#include <stdio.h>

#include <stdbool.h>

#include <stdlib.h>

#include <unistd.h>

#include <string.h>

int main(void) {

/\* Constants \*/

/\* Timing related constants because I'm lazy. \*/

const int t1s = 1000000;

const int t100ms = 100000;

const int t10ms = 10000;

const int t1ms = 1000;

/\* Master variable for character print delay, decrease this value to make

\* testing faster. \*/

/\* Good delay. \*

const int cpd = 25000;

/\*\*/

/\* Testing delay. \*/

const int cpd = 100;

/\*\*/

/\* The multiplier for how much slower a newline character is printed. \*/

const int newlinemultiplier = 5;

/\* Menu newline scroll speed. \*/

const int menuscroll = t10ms;

/\* Variables \*/

/\* Area to store text to be printed to the screen.

\* (It's huge because I'm extremely lazy.) \*/

char print[1000];

/\* Initialize receipt number to a number that looks random.

\* Randomised receipt numbers could be added later to prevent customers

\* guessing sequential numbers and cancelling each other's orders. \*/

int receiptNo = 784584;

/\* Structures \*/

/\* Struct to describe products. \*/

/\* Create default burgers to copy properties from. \*/

struct TastyBurger {

char name[30];

float price;

size\_t quantity;

char description[100];

int minweight\_grams;

int minquantity\_ml;

};

/\* Individual products. \*/

const struct TastyBurger Burgers[4] = {

{"Original Burger", 12.00, 0,

"The original \"TastyBurger\"", 0, 0},

{"Chicken Burger", 15.00, 0,

"The original but with chicken.", 0, 0},

{"Ham Burger", 17.00, 0,

"Original patented Skinnerburger, mmmm... Steamed hams.", 0, 0},

{"Vegie Burger", 13.00, 0,

"This is the food that my food eats.", 0, 0}

};

const struct TastyBurger Sides[2] = {

{"Chips (Regular)", 5.00, 0,

"Hot chips served with herbed mayo.", 200, 0},

{"Chips (Large)", 7.50, 0,

"Hot chips served with herbed mayo.", 500, 0}

};

const struct TastyBurger Drinks[2] = {

{"Cole", 3.50, 0, "Bepsi is better.", 0, 500},

{"Bepsi", 3.50, 0, "Cole is better.", 0, 500}

};

/\* This is not edited but the copies inside of the meals are. \*/

const struct TastyBurger MealDiscount[4] = {

{"Original meal", 19.00, 1,

"One original TastyBurger with chips and a drink.", 0, 0},

{"Chicken meal", 20.00, 1,

"One chicken TastyBurger with chips and a drink.", 0, 0},

{"Ham meal", 22.00, 1,

"One ham TastyBurger with chips and a drink.", 0, 0},

{"Vegie meal", 18.00, 1,

"One vegie TastyBurger with chips and a drink.", 0, 0}

};

struct TastyBurger OriginalMeal[] = {

Burgers[0],

Sides[0],

Drinks[0],

MealDiscount[0]

};

struct TastyBurger ChickenMeal[] = {

Burgers[1],

Sides[0],

Drinks[0],

MealDiscount[1]

};

struct TastyBurger HamMeal[] = {

Burgers[2],

Sides[0],

Drinks[1],

MealDiscount[2]

};

struct TastyBurger VegieMeal[] = {

Burgers[3],

Sides[0],

Drinks[1],

MealDiscount[3]

};

struct MealsArray {

size\_t mealptr\_len;

struct TastyBurger \* mealptr;

} Meals[] = {

{sizeof(OriginalMeal) / sizeof(\*OriginalMeal), OriginalMeal},

{sizeof(ChickenMeal) / sizeof(\*ChickenMeal), ChickenMeal},

{sizeof(HamMeal) / sizeof(\*HamMeal), HamMeal},

{sizeof(VegieMeal) / sizeof(\*VegieMeal), VegieMeal}

};

/\* This block of code is to set the price of the meal to a negative number

\* so that when it's added to the receipt it sets the price to the meal

\* deal's price instead of the sum of the parts. I could have set this

\* manually but I'm too lazy to do that. \*/

float calcMealPrice(const struct TastyBurger m[], size\_t s) {

float price = 0.00;

/\* When this loops through the items in the meal it will skip the last

\* one because that is the one we're taking the correct price from. \*/

for(int i = 0; i < s - 1; i++) {

price += m[i].price;

