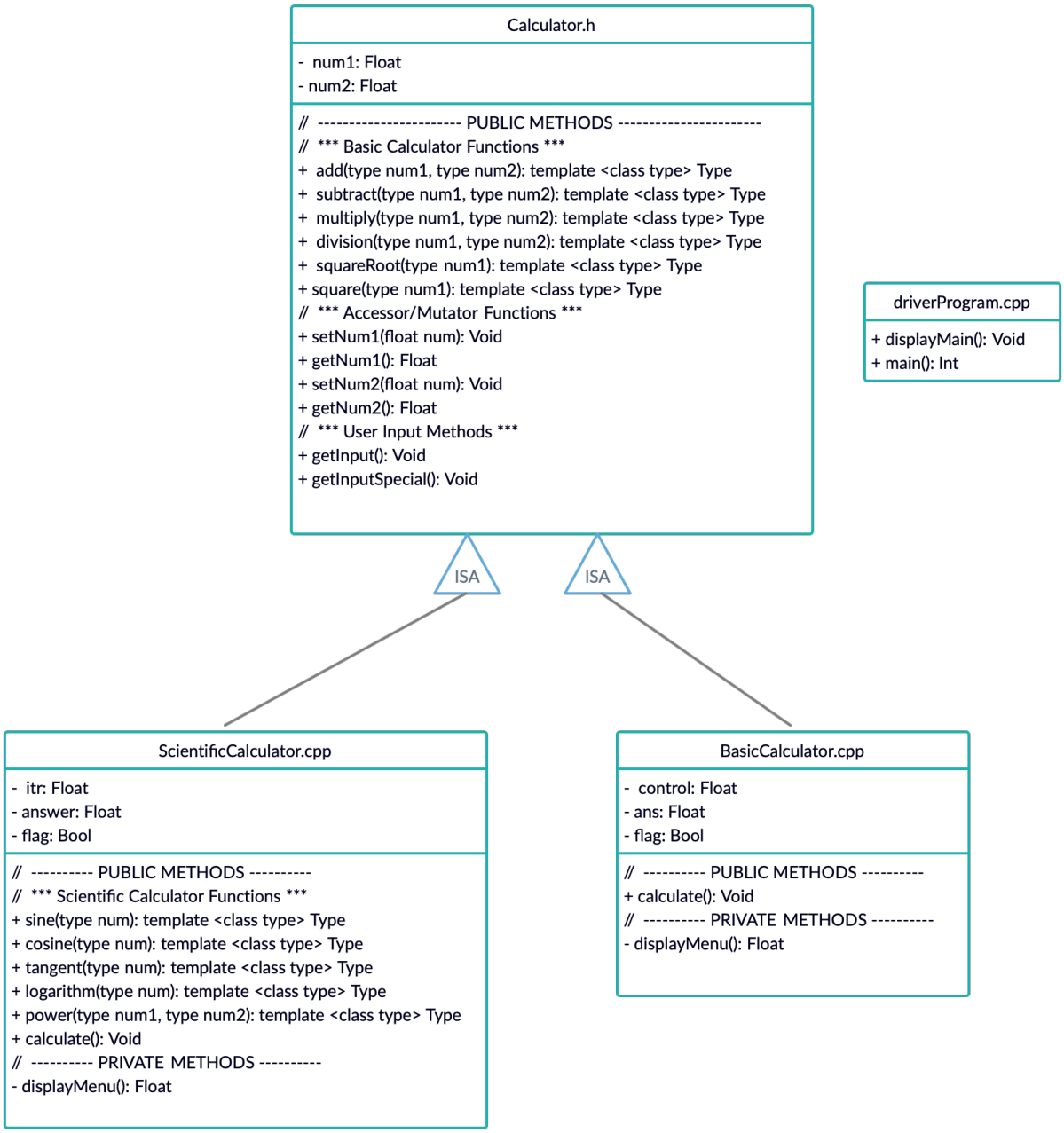
Code Plan: (UML Class Diagram & Explanation)

- **Superclass** "*Calculator.h*" is used to define all the operations that a calculator can perform. The *calculation functions* are created as templates which use a generic datatype, so all datatypes can be processed during user interaction with the menus.
- **Subclasses** "*BasicCalculator.cpp*" & "*ScientificCalculator.cpp*" inherits all the PROPERTIES and BEHAVIORS of superclass "*Calculator.h*" apart from constructors/destructors. Derived classes extend superclass to make more customizable calculator objects.
- "*driverProgram.cpp*" is used to create a menu which takes user input for the type of calculator the user wishes to use. Bonus features and hierarchy explained below…



Explained (→) I was very careful in the planning/creation of the hierarchy observed in the UML diagram above. I wanted to use the concept of **Inheritance** when creating the *Basic Calculator* and the *Scientific Calculator* classes, so I created a superclass named *Calculator*. I did this because a scientific calculator has the same functionalities as a basic calculator, with some extended "bonus" features. By using inheritance, a fundamental principal in OOP, we promote code **Reusability**, **Extendibility**, and **Data Hiding**. Reusability helps us eliminate duplicate code and speeds up the efficiency and performance of the program. Extendibility lets us extend on the superclass, which lets us derive new more customizable subclass (i.e. *Basic* and *Scientific Calc.)*. Data hiding hides private data which can only be accessed through getter/setter methods. These are all considered good OOP programming practices.

Next, I decided to make all the "calculation" functionalities (i.e. add, subtract, multiply, etc.) generic by using **templates** for two reasons. Recall, **function templates** are *special functions that can operate with generic types*. Therefore, templates are about the compiler generating code at **compile-time** instead of **run-time**. This allows us to accept all datatypes for processing user input, which is good because our program becomes more **Robust** and **Flexible**! However, it also became more error prone. For example, if we receive bad data, we typically handle the situation accordingly via an error handling/response. Error handling is a bonus feature I have included and will take up a large portion of the code.

I considered a "***Bounds Check***" error message to ensure the number being entered on the display menu is a valid option. If user input is invalid (i.e. Out-of-range) an error message is displayed and control is taken back to the main menu. This situation notifies the user of their mistake, and resumes control at the beginning of the program. Hopefully the user can correct their mistake. I also considered a "***Data Type Check***" to only accept numeric data types (i.e. integer, float, double, etc.). If a character or string is input, a type of fatal error messages should be displayed and the program should exit. The calculator cannot perform arithmetic on a character or string.

Finally, the main function that the "*driverProgram.cpp*" interacts with is separately defined in each subclass respectively and called the *"**calculate**()"* method. Once an instance of each class is created, the calculate method is called and the program executes. This all depends on which calculator the user has selected from the main menu. The *"**calculate**()"* method makes use of: 1) *Inherited functions* from superclass 2) *Defined functions* in derived class 3) *Private function* "***displayMenu()***". The private function "***displayMenu()***" is used to display a UI of the calculator object, another bonus feature. This is different for each type of calculator and I decided the user shouldn't need to know how the UI was created or configured. Therefore, I made the method private and not accessible in the main program to follow good programming practices concerned with OOP. The calculate method calls the display method **recursively**, while the program is executed or until an error is thrown.