Program #2 Design

Written by Alexander DuPree

Task and Purpose

Program #2's focus is in the design, implementation and manipulation of **Circular Linked List** (*CLL*), and **Linear Linked List of Arrays** data structures. These data structures are to be implemented as a **Queue** ADT and as **Stack** ADT. The **Stack** and **Queue** ADT's are tasked to simulate the process of comparing, evaluating and purchasing a computer based on a list of desired features. As such, we were tasked to perform the following:

- Enqueue all desired features of a computer onto the **Queue**
- Dequeue the features and compare them with a computer at a store.
- Push computers with matching features onto a **Stack** for later evaluation
- Display all viable computers from the **Stack**.

Design Considerations

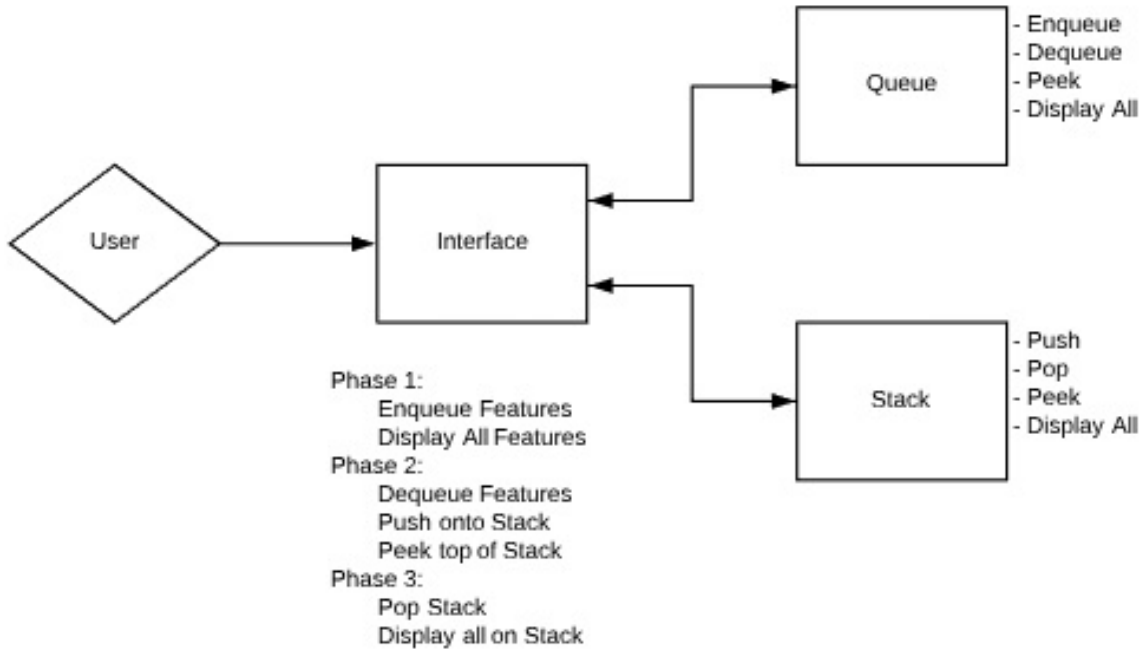
As before, in the last assignment, the user of the program must not be aware that stacks and queues are being used. Therefore, a high level of abstraction must be used to hide the *container's* "implementation". An interface class will be used to act as a middleman between communicating with the user and interacting with the **Queue** and **Stack** ADT's.

Data Flow

The nature of the assignment requires that the program be executed into three phases:

- First, the *Build* phase. The user must build his/her computer. Enqueueing desired features onto the **Queue**
- Second, the *Shopping* phase. The user will *"visit"* stores and be asked if a computer matches any of the features list. If a computer matches some, or all, of the desired features it can be added to the **Stack** for evaluation.
- Third, the *Evaluation* phase. Here the user will be presented with all the information related to the matching computers and make a selection.

