CS11 C++ Lab 6 - Class Inheritance

Algebraic Expressions

This week you will use your knowledge of C++ class inheritance to complete a program that can represent, evaluate, and manipulate simple algebraic expressions. Expressions can contain the standard arithmetic operators `+ - * /`, as well as numeric literals (e.g. `2.6`) and symbols (e.g. `pi`). Symbols will also support assignment, so that you can set them to different values, or you can assign the result of a calculation to a symbol.

Now, one of the harder parts of this program is actually parsing these expressions, and fortunately you don't have to worry about that. There are some well known tools like Bison and Flex, which take care of the hard parts for you. So, we can concentrate on the fun part - actually playing around with the algebraic expressions!

The parser you will use has been generated by Bison and Flex, and can handle simple expressions like this:

- `3.14159`
- `2.3 - 6.4 * 5`
- `b * b - 4 * a * c`
- `n * (n - 1) / 2`

Also, the parser can handle simple assignment operations, where a symbol appears on the lefthand side of the assignment operator:

- `pi = 3.14159`
- `discriminant = b * b - 4 * a * c`
- `sum_n = n * (n - 1) / 2`

By the end of this lab, you should have a simple program, unimaginatively named **`alge`**, which can take input like the above expressions, and print out their results.

NOTE 1:

Make sure to follow the class and method names specified in this lab. These are the names used by the parsing and evaluation code, and if you don't use the same names then the program won't compile.

NOTE 2:

You will be creating quite a few classes in this assignment. Commenting is always important, but it is especially important when you have this many classes declared in only a few files. **Make sure to provide at least a brief comment for each class you create.** There is no reason to be over-verbose, but each class should have a comment stating its purpose in the overall picture.

Internal Representation

We are used to reading algebraic expressions in the form above, but for a computer, that form is not very useful. What works better is building up a tree of objects in memory, that represents the expression to be evaluated. This is called an ***abstract syntax tree*** (AST for short), and can then be processed by a program to do various things. Of course, the implementation of the AST can make the processing very easy to do, or very complicated. Fortunately, we can apply C++ class inheritance to this problem to make it pretty easy.

The Environment

If we are going to evaluate expressions with symbols in them, or if we are going to assign values to symbols, we obviously need a place to store the names and values of symbols. We'll call this the "environment," and we will represent this with a class called `Environment`. You won't have to implement this class; a simple implementation is provided for you to use, later on this page.

There are two member functions on `Environment` that you care about:

`double getSymbolValue(const string &name) const`

> This member function allows you to fetch the value of a particular symbol. *If the symbol is not recognized, a `runtime_error` is thrown.* Of course, the `runtime_error` will contain a suitable message.

`void setSymbolValue(const string &name, double value)`

> This member function allows you to set the value of a symbol. If the symbol doesn't exist yet, it is added automatically. If the symbol already exists, the value is simply changed.

You will find this `Environment` class *very* useful in your implementation.

Expressions

An "expression" is anything that evaluates to a specific value. All of these are expressions:

- `4` (a number)

- `sum` (a symbol)
- `2 * pi` (an arithmetic expression, composed from two other expressions)

You can see that it is possible to build up very complicated expressions by chaining smaller, simpler expressions together. Therefore, you will implement a hierarchy of expression classes that represent these different components, and that can be hooked together into a structure that represents the overall expression.

All of the expression class declarations must appear in a header file named **`expressions.hh`**. (You should declare all of the classes inline in this one header file, since that will make everybody's lives much easier.)

The `Expression` Class

The first step in building up this hierarchy is to implement a base class called **`Expression`**, which declares what *every single expression* (or subexpression in a larger expression) will provide.

Expressions can be evaluated, to determine their value. Therefore, your `Expression` class should provide a member function with this signature:

```
double evaluate(const Environment &env) const
```

In other words, an expression can be evaluated, using an `Environment` object to provide the values of any symbols that appear in the expression. The result will always be a `double` value, and the process of evaluation won't change the expression itself.

Of course, this is only an abstraction. It doesn't make any sense to provide an implementation of this function in the `Expression` class itself, so make this function *pure virtual*. This means that subclasses of `Expression` must provide the appropriate implementation of this function.

