Nathan Roe
OOP with C++
EN.605.604 Section 81
4/3/2022

Module 6 Assignment: Expression Tree

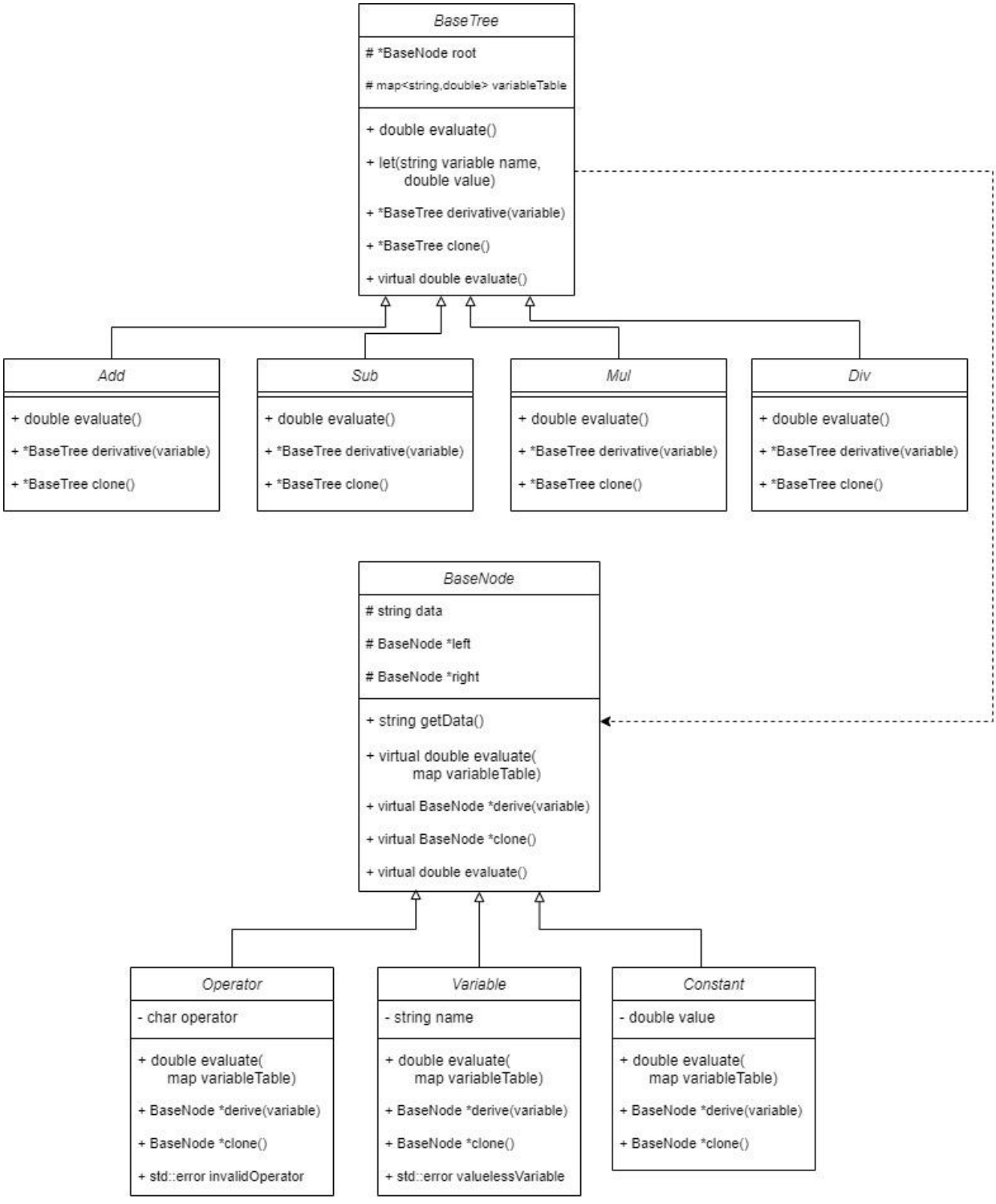
Design

For this assignment, the project requirements state that the code should generate a set of classes that can represent arithmetic equations for simple addition, subtraction, multiplication, and division for constants and variables. For variables, the values should not be stored in the variable nodes, so I will instead store them in map-based variable tables in the tree objects.

In addition to creating and evaluating simple equations, the program should be able to generate trees containing the derivation of the equations, and evaluate these as well. For operators, the derivatives are as follows:

- Addition:
  - $d(u + v) = du + dv$
- Subtraction:
  - $d(u - v) = du - dv$
- Multiplication:
  - $d(u \cdot v) = (u \cdot dv) + (du \cdot v)$
- Division:
  - $d(\frac{u}{v}) = \frac{((u \cdot dv) + (du \cdot v))}{v \cdot v}$

To do this, I will create a set of Trees, with one object to represent the four operator types modeled here (+, -, *, /). I will also create a BaseTree object with combined functionality that can also be used to create generic pointers to all operator types. I will then create a set of nodes containing the operators, constants, and variables, along with a BaseNode for standard functionality across all nodes.


