlayGrid()

{

// Print contents of playGrid

for(int i=0; i<height; i++)

{

for(int j=0; j<width; j++)

cout<<playGrid[i\*height+j]<< " ";

cout<<endl<<endl;

}

cout<<endl;

}

// SetShown function. It fills up playGrid with ASCII value 178, all locations hidden

void Minefield::setShown()

{

char a=178; //ASCII value 178

//Using fill algorithm to initialize playGrid STL vector

fill(playGrid.begin(), playGrid.end(), a);

}

// SetCount function. It cycles through all grid locations and stores the amount

//of mines surrounding each grid location using the CountNeighbors function

void Minefield::setCount()

{

//cycle through all grid locations

for (int i= 0;i<height;i++)

{

for(int j=0; j<width; j++)

{

//if mine, no need to count

if (grid.getElementAt(i\*height+j)==9)

continue;

int count=countNeighbors(i,j);

grid.operator[](i\*height+j)=count;

}

}

}

// CountNeighbors function. It cycles through all grid locations and counts how many

//mines surround the given location

int Minefield::countNeighbors(int row, int col)

{

//counter for mines surrounding grid location

int count=0;

//cycle through all surrounding locations of given grid location

for (int i=row-1; i<=row+1; i++)

{

for(int j=col-1; j<=col+1; j++)

{

//bounds check

if (i<0 || i>=height || j<0 || j>=width)

continue;

//skip middle location

if (i==row && j==col)

continue;

//current location is a mine

if (grid.getElementAt(i\*height+j)==9)

//increase count by one when a mine is found

count++;

}

}

//return how many mines were found

return count;

}

// SetMines function. It sets "heigth" many mines on grid

void Minefield::setMines()

{

//Will keep looping until "heigth" many mines are set on grid

for (int mineC=0; mineC<height; mineC++)

{

int row = rand() % height;

int col = rand() % width;

// If random location is already a mine, mineC decreases

// by one so another mine is set in a new location

if (grid.getElementAt(row\*height+col)==9)

mineC--;

// Random location stores a mine

grid.operator[](row\*height+col)=9;

}

}

// Instructions function. It displays rules of how to play game

void Minefield::instructions()

{

cout<<endl;

cout<<"1.Enter the desired row and column numbers to choose a location on the minefield"<<endl;

cout<<"2.Enter 'l' for left click or 'r' for right click during the right/left click "<<

"option to either ('l')reveal the content or ('r')place a flag on that location"<<endl;

cout<<"3.Whenever a 0 is revealed you can safely open its neighboring cells"<<endl;

cout<<"4.Whenever a number is revealed it is telling you how many mines are surrounding "<<

"the current location"<<endl;

cout<<"5.When you reveal all board's locations without revealing a mine you will win!"<<endl;

cout<<"6.There are "<<height<<" mines on the minefield. Only "<<height<<" flags can be placed "<<

"on the board. Flags help you remember where you believe a mine is"<<endl;

cout<<"7.To unflag a flag select the flag's location and press 'r' during the right/left "<<

"click option"<<endl;

cout<<"8.Press 'D' to restart or 'E' to exit the game while in the right/left click option"<<endl;

cout<<endl;

}

// HelpPlay function. It asks the user whether it needs help of how to play or wants to play

void Minefield::helpPlay()

{

int choice;

cout<<"Enter '0' for instructions or any other number to play: ";

cin>>choice;

//Will display how to play game if user enter a '0'

if(choice==0)

{

instructions();

//Value given by user will not be used, this lets user know that the game will start

cout<<"Enter any number to start the game...";

cin>>choice;

cout<<endl<<endl;

}

else

cout<<endl<<endl;

}

// PlayTime function. It gets the desired location and right/left click from user, restarts/exits

//game if user chooses to. It also either reveals location's content, places a flag, removes a

//flag, ends the game if a mine is uncovered, or ends the game when player wins

void Minefield::playTime(bool &set)