Finally, don't forget that since `Expression` is intended to be a base class, it *must* provide a virtual destructor. The destructor won't be pure virtual (i.e. it must have an implementation), but the body will be empty because the `Expression` class has nothing to clean up. In effect, you are simply telling the C++ compiler to make the destructor virtual.

Symbols and Values

Symbols and numeric values are both obviously expressions - really simple ones. A numeric value of course evaluates to itself, and a symbol evaluates to some numeric value, based on what the environment says. So, after your `Expression` class is implemented, you can implement these two subclasses next.

- Create a class called **`Value`**, which represents any literal numeric value specified in an expression. Since a numeric value *is* an expression, `Value` should derive from `Expression`.

  `Value` just requires a single data-member of type `double`. This is the actual numeric value that was specified in the expression. Also, provide a single constructor that takes the value to store.

  `Value` has a *very* simple implementation of `evaluate`. You should have no problem figuring that one out...

- Create another class called **`Symbol`**, which represents a symbol in an expression, like `pi` or `x`. Since a symbol *is* an expression... you get the idea.

  `Symbol` also requires a single data-member, the *name* of the symbol. Use a `string` member to store this name. Again, provide a constructor that takes a symbol-name to store. Finally, you also need to provide an accessor for the name. (You will see why this is necessary in a little while!)

  The `Symbol` implementation of `evaluate` is only slightly more complicated than for `Value`, because `Symbol` must actually look up the symbol's value in the environment. So, look up the symbol's value, and return the value as the result.

  I hear you ask, "What if the symbol isn't defined in the environment?!" Recall that the `Environment` class will throw an exception if the specified symbol name is undefined. This exception will propagate all the way to the main input-loop, where it will be caught and an appropriate error message will be displayed to the user. So, your implementation of `Symbol` doesn't have to do anything for that particular error scenario.

This should take care of the most basic kinds of expressions! Now we can move on to more complicated expressions.

Binary Operators

Our expressions can contain the `+ - * /` operators, corresponding to sums, differences, products, and quotients. You might notice that these all have something in common: They all take *two* expressions, a left-hand side (LHS) expression, and a right-hand side (RHS) expression. A good object-oriented design will incorporate this commonality into a single class, allowing for the specializations to be provided in subclasses.

Thus, you should create a **`BinaryOperator`** class that provides the common features of all binary operators. The parent-class of `BinaryOperator` should be `Expression`, but notice that we still don't know exactly how to implement the `evaluate` member-function in this class. This is because `BinaryOperator` is still an abstraction.

`BinaryOperator` should have two data-members:

- An `Expression`-pointer for the LHS of the operator
- An `Expression`-pointer for the RHS of the operator

`BinaryOperator` should also provide a single constructor that takes two arguments, the pointers to the LHS and RHS sub-expressions. Something like this:

```
BinaryOperator(Expression *pLHS, Expression *pRHS)
```

Of course, use assertions to make sure those pointers aren't null!

Provide an accessor for each of these sub-expression pointers. For safety, your accessors should return a `const` pointer, so that callers can't change the expressions you return! Also, don't provide mutators for these values - that will complicate things too much.

Finally, `BinaryOperator` should also provide a destructor, which should `delete` both the LHS and RHS sub-expressions.

Once you have completed this `BinaryOperator` class, go ahead and implement the following four classes, each of which derives from `BinaryOperator`. By setting their parent-class to be `BinaryOperator`, you are stating that they *are* binary operators, but also that they are expressions, since `BinaryOperator` is also an `Expression`.

- **`AddOper`**
- **`SubOper`**
- **`MulOper`**
- **`DivOper`**

Note that *constructors are not inherited,* so each of these classes should also provide a two-argument constructor that takes LHS and RHS pointers, and calls the parent-class constructor with these values.

Each of these classes should also provide the appropriate implementation of `evaluate`. Evaluation of the binary operator also involves evaluation of the LHS and RHS sub-expressions, which should be easy; just call `evaluate` on them to get the answers back. Note that you can run into a divide-by-zero situation with the `DivExpr` class! Don't just let that error happen; you should detect it and throw a `runtime_error` to indicate the problem to the user:

- Compute the RHS value first.
- If the RHS value is zero, throw a `runtime_error` with an appropriate message.
- Otherwise, compute the LHS value and then return the result of the division operation.

Because these classes derive from `BinaryOperator`, there is no need to provide separate accessors or a destructor. These things are automatically *inherited* from `BinaryOperator`. So, all these classes needed to do was to provide the proper implementation of `evaluate`. This has reduced the amount of code we have to write.

