UECS1643/ UECS1653:

Fundamentals of Programming

Group Assignment

Practical Lecturer:

Dr. Sugumaran a/l Nallusamy

Practical Group: 12

Name	ID
Fong Chien Yoong	1800565
Ling Sze Chie	1703258
Sim Jia Wei	1807209
Tong Zi-Shun	1806364

Assignment Marking Sheet

Personal Data

Name	ID	Programme	Practical Group
Fong Chien Yoong	1800565	3E	12
Ling Sze Chie	1703258	EC	
Sim Jia Wei	1807209	ET	
Tong Zi-Shun	1806364	ET	

Practical Lecturer:	Overall Mark:
Dr. Sugumaran a/l Nallusamy	/100%

Submission Status

No submission	Upload wrong file	Late submission	No issue

1) Compilation and Running

Does not compile	Compile but no output/ wrong output	Compile and produce output

2) Presentation (6%)

(a) Indentation (3%): ☐ None ☐ Inconsistent ☐ Good

(b) Variable and function names (3%): ☐ Poor choice ☐ Can be improved ☐ Meaningful

3) Program Component (54% + Presentation 6% = 60%)

Program Components	Missing/ Does not work	Major errors	Not robust/ Minor errors	No issue	Maximum Marks	Marks Obtained
Menu and main program					8	
Task I – Read from file					4	
Task II – List					4	
Task III – Search by item number					4	
Task IV – Search by keyword					6	
Task V – Show low inventories					4	
Task VI – Update					6	
Task VII – Add					4	
Task VIII – Delete					8	
Task IX – Write to file with updates					6	
Presentation (Indentation, variables, ...)					6	
					Total	0

4) Report and other components (40%)

Components	Missing	Poor	Good	Excellent	Maximum Marks	Marks Obtained
Structure Chart					**5**	
Flowcharts/Pseudocode					**10**	
Sample text file					**5**	
General: Efforts, etc.					**20**	
					Total	0

Comments

Comments

Table of Contents

About

- **Program Name:** Mini Market Inventory
- **Programming Language:** The Infamous C++, the one which gives programmers the Big C.
- **Purpose:** To pass our current semester.
- **Credits:**
    - UTAR for delaying our journey towards dementia.
    - UTAR lecturers for their guidance, especially **Dr. Sugumaran a/l Nallusamy**, our practical lecturer.
    - The janitors, for preventing the re-emergence of the Black Plague.
    - God, for keeping us alive.
- **Team motto:** "Rakyat didahulukan, pencapaian diutamakan."
- **Disclaimer**: No animals were harmed in the making of this program.

Note: The program may occupy a lot of horizontal space, especially when viewing the inventory table. Please try to adjust the font and the size of the console window if you encounter this issue.

Screenshots

Load File

```
[ Mini Market Inventory ]

Load file?

-----------------------------
[ OPTIONS ] [ Yes  :: <y> ]
            [  No  :: <n> ]
            [ Exit :: </> ]
-----------------------------

>>
```

Main Menu

```
[ Mini Market Inventory => Main Menu ]

Company name: Kedai Runcit Ah Beng

-----------------------------------------
[ OPTIONS ] [   Exit Profile    :: </> ]
            [    View Items     :: <1> ]
            [     Add Item     :: <2> ]
            [   Modify Item    :: <3> ]
            [   Delete Item    :: <4> ]
            [ Profile Overview :: <5> ]
            [  Delete Profile  :: <6> ]
            [      About       :: <7> ]
-----------------------------------------

What would you like to do?

>>
```

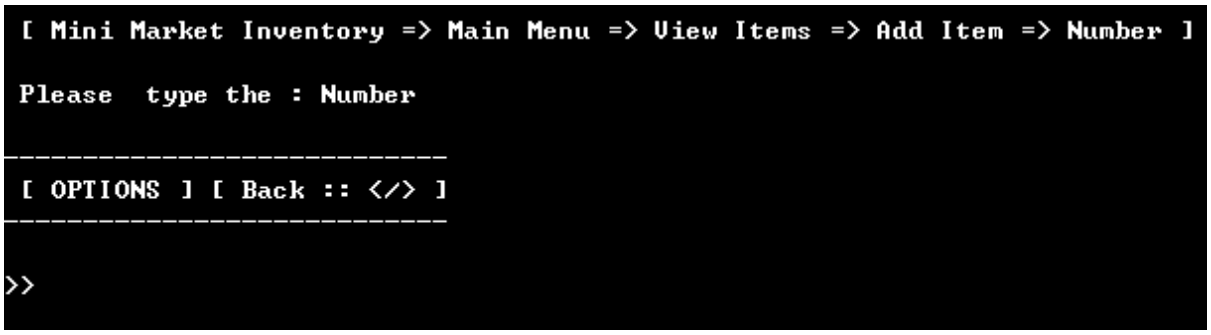
Inventory List

```
[ Mini Market Inventory => Main Menu => View Items ]
-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
[ Number ¦      Name      ¦ Quantity ¦  Size (per unit)  ¦    Net Size    ¦ Price (per unit) ¦  Price (per weight)  ¦ Net Price ¦      Brand      ¦    Supplier    ¦      Contact      ]
-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
[     27 ¦       Sugar      ¦        8 ¦          500.00 g  ¦    4000.00 g    ¦          8.00 ¦          0.13 g-1  ¦     64.00 ¦ 1 Malaysia      ¦    Najib Razak    ¦       imdb.com      ]
[      1 ¦       Mango      ¦       10 ¦          120.00 g  ¦    1200.00 g    ¦          2.00 ¦          0.17 g-1  ¦     20.00 ¦        -        ¦         -         ¦          -          ]
[      2 ¦       Apple      ¦        9 ¦          200.00 g  ¦    1800.00 g    ¦          1.00 ¦          0.04 g-1  ¦      9.00 ¦        -        ¦         -         ¦          -          ]
[      3 ¦       Grapes     ¦       11 ¦          100.00 g  ¦    1100.00 g    ¦          3.00 ¦          0.33 g-1  ¦     33.00 ¦        -        ¦         -         ¦          -          ]
[      4 ¦  Manuka Honey    ¦       14 ¦          800.00 g  ¦   11200.00 g    ¦         12.00 ¦          0.21 g-1  ¦    168.00 ¦       NZ        ¦        NZ         ¦    nz@gmail.com     ]
[      5 ¦       Chilli     ¦       20 ¦           50.00 g  ¦    1000.00 g    ¦          2.00 ¦          0.80 g-1  ¦     40.00 ¦        -        ¦         -         ¦          -          ]
[      6 ¦    HotWheels     ¦       20 ¦          350.00 g  ¦    7000.00 g    ¦         12.00 ¦          0.69 g-1  ¦    240.00 ¦      Mattel     ¦        Toyz       ¦    toys@mail.com    ]
[      7 ¦       Eggs       ¦       90 ¦          100.00 g  ¦    9000.00 g    ¦          1.00 ¦          0.90 g-1  ¦     90.00 ¦       A1        ¦        A1         ¦    a1@mail.com      ]
[      8 ¦     Spinach      ¦       14 ¦          120.00 g  ¦    1680.00 g    ¦          1.00 ¦          0.12 g-1  ¦     14.00 ¦        -        ¦       Muthu        ¦    0122502825       ]
[      9 ¦    Mushroom      ¦       12 ¦           80.00 g  ¦     960.00 g    ¦          0.00 ¦          0.00 g-1  ¦      0.00 ¦        -        ¦         -         ¦          -          ]
[     10 ¦      Pepper      ¦       14 ¦          150.00 g  ¦    2100.00 g    ¦          5.00 ¦          0.47 g-1  ¦     70.00 ¦      Babas      ¦       Babas       ¦      babas.com      ]
[     11 ¦  Curry powder    ¦       17 ¦          150.00 g  ¦    2550.00 g    ¦          4.00 ¦          0.45 g-1  ¦     68.00 ¦      Babas      ¦       Babas       ¦      babas.com      ]
[     12 ¦       Rice       ¦        8 ¦            5.00 kg  ¦      40.00 kg   ¦          8.00 ¦         12.80 kg-1 ¦     64.00 ¦   Faiza Emas    ¦       Faiza       ¦      faiza.com      ]
[     13 ¦      Bread       ¦        6 ¦          180.00 g  ¦    1080.00 g    ¦          2.00 ¦          0.07 g-1  ¦     12.00 ¦     Gardenia     ¦      Gardenia     ¦    gardenia.com     ]
[     14 ¦      Bread       ¦        4 ¦          200.00 g  ¦     800.00 g    ¦          3.00 ¦          0.06 g-1  ¦     12.00 ¦      Massimo     ¦      Massimo      ¦    massimo.com      ]
[     15 ¦      Garlic      ¦       35 ¦           75.00 g  ¦    2625.00 g    ¦          0.00 ¦          0.00 g-1  ¦      0.00 ¦        -        ¦      Ah Kong      ¦    0177568952       ]
[     16 ¦       Flour      ¦        3 ¦          500.00 g  ¦    1500.00 g    ¦          5.00 ¦          0.03 g-1  ¦     15.00 ¦     GoodChef     ¦      GoodChef     ¦    goodchef.com     ]
[     17 ¦      Cereal      ¦        6 ¦          400.00 g  ¦    2400.00 g    ¦          9.00 ¦          0.14 g-1  ¦     54.00 ¦     KokoKrunch   ¦       Nestle      ¦      nestle.com     ]
[     18 ¦      Cereal      ¦        3 ¦          400.00 g  ¦    1200.00 g    ¦         10.00 ¦          0.07 g-1  ¦     30.00 ¦     Honey Stars  ¦       Nestle      ¦      nestle.com     ]
[     19 ¦       Pasta      ¦        8 ¦          500.00 g  ¦    4000.00 g    ¦          8.00 ¦          0.13 g-1  ¦     64.00 ¦       Prego     ¦       Prego       ¦      prego.com      ]
[     20 ¦       Milk       ¦        5 ¦            1.00 kg  ¦       5.00 kg   ¦          5.00 ¦         25.00 kg-1 ¦     25.00 ¦     GoodDay      ¦      GoodDay      ¦    goodday.com      ]
[     21 ¦       Milk       ¦        3 ¦            1.00 kg  ¦       3.00 kg   ¦          8.00 ¦         24.00 kg-1 ¦     24.00 ¦        HL        ¦         HL        ¦      hl.com         ]
[     22 ¦       Milk       ¦        7 ¦            1.00 kg  ¦       7.00 kg   ¦          6.00 ¦         42.00 kg-1 ¦     42.00 ¦     Dutch Lady   ¦       Nestle      ¦      nestle.com     ]
[     23 ¦   Cooking Oil    ¦        7 ¦            1.00 kg  ¦       7.00 kg   ¦         15.00 ¦        105.00 kg-1 ¦    105.00 ¦     Naturel      ¦      Naturel      ¦    naturel.com      ]
[     24 ¦   Cooking Oil    ¦        4 ¦            1.00 kg  ¦       4.00 kg   ¦         12.00 ¦         48.00 kg-1 ¦     48.00 ¦     Helang       ¦      Kim Huat     ¦    kimhuat.com      ]
[     25 ¦      Butter      ¦        4 ¦          250.00 g  ¦    1000.00 g    ¦          7.00 ¦          0.11 g-1  ¦     28.00 ¦     Buttercup    ¦      Buttercup    ¦ ilovebutter.com ]
[     26 ¦      Butter      ¦        6 ¦          250.00 g  ¦    1500.00 g    ¦          9.00 ¦          0.22 g-1  ¦     54.00 ¦      Anchor      ¦       Anchor      ¦    anchor.com       ]
[     28 ¦       Tea        ¦        3 ¦          250.00 g  ¦     750.00 g    ¦         12.00 ¦          0.14 g-1  ¦     36.00 ¦       Boh        ¦        Boh        ¦      boh.com        ]
[     29 ¦       Tea        ¦        6 ¦          300.00 g  ¦    1800.00 g    ¦         11.00 ¦          0.22 g-1  ¦     66.00 ¦      Lipton      ¦       Lipton      ¦      lipton.com     ]
[     30 ¦ Instant noodles  ¦        9 ¦          150.00 g  ¦    1350.00 g    ¦          4.00 ¦          0.24 g-1  ¦     36.00 ¦     Mi Sedaap    ¦      WingsFood    ¦    wingsfood.com    ]
-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

-----------------------------
[ OPTIONS ] [     Back      :: </> ]
            [   Sort List   :: <1> ]
            [ Low Inventory :: <2> ]
            [   Add Item    :: <3> ]
            [  Modify Item  :: <4> ]
            [  Delete Item  :: <5> ]
-----------------------------

>>
```

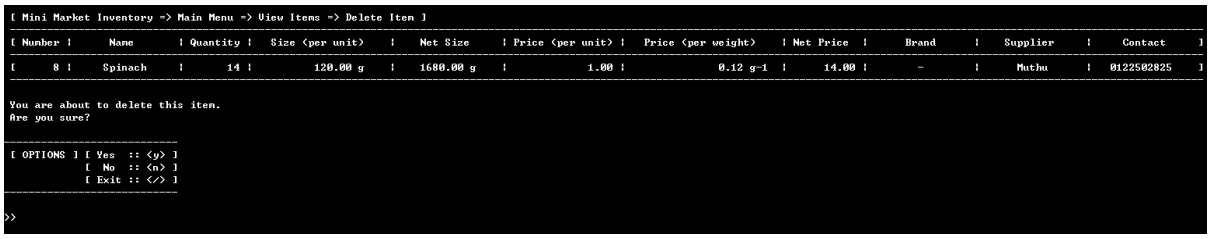
Sorting Options

```
[ Mini Market Inventory => Main Menu => View Items => Choose Sorting Pattern ]

-----------------------------------
[ OPTIONS ] [    Back     :: </> ]
            [ Ascending   :: <1> ]
            [ Descending  :: <2> ]
            [  Keyword    :: <3> ]
-----------------------------------

Sorting pattern?

>>
```

