Introduction

The project is to build a calculator program. The calculator allows users to perform basic and scientific calculations.

Requirements

The following are the requirements of the program:

For basic requirements, user can

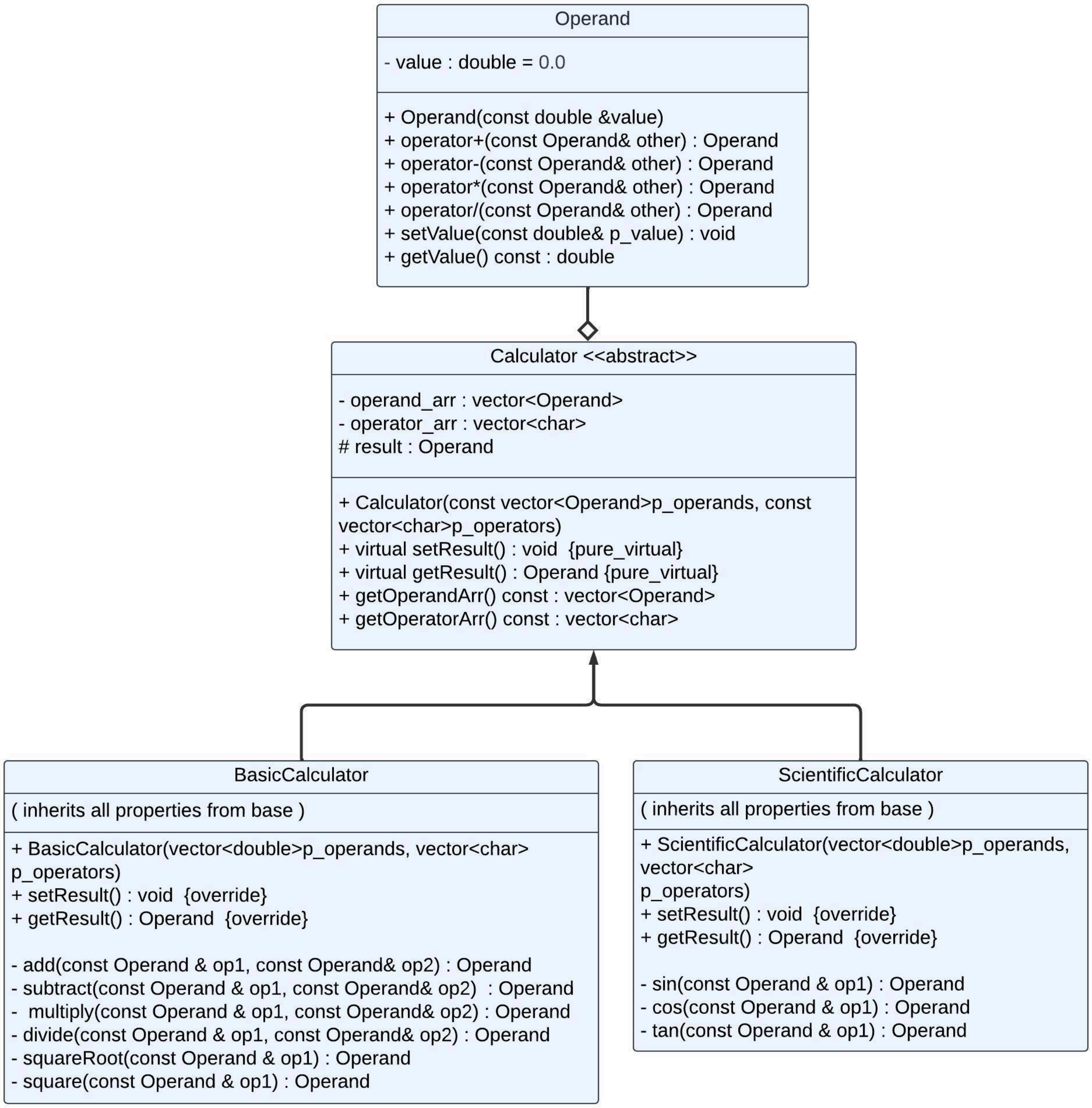
- Perform 9 types of calculations :
  Addition, Subtraction, Multiplication, Division, Square root, Square, sin, cos, tan
- Keep entering arithmetic expressions until they choose to quit
- Switch between different calculators

