**National University of Computer & Emerging Sciences**

**Karachi Campus**



**Project Report**

**Niedrige Erwartungen:**

**A modern online grocery shopping experience**

**[]**

**Course Teacher:**

**Muhammad Nouman Durrani**

**Nimra Iqbal**

**Group Members:**

**Saud Ahmed Abbasi 19K-0229**

**Fawad J.Fateh 19K-1449**

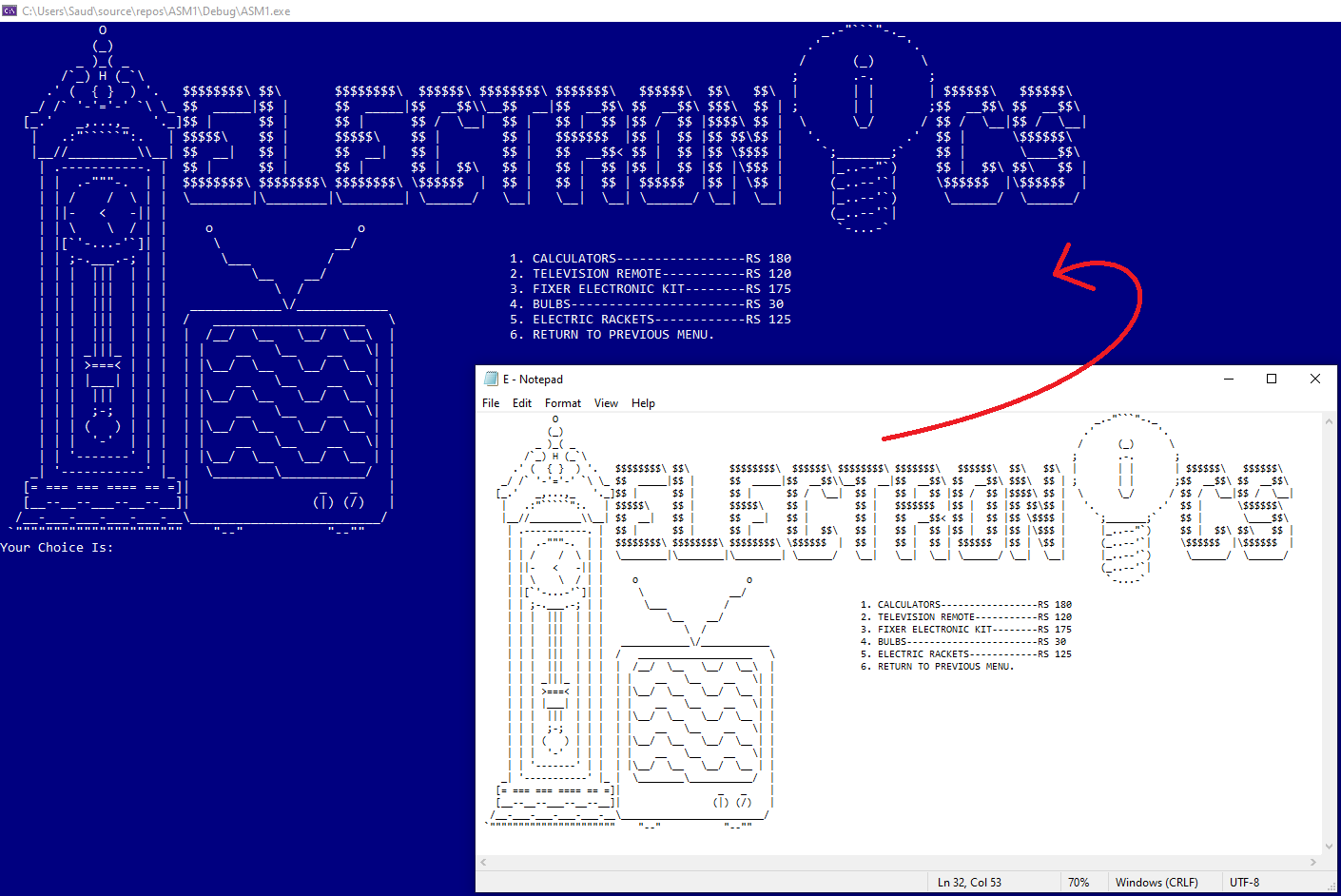
**Arun Jai 19K-1437**

* **Introduction**

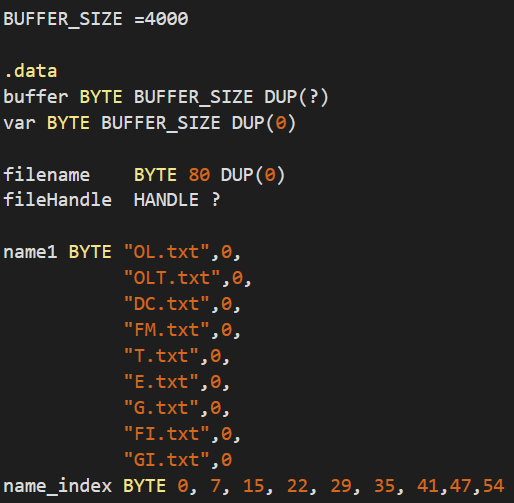
Niedrige Erwartungen allows you to carry out your daily shopping via this nifty little console application. Sounds boring but we’re putting our spin on it. It’s accompanied by compact, generalized ASCII supported visuals. Your purchasing list will be preserved and displayed when you check out of the system.

* **Breaking Down Its Features:**

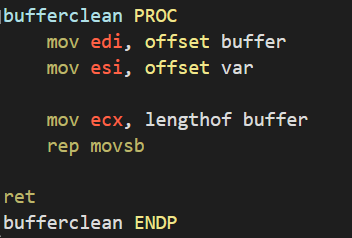
**Visuals:**



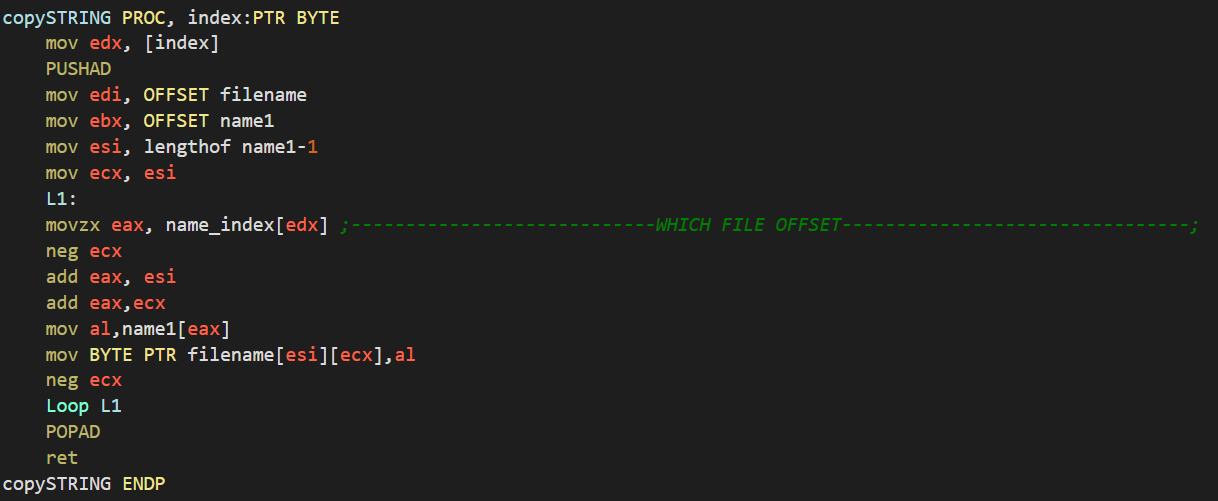
Working on the following data set:



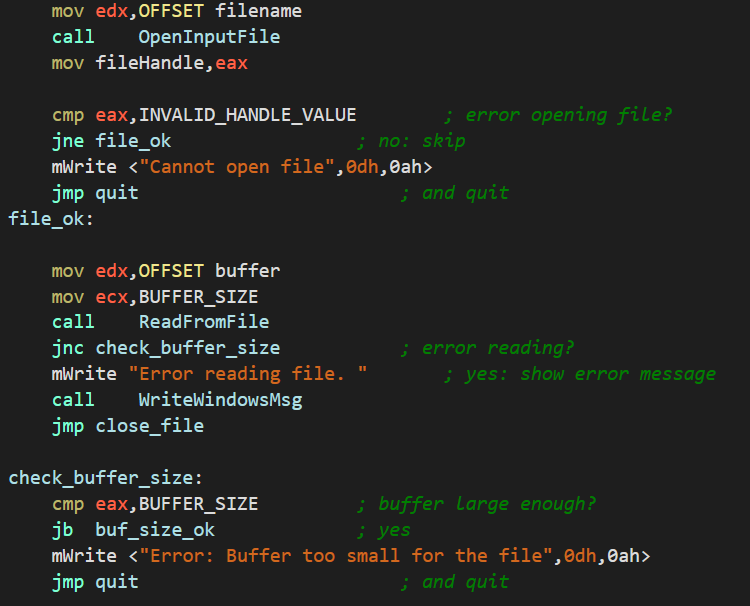
First clean buffer in every iteration so a previous visual doesn’t leak into the newer visual:



Copy the contents of the filename from the array name1 (you need only the starting index of each file as the terminating character **0** after each file name will be convincing enough for the filehandle to pick up the .txt file):



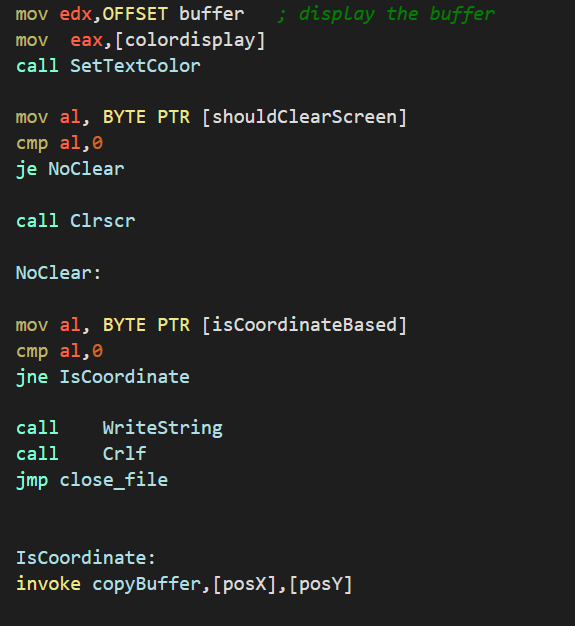
Check buffer size, then read all the contents of the file (name now stored in the variable filename) up to the buffer size limit, simultaneously do error handling accordingly (this code was inspired from the official Irvine website guide on filing, with modifications done to suit the program’s needs):



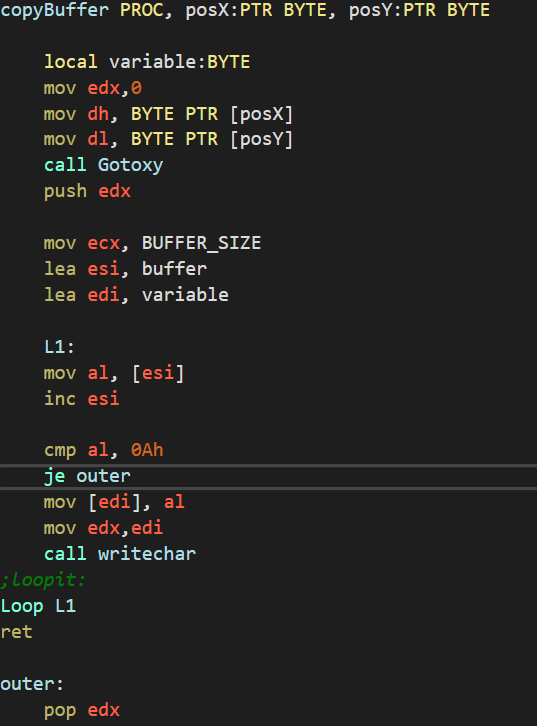
The function which prints the file takes the parameters *FILE NAME INDEX, COLOR OF TEXT+BACKGROUND, COORDINATES FOR X, COORDINATES FOR Y, IF COORDINATES APPLY, CLEAR SCREEN?*

Therefore in the following statement it will either print the visual on the specific coordinates provided in the parameters, no specific coordinates if the *IF COORDINATES APPLY* flag is 0, and clear screen before printing visual if *CLEAR SCREEN* is set,

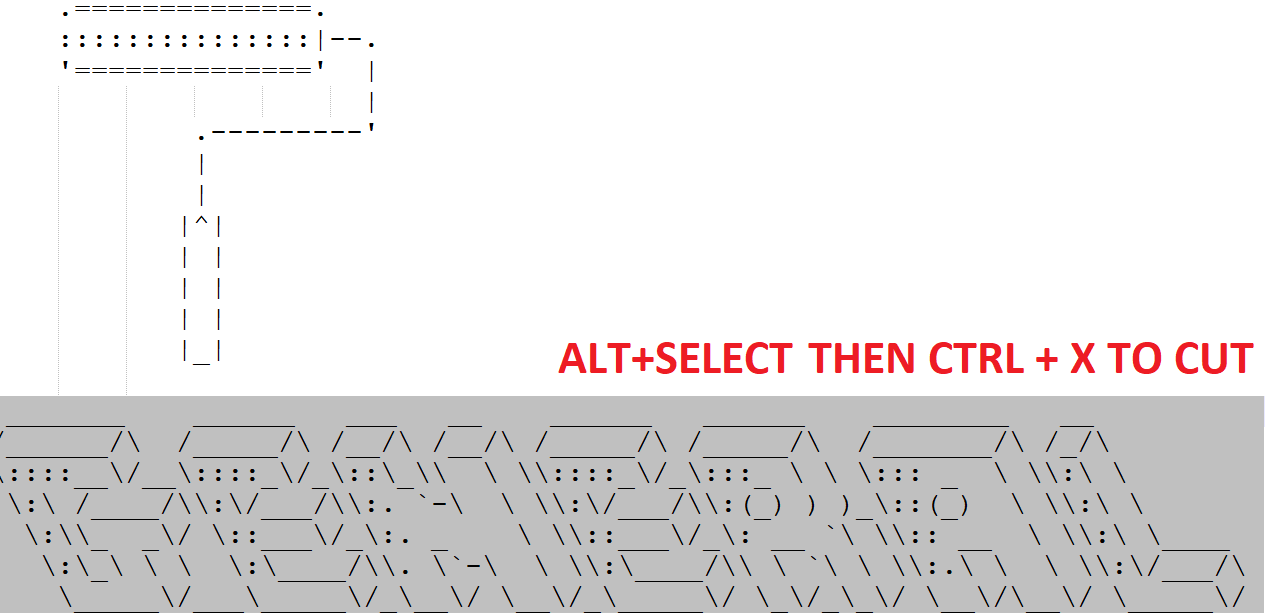
The *TEXT+BACKGROUND* parameter determines what kind of color the visual should be displayed in:

