Text Adventure (final)

Professor Caleb Fowler

Problem.

Create a simple text based adventure game. The game will be comprised of Room() constructs. These constructs will have a room_name, room_description, room_id, occupied code, and 4 exit codes ('N', 'S', 'E', 'W'). You are free to add other variables as necessary. The exit codes will contain the room_id of the room in that direction. If the room is occupied, randomly select one of 4 monsters to spawn in the room. What, exactly, they do with your character is up to you.

Requirements.

- Move throughout the adventure with the N,S, E, W keys (moving you in the four cardinal directions.
- When a room is occupied, spawn a monster.
- Create a minimum of 5 rooms in your dungeon.
- The game ends when the player leaves through the exit - tell them the game is over.
- Use functions whenever possible to modularize your code. Use function prototypes and code the functions under main().
- Don't forget to incorporate these elements from the style guide:
  - Comments
  - Source file header.
  - Program Greeting - this is NOT the same as the menu!
  - Constant variable style if applicable.
  - No global variables or global variable look a-likes.

Specification Bundles.

Code elements from the specification bundles to control the maximum potential grade you can get for this assignment. The more work you do, the better grade you can get. This is the starting point for your grade.

"C Grade" Specification Bundle.

☐ // Specification C1 - Five Rooms
   Create a minimum of 5 rooms in your dungeon. If you hard code your rooms, put this comment right before the last room.

- [ ] // Specification C2 - Player Name
Ask the player to enter their name. Use this name in some of your game text.
- [ ] // Specification C3 - Input Validation
Perform input validation with the command options. Only allow the entry of valid commands.
- [ ] // Specification C4 – Abbreviated Room Description
Display room_title and room_description the first time the player enters the room. Only display the room_title on subsequent visits to the room.
// Specification C5 - Detailed Look
Show the room_title and room_description every time the player enters the 'L'ook key.

"B Grade" Specification Bundle.

You are to select 3 features to add to your design to meet this specification bundle. Comment each feature you add the same way you've been doing it all class (// Specification B1 ....). I still want to see 3 specification comments. Don't forget a 1 sentence description of the feature you just added in your specification comment. You are free to create other specifications if you wish - just make sure they are equivalent in complexity to these (no "Ready player 1" cout's).

- [ ] Add more command options.
- [ ] Add combat.
- [ ] Add trap encounters. Once revealed (visible) these are permanent. They will appear whenever you subsequently reenter that room.
- [ ] Add treasure. This practically begs you to keep some sort of inventory as well. I suppose you'll also need an <i>nventory command to see it.
- [ ] Add the concept of light and dark rooms.
- [ ] Add more monsters (I'll give you credit for only 1 more monster, regardless of the number you actually add).
- [ ] Add more rooms (only 1 more counts here, more than 1 new room doesn't count).
- [ ] Use a random room generator to create an unlimited number of rooms.
- [ ] Use an array of structs.
- [ ] Put your monsters in a Monster() class.
- [ ] Create a pseudodynamic array. You don't need to grow or shrink this - easy!
- [ ] Store your room information in a file and read that file in at the start of the program.
- [ ] Randomly determine if the room is occupied.

"A Grade" Specification Bundle.

As "B" specification bundle, but add 3 more specifications of your choosing. Comment them individually using the format we have used all along in this class (// Specification A1 – description, etc. etc.). I'm looking for A1, A2 and A3 specifications. Use the suggestions above or come up with your own.

Homework Checklist.

Check the following before you turn in your work:

☐ You coded your homework.

☐ Does it meet all the requirements?

☐ Test your code.

   ☐ Does it compile?

   ☐ Does it have any compiler warnings?

   ☐ Does it run?

   ☐ Does it produce correct output?

   ☐ Did you use the grep trick to make sure I can see your work?

☐ Upload to Canvas.

☐ What's the plagiarism checker score?

Commentary.

This assignment generates more features from my students than any other assignment I give (in any class). The current record holder is 18 additional features. A general review of adventure games can be found here (www.cabinetmagazine.org/issues/64/lucas.php).

Due Date.

This assignment is due by the close of Canvas for the semester.

Late Work.

There is no late penalty for this assignment because it is due when Canvas closes. You simply cannot turn it in after that.

How to Turn in your Homework.

