COS10007

Assignment 2

Question 1

// Question1.c

// Correct ammount of arguments for program to expect.

#define ARG\_COUNT 1

// atoi

#include <stdlib.h>

// Basic io

#include <stdio.h>

// Boolean type.

#include <stdbool.h>

// strcpy

#include <string.h>

// usleep

#include <unistd.h>

// Header file.

#include "Question1.h"

// Argument priority is as follows: (higher == leftmost)

// FPTR fp

// ll \*\* o

// ll \*\* g

// ll \*\* s

// char \* \_

ll \* add\_game(ll \*\* g, char \* name);

ll \* add\_order(ll \*\* o, const size\_t on);

int read\_file(FPTR fp, ll \*\* g, const char \* fn);

int write\_file(FPTR fp, ll \*\* g, const char \* fn);

void print\_games\_details(ll \*\* g);

void display\_order(ll \*\* o);

void process\_next\_order(ll \*\* o, ll \*\* g, ll \*\* s);

void display\_all\_orders(ll \*\* o);

void cancel\_last\_order(ll \*\* o);

void display\_next\_order(ll \*\* o);

void display\_completed\_orders(ll \*\* s);

// argc == 2

// argv[2] == [this\_file, games.txt]

int main(int argc, char \*argv[]) {

// Order numbers are sequential and start from 1337 because memes.

size\_t ordnum = 1337;

// Pointer for storing the input file.

FPTR fp = NULL;

if(argc != ARG\_COUNT + 1) {

fprintf(stderr, "Invalid number of arguments, must be %d.\n", ARG\_COUNT);

return 1;

}

// Attempt to open last argument, parse to fill linked list.

// Kill program if this fails.

ll \* games\_ll = NULL;

if(1 == read\_file(fp, &games\_ll, argv[ARG\_COUNT])) {

fprintf(stderr,

"Failed to open file, check permissions.\n"

"Filename: %s\n",

argv[ARG\_COUNT]);

return 1;

}

if(games\_ll == NULL) {

printf("No games to sell.\nExiting...\n");

return 1;

}

// Storage for orders.

ll \* orders\_ll = NULL;

// Storage for games that have been sold.

ll \* sold\_ll = NULL;

// Use this to pass to user\_input\_char.

// like this: user\_input\_char(&input);

char input[50] = "\0";

bool menu(void) {

clear\_screen();

printf("GENERIC GAMES STORE NAME HERE\n"

"\n"

"1 : Display current stock\n"

"2 : Add a game to stock\n"

"3 : Add an order\n"

"4 : Cancel last order\n"

"5 : Process next order\n"

"6 : Display all orders\n"

"7 : Display next order\n"

"8 : Display completed orders\n"

"9 : Update games file\n"

"10 : Exit\n"

"\n"

"\n");

switch(user\_input\_int()) {

case 1:

clear\_screen();

printf("Print available stock.\n\n");

print\_games\_details(&games\_ll);

break;

case 2:

clear\_screen();

printf("Add a game to stock.\n\n");

printf("Adding a new game:\n"

"\n"

"Name: ");

user\_input\_char(&input);

if(1 < strlen(input)) {

add\_game(&games\_ll, input);

} else {

printf("Error: Input too short.\n");

}

break;

case 3:

clear\_screen();

printf("Add an order.\n\n");

printf( "Order #%d added to queue.\n",

(add\_order(&orders\_ll, ordnum++))->data->order.id);

break;

case 4:

clear\_screen();

printf("Cancel last order.\n\n");

cancel\_last\_order(&orders\_ll);

break;

case 5:

clear\_screen();

printf("Process next order.\n\n");

process\_next\_order(&orders\_ll, &games\_ll, &sold\_ll);

break;

case 6:

clear\_screen();

printf("Display all orders.\n\n");

display\_all\_orders(&orders\_ll);

break;

case 7:

clear\_screen();

printf("Display next order.\n\n");

display\_next\_order(&orders\_ll);

break;

case 8:

clear\_screen();

printf("Display completed orders.\n\n");

display\_completed\_orders(&sold\_ll);

break;

case 9:

clear\_screen();

printf("Update games stock file.\n\n");

if(0 == write\_file(fp, &games\_ll, argv[ARG\_COUNT])) {

printf("Write success.\n");

} else {

fprintf(stderr, "Failed to open file for writing.\n");

}

break;

case 10:

clear\_screen();

printf("Quit.\n\n");

printf("Exiting...\n");

return false;

default:

// No option selected, just reprint menu.

return true;

}

printf("\n\nPress enter to continue.\n");

user\_input\_int();

return true;

}

// Loop menu till it returns false.

while(menu()) {}

// Cleanup.

ll\_free(&games\_ll);

ll\_free(&orders\_ll);

ll\_free(&sold\_ll);

return 0;

}

// Print the names of the games.

void print\_games\_details(ll \*\* g) {

if(g != NULL) {

for( ll \* temp = \*g;

temp != NULL;

temp = temp->next) {

printf("%s", temp->data->game.name);

}

} else {

printf("No games to sell.\n");

}

}

// Add a new game to the game stock ll.

ll \* add\_game(ll \*\* g, char \* name) {

// Sanity check.

if(g != NULL) {

ll \* newgame = ll\_new\_node();

strcpy(newgame->data->game.name, name);

ll\_push(&newgame, g);

return newgame;

}

}

// Scan file for games into a ll, return a pointer to the start of the ll.

// Accepts:

// Filename

// File pointer

// \*\* to ll of games

// Returns:

// int (For error reporting)

int read\_file(FPTR fp, ll \*\* g, const char \* fn) {

if(open\_file(fn, &fp, "r") == 1) {return 1;}

char \* line = NULL;

size\_t len = 0;

while((getline(&line, &len, fp)) != -1) {

// Spawn a new node.

ll \* temp\_new\_game = ll\_new\_node();

// Give it game details.

strcpy(temp\_new\_game->data->game.name, line);

// Push it to the games list that will be returned.

ll\_push(&temp\_new\_game, g);

}

// Line will be malloc'd by getline, free the memory.

free(line);

fclose(fp);

return 0;

}

// Write games to file.

// Accepts:

// Filename

// File pointer

// \*\* to ll of games

// Returns:

// int (For error reporting)

int write\_file(FPTR fp, ll \*\* g, const char \* fn) {

if(open\_file(fn, &fp, "w") == 1) {return 1;}

for( ll \* temp\_games\_p = \*g;

temp\_games\_p != NULL;

temp\_games\_p = temp\_games\_p->next) {

// Write each game to the file.

fprintf(fp, "%s", temp\_games\_p->data->game.name);

}

fclose(fp);

return 0;

}

// Add a new order to the customer order ll.

ll \* add\_order(ll \*\* o, const size\_t on) {

ll \* neworder = ll\_new\_node();

neworder->data->order.id = on;

printf("Input name of game to be ordered (order ID: %d): ", on);

user\_input\_char(&neworder->data->order.contents.name);

ll\_tail\_push(&neworder, o);

return neworder;

}

// Display details of a single order.

