Brown University ENGN2912B Fall 2017

Scientific Computing in C++

Assignment 05

Evaluation of Algebraic Expressions

The main goal of assignments 5 and 6 is to implement a program that will take as input an algebraic expression such as

"(((3*5)+(4*(7+(8*5))))*(7-(9*3)))"

represented as a C-style string, and produce as output the result of evaluating such expression as a floating point number. You will have to implement a class named `AlgebraicTreeNode`, which together with the following `main()` function will compile in assignment 6 into a command line program named `calc6`. This program will take an algebraic expression such as the one shown above as the only argument, and will print the result of evaluating such expression. In future assignments, we will build an interactive calculator by adding a user interface to this class.

```
// Calc6.cpp
#include <iostream>
#include "AlgebraicTreeNode.hpp"

using namespace std;

int main(int argc, const char * argv[]) {

  // error handling
  if(argc<2) {
    cout << "usage: calc expression" << endl;
    return -1;
  }

  const char* inputExpression = argv[1];

  // constructor parses the expression and builds the tree
  AlgebraicTreeNode* root = new AlgebraicTreeNode(inputExpression);

  if(root->isInvalid()) {
    cout << "\"" << inputExpression << "\"" << "is an invalid expression" << endl;
    return -2;
  }

  // evaluate the tree
  double value = root->evaluate();

  cout << "\"" << inputExpression << "\" = " << value << endl;

  return 0;
}
```

But since this task can be quite time consuming, we will split the work into two parts. This is how software is developed, in small increments. It is very important in the design of software to organize the development in reasonable increments, so that small parts could be tested individually before they are integrated into large and complex software systems.

In this assignment you will start implementing the `AlgebraicTreeNode` class. You will implement constructors and additional functions to build the data structure by hand for specific expressions in the main() function. You will also implement a function to generate a string representation of the expression from the data structure, which will be a useful tool to debug your code, and a function to evaluate the expression from the data structure.

Once this functionality is properly implemented and debugged, in Assignment 6 you will add more functionality to the `AlgebraicTreeNode` class. You will implement a constructor to build the data structure from an expression string. This constructor will have to parse the expression string and populate the data structure.

In this assignment you should create a file named Calc5.cpp using the following code, which contains a new `main()` function, as well as a few functions where you will construct the data structure for a number of expressions by hand. You will also have to create a file name `AlgebraicTreeNode.hpp` containing the interface to the class, and a file named `AlgebraicTreeNode.cpp` containing the implementation of the class. You will have to set your CMake file to compile these files together, and to produce an executable named `calc5`.

```
// Calc5.cpp
#include <iostream>
#include "AlgebraicTreeNode.hpp"

using namespace std;

AlgebraicTreeNode* newExpr1() {
  AlgebraicTreeNode* root = new AlgebraicTreeNode();
  // construct data structure for expression "(8*5)"
  return root;
}

AlgebraicTreeNode* newExpr2() {
  AlgebraicTreeNode* root = new AlgebraicTreeNode();
  // construct data structure for expression "(4*((8*5)+7))"
  return root;
}

AlgebraicTreeNode* newExpr3() {
  AlgebraicTreeNode* root = new AlgebraicTreeNode();
  // construct data structure for expression "(((3*5)+(4*(7+(8*5))))*(7-(9*3)))"
  return root;
}

int main(int argc, const char * argv[]) {

  AlgebraicTreeNode* root = newExpr1();
  // AlgebraicTreeNode* root = newExpr2();
  // AlgebraicTreeNode* root = newExpr3();

  if(root->isInvalid()) {
    cout << "expression is invalid" << endl;
    return -2;
  }

  char*  str   = root->toString();
  double value = root->evaluate();

  cout << "\"" << str << "\" = " << value << endl;

  return 0;
}
```

Valid Algebraic Expressions

Regular expressions are families of expressions defined by recursive construction rules. In our case, valid algebraic expressions are defined by the following two rules:

1) A **number** is an expression.
2) Given two expressions, **expLeft** and **expRight**, and a binary operator **#**, the result of applying the operator to the two expressions **(exprLeft # exprRight)** is a new expression.

In our case, a **number** will be a value representable as a `double` data type, and an operator is one of the four arithmetic operators: `ADD(+)`, `SUBTRACT(-)`, `MULTIPLY(*)`,or `DIVIDE(/)`. The data structure that we are about to construct should be able to explicitly represent the recursive application of these rules in the construction of an algebraic expression. The evaluation of an expression yields a **number** as a result, but expressions will be represented in non-evaluated form. As a result, it will be possible to evaluate any sub-expression at any particular time.

