UCP Report

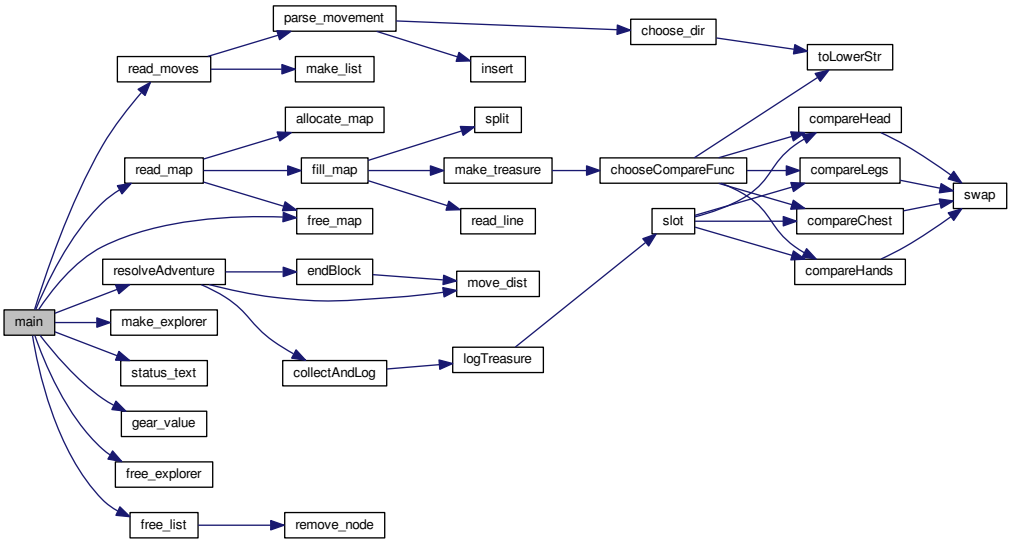
Jakob Wyatt

May 27, 2019

Contents

1 Purpose of Functions

This is the call graph of the program:



1.1 resolveAdventure

```
status resolveAdventure(map items, unsigned long rows, unsigned
    ↪ long cols, list movements, explorer* person, FILE* file);
```

Resolve an adventure, given a map and a list of movements.
`items` Map containing all items.
`rows` Number of rows in the map.
`cols` Number of columns in the map.
`movements` Movements to be made by the explorer.
`person` Explorer to collect items.
`file` File to write logs to.
Return: SUCCESS if the adventure was sucessful, CORRECTED if out of bounds movements were corrected, FAILED if out of bounds movements weren't corrected.
Precondition: file must be opened in "a" mode.
Postcondition: items is modified when treasure is picked up by person.
Postcondition: items and movements are not deallocated by this function. Define AI to correct out of bounds movements. Define LOG to log movements to stdout as well as file.

Implementation: Iterate through the list of movements, and keep track of the current location of person. This is done in 2 steps:

1. Get the next movement from the list, and calculate the final block person will be on after this movement. This is done with endBlock.
2. Travel in the direction of the movement using moveDist until the final block is reached. Treasures are collected and logged with collectAndLog.

This is done until movements is exhausted or an error status is set.

1.2 collectAndLog

```
void collectAndLog(FILE* file, map items, explorer* person, long
    ↪ i, long j);
```

Collects the treasure at the given location. Logs this event to a file.
`file` File to log information to.
`items` Map containing item to collect.
`person` Explorer to collect the item.
`i` y location of the treasure to collect.
`j` x location of the treasure to collect.
Precondition: file must be opened in "a" mode.
Postcondition: The treasure in items is either swapped, destroyed (and deallocated), or retained.

Implementation: Switch on the type of the treasure. coin: Deallocate and increase explorer.coin by treasure.value. magic: Deallocate and increase explorer.magic by treasure.value. gear: Use treasure.compare to compare and conditionally swap the two treasures. Log with logTreasure if any deallocation or swapping occured.

1.3 logTreasure

```
void logTreasure(FILE* file, treasure x, int collect, long i,
    ↪ long j);
```

Logs a treasure to a file, and optionally stdout.
`file` File to log information to.
`x` Treasure to log.
`collect` If collect == 0, then log x as collected. Else, log it as discarded.
`i` y location of the treasure.
`j` x location of the treasure.
Precondition: file must be opened in "a" mode. Define LOG to write the information to stdout as well as file.

Implementation: Switch on x.type, and log the treasure according to the assignment specification.