For additional features, user can

- Perform calculations using keystrokes (e.g., “5+6”)
- Perform left-to-right calculations with multiple operands and operators (e.g. “5 + 6 / 7 * 3)

Class Design

In this section, we briefly explain the responsibility of each class and their relationships with other classes. The below UML diagram includes all the classes that will be involved during calculation. At the top, the Operand class responsibility is to represent numerical values and support various operations. The Calculator base class, which below the Operand Class, performs calculations on operands and operators to give a result. Then, we have 2 derived classes. Firstly, the responsibility of the BasicCalculator is to evaluate arithmetic expressions for basic calculations. Secondly, the duty of the ScientificCalculator is to apply trigonometric functions on operands to get the result.



Operand

- value : double = 0.0

+ Operand(const double &value)
+ operator+(const Operand& other) : Operand
+ operator-(const Operand& other) : Operand
+ operator*(const Operand& other) : Operand
+ operator/(const Operand& other) : Operand
+ setValue(const double& p_value) : void
+ getValue() const : double

Calculator <<abstract>>

- operand_arr : vector<Operand>
- operator_arr : vector<char>
result : Operand

+ Calculator(const vector<Operand>p_operands, const vector<char>p_operators)
+ virtual setResult() : void {pure_virtual}
+ virtual getResult() : Operand {pure_virtual}
+ getOperandArr() const : vector<Operand>
+ getOperatorArr() const : vector<char>

BasicCalculator

(inherits all properties from base)

+ BasicCalculator(vector<double>p_operands, vector<char> p_operators)
+ setResult() : void {override}
+ getResult() : Operand {override}

- add(const Operand & op1, const Operand& op2) : Operand
- subtract(const Operand & op1, const Operand& op2) : Operand
- multiply(const Operand & op1, const Operand& op2) : Operand
- divide(const Operand & op1, const Operand& op2) : Operand
- squareRoot(const Operand & op1) : Operand
- square(const Operand & op1) : Operand

ScientificCalculator

(inherits all properties from base)

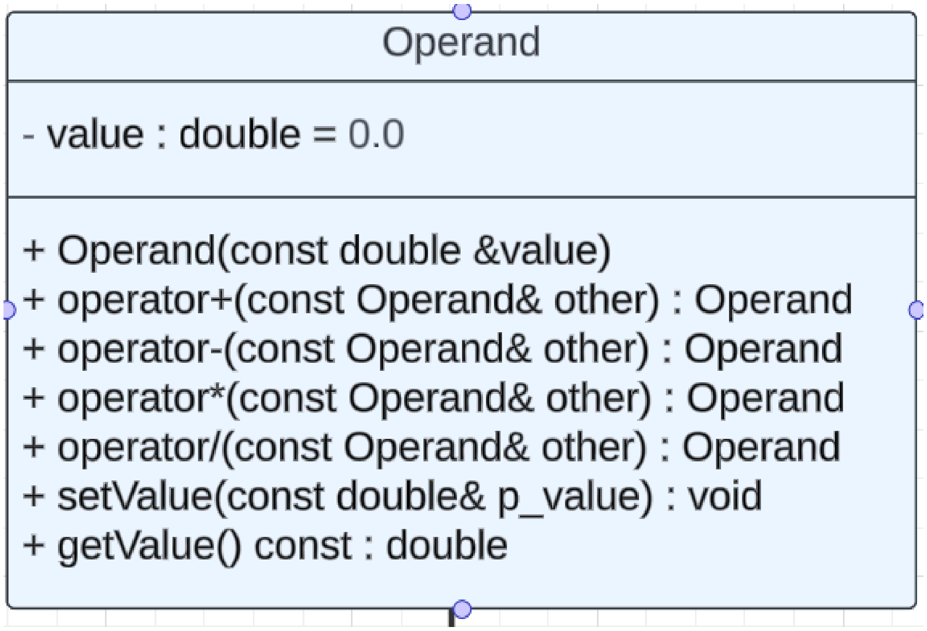
+ ScientificCalculator(vector<double>p_operands, vector<char>
p_operators)
+ setResult() : void {override}
+ getResult() : Operand {override}

- sin(const Operand & op1) : Operand
- cos(const Operand & op1) : Operand
- tan(const Operand & op1) : Operand

Class Definitions

In this section, we explain each class in detail. For each class, we define its attributes and member functions, explain relationships with other classes, and highlight the oop's principles applied.