}

price -= m[s - 1].price;

return price \* -1;

}

/\* Iterate through all of the Meals and calculate the price to subtract from

\* the sum in order to discount the meal to the correct price which is

\* stored in MealDiscount[i]. \*/

for(int i = 0; i < sizeof(Meals) / sizeof(\*Meals); i++) {

Meals[i].mealptr[Meals[i].mealptr\_len - 1].price =

calcMealPrice(Meals[i].mealptr, sizeof(Meals) / sizeof(\*Meals));

}

/\* Length of the Orders struct. \*/

size\_t Orders\_len = 0;

/\* Customer order details struct.

\* This contains an array of burgers that they have ordered. \*/

struct CustomerOrder {

int orderID;

float totalPrice;

size\_t contents\_len;

struct TastyBurger \* contents;

} \* Orders = (struct CustomerOrder \*) malloc(

Orders\_len \* sizeof(struct CustomerOrder)

);

/\* Functions \*/

/\* Allocate more memory for a new entry in the customer orders array. \*/

/\* This function needs some kind of error handling for when the memory isn't

\* allocated. \*/

struct CustomerOrder \* newOrder(void) {

/\* Reallocate memory. \*/

Orders\_len++;

Orders = (struct CustomerOrder \*) realloc(Orders,

Orders\_len \* sizeof(struct CustomerOrder));

/\* Make sure the receipt number is incrememted.

\* To avoid contiguous values a loop could be added here to check if the

\* receipt number is unique after randomizing a new number. \*/

receiptNo++;

/\* Make sure all of the values are set to 0 by default except the

\* receipt number. \*/

Orders[Orders\_len - 1].orderID = receiptNo;

Orders[Orders\_len - 1].totalPrice = 0.00;

Orders[Orders\_len - 1].contents\_len = 0;

/\* It will initially be of size 0 since contents\_len is 0. \*/

Orders[Orders\_len - 1].contents = (struct TastyBurger \*) malloc(

Orders[Orders\_len - 1].contents\_len \* sizeof(struct TastyBurger));

return &Orders[Orders\_len - 1];

}

/\* Print each character individually a bit like a dot matrix printer. \*/

void scrollprint(size\_t size, int delay, char printme[]) {

/\* Initial fflush(stdout) is to clear out any printf from outside this

\* function. \*/

fflush(stdout);

for(int i = 0;

i < size && printme[i] != '\0';

i++) {

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

usleep(delay \* newlinemultiplier);

} else {

usleep(delay);

}

printf("%c", printme[i]);

fflush(stdout);

}

}

/\* Print newlines with a delay, the delay is before each newline. \*/

void newlines(int n, int t) {

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

usleep(t);

printf("\n");

}

}

/\* Print a carriage return (\n)

\* Note that this clears the contents of print and replaces it with '\n'. \*/

void cret(void) {

strcpy(print, "\n");

scrollprint(sizeof(print) / sizeof(\*print), cpd, print);

}

/\* Take user input, return an int.

\* This can be used to wait for a user to press the enter key. \*/

int integerInput(void) {

char input[20];

fflush(stdin);

fgets(input, 10, stdin);

return atoi(input);

}

void cancelOrder(struct CustomerOrder \* order\_to\_delete) {

/\* We need to make a copy of the array and shift the contents of the

\* copy down to compensate for the empty slot we're about to create

\* then realloc the original to be smaller and restore the contents

\* of the array.

\* These are the variables inside the struct for reference.

\* int orderID

\* fload totalPrice

\* size\_t contents\_len

\* struct TastyBurger \* contents (NEEDS TO BE FREED SEPARATELY)

\*

\* Free the backup

\*/

/\* Set i to the index of the Order we want to delete. \*/

int i;

for(i = 0;

i < sizeof(Orders) / sizeof(\*Orders) &&

&Orders[i] != order\_to\_delete;

i++) {}

/\* Make a copy of the Orders struct that's one smaller. \*/

struct CustomerOrder \* OrdersBackup = (struct CustomerOrder \*) malloc(

(Orders\_len - 1) \* sizeof(struct CustomerOrder)

);

if(OrdersBackup != NULL) {

/\* Copy contents of Orders into OrdersBackup. \*/

/\* Before the index. \*/

if(i != 0) {

memcpy(OrdersBackup, Orders, i \* sizeof(struct CustomerOrder));

