CITS3007 Project 2023

Contents

Version:	0.2
Date:	28 Sept, 2022

1 Introduction

- This project contributes **30%** towards your final mark this semester, and is to be completed as individual work.
- The project is marked out of 65.
- The deadline for this assignment is **5:00 pm Thu 12 Oct**.
- You are expected to have read and understood the University Guidelines on Academic Conduct. In accordance with this policy, you may discuss with other students the general

principles required to understand this project, but the work you submit must be the result of your own effort.
- You must submit your project before the submission deadline above. There are significant penalties for late submission (click the link for details).

1.1 Project submission

Submission of the project is via the CSSE Moodle server.

- A "testing sandbox" Moodle area will be made available within 1 week of the specification being released, which will provide you with some (minimal) feedback and information you can use while developing and testing your project submission. It will not include space for "coding style and clarity" marks.
- A "final submission" Moodle area will be made available no less than 1 week prior to the project being due, where you can submit final code and answers to questions. It will include only minimal tests of code, and will include questions that do not require code or an answer to be submitted, but will be used by markers when assessing coding style and clarity.

1.2 Clarifications and changes to the project specification

You are encouraged to start reading through this project specification and planning your work as soon as it is released. Any queries regarding the project should be posted to the Help3007 forum with the "project" tag.

Any clarifications or amendments that need to be made will be posted by teaching staff on the Help3007 forum.

For an explanation of the process for publication and amendment of the project specification, see the CITS3007 "Frequently Asked Questions" site, under "How are problems with the project specification resolved?".

2 Background

Your software development team at WotW, Inc. ("Warlocks of the Waterfront") is developing a new online, virtual-reality, multiplayer role-playing game, "Pitchforks and Poltergeists" (P&P, for short), which will be added to WotW's catalog of popular games.

The game is being developed in C, and you have been tasked with implementing several important parts of the game.

Because the game will be played online, security is an important concern to the company.

You will need to implement C functions for several tasks, detailed below.

A header file, `p_and_p.h`, is provided which defines several datatypes used in the game. Several of these are described briefly below.

- `Characters` in the game have a character ID (a 64-bit unsigned integer) as well as various other characteristics, and also possess an inventory of items.
- The inventory consists of an array of `ItemCarried` structs. These have two fields: `itemID` (a 64-bit unsigned integer), and `quantity` (a `size_t`).

- Only the first `inventorySize` many elements of the `inventory` field are actually in use at any time. It is undefined what the other elements contain, and they are not considered to be part of the character's inventory.
- A character can never carry more than a total of `MAX_ITEMS` items. In other words, the sum of the `quantity` field in the `ItemCarried` structs for a character's inventory must not exceed `MAX_ITEMS`. Any `Character` struct which specifies that a character holds more than `MAX_ITEMS` items is considered *invalid*.
- The `itemID` in an `ItemCarried` refers to unique *class* of items – e.g. pitchfork, crucifix, copy of the Bible, etc. The data about each such class is stored in an `ItemDetails` struct, which contains a string name and description.

You are also provided with a `p_and_p.c` file which contains (currently empty) definitions for the required functions, but you need not use this specific file if you don't wish to – you need only submit a `.c` file that contains definitions for all the required functions.

3 File formats

Several binary file formats are used by the Pitchforks and Poltergeists game:

- `ItemDetails` file format
- `Character` file format

For convenience, we define two sorts of string field:

name field

> The characters contained in a name field must have a graphical representation (as defined by the C function `isgraph`). No other characters are permitted. This means that names cannot contain (for instance) whitespace or control characters.
>
> A name field is always a `DEFAULT_BUFFER_SIZE` block of bytes. The block contains a `NUL`-terminated string of length at most `DEFAULT_BUFFER_SIZE-1`. It is undefined what characters are in the block after the first `NUL` byte.

multi-word field

> A multi-word field may contain all the characters in a name field, and may also contain space characters (but the first and last characters must not be spaces).
>
> A multi-word field is always a `DEFAULT_BUFFER_SIZE` block of bytes. The block contains a `NUL`-terminated string of length at most `DEFAULT_BUFFER_SIZE-1`. It is undefined what characters are in the block after the first `NUL` byte.