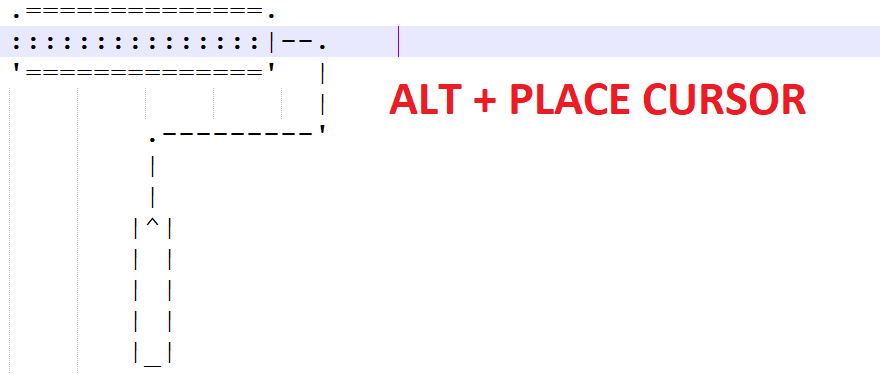
**

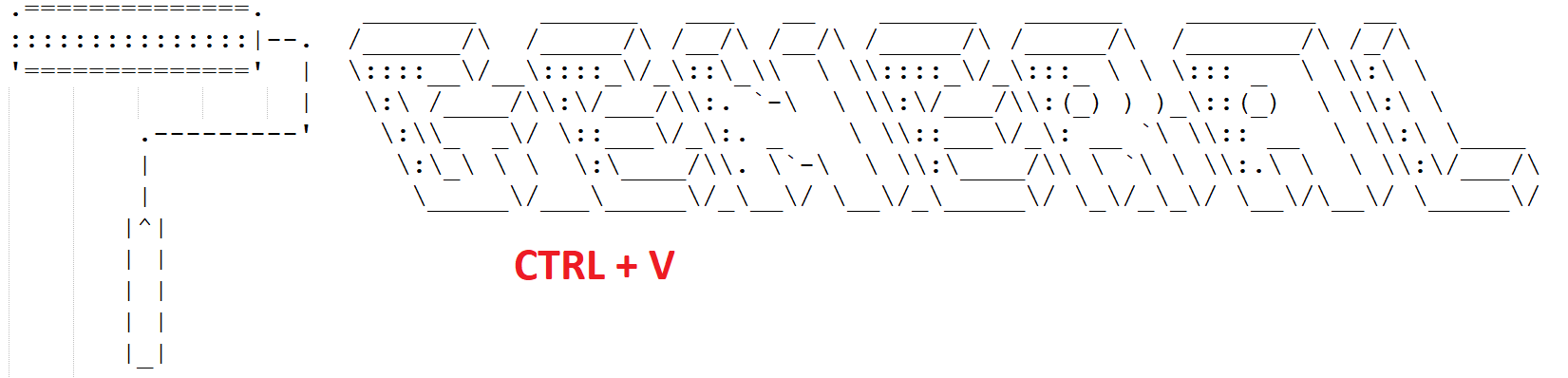
To print coordinates in a specific location the Gotoxy function is extensively used, line switching is maintained through the **dl,dh registers** which point to X and Y coordinates.

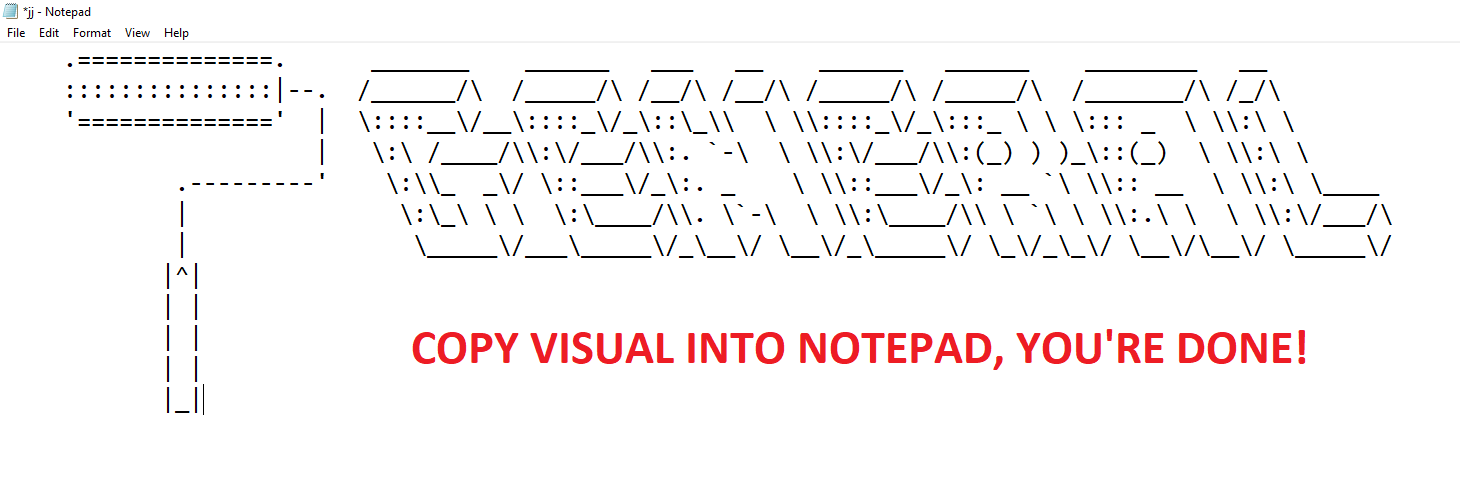
**

To print the visuals vertically aligned between one another, you can abuse the vertical select function in notepad++ which is reverse compatible with just the notepad, for example:

**

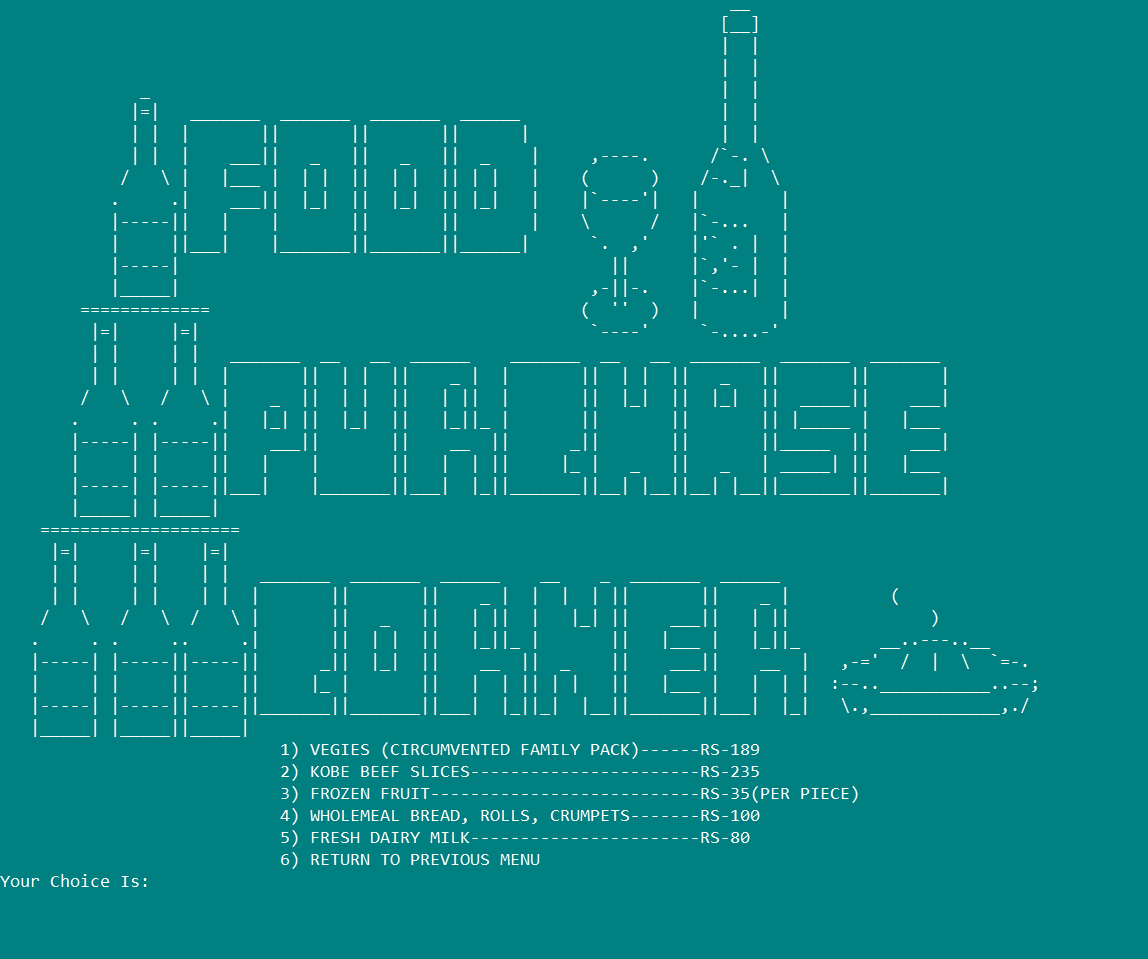
**

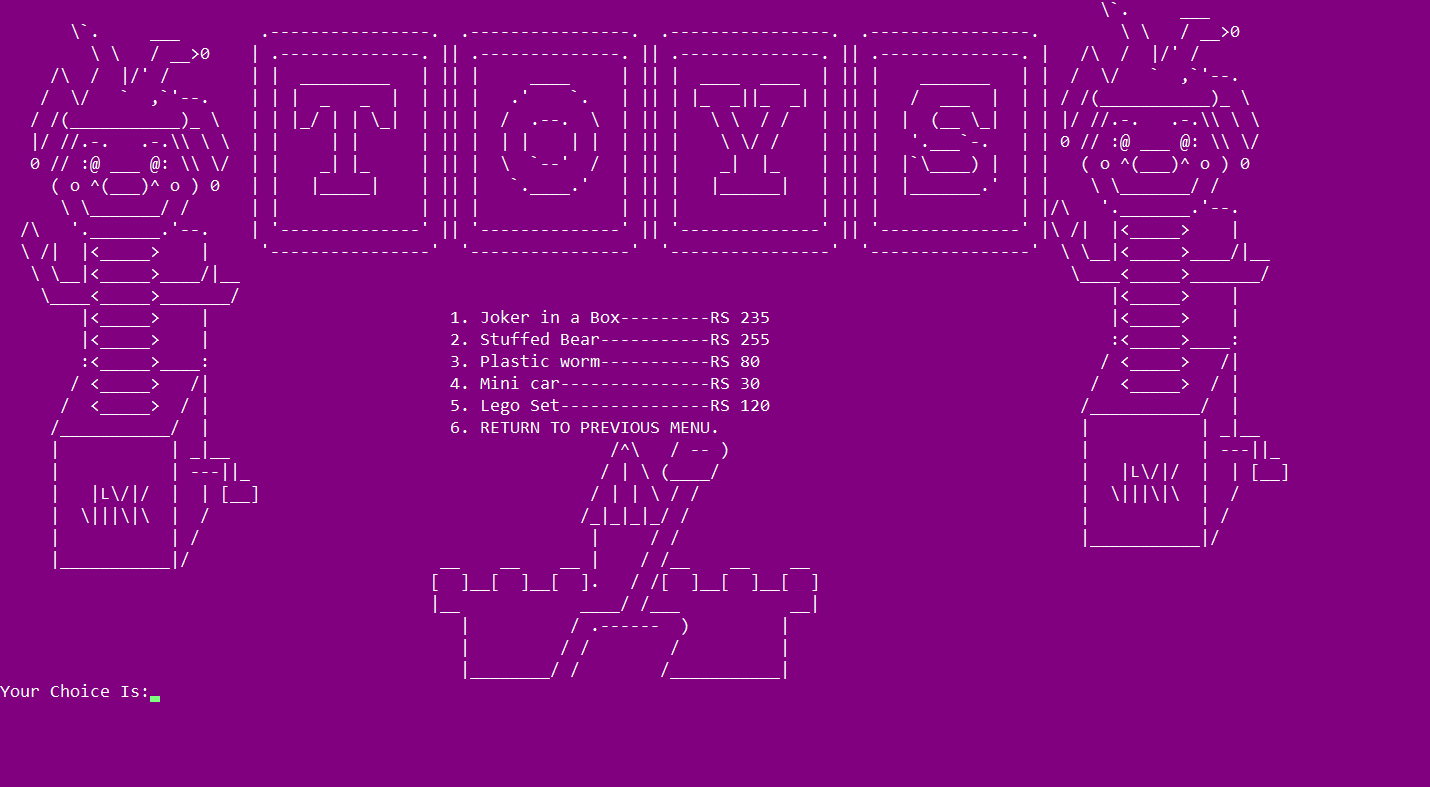
**

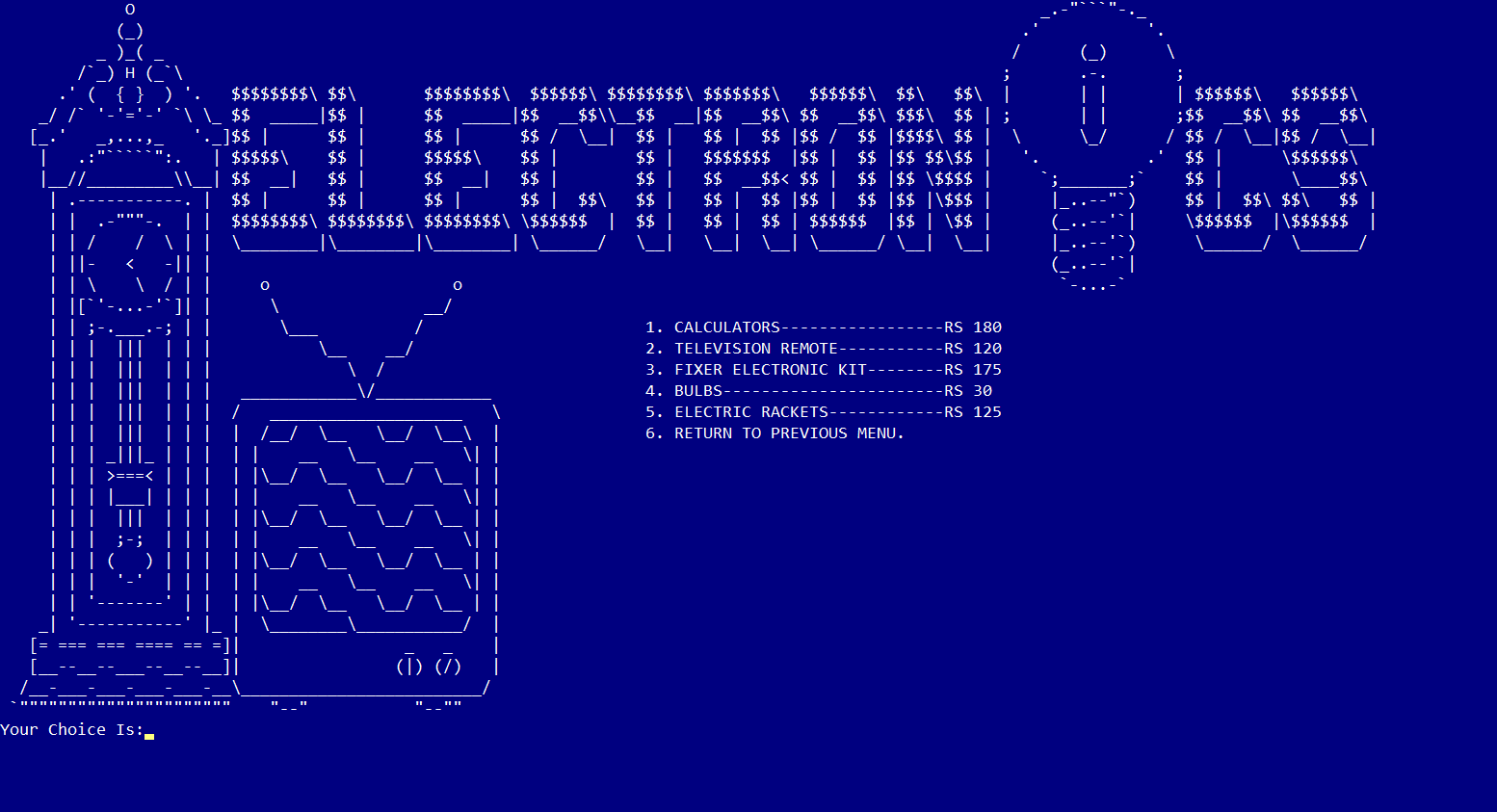
**

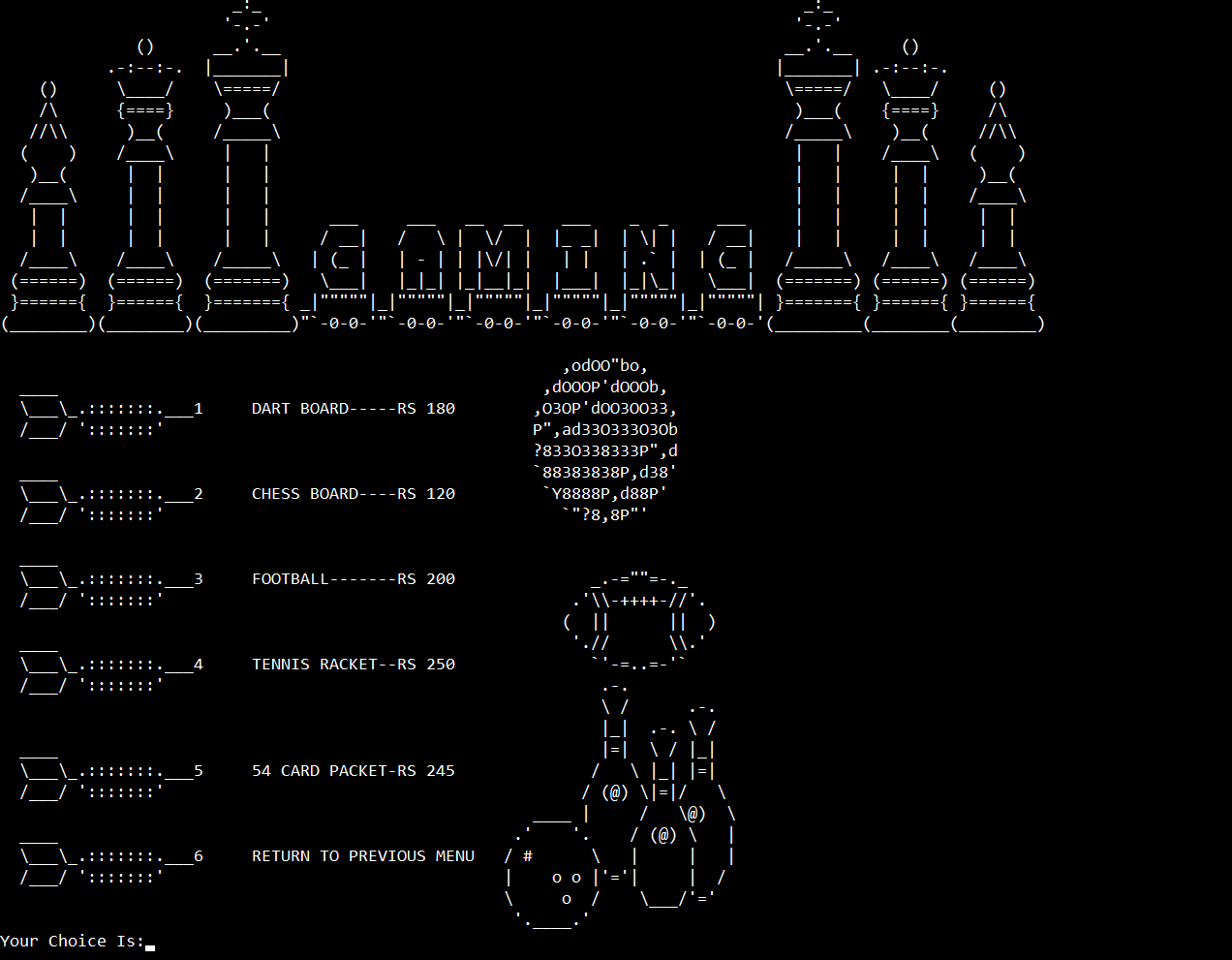
**All Visuals Created:**

**

**

**

**

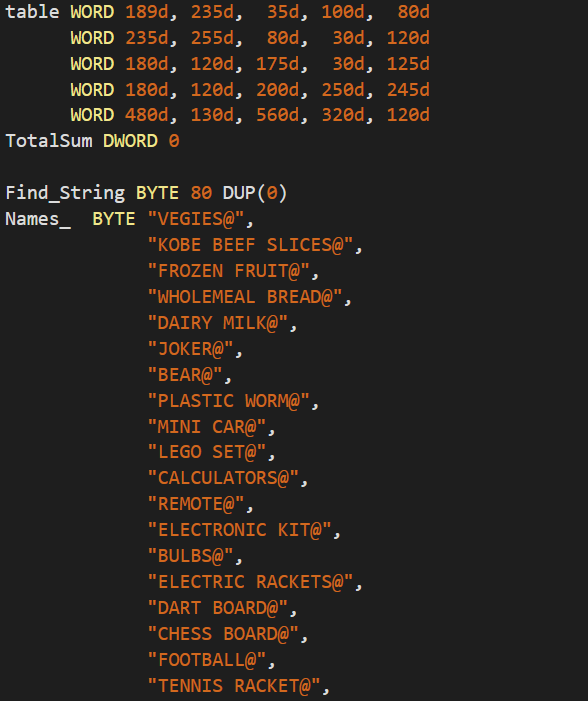
**

**

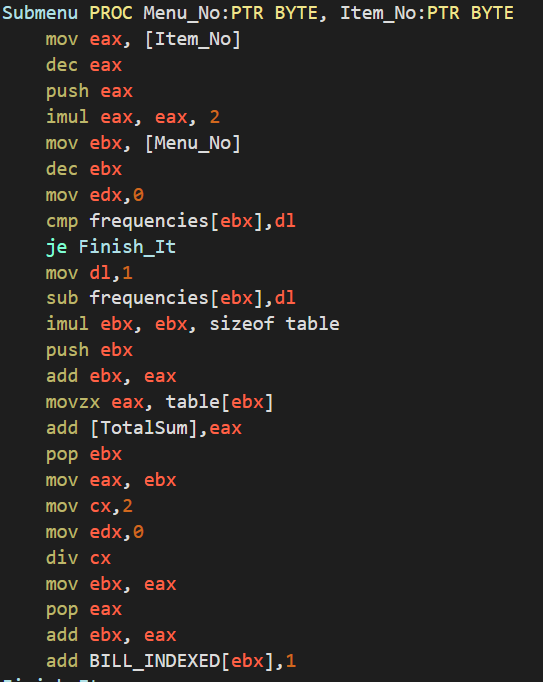
**

**Submenus:**

Declare all the prices in a 2 dimensional array, (for the sake of maintaining the properties of 2 dimensional traversal in MASM, all departments i.e. Food, Toys, Electronics, Gaming Supply and General Items have 5 items up for purchase)  
All names of items are maintained in Names\_ and Names\_2 to be printed during checkout.

**

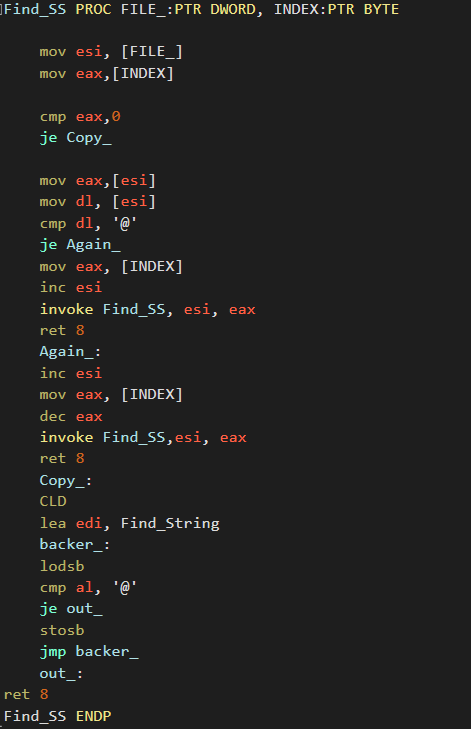
The submenu is able to a) procure the item from the 2d dimensional array (it decrements Menu\_No, multiplies it with size of one table row to get the designated row (or submenu for that specific department), and then it decrements Item\_No and gets the specific item that the user wants, while simultaneously updating the Stock Graph.

**

The Submenu procedure is also able to keep track of which item is being ordered at any given instant, what it does is for every 5 items in every department, it increments the stock count by 1 in the BILL\_INDEXED Array:

**

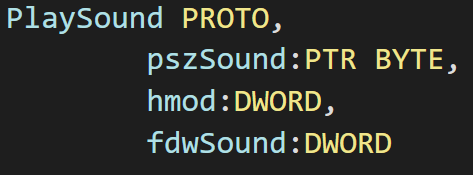