1.4 endBlock

```
status endBlock(unsigned long rows, unsigned long cols, long* i,
    ↪ long* j, move x);
```

Finds the final block after a movement is performed.
`rows` The number of rows in the map.
`cols` The number of cols in the map.
`i` The starting y location. Exports the new y location.
`j` The starting x location. Exports the new x location.
`x` The movement direction and distance.
Return: SUCCESS if the movement was performed correctly, CORRECTED if the movement was corrected, FAILED if the movement is out of bounds. Define AI to correct out of bounds movements.

Implementation: Move the given distance with move_dist. If AI is defined, correct the location if out of bounds. If the distance is still out of bounds, return FAILED.

1.5 move_dist

```
void move_dist(direction dir, unsigned long distance, long* i,
    ↪ long* j);
```

Moves a given distance in a given direction.
`dir` The direction to move.
`distance` The distance to move.
`i` The starting y location. Exports the new y location.
`j` The starting x location. Exports the new x location.
Postcondition: Does not check if the distance is valid.

Implementation: Depending on the direction, change the relevant coordinate.

1.6 insert

```
void insert(list* x, node* iter, void* data);
```

Insert an element into a linked list.
`x` The list to insert into.
`iter` Insert before this element. If iter is NULL, insert at the end of the list.
`data` The pointer to data that will be stored by the node.

Implementation: A node named new is dynamically allocated (on the heap), and data is referenced by it.

- If iter == NULL, insert at the end of the list.
  1. Node before new is x→tail.
  2. Node after new is NULL.
  3. x→tail is now new.
  4. If there are currently no elements in the list, set x→head to new. Otherwise, link the node before new to new.
- Otherwise, the next node is guarenteed to exist.
  1. Node before new is iter→prev.
  2. Node after new is iter.
  3. iter→prev is new.
  4. If iter→prev is NULL (start of list), x→head is new. Otherwise, new→prev→next is new.

1.7 remove_node

```
void remove_node(list* x, node* iter);
```

Remove a node from a linked list.
`x` The list to remove the node from.
`iter` The node to remove. Calls free() on iter.data

Implementation: First, free iter→data. If iter is the first element in x, change the head pointer to iter→next. Otherwise, the previous node.next is equal to the next node. When iter is the final element in x, perform the algorithm above but set the tail pointer and the next node.prev instead.

1.8 make_list

```
list make_list();
```

Initializes an empty list.
Return: The empty list. The list struct is statically allocated (on the stack).

Implementation: Set head and tail to NULL.

1.9 free_list

```
void free_list(list* x);
```

Free all elements in the list.
`x` The list to free. Calls free() on node.data for all nodes.

Implementation: Call remove_node on x→head until x→head is NULL (list is empty).

1.10 for_each

```
void for_each(list x, data_func func);
```

Applies a function to every node in the list.
`x` The list to apply func to.
`func` The function to apply to every node. func is called on each node in order.

Implementation:

1. Apply func to the current node, starting at x.head.
2. Move the current node forward.

Repeat this until the current node is NULL.

1.11 read_map

```
status read_map(map* read_into, long* rows, long* cols, char*
    ↪ filename);
```

Reads in a map from a file.
`read_into` Pointer to the created map.
`rows` The number of rows in the map.
`cols` The number of cols in the map.
`filename` The name of the file to read into.
Return: If an error occurs, returns ABORTED. Otherwise, returns COMPLETE.
Postcondition: read_into must be deallocated with free_map. Writes to stderr if an error has occured. Only deallocate read_into if this function returns COMPLETE.

Implementation: This function performs the majority of file opening, size determination, and allocation. If there is an error in any of the stages listed below, perform any neccesary cleanup and exit the function.

1. Open the file in "r" mode.
2. Read the size of the map.
3. Allocate the map with allocate_map.
4. Read values into the map with fill_map.

1.12 allocate_map

```
map allocate_map(size_t rows, size_t cols);
```

Allocates a map dynamically (on the heap).
`rows` The number of rows in the map.
`cols` The number of columns in the map.
Return: The allocated map. This is NULL if allocation failed.
Postcondition: free_map must be called on the returned map if it is not NULL.

Implementation: The map consists of an array of arrays. Allocate the top level array first. If this was successful, then allocate rows one at a time. If row allocation fails at any point, iterate backwards through the row arrays that have already been allocated, and free them. After allocation, set the type of each map element to 'N'. This represents the map having no elements, and allows deallocation to work as expected.

