Curtin University — Computing Discipline

UNIX and C Programming (COMP1000)

Semester 1, 2019

Assignment

Due: Monday 20 May at 8.00am

Weight: 20% of the unit mark.

1 Introduction

Your task for this assignment is to design, code (in C89), test and debug a treasure hunter program.

In short, your program will:

- Read in a 2 dimensional map from a file;
- Read a series of movements from a file depicting the journey through the map;
- Process each movement, keeping track of any treasures that were discovered;
- For each treasure that is found, record the co-ordinate and details in a log file;
- Finally display the total value of the collected items to the user.

There is a lot of detail here. Read it *carefully* before you start anything.

2 Documentation

You must thoroughly document your code using C comments (/* ... */). For each function you define and each datatype you declare (e.g. using struct or typedef), place a comment immediately above it explaining its purpose, how it works, and how it relates to other functions and types. Collectively, these comments should explain your design. (They are worth substantial marks - see Section 7.)

3 Task Details

Read this section carefully, there is a lot of detail, you may need to read it several times to ensure you do not miss anything. It is highly recommended that you spend some time coming up with a design and manually testing it before you start coding.

Input files

Your program should accept two command-line parameters: the name of the file containing the map as the first parameter, and the adventure instructions in the second file. For example:

```
[user@pc]$ ./TreasureHunter map.csv adventure.csv
```

The Map

The first line in the map file specifies the number of rows and columns in the map. Each row there after depicts one row of the map, as a csv, where each comma separator is a differenent column.

There are three different types of treasure that may appear on the map.

coins - these are just coins that have been found lying around, they have a value and are represented as follows:
`C <value>`

gear - you found a piece of gear (equipment), gear has a potentially multi-word description/detail, occupies an equipment location ("slot") on the explorers body and has a value. The possible slots a piece of gear can occupy are:

- head;
- chest;
- legs;
- hands.

Gear is represented as follows:
`G <detail>:<slot>:<value>`

magic - you found a magic item, magic items have a potentially multi-word description/detail and a value, magic items are represented as follows:
`M <detail>:<value>`

The determining character (C, G, M) may be upper or lower case and maps may contain a combination of both. This is valid, your program should process `C 250` exactly the same as `c 250`. The same applies for gear slot names, that is `chest`, `CHEST`, and `cHEst`, along with any other capitalisation alternatives all represent the same slot.

For example the following file represents a map with 5 rows and 4 columns, in total this example map has 6 treasures. The actual maps your program will be tested on may be of any size and have any number of treasures.

```
5,4
,,C 200,
,G Vibranium Shield:hands:990,,C 50
M Healing Potion:85,,M Defence Enchantment:360,
,,,
,,G Lightsaber:hands:850,
```

This input file would translate to the following map grid.

		item: coins **value:** 200	
	item: gear **detail:** Vibranium Shield **slot:** hands **value:** 990		**item:** coins **value:** 50
item: magic **detail:** Healing Potion **value:** 85	**item:** magic **detail:** Defence Enchantment **value:** 360		
		item: gear **detail:** Lightsaber **slot:** hands **value:** 850	

Your program should read the contents of this file into a dynamically allocated two-dimensional array. If your program arbitrarily allocates more memory than required to store the map then you will be penalised.

The Adventure

The second input file depicts the route the adventurer takes when exploring the map, this file will consist of a list of entries in the following format.
`<direction> <distance>`