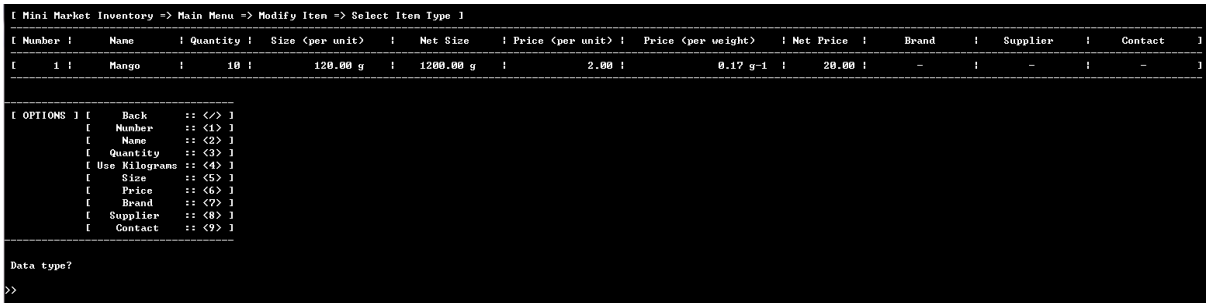
Add Item



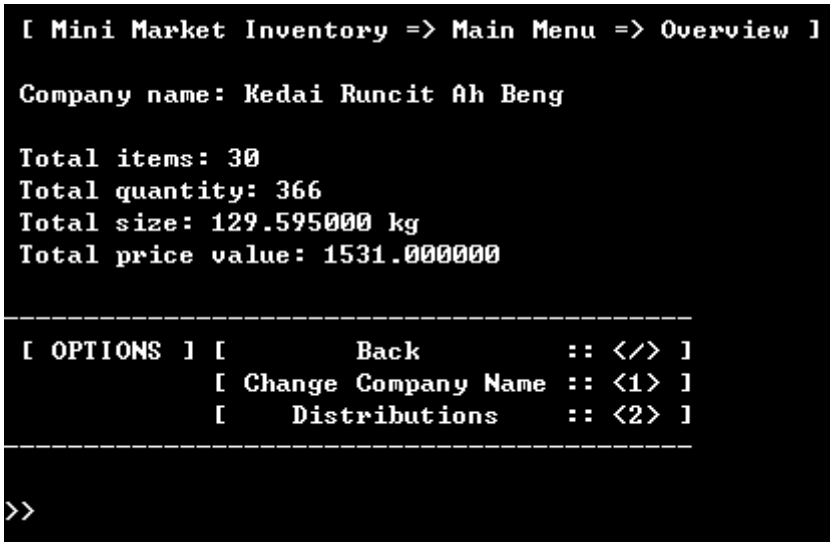
Delete Item



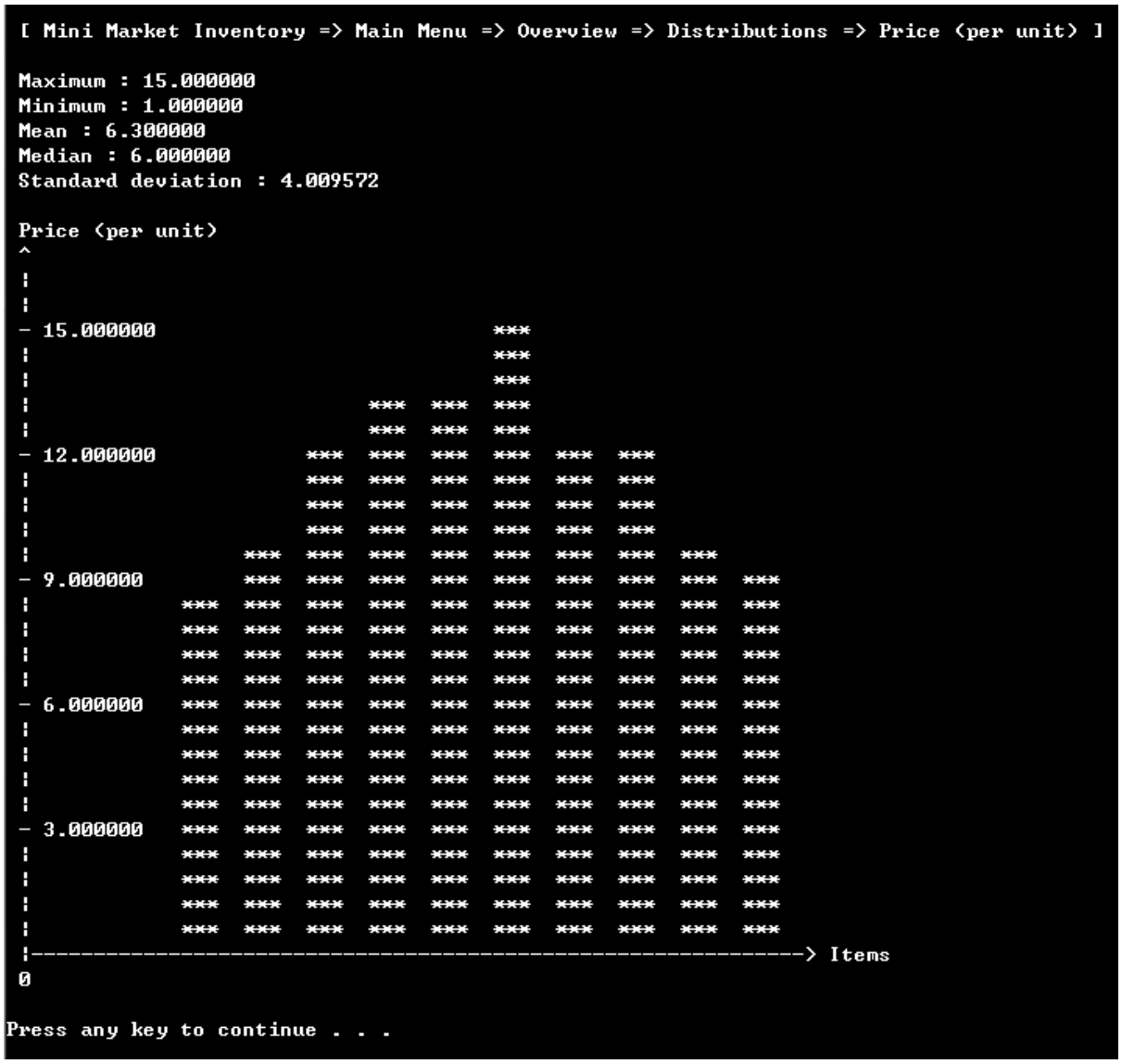
Modify Item



Profile Overview

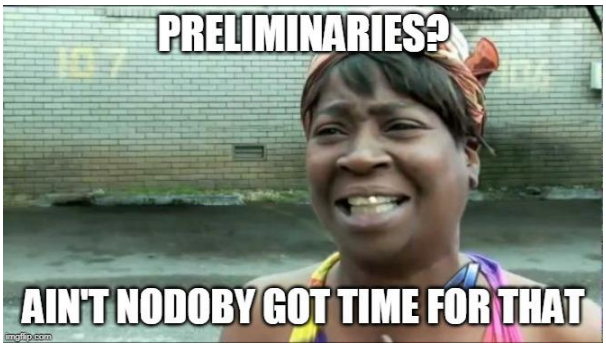


Distributions

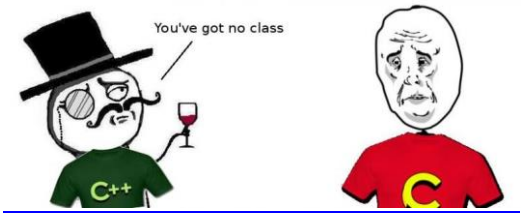


1: Preliminaries

This section is dedicated towards refreshing or introducing readers to some of the data structures and algorithms used in this program.



1.1: Classes



A student asks his lecturer about classes.

Student: What are classes? Are they UTAR's classrooms?

Lecturer: Yes.

Student: Or are they social classes like the proletariats and the bourgeoisie described in the Communist Manifesto?

Lecturer: Yes.

Student: Hmm, I'm confused now. Would you please at least define or elaborate a little on them?

Lecturer: Ok sure. Classes are objects that can be basically anything you want them to be. All you need to do is specify their properties in terms of functions and variables.

Student: I see. Oh, it seems like nature is calling me. May I go to the washroom?

Lecturer: No.

Although it was not the first to introduce it, C++ popularised the object-oriented paradigm way back in the 80s. Much like structs, classes can contain functions and variables. In fact, classes are structs on steroids with powerful capabilities.
To learn more about classes, check out the following website: tinyurl.com/apa-itu-class

1.2: Strings

Strings are fundamentally an ordered collection of characters with theoretically no limit. Characters in this context belong to the primitive *char* data type. For the sake of brevity, we will focus on two of the more relevant types of strings available for use in C++.

C-style strings are declared as an array of characters. To modify the content of a C-string manually, each of its character elements must be accessed and modified. Moreover, it is not possible to alter their lengths. Their fixed lengths are a consequence of the fixed nature of the size of arrays in C++. Without a doubt, the optimal solution to this is standard functions like *strncpy* or *strncat*. However, care must be taken to ensure that the C-string parameters of these functions are properly formatted. For instance, *strncat* concatenates a string A to another string B but results in an error if string B is not sufficient in length to accommodate A. It is also possible to reinvent the wheel and implement C-string operations manually to avoid these errors but they will require an extensive knowledge of pointers and memory allocation. Therefore, it should be obvious by now that C-strings make the program unnecessarily complicated.
More info: tinyurl.com/tak-guna-punya-c-string

Despite inheriting obscure C-style strings from C, C++ has stepped up its game by offering a standard class commonly expressed in code as ***std::string***. Besides offering the same string operations as C-strings, this class allows us to dynamically change its content and size easily without much hassle. It allocates and deallocates memory autonomously and enables us focus on bigger issues rather than trivial ones pertaining to strings.
More info: tinyurl.com/king-of-strings

1.3: String Manipulators

The iomanip library contains stream manipulators that are often used, like *fixed, showpoint* and *setprecision()*. Interestingly, the class ***ostringstream*** extends the use of these manipulators to strings instead. Similar to "cout-ting" or printing something to the console, we can actually utilise the same mechanism to write to strings. This simplifies the process of formatting strings.

More info: tinyurl.com/string-manip

1.4: Vectors

Vectors here are not the multi-coordinate arrows encountered in maths. C++ vectors, or ***std::vector*** in code, are instead a collection of objects, much like arrays. The major distinction is that vectors can adjust their elements and capacity dynamically. Just like std::string, memory allocation is taken care of behind the scenes. Vectors operate in a Last-In-First-Out (LIFO) manner, in the sense that the element last added to it will be the first to be removed. In computer science, vectors can essentially be classified as stacks.

More info: tinyurl.com/wat-is-vector

1.5: Tuples

Functions in C++ can only return one type of value and this can be a severe limitation when we would like the function to return multiple values of different types. One solution to this would be to create a struct or a class which encompasses the multiple types of values and return it from the function. However, this approach appears to be excessive especially if we only need to use it once. Moreover, a different struct/class may have to be created if we want to return a different set of values.

C++ tuples, or ***std::tuple*** in code, are like arrays which can store multiple types of values. One limitation of tuples is that the type of values and their order of arrangement must be specified in advance before initialisation. For instance, a tuple of the arrangement (double, int, string) is **not equivalent** to a tuple (double, string, int) because their order are different. Nevertheless, they are a practical solution better than the one outlined earlier.

More info: tinyurl.com/tuple-bantu-saya

1.6: Windows-Specific Functions

The library *windows.h* enables us to access a couple of Windows-specific functions with ease.

a) *Sleep(milliseconds)*

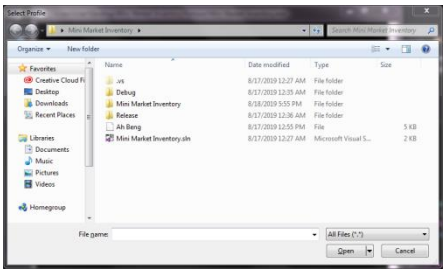
- This function delays the program depending on the parameter. The delay is specified in milliseconds.

b) *SetConsoleTextAttribute(color)*

- This function sets the color of the console text depending on the input parameter. The parameter is actually an integer and is only valid from 1 to 255. Each value corresponds to a unique color and style.

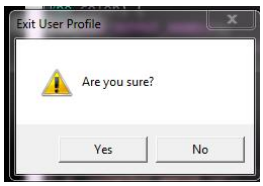
c) *OPENFILENAME*

- This class invokes a Windows-style explorer which prompts the user to select a file. It is a complicated system and one must go through the official documentation to utilise it properly. More info: tinyurl.com/give-me-my-file



d) *MessageBox(options)*

- Displays a box with a couple of options for the user to select from and then returns a value which signifies the selected option. The options available must be specified as a parameter of the function. More info: tinyurl.com/box-of-messages



1.7: Lambda functions

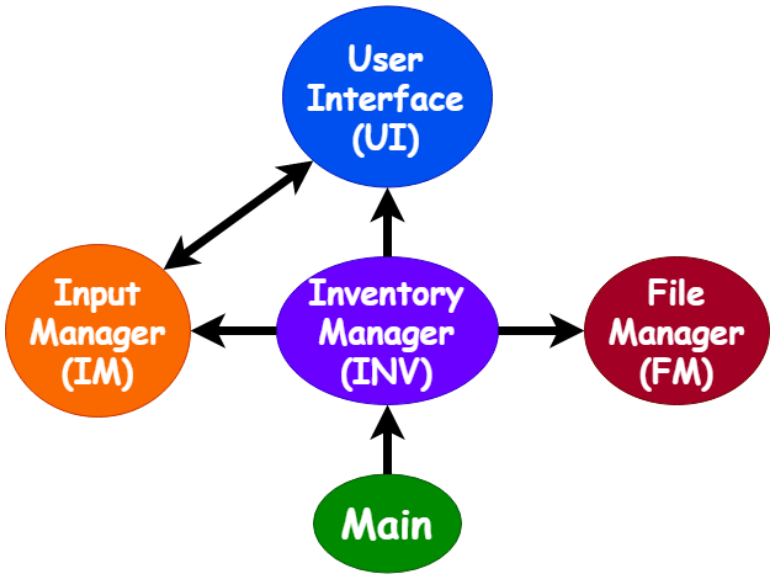
Imagine if we could encapsulate functions in variables. We would be able to dynamically invoke different functions without having to manually code for every specific case. Another desirable trait would the generalisation of function parameters or, in other words, the types of parameters of a particular function need not be specified in advance. Both of these fascinating concepts are one line of code away with the emergence of lambda functions in C++. Since our description here has been pretty vague, the reader should check out the following webpage for a clearer exposition: tinyurl.com/lambda-is-not-a-lamb

1.8: Bubble Sorting

Sorting a list of items either in a ascending or descending order has been a subject of scrutiny in computer science for a long time. Out of the many algorithms available, we decided to stick with bubble sort due to its simplicity and relatively good performance. More info: tinyurl.com/bubbles-are-good-for-you

2: Program Overview

To simplify its development, we have compartmentalised the program into five components. The following diagram summarises the program:



To simplify things, every component will referred to by their respective acronyms. The arrows indicate the dependency of the components on one another. For instance, INV is dependent on FM whereas IM and UI are mutually dependent. Meanwhile, FM is independent from the others. Although INV is Main's only dependency, Main indirectly relies on the other components via INV's dependencies on them.