1.13 free_map

```
void free_map(map x, size_t rows, size_t cols);
```

Deallocates a map that has been created with allocate_map.
`x` The map to free.
`rows` The number of rows in the map.
`cols` The number of columns in the map.

Implementation: The only dynamically allocated element in treasure is the detail string. First, iterate through the map and deallocate any detail strings if the type is either 'G' (gear) or 'M' (magic). Next, deallocate each row array. Finally, deallocate the column array.

1.14 fill_map

```
status fill_map(map read_into, size_t rows, size_t cols, FILE*
    ↪ file);
```

Fills a map with information from a csv file.
`read_into` The map to read values into.
`rows` The number of rows in the map.
`cols` The number of columns in the map.
`file` The file to read from.
Precondition: read_into must be created with allocate_map.
Precondition: file must be opened in "r" mode.
Postcondition: file is not closed by this function.
Return: If an error occurs, return ABORTED. Otherwise, return COMPLETE. Writes to stderr if an error occurs.

Implementation: Stop reading the file as soon as incorrect formatting is encountered. While there are still empty map rows:

1. Read the current line of the file with read_line.
2. Split the line with split.
3. If the correct number of tokens has been read, convert each token into a treasure with make_treasure and add it to the map.

Any errors are printed to stderr. Strtok cannot be used for this function, as it skips repeated delimiters.

1.15 read_line

```
char* read_line(FILE* file);
```

Reads a line from a file.
`file` The file to read from.
Return: The line that has been read from the file.
Precondition: file must be opened in "r" mode.
Postcondition: free must be called on the returned string after use.
Postcondition: Reads until a newline or EOF is encountered. The newline is not included in the returned string. If no characters can be read, an empty string ('
0') is returned.

Implementation: read_line attemps to read a line from a file using fgets, with an initial buffer size of 100. If the last character in the string has been overwritten then the buffer is assumed to be too small, and a bigger buffer is allocated. The file position is then returned to its state at the start of the function, and reading is attempted again. This process is repeated until the entire line has been read. Any newline characters in the string are found with strchr, and removed.

1.16 split

```
status split(char* line, char delim, char** tokens, size_t
    ↪ tokens_sz);
```

Splits a string into tokens, seperated by the given delimiter.
`line` The string to split.
`delim` The delimiter to split the string on.
`tokens` An array of char*. Each element in the array points to a null-terminated token string. These tokens are "stored" in line.
`tokens_sz` The size of tokens.
Return: If the amount of potential tokens in line does not equal tokens_sz, return ABORTED. Otherwise, return COMPLETE.
Postcondition: line should not be read after splitting. The delimiter is not

included in a token. If multiple delimiters occur in order, the corresponding tokens are empty strings. The lifetime of tokens is equal to the lifetime of line.

Implementation: The gist of this function is quite simple, although the implementation becomes more complex as error checking is introduced. At its core, this algorithm:

1. Finds the next delimiter in line.
2. Assigns the location of the delimiter in line to '\0'.
3. Adds a new element to tokens, located 1 after the delimiter. This is the start of the next token.

Now, assume that the above algorithm ends when either:

- The end of the string has been reached.
- The token array is full.

If the number of tokens is less than tokens_sz, there are too few tokens in line and the algorithm ends with status ABORTED. However, if the number of tokens is equal to tokens_sz, it is not yet known if all delimiters have been found or not. To resolve this, find the next character that is either a delimiter or a null terminator. If it is a null terminator, then we have read the correct number of tokens, and the algorithm ends with status COMPLETE. If it is a delimiter, we still have token(s) we have not read yet, and the algorithm ends with status ABORTED.

1.17 print_map

```
void print_map(map x, size_t rows, size_t cols);
```

Prints a map to stdout.
`x` The map to print.
`rows` The number of rows in the map.
`cols` The number of columns in the map. Used for debugging purposes.

Implementation: Iterate through the map, printing each treasure using print(treasure). Also print information about the location of the treasure (row and column).

1.18 read_moves

```
status read_moves(list* moves, char* filename);
```

Reads moves from a file into a linked list.
`moves` The list to read moves into.
`filename` The filename to read from.
Return: If the file contains incorrect formatting, return ABORTED. Otherwise, return COMPLETE.
Precondition: moves is created with make_list.
Postcondition: moves must be deallocated with free_list, regardless of the return status of this function. If the file contains incorrect formatting, print a message to stderr.

