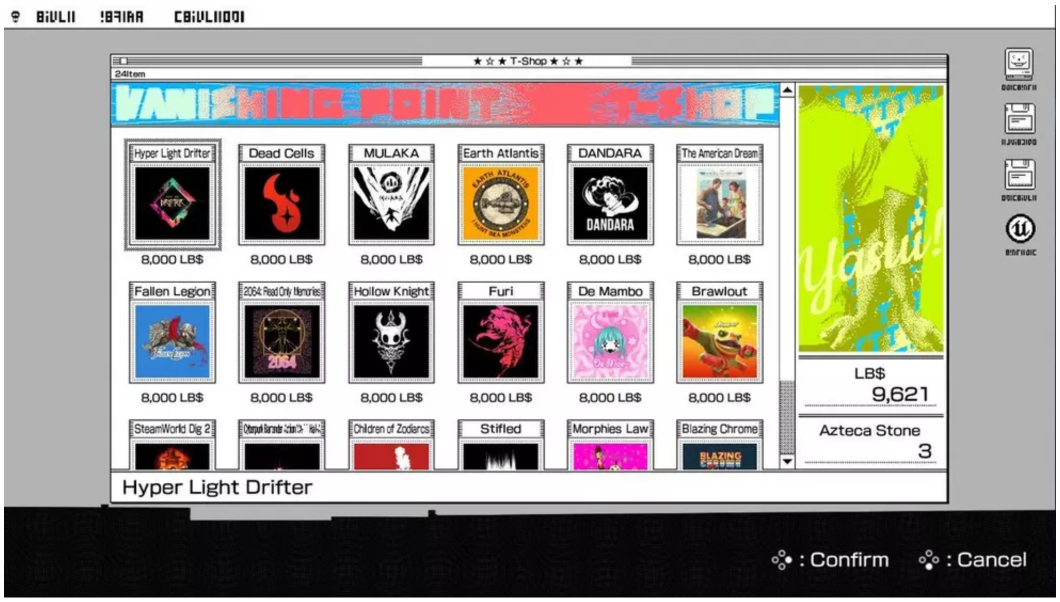
CS 202 Fall 2023 - Assignment 4

Pointers and Dynamic Memory Allocation



Overview

The modern age brings with it technological marvels not seen any time before in human history. Online marketplaces allow you to buy anything, anytime. Did you forget your grandma's birthday is tomorrow and need to buy her a last minute gift? Go for it. In this assignment, you will implement a simple digital storefront and allow a user to purchase and check out from a catalog of different items.

The storefront will be realized with three classes: one for Items offered, a ShoppingCart to store the items a user is planning to buy, and a Store to display the items and allow users to conveniently interact with what's for sale. You will be asked to implement some functions for these classes to get the store up and running.

Since what a user buys can change depending on their needs, the way data is handled in the program will need to be flexible. When objects or arrays need to be allocated on the fly - during the runtime of the program - we can use dynamic allocation. Dynamically allocated variables are created during the run of the program rather than at its start. They offer more control to the programmer in how data is actually stored.

Memory is dynamically allocated in C++ using the **new** keyword. Both objects and arrays are allocated this way. Objects are allocated by putting a **new** before the constructor for that object and arrays are allocated by putting a **new** before the type of thing the array will hold and how many. **new** always returns a pointer to where whatever was allocated was put, where a pointer is simply a data type used to hold addresses.

```cpp
Item* item = new Item("Vase", 4, 20.49); //Makes a new Item and have item point to it
Item* itemsArr = new Item[10]; //Makes an array of 10 Items and has itemsArr point to it
```