The AlgebraicTreeNode class

The data structure underlying the representation of the algebraic expressions as defined above is a binary tree. A binary tree is composed of leaf nodes, and regular nodes. Each regular node has exactly two children. Each child can be either another regular node or a leaf node. Except for the root node, which in our representation will not have a parent, every regular and leaf node has exactly one parent. For each node, there is a unique path from the node to the root node through parents.

We will implement the `AlgebraicTreeNode` class in an incremental fashion. You should carefully debug your code before you proceed to add more functionality. Instances of the `AlgebraicTreeNode` class will be used to represent both regular and leaf nodes of the tree. The leaf nodes will correspond to numbers, and the non-leaf nodes to operators. The instance of the `AlgebraicTreeNode` class constructed in the `main()`program above and pointed to by the pointer named `root`, will represent the root of the tree. The `toString()` and `evaluate()` functions applied to a particular node will recursively perform the evaluation of the branch of the children of the particular node and will yield a value for the sub-expression corresponding to the branch of the tree rooted at the particular node. When applied to the `root` node in the `main()` function, it will yield a value for the whole expression. Let us consider the following code fragment as the starting point for the class interface. You will have to add additional functions, as described below.

```
#ifndef _AlgebraicTreeNode_hpp_
#define _AlgebraicTreeNode_hpp_

#include <string.h>
#include <math.h>

enum AlgebraicTreeNodeType {
  INVALID, NUMBER, ADD, SUBTRACT, MULTIPLY, DIVIDE
};

class AlgebraicTreeNode {

public:

  // default constructor
  AlgebraicTreeNode();
  // destructor
  ~AlgebraicTreeNode();
  // string representation
  char* toString() const;
  // evaluator
  double evaluate() const;

private:

  AlgebraicTreeNode*          _parent;
  AlgebraicTreeNodeType       _type;
  double                      _value;
  AlgebraicTreeNode*          _childLeft;
  AlgebraicTreeNode*          _childRight;

};

#endif // _AlgebraicTreeNode_hpp_
```

The same class will be used to represent numbers as leaf nodes, and operations as regular nodes. In this design, an instance of the class has five private class variables, and initially three public class functions. You will have to add several additional functions as described below. The value of the private field

```
AlgebraicTreeNode* _parent;
```

should be a pointer to the parent node of the current node, if the current node is not the root node. For the root node the value of this private field should be the null pointer `(AlgebraicTreeNode*)0`. The private field

```
AlgebraicTreeNodeType _type;
```

will be used to indicate what type of node the instance of the class represents. The possible values are defined by the `enum`

```
enum AlgebraicTreeNodeType {
  INVALID, NUMBER, ADD, SUBTRACT, MULTIPLY, DIVIDE
};
```

Note that this `enum` is placed outside of the class interface definition. It can also be placed inside the class interface definition, either in the public or in the private interface sections. You should experiment moving this `enum` to the various possible places to understand what the implications are. The value `NUMBER` will be used to represent leaf nodes. In this case the private field

```
double _value;
```

should contain the value of the number associated with the leaf node. In this case the value of the two private fields

```
AlgebraicTreeNode* _childLeft;
AlgebraicTreeNode* _childRight;
```

should be equal to `(AlgebraicTreeNode*)0`. The `_type` values `ADD`, `SUBTRACT`, `MULTIPLY`, `DIVIDE` will be used to represent the corresponding arithmetic operations as regular nodes. In this case the values of the private fields `_childLeft` and `_childRight` should be valid pointers to class instances representing the left and right sub expressions, and the value of the private field `_value` should be 0.0. The `_type` value `INVALID` will be used to indicate errors encountered by other constructors, which you will implement later.

The default Constructor

You need to implement the default constructor

```
AlgebraicTreeNode();
```

The default constructor should initialize the private fields as follows: `_parent`, `_childLeft`, and `_childRight` to `(AlgebraicTreeNode*)0`; `_type` to `INVALID`; and `_value` to 0.0.