3: Classes Assemble!

3.1: Why Classes?

Despite classes not being taught in the syllabus, we have nevertheless decided to construct the program as a collection of self-contained classes which interact with one another. Technically, some of the classes are not purely autonomous as they derive a portion of their functions from others.

This compartmentalisation allows for flexible code and faster debugging. In fact, the inclusion of classes in this program is merely natural when the program grows in complexity. Since the program is loosely founded on the rocks of the object-oriented paradigm, it should be much easier to extend it to include a wider range of features in the future. However, the various facets of classes like polymorphism or inheritance are not yet used as the program is not sufficiently sophisticated.

Despite having proclaimed its benefits, we admit that the inclusion of classes here is quite superficial without much apparent advantage. Constructors and destructors are not even utilised. The main motivation behind using classes is their ability to minimise redundancy in a clear and concise manner. With the use of static variables and functions, many of the classes are ostensibly singletons. In other words, only one copy or instance of themselves exists. This comes in handy for classes which particularly handle user input, the user interface and file operations. Duplicate instances of the same class may result in inconsistencies in the program if not handled properly. In reality, they are not exactly singletons because it is entirely possible to construct duplicates of them. The proper approach to creating singleton classes involves some complicated memory allocation techniques so we avoided it.

Indeed, the classes could have been replaced with a couple of structs and the corresponding functions and variables could be embedded into each of the structs instead. This approach is a little flawed though because it results in duplicate instances of a particular struct when the struct is used by two or more other structs. For example, a struct A1 has to initialise a variable for the struct C to use it. Not only that, another struct B1 has to initialise its own copy of struct C to use it, resulting in two duplicate copies of struct C. To address this issue, the main thread/function has to declare a variable for struct C and then, using some reference mechanism or pointers, ensure that the two other variables of struct C in structs A1 and B1 are pointing to the same instance of C declared earlier in the main thread. This approach is unnecessarily convoluted and so we ditched it.

Another alternative would be to enclose the corresponding functions in namespaces. For instance, namespaces A2 and B2 are distinct and can use a namespace C via the declaration “using namespace C;” without having to deal with multiple instances of it. Furthermore, variables in namespaces are by default static so duplication is not an issue.

Essentially, the class-based approach is no different from the namespace-based approach. Both involve static variables and minimises redundancy. The only distinction lies in the adaptability or extensibility of the two approaches. As mentioned early on, the former (class) can easily adapted to include more features such as multiple distinct inventory lists in a single profile or even multiple profiles in a single file. The possible abstractions are limitless because classes can easily evolve to integrate with one another via inheritance and other features exclusive to class objects. On the other hand, namespaces are rigid and are difficult to extend to more general cases. Moreover, duplicate instances of a namespace are not possible when required.

In short, we opted for classes because they are optimal for the development cycle of the program. All talk and no practical results do not mean much. Let us dive into the details now:

3.2: Colour

This namespace handles the text colour of the console. It is the only exception where class is not utilised because no difference is made even if we turned it into a class.

Variables:

1) **ColorType**
   ```
   enum class ColorType { DEFAULT = 7,… };
   ```
   Description: Not a variable, but an enum class which contains standard values for various colours.

---

2) **defaultColor**
   ```
   static ColorType defaultcolor = ColorType::WHITE;
   ```
   Description: The default colour is white.

Functions:

1) **ChangeColor()**
   Prototype:
   ```
   static void ChangeColor(ColorType color);
   ```
   Description: Changes the colour using *SetConsoleTextAttribute()*.

---

2) **ShowAllColors()**

Prototype:

```
static void ShowAllColors();
```

Access: Public

Description: Displays all 255 colour styles available in the console. Only used for debugging purposes.

3.3: File Manager (FM)

FM basically handles anything pertaining files. He is the greatest sorcerer in the land and is a mediator to the deity called Windows.

Variables:

1) **delimiter**

Header:

```
static const char delimiter;
```

Implementation:

```
const char FileManager::delimiter = '\n';
```

Access: Public

Description: The delimiter used to distinguish between strings obtained from files. In the program, the line break, *'\n'*, is the delimiter.

---

2) **filePath**

Header:

```
static std::string filePath;
```

Implementation:

```
string FileManager::filePath = "";
```

Access: Private

Description: The delimiter used to distinguish between strings obtained from files. In the program, the line break, *'\n'*, is the universal delimiter.

Functions:

1) **SelectFile()**

Prototype:

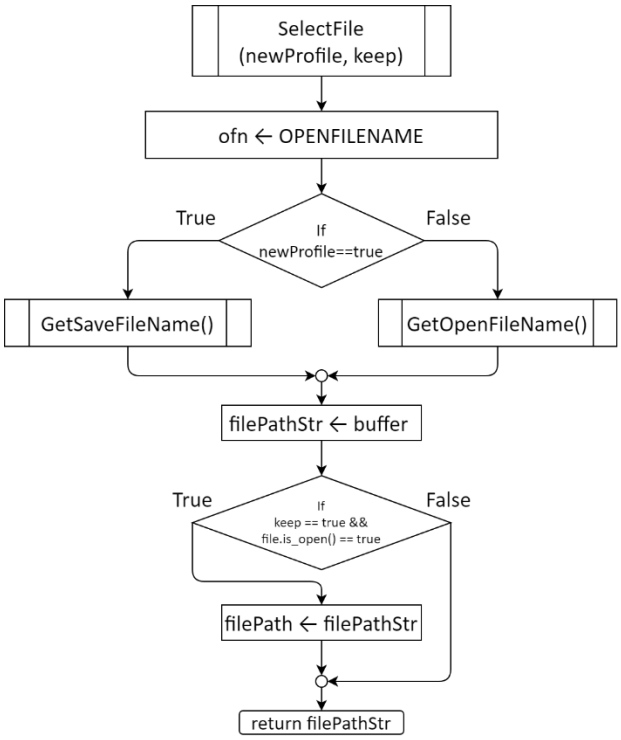
```
static std::string SelectFile(const bool newProfile, const bool& keep);
```

Access: Public

Description: This function utilises a variable of the class OPENFILENAME to initiate a file

explorer. If *newProfile* is true, it informs the system that the user intends to save a file. Otherwise, it will assume that the user is loading a file instead. The variable *buffer* stores the chosen file path as a char array. The function then converts *buffer* to an *std::string* stored in *filePathStr*. If *keep* is true and the file is valid, *filePath* is set to *filePathStr*. Finally, it returns *filePathStr.*

Flowchart:



---

2) **ReadFile()**

Prototype:

```
static std::ifstream ReadFile(const std::string& path);
```

Access: Public

Description: Returns an *ifstream* with the *path* parameter as the file location. Flowchart not required as it contains just one line of code.

---

3) **UpdateFile()**

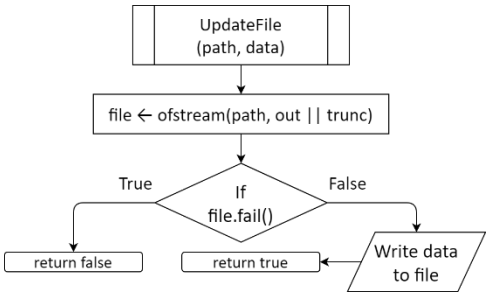
Prototype:

```
static const bool UpdateFile(const std::string& path, const std::string&
data);
```

Access: Public

Description: Using the parameter *path*, this function accesses a file and writes the string *data* to it. The use of *ios::out* and *ios::trunc* ensures that the file is data-wiped first being being modified.

Flowchart:



---

4) **DeleteFile()**

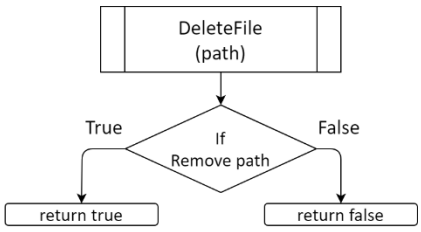
Prototype:

```
static const bool DeleteFile(const std::string& path);
```

Access: Public

Description: Using the parameter *path*, this function supplies a file path to *remove()* which deletes the file if it exists. If the operation fails, false is returned.

Flowchart:



---

5) **SetFilePath()**

Prototype:

```
static void SetFilePath(const std::string& path);
```

Access: Public

Description: Sets the private variable *filePath* to *path*.

---

6) **GetFilePath()**

Prototype:

```
static void GetFilePath();
```

Access: Public

Description: Returns the private variable *filePath*;

3.3: User Interface (UI)

UI handles any output to the console in general and is a completely public class. She has a marketing degree from UTAR and wants to brighten people's lives using a black-and-white Windows console.

Variables:

1) **TableColumn**

Header: `struct TableColumn {std::string title, …};`

Description: A struct definition which represents a table column. Its fields relate to the display format of the column, such as the precision and width.

---

2) **defaultColumns**

Header: `static const std::vector<TableColumn> defaultColumns;`

Implementation:

```
const vector<UI::TableColumn> UI::defaultColumns {
{ "Number", 0, 6, false, true, false }, … }
```

Description: Stores the default columns in a vector. These columns are used when displaying the default table in the inventory.

---

3) **printInterval**

Header: `static constexpr DWORD printInterval = 0;`

Description: The time interval between displaying characters and only used to emulate a typing animation when displaying stuff. It is set to zero by default because we found that even a delay of 1 millisecond is too slow due to the possibly significant overhead of the *Sleep()* function.

---

4) **sleepInterval**

Header: `static constexpr DWORD sleepInterval = 1000;`

Description: The default time interval when a pause is required. It is 1 second by default.

---

5) **maxDots**

Header: `static constexpr int maxDots = 2;`

Description: The number of trailing dots to display in the table when a particular value exceeds the maximum width. The default value is 2.

---

6) **rootTitle**

Header: `static const std::string rootTitle;`

Implementation: `const string UI::rootTitle = "Mini Market Inventory";`

Description: Titles in the program are chained to one another and displayed. At any stage of the program, the *rootTitle* will always be the first in the chain of titles.

---

7) **inputMarker**

Header: `static const std::string inputMarker;`

Implementation: `const string UI::inputMarker = "\n\n>> ";`

Description: This string will be displayed when user input is required. Rather than hardcoding it all over the program's code, we store it in a variable so that any modifications to it will be applied universally.

---

8) **optionList**

Header: `static std::vector<std::string> optionList;`

Implementation: `vector<string> UI::optionList { "" };`

Description: Stores the names of options to display when user input is required.

---

9) **optionKeyList**

Header: `static std::vector<std::string> optionKeyList;`

Implementation: `vector<string> UI::optionKeyList { "" };`

Description: Stores the corresponding keys of options to display when user input is required.

Functions:

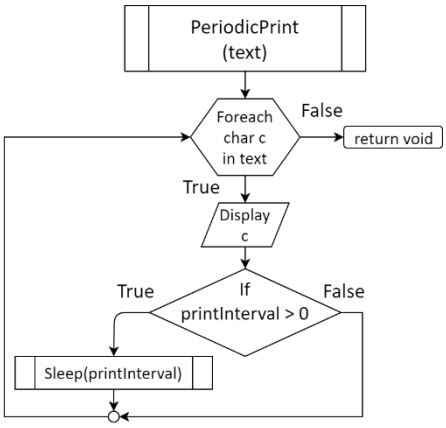
1) **PeriodicPrint()**

Prototype:

```
static void PeriodicPrint(const std::string& text);
```

Description: Prints the parameter *text* with delays between its characters. The variable *printInterval* is used as the delay period. This function is also overloaded to accept a custom time delay.

Flowchart:



---

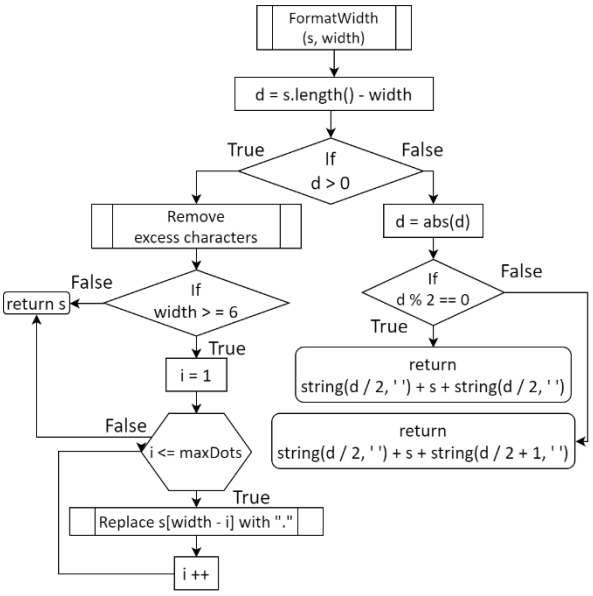
2) **FormatWidth()**

Prototype:

```
static std::string FormatWidth(std::string& s, int width);
```

Description: Adjusts the string, *s*, such that its length is the same as *width*. If the initial length is shorter than *width*, the contents of the string are centred with empty spaces at the sides. For instance, *"abc"* becomes *" abc "* if *width* is 5. If the number of empty spaces is odd, the greater half is distributed to the right side. If the initial length is longer, characters beyond the final allowed position are removed. After the removal, the last few remaining characters are replaced with trailing dots. The number of dots depends on the variable *maxDots*. Assuming that *width* is again 5, both *"abcdef"* and *"abcdefgh"* become *"abc.."*.