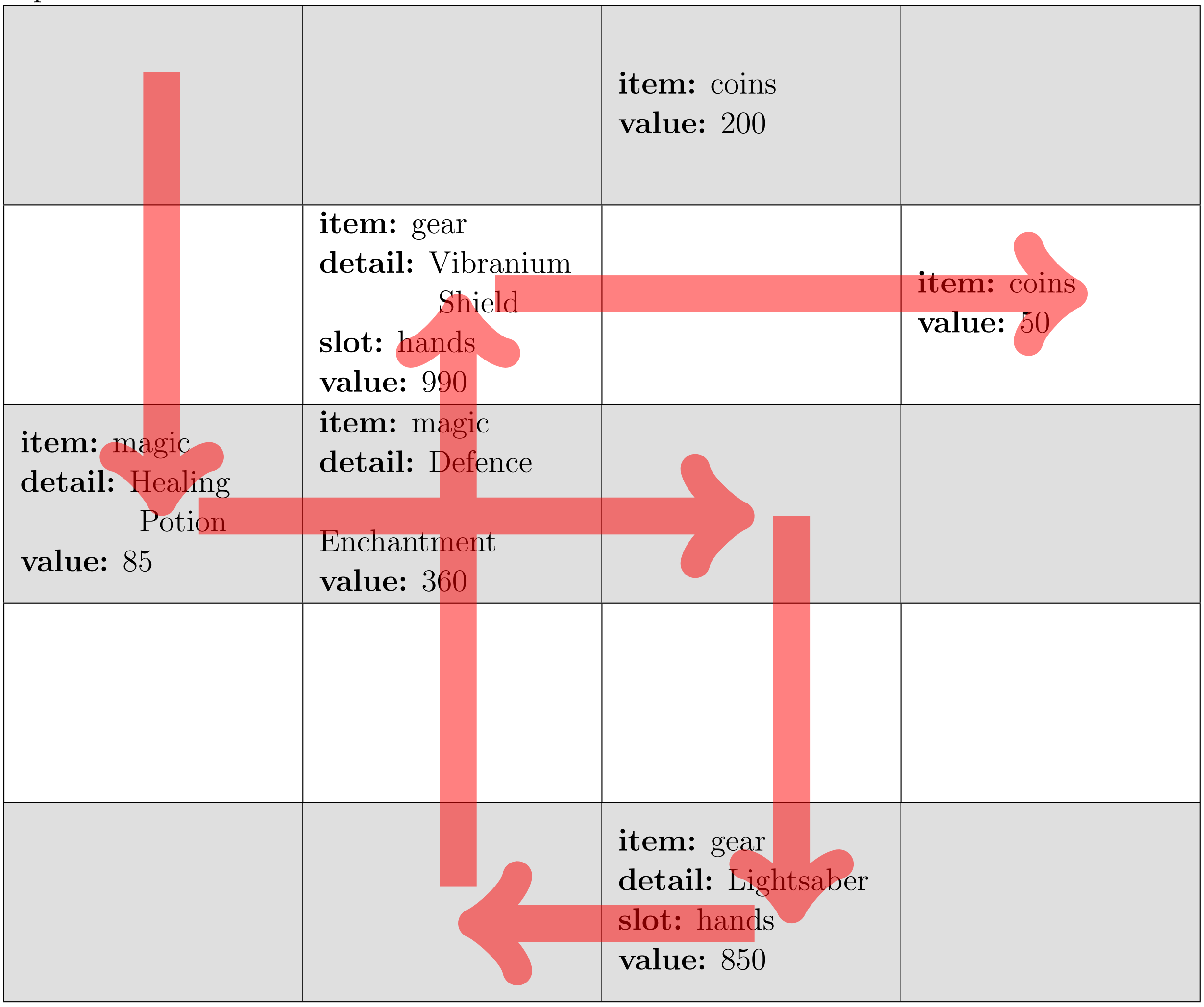
The possible directions are:

- UP
- DOWN
- LEFT
- RIGHT

As before any possible combination fo casing is acceptable for the direction. The distance is an **integer number** specifying how many rows/columns should be moved in that direction. An adventure always starts from the upper left hand corner of the map. An example of the adventure file with, 6 instructions is provided below.

```
DOWN 2
RIGHT 2
DOWN 2
LEFT 1
UP 3
RIGHT 2
```

When applied to the previous example map this adventure would result in the following exploration.



The adventure file may be of any length, its contents **must** be read and stored in a **generic linked list** before the adventure is resolved.

Adventure Resolution

The adventure is resolved by sequentially apply the move commands within the adventure file. This means you must keep track of the current position of the explorer and ensure they do not go outside of the map, that is, if a final move of “`UP 3`” was appended to the previous adventure example then the explorer would be trying to go to row -2 which is outside of the map. Details on how to deal with this situation are provided in the conditional compilation section.

Exploration always starts in the top left hand corner, what would be cell (0,0) in a 2d-array. Each move commands sends the explorers in the given direction for the given number of cells, that is a `<move> 0` would result in no movement regardless of the direction of the move command (you do not need to worry about what direction the explorer is "facing").

Use the treasure collection example below to see how an adventure should resolve.

Treasure Collection

A treasure is collected whenever the explorer enters a cell on the map where a treasure is located. This means that any treasure in the top left corner of the map is always collected as that is the starting location.