}

/\* After the index. \*/

if(i != (Orders\_len - 1)) {

memcpy( OrdersBackup + i,

Orders + i + 1,

(Orders\_len - i - 1) \* sizeof(struct CustomerOrder));

}

/\* Free the contents of the order that is being removed. \*/

free(Orders[i].contents);

/\* Decrement the size of Orders. \*/

Orders\_len--;

/\* realloc Orders \*/

Orders = (struct CustomerOrder \*) realloc(Orders,

Orders\_len \* sizeof(struct CustomerOrder));

/\*\*/

/\* Restore Orders from backup. \*/

memcpy( Orders,

OrdersBackup,

Orders\_len \* sizeof(struct CustomerOrder));

/\*\*/

/\*

free(Orders);

Orders = OrdersBackup;

/\*\*/

/\* Free the backup now that we're finished with it. \*/

free(OrdersBackup);

strcpy(print, "Order cancelled.\n");

scrollprint(sizeof(print) / sizeof(\*print), cpd, print);

} else {

strcpy(print, "Failed to cancel order.");

scrollprint(sizeof(print) / sizeof(\*print), cpd, print);

}

}

/\* Remove orders from the Orders array. \*/

/\* Don't run this function with an empty array, it will segfault. \*/

void cancelOrderUI(void) {

/\* int i is in this scope because it needs to be accessed from outside

\* of the for loop. \*/

int userInputReceiptNumber = 0;

strcpy(print, "Please enter in your receipt number: ");

scrollprint(sizeof(print) / sizeof(\*print), cpd, print);

userInputReceiptNumber = integerInput();

/\* Iterate through the orders to find an order number with the same

\* value.

\* The loop itself doesn't need any actual content, as we only care

\* about the value of i which we retain for later. \*/

int i;

for(i = 0;

i < Orders\_len &&

Orders[i].orderID != userInputReceiptNumber;

i++) {

}

/\* Check if the number is equal in case the loop terminated at the

\* end of the array. \*/

if(Orders\_len > i && Orders[i].orderID == userInputReceiptNumber) {

cancelOrder(&Orders[i]);

} else {

/\* Invalid input. (Incorrect number) \*/

strcpy(print, "Order not found.\n");

scrollprint(sizeof(print) / sizeof(\*print), cpd, print);

}

}

void order( struct CustomerOrder \* ord,

const struct TastyBurger product,

int quant) {

/\* This is where we make entries into our order order struct. \*/

ord->contents\_len++;

ord->contents = (struct TastyBurger \*) realloc(ord->contents,

ord->contents\_len \* sizeof(struct TastyBurger));

/\* Check if memory allocation actually worked. \*/

if(ord->contents) {

ord->contents[ord->contents\_len - 1] = product;

ord->contents[ord->contents\_len - 1].quantity = quant;

strcpy(print, product.name);

strcat(print, " added to order.\n");

scrollprint(sizeof(print) / sizeof(\*print), cpd, print);

} else {

/\* Some kind of error handling for when adding the

\* memory didn't work would be a good idea. \*/

printf( "Error: Something went wrong and you're seeing "

"an error allocating memory that I never\n"

"expected to encounter because I have too much "

"RAM.\n");

}

}

/\* Add meals (Grouped items) to the order. \*/

void orderMeal( struct CustomerOrder \* ord,

const struct MealsArray m[],

size\_t s) {

strcpy(print, "\nSelect which meal you would like to order:\n");

scrollprint(sizeof(print) / sizeof(\*print), cpd, print);

/\* Print the names and prices of the items. \*/

int index;

int meal\_item;

size\_t si;

for( index = 0;

index < s;

index++) {

/\* Track the size of the array we're currently looking at. \*/

si = m[index].mealptr\_len;

printf("%d", index + 1);

strcpy(print, " - ");

strcat(print, m[index].mealptr[si - 1].name);

strcat(print, " - $");

scrollprint(sizeof(print) / sizeof(\*print), cpd, print);

printf("%.2f", MealDiscount[index].price);

strcpy(print, "\n ");

strcat(print, m[index].mealptr[si - 1].description);

strcat(print, "\n");

scrollprint(sizeof(print) / sizeof(\*print), cpd, print);

for( meal\_item = 0;

meal\_item < m[index].mealptr\_len - 1;

meal\_item++) {

strcpy(print, " - ");

strcat(print, m[index].mealptr[meal\_item].name);

strcat(print, "\n");

scrollprint(sizeof(print) / sizeof(\*print), cpd, print);

