# Assignment Two: Vending Machine

*Advanced Programming Techniques, Semester 1, 2018*

You have been approached by a company to create a frontend system for a vending machine that should be written in C. The vending machine serves all kinds of pastry treats. The company's name is Penelope's Pastry Delights and the executable you are creating will be called "vm".

In this assignment you are expected to implement an application that will perform the functionalities of a vending machine which are details in this specification. The vending machine will work with a single user at a time.

You will need to demonstrate your understanding and programming ability, with respect to more advanced C programming principles. The concepts covered include:

- Command-line arguments
- File handling
- Dynamic memory allocation and linked lists
- Modularisation and multi-file programs
- Makefiles
- Function pointers
- All concepts covered in Assignment #1

## Submission

| | |
|---|---|
| **Total mark:** | 30% of the final mark + bonus |
| **Assignment demo:** | During tutorial / lab sessions of week 10 (7-11 May) |
| **Final submission:** | End of week 12 – Friday, 25 May 2018 – **10:00 PM** |

Your assignment must compile and execute cleanly on the coreteaching servers with the following commands:

Compile:
```
[s1234567@csitprdap02 ~]$ make
```
Execute:
```
[s1234567@csitprdap02 ~]$ ./vm [command line args]
```

### Startup Code:
The startup source code for vm will be provided on the Blackboard.

You are expected to submit a modularised solution with multiple source files for this assignment. Note the following issues.

- You are not permitted to alter function prototypes. A function prototype includes the name, return type and parameter list of a function.
- You are expected to use ALL the given data structures and functions.
- You may create typedefs for the provided data structures but you may not alter the function prototypes or the predefined types.

Permission to change the startup code is not normally given, unless there is a very good reason to do so (i.e., a bug). You will need to build upon the code that is provided. If you have any concerns about the startup code, please post your query to the Blackboard discussion forum.

Please note that you may find it necessary to modify the #includes for a specific file. This is not considered to be changing the startup code as the data structures and definitions remain the same.

## 1. Requirements
This section describes in detail all the requirements of this assignment. You must read and understand all the requirements in this specification. If something is not clear, you must seek clarification from the teaching team. Your assignment must behave as explained in the following requirements.

### Requirement #1: Command-Line Arguments
A user of your program must be able to execute it by passing in the names of a data file that the system must use. You need to check that the correct number of command-line arguments is entered. The specified name for the data file may or may not be “items.dat”, so your code should not depend on this particular filename or path. We may test your program with a different filename and path and it should still work.

Your program will be run using these command line arguments:
```
[s1234567@csitprdap02 ~]$ ./vm <items_filename>
```

For example:
```
[s1234567@csitprdap02 ~]$ ./vm items.dat
```

Later in the specification (i.e. Requirement #18), you will see that the program needs to support an additional command-line argument. However, the additional argument is necessary only if you decide to implement Requirement #18.

When you are implementing this requirement, and before you have implemented Requirement #18, your program will not have any mechanisms to load the *coin* file into the system. Therefore, before implementing Requirement #18, you will need to have a default coin list that is initialised to zero. This will later be changed and extended to become an actual coin list containing real values which are loaded from file.

## Requirement #2: Load Data

Your program needs to load into memory the data from the file specified in the command-line argument. You will need to tokenise this data before you can load it into the system.

In this requirement, you can safely assume that the specified file is completely valid. Dealing with invalid files is covered in a separate section of the assignment (i.e., Requirement #19).

You will need to use the *structure* definitions provided in the startup code to store your system data.

Item File Format:

```
[ItemID]|[ItemName]|[ItemDesc]|[Price]|[NumberOnHand]
```

Please note that the Price is stored as a set of numbers delimited by the '.' character. The number to the left side of the '.' is the dollars and the number to its right is the cents, for example:

```
I0001|Meat Pie|Yummy Beef in Gravy surrounded by pastry|3.50|50
```

Where "3" represents the dollar component and "50" represents the cents component of the price. The maximum price for a purchase in the system will be ten dollars. A valid data file will contain the leading zeros for the cents in the prices. For example, an item costing $3.05 will be represented in the file as 3.05 and an item costing $3.50 will be represented in the file as 3.50. These details should be paid attention to in the loading time as well as when you attempt to save content to the files (see Requirements #6 and #19).