BaseTree
*BaseNode root
map<string,double> variableTable
+ double evaluate()
+ let(string variable name, double value)
+ *BaseTree derivative(variable)
+ *BaseTree clone()
+ virtual double evaluate()
Add
+ double evaluate()
+ *BaseTree derivative(variable)
+ *BaseTree clone()
Sub
+ double evaluate()
+ *BaseTree derivative(variable)
+ *BaseTree clone()
Mul
+ double evaluate()
+ *BaseTree derivative(variable)
+ *BaseTree clone()
Div
+ double evaluate()
+ *BaseTree derivative(variable)
+ *BaseTree clone()
BaseNode
string data
BaseNode *left
BaseNode *right
+ string getData()
+ virtual double evaluate(map variableTable)
+ virtual BaseNode *derive(variable)
+ virtual BaseNode *clone()
+ virtual double evaluate()
Operator
- char operator
+ double evaluate(map variableTable)
+ BaseNode *derive(variable)
+ BaseNode *clone()
+ std::error invalidOperator
Variable
- string name
+ double evaluate(map variableTable)
+ BaseNode *derive(variable)
+ BaseNode *clone()
+ std::error valuelessVariable
Constant
- double value
+ double evaluate(map variableTable)
+ BaseNode *derive(variable)
+ BaseNode *clone()

```cpp
// @file
// @author Nathan Roe
//
// Demonstrates the functionality of for creating and evaluating
// Expression Tree equations and thier derivatives. Shows
// examples of calculating several trees with roots as operators
// of addition, subtraction, multiplication, and division, as
// well as generating and evaluating thier derivatives.

#include "Constant.h"
#include "Variable.h"
#include "BaseTree.h"
#include "Mul.h"
#include "Div.h"
#include "Add.h"
#include "Sub.h"
#include <iostream>

using namespace std;

// Run example expression trees and derivations, and print results
int main()
{
    // Create Expression Tree with root Addition node
    cout << "Test Addition Tree" << endl;
    BaseTree *t = new Add(new Mul(new Constant(2.3),
                                  new Variable("Xray")),
                          new Mul(new Variable("Yellow"),
                                  new Sub(new Variable("Zebra"),
                                          new Variable("Xray"))));
    t->let("Xray", 2.0);
    t->let("Yellow", 3.0);
    t->let("Zebra", 5.0);
    cout << *t << "=" << t->evaluate() << endl;

    // Evaluate and print derivative
    cout << "Test Addition Tree derivation" << endl;
    BaseTree *derived = t->derivative("Xray");
    cout << *derived << "=" << derived->evaluate() << endl;

    cout << endl;

    // Demonstrate clone capability
    cout << "Test Clone Capability" << endl;
    BaseTree *t2 = t->clone();
    cout << *t2 << "=" << t2->evaluate() << endl;

    cout << endl;

    // Demonstrate Expression Tree with root Subtraction node
    cout << "Test Subtraction Tree and Derivation" << endl;
    t = new Sub(new Mul(new Constant(2.3),
```

```
53                          new Variable("Xray")),
54                  new Mul(new Variable("Yellow"),
55                          new Sub(new Variable("Zebra"),
56                                  new Variable("Xray"))));
57      t->let("Xray", 2.0);
58      t->let("Yellow", 3.0);
59      t->let("Zebra", 5.0);
60      cout << *t << "=" << t->evaluate() << endl;
61      derived = t->derivative("Xray");
62      cout << *derived << "=" << derived->evaluate() << endl;
63
64      cout << endl;
65
66      // Demonstrate Expression Tree with root Multiplication node
67      cout << "Test Multiplication Tree and Derivation" << endl;
68      t = new Mul(new Mul(new Constant(2.3),
69                          new Variable("Xray")),
70                  new Mul(new Variable("Yellow"),
71                          new Sub(new Variable("Zebra"),
72                                  new Variable("Xray"))));
73      t->let("Xray", 2.0);
74      t->let("Yellow", 3.0);
75      t->let("Zebra", 5.0);
76      cout << *t << "=" << t->evaluate() << endl;
77      derived = t->derivative("Xray");
78      cout << *derived << "=" << derived->evaluate() << endl;
79
80      cout << endl;
81
82      // Demonstrate Expression Tree with root Division node
83      cout << "Test Division Tree and Derivation" << endl;
84      t = new Div(new Mul(new Constant(2.3),
85                          new Variable("Xray")),
86                  new Mul(new Variable("Yellow"),
87                          new Sub(new Variable("Zebra"),
88                                  new Variable("Xray"))));
89      t->let("Xray", 2.0);
90      t->let("Yellow", 3.0);
91      t->let("Zebra", 5.0);
92      cout << *t << "=" << t->evaluate() << endl;
93      derived = t->derivative("Xray");
94      cout << *derived << "=" << derived->evaluate() << endl;
95  } // End function main
96
```

```
1  /// @file
2  /// @author Nathan Roe
3  #pragma once
4  #include <iostream>
5  #include <string>
6  #include <map>
7
8  using namespace std;
9
10 /// Base object for creating nodes for Expression Trees
11 ///
12 /// Used for providing base functionality to expression
13 /// tree nodes for trees, variables, and operators
14 class BaseNode
15 {
16 public:
17     /// Default destructor for BaseTree objects
18     virtual ~BaseNode();
19
20     /// Set left node to give BaseNode
21     /// @param node - node to set as left linked node
22     void setLeft(BaseNode *node) { this->left = node; };
