Homework 1: Simple Dungeon Maps

**Due: Sunday, September 22th at 11:59pm (BEFORE midnight)**

Many early computer games were ASCII-based. For example, Rogue was a game that used the symbols found in the ASCII character set to represent an area the player was exploring:



Fortunately, this lines up well with the console applications we’ve been making so far. You are going to use that idea to create a simple visual dungeon in the console. Some code will be provided for you to use, since the program will be using some parts of C# that we have not covered in class yet.

# Task Overview

This is a brief overview of the tasks you must complete for this assignment. Specifics are given in the corresponding sections later in this document.

***Read the entire document*** ***before starting!!!***

## Activity 1

* Create a new C# project and add a new code file called Dungeon.cs
* Add appropriate fields, constructors and methods to the Dungeon class
* In Main(), create a Dungeon object and use it to draw a simple dungeon in the console

## Activity 2

* Enhance your Dungeon class methods with optional parameters
* Alter code in your Main method to use these enhancements

Even though there are two separate activities, you will simply be using and submitting ***one*** C# project for this assignment. This may differ from assignments given to students in other sections of 105.

# “Drawing” in the Console

You’ll be creating a Dungeon class with methods for drawing the pieces of a dungeon (walls, items, etc.) in the console window. Your Main method will create a Dungeon object and call its methods to draw a dungeon which matches the general layout of the Sample Dungeon later in this document. The exact symbols and colors you use are up to you, but the overall appearance (two rooms, a hallway and some objects) should be similar.

To “draw” in the console, you’ll be treating the console window as a two-dimensional grid of characters. The Console class has static CursorLeft and CursorTop properties, which correspond to the column and row of the cursor, respectively. By changing these properties, you can specify where the next output will appear. This allows us to use the Console window as a pseudo-drawing application, where we write out individual characters instead of coloring pixels. This is similar to how old text-based games like Rogue worked.

**Example (with coordinates):**

Columns

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 |
| 0 |  |  |  |  |  |
| 1 |  |  |  |  |  |
| 2 |  |  | (2,2) |  |  |
| 3 |  |  | (2,3) |  |  |
| 4 |  |  | (2,4) | (3,4) |  |
| 5 |  |  |  |  |  |

Rows

If you want to design a more complex dungeon, you may even want to start with a numbered piece of graph paper and draw the dungeon on it. The numbers will tell you starting points for walls or locations for objects.

# Activity 1: The Dungeon Class

After creating a new project, your first job will be to create a Dungeon class which contains the methods used to draw specific dungeon pieces to the console.

We’ll be covering the basics of adding new code files in class this week, but here’s a quick primer:

1. Right click your project (the bold one with the “C#” icon) in Visual Studio’s Solution Explorer
2. Go to the “Add” sub-menu, then choose “New Item…” or “Class…”
3. Make sure “Class” is highlighted in the center column of the window that comes up
4. Give it an appropriate name at the bottom, and press “Add”

(See the next page for the specific requirements of the Dungeon class.)

## Dungeon Class Members

### Fields

The Dungeon class should have two private integer fields, specifying the width and height of the dungeon (and, coincidentally, the console window). These will be set by the constructor, and used throughout the class to ensure we don’t attempt to write characters outside the bounds of the window.

### Constructor

There should be one parameterized constructor for the class (and no default constructor). It should accept three parameters: the width of the dungeon, the height of the dungeon and the name of the dungeon. Save the width and height in the appropriate fields, and then use them to set the Console class’s WindowWidth and WindowHeight properties. Also change the Console class’s Title property using the name parameter. This will update the text in the console window’s title bar. Ensure the width is not greater than 100 and the height is not greater than 60. *(Note: If you are using a screen with a lower resolution, you may get an error at values less than 60 – if so, still code for a maximum height of 60 but use a smaller value when calling the constructor.)*

### void DrawWallVertical(int x, int y, int length)

This method will be responsible for drawing vertical walls (as the name implies). It will be a void method that takes 3 parameters: the starting x position of the wall, the starting y position, and the length of the wall. “Draw” the wall to the console by looping and writing one character at a time, starting at the top and moving down.

Your method will need to handle cases where either end of the requested wall potentially hangs off the edge of the screen. In these cases, only the portion of the wall that would be visible on the screen should be shown. Attempting to move the cursor outside of the windows bounds will crash your program, so check each potential position against the Console class’s WindowWidth and/or WindowHeight properties as necessary.