{

int urow=0; //will hold the user-given row number

int ucol=0; //will hold the user-given col number

char uclick=0; //will hold user-given right/left click

int totalCtn=0;//will keep track of open locations

int total=grid.size();//holds total locations

int flagCtn=0;//will keep track of how many flags are placed by user

char hide=178; //ASCII value 178 for hidden locations on playGrid

// Will keep looping until a mine is uncovered, until the user wins, or if the user decides

//to restart or exit the game

for(int i=1; i>0; i++)

{

//Getting desired location from user and either right/left click

cout<<"Enter row number: ";

cin>>urow;

cout<<"Enter col number: ";

cin>>ucol;

cout<<"Enter right click('r') or left click('l'): ";

cin>>uclick;

//Decrease urow and ucol by one to match memory "coordinates"

urow--;

ucol--;

//Check if the location's "coordinates" are in-bounds

if(urow<0||urow>=height||ucol<0||ucol>=height)

continue;

////// Options if user wants to restart/exti game

//Restart game

if(uclick=='D'||uclick=='d')

{

int confirm;

cout<<"Confirm restart of game, press '0' to restart or any other number to continue playing: ";

cin>>confirm;

//If restart the game is confirmed

if(confirm==0)

{

//Set=true to restart game

set=true;

//Breaks for-loop and ends function

break;

}

//User decideds not to restart the game

else

//Skips all code after and starts a new iteration of for-loop

continue;

}

//Exit game

else if(uclick=='E'||uclick=='e')

{

int confirm;

cout<<"Confirm that you want to exit the game, press '0' to exit or any other number to continue playing: ";

cin>>confirm;

//If exit the game is confirmed

if(confirm==0)

{

//Set=false to exit game

set=false;

//Breaks for-loop and ends function

break;

}

//User decideds not to exit the game

else

//Skips all code after and starts a new iteration of for-loop

continue;

}

//If left click and chosen location is not a mine

if(uclick=='l'&&grid.getElementAt(urow\*height+ucol)!=9&&playGrid[urow\*height+ucol]==hide)

{

//temp2 holds current grid value

int temp2=grid.getElementAt(urow\*height+ucol);

//current location of playGrid set to char version of variable temp2

playGrid[urow\*height+ucol]=revealValue(temp2);

//increases total count by one

totalCtn++;

}

//Placing a flag

else if(uclick=='r'&&flagCtn<height)

{

playGrid[urow\*height+ucol]='F'; //places flag on chosen location

totalCtn++; //increases total count by one

flagCtn++; //increases flag count by one

}

//Removing a flag

else if(uclick=='r'&&playGrid[urow\*height+ucol]=='F')

{

playGrid[urow\*height+ucol]=hide; //removes flag by hiding chosen location

totalCtn--; //decreases total count by one

flagCtn--; //decreases flag count by one

}

//Chosing a mine

else if(uclick=='l'&&grid.getElementAt(urow\*height+ucol)==9&&playGrid[urow\*height+ucol]!='F')

{

//function that clears screen and prints playGrid with all mines

losePlayGrid();

cout<<"You just hit a mine, game over!"<<endl;

//Set=false to exit game

set=false;

//Breaks for-loop and ends function

break;

}

//Check if all user has win game

if(totalCtn==total)

{

//function that clears screen and prints playGrid with all flags

winPlayGrid();

cout<<"Congratulations, you win!"<<endl;

//Set=false to exit game

set=false;

//Breaks for-loop and ends function

break;

}

//function that clears the console screen

clearScreen();

//function that prints current version of playGrid

printPlayGrid();

//flags left to place on minefield

cout<<"Flags left: "<<height-flagCtn<<endl;

}

}

// RevealValue function. It converts an int value into a char value and returns it.

char Minefield::revealValue(int value)

{

char temp;