23     /// Set right node to give BaseNode
24     /// @param node - node to set as right linked node
25     void setRight(BaseNode *node) { this->right = node; };
26
27     /// Return the current info contained in node
28     string getData() { return this->data; };
29
30     /// Evaluated node and linked nodes as an expression
31     ///
32     /// Evaluates current and linked nodes to find equation
33     /// value based on variables contained in given variable
34     /// table
35     /// @param variableTable - map containing variable names
36     ///     and values
37     /// Returns float of the resulting equation value
38     virtual double evaluate(const map<string, double> *variableTable) = 0;
39
40     /// Calculate and return a derivative of current and linked nodes
41     ///
42     /// Creates a new tree containing the derivative of the current
43     /// nodes and linked sub-nodes on the given variable
44     /// @param variable - variable name to calulate derivative on
45     /// Returns a node containing derivative equation
46     virtual BaseNode *derive(string variable) = 0;
47
48     /// Clone node and linked sub-nodes
49     ///
50     /// Recursively clones this node and all linked nodes
51     /// Returns a clone of the current node
52     virtual BaseNode *clone() = 0;
```

```
53
54      /// Return clone of just left sub-node structure
55      BaseNode *getLeftClone();
56      /// Return clone of just right sub-node structure
57      BaseNode *getRightClone();
58
59  protected:
60      /// Set stream insertion operator as friend
61      friend ostream& operator<<(ostream& os, BaseNode& node);
62      BaseNode *left = nullptr;
63      BaseNode *right = nullptr;
64      string data = "BASE NODE";
65  };
66
67  /// Stream insertion override for printing node info
68  ostream &operator<<(ostream &os, BaseNode &node);
69
```

```
1  /// @file
2  /// @author Nathan Roe
3  #pragma once
4  #include <iostream>
5  #include <string>
6  #include <map>
7
8  using namespace std;
9
10 /// Base object for creating nodes for Expression Trees
11 ///
12 /// Used for providing base functionality to expression
13 /// tree nodes for trees, variables, and operators
14 class BaseNode
15 {
16 public:
17     /// Default destructor for BaseTree objects
18     virtual ~BaseNode();
19
20     /// Set left node to give BaseNode
21     /// @param node - node to set as left linked node
22     void setLeft(BaseNode *node) { this->left = node; };
23     /// Set right node to give BaseNode
24     /// @param node - node to set as right linked node
25     void setRight(BaseNode *node) { this->right = node; };
26
27     /// Return the current info contained in node
28     string getData() { return this->data; };
29
30     /// Evaluated node and linked nodes as an expression
31     ///
32     /// Evaluates current and linked nodes to find equation
33     /// value based on variables contained in given variable
34     /// table
35     /// @param variableTable - map containing variable names
36     ///     and values
37     /// Returns float of the resulting equation value
38     virtual double evaluate(const map<string, double> *variableTable) = 0;
39
40     /// Calculate and return a derivative of current and linked nodes
41     ///
42     /// Creates a new tree containing the derivative of the current
43     /// nodes and linked sub-nodes on the given variable
44     /// @param variable - variable name to calulate derivative on
45     /// Returns a node containing derivative equation
46     virtual BaseNode *derive(string variable) = 0;
47
48     /// Clone node and linked sub-nodes
49     ///
50     /// Recursively clones this node and all linked nodes
51     /// Returns a clone of the current node
52     virtual BaseNode *clone() = 0;
```

```
53
54      /// Return clone of just left sub-node structure
55      BaseNode *getLeftClone();
56      /// Return clone of just right sub-node structure
57      BaseNode *getRightClone();
58
59  protected:
60      /// Set stream insertion operator as friend
61      friend ostream& operator<<(ostream& os, BaseNode& node);
62      BaseNode *left = nullptr;
63      BaseNode *right = nullptr;
64      string data = "BASE NODE";
65  };
66
67  /// Stream insertion override for printing node info
68  ostream &operator<<(ostream &os, BaseNode &node);
69
```

```h
#pragma once
#include "BaseNode.h"
#include <limits>

using namespace std;

/// Create a new node for representing constants
///
/// Creates a constant with a given value
class Constant : public BaseNode
{
public:
    /// Constant node constructor
    ///
    /// Creates a constant with the given value
    Constant(double value);

    /// Evaluate constant
    ///
    /// @param variableTable - map linking variable names to values
    /// Returns value of this constant
    double evaluate(const map<string, double> *variableTable);

    /// Return derivative of this variable
    ///
    /// @param variable - variable the derivative is calculated on
    /// Returns 0
    BaseNode *derive(string variable);

    /// Create clone of current constant node
    BaseNode *clone();

private:
    /// Set default value to NaN
    double value = numeric_limits<double>::signaling_NaN();
};

