Brown University ENGN2912B Fall 2017

Scientific Computing in C++

Assignment 06

Where should the _parent variable be set?

In Assignment 5 you implemented the Number and the Operation constructors with the following signatures

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

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

which allow you to set the _parent variable within the constructor, if the parent is know at that time. Then you implemented the function

```
void setParent(const AlgebraicTreeNode* parent);
```

to change the value of the `_parent` variable, whenever it would be necessary. The use of this public function may lead to errors that are difficult to detect, resulting in inconsistent data structures. The fact is that for an Operation node, the only two other nodes which should refer to it as their parent are its two children. As a result, a better and safer design is to eliminate the parent argument from the constructors, remove the `setParent()` function from the interface, and make the functions `setChildLeft()` and `setChildRight()` set the _parent variable. Implement these changes and verify that your implementation still runs as before. The new signatures of the two constructors should be

```
AlgebraicTreeNode(const double value);
AlgebraicTreeNode(AlgebraicTreeNodeType operation);
```

The signatures of the following two functions should be the same as before, but their implementation should have changed

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

Since you should have removed the `setParent()` function from the interface, you should also remove all the calls to this function from your code.

The C-string Expression Constructor

We continue here developing the `AlgebraicTreeNode` class, which we started in Assignment 5. You will implement a new constructor

```
AlgebraicTreeNode(const char* expression);
```

which will take as argument a C-style string containing an algebraic expression such as

“(((3*5)+(4*(7+(8*5))))*(7-(9*3)))”

and will construct the data structure. You will add this new constructor to your previous implementation of the class, and you will test using the `main()` function shown below.

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

using namespace std;

// const char debugExpr[] = “(8*5)";
// const char debugExpr[] = “(4*((8*5)+7))";
const char debugExpr[] = "(((3*5)+(4*(7+(8*5))))*(7-(9*3)))";

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

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

  // const char* inputExpression = argv[1];
  const char* inputExpression = debugExpr;

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

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

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

  delete [] str;
  delete root;

  return 0;
}
```

You should compile your code into a command line program named `calc6` which will take an algebraic expression such as the one shown above as the only argument, will construct the data structure, and then it will perform the same steps as the program that you developed for Assignment 5. To speed up the debugging, you will define the expression string as a `const char` global array, and you will revert to getting the expression from the command line after your code works properly. You should test it with several different constant input strings before switching to the command line argument.

Valid Expressions

Remember that regular 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.

Where the binary operator **#** is one of the four arithmetic operators: `ADD(+)`, `SUBTRACT(-)`, MULTIPLY(*),or DIVIDE(/).

We need to define more precisely what we will consider a valid C-string representation of an algebraic expression. Our constructor should be able to parse all the valid expressions. It should also be able to detect invalid expressions, and fail gracefully. If your constructor detects invalid syntax in the input string, rather than crashing, it should return a node of type INVALID.

A) We will accept any string representation of a floating point number, such as those understood by the system functions atof() and strtod() (we will come back to this issue below), as a valid representation of a NUMBER.

B) Given two valid C-string representations strLeft and strRight of expressions, and a binary operator such as ADD, the result of concatenating the strings "(", strLeft, "+", strRight, and ")" is a valid C-string representation.

Note that with this definition valid C-string representations of expressions contain no white space (space, tab, or newline characters). Also note that all the operations are binary, and bound by parentheses. For example, "(7*3)" is a valid, but "7*3" is not; "((7*3)*4)" and "(7*(3*4))" are valid, and evaluate to the same numerical value, but neither "(7*3*4)" nor "7*3*4" are valid. Also note that unnecessary parentheses are not allowed. For example "(7.32)" is not valid, but "7.32" is valid. One potential problem is that the characters "+" and "–" can appear in a string both as operators, and as part of the representation of a number. For example, and contrary to standard practice, "(3+-7)" is valid, but "(3+(-7))" is not valid; "(-8.15e+3+-4.15)" is valid (and confusing) but "((-8.15e+3)+(-4.15))" is not. Despite all of these potential problems the two construction rules A & B described above are not ambiguous, and rather simple. To make things even simpler, you can first assume that the numbers are restricted to non-negative integers, i.e., "0", or sequences of digits which start with a non-zero digit, such as "1207" or "23600091". Once your constructor works properly on expressions with this restriction, you can go back to the general case. However, before you start coding read the discussion about parsing numbers below.

The parser Functions

You will implement two private functions with the following signatures