// All possible values of variable value should be between 0-8

for(int i=0; i<=8; i++)

{

if(value==i)

// Converts int(i) into char(temp)

temp='0'+i;

}

return temp;

}

// LosePlayGrid function. It clears the console screen and prints the contents

//of playGrid with all the mines when user loses

void Minefield::losePlayGrid()

{

//This clears the console screen

clearScreen();

// Print contents of playGrid with all mines

for(int i=0; i<height; i++)

{

for(int j=0; j<width; j++)

{

// If grid's current location is a mine, set playGrid's current location to 'X'

if(grid.getElementAt(i\*height+j)==9)

playGrid[i\*height+j]='X';

cout<<playGrid[i\*height+j]<< " ";

}

cout<<endl<<endl;

}

cout<<endl;

}

// WinPlayGrid function. It clears the console screen and prints the contents of

//playgrid with all the flags when user wins

void Minefield::winPlayGrid()

{

//This clears the console screen

clearScreen();

// Print contents of playGrid with all flags

for(int i=0; i<height; i++)

{

for(int j=0; j<width; j++)

{

// If grid's current location is a mine, set playGrid's current location to 'F'

if(grid.getElementAt(i\*height+j)==9)

playGrid[i\*height+j]='F';

cout<<playGrid[i\*height+j]<< " ";

}

cout<<endl<<endl;

}

cout<<endl;

}

// EndGame function. It thanks the user for playing

void Minefield::endGame()

{

cout<<"Thank you for playing, good bye."<<endl;

}

// ClearScreen function. It clears the console screen

void Minefield::clearScreen()

{

//Windows OS specific, might need to change

system("CLS");

}

/////// MyVectorDA class template

#ifndef MYVECTORDA\_H

#define MYVECTORDA\_H

#include <iostream>

#include <new> // Needed for bad\_alloc exception

#include <cstdlib> // Needed for the exit function

using namespace std;

template <class T>

class MyVectorDA

{

private:

T \*aptr; // To point to the allocated array

int arraySize; // Number of elements in the array

void memError(); // Handles memory allocation errors

void subError(); // Handles subscripts out of range

public:

// Default constructor

MyVectorDA()

{

aptr = 0; arraySize = 0;

}

// Constructor declaration

MyVectorDA(int);

// Copy constructor declaration

MyVectorDA(const MyVectorDA &);

// Destructor declaration

~MyVectorDA();

// Accessor to return the array size

int size() const

{

return arraySize;

}

// Allocate memory

void allocate(int );

// Accessor to return a specific element

T getElementAt(int position);

// Overloaded [] operator declaration

T &operator[](const int &);

// Overloaded = (Assignment) Operator

MyVectorDA &operator= (const MyVectorDA &);

// Function that adds a new item into the array

void push(int &, int&, T, MyVectorDA &);

// Function that deletes the last item added into the array

void pop(int &);

// Function that sorts the Vector from smallest to biggest value

void markSrt(int);

};

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Constructor for SimpleVector class. Sets the size of the \*

// array and allocates memory for it. \*

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

template <class T>

MyVectorDA<T>::MyVectorDA(int s)

{

arraySize = s;

// Allocate memory for the array.

try

{

aptr = new T[s];

}

catch (bad\_alloc)

{

memError();

}

// Initialize the array.

for (int count = 0; count < arraySize; count++)

\*(aptr + count) = 0;

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Copy Constructor for SimpleVector class. \*

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

template <class T>

MyVectorDA<T>::MyVectorDA(const MyVectorDA &obj)

{

// Copy the array size.

arraySize = obj.arraySize;

// Allocate memory for the array.

aptr = new T[arraySize];

if (aptr == 0)

memError();

// Copy the elements of obj's array.

for (int count = 0; count < arraySize; count++)

\*(aptr + count) = \*(obj.aptr + count);

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Destructor for SimpleVector class. \*

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

template <class T>

MyVectorDA<T>::~MyVectorDA()