Flowchart:



---

3) **GenerateOptionsList()**

Prototype:

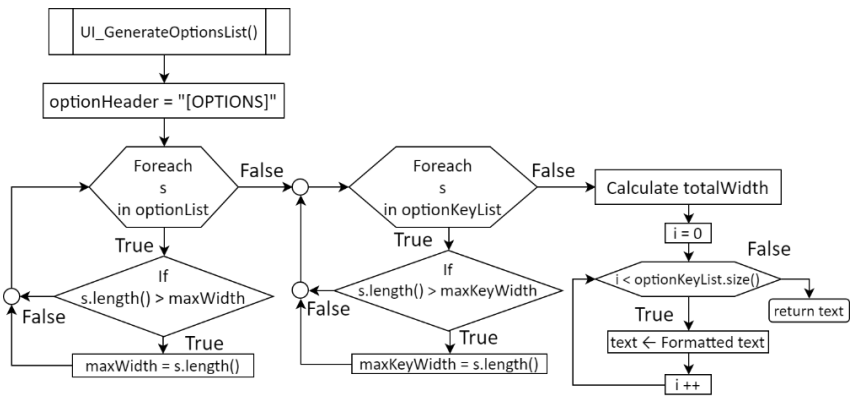
```
static std::string GenerateOptionsList();
```

Description: Generates a list of options for the user to select from. The list of options and their corresponding keys are acquired from the variables *optionList* and *optionKeyList*. The names of the options and their respective keys are properly spaced out using *FormatWidth()* to accommodate the longest name in the list.

Example:

```
-----------------------------------
[ OPTIONS ] [     Back       :: </> ]
            [   Sort List    :: <1> ]
            [ Low Inventory  :: <2> ]
            [   Add Item     :: <3> ]
            [  Modify Item   :: <4> ]
            [  Delete Item   :: <5> ]
-----------------------------------
```

Flowchart:



4) **GenerateTableRow()**

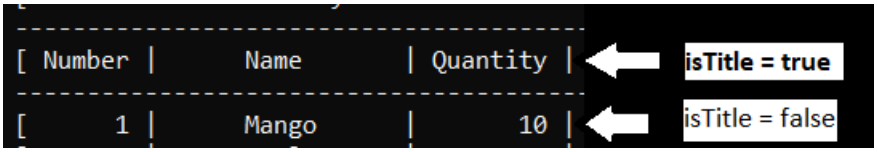
Prototype:

```
static std::string GenerateTableRow(bool isTitle, std::vector<TableColumn> format, std::vector<std::string> values, std::vector<std::string> units, const int unitsMaxLength);
```

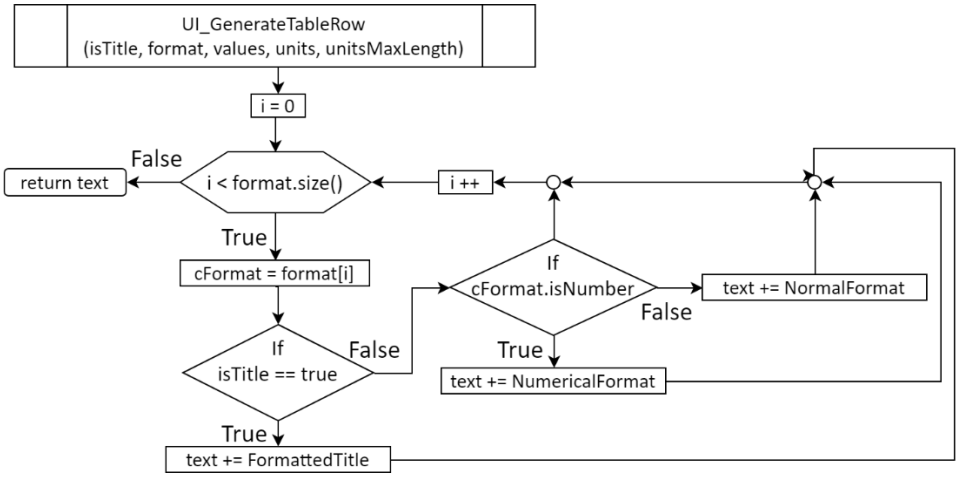
Description: Generates a row of items based on the *TableColumn* vector. The length of the vector *format* determines the number of columns. Not only that, units are appended to the values if required. Moreover, it is assumed that only numerical values have units. Although the width is maintained using *FormatWidth()*, the net width of a column is the sum of its default width in *TableColumn* and the length of its corresponding string of units.

When invoking this function, the value of *unitsMaxLength* must be pre-determined. If the values are numerical, the ***ostringstream*** class is used to apply their corresponding format. Otherwise, a normal, unsophisticated formatting is used instead. If *isTitle* is true, the default titles in *format* will be printed instead of the elements in the *values* vector.

Example:



Flowchart:



5) **GenerateTableBorder()**

Prototype:

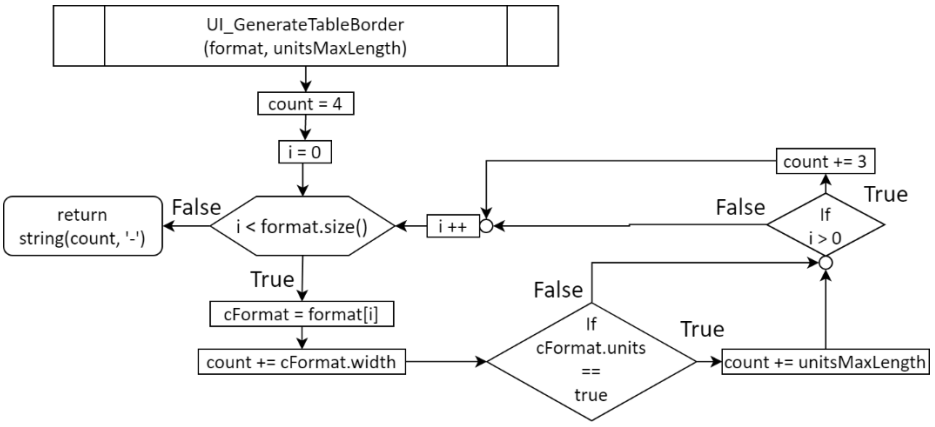
```
static std::string GenerateTableBorder(std::vector<TableColumn> format, const int unitsMaxLength);
```

Description: Generates a dynamic horizontal border consisting of dashes whereby its length is dependent on the elements in the *TableColumn* vector. With the inclusion of "magic numbers" 3 and 4, it has been fine-tuned to produce a border which is visually satisfying and accurate even if the columns are modified. (Edit: Instead of requiring the parameter *unitsMaxLength*, we could have iterated through the entire *units* vector and determined the longest unit length but it is too late to edit now because we have no time left.)

Example:

```
[ Number |      Name       | Quantity |  Size (per unit)   |    Net Size    | Price (per unit) |  Price (per weight)  | Net Price |      Brand      |     Supplier     |     Contact     ]
```

Flowchart:



---

6) Title()

Prototype:

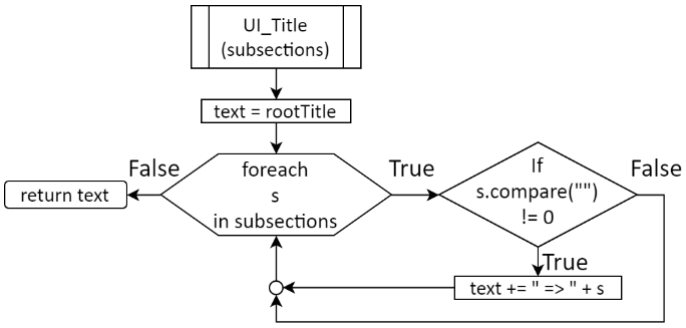
```
static std::string Title(std::vector<std::string> subsections);
```

Description: Iterates through the string vector and returns a string displaying an ordering of the elements. The variable *rootTitle* forms the first title. The first element of the vector is the second title whereas the last element is, well, the last title. If an element is empty, it is ignored.

Example:

```
[ Mini Market Inventory => Main Menu => View Items => Choose Item Type ]
```

Flowchart:



---

7) option_YesNo()

Prototype:

```
static std::string option_YesNo();
```

Description: Returns a binary list of options using *GenerateOptionsList()*.

Example:

```
----------------------------
[ OPTIONS ] [ Yes  :: <y> ]
            [  No  :: <n> ]
            [ Exit :: </> ]
----------------------------
```

---

8) option_Single()

Prototype:

```
static std::string option_Single();
```

Description: Returns a single “Back” option using *GenerateOptionsList()*.

Example:

```
----------------------------
[ OPTIONS ] [ Back :: </> ]
----------------------------
```

Class Summary:

UI is designed to display option lists and table rows dynamically. The format specified in *defaultColumns* is a good demonstration of these capabilities. One great advantage of this approach is that columns can be added, removed and modified with ease as long as the format is specified correctly. Moreover, the chaining of titles of different sections of the program minimise redundant code when displaying the titles for each section.

3.4: Input Manager (IM)

IM handles any user input from the console in general and is an entirely public class. He takes his role too seriously and impartially berates users if they give the wrong answers.

Variables:

1) **InputErrorType**

   Header: `enum class InputErrorType {GENERAL, EMPTY_INPUT, …};`

   Description: Not a variable, but an enum class representing various error types.

---

2) **yesKey, noKey, exitKey**

   Header: `static const std::string yesKey, noKey, exitKey;`

   Implementation: `const string yesKey = "y", noKey = "n", exitKey = "/";`

   Description: These three variables are the default keys for the options they represent.

---

3) **inputSize**

   Header: `static constexpr int inputSize = 30;`

   Description: The maximum user input length allowed. Actually, with the use of *std::string*, a maximum need not be imposed. Nevertheless, we decided to go with a limit for the sake of simplicity.

Functions:

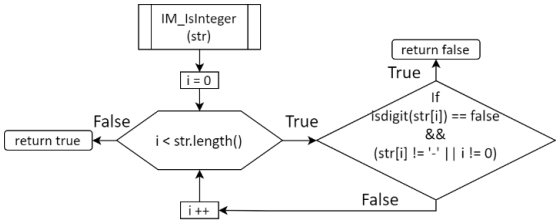
1) **IsInteger()**

   Prototype:

   `static bool IsInteger(const std::string& str);`

   Description: Checks if the string can be converted to a valid integer. The negative sign, -, is a false negative, so it is taken into account in the *if* condition.

   Flowchart:



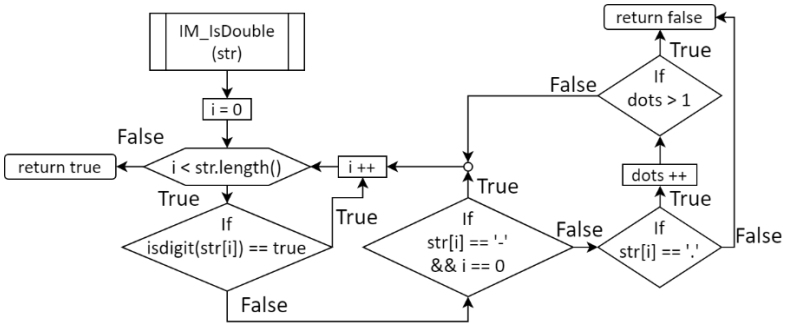
2) **IsDouble()**

Prototype:

```
static bool IsDouble(const std::string& str);
```

Description: Checks if the string can be converted to a valid double. The negative sign, -, is taken into account. Moreover, the dot/point character is another false-negative so it is ignored. However, the string is still invalid if there is more than one dot.

Flowchart:



3) **Flush()**

Prototype:

```
static void Flush(std::istream& in);
```

Description: In the early stages of the program's development, this function was invoked to flush the buffer whenever remnants of past inputs are left uncleared in the input/*cin* buffer due to errors. However, it is now made obsolete as the input system has been migrated to a string-based one where "flushing" is no longer necessary. Nevertheless, we leave this function unharmed in case we need to "flush the toilet again". All it does now is clear the *cin* flags using *cin.clear()*.

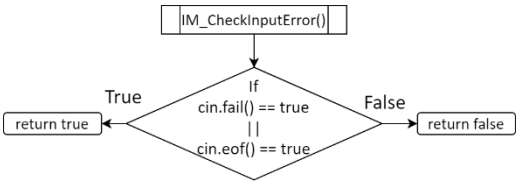
4) **CheckInputError()**

Prototype:

```
static bool CheckInputError();
```

Description: This function returns true if the input somehow fails. If the user inputs the EOF character, the program also assumes that the input failed.

Flowchart:



---

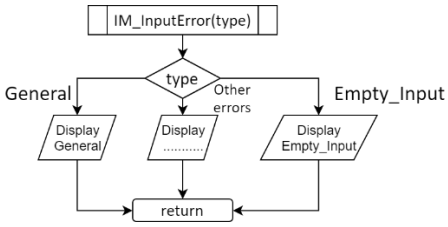
5) **InputError()**

Prototype:

```
static void InputError(InputErrorType type);
```

Description: Displays a message corresponding to the error type parameter.

Flowchart:



---

6) **RemoveWhiteSpace()**

Prototype:

```
static void RemoveWhiteSpace(std::string& s);
```

Description: Erases white spaces using an internal function of *std::string*.

---

7) **ToLowerCase()**

Prototype:

```
static void ToLowerCase(std::string& s);
```

Description: Changes all alpha characters to lower case using *transform()* from C++'s standard algorithms library.

---

8) **GetString()**

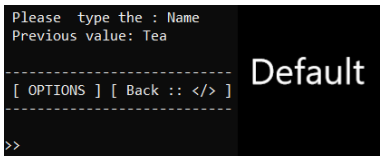
Prototype:

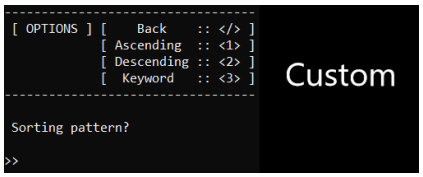
```
static const bool GetString(std::string& placeholder, const bool& showPrevious, const std::string& title, const std::string& name, const std::string& custom);
```

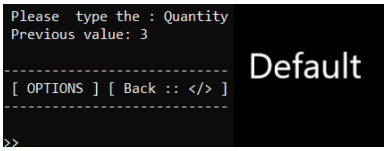
Description: Prompts the user to key in a string which will be stored in the variable *placeholder*. After displaying the title, the function proceeds to display a default interface whereby the user is prompted to key in some value for a variable.