A treasure can only be collection once, regardless of how many times the cell containing the treasure is explored. **Hint:** removing treasures as you collect them is easier then trying to remember where you have been.

Coins and magic items are collected and kept, there is no limit on the number of these items that can be kept.

Gear is worn by the explorer, so an explorer can only carry one piece of gear for each equipment slot (4 pieces of gear in total). If a piece of gear is encountered but its slot already occupied then the explorer keeps the piece of gear with the highest value. The lower value piece of gear can be dropped or destroyed — it does not matter as there is only one player. However if your choice of resolution results in any memory issues (eg: memory leaks) then you will be penalised.

Given the previous example, the final treasures collected (and kept) would be, 50 coins, vibranium shield, healing potion, and defence enchantment. The exploration would occur in the following sequence

DOWN 2 - Collect Healing Potion.
RIGHT 2 - Collect Defence Enchantment
DOWN 2 - Collect Lightsaber (equip to hands)
LEFT 1 - *nothing*
UP 3 - Defence Enchantment has already been collected so nothing happens (note that this cell is encountered *first*),
Collect and equip Vibranium Shield, discard Lightsaber as it has the lower value.
RIGHT 2 – Collect 50 coins

The Mandatory Function Pointer

When creating the map, each treasure item must be stored with a function pointer where the function that it points to compares the specific gear slot that the function occupies

This means that you will need 4 comparison functions, one for each piece of gear. For example:

- compareHead
- compareChest
- compareLegs
- compareHands

The parameters/return type for the function will depend greatly on your design. The function should take in all the equipped gear and itself decide which slot it is comparing. That is, do not have an if/case statement outside of the function to decide which piece of equipped gear should be passed in.

For example, when a gear item for the head slot is added to the map, its function pointer should point to the `compareHead` function. This function pointer must then be used to call the appropriate comparison function when a piece of gear for an already equipped slot is encountered.

The purpose of this function/function pointer is to decide if the new piece of gear should be kept and the old piece discarded or vice versa. This function may do the exchanging/memory management itself, or may return a flag specifying if a change should occur, the choice is up to you (though the former is a better design decision).

Note: Yes there are easier, more elegant ways of doing this (enums for example) and this is an overly complicated approach for a simple task. But we need to assess your ability to use function pointers. A more realistic usage of function pointers would require a far more complicated assignment.

So write the functions, have the function pointer, call the function pointer and get the marks. Do not over think it, there is no complex reason for this function pointer.

Error Handling

You must deal with any invalid files, this includes, but is not limited to:

- The file not existing;
- The file being empty;
- Lines that do not conform to the specified structure;
- The incorrect format of a treasure entry within a line, or incorrect values for any of the treasures details;
- The file not matching the specified number of rows and columns;
- Invalid move commands;
- Move commands that result in exploration exceeding the map dimensions (see the conditional complication section for amendments to this).

When an invalid file is encountered then an appropriate, meaningful error message should be displayed to the user on stderr.

Log file (Consider this when developing test cases!)

Your program must also *append* to (not overwrite) a log file called `adventure.log`.

The first line written to the log file, after its existing contents, should be "`---`". This will serve to separate the new contents from the old.

The log file should record the coordinates and details of each treasure collected/removed during the course of the adventure, the treasures should be recorded in the same order in which they are encountered.

Given the example map/adventure used previously. The following log file would be generated.

```
---
COLLECT<ITEM:MAGIC, XLOC:0, YLOC:2, DESCRIPTION:Healing Potion, VALUE:85>
COLLECT<ITEM:MAGIC, XLOC:1, YLOC:2, DESCRIPTION:Defence Enchantment, VALUE:360>
COLLECT<ITEM:GEAR, XLOC:2, YLOC:4, DESCRIPTION:Lightsaber, SLOT:hands, VALUE:850>
COLLECT<ITEM:GEAR, XLOC:1, YLOC:1, DESCRIPTION:Vibranium Shield, SLOT:hands, VALUE:990>
DISCARD<ITEM:GEAR, XLOC:1, YLOC:1, DESCRIPTION:Lightsaber, SLOT:hands, VALUE:850>
COLLECT<ITEM:COINS, XLOC:3, YLOC:1, VALUE:50>
```