By doing this whenever we run the bill printing function, to get the names of what was ordered the loop only has to check every non 0 value in the BILL\_INDEXED array, and subsequently it can easily map the non-zero indexes onto Find\_SS, a recursive function that can call the string from any index provided in its parameters:

**

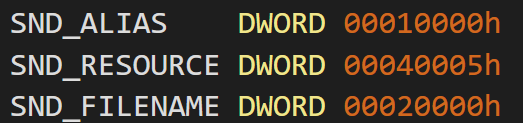
The corresponding bill total is incremented inside TotalSum and finally using all the features the following billing visual is created:

**

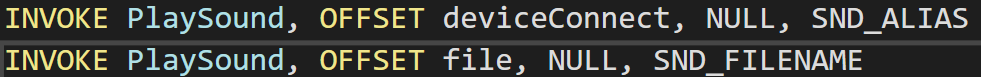
**Playing Sound:**

After extensively reading about Windows API, we were able to find out that the PlaySound function takes the following parameters: ****

By decoding the values of SND\_ALIAS, SND\_RESOURCE AND SND\_FILENAME we were able to find their respective values to pass as arguments into PlaySound, which are 00010000h, 00040005h and 00020000h respectively.

**

By feeding these arguments in the following format we can play sound, however until the sound completes playing, the program cannot move forward.   
(Offset file here is the offset of the name of the file we have defined, such as “**clapping.wav**”.

**

**Graph Plotting:**

The project contains a very interesting feature of bar graph plotting. A bar graph is plotted according to data generated at runtime from the user’s input.

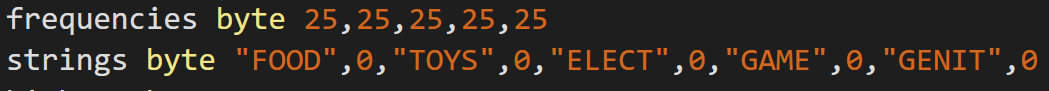
**Application:**

As our project possess the capability of ordering different type of products i.e. Food, Toys, Electronics, Gaming, and General items. This bar graph basically shows us the comparison between numbers of orders placed in each of these types.