```
unsigned _parseNumber(const char* expression);
unsigned _parseExpression(const char* expression);
```

which correspond to the two recursive rules of valid string construction. The basic structure of the constructor will be

```
AlgebraicTreeNode::AlgebraicTreeNode(const char* expression) {
  unsigned length = _parseExpression(expression);
}
```

This implementation is incomplete. To start with you need to initialize all the variables before the _parseExpression() function is called. If the expression string is valid, the _parseExpression() function should recursively construct the binary tree, and return the length of the input expression string. If any error occurs during the execution, the value returned by _parseExpression() will be smaller than the length of the expression string. In this case the construction of the binary tree would be aborted in the process resulting in a non valid data structure. You need to add code to the constructor, after the call to the `_parseExpression()` function, to clean up the partially constructed data structure in case of errors. If an error occurs during the parsing process, this constructor should return a node of type `INVALID`, identical to those constructed using the default constructor, i.e., with all the pointers equal to `(AlgebraicTreeNode*)0`, and the _value parameter equal to 0.0.

The _parseNumber() Function

```
unsigned _parseNumber(const char* expression);
```

This function should convert the initial portion of the string pointed to by `expression` to double, save the resulting value in the variable `this->_value`, and return the number of characters read from the string. If the string does not start with a number, it should set the variable `this->_value` to 0.0. For example, if the input string is "8.1*6.2)+7.5)", this function will set the variable `this->_value` to 8.1, and will return the value 3. If, on the other hand the input string is "*6.2)+7.5)", this function will set the variable `this->_value` to 0.0, and will return the value 0. An invalid syntax error has occurred when the function returns the value 0.

For you implementation, you have three options:

1) You can implement the whole function from scratch by analyzing the string characters one by one. The valid string representation of a floating point number consists of an optional plus ('+') or minus sign ('-') and then a decimal number. A *decimal number* consists of a nonempty sequence of decimal digits possibly containing a radix character '.', optionally followed by a decimal exponent. A decimal exponent consists of an 'E' or 'e', followed by an optional plus or minus sign, followed by a nonempty sequence of decimal digits, and indicates multiplication by a power of 10.

2) You can use the system function or `atof()` ("ascii to float", defined in the header file `<stdlib.h>`). If you use this function, you still need to figure out where the end of the initial substring parsed into a number ends

```
double atof(const char *nptr);
```

http://man7.org/linux/man-pages/man3/atof.3.html

3) You can use the system function `strtod()` ("string to double", defined in the system header file `<stdlib.h>`), which will do most of the necessary work. The second parameter `endptr` is a pointer to `char*`. If `endptr is not` NULL, a pointer to the character after the last character used in the conversion is stored in the location referenced by `endptr`. If you subtract the value of `nptr` from this value, the result is the number of characters used in the conversion.

```
double strtod(const char *nptr, char **endptr);
```

http://man7.org/linux/man-pages/man3/strtod.3.html

For example, the following code fragment will result in the value of the variable `value` equal to 0.034567, the value of the variable `length` equal to 9, and the pointer `endPtr` pointing to "+0.723e-4"

```
const char* begPtr = "3.4567e-2+0.723e-4;
char* endPtr = &begPtr;
double value = strtod(begPtr, &endPtr);
unsigned length = endPtr-begPtr;
```

The _parseExpression() Function

```
unsigned _parseExpression(const char* expression);
```

This function should parse an expression starting at the character pointed to by the parameter `expression`, set the variables of the current node, and return the number of characters used in the conversion. Given the two valid string construction rules, if the first character `expression[0]` is equal to a left parenthesis `'('`, then the node should be an operation node. In this case the left parenthesis should be followed by the left child expression, a character indicating the type of operation (`'+'`, `'-'`, `'*'`, or `'/'`), the right child expression, and a right parenthesis `')'`. The left and right children nodes should be allocated (using the default constructor), with the values saved in the corresponding variables, and the `_parseExpression()` function should be called on the left and right children with the `expression` pointer argument properly advanced to the start of each expression. If the first character `expression[0]` is not equal to a left parenthesis `'('`, then the string pointed to by the `expression` pointer should start with a number. In this case, the current node should be set to type `NUMBER`, and the `_parseNumber()` function should be called to set its fields.

If any syntax error is detected, or any of the various recursive calls to the two parsing functions returns 0, the node should be cleaned up (i.e., the non-null children should be deleted), and the node type should be set to `INVALID`. An alternative to this cleaning up process, after an error is detected, is to add a new bool variable `onlyCheckSyntax` to the two parser functions