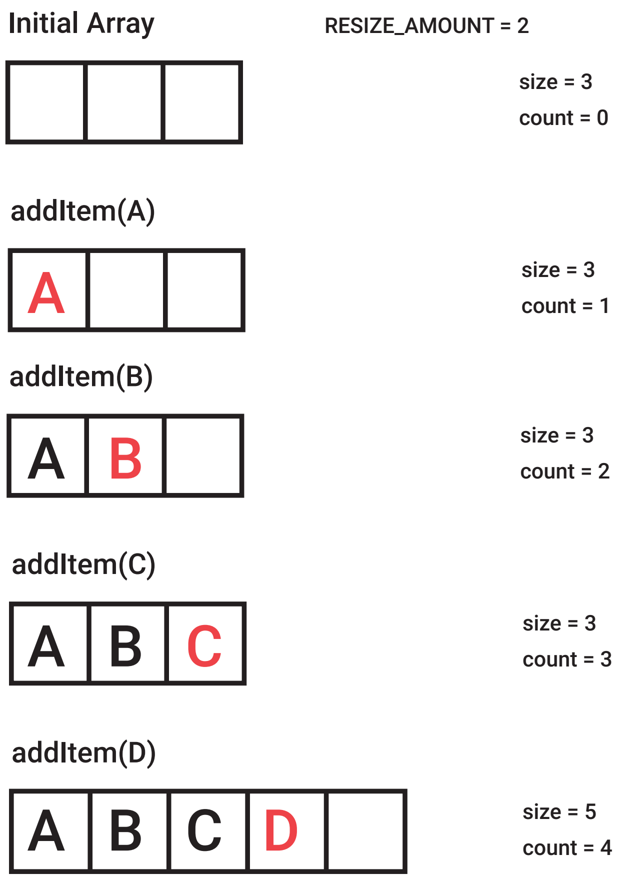
Part of this assignment will ask you to implement a dynamically resizing array. Let's try to understand how these work before it's implemented. First, consider a static array that you have used in the past:

```cpp
const int SIZE = 5;
int arr[SIZE];
```

Static arrays have fixed sizes which MUST be known before compiling (some compilers will not require the known before compiling part, but those are secretly not static). This means the size of the array is always going to be 5 in this example, no matter what happens in the program itself. If the array only needed to hold 3 ints, 2 spaces would go wasted. If the array needed more than 5 ints, then some of those ints would have to be thrown away. This creates a problem when the number of elements to be stored in an array cannot be known before running the program.

In the context of this assignment, think of an online shopping cart. The number of items that one customer buys is variable. One order might have 10 items, while another contains 2, and yet another 4. Because the order size cannot be known a priori, the array should be created during the execution of the program. Here, we will create an array with some small amount of open space. If less than that amount of space is needed, some elements of the array may be wasted, but will be minimal. If there is not enough space for the number of items needed, then we will simply make a bigger array.

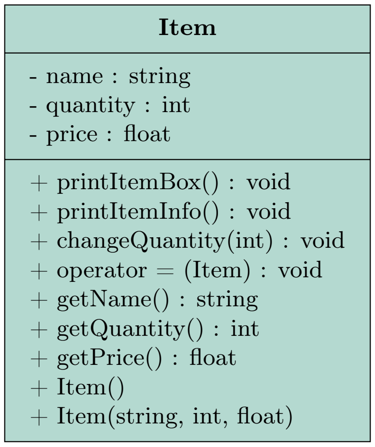
Below is an example of what an array might look like for a customer buying 4 items: A, B, C, and D. The array is initialized to hold 3 items maximum and grow by 2 elements each time it runs out of space. The count represents how many items are in the array, while the size represents how big the actual array is.