Implementation: First, open the file in "r" mode. Then, read and process each line until EOF is reached. The maximum valid line size is 29 characters long, as the largest valid integer representation in c is 22 characters long. (+ 5 characters for RIGHT direction, +1 character for space, +1 for null-terminator). Use fgets to read a line of the file into this buffer. parse_movement is then used to parse the line and add the movement to moves.

1.19 parse_movement

```
status parse_movement(list* moves, char* line);
```

Parse a line of movement and add it to the list.
`moves` The list to read the movement into.
`line` The line to read from.
Return: If there is an error in the line, return ABORTED. Otherwise, return COMPLETE.
Precondition: line must be null-terminated, and must not be NULL. This function modifies moves on faliure, and therefore does not have strong exception safety. If there is an error in the line, a message is printed to stderr.

Implementation: First, allocate space for the move and add it to the end of moves. Doing this early on reduces the chance memory safety is programmed incorrectly. Modifying moves is okay, as strong exception safety is not guarenteed by this function.

Next, check for a space in the line. This is the seperator between the direction and the distance. If there are no spaces in the line, the string is incorrectly formatted and the algorithm ends.

Finally, parse the direction with choose_dir and the distance with sscanf, and check for errors.

1.20 choose_dir

```
direction choose_dir(char* dir);
```

Converts a string to a direction.
`dir` The string describing the direction.

Return: The direction represented by dir. If the direction is invalid, return INVALID.
Precondition: dir must be null-terminated, and must not be NULL.
Postcondition: dir is converted to lowercase in this process.

Implementation: First, convert the string to lowercase. Next, use a simple else-if ladder to choose the correct direction.

1.21 direction_to_string

```
void direction_to_string(direction d, char* str);
```

Converts a direction to an uppercase string.
`d` The direction to convert.
`str` The string to write the direction to.
Postcondition: str must be able to hold at least 8 characters.

Implementation: Use a simple else-if ladder to determine the direction, and print the corresponding string into str. Another method of implementing this would be to create an array linking enumerations to strings.

1.22 print_move

```
void print_move(void* m);
```

Prints a move to stdout.
`m` A pointer to the move to print. m is a void* so that this function is compatible with for_each. Used for debugging purposes.

Implementation: Convert the direction to a string with direction_to_string, and print both the direction and distance to stdout.

1.23 main

```
int main(int argc, char* argv[])
```

The entry point of the program.
`argc` The number of command line arguments. The first argument is the name of the program.
`argv` The command line arguments.
Return: 0 Implements functionality described in the assignment brief.

1.24 make_treasure

```
status make_treasure(char* str, treasure* make);
```

Parses str to create a treasure object.
`str` The string to parse.
`make` The created treasure.
Precondition: str must be null-terminated, and must not be NULL.
Precondition: make must be dynamically allocated (on the heap) before calling this function.
Postcondition: If str is empty, make→type == 'N'.
Return: If the string is invalid, return ABORTED. Otherwise, return COMPLETE. If the string is invalid, print an error message to stderr. Negative treasure values are valid. This is so that negative effects can be impemented. For example, if an explorer stumbles across a potion that ends up being a poison, their magic value is reduced. Likewise, negative coins could be valid if the explorer is robbed by bandits.

Implementation: The first character in the string is the type of treasure.

- If the type is 'C' (coin), then read in the value of the treasure.
- If the type is 'M' (magic), then:
  1. Find the first colon in the string.
  2. Copy the detail string from str + 2 to the seperator. This skips the type character, and the space after the type character.
  3. Parse the value, located at one past the seperator to the end of str.
- If the type is 'G' (gear), then:
  1. Find both delimiters ':' in the string.
  2. Copy the detail string from str + 2 to the seperator.
  3. Find the gear slot with chooseCompareFunc. This is located between the two seperators.
  4. Parse the value, located at one past the second seperator to the end of str.
- If the type is '
  0', then str is empty and the type of treasure is 'N' (none).

1.25 swap

```
void swap(treasure* a, treasure* b);
```

Swaps two treasures.
`a` The first treasure to swap.
`b` The second treasure to swap.
Precondition: a and b must not be NULL.

Implementation: Create a temporary variable and use it to swap the two values.

1.26 chooseCompareFunc

```
compare_func chooseCompareFunc(char* str);
```

Chooses a compare function given a body part.
`str` The string representing a body part.
Return: A function pointer to the correct compare function.
Postcondition: If str is invalid, return NULL.

Implementation: First, convert the string to lowercase. Next, use a simple else-if ladder to determine which compare_func to use.

