**Preface**

Welcome to part one of your homework! For your homework, you will be building a video game, doing a piece of it each couple of weeks. At the end, you will have a fully functioning game that will be of a significant size (>1500 lines of code). For this piece, we will be doing the logistical parts, including loading data that we will need and handling save files. You will be provided a template with a few helper functions, as well as a skeleton that you will fill in with your own code. See below for more information.

Before we begin, make sure you…

1. **READ THIS ENTIRE DOCUMENT IN FULL FIRST**. There is a lot of information that will probably answer your questions!
2. Do not change any code that has been written for you, prototypes you are given, or file formats that you have been instructed to use. No exceptions. See later on for issues with prewritten code.
3. Pace yourself. This homework is bigger than any program you have ever dealt with, but it is very doable. The key is to start early and do a little every day. There are clearly laid out pieces to this homework, so it would be best to test your code a function or two at a time, as needed. Make sure to stay focused on just your pieces that you are implementing; it is easy to get lost in the size of it all. You don’t need to know anything about the implemented functions you are given to complete this homework, with the exception of findSaveFile().

You are allowed to hardcode any numerical values not passed as arguments or living in a file (definitions have been given at the top of the program). You are allowed to hardcode the filenames of ‘gamestate.txt’ and ‘Heroes.txt’, as needed. You may also create other functions as you like to help you debug and make things easier, but you are only responsible for implementing the functions described below. Only the libraries included for you are allowed. Pointers are required for all parts of the homework; doing this homework does not use static arrays.

We only have time for doing the logic pieces, so the homework will stick to text-based only.

**Overview**

The game will consist of a gauntlet. You will have three heroes that you will command in battle and fight through a series of unique bosses and see how far you get. Pick your heroes carefully; once you choose them, the choices cannot be undone until you lose the game or start a new game. The heroes will have base stats, and each will be equipped with a sword and/or a shield. The stats from the items combined with the base stats will dictate how well they do in battle. The boss will have a very simple AI consisting of a decision tree of actions based on its health.

At the end of this homework, you will have the following things:

* A main menu that is dynamic based on whether or not there is a saved game present
* The ability to start a new game, load a saved game, delete a saved game, or quit
* The ability to pick three heroes from a given list of SORTED heroes, save those heroes, and go back to the main menu
* A managed file system that can be loaded from and saved to, and is dynamic in its contents

**Function Prototypes**

**Functions that you will be implementing:**

**char\* getPlayerName();**

**Input: None**

**Return: The player’s chosen name**

The easiest function. Make a string, read the name the player gives into that string, and return it. No error checking is necessary for this function, but make sure to allocate enough space for the name according to the definition at the top of the program. You can also assume that player names won’t be duplicated across saved files, and the player name will not be “000” (this is reserved). Just make sure that when you are testing your code, keep to these assumptions.

**Hero\* pickHeroes(Hero\* heroes);**

**Input: a pointer to the list of heroes initially loaded**

**Return: a pointer to the chosen heroes copied over**

This function will handle the hero selection that the user will take part in. The list of heroes have been loaded, so the heroes will be sorted and displayed for the user to select. The user will select a hero one at a time, and each chosen hero will be copied into space handled by a separate pointer, which you will create and allocate here. You MUST copy the data over to a new pointer, otherwise freeing of memory will fail later. That second pointer will then be returned. The user cannot select the same hero twice, so make sure to handle invalid choices in this function.

**void sortHeroes(Hero\* heroes);**

**Input: a pointer to the list of loaded heroes**

**Return: None**

This function will sort the list of heroes alphabetically A-Z that will eventually be presented to the user. You may use selection or insertion sort to do this, but make sure that the list ends up in the same space that it started in (through the pointer passed to this function); do not use a separate array to sort them. Bubble sort is not allowed.