**WORKING:**

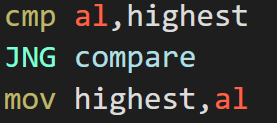
**Requirements or Inputs:**

The graph requires an array of integers basically which will contain the numbers of the orders of each type in sequence. For Example: 4, 3,6,5,8 orders of types Food, Toys, Electronics, Gaming, General items respectively.



**Overall working:**

In the initial phase of the program we have calculated the highest frequency among all. The reason for it is that because we are going to plot the vertical bar graph so we need to keep the record of what the maximum length of the bars is going to be.

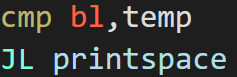


The real implementation of the graph resembles the techniques of designing patterns.

The implementation consists of a nested loop logic.

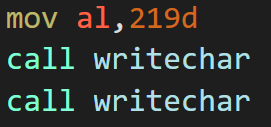
The outer loop runs until the highest value found, and the inner loop will run 5 times i.e. the number of types of categories we have of products.

The key to print the graph is the 219 ASCII character. This character is printed on the condition unless the spaces are printed.

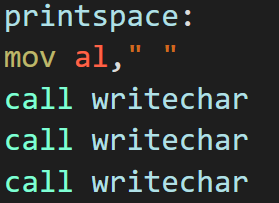


Temp contains the current outer loop running value, while register contains the frequency at that running index.

If the condition JL is false then the bar is printed with the help of 219 character.

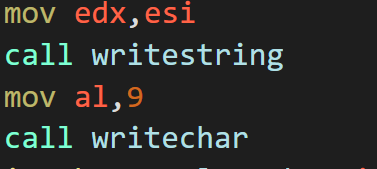


If the condition is true then only spaces are been printed.