}

cret();

}

/\* Print the cancel option. \*/

printf("%d", index + 1);

strcpy(print, " - Cancel\n");

scrollprint(sizeof(print) / sizeof(\*print), cpd, print);

/\* User inputs which item they wanted. \*/

index = integerInput();

if(index > s || index < 0) {

/\* They selected something other than a item number,

\* exit this sequence. \*/

strcpy(print, "Returning...\n");

scrollprint(sizeof(print) / sizeof(\*print), cpd, print);

} else {

index--;

si = m[index].mealptr\_len;

/\* Valid input, continue ordering. \*/

strcpy(print, "You have selected ");

strcat(print, m[index].mealptr[si - 1].name);

strcat(print, ".\n");

strcat(print, "How many would you like to order?\n");

scrollprint(sizeof(print) / sizeof(\*print), cpd, print);

/\* User inputs the quantity of items they would like to order. \*/

size\_t n = integerInput();

/\* Iterate through all of the items in the meal and add them one by

\* one. \*/

if(n > 0) {

/\* Add the item to the order. \*/

si = m[index].mealptr\_len;

for(meal\_item = 0; meal\_item < si; meal\_item++) {

order(ord, m[index].mealptr[meal\_item], n);

}

} else {

strcpy(print, "You have entered an invalid number of items.\n");

scrollprint(sizeof(print) / sizeof(\*print), cpd, print);

}

}

}

/\* Add individual items to the order. \*/

void orderItem( struct CustomerOrder \* ord,

const struct TastyBurger prod[],

size\_t s) {

strcpy(print, "\nSelect which item you would like to order:\n");

scrollprint(sizeof(print) / sizeof(\*print), cpd, print);

/\* Print the names and prices of the items. \*/

int index;

for(index = 0;

index < s;

index++) {

printf("%d", index + 1);

strcpy(print, " - ");

strcat(print, prod[index].name);

strcat(print, " - ");

strcat(print, prod[index].description);

strcat(print, " - $");

scrollprint(sizeof(print) / sizeof(\*print), cpd, print);

printf("%.2f", prod[index].price);

cret();

}

/\* Print the cancel option. \*/

printf("%d", index + 1);

strcpy(print, " - Cancel\n");

scrollprint(sizeof(print) / sizeof(\*print), cpd, print);

/\* User inputs which item they wanted. \*/

index = integerInput();

if(index > s || index < 0) {

/\* They selected something other than a item number,

\* exit this sequence. \*/

strcpy(print, "Returning...\n");

scrollprint(sizeof(print) / sizeof(\*print), cpd, print);

} else {

index--;

/\* Valid input, continue ordering. \*/

strcpy(print, "You have selected ");

strcat(print, prod[index].name);

strcat(print, ".\n");

strcat(print, "How many would you like to order?\n");

scrollprint(sizeof(print) / sizeof(\*print), cpd, print);

/\* User inputs the quantity of items they would like to order. \*/

size\_t n = integerInput();

if(n > 0) {

/\* Add the item to the order. \*/

order(ord, prod[index], n);

} else {

strcpy(print, "You have entered an invalid number of items.\n");

scrollprint(sizeof(print) / sizeof(\*print), cpd, print);

}

}

}

void viewOrder(struct CustomerOrder \* ord) {

/\* Calculate the price of the order.

\* Print out the bill for the customer. \*/

/\* For the purposes of keeping all of the prices vertically

\* aligned I'm printing 40 characters in before printing the

\* price at the end of the line.

\* Iterate through the items in the order here. \*/

/\* Keep track of the ammount of items in the order for the purposes of

\* discounting when over 4 products are purchased. \*/

size\_t orderTotalSize = 0;

/\* Reset the ord->totalPrice so that running this function repeatedly

\* doens't inflate it. \*/

ord->totalPrice = 0;

for(int a = 0;

a < ord->contents\_len;

a++) {

/\* Calculate the ammount of spaces needed to keep the '$' in

\* the correct column. \*/

size\_t len = strlen(ord->contents[a].name);

/\* Add 3 for the " x " plus 1 for the number of burgers. \*/

len += 4;

/\* Add one to len for each character the ordersize needs to

\* print. \*/

for(int i = 10; ord->contents[a].quantity >= i; i \*= 10) {

len++;