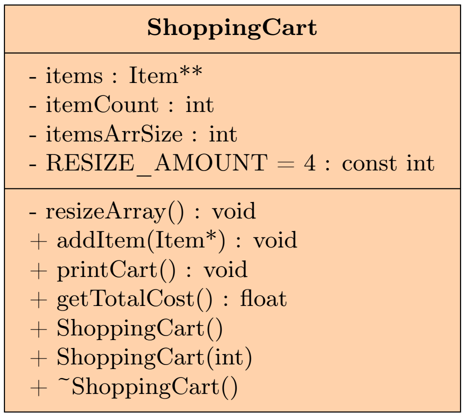


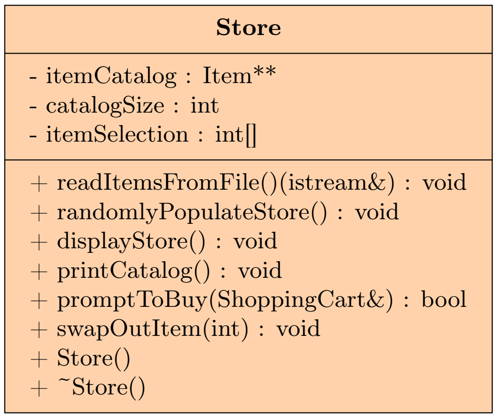
Note that if there is space (like there is on the addition of A, B, and C) the item can simply be added to the end of the array. However, when D is added and there is no space, a bigger array must be made. This can be done by dynamically allocating an array of size $3+2 = 5$, copying over the contents of the old array, and updating the size before adding D into the new array with enough space. The old array must be deallocated to avoid a memory leak.

UML Diagrams

Below is a list of UML diagrams containing useful class members for the three classes in this assignment. Note that some variables and functions are omitted that are used strictly by the skeleton and may not be relevant to your code or understanding of it for convenience.







The Assignment

The assignment itself is split into two parts. The first part is very short and is just to provide some practice with the different types of allocations. It will ask you to write a few (<20) lines of code to dynamically allocate an Item, an array of Items, and an array of dynamically allocated items. Remember that the **new** keyword always returns a pointer to what was allocated, regardless of whether that data is an array or an object. Please read through the comments in the skeleton for a more detailed overview.

The second part will involve the implementation of the storefront. You will need to finish some functions of the Store and ShoppingCart classes. Files to test your functions are also included and using them will be explored in the **Compiling and Testing** section later.

Important Classes and Functions

Below is a list of functions and variables that are important for this assignment. ***Variables are in green***, **functions that you will need to write are in red**, and **functions implemented for you already are in blue**.

Globals and Others

- ***ItemRef*** - A preprocessor directive definition for Item* (a reference to a Item). Basically, you can use this name in place of Item* if it easier for you. Since our ShoppingCart will contain an array of Item*s, you may have an easier time conceptualizing it as an array of ItemRefs. A reference here just means a pointer to some object that we will use, versus a pointer to a dynamically allocated array. An allocation of an Item.

```
ItemRef i = new Item("Cool Item", 3, 2.60);
```

  would return a pointer to an Item, i.e. an ItemRef.

- ***const int SELECTION_SIZE = 5*** - How many Items to display at a time on the Store's page. This is a constant amount and is not determined at run time

Item

A class representing an Item that can be purchased from the Store. This keeps track of the Item's name and price, along with the quantity (either how many the Store has on hand or how many the user has put in their cart). This class is implemented for you.

- ***string name*** - The name of the Item being sold
- ***float price*** - The cost of buying a single unit of the Item
- ***int quantity*** - Quantity of how many of this Item is available. This could be either how many are in the stock the Store is selling or how many the user has of the item in their ShoppingCart.
- **void printItemBox()** - Prints all of the Item's variables in a formatted way for displaying on the Store page. This is used by the skeleton code to display Items and is not needed in your code, but can be used for debugging.
- **void printItemInfo()** - Prints the Item's variables in a less formatted fashion than the previous function. This can be used whenever the Item's data needs to be displayed.
- **void changeQuantity(int change)** - Increases the quantity of the Item by the amount **change**.
- **void operator =(Item other)** - This exists to allow the assignment operator = to work with Items. You don't need to explicitly use this, it exists for convenience.

- **string getName()** - Getter for the ***name*** variable.
- **int getQuantity()** - Getter for the ***quantity*** variable.
- **int getPrice()** - Getter for the ***price*** variable.
- **Item()** - Default constructor. Creates an Item and does nothing else.
- **Item(string name, int quantity, float price)** - Constructor to set all three member variables of the class to the arguments of the corresponding name.

ShoppingCart

A class to hold the Items that a user is purchasing. Since a user can add an ever-growing number of items to their cart before buying, this will need to keep a dynamically allocated array containing the Items that will resize as more space is needed. This bulk of the assignment is writing functions for this class.