// For use in other functions to keep output consistent.

void display\_order(ll \*\* o) {

// Sanity check.

if(o != NULL && \*o != NULL) {

printf("Order ID: %d\n", (\*o)->data->order.id);

printf("Order contents: %s\n", (\*o)->data->order.contents.name);

printf("\n");

}

}

// !!! ISSUE !!!

// (refer to comment in func)

// Search for a game in the linked list, then return it's location.

// Returns -1 if not found or if the list is NULL.

int search\_for\_game(ll \*\* g, char \* name) {

if(g != NULL && \*g != NULL) {

// For storing location in list by index from head.

size\_t i = 0;

for(ll \* temp = \*g; temp != NULL; temp = temp->next) {

i++;

// Dumb search with strcmp, may need to be changed to strstr.

if(0 == strcmp(temp->data->game.name, name)) {

return i;

}

}

} else {

printf("Nothing to display.");

}

return -1;

}

// Process the order at the head of the ll, place the sold game in the sold ll.

void process\_next\_order(ll \*\* o, ll \*\* g, ll \*\* s) {

// Sanity check.

if(o != NULL && \*o != NULL) {

// Grab the next order.

ll \* ordertemp = ll\_pop(o);

// This is for storing the game that is found and removed from the list.

ll \* gametemp = NULL;

// search\_for\_game will return -1 when not found.

int gameindex = search\_for\_game(g, ordertemp->data->order.contents.name);

if(0 > gameindex) {

// Game was not found.

// There is a small issue where the game has a \n in the name,

// which makes this printf split to two lines unintentionally.

// I can't be bothered fixing this because it's unimportant.

printf( "The game: %s"

"is currently not available, "

"order sent to end of queue.",

ordertemp->data->order.contents.name);

// Push the order back to the tail as it is unavailable right now

// and there are other orders to be filled..

ll\_tail\_push(&ordertemp, o);

} else {

// Game was found gameindex is a positive value.

gametemp = ll\_remove\_by\_index(g, (size\_t) gameindex);

// This will fill the order's details out with any extra information

// that the game knows about itself.

ordertemp->data->order.contents = gametemp->data->game;

// We don't need the game any more because it has been bought.

ll\_free(&gametemp);

// Push the order into the completed orders ll.

printf("Order processed: \n");

display\_order(&ordertemp);

// Dump the order in the completed orders list.

ll\_push(&ordertemp, s);

}

} else {

printf("There are no orders in the queue.");

}

}

// Display all orders in the customer orders ll.

void display\_all\_orders(ll \*\* o) {

// Sanity check.

if(o != NULL && \*o != NULL) {

// List is ! NULL, step through orders and display them.

for(ll \* temp = \*o; temp != NULL; temp = temp->next) {

display\_order(&temp);

}

} else {

printf("No orders in the queue.\n");

}

}

// Remove the last order that was created from the tail of the ll.

void cancel\_last\_order(ll \*\* o) {

// Sanity check.

if(o != NULL && \*o != NULL) {

ll \* temp = ll\_tail\_pop(o);

printf("Order removed.\n");

display\_order(&temp);

ll\_free(&temp);

} else {

printf("No orders in the queue.\n");

}

}

// Display the next order in the queue.

void display\_next\_order(ll \*\* o) {

// Sanity check.

if(o != NULL && \*o != NULL) {

display\_order(o);

} else {

printf("No orders in the queue.\n");

}

}

// Display the order at the end of the ll.

void display\_completed\_orders(ll \*\* s) {

// Sanity check.

if(s != NULL && \*s != NULL) {

for(ll \* temp = \*s;

temp != NULL;

temp = temp->next) {

display\_order(&temp);

}

} else {

printf("No completed orders.\n");

}

}

// Question1.h

// typedef laziness.

typedef FILE \* FPTR;

// Game data.

typedef struct Game\_Data {

char name[50];

}game\_data;

// Order data.

typedef struct Order\_Data {

int id;

game\_data contents;

}order\_data;

// Union for storing different pointers inside the data section of the ll.

typedef union StructVariant {

game\_data game;

order\_data order;

}structVariant;

// Customer orders linked list.

typedef struct LL {

structVariant \* data;

struct LL \* next;

}ll;

// Print a bunch of endlines to screen to clear it the lazy way.

void clear\_screen(void) {

const size\_t line\_count = 100;

for(int i = 0; i < line\_count; i++) {

printf("\n");

}

}

// Open file, return int depending on if successful.

int open\_file( const char \* file\_to\_open,

FPTR \* fpp,

const char \* mode) {

if((\*fpp = fopen(file\_to\_open, mode)) == NULL) {

fprintf(stderr, "Failed to open file.");

return 1;

} else {

return 0;

}

}

// Take user input, return int.

int user\_input\_int(void) {

char input[128];

fflush(stdin);

fgets(input, 127, stdin);

return atoi(input);

}

void user\_input\_char(char (\*input)[50]) {

fflush(stdin);

fgets(\*input, 49, stdin);

// Make sure there's a carriage return before the '\0'.

for(int i = 0; 50 > i && '\0' != (\*input)[i]; i++) {

if('\0' == (\*input)[i + 1]) {

(\*input)[i] = '\n';

}

}

}

// LL PUSH

// Both accept:

// ll \*\* to the node to be pushed and

// ll \*\* to the head of the list.

// HEAD

// Push a node into the head of a singly linked list.

void ll\_push(ll \*\* pushme, ll \*\* goeshere) {

(\*pushme)->next = \*goeshere;

\*goeshere = \*pushme;

}

// TAIL

// Push a node into the tail of a singly linked list using the head as a

// reference point.

void ll\_tail\_push(ll \*\* pushme, ll \*\* goeshere) {

ll \* temp = \*goeshere;

if(temp != NULL) {

while(temp->next != NULL) {

temp = temp->next;

}

temp->next = \*pushme;

} else {

\*goeshere = \*pushme;

}

}

// LL POP

// Both accept:

// ll \*\* to the head of the list.

// Both return:

// ll \* to the popped node.

// HEAD

// Pop a node off of the head of a singly linked list.

ll \* ll\_pop(ll \*\* popme) {

// Save the pointer to the head.

ll \* temp = \*popme;

// Shift the head down one.