```
unsigned _parseNumber(const char* expression, bool onlyCheckSyntax);
unsigned _parseExpression(const char* expression, bool onlyCheckSyntax);
```

so that when these functions are called with this variable set to true, the same code is executed, except for the code associated to allocating or setting variables of `AlgebraicTreeNode` data type, which will be skipped. The values returned should be the same in both cases. If the first call to `parseExpression()` in the constructor returns the length of the input expression string, then the string had been parsed without errors. In this case a second call to `parseExpression()` with the variable `onlyCheckSyntax` set to false will populate the tree without errors. The running time will double, but there is no need to implement the clean up process.

```
AlgebraicTreeNode::AlgebraicTreeNode(const char* expression) {
  unsigned length = _parseExpression(expression,true);
  if(length == strlen(expression))
    _parseExpression(expression,false);
}
```

The behavior of the constructor should be the same in both cases. You can decide which approach you want to follow.

Allowing White Space in Valid Expressions

The system function

```
int isspace(int c);
```

defined in the system header file `<ctype.h>`, can be used to skip white space. For example, if `expression` is a pointer to a null-terminated C-string, the following code fragment will advance the pointer `str` to the first non white space character in the string, or to the end of the string. Note that this is not an infinite loop because the C-string is null terminated, and '\0' is considered white space.

```
char* str = expression;
while(isspace(static_cast<int>(*str++));
```

In fact, there is a large family of functions defined in the same system header file to classify and operate on characters

http://man7.org/linux/man-pages/man3/isspace.3.html

Modify your parser to allow valid expression strings to contain white space.

Implementing Operations Between Expressions

Now we want to operate on expressions represented as trees of `AlgebraicTreeNode` data structures. But since arithmetic operations between expressions should only be allowed at the root level, we are going to create a new class `AlgebraicTreeExpression`. Use the following code fragments to create the the files `AlgebraicTreeExpression.hpp` and `AlgebraicTreeExpression.cpp`, and modify your main() functions to make use of this class.

```
// AlgebraicTreeExpression.hpp
#ifndef _AlgebraicTreeExpression_hpp_
#define _AlgebraicTreeExpression_hpp_
#include "AlgebraicTreeNode.hpp"

class AlgebraicTreeExpression {

public:

  ~AlgebraicTreeExpression();
   AlgebraicTreeExpression();
   AlgebraicTreeExpression(const char* expression);

  double evaluate() const;
  char*  toString() const;

protected:

  AlgebraicTreeNode* _root;

};

#endif // _AlgebraicTreeExpression_hpp_


// AlgebraicTreeExpression.cpp
#include "AlgebraicTreeExpression.hpp"

AlgebraicTreeExpression::~AlgebraicTreeExpression() {
  if(_root!=(AlgebraicTreeNode*)0) delete _root;
}
AlgebraicTreeExpression::AlgebraicTreeExpression():
  _root(new AlgebraicTreeNode()) {
}
double AlgebraicTreeExpression::evaluate() const {
  return ((_root!=(AlgebraicTreeNode*)0)?_root->evaluate():0.0;
}
char* AlgebraicTreeExpression::toString() const {
  return ((_root!=(AlgebraicTreeNode*)0)?_root->toString():(char*)0;
}
AlgebraicTreeExpression::AlgebraicTreeExpression(const char* expression):
  _root(new AlgebraicTreeNode(expression)) {
}
```

Note that the two constructors initialize the pointer to the root node using the corresponding constructors of the AlgebraicTreeNode class, and the evaluate() and toString() functions just evaluate the functions with the same names from the AlgebraicTreeNode class and return the results. This is a standard mechanism to hide functionality from an existing class. Your first task here is to implement the copy constructor for this class

```
AlgebraicTreeExpression(AlgebraicTreeExpression& src);
```

The default copy constructor would only copy the value of the root pointer, resulting in two instances of the AlgebraicTreeExpression class pointing to the root of the same tree. This behavior could create problems as soon as one of the two instances of the AlgebraicTreeExpression class is deleted, since the other one would be pointing to a block of memory in the heap, which had been deleted. Accessing that memory may result in unexpected behavior and/or program crash. This copy constructor should traverse the tree pointed to by the src argument, and create a duplicate tree. To accomplish this task the best approach is to implement a copy constructor for the AlgebraicTreeNode class

```
AlgebraicTreeNode(AlgebraicTreeNode& src);
```

and then implement the copy constructor for the AlgebraicTreeExpression class in a similar fashion as the other constructors.