1.27 slot

```
void slot(treasure x, char* str);
```

Writes the textual representation of the gear slot into the string.
`x` The treasure containing the gear slot.
`str` The string to write the representation into.
Precondition: str must be able to hold at least 6 characters. If x.type != 'G', do not modify str.

Implementation: Use a simple else-if ladder to determine which string to use, comparing the x.compare function pointer.

1.28 toLowerStr

```
void toLowerStr(char* str);
```

Converts the given string to lowercase.
`str` The string to convert to lowercase.

Implementation: Iterate through the string, using tolower to convert each character to lowercase.

1.29 print

```
void print(treasure x);
```

Prints a treasure to stdout.
`x` The treasure to print. Used for debugging. If x.type == 'N', do not print anything.

Implementation: If the type is 'N', do not print anything. All non-empty treasures have types and values. These are printed. If the type is not 'C' (coin),

the treasure has a detail string. This is printed. If the type is 'G' (gear), print the slot it refers to. This uses slot to convert x.compare into a string.

1.30 compareHead

```
int compareHead(treasure* gear, explorer* person);
```

Implementation of compare_func for the head slot.

Implementation: If person→head.value is less than gear.value, use swap to swap the objects.

1.31 compareChest

```
int compareChest(treasure* gear, explorer* person);
```

Implementation of compare_func for the head slot.

Implementation: Same as compareHead, except the Chest slot is compared instead.

1.32 compareLegs

```
int compareLegs(treasure* gear, explorer* person);
```

Implementation of compare_func for the head slot.

Implementation: Same as compareHead, except the Legs slot is compared instead.

1.33 compareHands

```
int compareHands(treasure* gear, explorer* person);
```

Implementation of compare_func for the head slot.

Implementation: Same as compareHead, except the Hands slot is compared instead.

1.34 make_explorer

```
explorer make_explorer();
```

Creates an explorer.
Return: An explorer with no magic, coin, or gear.

Implementation: Initialize magic and coin to 0. Initialize gear.type to 'N'. Initialize gear.value to -1, so that when compared the empty slot is always initially swapped.

1.35 free_explorer

```
void free_explorer(explorer ex);
```

Frees any memory owned by the explorer.
`ex` The explorer to free.

Implementation: If gear.type == 'G' (non-empty), free the corresponding detail string.

1.36 gear_value

```
int gear_value(explorer ex);
```

Finds the total value of all gear owned by the explorer.
`ex` The explorer to calculate the total value of all gear for.
Return: The total value of all gear.
Precondition: ex must be created with make_explorer.

Implementation: If gear.type == 'G' (there is gear in that slot), increment the accumulator by that gear's value.

1.37 status_text

```
void status_text(status s, char* text_rep);
```

Converts the status to text.
`s` The status to convert.
`text_rep` The location to print the status string.
Precondition: text_rep must be able to hold at least 10 characters.

Implementation: Simple else-if ladder to read the correct string into text_rep.

2 Conversion of Input File to Coordinate System

2.1 Implementation

The top level function to convert the input file is read_map. This function opens the file, reads and validates the map size (rows and columns), and uses allocate_map to allocate memory for the map.

Once this is done, fill_map is called. This function parses the file line by line, and reads values into the map. First, read_line is called. This function reads a line of arbitrary size by reallocating larger buffers until the line is read in full. This means that the map can be as large as the user wants. Once the line is read, it is split into tokens using split. This function iterates over the provided string, finds a given delimiter (',' in this case) with strchr, and substitutes it will a null-terminator. It then exports an array filled with pointers to the start of each token. This has the effect of breaking a string up into many substrings, seperated by the delimiter.

Once this is done, each individual token is passed to make_treasure. This function uses a switch statement on the first character to determine the type of the treasure. Once the type is determined, the treasure is usually parsed by finding the colon character(s) with strchr, and either using strncpy or sscanf to parse each section. In the case of a gear treasure, the function chooseCompareFunc is used to determine which function pointer should be used with the given slot.

2.2 Alternate Implementation

An alternate approach would be to either use strtok, or the given tokenizer, to parse each line. This would mean that make_treasure would be passed tokens one at a time, rather than splitting the entire string and then passing all tokens to make_treasure. This method was not chosen, as strtok ignores repeated characters and is therefore unsuitable for any empty sections of the map. strtok could also be used to parse individual treasures, instead of using strchr.