\*popme = (\*popme)->next;

// Make sure it's ->next is NULL so that it no longer points back to this list.

temp->next = NULL;

// Return the severed head.

return temp;

}

// TAIL

// Pop a node of the tail of a singly linked list using the head as a reference

// point.

ll \* ll\_tail\_pop(ll \*\* popme) {

ll \* tail = \*popme;

// Stop stepping through when next == NULL.

while( tail != NULL &&

tail->next != NULL) {

tail = tail->next;

}

ll \* tail\_prev = \*popme;

if(tail\_prev->next != NULL) {

// The list is more than one item long.

// Stop stepping through when next == tail.

while( tail\_prev != NULL &&

tail\_prev->next != NULL &&

tail\_prev->next != tail) {

tail\_prev = tail\_prev->next;

}

tail\_prev->next = NULL;

} else {

// The list is only one item long.

\*popme = NULL;

}

// Make sure it's ->next is NULL so that it no longer points back to this list.

tail->next = NULL;

return tail;

}

// Create a node to be pushed onto a linked list.

ll \* ll\_new\_node(void) {

// Create a new node with a null pointer.

ll \* new\_node\_p = NULL;

new\_node\_p = (ll\*) malloc(sizeof(ll));

new\_node\_p->next = NULL;

//new\_node\_p->data.game = NULL;

// Give the node it's data.

new\_node\_p->data = (structVariant\*) malloc(sizeof(structVariant));

return new\_node\_p;

}

// Free up an ll and it's data.

void ll\_free(ll \*\* freethis) {

if(freethis != NULL) {

while(\*freethis != NULL) {

// Because the list is NULL terminated,

// \*freethis ends up getting set to null once we reach the end.

ll \* temp = ll\_pop(freethis);

// Check if this is the correct way to do this.

free(temp->data);

// Free the actual link in the list.

free(temp);

}

}

}

// Remove a specific element from the list n jumps from the head.

// head == 0

ll \* ll\_remove\_by\_index(ll \*\* head, const size\_t node\_index) {

ll \* pointernode = ll\_new\_node();

ll \* temp = pointernode;

temp->next = \*head;

// Find the node before the one to be removed.

for( int i = 0;

node\_index - 1 > i &&

head != NULL;

i++) {

temp = temp->next;

}

// Check it isn't a NULL pointer.

if(temp == NULL) {

// The index pointed past the end of the list.

fprintf(stderr, "Invalid index.\n");

return NULL;

}

// Make a join around it then return it by way of pointer.

ll \* removed = temp->next;

temp->next = removed->next;

// Make sure it's ->next is NULL

// so that it no longer points back to this list.

removed->next = NULL;

// Make sure if the head was the node that was removed then the head is set

// to the next node down.

\*head = pointernode->next;