}

/\* List items \*/

strcpy(print, ord->contents[a].name);

strcat(print, " x ");

scrollprint(sizeof(print) / sizeof(\*print), cpd, print);

printf("%d", ord->contents[a].quantity);

print[0] = '\0';

/\* Print spaces till the 40th character. \*/

for(int i = 0; i < 40 - len; i++) {strcat(print, " ");}

strcat(print, "$");

scrollprint(sizeof(print) / sizeof(\*print), cpd, print);

/\* Print the price at the end of the line. \*/

printf("%.2f", ord->contents[a].price);

strcpy(print, "ea.\n");

scrollprint(sizeof(print) / sizeof(\*print), cpd, print);

/\* Keep track of how many items the customer has bought. \*/

orderTotalSize += ord->contents[a].quantity;

/\* Keep track of the total price of the order. \*/

ord->totalPrice += ord->contents[a].price \*

ord->contents[a].quantity;

}

/\* Price of items \*/

strcpy(print, "Price: $");

scrollprint(sizeof(print) / sizeof(\*print), cpd, print);

printf("%.2f", ord->totalPrice);

cret();

/\* Calculate discounts. \*/

if(ord->totalPrice > 100.00) {

/\* 15% discount for an order over $100. \*/

ord->totalPrice = ord->totalPrice / 20 \* 17;

/\* Print discount on receipt. \*/

strcpy(print, "Discount: 15% $");

scrollprint(sizeof(print) / sizeof(\*print), cpd, print);

printf("%.2f", ord->totalPrice / -17);

cret();

} else if(orderTotalSize > 4) {

/\* 10% discount for an order with more than 4 items. \*/

ord->totalPrice = ord->totalPrice / 10 \* 9;

/\* Print discount on receipt. \*/

strcpy(print, "Discount: 10% $");

scrollprint(sizeof(print) / sizeof(\*print), cpd, print);

printf("%.2f", ord->totalPrice / -9);

cret();

}

/\* Death and taxes. \*/

strcpy(print, "Tax (included): $");

scrollprint(sizeof(print) / sizeof(\*print), cpd, print);

/\* Print tax (included) \*/

printf("%.2f", ord->totalPrice / 10);

cret();

/\* Final bill. \*/

strcpy(print, "Final bill after discounts and tax: $");

scrollprint(sizeof(print) / sizeof(\*print), cpd, print);

printf("%.2f", ord->totalPrice);

cret();

}

/\* Hidden debug menu item to list all orders. \*/

void listOrders(void) {

printf("!!! ORDER LIST DEBUG SCREEN !!!\n");

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

printf("i == %d\n", i);

printf("Orders[%d].orderID : %d\n", i, Orders[i].orderID);

printf("Orders[%d].totalPrice : %.2f\n", i, Orders[i].totalPrice);

printf("Orders[%d].contents\_len : %d\n", i, Orders[i].contents\_len);

for(int a = 0; a < Orders[i].contents\_len; a++) {

printf("Orders[%d].contents[%d]->name : %s\n",

i, a, Orders[i].contents[a].name);

printf("Orders[%d].contents[%d]->price : %.2f\n",

i, a, Orders[i].contents[a].price);

printf("Orders[%d].contents[%d]->quantity : %d\n",

i, a, Orders[i].contents[a].quantity);

}

}

}

/\* Secret admin function to show daily profits. \*/

float showRevenue(char pwd[]) {

const char correct\_password[] = "TastyBurger";

float total\_revenue = 0.00;

/\* Remove the endline character from the end of the string before

\* comparison. \*/

for(int i = 0; i < strlen(pwd); i++) {

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

pwd[i] = '\0';

}

}

/\* Check if the password is correct. \*/

if(strcmp(pwd, correct\_password) == 0) {

/\* Add the profits from all of the orders. \*/

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

total\_revenue = Orders[i].totalPrice;

}

printf("Total revenue today: %.2f\n", total\_revenue);

/\* Return revenue just in case I need it as a return value. \*/

return total\_revenue;

}

}

bool menu(void) {

bool cont = true;

/\* Clear the screen one line at a time to simulate a paper feed. \*/

newlines(100, menuscroll);

/\* Main menu/greeting text. \*/

strcpy(print, "Welcome to Tasty Burgers\n");

strcat(print, "\n");

strcat(print, "Ready to place your order\n");

strcat(print, "1 - Place an order\n");

strcat(print, "2 - Cancel an order\n");

strcat(print, "4 - Exit\n");

scrollprint(sizeof(print) / sizeof(\*print), cpd, print);

/\* This is the pointer to the current order to pass to functions.