Unary Operators

You might have noticed above that `-` was lumped in with the binary operators, but that's only *mostly* correct, because `-` can also be used as a *unary operator*.

So, create another class **`UnaryOperator`**, to represent unary operators. Its implementation should follow the same pattern as the `BinaryOperator` class, but instead of two `Expression`-pointers for LHS and RHS, `UnaryOperator` should just have *one* `Expression`-pointer for the child-expression. So, provide a constructor that takes the `Expression`-pointer, and an accessor that exposes it, just like before.

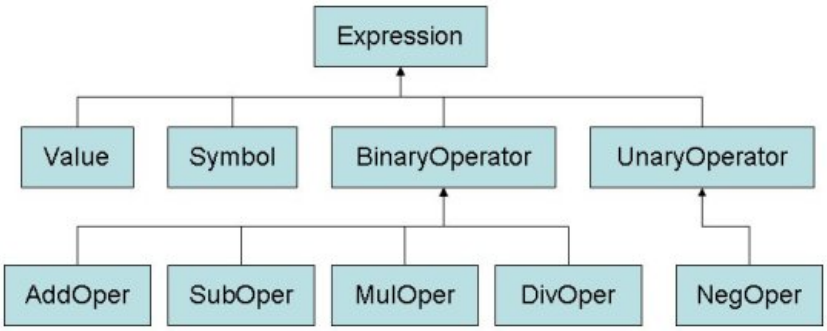
Right now we only have a single unary operator, the negate operator `-`, so implement one child-class that derives from `UnaryOperator`, called:

- **`NegOper`**

This class should provide an implementation of `evaluate` that simply negates whatever the child-expression evaluates to.

Testing Your Expressions

And that should do it! We should now have all the classes necessary to model and evaluate simple algebraic expressions. To review, this is the class hierarchy that you should now have:



Since you have created so many expression classes, and since the parsing code makes it hard to diagnose compile errors in your expression classes, you can use this simple test program to track down basic compilation errors. You can compile it with this command:

```
g++ -Wall exprtest.cc -o exprtest
```

If your expression code compiles properly with this test program, *giving no warnings*, then you shouldn't have too much trouble compiling your code with the full algebra program.

Next we need to hook in the commands that our program will support.

Commands

The other class hierarchy required for this program is for representing the different commands that can be performed. Don't worry; this class hierarchy is *much* simpler than the expression hierarchy. Put all of these class declarations into a header file named `commands.hh`. (Again, you should provide all the command implementations inline, to make everybody's lives simpler.)

The `Command` Class

The base class should be called **`Command`**, and it provides an abstraction of what all commands should be able to do. As such, it is very simple - it should provide a single member-function like this:

```
void run(Environment &env);
```

This `run` function simply performs the command. The `Environment` class is provided because all of our commands involve expressions, and of course our expressions can include symbols.

Just like the `Expression` class, `Command` is a generic abstraction, so it shouldn't provide any implementation for `run`; it simply declares that behavior, and child-classes of `Command` must provide the implementation. Also, don't forget to give `Command` the right kind of destructor.

The `PrintCommand` Class

After you have completed your abstraction, the first implementation should be the `PrintCommand` class. This is an easy command to implement; it just evaluates an expression and then prints out the result. Thus, `PrintCommand` should have a single constructor that takes one argument, an `Expression *` (a pointer to an `Expression` object). That pointer should be stored in a private member variable.

The `PrintCommand` implementation of `run` should be *really easy* to figure out. Just evaluate the expression, then print the result to `cout`. You might want to put something like `" = "` before you print the result, just to make it look nicer.

Don't forget to delete the `Expression`-pointer in the `PrintCommand` destructor.

The `AssignCommand` Class

The other command to implement is the assignment command, provided by the **`AssignCommand`** class. This command is similar to the `PrintCommand` class, but it must evaluate an expression and then set a symbol to that value.

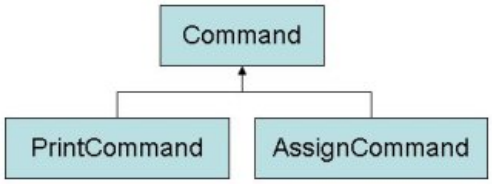
The `AssignCommand` class needs two data-members, a `Symbol`-pointer (the symbol to set), and an `Expression`-pointer (the value to set it to). Write a constructor that takes these two arguments, and store the pointers into internal data-members. Again, don't forget to delete them in the destructor.