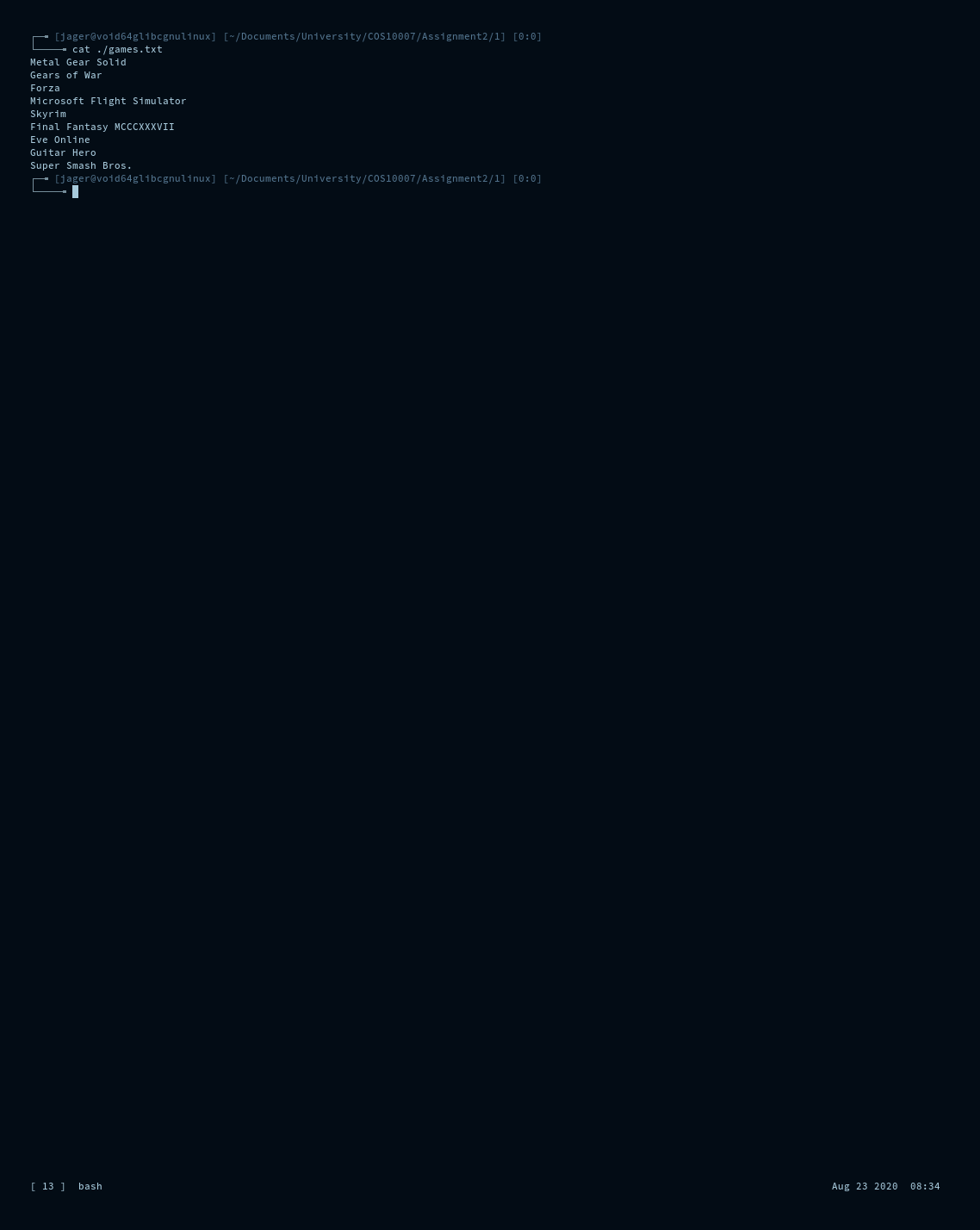
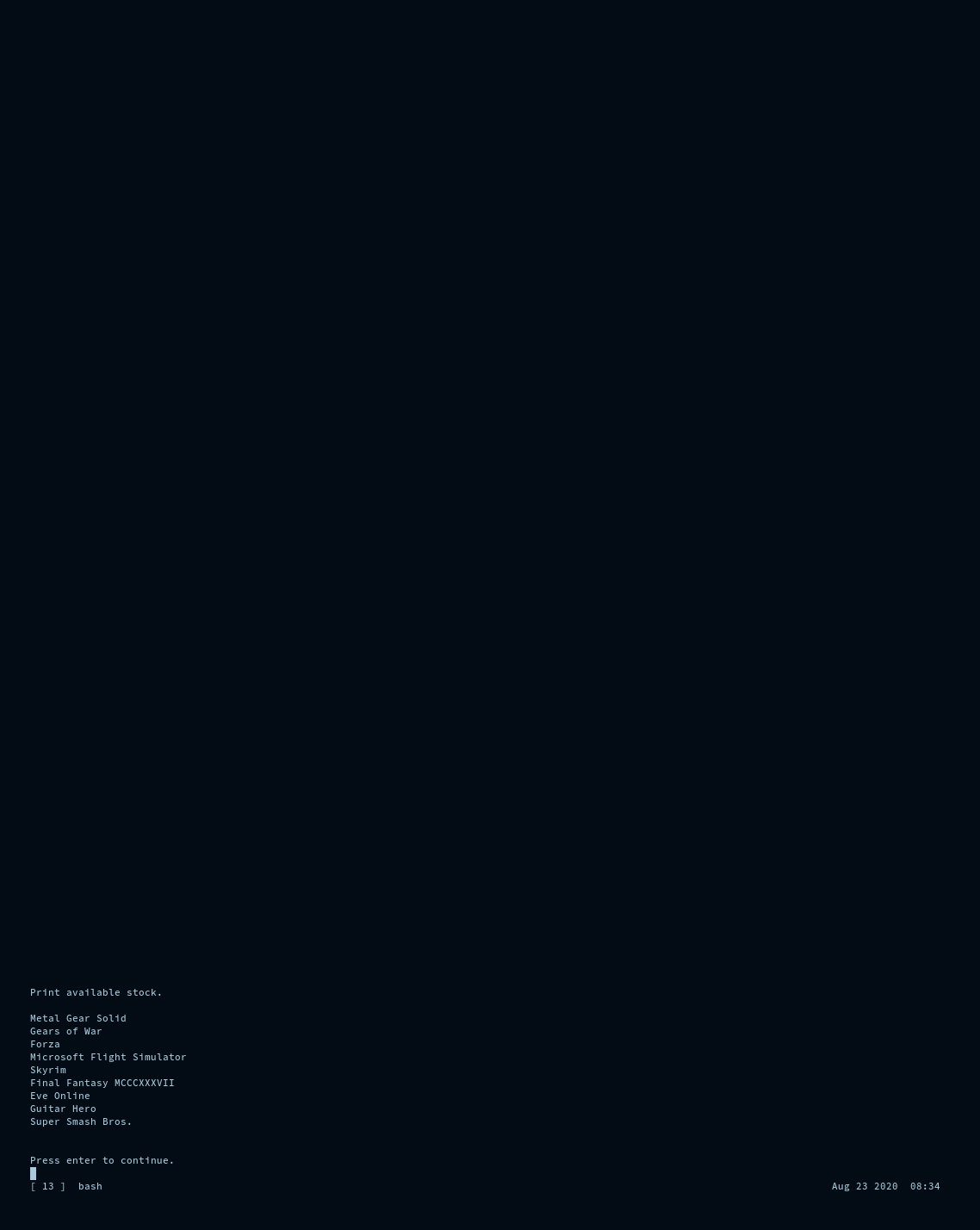
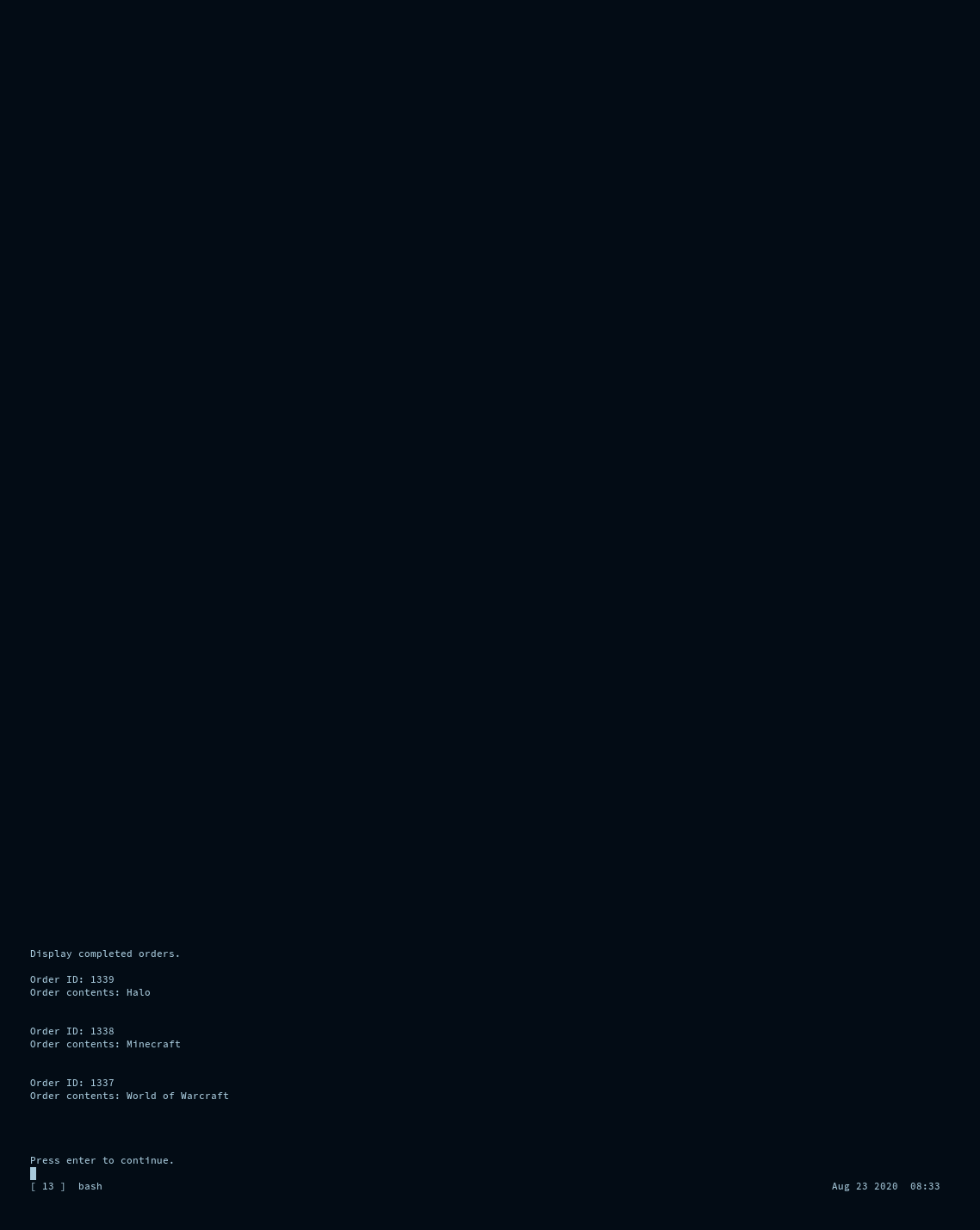
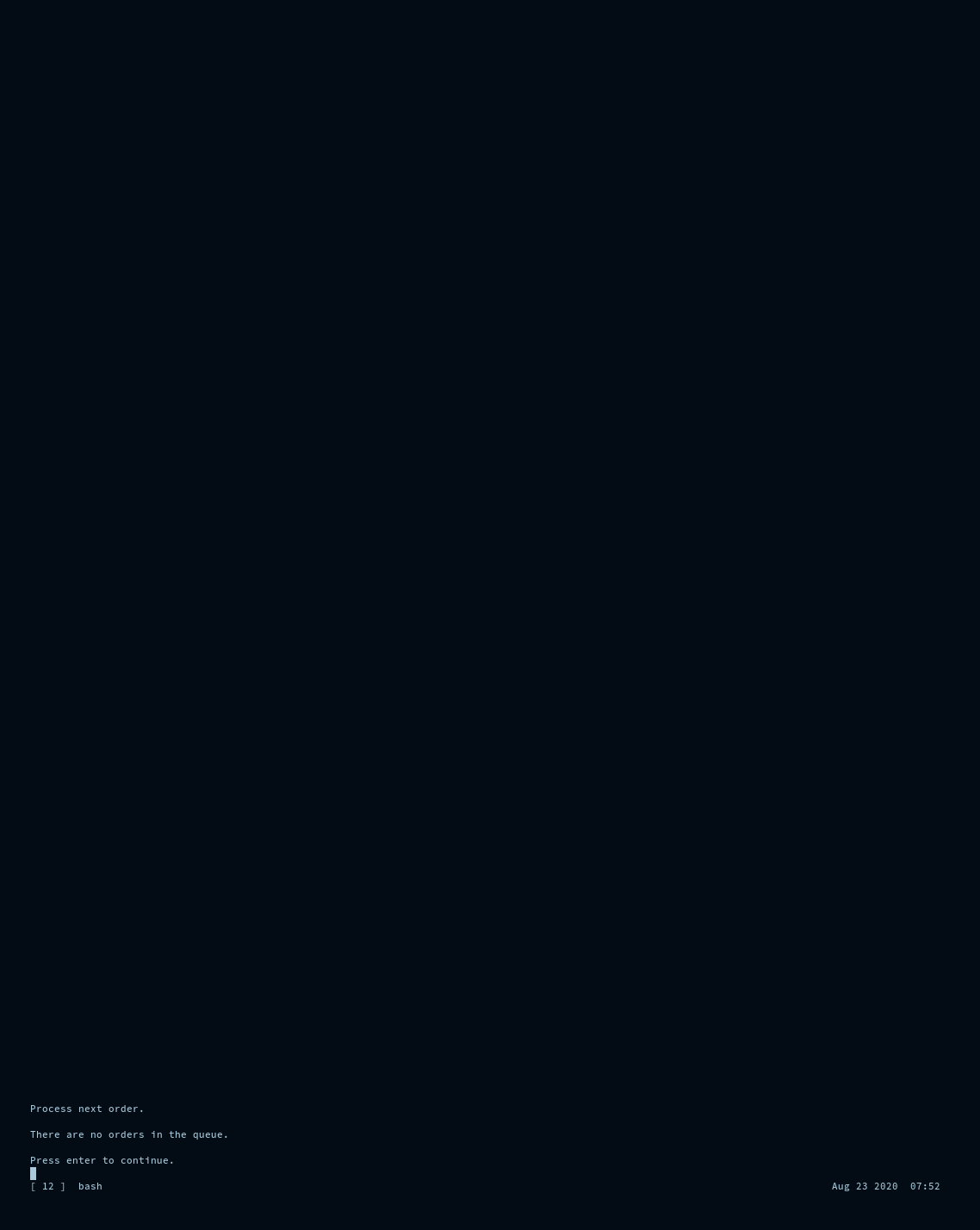