3 Sample Input and Output

All test results are generated from the TreasureHunter binary. This is called with the arguments:

```
./TreasureHunter <map file> <list file>
```

The only exception to this is the test for incorrect number of arguments. This test is executed as such:

```
./TreasureHunter
```

The expected output for this test is:

```
Usage: ./TreasureHunter <map_file> <movement_file>
```

A script to run these tests is located in test/all_tests.sh. There is no user input for any test cases. If example output from adventure.log is not provided, it should be assumed that the binary does not provide log file output for that test case.

The TreasureHunterLog binary will produce the same output as the TreasureHunter binary, except any output to the log file (adventure.log) is also output to stdout. This is done real time; as lines are printed to the log file, the same content is printed to stdout.

The TreasureHunterAI binary is assumed to produce the same output as the TreasureHunter binary, unless otherwise stated.

3.1 Example Input from Assignment Brief

Map

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

List

```
DOWN 2
RIGHT 2
DOWn 2
up 0
LEFT 1
UP 3
riGHT 0
RIGHT 2
```

stdout

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

Log file

```
---
COLLECT<ITEM:MAGIC, XLOC:2, YLOC:0, DESCRIPTION:Healing Potion,
    ↪ VALUE:85>
COLLECT<ITEM:MAGIC, XLOC:2, YLOC:2, DESCRIPTION:Defence
    ↪ Enchantment, VALUE:360>
COLLECT<ITEM:GEAR, XLOC:4, YLOC:2, 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:1, YLOC:3, VALUE:50>
```

3.2 Out of Bounds

3.2.1 TreasureHunter

Map

```
7,3
,g Thorn armour:CheST:120,C 200
,G Vibranium Shield:hands:990,
M Healing Potion:85,,M Defence Enchantment:360
,m Phoenix Blood:291,
,g IDEK_ANYMORE:heaD:21,G Lightsaber:hAnDs:850
c 350,g Crimson Plate:cHeSt:1020,
,G Infinity Pants:leGS:30000,
```

List

```
LEFt 3
RIGHT 1
down 2
LEfT 5
RIGHT 20
left 1
DOWN 100
UP 2000
```

stdout

```
STATUS: FAILED
```

Log file

```
---
```

3.2.2 TreasureHunterAI

Map

```
7,3
,g Thorn armour:CheST:120,C 200
,G Vibranium Shield:hands:990,
M Healing Potion:85,,M Defence Enchantment:360
,m Phoenix Blood:291,
,g IDEK_ANYMORE:heaD:21,G Lightsaber:hAnDs:850
c 350,g Crimson Plate:cHeSt:1020,
,G Infinity Pants:leGS:30000,
```

List

```
LEFt 3
RIGHT 1
down 2
LEfT 5
RIGHT 20
left 1
DOWN 100
UP 2000
```

stdout

```
STATUS: CORRECTED
COINS: 0
MAGIC: 736
GEAR: 32031
```

Log file

```
---
COLLECT<ITEM:GEAR, XLOC:0, YLOC:1, DESCRIPTION:Thorn armour, SLOT
    ↪ :chest, VALUE:120>
COLLECT<ITEM:GEAR, XLOC:1, YLOC:1, DESCRIPTION:Vibranium Shield,
    ↪ SLOT:hands, VALUE:990>
COLLECT<ITEM:MAGIC, XLOC:2, YLOC:0, DESCRIPTION:Healing Potion,
    ↪ VALUE:85>
COLLECT<ITEM:MAGIC, XLOC:2, YLOC:2, DESCRIPTION:Defence
    ↪ Enchantment, VALUE:360>
COLLECT<ITEM:MAGIC, XLOC:3, YLOC:1, DESCRIPTION:Phoenix Blood,
    ↪ VALUE:291>
COLLECT<ITEM:GEAR, XLOC:4, YLOC:1, DESCRIPTION:IDEK_ANYMORE, SLOT
    ↪ :head, VALUE:21>
COLLECT<ITEM:GEAR, XLOC:5, YLOC:1, DESCRIPTION:Crimson Plate,
    ↪ SLOT:chest, VALUE:1020>
DISCARD<ITEM:GEAR, XLOC:5, YLOC:1, DESCRIPTION:Thorn armour, SLOT
    ↪ :chest, VALUE:120>
COLLECT<ITEM:GEAR, XLOC:6, YLOC:1, DESCRIPTION:Infinity Pants,
    ↪ SLOT:legs, VALUE:30000>
```

3.3 Empty List

Map

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

List

stderr

```
File <list file> was empty.
```

3.4 Cannot open List file

Map

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

List Does not exist.