The `AssignCommand` implementation of `run` is only slightly more complicated than the `PrintCommand` implementation. Evaluate the expression, then store the result into the `Environment` object. You will need to know the symbol's name to perform the last part; this is where the name-accessor on the `Symbol` class will come in handy. After you have stored the symbol's value, you should also print out some details so that the user knows what happened. For example, you might print a line that says:

```
x = 6.35
```

This way they see confirmation of what they requested.

Again, here is a diagram of the command class hierarchy you should now have:



Multiple Definitions

One of the common issues with a program like this is that `commands.hh` must include `expressions.hh`, and other source files include both of these headers. Thus, you can end up with "multiple definition" errors from the compiler. To keep this from happening, you should add *include guards* to your `.hh` files. (Note that `.cc` files don't use include-guards, since `.cc` files shouldn't ever `#include` other `.cc` files!)

An include guard is simple to create. For `expressions.hh` you might do this:

```
#ifndef _EXPRESSIONS_HH_
#define _EXPRESSIONS_HH_

... // the rest of your header file contents go here

#endif // _EXPRESSIONS_HH_
```

All this does is to define a preprocessor constant named `_EXPRESSIONS_HH_`, the first time that the file is included in another source file. On the first time, the contents of the header file are included, but if the header file is included multiple times in a particular source file, the contents will be skipped because the constant has already been defined once. Thus, you can make sure header files are only included once by using this simple technique.

The comment after the final `#endif` is just there for the sake of readability; it is not required by the syntax. It is very helpful to have that comment when you are using other preprocessor directives inside the same file.

You should also add an include guard to your `commands.hh` header file, to keep it from being included multiple times in a particular source file.

Pulling It All Together

Once you have all of your classes written, you should be able to compile your program and try it out. Here are the other files you will need:

- `lex.yy.cc` - the lexer, generated by Flex
- `alge-parser.tab.hh` - the parser, generated by Bison
- `alge-parser.tab.cc`
- `environment.hh` - the `Environment` class
- `alge.hh` - the main program
- `alge.cc`

As usual, you need to specify all the source files to `g++`:

```
g++ -Wall lex.yy.cc alge-parser.tab.cc alge.cc -o alge
```

Or, you can do this the quick way:

```
g++ -Wall *.cc -o alge
```

Then, you should be able to run `alge`, and try entering some algebraic expressions! Make sure that your program correctly reports errors if an undefined symbol is used, or if a divide-by-zero situation occurs.

Once you have verified that everything works, and everything is commented, leave your work in your `~/cs11/cpp/lab6` directory.

Extra Credit

If you breeze through everything else in this lab, or you want to keep tinkering, here are some things you can do for "extra credit" (meaning: your own personal edification). You will probably need the lexer source and the parser source to do most of these tasks. Also, a makefile will help you build the lexer and parser descriptions into source files you can compile.

- Extend the grammar to use the `^` (the caret) to represent powers, and add a new binary-operator subclass to compute these powers. For example, `x ^ 2` is the same as `x * x`. Try supporting expressions on the right side of the `^` operator! You might want to use `pow(a, b)` (from the `cmath` header) to support non-integer powers.
- Extend the grammar to use the `!` (exclamation point) for computing factorials, and add a new unary-operator subclass to compute them. Note that the factorial operator should *follow* an expression, not precede it. For example, you would type `x!` or `(3 * x)!`. If the factorial operator is applied to a non-integer value, throw an exception (of type `std::runtime_error`) describing the problem.

- Extend the grammar and the program to support functions. Make sure to provide built-in functions such as `sin(x)`, `log(x)`, `atan2(y, x)`, etc.
- Further extend the function support to provide user-defined functions. This shouldn't be hard if you added built-in functions to the program in an intelligent way. A user should be able to type, for example:

```
f(x, y) = 3 * x - y
f(4, 17)
 = -5
```

- Provide built-in functions that implement branching. Verify that you can now implement recursive computations by providing a user-defined function to calculate factorials, or to calculate the `n`-th Fibonacci number. Detect when infinite recursions take place, and report an error.