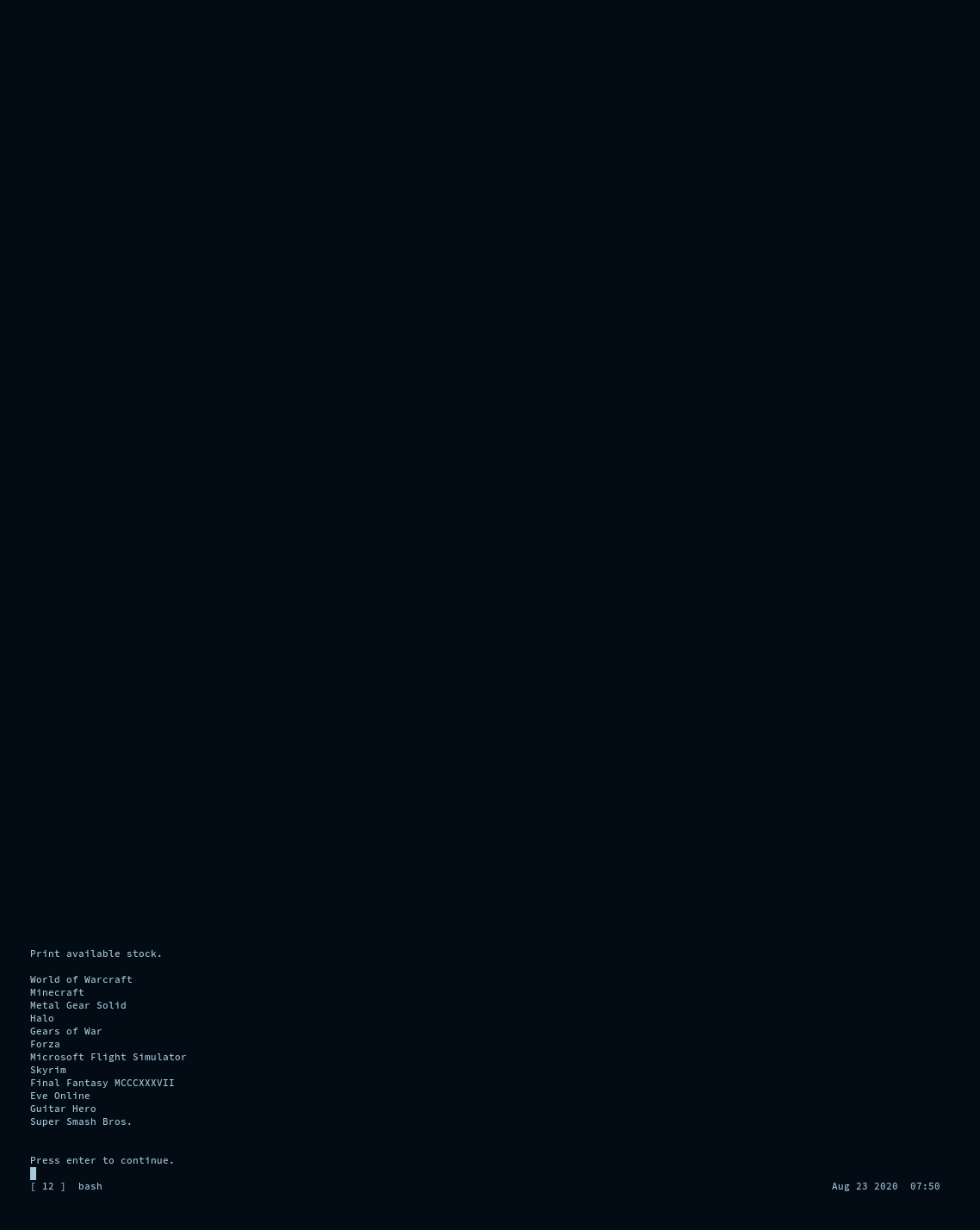
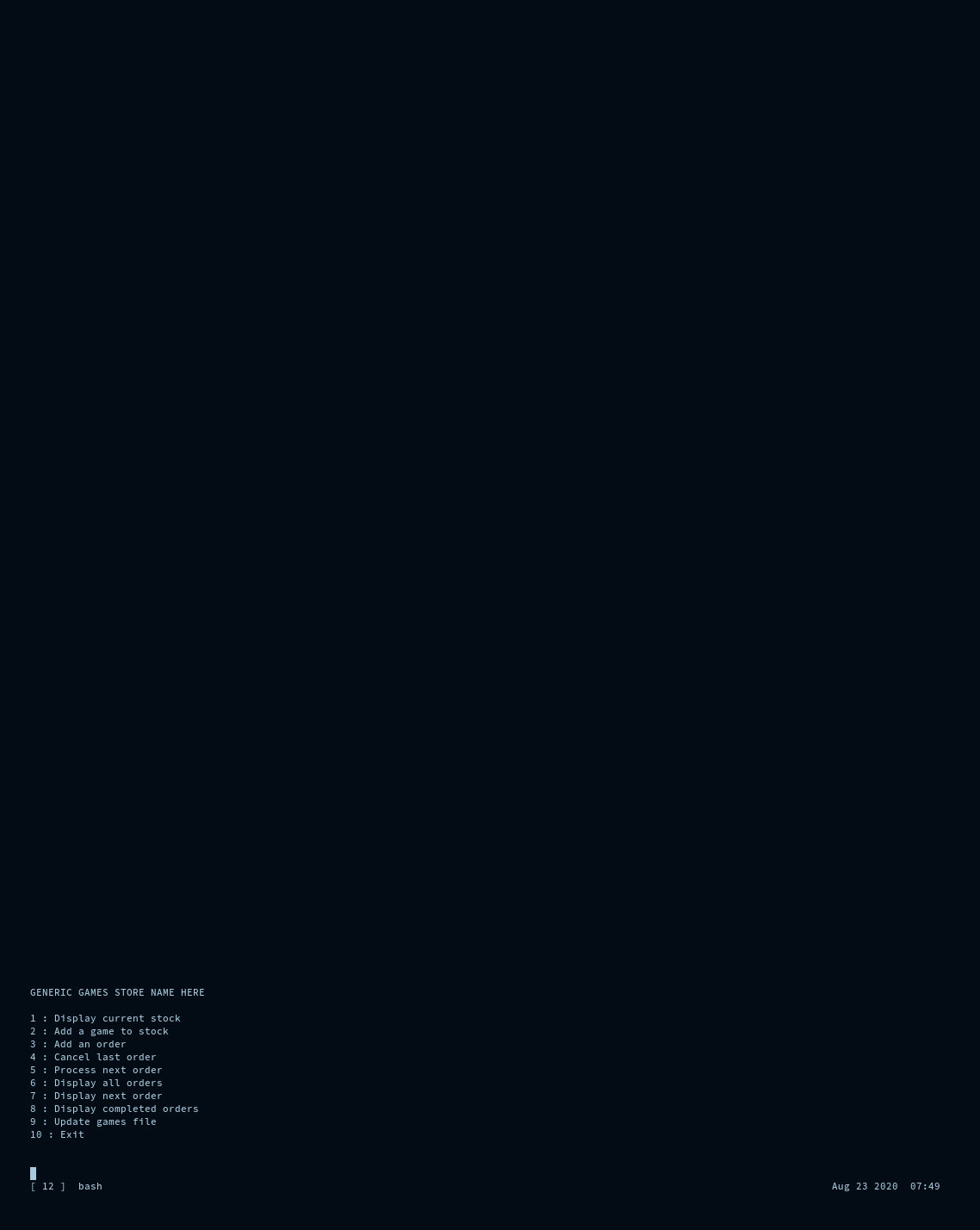
// This MUST be NULL so that ll\_free doesn't free the whole list.