It’s up to you if you’d like to accept negative lengths (walls drawn from the starting position upwards). If not, the method should return early for negative and zero length walls.

### void DrawWallHorizontal(int x, int y, int length)

This method will do the same thing as DrawWallVertical(), except horizontally.

### void DrawObject(int x, int y, char symbol)

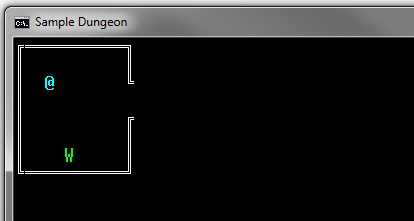
This method will draw a single character to the console. It will also be a void method that takes three parameters: The x position of the object, the y position and the actual character value to print.

Ensure the position is valid (within the bounds of the console) before attempting to write it.

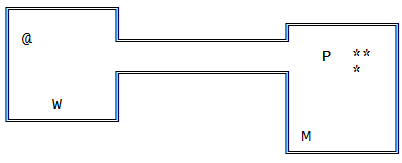
## Main Method

For Activity 1, the Main method of your program should instantiate a Dungeon object and use it to draw a map that resembles the sample dungeon at the bottom of this page.

Directly below is a color-coded example of what your dungeon might look like after completing just the left-most room. Make sure your final dungeon looks similar to the *full* sample dungeon below in that it has two rooms and a hallway – the exact design and orientation is up to you! If you want to match the corners in the sample below, they will need to be drawn using DrawObject().



### Sample Dungeon



The sample dungeon above was created in notepad to give you an idea of what a text-based dungeon *might* look like. The table below shows the meaning of the symbols in the file. Feel free to copy/paste from this document to use these exact characters.

|  |  |
| --- | --- |
| **Symbol** | **Meaning** |
| ║ | Vertical wall |
| ═ | Horizontal wall |
| ╔ ╗╚ ╝ | Corners |
| @ | Your character (the explorer/fighter) |
| \* | Gem |
| M | A monster |
| P | A potion |
| W | A weapon |

# Activity 2: A Slightly Fancier Dungeon

For this activity, you’re going to make some improvements to your Dungeon class. Test your progress as you go by updating the code in your Main method from Activity 1.

At the end of this activity, you should have at least the same basic dungeon layout (two rooms, a hallway and some objects). However, your second room should be “customized” by using the optional parameter feature you are implementing below. That is, each call to a Draw method in the Dungeon class in your second room should include custom characters and colors. If you’d like to expand the dungeon a bit by adding extra rooms and halls, feel free.

## Optional Parameters

Note: This part is NOT optional for you! The name of the language feature is “optional parameters”.

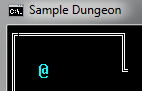
C# supports methods with optional parameters. That is, parameters which have a default value and can be omitted when calling the method. Inside the method, the parameter always exists, but its default value may be overwritten when called. Any optional parameters must appear after all required parameters in a method signature. Here is an example:

void ExampleMethod(int aRequiredNumber, int anOptionalNumber = 5) { }

The method can be called in two ways:

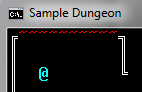
myObject.ExampleMethod(4); // 5 will be used for the optional parameter’s value

myObject.ExampleMethod(4, 99); // 99 will be used instead of 5

For this task, add two optional parameters to bothof the Dungeon class’s wall-drawing methods; these parameters will allow you to specify both a custom color and a custom character to use when “drawing” the dungeon. The third method of Dungeon, DrawObject(), should have one optional parameter for a custom color. The default values should be whatever values you were using during Activity 1. Here are some quick examples:

dungeon.DrawWallHorizontal(1, 0, 10);

The required parameters (start x, start y, and wall length) are present in the above method call. The wall will be drawn using the default symbol & default color, as seen to the right.

dungeon.DrawWallHorizontal(1, 0, 10, ConsoleColor.Red, '~');

The two optional parameters (color and character) allow you to change the color of this piece of the wall and the symbol that makes up the wall. Remember: individual characters use *single quotes* instead of double quotes. Double quotes are for strings only!

When adding optional parameters to these three methods, the code inside each method block will need to change to utilize those parameters instead of hard-coded (literal) data.