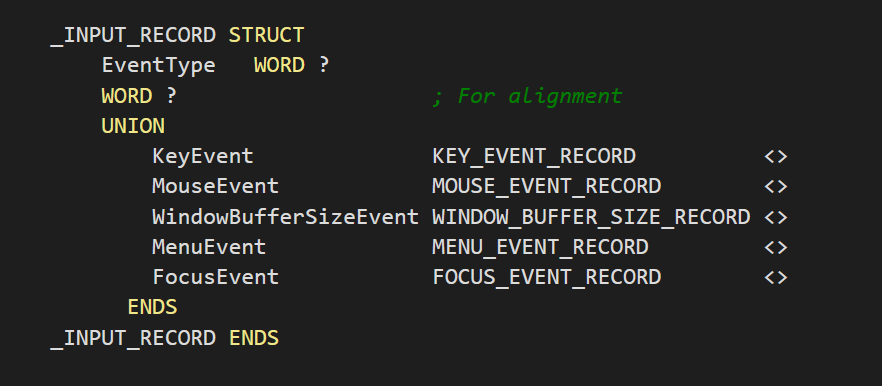
**Finishing:**

In the end the strings FOOD, TOYS, ELECT, GAME, GENIT below the bar which express that particular type.



**Mouse Implementation:**

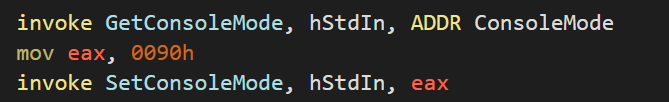
Make a structure with the key events as shown.

****

Official reference:

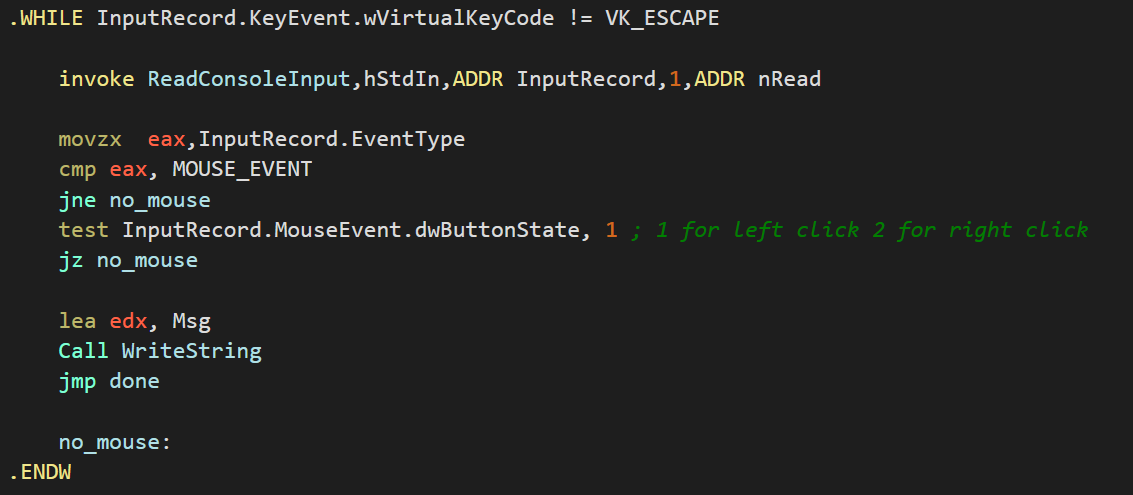
****

Do the following in code to enable mouse input, disable quick edit mode and enable extended flags:

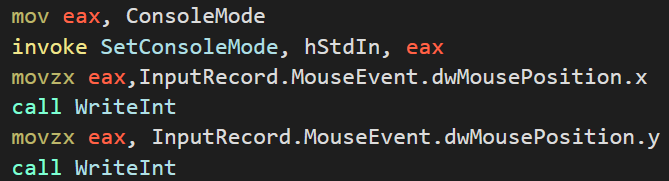
****

The following WHILE loop runs until the escape key is pressed on the keyboard (basically until the escape key is pressed, the MouseEvent will keep trying to get an input.)

Inside the MouseEvent testing, if the mouse event button state is 1, then there has been a left click, if there is a right click then the button state is 2.

****

Once a MouseEvent is detected then we can record the cursor position:

****

**Linked List Implementation with Heap Memory:**

Things to remember:

**GetProcessHeap** returns a 32-bit integer handle to the program’s heap area. Save this

handle and use it when calling other memory-related functions. Using this function, you can

request (allocate) memory without having to create your own heap.

**HeapAlloc** returns the address of block of memory from an existing heap, identified by a

heap handle. The allocated memory cannot be moved. If the memory cannot be allocated, the

function returns NULL (0).

**HeapFree** frees a block of memory previously allocated from a heap, identified by its

address and heap handle. If the block is freed successfully, the return value is nonzero. If the

block cannot be freed, the function returns zero and you can call the GetLastError API function

to get more information about the error.  
Note: GetProcessHeap allows you to allocate memory from the process heap without invoking **HeapCreate** first (by invoking HeapAlloc).

Example: .data

hHeapProc DWORD ?

dwBytes DWORD 1000 ;bytes to allocate

hHeapBlock DWORD ?

.code

INVOKE GetProcessHeap

mov hHeapProc,eax ;save handle to process heap

INVOKE HeapAlloc,

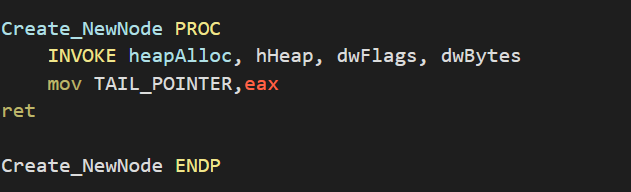
hHeapProc, ;handle to process heap

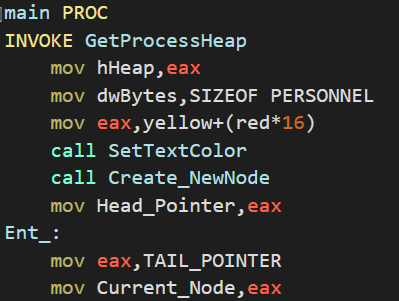
0,

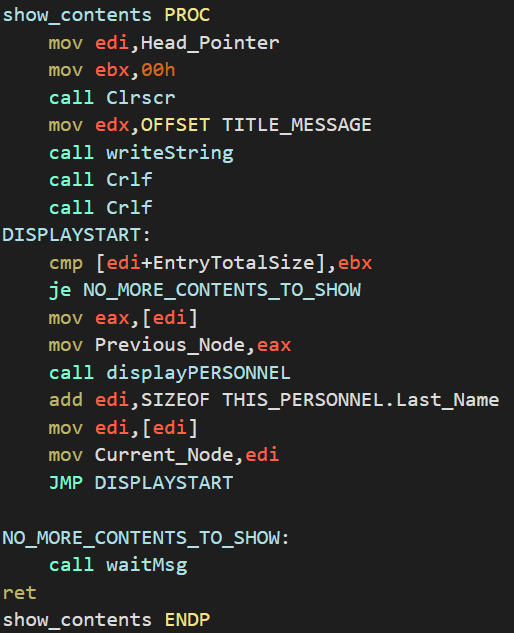
dwBytes

mov hHeapBlock,eax ;save handle to allocated block from heap

Call GetProcess heap to request memory to allocate memory without having to call your own heap.