Operand



```
Operand
- value : double = 0.0
+ Operand(const double &value)
+ operator+(const Operand& other) : Operand
+ operator-(const Operand& other) : Operand
+ operator*(const Operand& other) : Operand
+ operator/(const Operand& other) : Operand
+ setValue(const double& p_value) : void
+ getValue() const : double
```

Attributes

The variable 'value' stores the actual value of operand in double.

Methods

Constructor : We have the constructor that accepts and initializes the actual value of an operand using Member Initializer List.

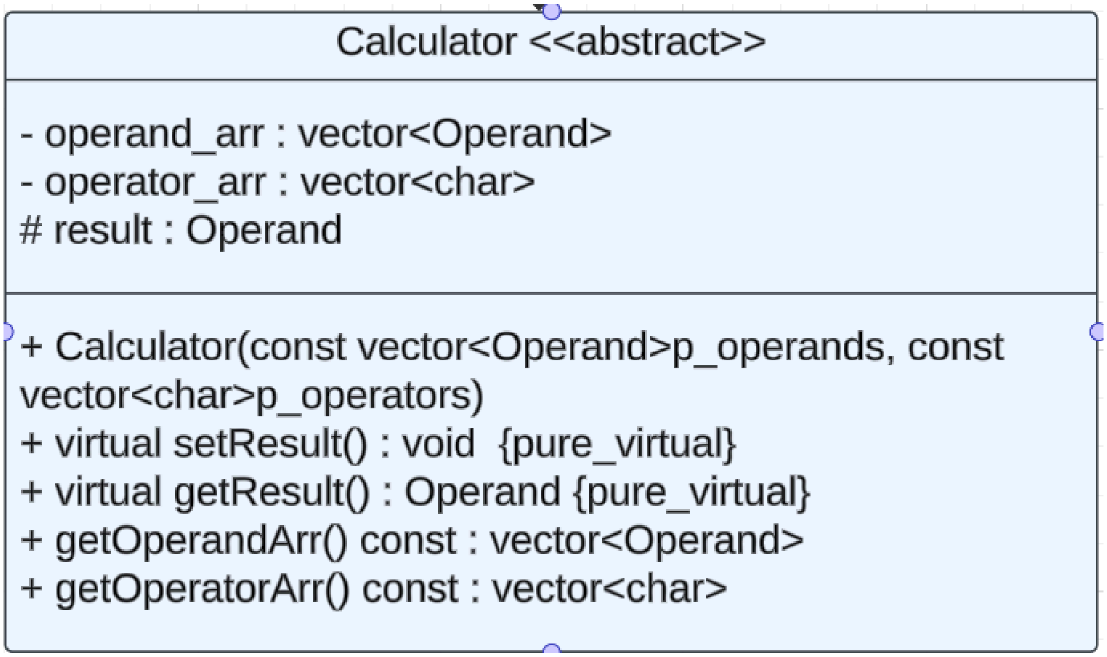
Operators Overloading : There are four main operator overloading functions: addition, subtraction, multiplication, and division. The purpose of overloading these operators is to enable one or two instances of the Operand class to perform calculations directly. Parameter 'other' is prepared and used to represent the right-hand side operand in these operations. The function returns a new Operand instance by combining the actual value and its right-hand side operand.

setValue() & getValue() : The ‘setValue()’ function sets the actual value of the operand, while the ‘getValue()’ function retrieves it; these methods are particularly useful for single-operand calculations like square or sine, which require direct access to the operand's value.

OOP analysis

We describe how encapsulation is used here. The internal state ‘value’ declared as private to avoid modification outside of class. Next, as operators return a new Operand instance, it ensures no modification made in the member variable ‘value’. In Addition, setValue() & getValue() assures controlled access to the data. For instance, we can use getValue() to validate whether the ‘value’ is initialized without medication when calculating.

Calculator



Calculator <<abstract>>

- operand_arr : vector<Operand>
- operator_arr : vector<char>
result : Operand

+ Calculator(const vector<Operand>p_operands, const vector<char>p_operators)
+ virtual setResult() : void {pure_virtual}
+ virtual getResult() : Operand {pure_virtual}
+ getOperandArr() const : vector<Operand>
+ getOperatorArr() const : vector<char>

Attributes

There are 3 member variables involved. The ‘operand_arr’ vector purpose is to store Operand objects, while ‘operator_arr’ is to store operators in char type. And, the ‘result’ of type Operand is used to store the result of a calculation. For instance, if the user inputs an expression “1+3”, then ‘operand_arr’ and ‘operator_arr’ will store [Operand(1.0), Operand(3.0)] and [“+”].