- ***Item** items*** - A 1D array of references to Items in the cart. Don't be thrown off by the two **, this is still a 1D array. The **Item*** comes from the contents of the array being Item pointers and the second star cmoes from this being a pointer to that array.
- ***int itemCount*** - How many Items are currently in the ***items*** array.
- ***int itemsArrSize*** - The actual size of the ***items*** array. This will tell how many elements the array has, which may or may not contain Items.
- ***const int RESIZE_ AMOUNT = 4*** - The amount to resize the ***items*** array by when it needs to be expanded. This determines how many elements larger a new array should be when the current array runs out of space.
- **ShoppingCart()** - Creates an empty shopping cart with no allocated ***items*** array. Leaves ***items*** pointing to nullptr.
- **ShoppingCart(int size)** - Creates an empty shopping cart and allocates the ***items*** array to the given size.
- **void addItem(Item* item)** - Adds the given item to the end of the ShoppingCart. This should put the item into the array pointed to by ***items*** similar to the diagram from earlier. If there is no space to fit the item into the array, obtain a bigger array using the **resizeArray()** function before adding the item to the end. Make sure to update the count.
- **void resizeArray()** - Increases the size of the array for the ShoppingCart. This should allocate a bigger array for ***items*** to point to. Start by allocating the bigger array with***RESIZE_ AMOUNT*** many more spaces than the previous array. Then, copy over the contents of the existing array to the new array. Finally, deallocate the old array and make ***items*** point to the newer, bigger array. You will need to update ***itemsArrSize*** to reflect the size of the ***items*** array so that other functions know how big it is.
- **void printCart()** - Prints all of the Items in the ShoppingCart using their **printItemInfo** function.
- **void getTotalCost()** - Gets the total cost of all Items in the ShoppingCart so that the user knows how much they will have to pay. Make sure to count all Items using both the price and the quantity. For example, if there are 4 loaves of breads each costing \$3.50, the price of that Item should be \$14.00. Find a running sum for all Items.
- **~ShoppingCart()** - The destructor for the ShoppingCart class. This should deallocate all of the items in the ShoppingCart, followed by the ***items*** array that held them. Remember that there are two different ways to use the **delete** keyword for objects and arrays! This function should be about three lines, don't overthink it.

Store

The Store which the user actually buys things from. This serves as a convenient method of interfacing with the Items available and purchasing them. This class will keep a catalog of all Items available, which will be read from a database file, a selection of limited items able to be purchased from that catalog, and provides functionality to interact with the ShoppingCart of a user. Since only a few Items will be displayed at a time, this class comes with some helper functions to easily pick out which Items to display and replace Items as they become sold out.

- ***Item** itemCatalog*** - A 1D array of Item*s. This represents all of the items available for the user to purchase.
- ***int catalogSize*** - How large the catalog of Items is.
- ***int itemSelection[]*** - Tells which Items to display to the user. Only ***SELECTION_SIZE*** many Items will be shown at a time so as not to overwhelm the user. Holds indices into the itemCatalog for looking up Item information. This array is static and will always be whatever size ***SELECTION_SIZE*** is set to.
- **void readItemsFromFile(istream& itemsFile)** - Reads the Items that will be available from an input file. This will create and populate the ***itemCatalog***. The first value in the file should be the number of Items in the catalog. Read the value in as the size of the catalog, and then allocate an array to hold all of the Items that will be in the catalog. Since it will consistently contain the amount of Items from the file, there is no need to resize later. After allocating the array, fill it with Items read from the file. Each line will contain the quantity, price, and name of each Item. Read each into appropriate variables and then create a new Item using a constructor and place it into the catalog. Do this until all items have been read. Please see the sample files in the input folder for reference. The function does not need to error check anything.
- **~Store()** - Destructor for the Store class that deallocates all of the items in the ***itemCatalog***, followed by the catalog itself. Should be very similar to the ShoppingCart destructor.
- **Store()** - Constructor for the store which sets the catalog to empty and initializes variables for later random population.
- **void randomlyPopulateStore()** - Randomly picks Items from the ***itemCatalog*** to sell to the user.
- **void displayStore()** - Displays all of the Items currently available for purchase from the ***itemSelection*** in a pretty, formatted fashion.
- **void printCatalog()** - Prints the items from the ***itemCatalog***. Exists mostly for debugging purposes.
- **bool promptToBuy(ShoppingCart& cart)** - Prompts the user with the choices available to buy and handles taking their order. Used by the main in the skeleton as an entry point for interfacing with the class. Returns a boolean on whether or not the user would like to buy more.
- **void swapOutItem(int index)** - Swaps out the item from the ***itemSelection*** with the next item in the random sequence if there is a valid one left. Replaces the item at the given index.