After the user has made an evaluation the application will exit unless the user desires to restart the program. With these design considerations in mind, the data will flow similiar to this:



Data Structures and Abstract Data Types

The following is a summary of the data structures and ADT's directly related to this assignment. Other classes, such as the interface, will be used but has been summarized in other design documents and are utility classes not directly related to this assignment.

circular_linked_list

The *circular_linked_list* will be the underlying data structure for the *Queue* ADT. In this program the *CLL* will be specialized to hold c-strings that represent the desired features of the computer. As such the *CLL* public interface will consist of the following:

Function	Summary	Parameters	Return
push_back	Adds an element to the rear of the list	Read only reference to the copyable data	A read/write reference to the list, this allows chaining of methods.
pop_front	Removes the element at the front of the list	None	A read/write reference to the list
pop_front	Removes the element at the front of the list and copies it onto an out parameter	A reassignable object that will be overwritten with the front element	A reference to the out_parameter
clear	Recursively clears the list, deleting each node	None	A read/write reference to the list
size	Returns the number of elements in the list	None	The *CLL* keeps an internal counter of its size so this is a constant time operation

empty	Tests if the list is empty	None	returns true if the list is empty
begin	Returns an iterator to the front of the list	None	If the list is empty the iterator will be NULL, this can be checked with the iterators null() method
end	Returns an iterator to one past the rear of the list	None	Because the list is circular, the end iterator is equivalent to the begin iterator. To properly iterate through the list, a do-while loop must be used
front	Peeks the front element in the list	None	A read only reference to the front of the list
back	Peeks the rear element in the list	None	A read only reference to the back of the list
Equality operators	The == and != operators have been overloaded, they first compare sizes then compare each element in the list	None	True if the lists are the equal

Feature_Queue

The *Feature_Queue* utilizes a *circular_linked_list* to manage a *Queue* of "features" for a desired computer. These features will be stored as c-strings. The *Feature_Queue* will contain the following public interface:

Function	Summary	Parameters	Return
enqueue	Uses the *CLL* push_back method to add a feature to the queue	A Read only reference to the copyable data	True if the operation was successful
dequeue	Uses the *CLL* pop_front method to remove a feature from the queue	An allocated cstring to be overwritten with the popped feature	True if the operation was successful
peek_front	Uses the *CLL* front method to peek the front of the queue	None	A read only reference to the front of the queue
peek_back	Uses the *CLL* back method to peek the back of the queue	None	A read only reference to the back of the queue
display_all	Uses the *CLL* iterator methods to loop through the queue displaying each feature	None	None

linear_linked_list

The *linear_linked_list* will be the underlying data structure for the *Stack* ADT. In this program the *LLL* will be specialized to hold a dynamic array of *Product* ADT's. The *LLL* will contain the following public interface:

Function	Summary	Parameters	Return
push_front	Adds an element to the front of the list	Read only reference to the copyable data	A read/write reference to the list, this allows chaining of methods.
pop_front	Removes the element at the front of the list	None	A read/write reference to the list
pop_front	Removes the element at the front of the list and copies it onto an out parameter	A reassignable object that will be overwritten with the front elemment	A reference to the out_parameter
clear	Recursively clears the list, delete each node	None	A read/write reference to the list
size	Returns the number of elements in the list	None	The *LLL* keeps an internal counter of its size so this is a constant time operation
empty	Tests if the list is empty	None	returns true if the list is empty
begin	Returns an iterator to the front of the list	None	If the list is empty the iterator will be NULL, this can be checked with the iterators null() method
end	Returns an iterator to one past the end of the list	None	Dereferencing end iterators causes undefined behavior
Equality operators	The == and != operators have been overloaded, they first compare sizes then each element in the list	None	True if the lists are equal

Product_Stack

The *Product_Stack* utilizes a *linear_linked_list* of arrays to manage a *Stack* of *Product* ADT's that represent a computer with some, or all, matching features. Each node in the *LLL* will contain a dynamic array of 5 *products*.
When the *Product* array is full, another node will be created with another 5 *Product* array. This combines the fast random-access utility of an array with the dynamic growth/shrinkage of a *LLL*. The *Product_Stack* public interface will consist of the following:

Function	Summary	Parameters	Return
push	Uses the *LLL* push_front method to add a *Product* to the top of the stack	A read only reference to the *Product* to be pushed onto the *Stack*	True of the operation was successful
pop	Uses the *LLL* pop_front method to pop the top of the *Stack*	A *Product* object that will be overwritten with the popped *Product* data	True if the operation was successful
	Dereferences the *LLL* begin		A read only reference

peek	iterator to peek the top of the *Stack*	None	to the top element in the *Stack*

Product

The *Product* ADT is a utility class that stores information related to a computer for later evaluation. Each *Product* will contain the following data:

1. Store's name
2. Computer model name
3. A count of matching features

This information will be immutable, and only accessible in a read only format via the '<<' operator for printing to the console.