Methods

Constructor : The constructor accepts and initializes ‘operand_arr’ and ‘operator_arr’, then sets the ‘result’ Operand to 0.0. The initialized attributes will be then used by its derived class.

setResult() & getResult() : The setResult()’s responsibility is to calculate the result from a given arithmetic expression or a mathematical function. In general, the setResult() iterates through ‘operand_arr’, invokes the specific calculation based on the element ‘operator_arr’, and stores the result in the ‘result’ member variable.

OOP analysis

Encapsulation : ‘Operand_arr’, ‘Operator_arr’, ‘result’ are declared as private to avoid modification, while the setResult() and getResult() are declared as public to access those private data members.

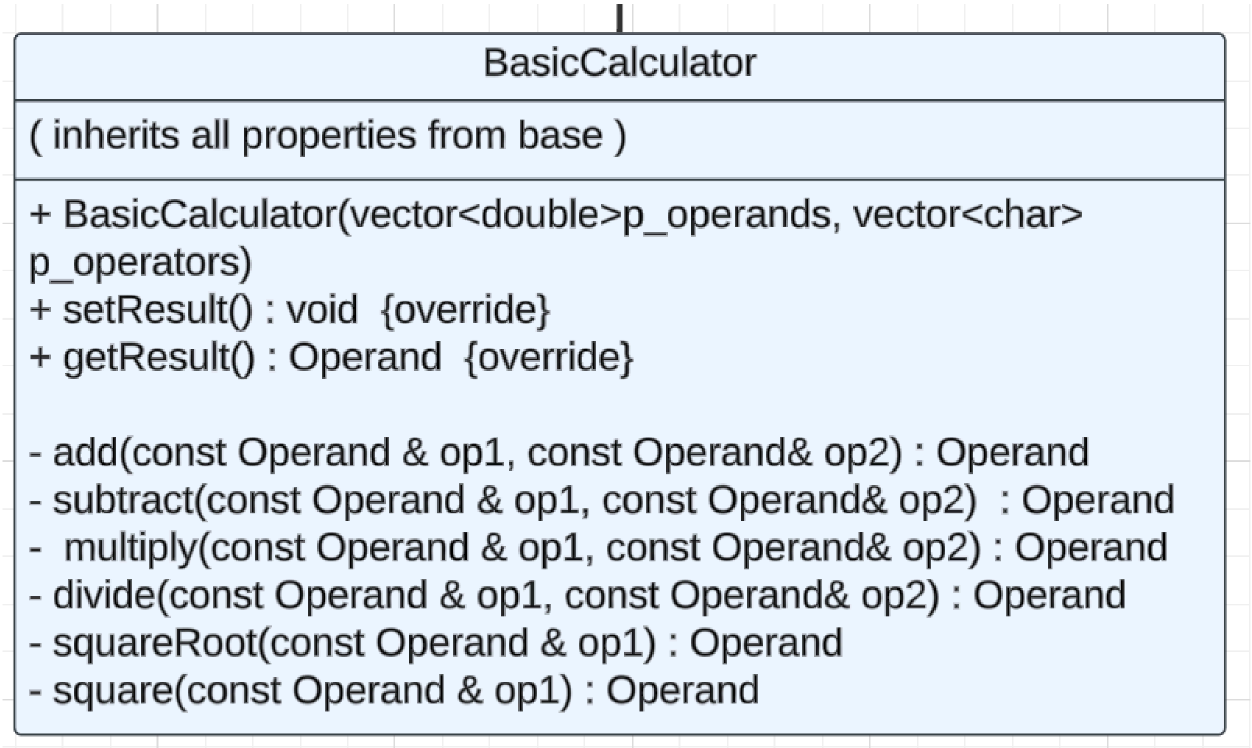
Abstraction : The Calculator class is never exposed to the user as they can’t be instantiated due to the existence of 2 pure virtual functions. From the user 's perspective, the calculator program hides the existence and details of this entire class.