**void printHeroes(Hero\* heroes, int numHeroes)**

**Input:**

**heroes: a pointer to a list of heroes**

**numHeroes: the number of heroes that are being printed out**

**Return: None**

This function will print out the hero data for the list of heroes taken as a parameter. See the sample output for an example format. HINT: this function is primarily used for when loading a save file, but you can use it in pickHeroes too!

**void deleteFile(int choice, char\* filename);**

**Input:**

**choice: the chosen file slot that will be deleted in gamestate.txt**

**filename: the name of the file that will be deleted**

**Return: None**

This function will do two things. First, this function will delete the file that needs to be deleted. See the documentation for and use the built-in function remove() to do this (double-check when the program is done to make sure it is removed!).

Second, the file gamestate.txt needs to be updated to reflect that file being removed. Decrement the number of save files by one, and override the data for that file from the list of saved files. Be careful! You don’t want to override any of the other save files already present. Doing this might take some planning. Also note: the choice passed to you is based off of the output to the user (see sample output). This means that if the user chooses slot 2, modify the second file slot, not the third.

When you override the data, put this in its place:

000 | 0

**void saveFileData(int choice, char\* filename, Hero\* heroes, char\* playerName, int progress);**

**Input:**

**choice: the chosen save file slot**

**filename: the name of the save file to use that will hold the heroes**

**heroes: the array of chosen heroes**

**playerName: the name of the player playing the game**

**progress: a symbolic number that tells the function what is happening in the game**

**Return: None**

Easily the hardest function of the homework. This function will be modifying gamestate.txt, as well as the chosen save file. You will need to follow specific guidelines based on the value of progress.

The first thing you should do is write the playerName and progress value to gamestate.txt. If the value of progress is 1, you will override the values in gamestate.txt at the given file slot, and increment the number of save files by one. Notice that the save file choice isn’t always the next logical file choice (e.g. the slot choice is 3, but there is only one previous save file slot taken. In this case, you will need to modify the third slot only). If the value of progress is 2, there is already an entry in the gamestate.txt file for the given player name, which you will have to find and update (there will be no duplicates, and the active player name will never be “000”). You then have to find a way to update the progress value of that line only (you cannot jump to a specific line in the file or open the file for writing only, otherwise the file data will be erased!). BE VERY CAREFUL! If you destroy other save files or mess up the format, you will have to fix the file manually, and start over with your testing.

The second thing you have to do is write the heroes to the specific save file, if applicable. If the value of progress is 1, the heroes pointer will be NULL, and you should skip this step. If the value of progress is 2, there are selected heroes present, and you need to write those heroes to the save file given, using the format described later on. The saved heroes file should only exist beyond this point.

The value of progress will only be 1 or 2.

HINT: there is a function called loadGameStateData which reads in everything from gamestate.txt, and it has been written for you. It might be helpful to refer to this function to get ideas on how to preserve data.

NOTE: the save files for the heroes are created on the fly, as they are needed. The save files only exist for the file slots that have players who have selected heroes. Use the filename passed to this function.

**Hero\* loadSavedData(char\* filename); Hero\* loadInitialData();**

**Input: [varies]**

**Return: A list of heroes**

These two functions go together, as they are very similar. LoadSavedData will load saved chosen heroes from the save file which is passed into the function, using the file format given. LoadInitialData will load the list of heroes from Heroes.txt, the file with all of the possible hero choices. HINT: the only differences are the files you are dealing with, and the number of heroes you are dealing with.

**int main(void);**

Main will house the main menu logic for the game. You will first need to find out if there is a save file in place by using the findSaveFile() function:

* If a save file exists, the following options should be presented to the user:
  + Start a new game (startNewGame)
  + Load a saved game (loadSavedGameHandler)
  + Delete a saved file (DeleteSavedGameHandler)
  + Quit
* If a save file does not exist, there will be the following options:
  + Start a new game (startNewGame)
  + Quit

Use the appropriate function calls to handle all of these options (listed next to each option above). The menu should loop after the user choice has finished being handled. See below for details on findSaveFile().