If the program was run a second time with the same map/adventure then the contents of the log file would be (see the provided log file for an example without a page width limit):

```
– – –
COLLECT<ITEM:MAGIC, XLOC:0, YLOC:2, DESCRIPTION:Healing Potion, VALUE:85>
COLLECT<ITEM:MAGIC, XLOC:1, YLOC:2, DESCRIPTION:Defence Enchantment, VALUE:360>
COLLECT<ITEM:GEAR, XLOC:2, YLOC:4, DESCRIPTION:Lightsaber, SLOT:hands, VALUE:850>
COLLECT<ITEM:GEAR, XLOC:1, YLOC:1, DESCRIPTION:Vibranium Shield, SLOT:hands, VALUE:990>
DISCARD<ITEM:GEAR, XLOC:1, YLOC:1, DESCRIPTION:Lightsaber, SLOT:hands, VALUE:850>
COLLECT<ITEM:COINS, XLOC:3, YLOC:1, VALUE:50 >
– – –
COLLECT<ITEM:MAGIC, XLOC:0, YLOC:2, DESCRIPTION:Healing Potion, VALUE:85>
COLLECT<ITEM:MAGIC, XLOC:1, YLOC:2, DESCRIPTION:Defence Enchantment, VALUE:360>
COLLECT<ITEM:GEAR, XLOC:2, YLOC:4, DESCRIPTION:Lightsaber, SLOT:hands, VALUE:850>
COLLECT<ITEM:GEAR, XLOC:1, YLOC:1, DESCRIPTION:Vibranium Shield, SLOT:hands, VALUE:990>
DISCARD<ITEM:GEAR, XLOC:1, YLOC:1, DESCRIPTION:Lightsaber, SLOT:hands, VALUE:850>
COLLECT<ITEM:COINS, XLOC:3, YLOC:1, VALUE:50>
```

Terminal Output

At the end of the adventure a summary should be displayed to the user.

STATUS: The result of the adventure

- **COMPLETE** - everything went smoothly, there were no issues with the map or the adventure.
- **CORRECTED** - The adventure had some issues but you managed to correct them and continue to the end (see the Makefile Targets section for information on this status).
- **FAILED** - The map was acceptable but the adventure itself failed somehow. None of the other summary data should follow this status.
- **ABORTED** - The map could not be processed so the adventure never took place. None of the other summary data should follow this status.

COINS: The total value of the coins collected.
MAGIC: The total value of the magic items collected.
GEAR: The total value of the gear collected (and kept).

Given the example adventure we have been on the following output summary would be provided. note that the value of the gear is only 1500 as the lightsaber was discarded and not counted.

```
STATUS: COMPLETE
COINS: 50
MAGIC: 445
GEAR: 990
```

Makefile Targets

You must provide a makefile for your assignment.

Warning: Failure to provide a makefile will result in zero (0) marks being awarded for compilation/execution/test cases. ie: a maximum of 40% of the available marks will be awarded.

TreasureHunter – The compiled assignment as described above. If the move commands exceed the map boundaries then the adventure should be stopped and the status set as FAILED.

TreasureHunterAI – When this version of the assignment exceeds the map boundaries then the move distance should be changed to go to the edge of, but not past, the map. For example, if a move command would change the y coordinate to -2 it should instead be set to 0, and the adventure continued. If this type of correction does occur then the status should be CORRECTED.

TreasureHunterLog – This version of the assignment should print the log file entries to the terminal as well as the file. The log file entries should be printed as each treasure is encountered, that is, if the map and adventure were large enough that they took several hours to process the user would be able to see a real time summary of the adventure.

You must use conditional compilation (preprocessor macros) to achieve the different functionalities, ie: the compiled code for each version must be *different*. If you achieve these difference with normal C code then only the basic version will receive marks,

4 Report

You must prepare a report that outlines your design and testing. Specifically:

1. For each function you write, describe its purpose (in a paragraph).
2. On how you converted the input file to a coordinate system:
   - Describe (in two or three paragraphs) how you implemented this.
   - Describe (in one or two paragraphs) an alternative approach that would achieve the same outcome.
3. Demonstrate that your program works by showing sample input and output, including:
   - The command-line used to execute your program.
   - The user input provided via the terminal.
   - The contents of the input file.
   - The output, as shown on the screen (and written to the output file).

Your report should be professionally presented, with appropriate headings, page numbers and a contents page. You will loose marks for poorly written/badly formatted reports!