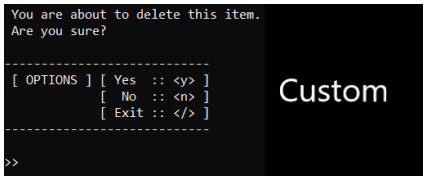
The string *name* represents the placeholder's name. If *name* is empty, the default format is not used and the string *custom* is displayed instead. In this case, *custom* is assumed to contain formatted data which deviates from the default format. It is up to the programmer to decide between the two. It is also his/her responsibility to format the custom string to suit the context in which the function is invoked. Lastly, the previous/initial value of the placeholder is shown if *showPrevious* is true.

Example:

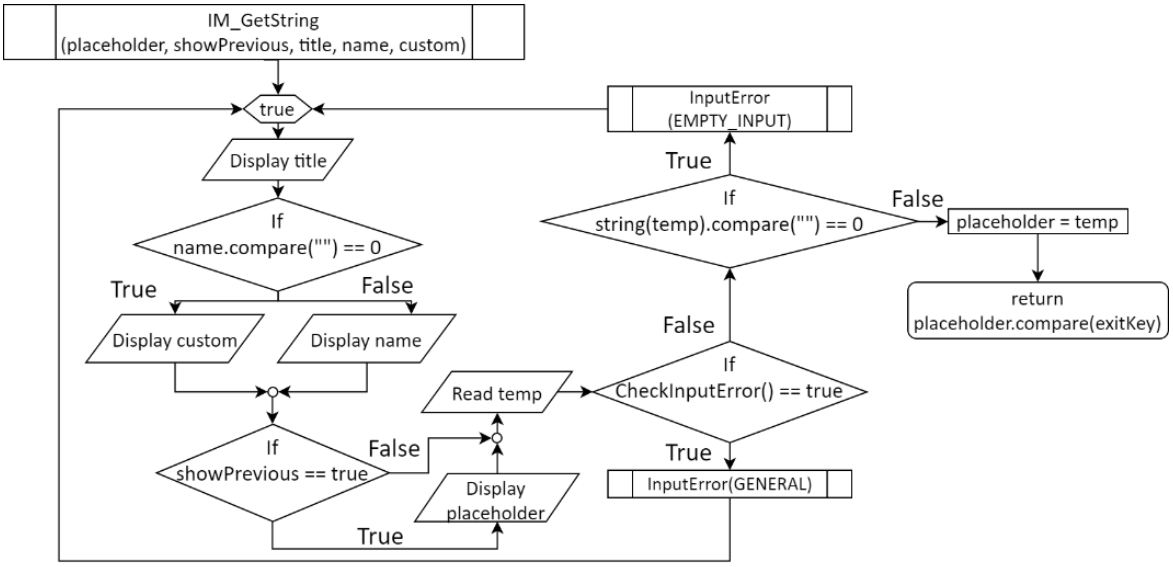








Flowchart:



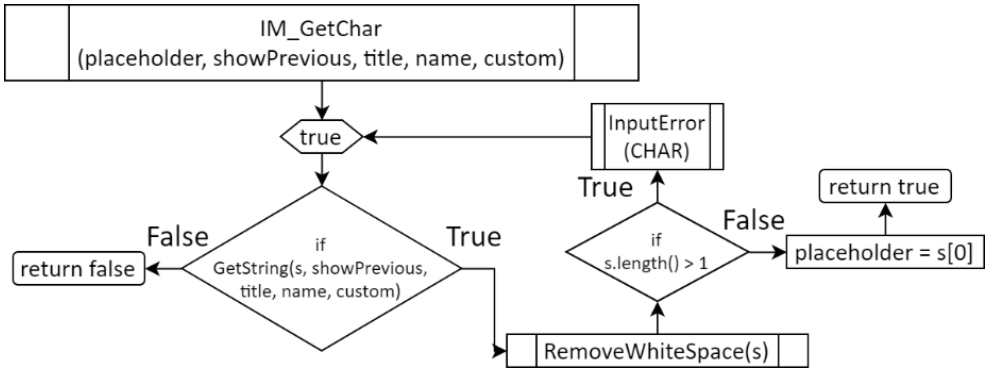
9) **GetChar()**

Prototype:

```
static const bool GetChar(char& placeholder, const bool& showPrevious, const std::string& title, const std::string& name, const std::string& custom);
```

Description: *GetString()* is invoked to acquire the user's input which will be converted to *char* and finally stored in the placeholder.

Flowchart:



---

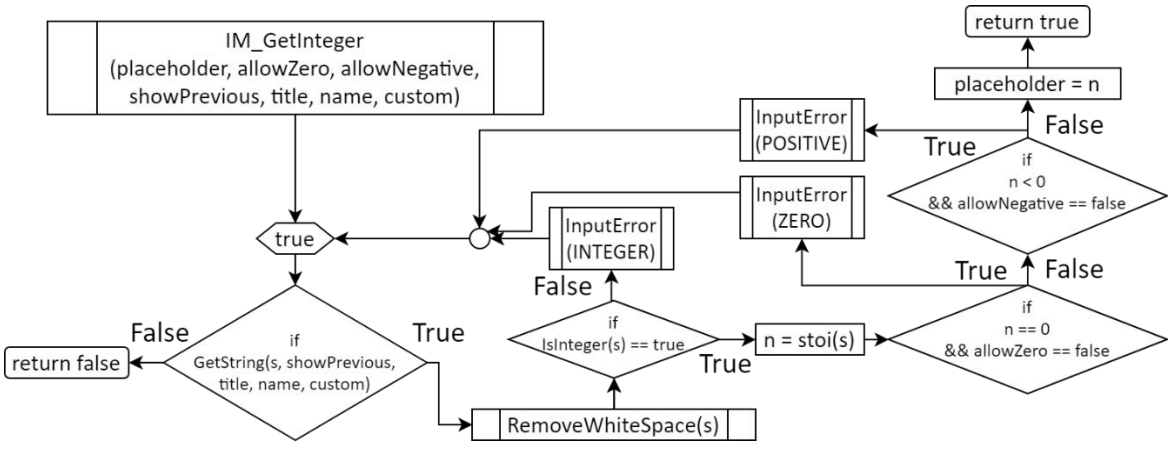
10) **GetInteger()**

Prototype:

```
static const bool GetInteger(int& placeholder, const bool& allowZero, const bool& allowNegative, const bool& showPrevious, const std::string& title, const std::string& name, const std::string& custom);
```

Description: *GetString()* is invoked to acquire the user's input which will be converted to *int* and finally stored in the placeholder. Zero is not allowed when *allowZero* is true whereas negative numbers are permitted when *allowNegative* is true. Other than those two parameters, the rest are passed to *GetString()*.

Flowchart: .



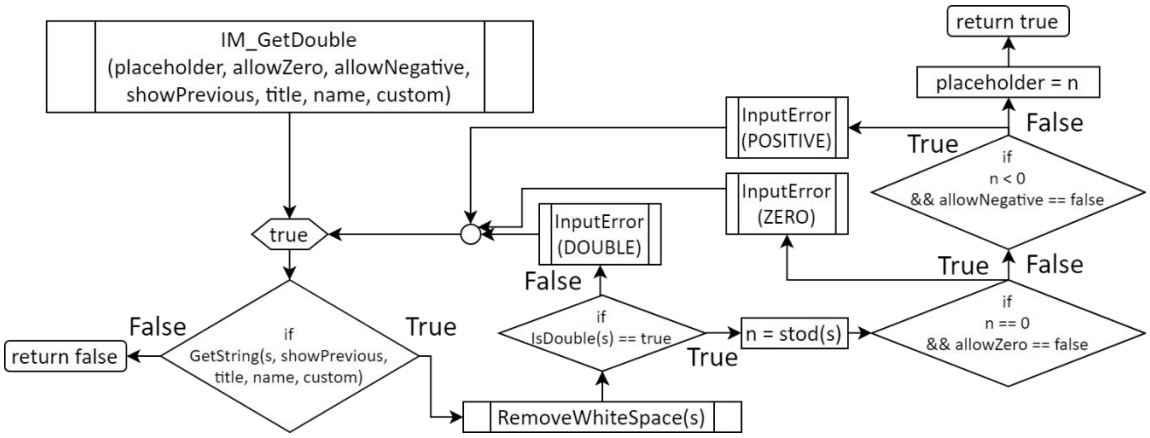
11) **GetDouble()**

Prototype:

```
static const bool GetDouble(double& placeholder, const bool& allowZero, const bool& allowNegative, const bool& showPrevious, const std::string& title, const std::string& name, const std::string& custom);
```

Description: Acquire a double from the user and stores it in the placeholder. It is almost identical to *GetInteger()*.

Flowchart:



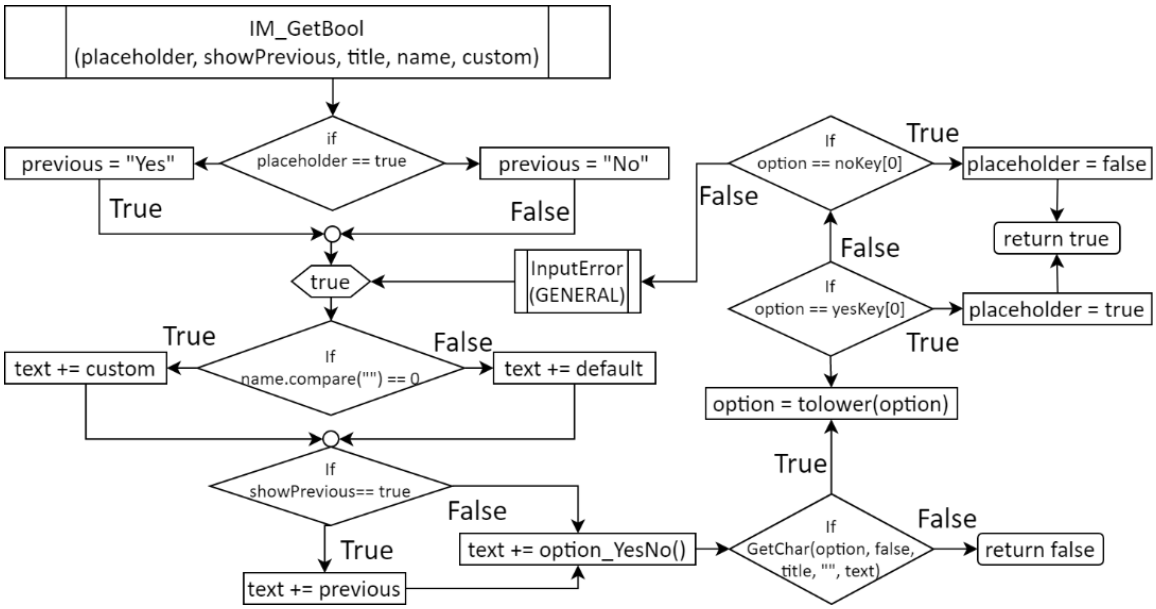
12) **GetBool()**

Prototype:

```
static const bool GetBool(bool& placeholder, const bool& showPrevious, const std::string& title, const std::string& name, const std::string& custom);
```

Description: Obtains a Boolean from the user input and stores it in the placeholder. Since the input is expected to be purely binary, a custom string is channelled to *GetChar()* with the options generated by *option_YesNo()*.

Flowchart:



Class Summary:

Essentially, IM revolves around the function *GetString()*, which serves to obtain the user's input in its "purest" form, that is in strings. Strings are "pure" in the sense that they can morphed easily into other data types with ease. Closely-related functions like GetBool and GetInteger work upon these foundations and narrow down the input into more useful forms.

Since every data type has its own idiosyncrasies, error-checking mechanisms may vary from one to another. An error-detection mechanism may work well for a specific data type, but fail miserably for another. Hence, a distinct conversion function is allocated for each of the fundamental data types. The conversion function for each data type can execute its own unique error-checking algorithm making it easier to screen for errors in the user input in general.

With the foundations in place, it is now relatively easy to acquire user input. Gone are the days where similar copies of the same code for user input are littered in various parts of the program.

3.5: Inventory Manager (INV)

INV manages the inventory and knows it inside out. She actively searches for dust and keeps the storeroom sparkly clean.

Types:

1) **ValueType**

   Header: `enum class ValueType {INT, DOUBLE, STRING, BOOL};`

   Access: Public

   Description: An enum class representing some fundamental data types.

---

2) **Item**

   Header: `struct Item {int itemNumber = 0, …};`

   Access: Public

   Description: A struct definition for an inventory item.

---

3) **ItemType**

   Header: `enum class ItemType {NUMBER, NAME, QUANTITY, …};`

   Access: Public

   Description: An enum class representing the types of data in an inventory item.

---

4) **ItemTypeInfo**

   Header: `struct ItemTypeInfo {std::string name, …};`

   Access: Public

   Description: A struct definition containing deeper characteristics of an item type.

---

5) **SortType**

   Header: `enum class SortType {GENERAL = 0, ASCENDING, …};`

   Access: Public

   Description: An enum class representing the types of sorting available.

---

6) **SortItemType**

Header: `enum class SortItemType {GENERAL = 0, NUMBER, …};`

Access: Public

Description: An enum class representing the types of data that can be sorted.

Variables:

1) **e1, e2, e3, e4**

Header: `int e1; double e2; std::string e3; bool e4;`

Access: Public

Description: Empty variables which are used as empty references in *PtrFromItemType()*.

---

2) **companyName**

Header: `std::string companyName;`

Access: Public

Description: Stores the company name.

---

3) **maxItems**

Header: `static constexpr int maxItems = 100;`

Access: Public

Description: The maximum number of items in the inventory. The default value is 100.

---

4) **inventory**

Header: `Item inventory[maxItems];`

Access: Private

Description: An array of structs containing the entire inventory.

---

5) **itemInfos**

Header: `static const std::vector<ItemTypeInfo> itemInfos;`

Implementation: `const vector<ItemTypeInfo> itemInfos{{“Number”, …}, …};`

Access: Private

Description: A vector of default *ItemTypeInfo*s.