Both file types store integer values in "little-endian" order.

3.1 `ItemDetails` file format

Information from `ItemDetails` structs are normally stored in a file called "`itemdets.dat`". This file type has the following structure:

1. File Header

The file begins with a header that contains metadata about the saved data. It contains one datum:

- Number of items: a 64-bit unsigned integer indicating the number of `ItemDetails` structs that follow in the file.

2. ItemDetails data:

- Following the file header, there are multiple blocks of data, each representing an `ItemDetails` struct.
- Each `ItemDetails` block consists of:
  - `itemID`: An 64-bit unsigned integer representing the item's unique identifier.
  - `itemName` (char array): A block of characters of size `DEFAULT_BUFFER_SIZE` (i.e., 512 bytes) containing the item's name. This is a *name field*.
  - `itemDesc` (char array): A character array of size `DEFAULT_BUFFER_SIZE` containing the item's description. This is a *multi-word field*.

3.2 `Character` file format

The character file format is a binary format designed to store an array of `Character` structs along with their associated inventory of `ItemCarried` structs. Character files are typically named "`characters.dat`". The format consists of the following components:

1. Header Information:

- Size of the `Character` array: A 64-bit, unsigned integer value indicating the number of `Character` structs stored in the file. This is the total number of characters to be loaded.

2. Character Records:

For each `Character`, the file contains the following:

- **Character Fields:**
  - `characterID`: A 64-bit, unsigned integer value representing the unique identifier of the character.
  - `socialClass`: An 8-bit, unsigned integer representing the character's social class. Each value (from 0 to 4) specifies one of the enumerated members of the `CharacterSocialClass` enum.
  - `profession`: A block of characters of length `DEFAULT_BUFFER_SIZE`, containing the character's profession. This is a *name field*.
  - `name`: A block of characters of length `DEFAULT_BUFFER_SIZE`, containing the character's name. This is a *multi-word field*.
- **Inventory Size:**
  - `inventorySize`: A 64-bit, unsigned integer value indicating the number of items carried by the character.
- **Inventory Items:**
  - This consists of `inventorySize` many blocks of data.
  - In each block, the file contains:
    - `itemID`: A 64-bit, unsigned integer value representing the unique identifier of the item class.

    * `quantity`: A 64-bit, unsigned integer value indicating the quantity of the item carried by the character.

Notes on the character file format

- Although the `inventory` field of each `Character` struct always contains `MAX_ITEMS` elements, only the used portion of the inventory (that is, `inventorySize` many elements) is written to (or read from) a character file.

4 Tasks

You should complete the following tasks and submit your completed work using Moodle.

4.1 ItemDetails format "load" and "save" functions (20 marks)

You are required to implement the functions to load and save data in the `ItemDetails` format, as follows:

`saveItemDetails`

This function has the prototype

```
int saveItemDetails(const struct ItemDetails* arr, size_t nmemb, int fd)
```

and serializes an array of `ItemDetails` structs. It should store the array using the `ItemDetails` file format.

If an error occurs in the serialization process, the function should return a 1. Otherwise it should return 0.

`loadItemDetails`

This function has the prototype

```
int loadItemDetails(struct ItemDetails** ptr, size_t* nmemb, int fd)
```

It takes as argument the address of a pointer-to-`ItemDetails` struct, and the address of a `size_t`, which on successful deserialization will be written to by the function, and a file descriptor for the file being deserialized.

If deserialization is successful, the function will:

- allocate enough memory to store the number of records contained in the file, and write the address of that memory to `ptr`. The memory is to be freed by the caller.
- write all records contained in the file into the allocated memory.
- write the number of records to `nmemb`.

