1 Submission Instructions

Submit to Brightspace on or before the due date a compressed file (.tar or .zip) that includes

1. Header and source files for all classes instructed below.
2. A working Makefile that compiles and links all code into a single executable. The Makefile should be specific to this assignment - do not use a generic Makefile.
3. A README file with your name, student number, a list of all files and a brief description of their purpose, compilation and execution instructions, and any additional details you feel are relevant.

2 Learning Outcomes

In this assignment you will learn to

1. Use inheritance.
2. Use linked list type data structures.
3. Make a UML diagram of the application.

3 Overview

In this assignment you will be writing a (vastly simplified) inventory system for a store / warehouse (like Ikea or Costco) using C++.

This store will sell a number of `Product`s. Each product has a `StoreLocation`. These locations are in the main store area where shoppers shop. Each product may have zero or more overstock locations in the warehouse. These are stored on skids (Figure 1) of all the same `Product` wrapped in plastic. We will call these `WHLocation`s (warehouse locations). Because `WHLocation`s can only accept skids, they will be treated differently than `StoreLocation`s, and we will use inheritance to implement to two different styles of `Location`s.

To help us with our inventory system we will use a few different data structures. We will use the linked `List` that we have seen in class, but with some modifications. We will also implement a `Queue`, which is a linked list but with different rules for adding and removing. We will also use primitive arrays.

4 Classes Overview

This application will consist of 8 classes. The classes are listed below along with their respective categories.

1. `Location` (Entity object):
   (a) A virtual base class for `StoreLocation` and `WHLocation` classes.
2. `StoreLocation` (Entity object):
   (a) Concrete implementation for in-store product locations.
3. `WHLocation` (Entity object):
   (a) Concrete implementation for warehouse product locations.
4. `Product` (Entity object):

Figure 1: "Skid" by Lee Bennett is licensed under CC BY-NC-SA 2.0 and "Warehouse" by toolstop is licensed under CC BY 2.0

   (a) Contains information about the product, including the `StoreLocation` and `WHLocation`s it is stored in.

5. `List` (Container object):

   (a) A list of `Product`s that can grow arbitrarily large.

6. `Queue` (Container object):

   (a) A first-in-first-out (FIFO) data structure for storing `WHLocation`s. The FIFO nature of the data structure ensures that we use older stock first.

7. `Store` (Control object):

   (a) Provides an interface for interacting with the inventory system.

8. `Control` (Control object):

   (a) Controls the interaction between the inventory system (`Store`) and the user.

9. `View` (View object):

   (a) Collects user input and provides system output.

5 Instructions

All member variables are `private` unless otherwise noted. All member functions are `public` unless otherwise noted. Some return values are not explicitly given. You should use your best judgment (they will often be `void`, but not always). ALL CLASSES MUST HAVE A PRINT FUNCTION. This print function should display the metadata of the class using appropriate formatting.

Once you download and unzip the assignment files you will find some full classes (`Control` and `View`) and some classes with partial implementations (`Product` and `Store`). There is a `Makefile` provided (which you may adjust as needed), and the files `main.cc` and `defs.h`. It will not compile, and there will be errors that should clear up as you design and implement the rest of the classes.

5.1 The Location Class

Implement the `Location` base class. For the `virtual` functions `add` and `remove`, you should carefully consider what code goes in the base class and what code in the derived class, or whether these should be pure virtual.

1. Member variables:
   (a) `string id`: The label of this location.
   (b) `string product`: The name of the product stored at this location. Although this is slightly unrealistic, since normally we would uniquely identify a product by its sku code, in this program we will also use the product name as the unique identifier (this is due to the circular reference problem mentioned in Section 4).
   (c) `int quantity`: The quantity of product stored at this location.
   (d) A string constant `NONE = "Empty"`. This should be a class variable. When there is no product in the location, we will set the `product` variable to `NONE`.
2. Make getters for the member variables (except for `NONE`).
3. Make a two argument constructor that takes a character code (such as 'A' or 'B') and a number. This constructor should concatenate the character code with the number and store it in the `id` string. For instance, if I pass in 'A' and 23 as arguments, then `id = "A23"`.
4. Member functions. In addition to the functions listed here, you may add other functions at your discretion. Be sure to properly document them.
   (a) `bool isEmpty()`. Returns true if `quantity == 0` and false otherwise.
   (b) `bool isAvailable()`. Returns true if `product == NONE` and false otherwise.
   (c) `void print()`
5. Virtual member functions. These functions will behave differently depending on the subclass that calls them.
   (a) `int getCapacity()`: This will return the maximum number of products this location can hold.
   (b) A `bool add` function that takes a string (product name) and an integer (quantity) as arguments. The integer is an input/output variable which will be described in the subclass.
   (c) A `void remove` function that takes an integer (quantity) as an argument. The integer is an input/output variable which will be described in the subclass.

