CPS393 - Assignment: vending machine

Aims

- To raise awareness that software development is often a commercial (paid) competitive activity, and the importance of designing for reducing software maintenance costs.
- To see value (or problems) from the client's perspective.
- To demonstrate your problem solving abilities using the C programming language.

Assignment Description

Write a C program called "pop" which controls the operation of a soft drink vending machine.

As we do not have a real physical pop machine on which to run your code, you will simulate the inputs and outputs using the keyboard and screen.

	Actual	**Simulated**
Inputs	• signals from coin detector • key switch • internal buttons to set price	• keyboard input
Outputs	• LCD display (price, amount tendered, amount owing, etc.) • relay signal to dispense pop • relay signals to dispense change	• messages on screen

There are two modes of operation: Maintenance and Customer. For Maintenance, think of the grey-uniformed person who has that special key to open the machine, set the price, re-stock, etc. In Customer mode, the machine is waiting for thirsty people to walk up and buy a pop. It needs to process sales.

Maintenance & Service Requirements

- set the selling price in centimes (range 30 centimes to $1.05) from a command line argument (Assume valid integer on command line.)
- provide a specific message (as in sample below) for the following error cases and return to the operating system:
    - command line argument is missing
    - prices out of range
    - price not a multiple of 5 (**NB Don't use nickel denomination**. Expect that the multiple might change. Marketing might say, "We like our price to be a multiple of 15.")
- with a valid selling price, put machine in-service for continual sales (See Customer Requirements)
- as a maintenance action (hidden from menu), allow service person to shutdown (exit) the program altogether at the coin prompt by pressing E or e; refund any pending amount

Customer Requirements

- First assume that there is only one flavour of pop, and that the machine won't run out of pop (no need to track inventory or total sales)
- display a welcome message stating the pop price, and what coins and commands are accepted
- prompt the user to insert coins (via keyboard entry)
- accept a nickel [N or n] (worth 5), dime [D or d] (worth 10) or pente [P or p] (worth 20); reject all other input except maintenance actions, then re-prompt for coin
- after each coin, display how much the user has inserted in total and how much more the user needs to insert
- you can assume coins will be fed to your program one-at-a-time (equivalent keyboard input would be one coin character ("n") at a time and never be "ndj"
- dispense product on collecting sufficient money (only one flavour)
- provide change
    - for overpayment or
    - if the customer aborts the transaction by pressing coin return (R or r).
- refund using only dimes and nickels (assume machine won't run out) using the fewest number of coins.
- always display the change message at the end of each transaction even if there is no change to return ("Change given: 0 centimes as 0 dime(s) and 0 nickel(s).")
- re-display selling price before new sale
- continually prompt for additional sales -- do not exit the program
- match your output with the dialog shown below

Example sequence (not exhaustive testing). (NB: The dollar sign ($) represents the command prompt)

```
$ ./pop
Please specify selling price as a command line argument.
Usage: pop [price]
$ ./pop 225
Price must be from 30 to 105 centimes inclusive
$ ./pop 86
Price must be a multiple of 5.
$ ./pop 40
Welcome to my C Pop Machine!
Pop is 40 centimes. Please insert any combination of nickels [N
or n], dimes [D or d] or Pentes [P or p]. You can also press R
or r for coin return.
Enter coin (NDPR): n
  Nickel detected.
    You have inserted a total of 5 centimes.
    Please insert 35 more centimes.
Enter coin (NDPR): d
  Dime detected.
    You have inserted a total of 15 centimes.
    Please insert 25 more centimes.
Enter coin (NDPR): P
  Pente detected
    You have inserted a total of 35 centimes.
    Please insert 5 more centimes.
Enter coin (NDPR): j
  Unknown coin rejected.
    You have inserted a total of 35 centimes.
    Please insert 5 more centimes.
Enter coin (NDPR): p
  Pente detected.
    You have inserted a total of 55 centimes.
    Pop is dispensed. Thank you for your business! Please come
again.
    Change given: 15 centimes as 1 dime(s) and 1 nickel(s).
Pop is 40 centimes. Please insert any combination of nickels [N
or n], dimes [D or d] or Pentes [P or p]. You can also press R
or r for coin return.
Enter coin (NDPR): p
  Pente detected.
    You have inserted a total of 20 centimes.
    Please insert 20 more centimes.
```

```
Enter coin (NDPR): R
    Change given: 20 centimes as 2 dime(s) and 0 nickel(s).
Pop is 40 centimes. Please insert any combination of nickels [N
or n], dimes [D or d] or Pentes [P or p]. You can also press R
or r for coin return.
Enter coin (NDPR): n
  Nickel detected.
    You have inserted a total of 5 centimes.
    Please insert 35 more centimes.
Enter coin (NDPR): n
  Nickel detected.
    You have inserted a total of 10 centimes.
    Please insert 30 more centimes.
Enter coin (NDPR): E
    Change given: 10 centimes as 1 dime(s) and 0 nickel(s).
Shutting down. Goodbye.
$
```