Compiling and Testing

This assignment provides a makefile for each part. If you're unfamiliar with a makefile, it can be basically be seen as a series of commands describing how to compile and do important tasks. Here, our makefile compiles the cpp files and generates several executables. To use the makefile, ensure that it is in the same directory as your source files and then type **make**. You should execute **make** from within the part1 and part2 directories. Note that **make** is a bash command, so you can only use it on Mac or Linux, not Windows. If at any point it seems that your files may be corrupted and the makefile is not working properly, please try **make clean** before trying to build again.

The makefile for part 1 will generate one program called **practice**. It takes no arguments and be ran with just **./practice**. The makefile for part 2 will generate three programs: one for testing the ShoppingCart class, one for testing reading from a file in the Store class, and one for the actual store front program. The files are called **shopping_test**, **reading_test**, and **shopping_prog**, respectively. The first two testing programs take no arguments. The **shopping_prog** program takes two command line args: the file to use for the catalog and an optional argument for the seed to use for randomly picking items. Below are some examples of usage:

```
./shopping_prog input/food_catalog.txt
./shopping_prog input/food_catalog.txt 14
./shopping_prog input/toy_catalog.txt 205
```

The main functions for each of these programs are sorted into the **main_files** directory in part2 to keep things organized. **PLEASE DO NOT REMOVE THEM FROM THIS DIRECTORY UNLESS YOU INTEND TO IGNORE THE MAKEFILE!** You do not need to add any code to them and the makefile expects them to be there, although you are welcome to look at the code.

This is also the first assignment in which we will be using dynamically allocated memory. There is a program called valgrind that will be used to check your program for memory leaks. You will see the result of it running on CodeGrade, but if you'd like to test your program locally, simply type **valgrind** before the name of the program to run. Here, either the shopping_test or shopping_prog would be appropriate. For example: **valgrind ./shopping_prog input/food_catalog.txt**. Valgrind can only tell you when a memory leak occurs and what memory has been leaked, but it will not be able to tell you where the leak occurred exactly in your code. If you have 0 blocks still allocated by the end of the program, then everything is good. Otherwise, try double-checking your destructors, it is possible that memory was not deallocated properly. Also note that if a program crashes, valgrind may report memory leaks since some destructors may not have been called before the program crashed. Recall that there are two versions of delete, as well. If they are used with the incorrect kind of data, you may experience a crash and/or a leak.

TO-DO

It is recommended to start with part 1 with allocation_practice.cpp to get a hang of dynamically allocating objects, arrays, and arrays of dynamically allocated objects. Once the output looks good there and you feel confident about using the **new** keyword, move on to part 2, the actual shopping program.

For part 2, you need only modify **shopping_cart.cpp** and **store.cpp**, although you can look at the header files for reference. Start with writing the addItem and resizeArray functions, test them with the shopping_test program. Then, move on to the other Shoppingcart functions. Afterwards, try writing the Store's readItemsFromFile function and test it using the reading_test program. Once you feel the tests look good, move onto trying them out via the big shopping_prog program.

Please submit the three modified files (allocation_practice.cpp, store.cpp, and shopping_cart.cpp) to CodeGrade. If you are checking part2, you can skip uploading allocation_practice.cpp and vice versa for part1, but please double-check that your final submission has all three.

Sample Runs

Due to the length of the sample output, it is include in the sample_runs.pdf. Please see for formatted output. It may be quicker and easier to run the main shopping_prog on your own.