**You must ensure that items are inserted in the linked list in an orderly fashion and sorted by their names.**

Some sample and valid data files are provided as part of the startup code. **Make sure that your program works with these files.**

## Requirement #3: Display Main Menu

Your program must display an interactive menu displaying 9 options. Your menu should look like this:

```
Main Menu:
1.Display Items
2.Purchase Items
3.Save and Exit
Administrator-Only Menu:
4.Add Item
5.Remove Item
6.Display Coins
7.Reset Stock
8.Reset Coins
9.Abort Program
Select your option (1-9):
```

Your program must print out the above options and then allow the user to select these options by typing the number and hitting enter. Upon selection of an option, appropriate function will be called.

Upon completion of all options, except for the case of “Exit” and “Abort”, **the user must be returned to the main menu**.

You can assume that customers can only see the first three menu options, and the administrator can see all of them. Note that this is just an assumption and *must not be implemented*. **You are not allowed to password protect the administrator functions of this program. If the marker cannot access parts of your application, they cannot mark it and you will get zero for those components.**

The behaviour of these menu options is described in following requirements.

## Requirement #4: Display Items

This option allows the user to see the list of the items available for purchasing. This is the data that should have been loaded into the linked list in the Requirement #2. When the user selects 1 from the main menu, the following list should be displayed in the same format.

```
Items Menu

ID    | Name               | Available | Price
---------------------------------------------
I0001 | Meat Pie           | 12        | $ 3.50
I0002 | Apple Pie          | 0         | $ 3.50
I0003 | Lemon Cheesecake   | 4         | $ 4.00
I0004 | Lemon Meringue Pie | 3         | $ 3.00
I0005 | Lemon Tart         | 5         | $ 3.75
```

Note that the size of each column depends on the size of the largest element in that column. This can be seen in the above example where “Lemon Meringue Pie” is longest name and thus specifies the size of the “Name” column.

## Requirement #5: Purchase Item

This option allows the user to purchase an item and pay for it.

If the user pays an extra amount of money, the correct amount of change will be calculated and returned to them. The number of available items should be adjusted according to the purchase. You should not allow an item to be purchased if there is no such item available in the system (amount=0).

See the example below. Note that the money that the user enters is in cents (i.e. 200 for $2.00, 300 for $3.00, and 500 for $5.00).

```
Purchase Item
--------------
Please enter the id of the item you wish to purchase: I0001
You have selected “Meat Pie  Yummy Beef in Gravy surrounded by pastry”. This will
cost you $3.50.
Please hand over the money – type in the value of each note/coin in cents.
Press enter on a new and empty line to cancel this purchase:
You still need to give us $3.50: 200
You still need to give us $1.50: 300
Error: $3.00 is not a valid denomination of money.
You still need to give us $1.50: 500
Thank you. Here is your Meat Pie, and your change of $3.50.
Please come back soon.
```

After the purchase transaction completes, the control should be returned to the main menu. In the event that the user presses enter on an empty line (instead of paying money), the program should refund all the money entered and return the control to the main menu.

Note: Requirement #18 extends this requirement and adds an additional feature to it. However, to satisfy Requirement #5, you only need to complete the functionality explained in this section.

## Requirement #6: Save and Exit

You must save all the data to the data file(s) that were provided on the command-line when the program loaded up. When the saving is completed, the program must exit.

The format of the data file(s) must be maintained and the program must be able to load up your saved file(s) as easily as it loaded up the file(s) provided with the startup code.

At this point, once you have implemented dynamic memory allocation, you must free all the memory allocated before exiting the program. Please ensure that the data file(s) contain a new-line character at the end of the file or this can create problems when data is loaded in with fgets().