Implementation Requirements

More money is spent on on-going software maintenance than the initial program development. An organisation can be more competitive by reducing or avoiding unnecessary labour in software maintenance at the design stage. One significant but easy way is to avoid putting in hard-coded constants within one's code. There are preferred ways to minimise the extent of code changes when these constants need updating. Follow the **Single Point of Truth Principle** (which you should know from CCPS 209 sec 3.2). See examples and explanations below.

All constants (any value which should not change during program execution) must be specified with a #define or const qualifier. Also, comments must match program behaviour. Never have hard-coded constants embedded within your code!

Bad	Good

`if (temperature < 21 )`
`  printf("Too cold! ");`
`  printf("I prefer at least 21 degrees");`
`/* checks if temp is less than 21 */`
	`#define COMFORT 21`
	`...`
	`if (temperature < COMFORT)`
	`  printf("Too cold! ");`
	`  printf("I prefer at least %d degrees",COMFORT)`
	`/* checks if temp is less than COMFORT */`

`char userinput[5];`	`#define MAXBUFFER 5`
	`…`
	`char userinput[MAXBUFFER];`

- Choose self-describing names as you would variables. If a constant changes, there should be one and only one place that needs to be updated.

Bad	Good

`#define THIRTEEN 13`
`...`
`gross_amt= THIRTEEN * base;`
`...`
`/* current tax rate is THIRTEEN */`
 	
What if the tax rate becomes 15?
	`#define TAXRATE 13`
	`...`
	`gross_amt= TAXRATE * base;`
	`...`

`#define EIGHT 8`
`...`
`n cookies=n box * EIGHT;`
	`#define COOKIES_PER_BOX 8`
	`...`
	`n_cookies=n_box * COOKIES_PER_BOX;`

- Constants must not be stored in a program variable because a variable can be changed (unless using a const qualifier to make variable read-only).

Bad	Good

`int maxheight = 50;`
`...`
`if (height > maxheight)`
`  printf("Too high\n");`
`...`
`maxheight = 75; /* oh, oh */`
`...`
`printf("Maximum allowable height is %d", maxheight);`
`/* max height is 50; print 50 */`
	`#define MAXHEIGHT 50`
	`...`
	
	`if (height > MAXHEIGHT )`
	`  printf("Too high\n");`
	`...`
	`MAXHEIGHT = 75; /* syntax error, automatic error prevention */`

- If program requirements change (e.g. price limit), and you would have to change a value in more than one place anywhere in your program, this is a red flag that you should be using a #define with the symbol in each instance (e.g. PI not 3.142).

Bad	Good

`circum = 2 * 3.142 * radius;`
`area = 3.142 * radius * radius;`
	`#define PI 3.142`
	`...`
	
	`circum = 2 * PI * radius;`
	`area = PI * radius * radius;`

- For this assignment only, it is acceptable (i.e. no penalty) for hard-coding the coin denomination values. Why should you think of this? Consider ease of deployment to a

country with a different coin set (e.g. 20, 50, 100).