5.2 The StoreLocation Class

Implement the `StoreLocation` class.

1. Member variables - these are all *class* variables:
   (a) `const char code` - this should be set to the character 'A'.
   (b) `const int capacity` - this should be set to the `SLOC_CAPACITY` macro from the file `defs.h`.
   (c) `const int nextId` - this will produce the next id number that is passed into the `Location` constructor.
2. Make a no argument constructor that calls the `Location` constructor and passes in `code` and `nextId` as arguments. `nextId` should be incremented before each use so that every location gets a unique number.
3. Member functions:

(a) A `void setProduct` function that takes a string as an argument and sets the `product` member variable to that string.

(b) `int getFreeSpace()`. Return the difference between `capacity` and `quantity`.

(c) Any other (well-documented) member function that you need to complete the tasks assigned.

4. Inherited virtual member functions.

(a) `virtual int getCapacity`: return the `capacity`.

(b) `virtual bool add`: Attempt to add the `amount` of the `product` specified. Return false if the location has a different product, or if it cannot fit all the product.

(c) `virtual bool remove(int amount)`: Attempt to remove the `amount` specified. Return false if the amount is greater than the quantity. Do not change the `product` variable even if the `quantity` goes to 0.

5.3 The WHLocation Class

Implement the `WHLocation` class.

1. Member variables - these are all *class* variables.

(a) `const char code` - this should be set to the character 'B'.

(b) `const int capacity` - this should be set to the `WHLOC_CAPACITY` macro from the file `defs.h`.

(c) `const int nextId` - this will produce the next id number that is passed into the `Location` constructor.

2. Make a no argument constructor that calls the `Location` constructor and passes in `code` and `nextId` as arguments. `nextId` should be incremented before each use.

3. Inherited virtual member functions.

(a) `virtual int getCapacity`: return the `capacity`.

(b) `virtual bool add`: Since these are warehouse locations they only accept complete skids. So we can only add `product` if the location `isAvailable`, and if it will fit (i.e., the `capacity` is sufficient). In which case we set the `product` member variable to the product specified and the `quantity` to the quantity specified. If the location is not empty, even if we are attempting to add the same product, do not change any values and return false.

(c) `virtual bool remove(int amount)`: the same behaviour as the `StoreLocation` class except that if the `quantity` goes to 0 we set `product = NONE`. Once we finish a skid that warehouse location is ready for any type of product. This reflects that `StoreLocations` are reserved for a certain product but `WHLocations` will accept a skid of any products.

5.4 The Queue Class

This has a similar structure to the `List` class we saw in class. It would make sense to make `Queue` a `List` subclass. However, we have not learned about templates yet, and in this application `Queue` and `List` use different classes as data. So we must keep them separate classes, which means that you will be copying a lot of code from the `List` class to put into `Queue`.

1. Nested class - make a private nested class `Node`. You may use the `Node` class from the `List` class, however, change the `data` to type `WHLocation*`.

2. Member variables:

(a) `Node* head` - same function as the `head` variable in the `List` class.

   (b) `Node* tail` - similar to `head` except `tail` should always point to the last element in the `Queue`. This will make it easy to add elements to the back of the `Queue`.

3. Constructor - initialize both `head` and `tail` to `NULL`.
4. Destructor - Delete all `Node`s in the `Queue`. DO NOT destroy the data.
5. Member functions:
   (a) `isEmpty()` - return true if the `Queue` is empty.
   (b) `void peekFirst(WHLocation** loc)` - return the `WHLocation*` data from the first location if it exists, or assign `NULL` to `*loc` otherwise. DO NOT delete the `Node`.
   (c) `void popFirst(WHLocation** loc)` - return the `WHLocation*` data from the first location if it exists, or assign `NULL` to `*loc` otherwise. Delete the `Node` if it exists.
   (d) `addLast(WHLocation* loc)` - Add `loc` to the end of the `Queue`.