---

6) **unitsMaxLength**

Header: `static const int unitsMaxLength = 5;`

Access: Private

Description: The maximum length of the string of units.

---

7) **inventoryLength**

Header: `int inventoryLength;`

Access: Private

Description: The inventory's length.

Functions:

1) **IsValidItem()**

Prototype:

```
const bool IsValidItem(Item& item);
```

Access: Private

Description: Returns true if item number is greater than zero.

---

2) **UpdateInventoryLength()**

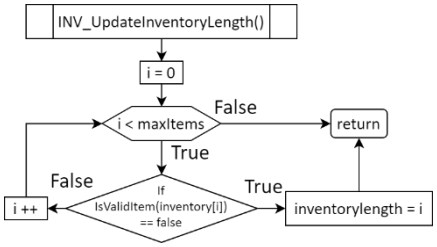
Prototype:

```
void UpdateInventoryLength();
```

Access: Private

Description: Updates the inventory length.

Flowchart:



3) **GetItemIndex()**

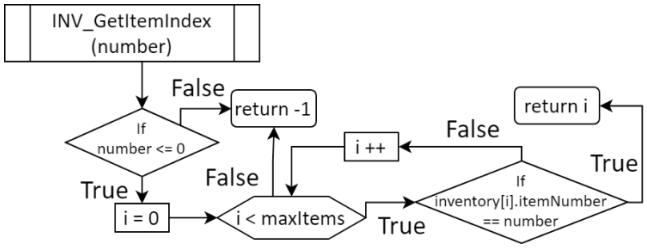
Prototype:

```
const int GetItemIndex(const int number);
```

Access: Private

Description: Returns the corresponding item index using the item number. -1 is returned if no index is found.

Flowchart:



4) **IsUsedItemNumber()**

Prototype:

```
const bool IsUsedItemNumber(const int number);
```

Access: Private

Description: Returns *true* if number is used by one of the items. GetItemIndex() is used to invoked to check this condition.

5) **PtrFromItemType()**

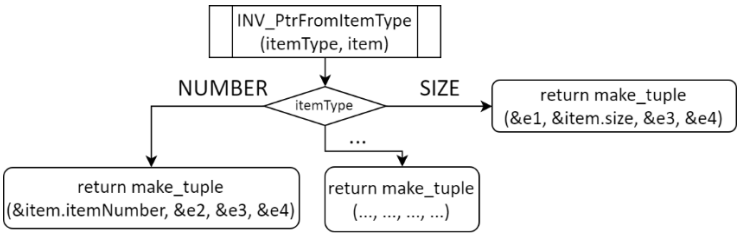
Prototype:

```
std::tuple<int*, double*, std::string*, bool*> PtrFromItemType(ItemType itemType, Item& item);
```

Access: Private

Description: Returns a tuple containing a pointer to a variable in the item which corresponds with the Item type. For instance, a pointer to the variable *size* is returned when *SIZE* is the item type. Depending on the item type, only one of the pointers is significant while the rest are empty.

Flowchart:



---

6) **ValueFromSortItemType()**

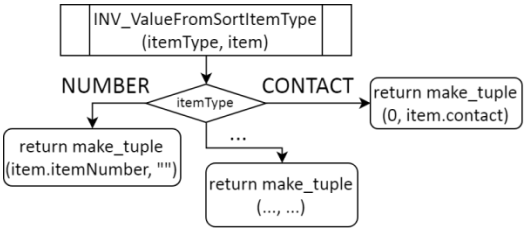
Prototype:

```
std::tuple<double, std::string> ValueFromSortItemType(SortItemType itemType, Item& item);
```

Access: Private

Description: Similar to *PtrFromItemType()*, this function returns a value in an item which corresponds with the item type. However, it only returns the value and not the address. It also crucial to note this function is limited to *SortItemType* whereas *PtrFromItemType()* caters to *ItemType*.

Flowchart:



---

7) **UpdateItems()**

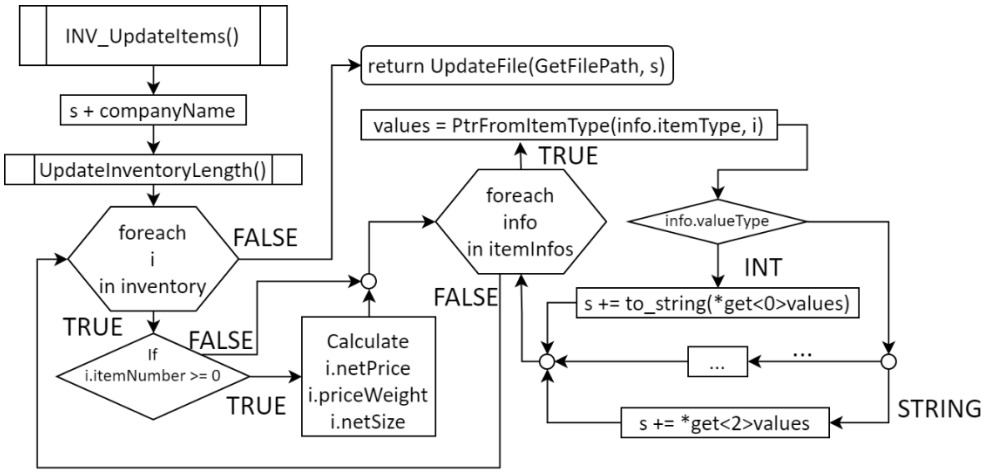
Prototype:

```
const bool UpdateItems();
```

Access: Private

Description: Updates the program's data by recalculating some values and also updating the user file.

Flowchart:



---

8) **LoadItems()**

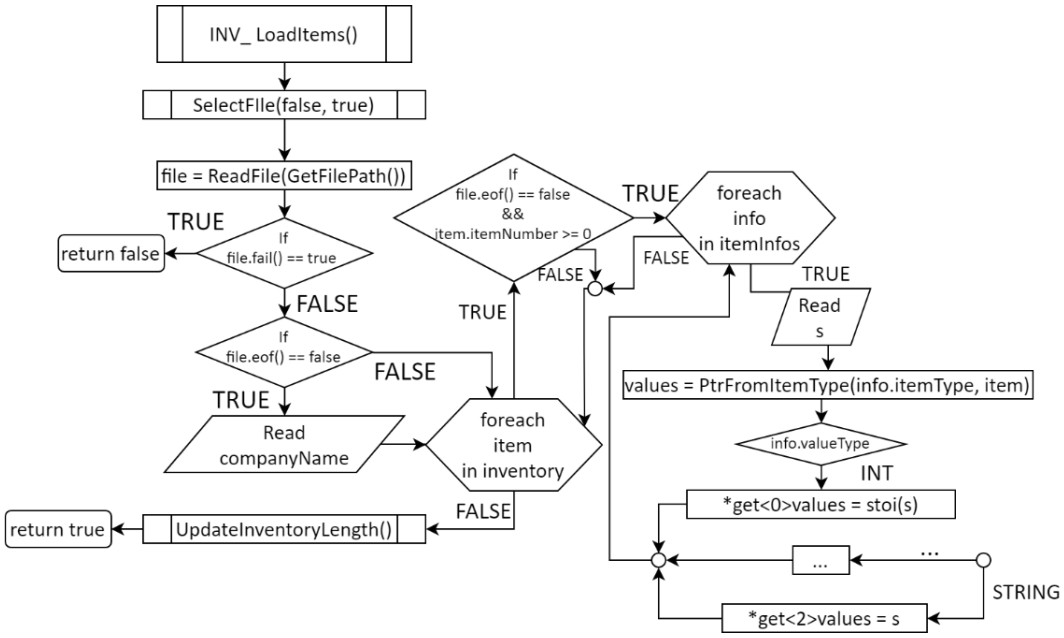
Prototype:

```
const bool LoadItems();
```

Access: Public

Description: Acquires data from the user's file and stores them in relevant variables. .

Flowchart:



---

9) **ChangeCompanyName()**

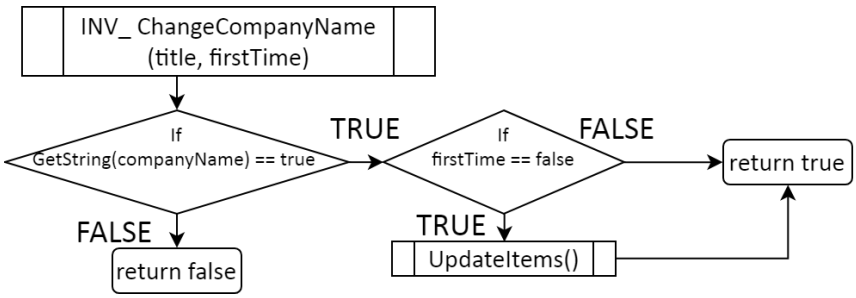
Prototype:

```
const bool ChangeCompanyName(std::vector<std::string> title, const bool firstTime);
```

Access: Public

Description: Prompts the user to key in a new company name.

Flowchart:



---

10) **NewProfile()**

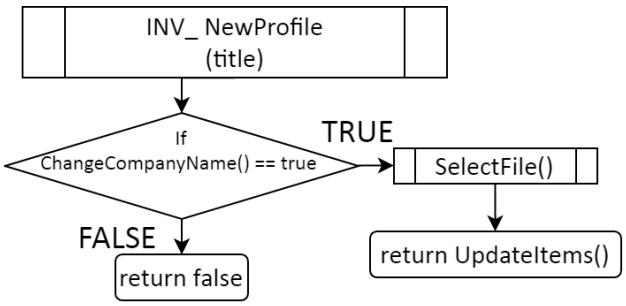
Prototype:

```
const bool NewProfile(std::vector<std::string> title);
```

Access: Public

Description: Handles the process of creating a new user profile. The company name is specified first, followed by the file location of the profile.

Flowchart:



---

11) **GenerateItemsTable()**

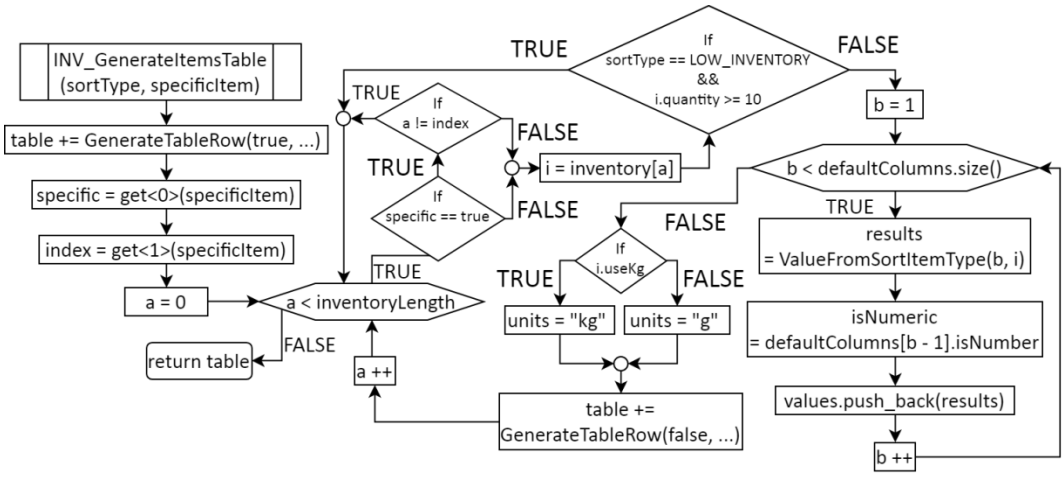
Prototype:

```
std::string GenerateItemsTable(SortType sortType, std::tuple<bool, int> specificItem);
```

Access: Private

Description: By default, it generates a table of the entire inventory by invoking *GenerateTableRow()* for every item. If the first value in the tuple *specificItem* is true, the table only displays a particular item. The index of the particular item is the second value of the tuple. The function also includes the relevant units of the values when required.

Flowchart:



---

12) **SelectItem()**

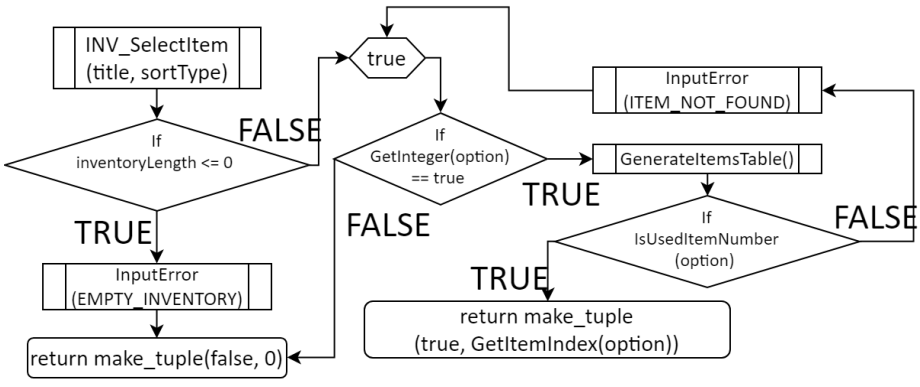
Prototype:

```
std::tuple<bool, int> SelectItem(std::vector<std::string> title, SortType sortType);
```

Access: Public

Description: Prompts the user to select a specific item in the inventory by specifying its index. The first value in the returned tuple indicates whether the user has chosen or not whereas the second value represents the index of the chosen item.

Flowchart:



---

13) **SelectItemType()**

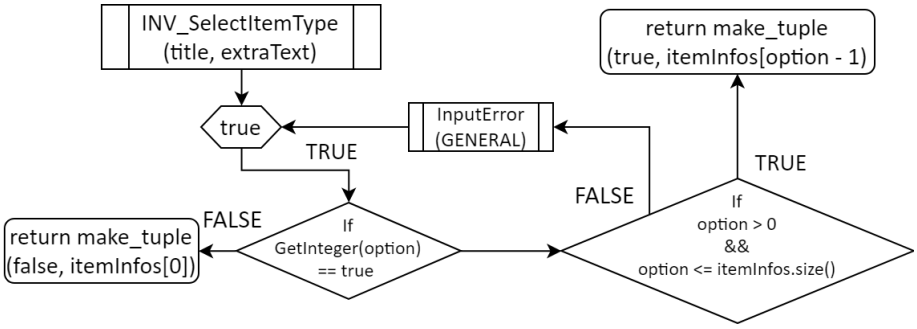
Prototype:

```
std::tuple<bool, ItemTypeInfo> SelectItemType(std::vector<std::string> title,
const std::string extraText);
```

Access: Private

Description: Prompts the user to select a particular item type (name, contact etc). The first value in the returned tuple indicates whether the user has chosen or not whereas the second value represents the *enum* value of the chosen item type.