pointernode->next = NULL;

ll\_free(&pointernode);

return removed;

}



Question 2

// Question2.c

#include <time.h>

#include <stdlib.h>

#include <stdio.h>

#include <string.h>

#include <stdbool.h>

#include "Question2.h"

int main(int argc, char \*argv[]) {

srandom(time(NULL));

size\_t student\_marks\_array[50];

for(int i = 0; 50 > i; i++) {

student\_marks\_array[i] = random() % 101;

}

size\_t marks[50];

void reset\_students(void) {

memcpy(marks, student\_marks\_array, sizeof(marks));

}

reset\_students();

void print\_students(size\_t min, size\_t max) {

for(int i = 0, ticker = 0; 50 > i; i++) {

if(marks[i] >= min && marks[i] < max) {

printf("Mark %d : %d\t", i + 1, marks[i]);

// Newline every 5 columns.

if((++ticker) % 5 == 0) {printf("\n");}

}

}

printf("\n");

}

bool menu(void) {

clear();

printf( "STUDENT MARKS\n\n"

"1\t:\tShow original order of marks\n"

"2\t:\tBubble Sort\n"

"3\t:\tSelection Sort\n"

"4\t:\tInsertion Sort\n"

"5\t:\tAll Sorting Methods\n"

"6\t:\tGenerate Report\n"