```

```cpp
#include "Constant.h"

/// Constant node constructor
Constant::Constant(double value)
{
    this->value = value;
    this->data = to_string(this->value);
} // End function Constant constructor

/// Evaluate constant, return value of constant
double Constant::evaluate(const map<string, double> *variableTable)
{
    return this->value;
} // End function evaluate

/// Create clone of current constant node
BaseNode *Constant::clone()
{
    BaseNode* clone = new Constant(this->value);
    return clone;
} // End function clone

/// Return derivative of this variable (0.0)
BaseNode *Constant::derive(string variable)
{
    return new Constant(0.0);
} // End function derive

```

```
1  /// @file
2  /// @author Nathan Roe
3  #pragma once
4  #include "BaseNode.h"
5  #include <exception>
6
7  using namespace std;
8
9  /// Create a new node for representing Variables
10 ///
11 /// Creates a vairable node with a given name
12 class Variable : public BaseNode
13 {
14 public:
15     /// Variable node constructor
16     ///
17     /// Creates a variable with the given name
18     Variable(string name);
19
20     /// Evaluate variable based on given variable table
21     ///
22     /// @param variableTable - map linking variable names to values
23     /// Returns value from variable table for current variable
24     double evaluate(const map<string, double> *variableTable);
25
26     /// Return derivative of this variable
27     ///
28     /// @param variable - variable the derivative is calculated on
29     /// Returns 1 if variable name is this variable, 0 otherwise
30     BaseNode *derive(string variable);
31
32     /// Create clone of current variable node
33     BaseNode *clone();
34
35     /// Error for attempting to evaluate variable if node value
36     ///    is not contained in table
37     class valuelessVariable : public std::exception
38     {
39     public:
40         virtual const char *what() const throw()
41         {
42             return "Cannot Evaluate Variable; No Value Set";
43         }
44     } valuelessVariable;
45 };
46
```

```cpp
#include "Variable.h"
#include "Constant.h"
#include <string>

using namespace std;

/// Variable node constructor
Variable::Variable(string name)
{
    this->data = name;
} // End function Variable constructor

/// Evaluate variable based on given variable table
double Variable::evaluate(const map<string, double> *variableTable)
{
    // Attempt to find variable name in table
    auto result = variableTable->find(this->data);
    // Throw error if variable not found
    if (result == variableTable->end())
    {
        throw valuelessVariable;
    }
    // If variable is found, return value
    return result->second;
} // End function evaluate

/// Create clone of current variable node
BaseNode *Variable::clone()
{
    BaseNode *clone = new Variable(this->data);
    return clone;
} // End function clone

/// Return derivative of this variable
BaseNode *Variable::derive(string variable)
{
    // If derivative is on this variable, return 1
    if (variable == this->data)
    {
        return new Constant(1.0);
    }
    // Otherwise, return 0
    return new Constant(0.0);
} // End function derive

```

```h
/// @file
/// @author Nathan Roe
#pragma once
#include "BaseNode.h"
#include <exception>

using namespace std;

/// Create a new node for representing operators
///
/// Creates an operator of a given type
class Operator : public BaseNode
{
public:
    /// Operator node constructor
    ///
    /// Creates an operator of the given type
    Operator(char oper);

    /// Evaluate equation by finding the result of left *oper* right
    ///
    /// @param variableTable - map linking variable names to values
    /// Return double of the result of equation evaluation
    double evaluate(const map<string, double> *variableTable);

    /// Return derivative of this operator and linked nodes
    ///
    /// @param variable - variable the derivative is calculated on
    /// Returns BaseNode pointer with derivative based on operator type
    BaseNode *derive(string variable);

    /// Create clone of current operator node
    BaseNode *clone();

    /// Error for creating an operator of an invalid type or
    ///    attempting evaluation of a node without two lined
    ///    sub-nodes
    class invalidOperator : public std::exception
    {
    public:
        virtual const char *what() const throw()
        {
            return "Cannot Evaluate Operator";
        }
    } invalidOperator;

private:
    char oper = ' ';
};