I ONLY accept homework through the Canvas Dropbox. Do not add it to the submission comments or email it to me - I will not accept it. Turn homework in by uploading to the appropriate Canvas Dropbox folder. Save your homework as a .cpp file. Don't zip or otherwise

compress your files. Do NOT split your file up into multiple files. I know that is a standard industry practice, but it just get's in the way for this class.

Create a file with the following naming format: W12345678.cpp (your w number). This allows me to sort the class in alphabetical order - don't stand out here! If you are having trouble submitting the assignment, email me immediately. Don't change your filename if you make multiple submissions - Canvas will keep track of them and download the latest one by default.

Style Guide.

All programs you write MUST have the following code and/or comments. Again, I look for these elements with my scripts, you want me to find them.

Comments.

Use white space and comments to make your code more readable. I run a program called cloc (count lines of code) which actually looks for this stuff.

End of line comments are only permitted with variable declarations. Full line comments are used everywhere else.

Specification Comments.

Specifications are bundled into groups: "A", "B", "C". You must meet the specifications of the lowest group before I will count the specifications for the highest group. For example, you must meet the "B" specifications before I will count the "A" specifications. If you miss one element of a specification bundle, that is the grade you will get for the assignment - regardless of how much extra work you do.

Use whole line comments for Specifications. Put the comment on the line above the start of the code implementing the Specification. If the same Specification code appears in more than 1 place, only comment the first place that Specification code appears. Number your Specifications according to the specification bundle and the specific specification you are using, also provide a very short description. DO NOT BUNCH ALL YOUR SPECIFICATIONS AT THE TOP OF THE SOURCE FILE. Example specification comment:

```
// Specification A2 - Display variables
Your code to do this starts here;
```

It's very important to get the specifications down correctly. If your specification code isn't commented, it doesn't count. I use the grep trick to find your specification code. Proper documentation is part of the solution, just like actually coding the solution is.

Compiler Warnings.

Compiler warnings are a potential problem. They are not tolerated in the production environment. In CISP 360 you can have them. I will deduct a small number of points. CISP 400 - I will deduct lots of points if compiler warnings appear. Make sure you compile with -Wall option. This is how you spot them.

C++ Libraries.

We are coding in C++, not C. Therefore, you must use the C++ libraries. The only time you can use the C libraries is if they haven't been ported to C++ (very, very rare).

Non-Standard Language Extensions.

Some compilers support unapproved extensions to the C++ syntax. These extensions are **unacceptable.** Unsupported extensions are compiler specific and non-portable. Do not use them in your programs.

Program Greeting.

Display a program greeting as soon as the program runs. This is a simple description of what the program does. Example:

```
// Program Greeting
cout « "Simple program description text here." « endl;
```

Source File Header.

Start your source file with a program header. This includes the program name, your name, date and this class. I use the grep trick for .cpp (see below) to look for this. I focus on that homework name and display the next 3 lines. Example:

```
// drake.cpp
// Pat Jones, CISP 413
// 12/34/56
```

Specifications and Specification Bundles.

You document specifications like this: // Specification C1 - Some stuff

You do not need to code them in order. You will probably want to because the specifications get harder as you move up in bundles (not THAT much harder). You also don't need to worry about the specification comments appearing in order in your code, either.

However, all of a specification bundle must be coded to reach that bundle grade (ie all C bundle to get a C). Partially completed bundles

DO NOT COUNT. Say you code all specifications for a B bundle and only 1 for an A bundle (out of 5 for example). The highest grade you would get would be a B because that's the last bundle you've completed.

You can stop at any bundle you want, you just can't get a higher grade (ex, you code all specifications for bundle B - the best you can get for this homework is a B). This is designed to mirror the work word, the more features your code has, usually, the happier your clients are. This also gives you some control over your grade.

This style guide has more information on the specifics of these comments.

Variables.

Constant variables - anytime you have a value which is not supposed to change, that's a constant. We make it read only with the const keyword and signify it with the ALL CAPS style: const PI = 3.14; We prefer using constants because they make the code easier to read. There are a few situations where we do not usually use them, such as starting a loop at zero. However, if we have that loop end at, say, 33, then it's a magic number. What's 33? Who knows? If we use const SIZE = 33; we know what 33 is.

When we have numeric literals appearing in the program we call these magic numbers. We don;t know what they are, but if we change them, the program breaks. hence, magic. Magic numbers are generally frowned upon.

Grep Trick.

Always run your code immediately before your turn it in. I can't tell you how many times students make 'one small change' and turn in broken code. It kills me whenever I see this. Don't kill me.