\* It is created outside of the switch because you can't create it as

\* the first line of the switch.

\* """

\* error: a label can only be part of a statement and a declaration is

\* not a statement

\* """

\*/

struct CustomerOrder \* o;

char userTextInput[100];

/\* Main menu. \*/

switch(integerInput()) {

case 1:

o = newOrder();

/\* Loop till the order no longer exists. \*/

do {

strcpy(print, "\nWhat would you like to order?\n");

strcat(print, "1 - Meals\n");

strcat(print, "2 - Burgers\n");

strcat(print, "3 - Sides\n");

strcat(print, "4 - Drinks\n");

strcat(print, "5 - View order contents\n");

strcat(print, "6 - Finalize order...\n");

strcat(print, "0 - Cancel order\n");

scrollprint(sizeof(print) / sizeof(\*print), cpd, print);

switch(integerInput()) {

case 1:

/\* Order items as a group. \*/

orderMeal(o, Meals, sizeof(Meals) / sizeof(\*Meals));

break;

case 2:

/\* Order Burgers. \*/

orderItem(o, Burgers, sizeof(Burgers) / sizeof(\*Burgers));

break;

case 3:

/\* Order Sides. \*/

orderItem(o, Sides, sizeof(Sides) / sizeof(\*Sides));

break;

case 4:

/\* Order Drinks. \*/

orderItem(o, Drinks, sizeof(Drinks) / sizeof(\*Drinks));

break;

case 5:

/\* View order contents without creating receipt. \*/

viewOrder(o);

break;

case 6:

/\* Finalize order and create a receipt. \*/

if(o->contents\_len > 0) {

/\* Go to display order contents. \*/

viewOrder(o);

/\* Print receipt information. \*/

strcpy(print, "Please go to a register and make the payment ");

strcat(print, "by quoting the receipt number.\n");

strcat(print, "Your receipt number is: ");

scrollprint(sizeof(print) / sizeof(\*print), cpd, print);

printf("%d", o->orderID);

cret();

/\* Set o to null to break the loop. \*/

o = 0;

} else {

/\* The order is empty, just flush it and go back

\* to the main menu. \*/

strcpy(print, "Order is empty, ");

strcat(print, "returning to main menu.\n");

scrollprint(sizeof(print) / sizeof(\*print), cpd, print);

cancelOrder(o);

}

break;

case 0:

strcpy(print, "Are you sure you want to cancel your order?\n");

strcat(print, "1 - Yes\n");

strcat(print, "2 - No\n");

scrollprint(sizeof(print) / sizeof(\*print), cpd, print);

if(integerInput() == 1) {

/\* Remove the order \*/

cancelOrder(o);

/\* Set o to null to break the loop. \*/

o = 0;

}

break;

default:

break;

}

} while(o);

break;

case 2:

/\* If this function runs with an empty array of orders it will

\* result in a segfault. \*/

if(Orders\_len != 0) {

cancelOrderUI();

} else {

strcpy(print, "No orders to cancel.\n");

scrollprint(sizeof(print) / sizeof(\*print), cpd, print);

}

break;

case 9:

strcpy(userTextInput, "");

printf("Enter administrator password: ");

showRevenue(fgets(userTextInput, 64, stdin));

break;

case 0:

listOrders();

break;

default:cont = false;

}

/\* Don't return to the main menu until the user presses enter. \*/

if(cont == true) {

strcpy(print, "Press enter to continue...\n");

scrollprint(sizeof(print) / sizeof(\*print), cpd, print);

integerInput();

}

return cont;

}

/\* Loop menu till it returns false. \*/

while(menu()) {}

/\* Free any memory reserved to heap here. \*/

/\* Orders[i].contents \*/

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

free(Orders[i].contents);

}

/\* Orders \*/

free(Orders);

strcpy(print, "Exiting");

scrollprint(sizeof(print) / sizeof(\*print), cpd, print);

strcpy(print, "...");

scrollprint(sizeof(print) / sizeof(\*print), t100ms \* 5, print);

usleep(t1s);

printf("\n");

return 0;

}