```

```cpp
#include "Operator.h"
#include <map>

using namespace std;

/// Operator node constructor
Operator::Operator(char oper)
{
    // Verify operator is a valid type
    if (!(oper == '*' or oper == '/' or
          oper == '+' or oper == '-'))
    {
        throw invalidOperator;
    }
    this->oper = oper;
    this->data = this->oper;
} // End function Operator constructor

/// Evaluate equation by finding the result of left *oper* right
double Operator::evaluate(const map<string, double> *variableTable)
{
    // If no left/right nodes are found, throw error
    if (!this->left || !this->right)
    {
        throw invalidOperator;
    }

    // Evaluate left and right substructures
    double left = this->left->evaluate(variableTable);
    double right = this->right->evaluate(variableTable);

    // Calculate left *oper* right based on operator type
    switch (this->oper)
    {
    case '*':
        return left * right;
    case '/':
        return left / right;
    case '+':
        return left + right;
    case '-':
        return left - right;
    default:
        throw invalidOperator;
    }
} // End function evaluate

/// Create clone of current operator node
BaseNode *Operator::clone()
{
    BaseNode *clone = new Operator(this->oper);
    // Clone left substructure
```

```
53      if (this->left)
54      {
55          clone->setLeft(this->left->clone());
56      }
57      // Clone right substructure
58      if (this->right)
59      {
60          clone->setRight(this->right->clone());
61      }
62      return clone;
63  } // End function clone
64
65  /// Return derivative of this operator and linked nodes
66  BaseNode *Operator::derive(string variable)
67  {
68      // If no left/right nodes are found, throw error
69      if (!this->left || !this->right)
70      {
71          throw invalidOperator;
72      }
73
74      BaseNode *newOper = nullptr;
75      BaseNode *leftBranch = nullptr;
76      BaseNode *rightBranch = nullptr;
77      BaseNode *numerator = nullptr;
78      BaseNode *denominator = nullptr;
79
80      switch (this->oper)
81      {
82      // Calculate derivative of u + v
83      // (du + dv)
84      case '+':
85          newOper = new Operator('+');
86          newOper->setLeft(this->left->derive(variable));
87          newOper->setRight(this->right->derive(variable));
88          return newOper;
89
90      // Calculate derivative of u - v
91      // (du - dv)
92      case '-':
93          newOper = new Operator('-');
94          newOper->setLeft(this->left->derive(variable));
95          newOper->setRight(this->right->derive(variable));
96          return newOper;
97
98      // Calculate derivative of u * v
99      // (u*dv + v*du)
100     case '*':
101         newOper = new Operator('+');
102
103         leftBranch = new Operator('*');
104         leftBranch->setLeft(this->left->clone());
```

```
105         leftBranch->setRight(this->right->derive(variable));
106
107         rightBranch = new Operator('*');
108         rightBranch->setRight(this->right->clone());
109         rightBranch->setLeft(this->left->derive(variable));
110
111         newOper->setLeft(leftBranch);
112         newOper->setRight(rightBranch);
113         return newOper;
114
115     // Calculate derivative of u / v
116     // ((v*du - u*dv) / v*v)
117     case '/':
118         newOper = new Operator('/');
119
120         numerator = new Operator('-');
121
122         leftBranch = new Operator('*');
123         leftBranch->setLeft(this->left->clone());
124         leftBranch->setRight(this->right->derive(variable));
125
126         rightBranch = new Operator('*');
127         rightBranch->setRight(this->right->clone());
128         rightBranch->setLeft(this->left->derive(variable));
129
130         numerator->setLeft(leftBranch);
131         numerator->setRight(rightBranch);
132
133         denominator = new Operator('*');
134         denominator->setLeft(this->right->clone());
135         denominator->setRight(this->right->clone());
136
137         newOper->setLeft(numerator);
138         newOper->setRight(denominator);
139         return newOper;
140
141     // Throw error for invalid operator
142     default:
143         throw invalidOperator;
144     }
145 } // End function derive
146
```

```
1  /// @file
2  /// @author Nathan Roe
3  #pragma once
4  #include "BaseNode.h"
5  #include <iostream>
6  #include <string>
7
8  using namespace std;
9
10 /// Base object for creating Expression Trees
11 ///
12 /// Used for creating generic Expression Tree object
13 /// pointers and providing functionality
14 /// for all Expression Tree types
15 class BaseTree
16 {
17 public:
18     /// Constructors for base tree functionality
19     ///
20     /// A set of constructors for possible combinations of
21     /// variable/constant nodes and sub-trees
22     /// @param *left - constant, variable, or subtree for left
23     ///     side of equation
24     /// @param *right - constant, variable, or subtree for right
25     ///     side of equation
26     /// @param oper - character representing equation operator,
27     ///     +, -, /, or *
28     BaseTree(BaseNode *left, BaseNode *right, char oper);
29     BaseTree(BaseTree *left, BaseNode *right, char oper);
30     BaseTree(BaseNode *left, BaseTree *right, char oper);
31     BaseTree(BaseTree *left, BaseTree *right, char oper);
32
33     /// Default destructor for BaseTree objects
34     virtual ~BaseTree();
35
36     /// Evaluate equation based on current variable values
37     ///
38     /// Returns a double of the calculated equation value
39     double evaluate() { return this->root->evaluate(&this->variableTable); };
40
41     /// Set given variable to provided value
42     ///
43     /// variableName - string name of variable to set
44     /// value - float value to set
45     void let(string variableName, double value);
46
47     /// Find the drivative of the equation on the given variable
48     ///
49     /// Calculate the derivative on given value and return
50     /// a new expression tree with the derived equation and
51     /// the same variables as the original.
52     /// @param variable - variable one which to calculate derivative
```

```
53      /// Return a new variable tree with the derivative
54      virtual BaseTree *derivative(string variable) = 0;
55
56      /// Create a clone of a given expresson tree
57      ///
58      /// Create and return a copy of the expression tree with same
59      /// equation and variable values
60      /// Return a BaseTree pointer with a copied version of the tree
61      virtual BaseTree *clone() = 0;
62
63  protected:
64      /// BaseTree constructor for setting by root node
65      ///
66      /// Used internally to create derivatives from
67      /// the derivitive of the current root node
68      /// @param *root -  BaseNode to use as the new tree's root
69      BaseTree(BaseNode *root);
70
71      /// Set the stream operator as a friend
72      friend ostream& operator<<(ostream& os, BaseTree& tree);
73      BaseNode *root = nullptr;
74      map<string, double> variableTable;
75
76      /// Copy variable table names and values to another tree
77      ///
78      /// @param newTree - Copies current tree variableTable
79      ///     to this tree
80      void copyVariableTableTo(BaseTree *newTree);
81
82  private:
83      /// Creates a copy of all nodes in the tree
84      ///
85      /// Copies all nodes in the tree and returns
86      /// the root node of the copy
87      /// Returns the root node of the copied tree
88      BaseNode *cloneSubStructure();
89  };
90
91  /// Stream insertion override for printing tree info
92  ostream& operator<<(ostream& os, BaseTree& tree);
93
```

```cpp
#include "BaseTree.h"
#include "Operator.h"
#include <iostream>
#include <string>
#include <map>

using namespace std;

/// Constructor for base tree functionality using two BaseNodes
BaseTree::BaseTree(BaseNode *left, BaseNode *right, char oper)
{
    this->root = new Operator(oper);
    this->root->setLeft(left->clone());
    this->root->setRight(right->clone());
} // End function BaseTree constructor (node, node)

/// Constructor for base tree functionality a BaseNode and BaseTree
BaseTree::BaseTree(BaseTree *left, BaseNode *right, char oper)
{
    this->root = new Operator(oper);
    this->root->setLeft(left->cloneSubStructure());
    this->root->setRight(right->clone());
} // End function BaseTree constructor (tree, node)

/// Constructor for base tree functionality a BaseNode and BaseTree
BaseTree::BaseTree(BaseNode *left, BaseTree *right, char oper)
{
    this->root = new Operator(oper);
    this->root->setLeft(left->clone());
    this->root->setRight(right->cloneSubStructure());
} // End function BaseTree constructor (node, tree)

/// Constructor for base tree functionality using two BaseTree
BaseTree::BaseTree(BaseTree *left, BaseTree *right, char oper)
{
    this->root = new Operator(oper);
    this->root->setLeft(left->cloneSubStructure());
    this->root->setRight(right->cloneSubStructure());
} // End function BaseTree constructor (tree, tree)

/// Constructor for base tree using just a node for the root
BaseTree::BaseTree(BaseNode *root)
{
    this->root = root->clone();
} // End function BaseTree constructor from root

/// Default destructor for BaseTree objects
BaseTree::~BaseTree()
{
    delete this->root;
} // End function ~BaseTree