**Functions that you will be calling:**

These functions, and any functions that go with them, have been completed for you, with the exception of the functions you are implementing. You only have to call these functions from main; do not modify their contents.

**FileFinder findSaveFile(); void DeleteSavedGameHandler();**

**void loadSavedGameHandler(); void startNewGame();**

The only thing to keep in mind is the return type of findSaveFile. If there is a saved file present, the function will return the enum value ‘EXISTS’. Otherwise, the value will be ‘DOESNOTEXIST’.

**File Formats**

(Note: the ‘…’ means that the pattern continues for however many there are. The ‘|’ are also in the file, so you have to read and write data to accommodate that!)

Heroes.txt (where the list of all heroes are):

<name of hero 1> | <hero health> | <hero defense> | <hero attack> | <hero speed>

<name of hero 2> | <hero health> | <hero defense> | <hero attack> | <hero speed>

…

file<num>.txt (the save files containing the chosen heroes for the player with that save slot):

<name of chosen hero 1> | <hero health> | <hero defense> | <hero attack> | <hero speed>

<name of chosen hero 2> | <hero health> | <hero defense> | <hero attack> | <hero speed>

<name of chosen hero 3> | <hero health> | <hero defense> | <hero attack> | <hero speed>

gamestate.txt:

<number of saved files>

<name of player in save 1> | < progress value of save 1>

<name of player in save 2> | < progress value of save 2>

…

**There must always be a number of save files and 5 file slots in gamestate.txt. Otherwise, the program might not work properly. Modify the text file if necessary to ensure this. The file also must exist.**

**Sample Output**

See the powerpoint slides on blackboard for pictures and notes.

**Grading Breakdown**

Your code must compile with no warnings and no errors. Failure to do so will result in a **ZERO**, no exceptions. Remember –Wall –Werror!

Remember gdb? It is your best friend in times like these! There’s no better time to learn how it works.

Any segmentation fault or other runtime error will result in a minimum deduction of 50%. This will be increased at your TA’s discretion. Make sure you error check everything!

Some mallocing might be required to complete a function implementation. If you are returning any pointers or using pointers passed as a parameter, you are not responsible for freeing them. However, any pointers that stay local to a function you are required to manage when it comes to their allocated memory.

If you think you found a bug in the code given to you, let Daniel know ASAP. If there is a bug and it is severe enough, you will get extra credit for it! Any clarifications regarding this document should also be directed to Daniel.

15 points: Sorting the heroes

* The heroes are sorted alphabetically A-Z
* The sorting was done using an acceptable method

20 points: Hero selection

* Three, and only three, heroes can be selected from the imported list of hero options
* Appropriate error checking is in place given the rules stated in the function description
* Acceptable display of the hero options and feedback to the user are shown

15 points: File management

* The save files are created/destroyed properly and dynamically, and follow the correct format
* gamestate.txt follows the correct format, and correctly reflects what save files exist and what progress value they have
* Heroes are correctly read from save files and the initial file.
* Error checking is in place to give feedback in the event of something failing

10 points: Main menu

* The menu has different options depending on if there is a save file or not
* All menu options work correctly regardless of the number of options
* The program correctly loops back to the beginning once the selection is done
* Appropriate error checking is in place preventing any invalid numerical inputs

(Continued on the next page)

Failure to follow these guidelines will result in deductions (At your TA’s discretion):

* All prewritten code, prototypes and files given to you are unmodified (any tampering with prewritten code without Daniel’s consent = major point deductions). If you do not follow the prototypes you are told to use, you will receive a zero.
* Code is formatted in an acceptable and readable manner
* Comments are present explaining logical reasoning (HINT: more meaningful/thoughtful comments = more likely to get partial credit)
* Appropriate output is given for the user for any invalid inputs

Any additions to make the game your own are welcome! Make sure not to add too much in case you will have to remove it to complete HW 2 and 3, but feel free to add flavor text to add to the game’s feel