Program Design Document

To implement the calculator, I used two classes: a calculator class (Calculator) and a node class (Variable). The calculator class handles all the functionality. The Node class is used to allow for dynamic memory allocation for variables stored in the calculator. In order to run the calculator, create an instance, and run the start() method.

Main Program:

Purpose:

To allow the user to run the calculator, and any other code.

Implementation:

The only code included is the instantiation of a calculator, a call to run it, and a delete. This allows for simple and clean code that can be easily added onto. The calculator itself is dynamically allocated and later deleted to minimize memory usage in the case the main file is to be used for other purposes.

Calculator Class:

Purpose:

This class is the calculator and contains all functionality and data to keep it contained in one spot to allow for neat organization and easy implementation when using the calculator in the rest of you code.

Implementation:

In order to simplify interaction with the calculator, only one was public method that was made, start(), in order to allow for the highest level of abstraction when running the calculator, as it is not necessary for the user to know the details of the code, how it is implemented, or what the internal data is; just like in a real calculator.

All the functionality (other than the loop in start() to determine what function to run) is handled in private methods, as these functions would only be used in the context of a running calculator. As the data is only used in the context of a running calculator through various functions, all the variables were made private (encapsulated), as there is no situation that would require external memory access (just like in a real calculator).

The data in the class includes a pointer to a linked list to store all the data used in the calculator. This allows user to dynamically allocate and delete the variables they need don't need to use. There are also two additional variables (int) to make sure the maximum number of variables allowed to be stored is not exceeded.

When the calculator is no longer needed and is destroyed, the destructor will delete all the dynamically allocated variables to prevent memory leaks.

Variable Class:

Purpose:

Node class to store the name and value of a variable in the calculator, as well as a pointer to the next element.

Implementation:

All the variables are encapsulated in order to insure that data is only accessed when explicitly needed

Run Times:

CRT: O(1)

Simply sets the value of one and only variable, therefore always constant

DEF: O(n)

Due to the fact that the same variable cannot be defined twice, we must traverse all variables to make sure the new on being added is not a duplicate. This requires checking all n elements, therefore c*n = O(n) runtime. This was implemented in Calculator::append_variable using a while loop, which goes through each element from the Head of the list to the tail (nullptr). The addition of a new node requires constant time. O(n) + O(1) = O(n).

ADD: O(n)

To find the values to add an update, we must first find all three variables. If one of the variables used is last on the linked list, we must traverse each element, giving a worst case of c*n = O(n) runtime. This was implemented in Calculator::add, where a while loop was used to go through each element from the head of the list to the tail (nullptr). The addition operation is only ran once giving constant time. O(n) + O(1) = O(n).

SUB: O(n)

Same case as addition, except instead of the constant operation of addition, do subtraction. The method used is: Calculator::subtract

REM: O(n)

In the worst case of the desired element being at the end of the list, we must traverse the whole list with a while loop giving O(n) runtime. The removal process requires constant time. O(n) + O(1) = O(n). The method used is: Calculator::delete_variable

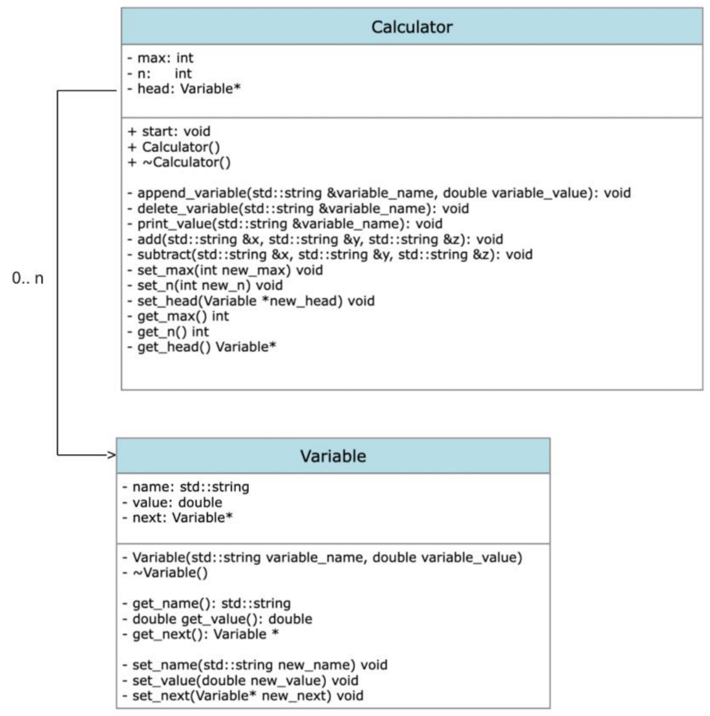
PRT: O(n)

In the worst case of the desired element being at the end of the list, we must traverse the whole list with a while loop giving O(n) runtime. The print process requires constant time. O(n) + O(1) = O(n). The method used is: Calculator::print_value

Calculator Destructor: O(n)

In order to prevent memory leaks, all the dynamically allocated memory must be deleted. To do this, we must loop through all n elements and delete them giving O(n) time. The deletion process takes constant time. O(n)*O(1) = O(n).

UML Design:


Calculator
- max: int
- n: int
- head: Variable*
+ start: void
+ Calculator()
+ ~Calculator()
- append_variable(std::string &variable_name, double variable_value): void
- delete_variable(std::string &variable_name): void
- print_value(std::string &variable_name): void
- add(std::string &x, std::string &y, std::string &z): void
- subtract(std::string &x, std::string &y, std::string &z): void
- set_max(int new_max) void
- set_n(int new_n) void
- set_head(Variable *new_head) void
- get_max() int
- get_n() int
- get_head() Variable*
0.. n
Variable
- name: std::string
- value: double
- next: Variable*
- Variable(std::string variable_name, double variable_value)
- ~Variable()
- get_name(): std::string
- double get_value(): double
- get_next(): Variable *
- set_name(std::string new_name) void
- set_value(double new_value) void
- set_next(Variable* new_next) void