```

```
53  /// Set given variable to provided value
54  void BaseTree::let(string variableName, double value)
55  {
56      this->variableTable.insert(pair<string, double>(variableName, value));
57  } // End function let
58
59  /// Creates a copy of all nodes in the tree, return the root node
60  BaseNode *BaseTree::cloneSubStructure()
61  {
62      return this->root->clone();
63  } // End function cloneSubStructure
64
65  /// Copy variable table names and values to given tree
66  void BaseTree::copyVariableTableTo(BaseTree *newTree)
67  {
68      map<string, double>::iterator itr;
69      for (itr = this->variableTable.begin(); itr != this->variableTable.end(); +
        +itr)
70      {
71          newTree->let(itr->first, itr->second);
72      }
73  } // End function copyVariableTableTo
74
75  /// Stream insertion override for printing tree info
76  ostream& operator<<(ostream& os, BaseTree& tree)
77  {
78      // Print out current variable settings
79      map<string, double>::iterator itr;
80      for (itr = tree.variableTable.begin(); itr != tree.variableTable.end(); ++itr)
81      {
82          os << itr->first << " = " << itr->second << '\n';
83      }
84      // Print out tree nodes
85      os << *tree.root;
86      return os;
87  } // End function << override
88
```

```
1  /// @file
2  /// @author Nathan Roe
3  #pragma once
4  #include "BaseTree.h"
5
6  /// Add tree for creating an Addition Expression Treee
7  ///
8  /// Used for creating Addition Tree object
9  /// pointers and providing functionality
10 /// for adding two values/equations together
11 class Add : public BaseTree
12 {
13 public:
14     /// Constructors for creating an addition tree
15     ///
16     /// A set of constructors for possible combinations of
17     /// variable/constant nodes and sub-trees
18     /// @param *left - constant, variable, or subtree for left
19     ///     side of equation
20     /// @param *right - constant, variable, or subtree for right
21     ///     side of equation
22     Add(BaseNode *left, BaseNode *right) : BaseTree(left, right, '+') {};
23     Add(BaseTree *left, BaseNode *right) : BaseTree(left, right, '+') {};
24     Add(BaseNode *left, BaseTree *right) : BaseTree(left, right, '+') {};
25     Add(BaseTree *left, BaseTree *right) : BaseTree(left, right, '+') {};
26
27     /// Create a clone of a given addition tree
28     BaseTree *clone();
29
30     /// Find the drivative of given addition tree on the given variable
31     BaseTree *derivative(string variable);
32
33 protected:
34
35 private:
36     /// Add tree constructor for setting by root node
37     Add(BaseNode *root) : BaseTree(root) {};
38 };
39
```

```cpp
#include "Add.h"
#include "BaseTree.h"
#include "BaseNode.h"

/// Create a clone of a given addition tree
BaseTree *Add::clone()
{
    BaseNode *left = this->root->getLeftClone();
    BaseNode *right = this->root->getRightClone();
    BaseTree *clone = new Add(left, right);

    copyVariableTableTo(clone);
    return clone;
} // End function clone

/// Find the drivative of given addition tree on the given variable
BaseTree *Add::derivative(string variable)
{
    // Create derivative of root node
    BaseNode *derivationNode = this->root->derive(variable);
    // Copy derivative node set to new tree
    BaseTree *derivation = new Add(derivationNode);
    delete derivationNode;

    // Copy variable table to new tree
    copyVariableTableTo(derivation);
    return derivation;
} // End function derivative