If an error occurs in the serialization process, the function should return a 1, and no net memory should be allocated (that is – any allocated memory should be freed). Otherwise, the function should return 0.

Up to 10 marks are awarded for a successful implementation of these functions which can load and save files. In order to obtain these marks, the functions should be able to successfully load

and save *valid* files and structs, but need not behave correctly if the files or structs are invalid. (Validation is a separate task; see section 4.2 "Validation functions".) 10 further marks are awarded for coding style and quality of the implementation.

4.2 Validation functions (16 marks)

Implement the following validation functions:

`isValidName` This function has the prototype

```
int isValidName(const char *str)
```

and checks whether a string constitutes a valid *name field*. It returns 1 if so, and 0 if not.

`isValidMultiword` This function has the prototype

```
int isValidMultiword(const char *str)
```

and checks whether a string constitutes a valid *multi-word field*. It returns 1 if so, and 0 if not.

`isValidItemDetails` This function has the prototype

```
int isValidItemDetails(const struct ItemDetails *id)
```

and checks whether an `ItemDetails` struct is valid – it is valid iff all of its fields are valid (as described in the documentation for the struct and elsewhere in this project specification). The `name` and `desc` fields must be valid *name* and *multi-word* fields, respectively; they also must not be empty strings. This function returns 1 if the struct is valid, and 0 if not.

`isValidCharacter` This function has the prototype

```
int isValidCharacter(const struct Character *c)
```

and checks whether a `Character` struct is valid – it is valid iff all of its fields are valid (as described in the documentation for the struct and elsewhere in this project specification).

The following are all necessary preconditions for validity of the struct:

- the `profession` field must be a valid *name* field, and must not be the empty string;
- the `name` field must be a valid *multi-word* field, and must not be the empty string;
- the total number of items carried (that is: the sum of the `quantity` fields of the `ItemCarried` structs that form part of the inventory) must not exceed `MAX_ITEMS`; and
- `inventorySize` must be less than or equal to `MAX_ITEMS`.

This function returns 1 if the struct is valid, and 0 if not.

Up to 8 marks are awarded for a correct implementation of these functions. Up to 8 further marks are awarded for coding style and quality of the implementation.

Once your validation functions are complete, you should incorporate them into `loadItemDetails` and `saveItemDetails` where applicable, and those functions should return an error if they encounter an invalid struct or file record.

4.3 Character format "load" and "save" functions (12 marks)

Implement functions to load and save in the `Character` file format.

You should implement the following two functions:

`saveCharacters` This function has prototype

```
void saveCharacters(struct Character *arr, size_t nmemb, int fd)
```

`loadCharacters` This function has prototype

```
void loadCharacters(struct Character** ptr, size_t* nmemb, int fd)
```

The two functions load and save in the `Character` file format, and should validate records using the `isValidCharacter` function, but otherwise behave in the same way as `saveItemDetails` and `loadItemDetails`.

Up to 6 marks are awarded for a correct implementation of these functions. Up to 6 further marks are awarded for coding style and quality of the implementation.

4.4 "Load" and "save" secure coding practices (7 marks)

Up to 7 marks are awarded for following secure coding practices in the "load" and "save" functions for `ItemDetails` and `Characters`. This includes (but is not limited to) correctly incorporating the validation functions into the "load" and "save" functions.

4.5 Secure load function (10 marks)

In a working implementation of the game, the `ItemDetails` database is stored in a file owned by a user with username and primary group `pitchpoltadmin`. The executable for the game is a setuid and setgid program, owned by that user and that group.

When the executable starts running, it does the following things (which you need not implement):

- Temporarily drops privileges, then loads and saves files owned by the user who invoked the executable.
- Calls a function `secureLoad`, implemented by you; this acquires appropriate permissions, loads the `ItemDetails` database, and then permanently drops permissions.