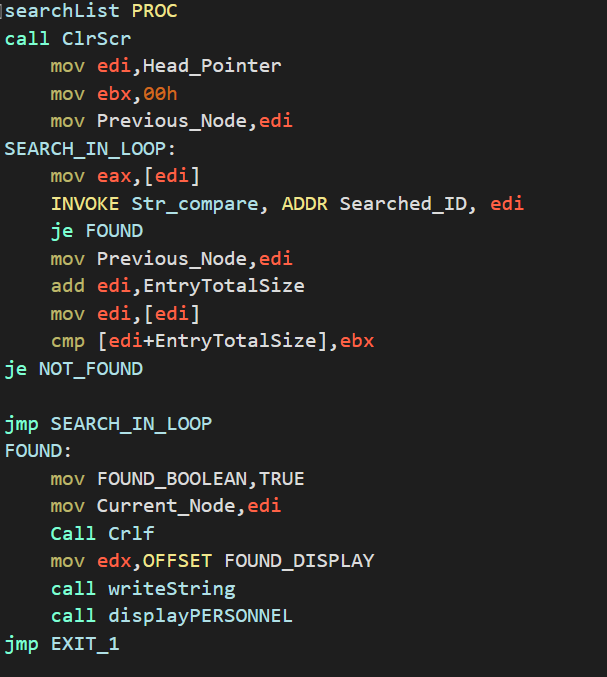
Set dwBytes with size of the PERSONNEL structure, make a rootpointer with Create\_NewNode, set the tail pointer.  


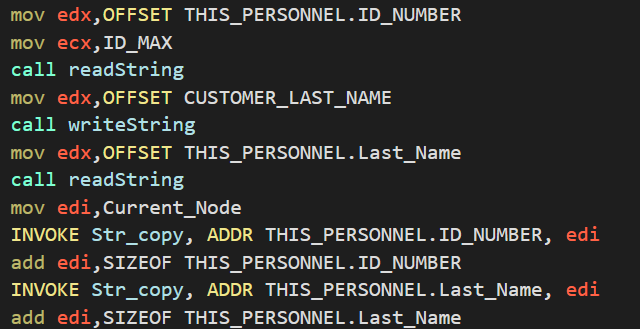
****

How to show all records?  
Move the head pointer into edi, now to get to each record sequentially, add the EntryTotalSize DWORD variable to edi to get to every successive entry, call displayPERSONNEL after every entry traversed. ****

In displayPERSONNEL, the first entry is the ID number, simply move the edi into edx and print the contents, when you want to print the name, simple add the offset inside edi with **SIZEOF THIS\_PERSONNEL.ID\_NUMBER,** to get the name entry inside the record.

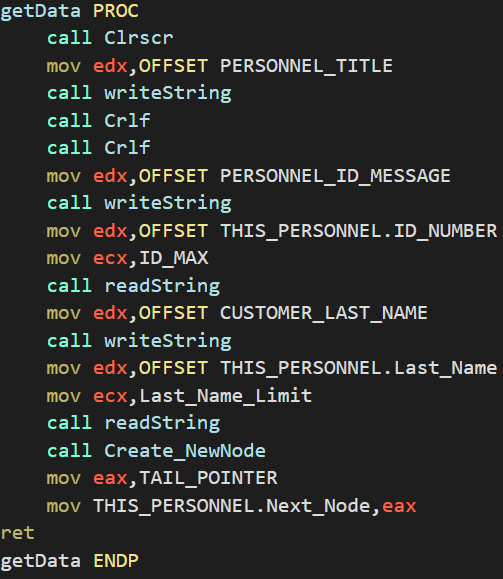
Similar to displaying the records, we can mold this function to search the records, all that’s different now is that during traversal we compare our searching ID with the contents in edi (remember, whenever we add EntryTotalSize with edi, the first variable inside the record is ID, hence it is the first thing compared)

****

For the updating of a particular record, simply input the values inside the record THIS\_PERSONNEL, then copy the contents of this structure into our offset of the record that we want to update (contained in edi), here’s a snippet of code to emphasize this:  
****

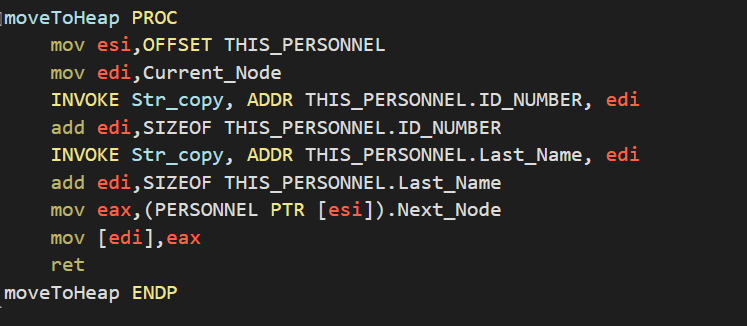
**How is a fresh new node created?**

Simply read the information in the previously aforementioned THIS\_PERSONNEL structure, this time call the Create\_NewNode which will put the value of a pointer pointing to a new allocated space of memory in the heap, move this value into the tail pointer, have the THIS\_PERSONNEL structure point to this new allocated space of memory inside the heap.



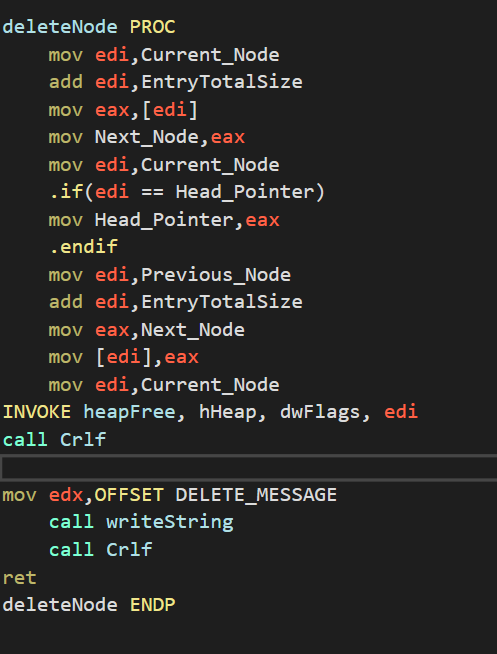
**How is Current\_Node being updated when we create a new node?**

Once you’re done with the creation of a new node, copy the values of this newly made node inside the current node (previously updated), traverse all record entries one by one (edi + sizeof THIS\_PERSONNEL.ID\_NUMBER + sizeof THIS\_PERSONNEL.Last\_Name to get to the Next Node DWORD variable inside the record, update the current node to point to this)

****

**How is node deleted?**If the current node == pointer, set the head pointer to equal to the next pointer,

Move the value after Previous\_Node into edi, and then have this value point to the Next\_Node, and then point the value inside edi to the current node, and then free the memory allocated in this register.

****

* **Existing System**

Preexisting online grocery systems are for general essential items, we plan to diversify our grocery list by having a bakery & perfume section etc.

* **Problem Statement**

Too many moving parts, convoluted UI and vendors often switch up item placement configuration so that people try out new products.

* **Proposed Solution** (In the Proposal)

Customer care and satisfaction is the goal. Your shopping experience doesn’t need to be extremely grandiose, here our aim is to get you to your goal item in the quickest steps possible. The visuals are an embellishment, the goal is to make a robust program that defies the expectation of what can be achieved by assembly language.

* **Salient Features** (In the Proposal)
* Regular Ordering Procedures
* Accompanying Visuals
* Diverse grocery vendors
* Checkout list
* Accompanying Sound
* Mouse Support
* Linked List Support for Records
* **Tools & Technologies**

MASM.