```

```
1  /// @file
2  /// @author Nathan Roe
3  #pragma once
4  #include "BaseTree.h"
5
6  /// Sub tree for creating an Addition Expression Tree
7  ///
8  /// Used for creating Subtraction Tree object
9  /// pointers and providing functionality
10 /// for subtracting two values/equations
11 class Sub : public BaseTree
12 {
13 public:
14     /// Constructors for creating a subtraction tree
15     ///
16     /// A set of constructors for possible combinations of
17     /// variable/constant nodes and sub-trees
18     /// @param *left - constant, variable, or subtree for left
19     ///     side of equation
20     /// @param *right - constant, variable, or subtree for right
21     ///     side of equation
22     Sub(BaseNode *left, BaseNode *right) : BaseTree(left, right, '-') {};
23     Sub(BaseTree *left, BaseNode *right) : BaseTree(left, right, '-') {};
24     Sub(BaseNode *left, BaseTree *right) : BaseTree(left, right, '-') {};
25     Sub(BaseTree *left, BaseTree *right) : BaseTree(left, right, '-') {};
26
27     /// Create a clone of a given subtraction tree
28     BaseTree *clone();
29
30     /// Find the drivative of given subtraction tree on the given variable
31     BaseTree *derivative(string variable);
32
33 protected:
34
35 private:
36     /// Sub tree constructor for setting by root node
37     Sub(BaseNode *root) : BaseTree(root) {};
38 };
39
```

```cpp
#include "Sub.h"
#include "BaseTree.h"
#include "BaseNode.h"

/// Create a clone of a given subtraction tree
BaseTree *Sub::clone()
{
    BaseNode *left = this->root->getLeftClone();
    BaseNode *right = this->root->getRightClone();
    BaseTree *clone = new Sub(left, right);

    copyVariableTableTo(clone);
    return clone;
} // End function clone

/// Find the drivative of given addition tree on the given variable
BaseTree *Sub::derivative(string variable)
{
    // Create derivative of root node
    BaseNode *derivationNode = this->root->derive(variable);
    // Copy derivative node set to new tree
    BaseTree *derivation = new Sub(derivationNode);
    delete derivationNode;

    // Copy variable table to new tree
    copyVariableTableTo(derivation);
    return derivation;
} // End function derivative

```

```
1  /// @file
2  /// @author Nathan Roe
3  #pragma once
4  #include "BaseTree.h"
5
6  /// Mul tree for creating an Multiplication Expression Tree
7  ///
8  /// Used for creating Multiplication Tree object
9  /// pointers and providing functionality
10 /// for multiplying two values/equations together
11 class Mul : public BaseTree
12 {
13 public:
14     /// Constructors for creating a multiplication tree
15     ///
16     /// A set of constructors for possible combinations of
17     /// variable/constant nodes and sub-trees
18     /// @param *left - constant, variable, or subtree for left
19     ///     side of equation
20     /// @param *right - constant, variable, or subtree for right
21     ///     side of equation
22     Mul(BaseNode *left, BaseNode *right) : BaseTree(left, right, '*') {};
23     Mul(BaseTree *left, BaseNode *right) : BaseTree(left, right, '*') {};
24     Mul(BaseNode *left, BaseTree *right) : BaseTree(left, right, '*') {};
25     Mul(BaseTree *left, BaseTree *right) : BaseTree(left, right, '*') {};
26
27     /// Create a clone of a given multiplication tree
28     BaseTree *clone();
29
30     /// Find the drivative of given multiplication tree on the given variable
31     BaseTree *derivative(string variable);
32
33 protected:
34
35 private:
36     /// Mul tree constructor for setting by root node
37     Mul(BaseNode *root) : BaseTree(root) {};
38 };
39
```

```cpp
#include "Mul.h"
#include "BaseTree.h"
#include "BaseNode.h"

/// Create a clone of a given multiplication tree
BaseTree *Mul::clone()
{
    BaseNode *left = this->root->getLeftClone();
    BaseNode *right = this->root->getRightClone();
    BaseTree *clone = new Mul(left, right);

    copyVariableTableTo(clone);
    return clone;
} // End function clone

/// Find the drivative of given multiplication tree on the given variable
BaseTree *Mul::derivative(string variable)
{
    // Create derivative of root node
    BaseNode *derivationNode = this->root->derive(variable);
    // Copy derivative node set to new tree
    BaseTree *derivation = new Mul(derivationNode);
    delete derivationNode;

    // Copy variable table to new tree
    copyVariableTableTo(derivation);
    return derivation;
} // End function derivative