stderr

```
Error opening file <list file> for reading: No such file or
    ↪ directory
```

3.5 Invalid Direction

Map

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

List

```
LEFT 3
riGhtt 1
DOWN 2
LEFT 5
RIGHT 20
NOrtH 1
DOWN 100
UP 2000
```

stderr

```
Invalid direction
At line 2.
```

3.6 Invalid Distance

Map

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

List

```
DOWN 2
RIGHT 2
DOWn %s
up 0
LEFT 1
UP 3
riGHT 0
RIGHT 2
```

stderr

```
Distance must be an integer
At line 3.
```

3.7 No Space in List

Map

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

List

```
DOWN 2
RIGHT 2
DOWn 2
up 0
LEFT1
UP 3
riGHT 0
RIGHT 1
```

stderr

```
No space
At line 5.
```

3.8 Negative Distance

Map

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

List

```
DOWN 2
RIGHT 2
DOWn 2
up 0
LEFT 1
UP 3
riGHT 0
RIGHT -100
```

stderr

```
Distance must be positive
At line 8.
```

3.9 Floating Point Distance

Map

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

List

```
DOWN 2
RIGHT 2
DOWn 2
up 0
LEFT 1
UP 3
riGHT 0.5
RIGHT 100
```

stderr

```
Distance must be an integer
At line 7.
```

3.10 Empty Map

Map

```
```

List

```
DOWN 2
RIGHT 2
DOWn 2
up 0
LEFT 1
UP 3
riGHT 0
RIGHT 2
```

stderr

```
Incorrect formatting in <map file>, line 1: Expected positive
    ↪ integers <rows>,<cols>
```

3.11 Map File does not Exist

Map Does not exist.
List

```
DOWN 2
RIGHT 2
DOWn 2
up 0
LEFT 1
UP 3
riGHT 0
RIGHT 2
```

stderr

```
Error opening file <map file> for reading: No such file or
    ↪ directory
```

3.12 Incorrect Rows

Map

```
7,3
,g Thorn armour:CheST:120,C 200
,G Vibranium Shield:hands:990,
M Healing Potion:85,,M Defence Enchantment:360
,m Phoenix Blood:291,
,g IDEK_ANYMORE:heaD:21,G Lightsaber:hAnDs:850
c 350,g Crimson Plate:cHeSt:1020,
```

List

```
DOWN 2
RIGHT 2
DOWn 2
up 0
LEFT 1
UP 3
riGHT 0
RIGHT 2
```

stderr

```
Incorrect number of rows: read 6, expected 7.
```

3.13 Incorrect Columns

Map

```
7,3
,g Thorn armour:CheST:120,C 200
,G Vibranium Shield:hands:990,
M Healing Potion:85,,M Defence Enchantment:360
,m Phoenix Blood:291,
,g IDEK_ANYMORE:heaD:21,G Lightsaber:hAnDs:850
c 350,g Crimson Plate:cHeSt:1020
,G Infinity Pants:leGS:30000,
```

List

```
DOWN 2
RIGHT 2
DOWn 2
up 0
LEFT 1
UP 3
riGHT 0
RIGHT 2
```

stderr

```
Incorrect number of columns at line 7: expected 3.
```

3.14 Negative Columns

Map

```
7,-3
,g Thorn armour:CheST:120,C 200
,G Vibranium Shield:hands:990,
M Healing Potion:85,,M Defence Enchantment:360
,m Phoenix Blood:291,
,g IDEK_ANYMORE:heaD:21,G Lightsaber:hAnDs:850
c 350,g Crimson Plate:cHeSt:1020,
,G Infinity Pants:leGS:30000,
```

List

```
DOWN 2
RIGHT 2
DOWn 2
up 0
LEFT 1
UP 3
riGHT 0
RIGHT 2
```

stderr

```
Incorrect formatting in <map file>, line 1: Expected positive
    ↪ integers <rows>,<cols>
```

3.15 Invalid Rows

Map

```
7.5,3
,g Thorn armour:CheST:120,C 200
,G Vibranium Shield:hands:990,
M Healing Potion:85,,M Defence Enchantment:360
,m Phoenix Blood:291,
,g IDEK_ANYMORE:heaD:21,G Lightsaber:hAnDs:850
c 350,g Crimson Plate:cHeSt:1020,
,G Infinity Pants:leGS:30000,
```