Flowchart:



---

14) **SelectSortItemType()**

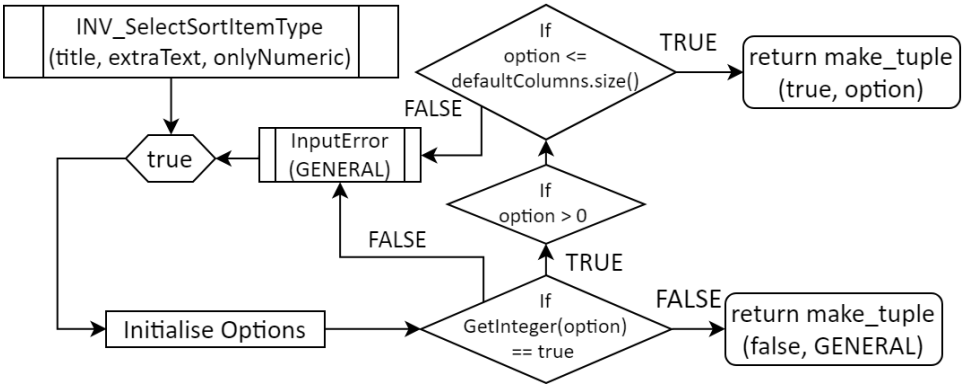
Prototype:

```
std::tuple<bool, SortItemType> SelectSortItemType(std::vector<std::string>
title, const std::string extraText, const bool onlyNumeric);
```

Access: Private

Description: Prompts the user to select the sorting item type. The first value in the returned tuple indicates whether the user has chosen or not whereas the second value represents the chosen sort item type.

Flowchart:



---

15) **SelectSortType()**

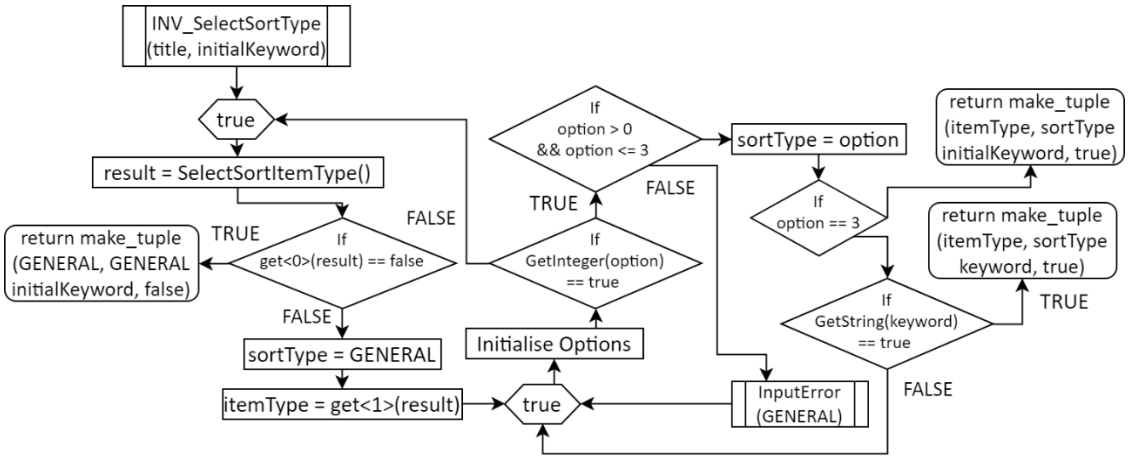
Prototype:

```
std::tuple<SortItemType, SortType, std::string, bool>
SelectSortType(std::vector<std::string> title, const std::string&
initialKeyword);
```

Access: Private

Description: Prompts the user to select the sorting mechanism (ASCENDING, DESCENDING & KEYWORD). It returns both the sorting item type and the chosen sorting type. If KEYWORD is selected, it prompts for a user-specified keyword which is included in the returned tuple. An extra boolean is also included in the tuple to indicate success or failure. The function first invokes SelectSortItemType() to obtain the sorting item type. If no item type is chosen, it returns a false boolean value and the initial keyword to indicate failure. Otherwise, it proceeds to prompt for the preferred sorting mechanism and a keyword if required. At default, the sorting type is GENERAL.

Flowchart:



---

16) **SortItems()**

Prototype:

```
void SortItems(const SortItemType& itemType, const SortType& sortType, const
std::string& keyword);
```

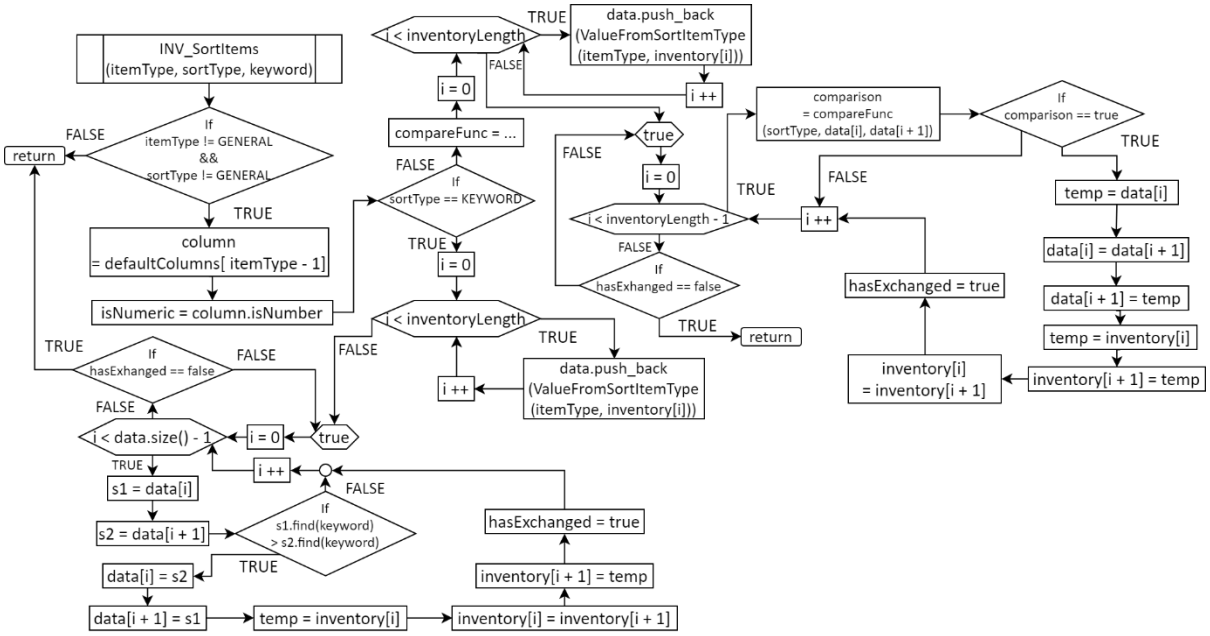
Access: Private

Description: Sorts the inventory according to the given sorting item type and the sorting type. A distinction is made between strings and numerical values (*double* and *int*) because all the numerical values like size and weight have to be converted to grams to streamline the comparisons carried out later.

Furthermore, lambda functions are used to generalise the comparisons such that they are equivalent for both numerical values and strings. Hence, the inclusion of duplicate code to cater for the two types of values is avoided. However, they have been excluded in the flowchart below to avoid overcomplicating it.

Another distinction is also made between the KEYWORD sorting type and the ACSCENDING/DESCENDING type because of their different approaches to making comparisons. Generally, bubble sorting is used for all three sorting types.

Flowchart:



---

17) **AddItem()**

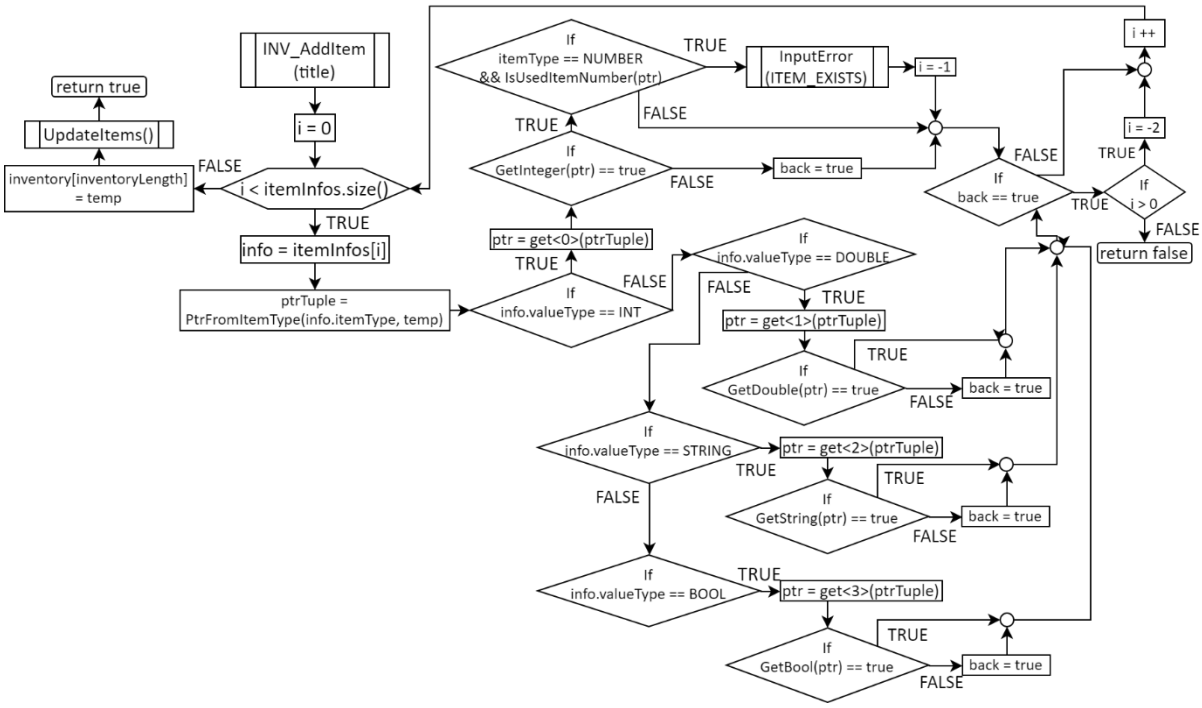
Prototype:

```
const bool AddItem(std::vector<std::string> title);
```

Access: Public

Description: Prompts the user to indicate all the values in a new item which will be added to the inventory.

Flowchart:



---

18) **ModifyItem()**

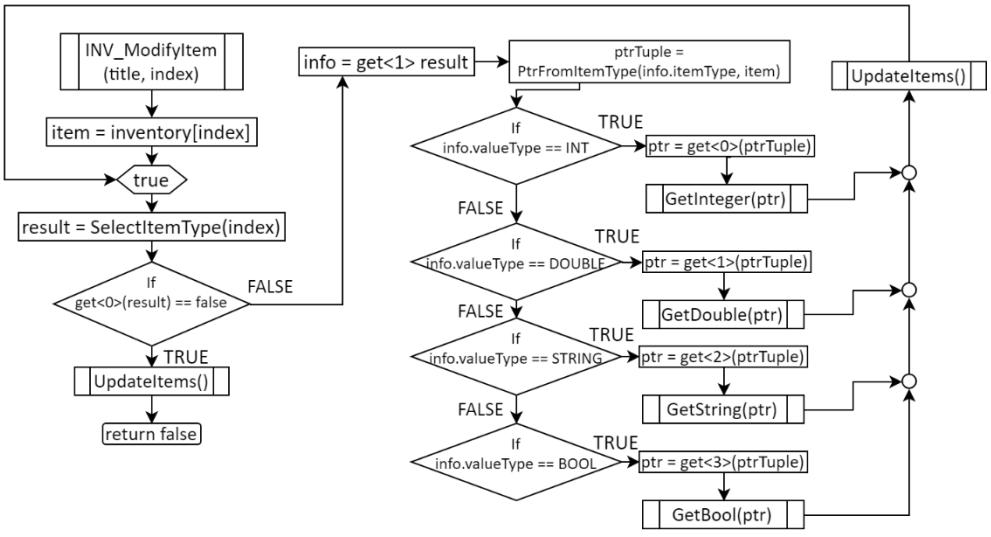
Prototype:

```
const bool ModifyItem(std::vector<std::string> title, const int index);
```

Access: Public

Description: Prompts the user to specify a new value for a particular field in an item of the inventory. The item is specified by the *index* parameter. *SelectItemType()* is invoked at the start to enable the user to select the field to be modified.

Flowchart:



19) **DeleteItem()**

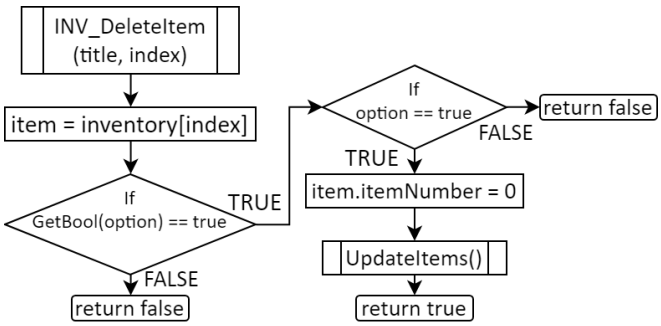
Prototype:

```
const bool DeleteItem(std::vector<std::string> title, const int index);
```

Access: Public

Description: Deletes an item specified by the user. The parameter *index* refers to the item.

Flowchart:



20) **DeleteInventoryFile()**

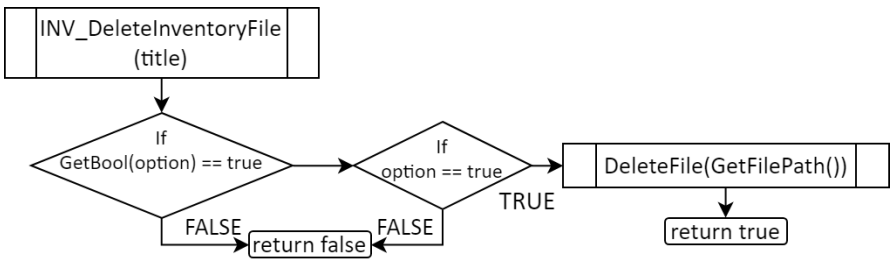
Prototype:

```
const bool DeleteInventoryFile(std::vector<std::string> title);
```

Access: Public

Description: Deletes the inventory profile by invoking *DeleteFile()*.