```

```
1  /// @file
2  /// @author Nathan Roe
3  #pragma once
4  #include "BaseTree.h"
5
6  /// Div tree for creating an Division Expression Tree
7  ///
8  /// Used for creating Division Tree object
9  /// pointers and providing functionality
10 /// for dividing two values/equations
11 class Div : public BaseTree
12 {
13 public:
14     /// Constructors for creating a division tree
15     ///
16     /// A set of constructors for possible combinations of
17     /// variable/constant nodes and sub-trees
18     /// @param *left - constant, variable, or subtree for left
19     ///     side of equation
20     /// @param *right - constant, variable, or subtree for right
21     ///     side of equation
22     Div(BaseNode *left, BaseNode *right) : BaseTree(left, right, '/') {};
23     Div(BaseTree *left, BaseNode *right) : BaseTree(left, right, '/') {};
24     Div(BaseNode *left, BaseTree *right) : BaseTree(left, right, '/') {};
25     Div(BaseTree *left, BaseTree *right) : BaseTree(left, right, '/') {};
26
27     /// Create a clone of a given division tree
28     BaseTree *clone();
29
30     /// Find the drivative of given division tree on the given variable
31     BaseTree *derivative(string variable);
32
33 protected:
34
35 private:
36     /// Div tree constructor for setting by root node
37     Div(BaseNode *root) : BaseTree(root) {};
38 };
39
```

```cpp
#include "Div.h"
#include "BaseTree.h"
#include "BaseNode.h"

/// Create a clone of a given division tree
BaseTree *Div::clone()
{
    BaseNode *left = this->root->getLeftClone();
    BaseNode *right = this->root->getRightClone();
    BaseTree *clone = new Div(left, right);

    copyVariableTableTo(clone);

    return clone;
} // End function clone

/// Find the drivative of given division tree on the given variable
BaseTree *Div::derivative(string variable)
{
    // Create derivative of root node
    BaseNode *derivationNode = this->root->derive(variable);
    // Copy derivative node set to new tree
    BaseTree *derivation = new Div(derivationNode);
    delete derivationNode;

    // Copy variable table to new tree
    copyVariableTableTo(derivation);
    return derivation;
} // End function derivative

```

```
Test Addition Tree
Xray = 2
Yellow = 3
Zebra = 5
((2.300000*Xray)+(Yellow*(Zebra-Xray)))=13.6
Test Addition Tree derivation
Xray = 2
Yellow = 3
Zebra = 5
(((2.300000*1.000000)+(0.000000*Xray))+((Yellow*(0.000000-1.000000))+(0.000000*
  (Zebra-Xray))))=-0.7

Test Clone Capability
Xray = 2
Yellow = 3
Zebra = 5
((2.300000*Xray)+(Yellow*(Zebra-Xray)))=13.6

Test Subtraction Tree and Derivation
Xray = 2
Yellow = 3
Zebra = 5
((2.300000*Xray)-(Yellow*(Zebra-Xray)))=-4.4
Xray = 2
Yellow = 3
Zebra = 5
(((2.300000*1.000000)+(0.000000*Xray))-((Yellow*(0.000000-1.000000))+(0.000000*
  (Zebra-Xray))))=5.3

Test Multiplication Tree and Derivation
Xray = 2
Yellow = 3
Zebra = 5
((2.300000*Xray)*(Yellow*(Zebra-Xray)))=41.4
Xray = 2
Yellow = 3
Zebra = 5
(((2.300000*Xray)*((Yellow*(0.000000-1.000000))+(0.000000*(Zebra-Xray))))
  +(((2.300000*1.000000)+(0.000000*Xray))*(Yellow*(Zebra-Xray))))=6.9

Test Division Tree and Derivation
Xray = 2
Yellow = 3
Zebra = 5
((2.300000*Xray)/(Yellow*(Zebra-Xray)))=0.511111
Xray = 2
Yellow = 3
Zebra = 5
((((2.300000*Xray)*((Yellow*(0.000000-1.000000))+(0.000000*(Zebra-Xray))))-
  (((2.300000*1.000000)+(0.000000*Xray))*(Yellow*(Zebra-Xray))))/((Yellow*(Zebra-
  Xray))*(Yellow*(Zebra-Xray))))=-0.425926

```