Program #1 Design

Written by Alexander DuPree

Task and Purpose

Program #1's focus is in the design, implementation and manipulation of a **linear linked list** (*LLL*) data structure to store and manage an **abstract data type** (*ADT*). As such, we were tasked to perform the following:

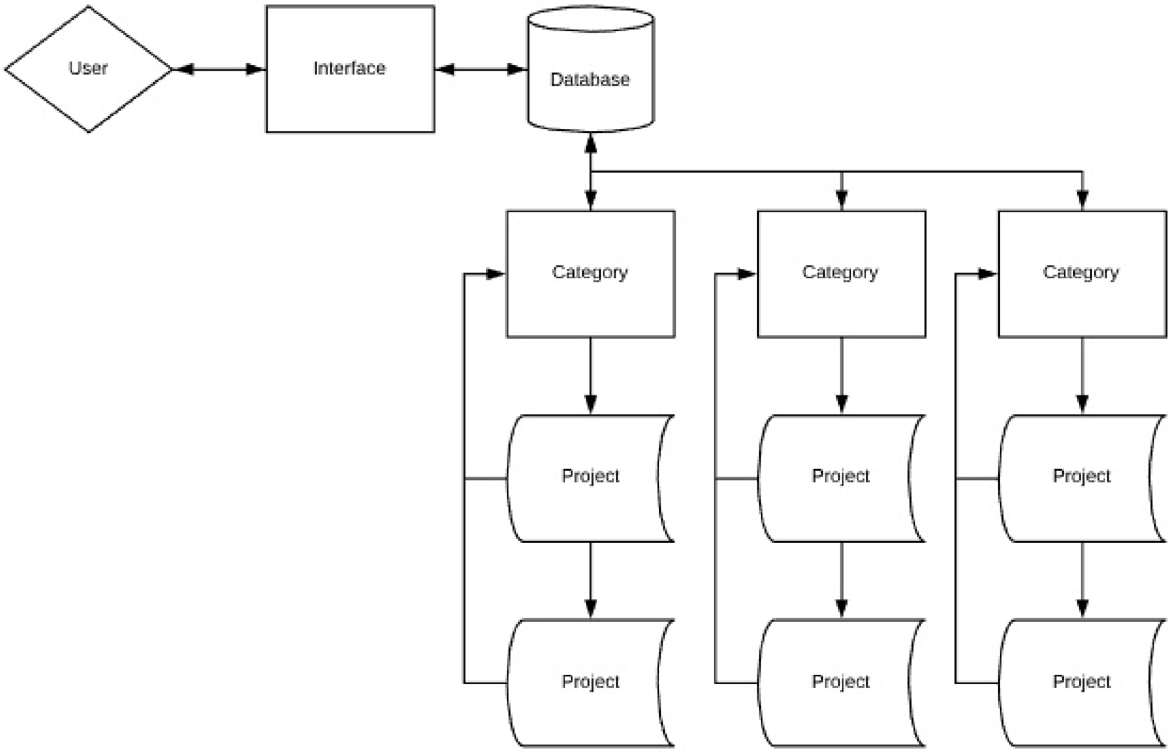
- Create a *LLL* that will store a unique set of *categories* (e.g., assignments, readings, labs)
- Each *category* contains a *LLL* that stores a unique set of *projects*.
    - A *project* is an item of work belonging to a specific category (e.g., reading #1 belongs to readings *category*)
- The client should have the following functionality:
    1. Add a new *category* of work.
    2. Remove a *category* of work.
    3. Add a *project* within a *category*
    4. Remove a *project* from a *category*
    5. Display all *categories*
    6. Display all *projects* in a *cotegory*

Design Considerations

One of the primary goals of this project is to create an *ADT* that "hides" the implementation of our *LLL* data structure. It is also important to design the *LLL* and *ADT's* to be "plugs" and "sockets". Meaning that even though the *ADT* is dependent on a *container*, the exact implementation of that container should be an abstract detail. This will allow future maintainers of this program to remove the *container* from it's "socket" and "plug" it with a better performing data structure.

Data Flow

With this design consideration in mind, the data will flow simliar to this:



- The user interacts with the interface generating *requests* to add/remove/display a *category* of work or to add/remove/display a *project* within a *category*.
- The interface validates, cleans, and packages the user's input into a format the *database* will accept.
- The *database* will execute the *request* by performing one of the following actions:

    1. Create a *category* of work.
    2. Remove a *category* of work.
    3. Request *project* information from a *category*.

- After the request has been executed the *database* returns either the requested information or a confirmation receipt.
- The interace presents the requested data and then listens for input.

Data Structures and Abstract Data Types

The scope of this assignment required the liberal use of the **linear linked list** data structure to perform insertion, deletion, and find operations. The *Database* and *Category ADT's* will act as wrapper classes for the *LLL* data structure; utilizing the *LLL* to perform operations within the it's scope of responsibility.

The following is a summary of each *ADT* and Data structure present in this program:

linear_linked_list

The *linear_linked_list* class is the primary data structure for this program. It contains methods to add, remove, and inspect data. The most imporant feature of the *linear_linked_list* structure is that it is fully-templated. This feature is to accommodate the need to create a *LLL* that holds *category* objects, and have each *category* hold a *LLL* of *projects*. Because of this requisite there seemed to be two options:

1. Write the *LLL* container to support a *project ADT* and then duplicate that code; replacing any reference to a *project ADT* with a *category ADT*.

or

2. Write the code once as a template and be done with it.

Alternatively, an abstract base class could have been created for the *category* and *project ADT's* and then they could have been passed around polymorphically; however, I felt the templated approach to be the most suitable.