5.5 The Product Class

1. Member variables. These have been added for you. If you change the names be sure to change them everywhere:
   (a) `string name`: The product name. Each product name must be unique.
   (b) `StoreLocation*`: The shelf in the store where this product is available for purchase.
   (c) `Queue*`: A FIFO data structure of `WHLocation` to store the warehouse locations that contain the `Product`.
2. Make a constructor that takes a `const string& name` and `StoreLocation*` as arguments. Use them to initialize the proper member variables. Make a new `Queue` class.
3. Make a destructor. The `Product` class is only responsible for destroying the `Queue` class. You should NOT destroy any other data in this destructor.
4. Member functions:
   (a) Make getters for `string name` and `StoreLocation*`. Make a setter for `StoreLocation*`.
   (b) `void addWHLocation(WHLocation*)`: add a `WHLocation*` to the `Queue`.
5. The following Product member functions have been done for you. Consider the following as documentation.
   (a) `void getFromStoreLocation(int& amount)`: Subtract the `amount` from the `StoreLocation` member variable. If `amount > quantity` return the difference. If `amount <= quantity` subtract `amount` from `quantity` and return 0.
   (b) `void getFromWHLocations(int& amount)`: Subtract `amount` from the `WHLocations` in the `Queue` starting with the first, the proceeding to the second, etc. If a `WHLocation` becomes empty delete it from the `Queue` and move to the next. If all `WHLocations` become empty, return the `amount` you were unable to subtract.
   (c) `void fillOrder(int& amount)`: remove `amount` of product from the store, starting with the `StoreLocation` and then, if there is still more `amount` left, move to the `WHLocations`. If no stock is left, return the `amount` of the order left to fill.
   (d) `void stockStoreLocation()`: Sometimes when filling an order the `StoreLocation` becomes empty. Add product to the `StoreLocation` until it is full (or you run out of product) by removing the same amount of product from the `WHLocations`. If the `WHLocations` become empty return from the function.

5.6 The List Class

You will use the `List` example seen in class with a few modifications.

1. Change the `data` member in `Node` to be of type `Product*`. When adding a `Product`, add it in alphabetical order by `Product` name.
2. New member functions:
   (a) `bool isEmpty()`: returns true if the `List` is empty and false otherwise.
   (b) `void findProduct(const string& name, Product** prod)`: find the `Product` with `name` in the `List` and point the pointer referenced by `prod` to it. If there is no such product, point the pointer referenced by `prod` at `NULL`.

5.7 The Store Class

1. Member variables:
   (a) `string name`: the name of the store.
   (b) A *statically allocated* array of `StoreLocation` objects. Use the `SLOCS` macro from `defs.h` to initialize the size.
   (c) A *statically allocated* array of `WHLocation` objects. Use the `WHLOCS` macro from `defs.h` to initialize the size.
   (d) A `List*` of products.
2. Make a constructor that takes one argument `const string& name`. Initialize all necessary member variables.
3. Make a destructor. The `Store` class is responsible for deallocating or causing to be deallocated all dynamically allocated memory it has access to.
4. Member functions:
   (a) `void findAvailableSLoc(StoreLocation** sloc)`: This should find the first available `StoreLocation` from the array (using `isAvailable`) and assign it to `sloc`.
   (b) `void findAvailableWHLoc(WHLocation** wloc)`: This should find the first available `WHLocation` from the array (using `isAvailable`) and assign it to `wloc`.
   (c) `void findProduct(const string& name, Product** prod)`: return a `Product` with the given name or `NULL` if such a product is not found.
5. Member functions for printing:
   (a) `void printStoreStock()`: print all `StoreLocations` that have a product assigned to them (even if the quantity is 0).
   (b) `void printBackStock()`: print all the `WHLocations` that are not empty.
   (c) `void printProducts()`: print all the products in the product `List`, whether or not there is currently any stock.
6. The following Store member functions have been done for you. Consider the following as documentation.
   (a) `void receiveProduct(const string& product, int quantity)`: First search for the product. If there is no such product, make a new `Product` object and add it to the `List` of products. Then find available `WHLocations` to store all of the product (it may require many such locations). Then call `stockStoreLocation` to fill up the in-store location.
   (b) `void fillOrder(const string&, int& amount)`: Remove `amount` of the product from all locations starting with the `StoreLocation`. Return the `amount` of the order that you could not fill. For example, if `amount = 20` but there is only `15` of the product available in all locations, then return `5`.