"10\t:\tExit\n\n");

switch(user\_input\_int()) {

case 1:

// No sort.

printf("Original order:\n");

reset\_students();

print\_students(0, 100);

break;

case 2:

// Bubble.

printf("Bubble sort:\n");

reset\_students();

memcpy(marks, bubble\_sort(marks), sizeof(marks));

print\_students(0, 100);

break;

case 3:

// Selection.

printf("Selection sort:\n");

reset\_students();

memcpy(marks, selection\_sort(marks), sizeof(marks));

print\_students(0, 100);

break;

case 4:

// Insertion.

printf("Insertion sort:\n");

reset\_students();

memcpy(marks, insertion\_sort(marks), sizeof(marks));

print\_students(0, 100);

break;

case 5:

// All sorting methods.

printf("All sorting methods:\n");

// Unsorted

printf("Unsorted\n");

reset\_students();

print\_students(0, 100);

// Bubble sort

printf("Bubble\n");

reset\_students();

memcpy(marks, bubble\_sort(marks), sizeof(marks));

print\_students(0, 100);

// Selection sort

printf("Selection\n");

reset\_students();

memcpy(marks, selection\_sort(marks), sizeof(marks));

print\_students(0, 100);

// Insertion sort

printf("Insertion\n");

reset\_students();

memcpy(marks, insertion\_sort(marks), sizeof(marks));

print\_students(0, 100);

break;

case 6:

// Report

reset\_students();

memcpy(marks, insertion\_sort(marks), sizeof(marks));

printf( "Generate a report for which group?"

"\n"

"\n"

"1 HD\n"

"2 D\n"

"3 C\n"

"4 P\n"

"5 N\n"

"\n");

switch(user\_input\_int()) {

case 1:

printf("\nHD (80 - 100)\n");

print\_students(80, 100);

break;

case 2:

printf("\nD\t(70 - 80)\n");

print\_students(70, 80);

break;

case 3:

printf("\nC\t(60 - 70)\n");

print\_students(60, 70);

break;

case 4:

printf("\nP\t(50 - 60)\n");

print\_students(50, 60);

break;

case 5:

printf("\nN\t (0 - 50)\n");

print\_students(0, 50);

break;

default:

break;

}

break;

case 10:

return false;

default:

return true;

}

printf("Press enter to continue...");

user\_input\_int();

printf("\n\n\n\n");

}

while(menu()) {}

return 0;

}

// Question2.h

int user\_input\_int(void) {

char input[100];

fflush(stdin);

fgets(input, 99, stdin);

return atoi(input);

}

void clear(void) {for(int i = 0; i < 100; i++) {printf("\n");}}

size\_t \* bubble\_sort(size\_t m[]) {

for(int i = 50; 0 <= i; i--) {

for(int j = i; 50 > j + 1; j++) {

if(m[j] > m[j + 1]) {

// Switch them.

size\_t temp = m[j];

m[j] = m[j + 1];

m[j + 1] = temp;

}

}

}

return m;

}

size\_t \* selection\_sort(size\_t m[]) {

for(int i = 0; 50 > i; i++) {

int smol = i;

for(int j = i; 50 > j; j++) {

if(m[j] < m[smol]) {

smol = j;

}

}

// Switch them.

size\_t temp = m[smol];

m[smol] = m[i];

m[i] = temp;

}

return m;

}

size\_t \* insertion\_sort(size\_t m[]) {

for(int i = 1; i < 50; i++) {

for(int j = i; j > 0 && m[j - 1] > m[j]; j--) {

size\_t temp = m[j - 1];

m[j - 1] = m[j];

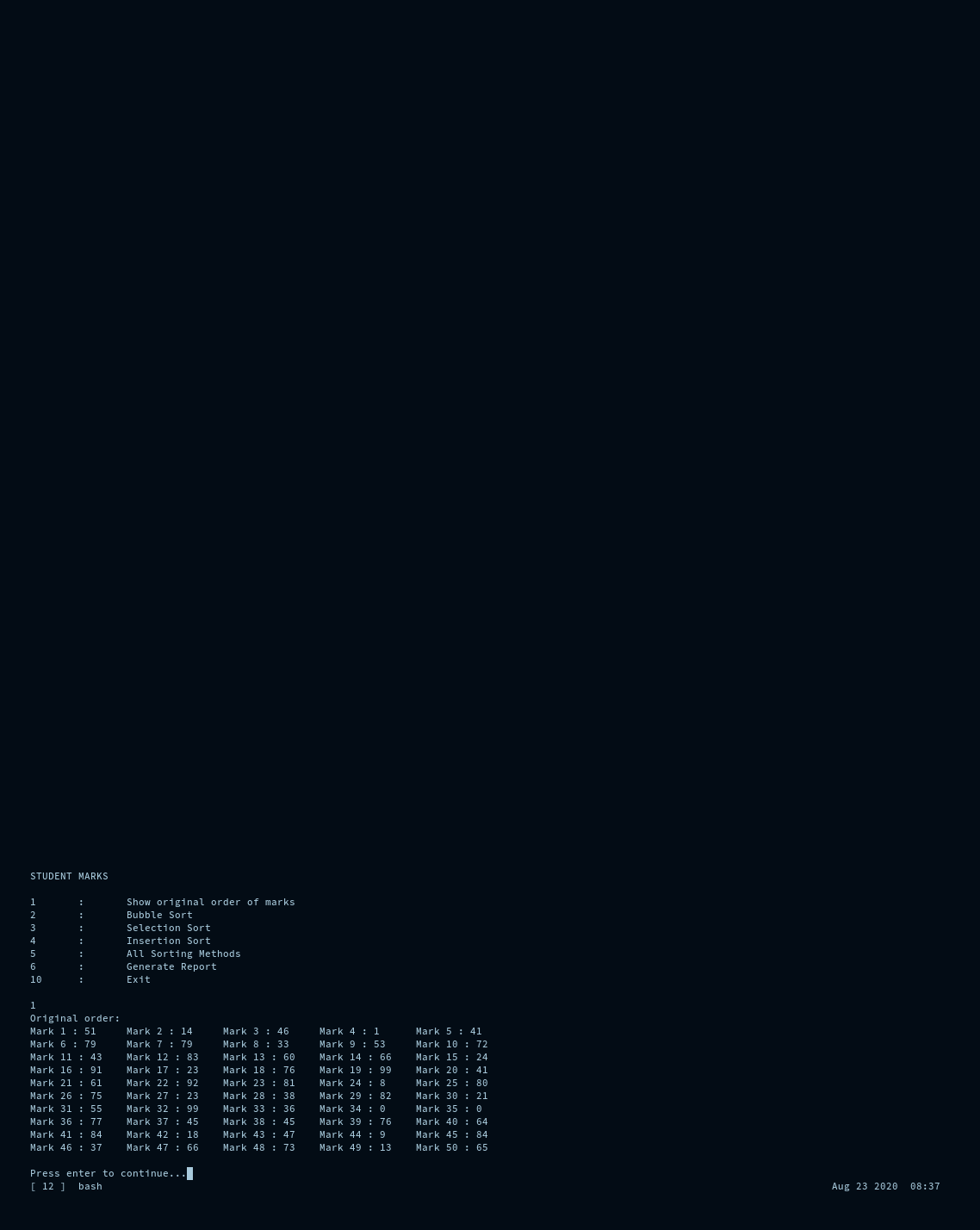
m[j] = temp;