Inheritance : The Calculator base class will allow its derived class to use its methods intensively. This covers from setResult() to getOperatorArr(). Firstly, the setResult() and getResult() is pure virtual since every specific calculator requires to produce and retrieve a result of a calculation. Secondly, the getOperandArr() and getOperatorArr() are public since the Calculator class has to access these methods to receive ‘operand_arr’ and ‘operator_arr’ to perform calculations.

Composition :The Calculator class uses Operand class to perform calculations. This allows the Calculator to have a “has-a” relationship with Operand objects.

Polymorphism : The class allows setResult() and getResult() to act differently based on the object. For instance, if a user instantiates Calculator (via pointer) and wants to invoke a different body of setResult(), then this features the setResult() methods to “do different things based on the object”.

BasicCalculator



BasicCalculator

(inherits all properties from base)

+ BasicCalculator(vector<double>p_operands, vector<char> p_operators)
+ setResult() : void {override}
+ getResult() : Operand {override}

- add(const Operand & op1, const Operand& op2) : Operand
- subtract(const Operand & op1, const Operand& op2) : Operand
- multiply(const Operand & op1, const Operand& op2) : Operand
- divide(const Operand & op1, const Operand& op2) : Operand
- squareRoot(const Operand & op1) : Operand
- square(const Operand & op1) : Operand

Methods

Constructor : The Constructor accepts the vector array ‘p_operands’, ‘p_operators’, and the result to 0.0. It then passes those arrays into the constructor of the Calculator class to initialize them.

Mathematical Methods : We have 6 methods as seen in the above figure. Let's first consider their body definition for addition to division. The body receives 2 Operand objects op1 and op2, adds them directly using the overloaded operator, and returns the result as an Operand object. Secondly, for squareRoot() and square(), it takes in an Operand object and receives its actual value by invoking getValue(). The received value is then applied with built in functions such as sqrt() from ‘cmath’ library.

set_result() & get_result() :
set_result() purpose is to calculate the result of an arithmetic expression. It consists of a few steps to get the result. Firstly, within set_result(), a local variable Operand type ‘result’ creates a copy of the first element of ‘operand_arr’. Secondly, each element of ‘operators_arr’ is iterated, and passed into a switch case to invoke appropriate mathematical methods. The invoked method will receive the previous result and

subsequent elements in ‘operand_arr’, then store the result by overriding local variable ‘result’.

OOP analysis

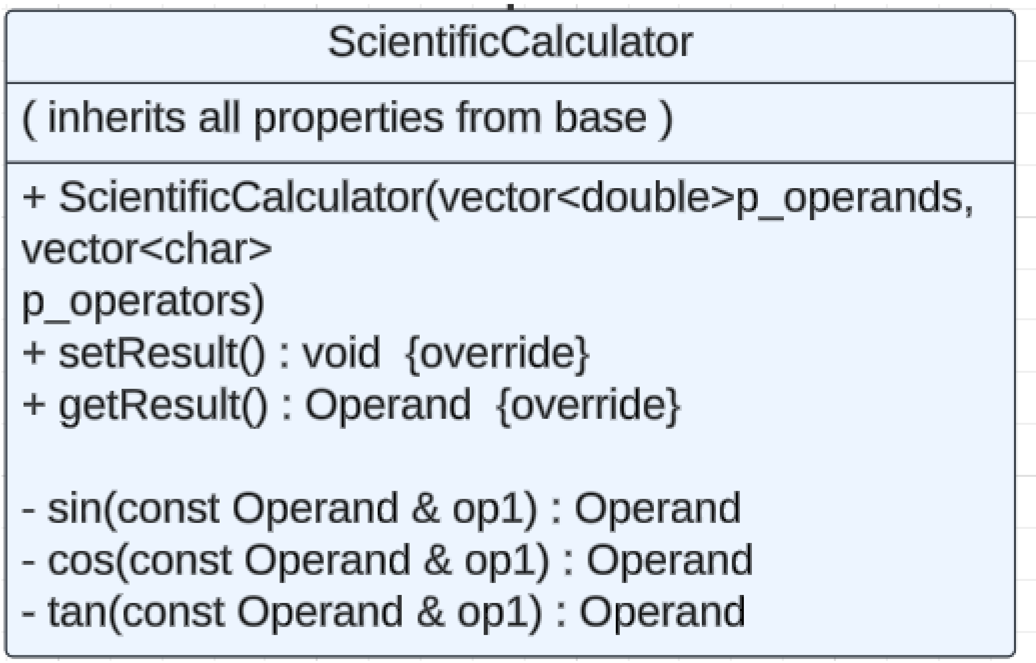
Encapsulation : is shown by encapsulating the inherited attribute ‘result’ within the class and providing getResult() and setResult() methods to modify and access it.

