Q2 [35 marks] STL
Now that the composite MenuComponent class structure from question 1 has a forward iterator, we can use STL algorithms to perform some useful operations on Menu structures. For each of the following operations, you are to extend the provided IteratorTestHarness.cc file to perform the specified operation using STL algorithms of your choice. In particular, you are simply adding the hard-coded fragment that answers the question to the iterator test harness by having it included via preprocessor macros just after the main loop that builds the menu.

In each case, produce a new pair of main_[a-f].{h,cc} files. The main_[a-f].h file, which may be empty, is included in IteratorTestHarness.cc after the initial include statements and before the main routine. The main_[a-f].cc file is inserted into the main routine of IteratorTestHarness.cc, just after constructing a composite Menu, whose construction ends with the d command, and before the code that destroys it. The new code operates on the Menu/MenuItem stored in menus[0]. You may assume that for all test cases, menus[0] points to an instantiated MenuComponent, but it could be MenuItem or a Menu.¹ Note that all prices must be printed with a fixed two decimal positions. (Don't worry about rounding, just use the stream formatting operations.) All string searches are case-sensitive, and you may assume that there are no duplicates.

a) Print the vegetarian menu items in menus[0]. Start by printing a header string:

```
Veggie Items:
```

Use STL algorithm(s) to print all of the vegetarian menu items in menus[0]. Each menu item has the following format:

```
\t<name>, $<price>, cal=<calories> (V)
```

where <name> is the unquoted name of the menu item, <price> is the price of the menu item formatted to exactly two decimal places, and <calories> is the number of calories in a serving.

b) Print a submenu from menus[0]. Start with a prompt that asks for the quoted name of the menu element to be printed²:

```
Enter name of menu element to be printed:<sp>
```

Read in the quoted name of a menu. Use STL algorithm(s) to search the entire contents of menus[0] for the entry whose name matches the entered name. If an entry is found, use STL algorithm(s) to print the contents of the found entry. (Print using operator<< for (MenuComponent*) so that the submenu and its contents are appropriately indented.) If no entry matching the provided name is found, print an error message with the following format:

```
Menu <name> does not exist.
```

c) Remove an entry from menus[0]. Start with a prompt that asks for the quoted name of the menu from which an element is to be removed:

```
Enter name of parent menu element::<sp>
```

Print a second prompt that asks for the quoted name of the menu element to be removed:

```
Enter name of child menu element to be removed::<sp>
```

¹ Our sample executables verify that menus[0] isn't a nullptr, but you're not expected to do so.
² <sp> denotes a space/blank character.
Read in the quoted names. Use STL algorithm(s) to search the entire contents of `menus[0]` for an entry whose name matches the first name. If an entry is found, use the `MenuComponent::remove` method to remove the child element whose name matches the second name. Use STL algorithm(s) to print the entire contents of the resulting `menus[0]`. (Print using `operator<<` for `(MenuComponent*)` so that the submenu and its contents are appropriately indented.) If no entry in `menus[0]` matches the first name, print an error message with the following format:

```
\nMenu <name> does not exist.\n```

If an `InvalidOp` exception is thrown because `MenuComponent::remove` is invoked on a `MenuItem`, catch the exception and print the error message:

```
\nCommand is invalid on Menu Items. No change to menus.\n```

d) Increment prices by a user-specified amount. Start with a prompt that asks for the size of the price increment:

```
\nEnter the size of the price increment:<sp>
```

Read in the entered amount as a `float`. Use STL algorithm(s) to increment the price of every menu item in `menus[0]` by the entered amount. Use STL algorithm(s) to print the entire contents of the resulting `menus[0]`. (Print using `operator<<` for `(MenuComponent*)` so that the submenu and its contents are appropriately indented.) Ensure that the precision of printed prices is *exactly two decimal places.*

e) Place an order. Start with a welcome prompt and an initial "> " for an order:

```
\nMay I take your order?\n>\n```

Thereafter, read in a series of zero or more items. Each entered item has the following format:

```
\no "<name>"
```

So, if you want to order an item multiple times, you will have to have an "o" entry for each instance.

For each ordered item, use STL algorithm(s) and container(s) to record and update the current quantities of each `MenuItem` that has been ordered. After each entered order, print a prompt for the next order:

```
>\n```

The order is terminated by entering a `t` for total rather than an `o` for order. Use STL algorithm(s) and container(s) to print out the quantities of each ordered `MenuItem` (and only ordered menu items), along with a subtotal for each ordered `MenuItem`:

```
(<num>) <name>, $<price> = $<subtotal>
```

where `<num>` is the quantity of `MenuItem` named `<name>` that has been ordered, `<price>` is the unit price of the `MenuItem`, and `<subtotal>` is the cost of `<num>` units of the `MenuItem`. The items are printed in the order in which they were first ordered. So, if the client ordered tofu, parsley, and tofu, you would print 2 orders of tofu, then 1 order of parsley. (We're not worrying about calculating taxes.)

Use `accumulate` from the `numeric` library to sum up the subtotals and report the total cost of
the order:
TOTAL = $<cost> 

Define functions or functors where needed to specialize the STL algorithms that you use.

f) Redo part c) using lambda functions rather than functions or functors to specialize the STL algorithms.

The expectation is that you use your solution to Q1 in your answers to Q2. But, in case you did not complete Q1, or cannot get your code to work with the STL algorithms, we provide object files for the Menu classes and the Iterator base class that you can use instead.

Provided files
We provide files that implement the Composite objects (MenuComponent, Menu, MenuItem), as well as a Makefile. The implementations are complete with respect to creating composite objects of menus, submenus, and individual menu items. But these files will need to be amended in order to implement the class’s iterators.

We also provide a main program for your solution, IteratorTestHarness.cc. The main program is a test harness that you can use to test your implementation by creating various menus and using your iterators to print them. The test harness is not robust (it is throwaway code). If you enter invalid commands, it might cause the program to terminate. Your program will be tested only on valid input commands.

Execution
There are 6 valid commands that the test harness will recognize. The provided code implements all of the commands except the print command. The print command needs your iterator. If you comment out the print command, the program will compile and can be used to build composite menus.

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
</table>
| m <#> "<name>" | Creates a new menu at index <#> named <name>.
| i <#> "<name>" <price> <calories> <bool> | Creates a new menu item named name, stored at index #, with the specified price and calorie count, calories. The item is vegetarian if the boolean value is true. name is case-sensitive and you may assume it is unique.
| a <#1> <#2> | Add the menu (or submenu) #2 as a child of menu #1. Menu #2 is set to a nullptr.
| r <#> "<name>" | Remove the menu item or submenu name from menu #1. The search is not recursive, since that would require your iterator to be implemented. name is case-sensitive and you may assume it is unique.
| p <#> | Prints all elements and descendants of the menu stored at index #. For every level in the composite tree that the iterator descends, a tab character is added to the prefix for the item being printed. Note that the test harness already prints each item on a separate line. Each Menu/MenuItem has its name printed, followed by the information from its components, where applicable. A Menu

---

3 The name is surrounded by double-quotiation marks to allow whitespace in the name, though the test harness removes them from the string.
when printed, starts with a newline, the appropriate number of tabs, its name, a colon (":'"), and then its contents.

See the sample output and the executable for details on the format for the `MenuItem`.

<table>
<thead>
<tr>
<th>d</th>
<th>Terminates the menu creation phase in the program.⁴</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ctrl-d</td>
<td>Terminates the program.</td>
</tr>
</tbody>
</table>

Your implementation will be tested using our test harness, so do not make any changes to this code.

**Hints**

1. If you want to use your iterator in question 2 rather than the provided iterator, you will need to implement a non-polymorphic iterator that works when "iterating" over a `MenuItem` as well as when iterating over the elements of `Menu`. You will still need to implement a stack-based iterator, but it will need to work for all types of `MenuComponent` classes. Then your nested iterator classes simply inherit from this non-polymorphic iterator. (See the Factory Method design pattern code for an example of nested classes that inherit from another class.)

2. See Eckel volume 1, chapter 12 for advice on overloading operators, in particular `operator->`.
   Other useful resources are:

3. A good general rule is to think about what you want to do and figure out if an STL algorithm exists that is related to what you need to do.

---

⁴ In Q1, the program ends. In Q2, this allows you to test the extra code you are inserting.