List

```
DOWN 2
RIGHT 2
DOWn 2
up 0
LEFT 1
UP 3
riGHT 0
RIGHT 2
```

stderr

```
Incorrect formatting in <map file>, line 1: Expected positive
    ↪ integers <rows>,<cols>
```

3.16 Invalid Treasure Type

Map

```
7,3
,g Thorn armour:CheST:120,C 200
,G Vibranium Shield:hands:990,
M Healing Potion:85,,M Defence Enchantment:360
,m Phoenix Blood:291,
,g IDEK_ANYMORE:heaD:21,f Lightsaber:hAnDs:850
c 350,g Crimson Plate:cHeSt:1020,
,G Infinity Pants:leGS:30000,
```

List

```
DOWN 2
RIGHT 2
DOWn 2
up 0
LEFT 1
UP 3
riGHT 0
RIGHT 2
```

stderr

```
F is not a valid treasure type.
At row 5, column 3.
```

3.17 Invalid Slot

Map

```
7,3
,g Thorn armour:CheST:120,C 200
,G Vibranium Shield:hands:990,
M Healing Potion:85,,M Defence Enchantment:360
,m Phoenix Blood:291,
,g IDEK_ANYMORE:heaD:21,G bow and arrows:feet:850
c 350,g Crimson Plate:cHeSt:1020,
,G Infinity Pants:leGS:30000,
```

List

```
DOWN 2
RIGHT 2
DOWn 2
up 0
LEFT 1
UP 3
riGHT 0
RIGHT 2
```

stderr

```
Incorrect formatting. Gear is represented as: "G <detail>:<slot
    ↪ >:<value>"
At row 5, column 3.
```

3.18 Invalid Value

Map

```
7,3
,g Thorn armour:CheST:120,C 200
,G Vibranium Shield:hands:99.5,
M Healing Potion:85,,M Defence Enchantment:360
,m Phoenix Blood:291,
,g IDEK_ANYMORE:heaD:21,G Lightsaber:hAnDs:850
c 350,g Crimson Plate:cHeSt:1020,
,G Infinity Pants:leGS:30000,
```

List

```
DOWN 2
RIGHT 2
DOWn 2
up 0
LEFT 1
UP 3
riGHT 0
RIGHT 2
```

stderr

```
Incorrect formatting. Gear is represented as: "G <detail>:<slot
    ↪ >:<value>"
At row 2, column 2.
```

3.19 Too Many Colons

Map

```
7,3
,g Thorn armour:CheST:120,C 200
,G Vibranium Shield:hands:990,
M Healing:Potion:85,,M Defence Enchantment:360
,m Phoenix Blood:291,
,g IDEK_ANYMORE:heaD:21,G Lightsaber:hAnDs:850
c 350,g Crimson Plate:cHeSt:1020,
,G Infinity Pants:leGS:30000,
```

List

```
DOWN 2
RIGHT 2
DOWn 2
up 0
LEFT 1
UP 3
riGHT 0
RIGHT 2
```

stderr

```
Incorrect formatting. Magic items are represented as: "M <detail
    ↪ >:<value>"
At row 3, column 1.
```

3.20 Too Little Colons

Map

```
7,3
,g Thorn armour:CheST:120,C 200
,G Vibranium Shield:hands:990,
M Healing Potion:85,,M Defence Enchantment:360
,m Phoenix Blood:291,
,g IDEK_ANYMORE:heaD:21,G Lightsaber:hAnDs:850
c 350,g Crimson Plate:cHeSt:1020,
,G Infinity Pants leGS:30000,
```

List

```
DOWN 2
RIGHT 2
DOWn 2
up 0
LEFT 1
UP 3
riGHT 0
RIGHT 2
```

stderr

```
Incorrect formatting. Gear is represented as: "G <detail>:<slot
    ↪ >:<value>"
At row 7, column 2.
```

3.21 No Space in Map

Map

```
7,3
,g Thorn armour:CheST:120,C 200
,G Vibranium Shield:hands:990,
MHealing Potion:85,,M Defence Enchantment:360
,m Phoenix Blood:291,
,g IDEK_ANYMORE:heaD:21,G Lightsaber:hAnDs:850
c 350,g Crimson Plate:cHeSt:1020,
,G Infinity Pants:leGS:30000,
```

List

```
DOWN 2
RIGHT 2
DOWn 2
up 0
LEFT 1
UP 3
riGHT 0
RIGHT 2
```

stderr

```
Incorrect formatting. Magic items are represented as: "M <detail
    ↪ >:<value>"
At row 3, column 1.
```