## Requirement #7: Add Item

This option adds an item to the system. When the user selects this option, the system should generate the next available item *id* and associate that with this item. The user should then be prompted for the Name and Description and Price. The Price should be entered as a valid amount of money in dollars and cents as shown in the example below. The item should then be allocated the default "on hand" value specified in the startup code. The new item id shall have an 'I' prepended to it and will be 5 characters long. What follows is an example.

```
This new meal item will have the Item id of I0006.
Enter the item name: Baklava
Enter the item description: rich, sweet pastry made of layers of filo filled with
chopped nuts and sweetened and held together with syrup or honey.
Enter the price for this item: 8.00
This item "Baklava – rich, sweet pastry made of layers of filo filled with chopped
nuts and sweetened and held together with syrup or honey." has now been added to the
menu.
```

## Requirement #8: Remove Item

Remove an item and delete it from the system. You must free the memory that is no longer being used. The following is an example.

```
Enter the item id of the item to remove from the menu: I0001

"I0001 – Meat Pie  Yummy Beef in Gravy surrounded by pastry" has been removed from
the system.
```

## Requirement #9: Reset Stock Count

This option will require you to iterate over every stock in the list and set its 'on hand' count to the default value specified in the startup code.

You should display a message once this is done such as *"All stock has been reset to the default level of X"* where X is the default stock level specified in the startup code.

## Requirement #10: Abort

This option should terminate your program. All program data *will be lost*. You should also free any dynamically allocated memory before aborting the program.

## Requirement #11: Return to the Main Menu

Your program should allow the user to return to the main menu at any point during these options. The user can do this by hitting enter on an empty line. If the user is in the middle of a transaction, that transaction should be cancelled.

## Requirement #12: Makefile

We should be able to compile your program using a Makefile, which should be submitted along with the code of your assignment. All compile commands must include the *"-ansi -Wall -pedantic"* compile options. You program should compile cleanly with these options; **no error or warning is acceptable even during the demo and prior to the final submission.**

Your Makefile needs to compile your program incrementally. That is, it should use *object* files as an intermediate form of compilation.

You should also include a target called "clean" that deletes unnecessary files from your working directory such as object files, executable files, core dump files etc. This directive should only be executed when the user types "make clean" at the command prompt.

You can find multiple examples about how a Makefile should be written in the lecture material.

## Requirement #13: Memory Leaks and Abuses

The startup code requires the use of dynamic memory allocation. Therefore, you will need to check that your program does not contain memory leaks. Use the following *valgrind* command to check for memory leaks, and submit the report in a text file named Requirement13Part2.txt along with the rest of the files in your project. Note that, inclusion of a new file in the submission package requires an update to the makefile accordingly.

```
valgrind ––leak-check=full --show-reachable=yes <command> <arguments>
```

Another common problem is memory abuses, which refers to situations such as reading from uninitialised memory, writing to memory addresses you should not have access to, and conditional statements that depend on uninitialised values. You can test for these again by using valgrind:

```
valgrind --track-origins=yes <command> <arguments>
```

The report generated by this command must be stored in a new file named Requirement13Part**2**.txt and included in your submission. Marks will only be awarded for this requirement if the reports generated by valgrind indicate that there are no memory leaks nor any other memory related problems.

## Requirement #14: Proper Use of an ADT

In this assignment, you are implementing two Abstract Data Types; that is, two lists. The first list is implemented as an array and the second list is implemented as a linked list. For this requirement, you will need to propose a list of interface functions for each list and implement these. All references to these types should be via these interface functions.

## Requirement #15: General Requirements

You must read the "Functional Abstraction" "Buffer Handling", "Input Validation" and "Coding Conventions and Practices" requirements written in the "General Requirements" section of this specification.

## Requirement #16: Assignment Demonstration

You are required to attend an assignment demonstration in week 10. You will be required to demonstrate requirements 1, 2, 3, 4 and 5 in this demonstration. Please note that this includes implementation of any requirements that these options depend upon. You will be required to compile and run your program on the RMIT Unix machines regularly used in this course.