```
AlgebraicTreeExpression::AlgebraicTreeExpression(AlgebraicTreeExpression& src):
  _root(new AlgebraicTreeNode(/* what should we put here ?*/)) {
}
```

Your second task here is to implement the copy constructor for the AlgebraicTreeExpression class.

```
AlgebraicTreeExpression& operator = (AlgebraicTreeExpression& rhs);
```

Note that the first thing that this operator has to do is to delete the tree pointed to by the left hand side of the assignment. In general, the copy constructor and the assignment operator are closely related. One easy way to way to implement the assignment operator is to first implement a swap function

```
void swap(AlgebraicTreeExpression& rhs);
```

which should interchange the root pointers this->_root and rhs._root. The assignment operator could then make a copy of rhs, apply swap to the copy just made, and finally delete the copy, which at this point would point to the root of the tree originally pointed to by this->_root.

Add this code fragment to your main() function, and test

```
AlgebraicTreeExpression exp1("(7-(3+4))");
AlgebraicTreeExpression exp2("((5*7)+(100/3))");
AlgebraicTreeExpression exp3(exp1);
AlgebraicTreeExpression exp4 = exp2;
cout << "exp1 = \"" << exp1.toString() << "\"" <<endl;
cout << "exp2 = \"" << exp2.toString() << "\"" <<endl;
cout << "exp3 = \"" << exp2.toString() << "\"" <<endl;
```

Finally, once your code work, read the first two expressions from the command line. Usage details are explained in the sample Calc6.cpp file.

```
AlgebraicTreeExpression exp1(argv[1]);
AlgebraicTreeExpression exp2(argv[2]);
```

The third task here is to implement some arithmetic operators. We will start with the increment operator

```
AlgebraicTreeExpression& operator += (AlgebraicTreeExpression& src);
```

Which will enable the following statement

```
exp1 += expr2;
```

This operator should create a new AlgebraicTreeNode of type ADD, make the left child of this node point to the root of expr1, make a copy of expr2, set the right child of the new node to the root of the copy of expr2, delete the copy of expr2 being careful not to delete the tree pointed to, and finally, making the new node the root of exp1. For example, this code segment

```
AlgebraicTreeExpression exp1("(7-(3+4))");
AlgebraicTreeExpression exp2("((5*7)+(100/3))");
exp1 += exp2;
cout << "exp1 \"" << exp1.toString() << "\"" <<endl;
```

should result in the following output to the terminal

```
((7-(3+4))+((5*7)+(100/3)))
```

We have decided not to implement an addition operator with this signature

```
AlgebraicTreeExpression operator + (AlgebraicTreeExpression& rhs);
```

Because implementing a strategy to prevent memory leaks gets too complicated at this point.

The implementation of the following three in place operators

```
AlgebraicTreeExpression& operator -= (AlgebraicTreeExpression& src);
AlgebraicTreeExpression& operator *= (AlgebraicTreeExpression& src);
AlgebraicTreeExpression& operator /= (AlgebraicTreeExpression& src);
```

would be almost identical, except for the type of the new node created to combine the left and right hand sides of the operation. A single private function with an additional type parameter should be implemented containing the common code, and called from the various operators. Making almost identical copies of similar functions is a common source of errors. Implement such function with the following signature

```
AlgebraicTreeExpression& _operatorInPlace
  (AlgebraicTreeNodeType type, AlgebraicTreeExpression& src);
```

Then, the public in place operators can be implemented as follows

```
AlgebraicTreeExpression& AlgebraicTreeExpression::operator+=(AlgebraicTreeExpression& rhs) {
  this->_operatorInPlace(ADD,rhs);
  return *this;
}
```

Note that if we define the following variables

```
AlgebraicTreeExpression exp1("(3+4)");
AlgebraicTreeExpression exp2(“((5/7)+(6*8))”);
AlgebraicTreeExpression exp3(“((7-(8/9))”);
AlgebraicTreeExpression exp4;
```

and then add the following statements

```
exp4 = exp1+expr2+expr3;
cout << “expr4 = \” =” << expr4.toString() << endl;
```

The compiler will report an error, because operator+() is not defined. However, the following code should compile and run

```
exp4 = exp1;
exp4 += expr2;
exp4 += expr3;
cout << “expr5 = \” =” << expr5.toString() << endl;
```

Add the previous two code fragments to your main() function and compare the results.