Flowchart:



21) **ViewItems()**

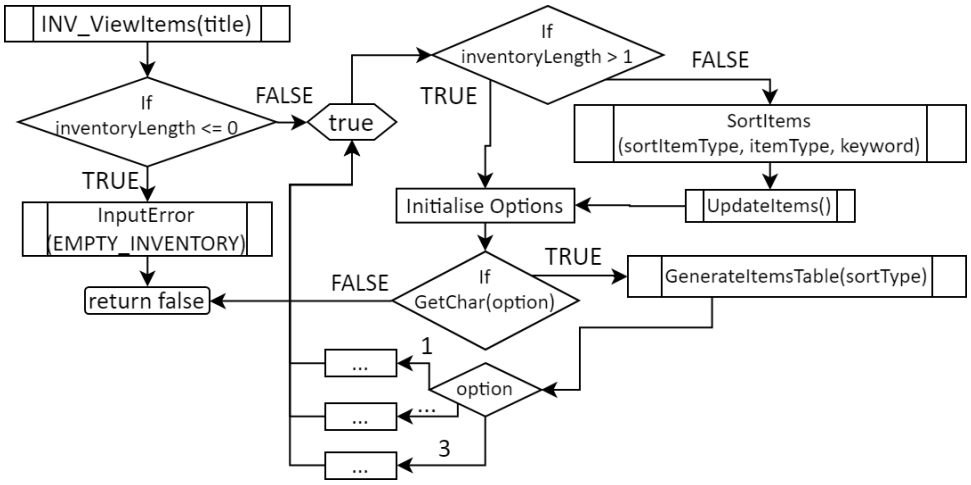
Prototype:

```
const bool ViewItems(std::vector<std::string> title);
```

Access: Public

Description: Invokes *GenerateItemsTable()* to display the inventory. It also prompts the user to choose from a variety of options like adding items and showing low inventories.

Flowchart:



---

22) **Distributions()**

Prototype:

```
const bool Distributions(std::vector<std::string> title);
```

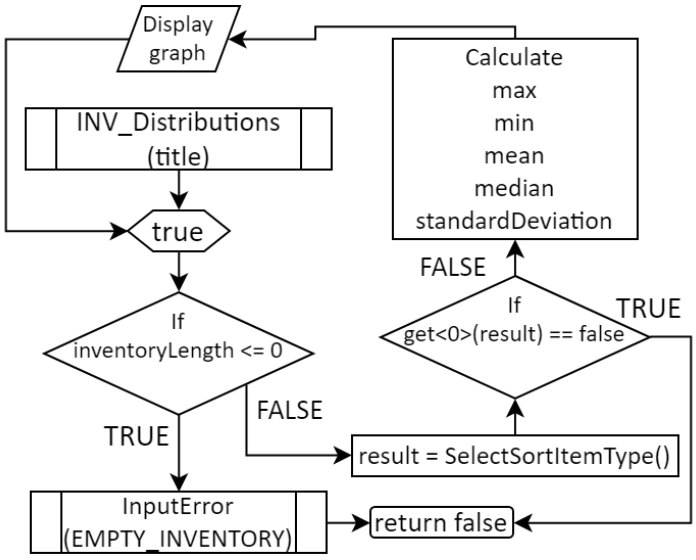
Access: Public

Description: Displays a normally distributed graph of a particular item type. SelectSortItemType() is invoked to prompt the user for the item type. The algorithm which "draws" the graph is not visualised in the flowchart as it is too complicated.

Essentially, the algorithm divides the vertical axis into a set of evenly spaced points on the vertical axis known as **divisions**. The number of divisions is specified in the program in advance, which is 5 at default. The fixed **spacing** between these points is obtained by dividing the maximum value by the number of divisions. In addition, the spacing itself is divided into more points known as **subdivisions**. The number of subdivisions is also 5 at default. The **sub-spacing** is obtained by dividing the spacing by the number of subdivisions.

The algorithm assumes that every line in the console represents a subdivision. It starts from a starting value, **V**, equal to the maximum value and decrements V by the calculated sub-spacing iteratively until it becomes zero. In every iteration, another iteration occurs whereby V is compared to every relevant value in the inventory. If V is marginally different within a maximum percentage error, a bar chart is plotted for the particular inventory value. The maximum number of bar charts and the gap between them is specified in advance. If the values exceed the maximum number of bars, only the largest few will be displayed and the rest are not displayed. This graphing algorithm is probably the most sophisticated one in this program but is also the most satisfying to code into reality.

Flowchart:



---

23) **Overview()**

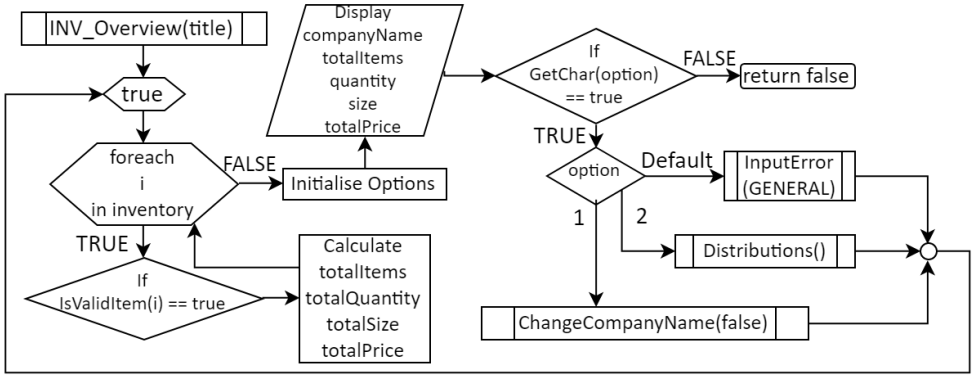
Prototype:

```
const bool Overview(std::vector<std::string> title);
```

Access: Public

Description: Displays an overview of the profile in the form of some statistical info. The user can invoke *Distributions()* and *ChangeCompanyName()* from here.

Flowchart:



Class Summary:

```
const vector<InventoryManager::ItemTypeInfo> InventoryManager::itemInfos{
{"Number", IT::NUMBER, VT::INT, false}, //Name, itemType, dataType, allowZero
{"Name", IT::NAME, VT::STRING, true},
{"Quantity", IT::QUANTITY, VT::INT, true},
{"Use Kilograms", IT::USE_KG, VT::BOOL, true},
{"Size", IT::SIZE, VT::DOUBLE, true},
{"Price", IT::PRICE, VT::DOUBLE, true},
{"Brand", IT::BRAND, VT::STRING, true},
{"Supplier", IT::SUPPLIER, VT::STRING, true},
{"Contact", IT::CONTACT, VT::STRING, true}
};
```

The vector *itemInfos* dictates the order in which the data is interpreted from and written to the user's file. Based on the figure above, the item number comes first, followed by the name and so on. If the order is changed, the user file becomes obsolete and incompatible with the modified program as its contents are formatted in a different order. Nevertheless, this approach expands the program's customisability and renders the addition and removal of item parameters easier.

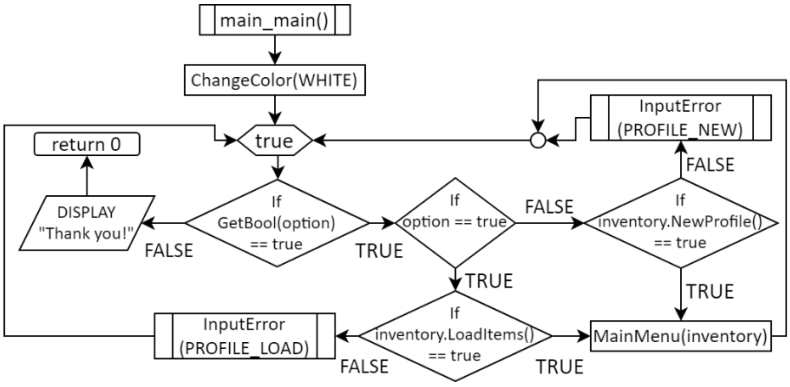
By obtaining pointers to specific variables of an inventory item using *PtrFromItemType()* and differentiating them with respect to fundamental data types using a switch control flow, redundant code is vastly reduced because we do not have to allocate a line of code for every single item parameter. As emphasised in the previous paragraph, the item parameters are modified or accessed in respect to the order specified in *itemInfos*. Although this approach aggravates the program's complexity and makes it abstruse to the uninitiated, its elegance far outweighs the gut-wrenching sight of overly redundant code.

3.6: Main

This segment covers the main function and its complementary functions, *MainMenu()* and *About()*.

Functions:

1) **main()**
   Prototype:
   `int main();`
   Description: It prompts the user whether to load or create a profile.
   Flowchart:



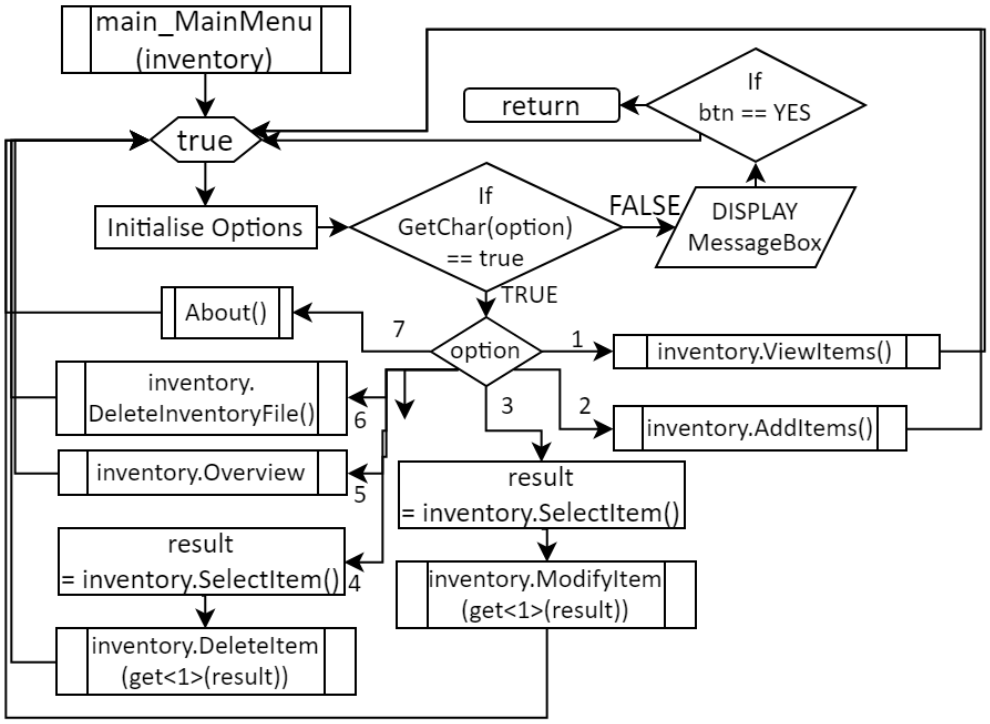
2) **MainMenu()**

Prototype:

```
void MainMenu(InventoryManager& inventory);
```

Description: Displays the main menu.

Flowchart:



3) **About()**

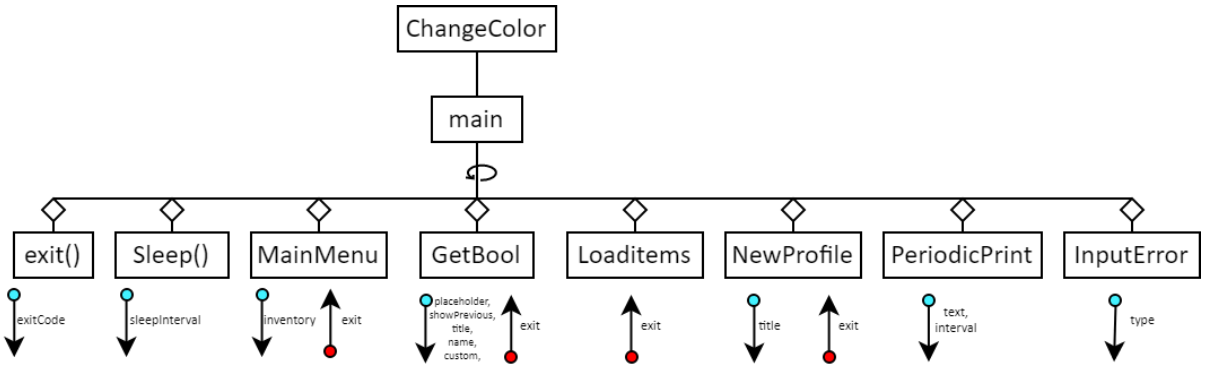
Prototype:

```
void About(vector<string> title);
```

Description: Displays general info regarding the program.

4: Conclusion

After having perused various areas of the program in detail, we can finally paint a coherent picture of it. The following is a structure chart summarises of the main thread:



It is quite likely that the structure chart above is not structured in the correct format. Nevertheless, it raises a question of practicality, especially when considering the growing complexity of structure charts the more functions there are. With around 55 functions in our program, we concede defeat when it comes to structure charts. Even drafting the report up to this stage has driven us to the point of insanity. We do not dare imagine where another 54 diagrams would lead us to. To the lecturer marking, go ahead and do what it is required. Our marks always seem to slip away like money anyway.

We admit that it is entirely our fault for unnecessarily complicating the program. Flowcharts and structure charts were the least of our concerns so we did not consider the consequences of the type of journey we chose to embark on. One question still lingers: should programs be designed to accommodate structure charts or the other way around? After all, structure charts seem to have no practical use.

Regardless, the regret is swept away by the insight that we have gained from our efforts. The countless bugs and inexplicable breakthroughs that we have experienced have inspired us to be better programmers. We would also like to thank the lecturers for all their guidance and their commitment towards a better education environment.