5 Submission

Warning: If you submit code that does not compile or you do not have a valid, functioning makefile, then you will not be able to get a mark above 40% for the assignment.

Submit a single .tar.gz file containing all your files (projects, report, etc.).

Submit your entire assignment electronically, via Blackboard (`http://lms.curtin.edu.au`), before the deadline.

Submit one .tar.gz file containing:

A declaration of originality – whether scanned or photographed or filled-in electronically, but in any case *complete*.
Your implementation – including all your source code (.c files, .h files, makefile, etc. . .).
Your report – a single PDF file containing everything mentioned in Section 4.

Note: do not use .zip, .zipx, .rar, .7z, etc, they will be taken as **non-submissions** and instantly receive zero (0) for the whole assignment.

You are responsible for ensuring that your submission is correct and not corrupted. You may make multiple submissions, but only your latest submission will be marked.

No extensions will be granted. If exceptional circumstances prevent you from submitting an assignment on time, contact the Unit Coordinator ASAP with relevant documentation. After the due date and time is too late.

Late submission policy, as per the Unit Outline, will be applied.

6 Demonstration

You will be required to demonstrate each of the worksheet submissions in your practical session. The final assignment submission will also be demonstrated in the practical session in the week starting the 20th of May. If you cannot attend your registered practical session you must make arrangements with the lecturer by the 13th of May. You may be asked several questions about your assignment in this demonstration.

Failure to demonstrate the assignment during your registered practical will result in a mark of ZERO (0) for the entire assignment. During the demonstration you will be required to answer questions about your design. If you cannot answer the questions satisfactorily, you will receive a mark of zero for the assignment.

7 Mark Allocation

Every valid assignment will be initially awarded 100 marks. Marks will then be deducted for the following:

50 marks — Coding
If your assignment does not compile or lacks a makefile then you will receive zero marks for this section.

15 marks — Commenting.
You will not lose any marks if you have provided good, meaningful explanations of all the files, functions and data structures needed for your implementation.

15 marks Coding practices and coding standard.
You will not lose any marks if you have followed the coding standard and good coding practices (see Blackboard, resources section), and your code is well-structured, including being separated into various, appropriate .c and .h files.

20 marks — Report.
You will not lose any marks if your report is complete and professionally presented.

Mark Limit — Working Product.
You will not lose any marks if your program compiles, runs and performs the required tasks without unexpected error. The marker will use test data, representative of all likely scenarios, to verify this. You will not have access to the marker's test data. Your assignment will be marked according to the above categories, and then a multiplier applied as follows:

40%-100% — Depending on if your code compiles, and the quality of your makefile (including multi-targets).

60%-100% — Depending on how much functionality you implemented and have working.

80%-100% — Depending on how much error checking/handling you include.

60%-100%: — How well you manage your memory (ie: do you have memory leaks or errors)?

For each category you will receive a maximum percent, the **lowest** maximum will then be applied as a multiplier.

Warning: While these limits apply, do not be deceived. If you only implemented part of the functionality then you are likely missing several key parts of code (or other categories) and will not be able to get full marks in those sections.

Signoffs:

Once your mark has been calculated, the practical signoff result will be applied as follows:

Your total practical signoffs will be awarded a mark out of 10, which will have the following effect (multiplier) on the assignment mark.

Total	Effect
8 - 10	100%
7 - 7.9	90%
6 - 6.9	80%
5 - 5.9	70%
3 - 4.9	50%
0 - 2.9	30%

8 Academic Misconduct – Plagiarism and Collusion

This is an assessable task, and as such there are strict rules. You must not ask for or accept help from *anyone* else on completing the tasks. You must not show your work to another student enrolled in this unit who might gain unfair advantage from it.

These things are considered **plagiarism** or **collusion**.

Staff can provide assistance in helping you understand the unit material in general, but nobody is allowed to help you solve the specific problems posed in this document. The purpose of the assignment is for *you* to solve them *on your own*.

Please see Curtin's Academic Integrity website for information on academic misconduct (which includes plagiarism and collusion).

The unit coordinator may require you to provide an oral justification of, or to answer questions about, any piece of written work submitted in this unit. Your response(s) may be referred to as evidence in an Academic Misconduct inquiry. In addition, your assignment submission may be analysed by Turnitin and/or other systems to detect plagiarism and/or collusion.

End of Assignment