The Number Constructor

You need to implement a second constructor with the following signature

```
AlgebraicTreeNode(const double value,
                  AlgebraicTreeNode* parent=(AlgebraicTreeNode*)0);
```

This constructor should initialize the private fields as follows: `_parent` to the value of the second constructor parameter `parent`; `_childLeft`, and `_childRight` to `(AlgebraicTreeNode*)0`; `_type` to `NUMBER`; and `_value` to the value of the first constructor parameter `value`. Since in this assignment we want to construct the tree structure in the main() function, this constructor should be declared public.

The Operation Constructor

Since the left and right children nodes may not have been constructed at the time an operation node is constructed, we will not pass the pointers to the left and right children as parameters to the constructor, and we will set the values of the `_childLeft` and `_childRight` private fields later. You need to implement a third constructor with the following signature

```
AlgebraicTreeNode(AlgebraicTreeNodeType operation,
                  AlgebraicTreeNode*    parent=(AlgebraicTreeNode*)0);
```

This constructor should initialize the private fields as follows: `_parent` to the value of the second constructor parameter `parent`; `_childLeft`, and `_childRight` to `(AlgebraicTreeNode*)0`; `_type` to the value of the first constructor parameter `operation` if this value is `ADD`, `SUBTRACT`, `MULTIPLY`, or `DIVIDE`, and otherwise to `INVALID`; and `_value` should be initialized to 0.0.

Connecting the tree nodes

Since in this assignment we want to construct the tree structure in the main() function, and `_parent`, `_childLeft` and `_childRight` are private fields, you need to implement set functions with the following signatures to interconnect the nodes of the tree

```
void setParent(const AlgebraicTreeNode* parent);
void setChildLeft(const AlgebraicTreeNode* childLeft);
void setChildRight(const AlgebraicTreeNode* childRight);
```

In every directed tree there is a unique path from every node to the root node. The function of the `_parent` private field is to enable the construction of this path. We will find uses for this field later.

Examples

The following code can be used to construct the data structure for the expression "5.0"

```
AlgebraicTreeNode* root = new AlgebraicTreeNode(5.0);
```

The following code can be used to construct the data structure for the expression "(5+6)"

```
AlgebraicTreeNode* root     = new AlgebraicTreeNode(AlgebraicTreeNodeType::ADD);
AlgebraicTreeNode* rootLeft = new AlgebraicTreeNode(5.0, root);
root->setChildLeft(rootLeft);
AlgebraicTreeNode* rootRight = new AlgebraicTreeNode(6.0, root);
root->setChildRight(rootRight);
```

A more compact implementation

```
AlgebraicTreeNode* root = new AlgebraicTreeNode(AlgebraicTreeNodeType::ADD);
root->setChildLeft(new AlgebraicTreeNode(5.0, root));
root->setChildRight(new AlgebraicTreeNode(6.0, root));
```

The Destructor

You need to implement the class destructor

```
~AlgebraicTreeNode();
```

The class destructor should not do anything if the node type is `NUMBER or INVALID`. Otherwise, it should delete `_childLeft` and/or `_childRight` because these are pointers to instances of the class created in the heap.

Add the following two statements to the main() function, before the last return statement, and make sure that your program completes successfully without crashing in the destructor.

```
delete [] str;
delete root;
```

Auxiliary functions

You need to implement the following public auxiliary functions, which will return `true` or `false` depending on the value of the private variable `_type`.

```
bool isInvalid() const;
bool isNumber() const;
bool isOperation() const;
```

The evaluate() function

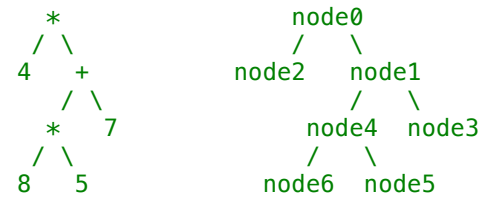
In this assignment you will implement a recursive function to evaluate an expression.

```
double evaluate() const;
```

If the node type is NUMBER, the function should return the value stored in the variable _value; otherwise, the function should evaluate the left child and the right child and return the result of applying the operation to those two values. In addition, if the node type is INVALID, the function should return 0.

Depth-First traversal

Note that any recursive function, which returns from a leaf node, and calls itself on the left and right children before returning from a regular node, results in a depth-first traversal of the tree. The evaluate() function is a particular case. For example, the expression "(4*((8*5)+7))" should result in a tree with 7 nodes with the following structure, where the numbering of the nodes reflects the particular order of creation, which is not unique.