Abstraction : is achieved by declaring mathematical methods as private, thus hiding their complex implementations and only exposing essential features such as setResult() and getResult() to the user.

Inheritance : is utilized by having the class inherit and invoke the Calculator base class’s constructor along with all its attributes. And, the methods getOperandArr() and getOperatorArr() are inherited and used within setResult() to access operand_arr and operators_arr for performing calculations. Note that ‘result’ data member from base is also inherited.

Polymorphism : is illustrated by treating BasicCalculator as an object of the Calculator class, where setResult() and getResult() are overridden to implement the specific version of the Calculator class’s setResult() method.

ScientificCalculator



ScientificCalculator

(inherits all properties from base)

+ ScientificCalculator(vector<double>p_operands,
vector<char>
p_operators)
+ setResult() : void {override}
+ getResult() : Operand {override}

- sin(const Operand & op1) : Operand
- cos(const Operand & op1) : Operand
- tan(const Operand & op1) : Operand

Methods

Constructor: The constructor's definition is identical to BasicCalculator’s constructor.

Trigonometric Methods : There are 3 methods : sin, cos, tan. The body definition of these methods are similar to BasicCalculator. It receives an Operand object as a reference, applies a mathematical function provided by ‘cmath’ library, and returns the applied result in Operand.

setResult() & getResult() : The method's purpose is to apply trigonometric functions on Operand’s value. The setResult() stores the result after applying the trigonometric functions, It unpacks the input value in ‘operand_arr’, applies the appropriate trigonometric methods, and sets the result to the member variable ‘result’.

OOP analysis

The use of the four main OOP principles in this class is similar to their use in the BasicCalculator class.

Encapsulation: The encapsulation is completely similar to BasicCalculator.

Abstraction: For abstraction, as BasicCalculator declares its specific calculations as private, the same applies to ScientificCalculator, as seen for the sin to tan functions.

Inheritance, Reusability, Maintainability : Inheritance is shown in the same manner as BasicCalculator. In addition, the getOperandArr() and getOperatorArr() methods provide code reusability. Since both methods are declared in the base class, it avoids the extra work of rewriting those methods in the derived class. This also improves maintainability. For instance, in the future, if getOperatorArr() is required to return a vector with different types, one can just modify getOperatorArr() in the base class without any changes to the derived class.

Polymorphism, Reusability : In the BasicCalculator class discussion, we see the base class setResult() and getResult() methods are overridden by the BasicCalculator class. A similar idea is applied here as both methods are overridden by ScientificCalculator. This enhances code reusability. For instance, if a user wants to iterate through different derived Calculator classes and get their results, then getResult() can be used without the extra work of writing different getResult() methods for each class.

Others

Headers and implementations

The following are all the headings and implementation files involved in the Calculator program. All the file implementations except test_cases.h, test_cases.cpp, and main.cpp are explained in the previous section.

- .h
    - Operand.h
    - Calculator.h
    - BasicCalculator.h
    - ScientificCalculator.h
    - test_cases.h
- .cpp
    - Operand.cpp
    - Calculator.cpp
    - BasicCalculator.cpp
    - ScientificCalculator.cpp
    - test_cases.cpp
    - Main.cpp

Test cases

The ‘test_cases’ headers and implementations include test cases for individual classes (e.g. test case only for Operand.h, etc) and for the calculator program. Each test case is written as a function. The test case for the calculator program is created by comparing the results calculated using the C++ standard arithmetic operators with the results from the calculator program.

Driver program

The driver program displays the calculator menu and prompts the users to perform calculations. This action consists of a few steps.

1. Users enters one of the 3 options : basic calculator, scientific calculator, quit
2. If calculators are selected, User can enter a string arithmetic expression.
3. The string expression is then tokenized into an array of strings.
4. The function getOperatorAndOperand() receives the given array, differentiates operands and operators into 2 new arrays.
5. The 2 new arrays are then passed into specific calculator classes to get the result of an expression.