}

}

return m;

}



Question 3

// Question3.c

// basic io

#include <stdio.h>

// atoi

#include <stdlib.h>

// Boolean return from menu

#include <stdbool.h>

// strcpy

#include <string.h>

// random seed

#include <time.h>

// Header file

#include "Question3.h"

int main(int argc, char \*argv[]) {

srandom(time(NULL));

const char namelist[4][50] = {

{"Jerry Seinfeld"},

{"George Costanza"},

{"Cosmo Kramer"},

{"Elaine Benes"},

};

// Newman

// The Soup Nazi

// Lloyd Braun

// Mickey Abbott

// Uncle Leo

// George Steinbrenner

// Make a list of employees.

// Give random ids and salaries to the employees and assign them a name from

// the list of valid names.

employee Employees[10];

// Create 4 fixed employees that never change names, but with random ids and

// salary.

for(int i = 0; i < 4; i++) {

Employees[i].id = random() % 10000 + 1;

strcpy(Employees[i].name, namelist[i]);

Employees[i].salary = random() % 50000 + 40001;

}

// Let the user decide the rest.

for(int i = 4; i < 10; i++) {

Employees[i].id = random() % 10000 + 1;

printf("Input a name for employee #%d: ", Employees[i].id);

char tempname[50];

user\_input\_char(tempname);

strcpy(Employees[i].name, tempname);

Employees[i].salary = random() % 50000 + 40001;

}

// Print an employee's details.

void print\_employee(employee e) {

printf( "ID\t:\t%d\n"

"Name\t:\t%s\n"

"Salary\t:\t$%d\n\n",

e.id,

e.name,

e.salary);

}

// Print all employees' details.

void print\_employees(void) {

for(int i = 0; i < 10; i++) {

print\_employee(Employees[i]);

}

}

// Main menu.

bool menu(void) {

printf( "EMPLOYEE RECORD SEARCH\n"

"\n"

"\n"

"1 : Print all employee details\n"

"2 : Linear search by id\n"

"3 : Binary search by id\n"

"10 : Exit\n"

"\n");

char input[100];

// The function call was too long so I stored the bulk of it in this.

int temp;

switch(user\_input\_int()) {

case 1:

print\_employees();

break;

case 2:

printf( "Which employee id are you linear searching for? ");

temp = linear\_search(

Employees,

sizeof(Employees) / sizeof(\*Employees),

user\_input\_int()

);

if(temp >= 0) {

print\_employee(Employees[temp]);

} else {

printf("Employee not found.\n");

}

break;

case 3:

printf( "Which employee id are you binary searching for? ");

temp = binary\_search(

Employees,

sizeof(Employees) / sizeof(\*Employees),

user\_input\_int()

);

if(temp >= 0) {

print\_employee(Employees[temp]);

} else {

printf("Employee not found.\n");

}

break;

case 10:

return false;

default:

return true;

}

printf("Press enter to continue...\n");

user\_input\_int();

return true;

}

// Sort the array so that the binary search works on it.

for(int i = 1; i < 10; i++) {

for(int j = i; j > 0 && Employees[j - 1].id > Employees[j].id; j--) {

employee temp = Employees[j - 1];

Employees[j - 1] = Employees[j];

Employees[j] = temp;

}

}

// Loop the menu.

while(menu()) {}

printf("Exiting...\n");

return 0;

}

// Question3.h

typedef struct Employee {

size\_t id;

char name[50];

size\_t salary;

}employee;

int user\_input\_int(void) {

char input[100];

fflush(stdin);

fgets(input, 99, stdin);

return atoi(input);

}

void user\_input\_char(char \* input) {

fflush(stdin);

fgets(input, 49, stdin);

for(int i = 0; input[i] != '\0' && i < 50; i++) {

if(input[i] == '\n') {

input[i] = '\0';

}

}

}

int linear\_search\_name(employee \* e, const size\_t e\_len, char \* s) {

for(int i = 0; i < e\_len; i++) {

// Compare the string to see if it is a substring.

if(strstr(e[i].name, s) != NULL) {

return i;

}

// Compare the string to see if it's the exact same string.

if(0 == strcmp(e[i].name, s)) {

return i;

}

}

return -1;

}

int linear\_search(employee \* e, const size\_t e\_len, int s) {

// The sensible way to do this.

/\*

for(int i = 0; i < e\_len; i++) {

if(e[i].id == s) {

return i;

}

}

return -1;

\*/

// How I have been asked to do this for some reason.

int chunk(int index) {

if(index == e\_len) {return -1;}

if(e[index].id == s) {

return index;

} else {

chunk(++index);

}

}

return chunk(0);

}

int binary\_search(employee \* e, const size\_t e\_len, int s) {

int divide(employee \* e, int s, size\_t left, size\_t right) {

int midpoint = (left + right) / 2;

// Check if the search term has been found.

if(e[midpoint].id == s) {return midpoint;}

// Check if the search term can not be found.

if(left >= right) {return -1;}

// Search the correct side of the array.

if(e[midpoint].id > s) {

// midpoint is greater than value we're looking for.

return divide(e, s, left, midpoint);

} else {

// midpoint is lesser than value we're searching for.

return divide(e, s, midpoint, right);

}

}

return divide(e, s, 0, e\_len - 1);

}

The linear search checks each element in the array one at a time until it gets a

match and then returns.

It is extremely easy to implement since it only requires a loop and an if

statement.

The binary search checks the midpoint then goes in the direction of the match

assuming it will be there.

This assumes that the array is sorted already.

It repeats this process for smaller and smaller chunks till the chunk size is 1.

Once the chunk size is 1 if the match isn't found it returns an error.

If found, it returns the location of the match.

This is far more efficient as it halves the search area each time instead of

searching the entire array one element at a time.

The binary search is much faster but it requires that the array is already

sorted, which means that it may not be faster in some cases since it may be

difficult or impossible to sort.

When the search term is impossible to sort, such as when you're searching by

name then the linear search is your best option.

When you're searching an ordered list of items then the binary search will cut

the search time down in all but the rarest of cases.