The following is a summary of the *linear_linked_list* public interface:

Function	Summary	Parameters	Return
push_front	Adds an element to the front of the list	Read-only reference to the data to be copied	Void, updates the tail and head pointers internally
push_back	Adds an element to the back of the list	Read-only reference to the data to be copied.	Void, if the list is empty function calls push_front
add_unique	Adds an element to the list if there is no duplicate of it	Read-only reference to the data to be copied.	True if the data contains no duplicates and it was added successfully
begin	Returns a **read only** iterator to the beginning of the list	None	If the list is empty, the begin iterator will be equal to the end iterator
end	Returns an iterator to the element "one-past" the end of the list	None	Dereferencing an end iterator causes undefined behavior
size	Returns the number of elements in the list	None	The sorted_list contains an internal counter so that calling size() is a constant time operation
empty	Tests if the list is empty	None	Returns true if the list is empty
remove_if	Deletes the first element fulfilling the functional predicate	A pointer to a predicate fulfilling function	Returns true if an element is deleted
	Delete each node		Recursively travels the list, deleting

clear	in the list	None	each node as the stack unwinds
sort	Uses insertion sort to order the list	Pointer to a boolean returning comparator function	Void
Relational Comparisons	Only the == and != operators are overloaded	A read-only reference to a list containing the same data type	Returns false if the sizes are different, then compares each element of the both lists returning false on any inequality.

const_forward_iterator

The *const_forward_iterator* is a nested helper class used within the *linear_linked_list* class. It was important to create an iterator class so that the client can access his data directly without the need of manipulating a node pointer. Using an iterator to abstract a pointer helps keep the node members of the *linear_linked_list* private. It is important to note that this iterator type can only traverse the *linear_linked_list* in ONE direction only.

The following is a summary of the *const_forward_iterator* public interface:

Function	Summary	Paramters	Return
Operator *	Dereferences the iterator	None	Returns a read only reference to the Node's data member
Operator ->	Dereferences the iterator	None	Returns a read only pointer to the Node's data member
Operator++	Increments the iterator to the next element in the lsit	None	Points the iterator to the next node in the list
Comparison Operators	Only the == and != operators are overloaded	const_forward_iterator reference	Returns true if both iterators are pointing to the same block in memory

Project

The *Project ADT* stores specific data related to it's *category*. Each *Project* structure contains the following data:

1. Name
2. Due Date
3. Due Time
4. Late Date
5. Data Structure

- Each *project* will contain the requisite field data in the form of a custom character container type called SString (short string).

The following is a summary of the *project* public interface:

Function	Summary	Parameters	Return
name	Returns the name identifier	None	Returns a read-only reference to a custom string class object (SString)
Relational Comparisons	The equality and comparison operators are both overloaded	A read-only reference to another *Project* type	Uses the *Project's* name field as the basis of comparison
operator <<	Formats the project data onto an outstream object	A reference to an outstream object	Returns the formatted outstream object to the caller

Category

The *category ADT* acts as a wrapper class for the *LLL* data structure. A *category* object holds a *LLL* of *projects* related to it and handles the passing of data to and from it's *project* members.

The *Category ADT* will have the following public interface:

Function	Summary	Parameters	Return
name	Returns the name identifier	None	Returns a read-only reference to the name of the category
add_project	Adds a new *project* to its collection	An array of strings that are mapped onto the *project* fields	True if the *project* is unique and added correctly
remove_project	Removes a *project* from its collection	a string containing the name of the *project* to be removed	True if the project was successfully removed
display_projects	Displays all projects to the console	None	Void

Database

The *database ADT* acts as another wrapper class for the *LLL* data structure. The *database* is the only *ADT* that publicly interacts with the user interface. Its primary responsibility is to handle user requests by forwarding them to *categories* or executing the request at the *database* level.

The *database* will have the following public methods:

Function	Summary	Parameters	Return
add_category	Adds a new *category* to its collection	A string containing the name for the *category*	True if the *category* is unique and added successfully
remove_category	Removes a *category* from its	A string containg the name of the *category* to be	True if the *category* was successfully

	collection	removed	removed
display_categories	Displays all *categories* to the console	None	Void
add_project	Forwards a request to add_project to the correct *category*	The name of the *category* and an array of strings to be mapped onto the *project* fields	True if the *category* exists and the *category* reported back success
remove_project	Forwards the remove request to the correct *category*	The name of the *category* and the name of the *project* to be removed	True if the *category* exists and the *category* reported back success

Utility Classes

To assist in the production of this assignment, and for future assignments, I felt the need to develop utlity *ADT*'s to help streamline the code.

SString

Each *ADT* in this assignment required some sort of a string to store relevant data. Instead of making the same functions to allocate, deallocate, copy, and compare C-style strings, I thought it best to create my own C-string wrapper class. The *SString* (Short String) acts as an immutable char buffer. On construction it allocates, by default, 50 characters and copies the passed in string onto its buffer. *SString* has all equality and comparison operators overloaded along with the stream buffer operator and copy-assignment operator.

Interface

The *Interface ADT* holds a collection of *menu_item* abstract base classes to map input to a corresponding action. The *Interface* has methods to build and display a menu and poll for user input.

menu_item

The *menu_item* is a pure abstract base class to be used in conjunction with the *interface ADT*. The *menu_item* mandates that all concrete implementations contain an option() method as well as an action() method. This allows the client to inherit from *menu_item* and populate the option method to print whatever menu option is required and map it to specific sequence of function calls inside the action() method.