As part of the assignment demo, **the tutor may ask you random questions about your source code**. You may be asked to open your source files and explain the operation of a randomly picked section. In the event that you will not be able to answer the questions, we will have to make sure that you are the genuine author of the program.

You must attend the lab session in which you are enrolled. If you cannot attend that session, you must seek permission from the Head Tutor to attend another session in advance. Failure to do so means you will not receive the mark allocated for this requirement.

## Requirement #17: Implement Main Menu Using Function Pointers

The main menu data structure is to be implemented as an array of MenuItem.

A MenuItem is defined as follows:

```
typedef void (*MenuFunction)(VmSystem *);
typedef struct menu_item
{
    char text[MENU_NAME_LEN + NULL_SPACE];
    MenuFunction function;
} MenuItem;
```

In this structure, `text` is the text to be displayed for a menu item, e.g., "Display Items"; and `function` is the function that implements that option, which, in this case, would be: `void displayItems(VmSystem * system);`
Your task for this requirement is to implement the `initMenu(MenuItem * menu)` function, which will initialise the menu array. The elements of the MenuItem array were

specified in Requirement #3. You need to initialise this array with the correct values for each menu item.

You should then implement the `getMenuChoice(MenuItem * menu)` function, which will contain the logic for allowing the user to select the choice they wish to run. You will then return a pointer to the menu item that has been selected. You can then call the function.

## Requirement #18: Support Coins in the System

This requirement contains 5 parts that should be implemented together.

**Part 1) Loading the coin file and storing the data in memory:** you are expected to support an additional command-line argument which contains information about the available money denominators. Your command-line arguments will look like below.

```
[s1234567@csitprdap02 ~]$ ./vm <itemsfile> <coinsfile>
```

For example, this program could be called as:

```
[s1234567@csitprdap02 ~]$ ./vm stock.dat coins.dat
```

You should not validate the file name or path. Instead, validate that the file contains valid content in the right format.

Money data file format is as follows:

```
[denomination],[quantity]
```

That is, there will be a row for each value of money that exists and the system will have an amount of each denomination in cents. For example:

```
1000,3
500,4
200,20
100,30
50,5
20,3
10,40
5,20
```

This means that the system currently has 3 x 10 dollar notes, 4 x 5 dollar notes, 20 x 2 dollar coins, etc. Note that the above denominations are the only valid denominations for your vending machine. The vending machine does not accept $20 or $50 notes. A valid file will always contain exactly 8 denominations.

If you wanted to initialise the vending machine with no change, then the valid way to do so would be to load a file containing the following data:

```
1000,0
500,0
200,0
```

```
100,0
50,0
20,0
10,0
5,0
```

**Part 2) Calculating the change that should be returned according to coin availability:** this will be reported at the end of the purchase process as shown in the below example. Note the underlined section at the end.

```
Purchase Item

Please enter the id of the item you wish to purchase: I0001
You have selected “Meat Pie  Yummy Beef in Gravy surrounded by pastry”. This will
cost you $3.50.
Please hand over the money – type in the value of each note/coin in cents.
Press enter on a new line to cancel this purchase:
You still need to give us $3.50: 200
You still need to give us $1.50: 300
Error: $3.00 is not a valid denomination of money.
You still need to give us $1.50: 500
Thank you. Here is your Meat Pie, and your change of $3.50: $2 $1 50c
Please come back soon.
```

The process of calculating the change should start with the largest available denominator. If there is insufficient amount of that denomination available in the machine, or if the required change cannot be returned using that denomination, then the next largest available denominator should be used. This will make sure that the machine will always return the smallest number of “coins” to the customer.

When refunding money, you must display each note or coin separately used in the refund. You must ensure that, prior to the sale, there is sufficient denominations in the system so that the customer is given the correct amount of change. You must also subtract these coins from the coins-array if a sale can take place. Note that coins entered to pay for an item will form part of the change that can be given to the customer. If pdd cannot give correct change then the sale should not occur, and your program should display an appropriate message explaining why.