Okay, not great	Flexible and extendible

`if (coin == 5)`
`  tender = tender+5;`
	`#define COIN1 5`
	`...`
	
	`if (coin == COIN1)`
	`tender = tender+COIN1;`

- **Avoid duplication of code**. Consider the (similarity of) computation for returning change and providing a refund. If Pentes ever needed to be added to the change denominations, it should not require code changes in more than one place.

Notes on typical problems:

Q: I am having problems with scanf/getchar; I have to hit enter twice, or it skips input.
A1: If you use scanf, you need to put a space character inside the quotes before the placeholder, like this:

```
#include <stdio.h>
int main (void) {
char command;
do
{
   printf("Enter a command (X to exit): ");
   scanf(" %c", &command); /* note SPACE before %c
*/
   printf("You entered: %c\n", command);
}
while (command != 'X');
printf("Goodbye\n");
return (0);
}
```

A2: If you use getchar (or scanf("%c")), remember that this reads only a single character, but your program does not see it until the user presses <enter> adding the newline character. You have to add another getchar (or scanf("%c")) after each getchar to "swallow" the newline like this:

```
letter=getchar();   /* OR scanf("%c");  no space before %c */
while (getchar() != '\n'); /* eat newline */
```

Q: I am getting a "Segmentation Fault" error message.

A: This is almost always caused by using a pointer (dereferencing) which is not initialized. Hint: Even if you have not declared a pointer in your code, you may be using a data structure which contains pointers. Try putting some printf statements to "sandwich" the problem and identify which line of code is triggering the Segmentation Fault.

Submission Instructions

Please refer to the submission instructions in D2L: Content -> Assignments -> Submission instructions

---

Keep reading … scroll down.

Grading Scheme

Guiding Principles

This assignment emphasises future software maintenance costs on par with functionality, hence the attention to embedded hard-coded constants and code duplication (see below).
For functionality, imagine being the owner of pop machines to be deployed at locations across the country: If the code was delivered "as-is" and had a few minor mistakes (e.g. sometimes gave out too much change), how much would you still pay for the (mostly) working program? What will happen when word gets out that if you go to your pop machine, do a sequence of actions, you get free money!?

Tier	Tier Requirements	Mark out of 5
0	● program compiles with no errors ● no run-time errors (e.g. program bombs, Segmentation Fault) ● rudimentary functionality toward solving required problem	1
1	● at most 2 error situations related to: ○ price display ○ amount tendered ○ balance owing ○ unknown coin rejection ○ providing change ○ dispensing pop ○ control flow (e.g. program exits after one sale, sequencing of prompts, etc.) ● at most 3 blocks of code duplication (truth + 2 copies) ● at most 5 hard-coded constants	2
2	● correct product dispensing (right time, right price) ● correct pricing and collection ● correct change ● at most 1 block of code duplication (truth + copy) ● at most 3 hard-coded constants ● proper submission to your remote repository on GitHub	3
3	● at most 1 hard-coded constants ● no code duplication ● no control flow errors ● displays customer instructions to use machine ● at most 2 errors in title block	4
4	Perfect or almost perfect: No more than one issue with ● hard-coded constants (none allowed) ● global variables (none allowed)	5

	• errors in title block • missing citations • self-described variable names • clean-up temp files (if used)	

You need to meet all requirements of a lower tier before being eligible for a higher tier. If you meet all Tier 1 requirements and most of the Tier 2 requirements, you are still at only Tier 1. Even if you had all the Tier 4 requirements met, but did not meet Tier 2, you would still be at Tier 1.

Programs are expected to be free from syntax errors. Programs will be tested on the Ryerson SCS (moon) system. Programs with compile problems are not ready for submission and will receive zero.

Grading Examples

Eg 1

Program does not compile due to syntax error. Result: Tier 0 not met; mark = 0/5.

Eg 2

Program contains 2 hard-coded constants. Everything else is okay. Result: Tier 3 not met; highest Tier met is 2. Mark = 3/5.

Eg 3.

Programs dispenses wrong change in one situation. Everything else is okay. [What do people do to pop machines when they don't get their money back?] Result: Tier 2 not met; highest Tier met is 1. Mark = 2/5.

---

end of document