You can check to see if I will find your specification and feature comments by executing the following command from the command line. If you see your comments on the terminal, then I will see them. If not, I will NOT see them and you will NOT get credit for them. The following will check to see if you have commented your specifications:

```
grep -i 'specification' homework.cpp
```

This will generate the following output. Notice the specifications are numbered to match the specification number in the assignment. This is what I would expect to see for a 'C' Drake assignment. Note

the cd Desktop changes the file location to the desktop - which is where the source file is located.

```
calebfowler@ubuntu:~$ cd Desktop
calebfowler@ubuntu:~/Desktop$ grep -i 'specification' cDrake.cpp
    // Specification C2 - Declare Variables
    // Specification C3 - Separate calculation
    // Specification C1 - Program Output
calebfowler@ubuntu:~/Desktop$ 
```

This is what I would expect to see for an 'A' level Drake assignment.

```
calebfowler@ubuntu:~/Desktop$ grep -i 'specification' aDrake.cpp
{    // Specification C2 - Declare Variables
    // Specification C3 - Separate calculation
    // Specification B1 - Calculation
    // Specification C1 – Program Output
    // Specification B 2 - double and half
    // Specification A1 – Output Headers
    // Specification A2 – Display variables
calebfowler@ubuntu:~/Desktop$ 
```

We can also look at the line(s) after the grep statement. I do this to pay attention to code segments.

```
grep -i -C 1 'specification' aDrake.cpp
```

```
calebfowler@ubuntu:~/Desktop$ grep -i -C 1 'specification' aDrake.cpp
int main()
{    // Specification C2 - Declare Variables
    int r_starcreation = 7;              // rate of star creation
--

    // Specification C3 - Separate calculation
    float drake = 0;                      // initialize to 0
    // Specification B1 - Calculation
    drake =  r_starcreation * perc_starswithplanets * ave_numberofplanetslife *
perc_devlife * perc_devintlife * perc_comm *  exp_lifetime;

    // Specification C1 – Program Output
    cout << "The estimated number of potential alien civilizations in the univer
se is ";
--

    // Specification B 2 - double and half
    cout << "Half this value: " << drake * .5 << endl;
--

    // Specification A1 – Output Headers
    cout << endl;
--

    // Specification A2 – Display variables
    cout << "Variables:" << endl;
calebfowler@ubuntu:~/Desktop$ 
```

We can also use this to look for other sections of your code. The grep command searches for anything withing the single quotes ", and the -i option makes it case insensitive. This is how I will look for your program greeting:

```
calebfowler@ubuntu:~/Desktop$ grep -i -C 1 'greeting' aDrake.cpp

    // Program Greeting
    cout << "This program calculates and displays the number of potential";
calebfowler@ubuntu:~/Desktop$ 
```

The grep trick is extremely powerful. Use it often, especially right

before you turn in your code. This is the best way I can think of for you to be sure you met all the requirements of the assignment.

Client System

Your code must compile and run on the client's system. That will be Ubuntu Desktop Linux, version 18.04. Remember, sourcefile.cpp is YOUR program's name. I will type the following command to compile your code:

```
g++ -std=c++14 -g -Wall sourcefile.cpp
```

If you do not follow this standard it is likely I will detect errors you miss - and grade accordingly. If you choose to develop on another system there is a high likelihood your program will fail to compile. You have been warned.

Using the Work of Others

This is an individual assignment, you may use the Internet and your text to research it, but I expect you to work alone. You **may** discuss code and the assignment. Copying code from someone else and turning it in as your own is plagiarism. I also consider isomorphic homework to be plagiarism. You are ultimately responsible for your homework, regardless of who may have helped you on it.

Canvas has a built in plagiarism detector. You should strive to generate a green color box. If you submit it and the score is too high, delete it, change your code and resubmit. You are still subject to the due date, however. This does not apply if I have already graded your homework.

Often, you will not be able to change the code to lower the score. In this case, include as a comment with your homework, what you did and why you thought it was ineffective in lowering your score. This shows me something very important - you are paying attention to what you are doing and you are mindful of your plagiarism score.

ProTip: Get a bare bones copy of your code running and turn it in. Then go ahead and modify it with bonuses and whatnot. Upload it with the same name so it replaces your previous homework. This way, if something comes up or you can't finish your homework for some reason, you still have something turned in. A "C" is better than a zero. Risk management class, risk management.