**Part 3) Reset Coin Counter:** This option will require you to iterate over every coin in the coin list and set its 'count' to the default value specified in the startup code. You should display a message once this is done such as *“All coins have been reset to the default level of X”* where X is the default amount of coins specified in the startup code.

**Part 4) Display Coins:** This option will require you to display the coins as follows. In particular, the count of coins should be correctly aligned and sorted from lowest to highest value as shown below.

```
Coins Summary
---------
Denomination | Count

5 cents      | 20
10 cents     | 40
20 cents     | 3
50 cents     | 5
1 dollar     | 30
2 dollar     | 20
5 dollar     | 4
10 dollar    | 3
```

**Part 5) Save the coins file:** you are expected to save the coins file when exiting the program. The format must stay the same and your program should be able to load the file next time.

## Requirement #19: Validating the Input Files

In the previous requirements, it was assumed that the data files that you have loaded into memory were valid and contained no error. However, in real life this is not necessarily the case and you will be required to handle the errors that may arise from reading a corrupt or invalid data file.

Examples of the corrupt and/or invalid data files include the following:

1) There are lines of data in the file which does not contain enough fields to constitute a valid record;
2) There are lines of data in the file which contain too many fields for a valid record;
3) The number of fields is correct; however, the type of data available for one or more of the fields does not match the expectation of the program;
4) The data types match the expectation, but they are out of the acceptable range; and
5) The data file is empty; and
6) The price does not contain a dollars and cents separator character (i.e., the '.' character).

To complete this requirement, you are required to handle all the scenarios mentioned above for both the data files that your program is supposed to load.

We will test your program with invalid data files to make sure that you have handled errors properly.

# 2. General Requirements

## Buffer Handling

This requirement relates to how well your programs handle "extra input".
To aid you with this requirement, we want you to use the *readRestOfLine()* function provided in the start-up code.

Marks will be deducted for the following buffer-related issues:

- Prompts being printed multiple times or your program "skipping" over prompts because left over input was accepted automatically. These problems are a symptom of not using buffer overflow code often enough.
- Program halting at unexpected times and waiting for the user to hit enter a second time. Calling your buffer clearing code after every input routine without first checking if buffer overflow has occurred causes this.
- The use of buffer clearing code may have been avoided with *gets()* or *scanf()* for scanning string input. These functions are not safe because they do not check for the amount of input being accepted.
- Using long character arrays as a sole method of handling buffer overflow. We want you to use the *readRestOfLine()* function.
- Other buffer related problems.

For example, what happens in your program when you try to enter a string of 40 characters when the limit is 20 characters?

## Input Validation

Where you need the user to enter input, you will need to check the length, range and type of all inputs where applicable. For example, you will be expected to check the length of strings (e.g. 1-20 characters), the ranges of numeric inputs (e.g. an integer with value 1-7) and the type of data (e.g. is this input numeric?). For any detected invalid inputs, you are asked to re-prompt for this input. You should not truncate extra data or abort the function.

## Coding Conventions and Practices

Marks are awarded for good coding conventions/practices such as:

- Adding author identification comments (i.e. name, student number, course and year) to the beginning of every source file in your assignment.
- Avoiding global variables.
- Avoiding goto statements.
- Consistent use of spaces or tabs for indentation. Be careful to not mix tabs and spaces. Each "block" of code should be indented one level (i.e. either 3 spaces or one tab).
- Keeping line-lengths to a reasonable maximum such that they fit into 80 columns.
- Commenting (including function header comments).
- Appropriate identifier names.
- Avoiding magic numbers.
- Avoiding platform specific code with *system()*.
- Checking the return values of important functions such as *fopen()*, *malloc()* and so on.

- General code readability.
- Not removing the default return values and replacing these with meaningful ones.

### Functional Abstraction

We encourage the use of functional abstraction throughout your code. It is considered to be a good practice when developing software with many benefits. Abstracting code into separate functions reduces the possibility of bugs in your project, simplifies programming logic and eases the need for debugging. We are looking for some evidence of functional abstraction throughout your code. As a rule, if you notice that you have the same or similar block of code in two different locations of your source, you can abstract this into a separate function. Another rule of thumb is to check if any given function is getting longer than 50 lines.