You must implement the `secureLoad` function. It has prototype

```
int secureLoad(const char *filepath)
```

It should attempt to acquire appropriate permissions for opening the `ItemDetails` database (that is: the effective userID should be set to the userID of `pitchpoltadmin`), should load the database from the specified file, and then (after permanently dropping privileges), call the function

```
void playGame(struct ItemDetails* ptr, size_t nmemb)
```

to which it should pass the loaded data and the number of items in the loaded data. If an error occurs during the deserialization process, the function should return 1. It should check the running process's permissions to ensure that the executable it was launched from was indeed a setUID executable owned by user `pitchpoltadmin`. If that is not the case, or if an error occurs

in acquiring or dropping permissions, the function should return 2. In all other cases, it should return 0.

It should follow all best practices for a setUID program.

Up to 5 marks are awarded for correct implementation of the function, and 5 marks for style and quality of the implementation.

5 Annexes

5.1 Marking rubric

Submissions will be assessed using the standard CITS3007 marking rubrics.

Except where otherwise noted, questions requiring long English answers are marked as per the standard long answer rubric (see https://cits3007.github.io/faq/#marking-rubric).

Questions requiring code will have marks allocated for *correctness*, and for *style and clarity*.

5.2 Code requirements

Note that, as per the standard rubric, your code must compile without errors using gcc in the standard CITS3007 development environment. If it fails to compile, teaching staff will not fix it for you, and you are likely to receive only minimal marks for implementation of functions (though it is still possible to receive marks for style and clarity of code, for those portions of the project you've completed).

If, at the time of submission, some portion of your code is not compiling, you should either comment it out, or surround it with "`#if 0 ... #endif`" preprocessor instructions, so that your code does compile.

Any `.c` files submitted should `#include` the `p_and_p.h` as follows:

```
#include <p_and_p.h>
```

Your code must contain the functions required by this specification, and they must exactly match the prototypes in the `p_and_p.h` file, but you may also write whatever "helper functions" you wish.

Your code must not include a "`main()`", nor a "`playGame()`" function. If it does, your code will fail to compile when automated tests are run on it, and you are likely to receive minimal marks for implementation.

You may `#include` any header files that you need to, as long as they are available in the standard CITS3007 development environment, and may use non-standard C functions as long as they are available in that environment.

Although you are strongly encouraged to test your code using the "check" testing framework, you should not submit your test code, and it is not assessed.

5.3 Coding style

All code submitted should comply with the coding guidelines listed at https://cits3007.github.io/faq/#marking-rubric.

All C code written should:

a. adhere to secure coding best practices, and
b. be properly documented.

The documentation requirement means that, at minimum, all functions should have a comment just above them describing what the function does; functions forming part of the API (and any particularly important internal functions) should have a *documentation block* parseable by Doxygen. When writing documentation, you may assume

a. that readers have read the two file format specifications and the `p_and_p.h` header file, so you need not repeat information from them;
b. that any function with a prototype in the `p_and_p.h` header file forms part of the API; and
c. that if you are unable to complete all the project by the time of submission, there is no need to write documentation for functions you haven't completed.

Except for documentation blocks, all comments should be written as single-line ("`//`") comments – do not use multiline ("`/* .. */`") comments.

As part of keeping your code readable, lines should generally be kept to less than 100 characters long.

Code should handle errors gracefully when reading or writing files – such errors include file open failures, insufficient memory, and file corruption.

Additionally, note that your code should be considered "library" code – it should not be interacting directly with a user or the terminal (you may assume that other developers are working on the user-facing parts of the project). This means your code should:

- never print to standard out or standard error (unless the specification states otherwise); and
- never exit or abort, but instead return with an error value, unless the specification states otherwise.

If your code prints to standard out or standard error, it is likely to fail the automated tests used to mark portions of your code. If that happens, those portions of your code are likely to receive a mark of 0. Teaching staff will not fix your code to remove statements that print to standard out or standard error.