However, the depth-first traversal order of this tree is

```
node0,node2,node1,node4,node6,node5,node3
```

To verify that your tree is properly constructed and the implementation of the evaluate() function performs the correct steps you should do this: 1) add a new private int variable named _id to the class; 2) implement a mechanism to enumerate the nodes as they are constructed by the different constructors, and assign consecutive _id values to new instances of the class; 3) add a print statement before the return statement, showing the _id value, and to indicate what operation has been performed. For a node of type NUMBER, such as node2 above, it should print

```
node2 : 4
```

For a node of type ADD, such as node1 above, it should print

```
node1 : (40+7)=47
```

where 40 is the result of evaluating the left child, and 7 is the result of evaluating the right child. Although this case should never happen, for a node of type INVALID, it should print

```
node34 : INVALID
```

The toString() function

To further verify that your data structure is correct, you will implement the public method

```
char* toString() const;
```

which should return a C-style (null terminated) string allocated in the heap, with a string representation of the expression, reconstructed from the data structure. Let's ignore for the moment that we would not know before hand how long the string should be, and let's assume that we have already allocated a string of the proper length, and that we have initialized it to the value "".

For a node of type NUMBER this function should concatenate a string representation of the `_value` field to the string. For an operation type node such as ADD, this function should append "(" to the string, recursively call itself on _childLeft with a pointer to the end of `str` as argument, append the operator symbol "+" to `str`, recursively call itself on _childRight with a pointer to the end of `str` as argument, append ")" to `str`, and return.

Since we first need to allocate the string, and we need to recursively call the function with the string as an argument, we will use this implementation of the function `toString()` :

```
char* AlgebraicTreeNode::toString() const {
  unsigned N = this->_toStringLength();
  char* str = new char[N];
  memset(str,'\0',N*sizeof(char));
  _toString(str);
  return str;
}
```

The default behavior in C and C++ is not to initialize arrays of built-in data types. The library function `memset()`, defined in the header file <cstring>, can be used to initialize any array to a constant value. You need to implement the following two additional private functions

```
unsigned _toStringLength() const;
unsigned _toString(char* str) const;
```

Note that we defined the function `_toString()` as returning an unsigned value, which will be ignored in the call to the `toString()` function. To prevent you from having to compute the length of the string every time you have to concatenate something to it, this function should return the length of the string printed within the function call. You should use this value to figure out where to continue printing on the string. For example, if the expression to be printed is "(8*5)", the initial value of the string pointed to by `str` on the first call to `_toString` will be "", the final value will be "(8*5)", and the returned value will be 5. The function `_toString` will first concatenate "(" to `str`. Since the string "(" has length 1, then the function will call itself on the left child of the current node with parameter `str+1`. This call will result in the string "8" written on the string starting at the value of the pointer passed as the argument, and the value 1 being returned. Then "*" will be written starting at str+2, another call on the left child of the current node with parameter str+3, which will result in the string "5" written starting at that position and the value 1 returned. Finally the string ")" will be written starting at position str+4, resulting in the desired string of length 5 written on the array. The function `_toStringLength()` can be identical to the function , but with the statements where strings are written removed. To know the length of the string resulting from printing a number, the `_toStringLength()` function could use a static char array of sufficient length (say 1024). In fact, the `_toString()` function could use the same process, and then copy the printed string from the static array onto the pointer received as argument. To do this without making mistakes, it is a good idea to implement the method to write on the static

string as a separate function

```
unsigned _ftoa(const double value, char* str = (char*)0);
```

which should not be a class function, since it doesn't need to have access to class variables. This function should return the length of the written string. If a non-null pointer is passed as the second argument, the function should write the corresponding string starting at that location. If a null pointer is passed as argument, the function should use a static array as described above.

Include the signature in the class header file as a global function, and the implementation in the class implementation file. The static temporary array should be defined inside the scope of this function, and used only when a null pointer is passed as the second argument.

In the implementation of this function you could use the C library function `sprintf()`, which is defined in the header file `<string.h>`, and returns the length of the printed string. For example, the following statement will result in the value printed with four decimals

```
unsigned n = sprint(str,"%.4f",_value);
```

Look for the definition of the `sprintf()` and `printf()` functions on-line and learn about formatted printing.

http://www.cplusplus.com/reference/cstdio/sprintf/
http://www.cplusplus.com/reference/cstdio/printf/

It is also acceptable for you to implement this function from scratch.