## 3. Marking Guide

The following table contains the detailed breakdown of marks allocated for each requirement.

| **Requirement** | **Mark** | **Bonus** |
|---|---|---|
| #1: Command-line arguments | 8 | |
| #2: Load Data | 10 | |
| #3: Display Main Menu | 3 | |
| #4: Display Items | 3 | |
| #5: Purchase Item | 10 | |
| #6: Save and Exit | 8 | |
| #7: Add Item | 6 | |
| #8: Remove Item | 4 | |
| #9: Reset stock Count | 2 | |
| #10: Abort | 2 | |
| #11: Return to the Main Menu | 3 | |
| #12: Makefile | 5 | |
| #13: Memory Leaks and Abuses | 4 | |
| #14: Proper Use of an ADT | 4 | |
| #15: General Requirements | 15 (4+3+4+4) | |
| #16: Assignment Demonstration | 13 | |
| #17: Implement Main Menu Using Function Pointers | | 2 |
| #18: Support Coins in the System | | 6 |
| #19: Validating the Input Files | | 2 |
| **Total** | **100** | **10** |

### Penalties

Marks will be deducted for the following:

- Compile errors and warnings.
- Fatal run-time errors such as segmentation faults, bus errors, infinite loops, etc.
- Missing files (affected components get zero marks).
- Files incorrectly named, or whole directories submitted.
- Not using start-up code or not filling-in the readme file.

Programs with compile errors that cannot be easily corrected by the marker will result in a maximum possible score of 40% of the total available marks for the assignment.
Any sections of the assignment that cause the program to terminate unexpectedly (i.e., segmentation fault, bus error, infinite loop, etc.) will result in a maximum possible score of 40% of the total available marks for those sections. Any sections that cannot be tested as a result of problems in other sections will also receive a maximum possible score of 40%.
It is not possible to resubmit your assignment after the deadline due to mistakes.

## 4. Submission Information

Late submissions attract a marking deduction of 10% of the maximum mark attainable per day for the first 5 days, including weekdays and weekends. After this time, a 100% deduction is applied.
We recommend that you avoid submitting late where possible because it is difficult to make up the marks lost due to late submissions.

## 5. Submission Content

You need to submit at least the following files:

| File | Description |
|---|---|
| vm.c | This file will contain the main function. |
| vm.h | Header file for vm.h |
| vm_menu.c | Contains code for the initialisation and display of the menu using function pointers. |
| vm_menu.h | Header file for vm_menu.c |
| vm_options.c | This file contains functions for each of the major menu options. |
| vm_options.h | Header file for vm_options.c |
| vm_stock.c | This file will contain the interface functions you create to manage the linked list of stock. |
| vm_stock.h | Header file for vm_stock.c |
| vm_coins.c | This file will contain any additional functions you create to manage the collection of coins. |
| vm_coins.h | Header file for vm_coins.c |
| vm_system.h | This file contains the definition of the primary data-structures used in the program. |
| utility.c | This file will contain additional code for the running of your program. For example this can include your own functions that help you collect and validate user input or any other general use functions. |
| utility.h | Header file for utility.c |
| Makefile | This file will compile your program in an incremental fashion. |
| README.txt | This file will only be necessary if you have obtained the permission for a late submission. You should copy the text of the email confirming your extension into this file. Note that, if you |

| | |
|---|---|
| | are including this file in your submission, you must modify your Makefile so that the README.txt file is also included in the final archive. |
| Requirement13Part1.txt | Refer to Requirement 13 for details. |
| Requirement13Part2.txt | Refer to Requirement 13 for details. |

## Submission Instructions

Create a zip archive using the following command on the server:

```
[s1234567@csitprdap02 ~]$ make archive
```

- This command would generate a zip file named “usernamea2.zip” with your username substituted for “username”.
- Submit the produced archive file.