{

if (arraySize > 0)

delete[] aptr;

arraySize=0;

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// memError function. Displays an error message and \*

// terminates the program when memory allocation fails. \*

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

template <class T>

void MyVectorDA<T>::memError()

{

cout << "ERROR:Cannot allocate memory.\n";

exit(EXIT\_FAILURE);

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// subError function. Displays an error message and \*

// terminates the program when a subscript is out of range. \*

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

template <class T>

void MyVectorDA<T>::subError()

{

cout << "ERROR: Subscript out of range.\n";

exit(EXIT\_FAILURE);

}

// Allocate memory

template <class T>

void MyVectorDA<T>::allocate(int s)

{

arraySize = s;

// Allocate memory for the array.

try

{

aptr = new T[s];

}

catch (bad\_alloc)

{

memError();

}

// Initialize the array.

for (int count = 0; count < arraySize; count++)

\*(aptr + count) = 0;

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// getElementAt function. The argument is a subscript. \*

// This function returns the value stored at the sub- \*

// cript in the array. \*

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

template <class T>

T MyVectorDA<T>::getElementAt(int sub)

{

if (sub < 0 || sub >= arraySize)

subError();

return aptr[sub];

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Overloaded [] operator. The argument is a subscript. \*

// This function returns a reference to the element \*

// in the array indexed by the subscript. \*

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

template <class T>

T &MyVectorDA<T>::operator[](const int &sub)

{

if (sub < 0 || sub >= arraySize)

subError();

return aptr[sub];

}

// Overloaded = (Assignment) Operator

template <class T>

MyVectorDA<T> &MyVectorDA<T> :: operator= (const MyVectorDA &a)

{

// Copy the array size.

arraySize = a.arraySize;

// Allocate memory for the array.

aptr = new T[arraySize];

if (aptr == 0)

memError();

// Copy the elements of a's array.

for(int count = 0; count < arraySize; count++)

\*(aptr + count) = \*(a.aptr + count);

return \*this; // Returning current object

}

// Push function. It adds a new item into the array

template <class T>

void MyVectorDA<T>::push(int &current, int &usage, T nItem, MyVectorDA & obj)

{ //Regular push, new item added into array

if (usage < current)

{

operator[](usage) = nItem;

usage++;

}

//Complex push, array size is doubled before addding new item into array

else if (usage = current)

{

current = current \* 2; //double the current size of array

int\* temp = new int[current]; //create a temporary array to store current contents of array

//copy current contents of array into temp array

for (int i = 0; i<usage; i++)

temp[i] = obj.operator[](i);

//initiallize unused space on temp array to 0

for (int i = usage; i<current; i++)

temp[i] = 0;

temp[usage] = nItem;//copy new item into next avaliable index on array

arraySize = current;//change original array size to new current size

usage++;//increase usage counter

delete[] obj.aptr;//delete old original array

obj.aptr = temp;//point original array's pointer to temp array

}

else

{

cout << "ERROR: There's no more space on the array.\n";

exit(EXIT\_FAILURE);

}

}

// Pop function. It deletes the last item added into the array

template <class T>

void MyVectorDA<T>::pop(int &usage)

{ //pops one item from array

if (usage >= 0)

{

operator[](usage - 1) = 0;

usage--;//decrease usage counter

}

else

{

cout << "ERROR: There's no more items on the array that can be pop.\n";

exit(EXIT\_FAILURE);

}

}

// MarkSort function. It sorts the Vector from smallest to biggest value

template <class T>

void MyVectorDA<T>::markSrt (int n)

{

for(int position=0; position<n-1; position++)

{

for(int next=position+1; next<n; next++)

{

if(\*(aptr+position) > \*(aptr+next) )

{

int temp=\*(aptr+position);

\*(aptr+position)=\*(aptr+next);

\*(aptr+next)=temp;

}

}

}

}

#endif