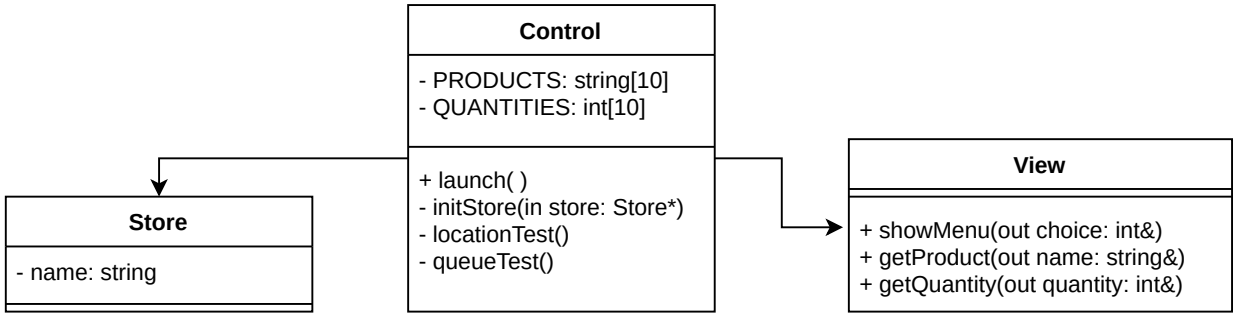

Figure 2: Partial UML diagram

5.8 The Control Class

This class has a `launch` function that instantiates and displays a `View` object to gather user input. Based on the user input, it calls the necessary `Store` functions, or groups of functions in order to test your application.

5.9 The View Class

Displays the menu and takes user input.

5.10 The `main` Function

Instantiates and `launch`s a `Control` object.

5.11 UML Diagram

Draw a UML class diagram of the finished application using any UML drawing program you like (though `draw.io` is excellent). You must represent inheritance and composition but do not need to represent "uses". Be sure to represent all member variables required and all member functions with the exceptions of getters, setters, and print. Be careful - just because a function "gets" something or has "get" in the name does not necessarily mean it is a getter. A "getter" returns a member variable as a return value. A partial diagram, Figure 2 has been provided.

6 Constraints

Your program must comply with all of the rules of correct software engineering that we have learned during the lectures, including but not restricted to:

1. The code must be written in C++98 and it must compile and execute in the default course VM provided. It must NOT require any additional libraries, packages, or software besides what is available in the standard VM.
2. Your program must not use any classes, containers, or algorithms from the standard template library (STL) *unless expressly permitted.*
3. Your program must be written in Object-Oriented C++. To wit:
   (a) Do not use any global functions or variables other than `main`.
   (b) Do not use `struct`s, use classes.
   (c) Do not pass *objects* by value. Pass by reference or by pointer.

(d) Except for simple getters or error signalling, data must be returned from functions using output parameters.

(e) Reuse existing functions wherever possible. If you have large sections of duplicate code, consider consolidating it.

(f) Basic error checking must be performed.

(g) All dynamically allocated memory must be deallocated. Every time you use the `new` keyword to allocate memory, you should know exactly when and where this memory gets `delete`d. Use `valgrind`.

4. All classes should be reasonably documented (remember the best documentation is expressive variable and function names, and clear purposes for each class).

7 Grading

7.1 Marking Components

1. 10 marks: UML class diagram.
2. 10 marks: Correct implementation of `Location`.
3. 14 marks: Correct implementation of `StoreLocation` and `WHLocation` classes.
4. 10 marks: Correct implementation of `Queue` class.
5. 6 marks: Correct implementation of `Product` class.
6. 4 marks: Correct modification of the `List` class.
7. 16 marks: Correct implementation of `Store` class.

Total marks: 70

7.2 Execution and Testing Requirements

1. All marking components must be called and execute successfully to earn marks.
2. All data handled must be printed to the screen to earn marks (make sure `print` prints useful information, such as the object member variables, where appropriate).

7.3 Deductions

7.3.1 Packaging errors:

1. 10 marks: Missing Makefile
2. 5 marks: Missing README
3. up to 10 marks: Failure to separate code into header and source files.
4. up tp 10 marks: Readability - bad style, missing documentation.

7.3.2 Major design and programming errors:

1. 50%: marking component that uses global variables or `structs`.
2. 50%: marking component that consistently fails to use correct design principles.
3. 50%: marking component that uses prohibited library classes or functions.
4. up to 10 marks: memory leaks reported by `valgrind`.

7.3.3 Execution errors:

1. 100% of any marking component that cannot be tested because it doesn't compile or execute in the course VM, or the feature is not used in the code, or data cannot be printed to the screen. In short: your program must convince, without modifcation, myself or the TA that it works and works properly. TAs are not required to debug or fix non-working code.