Module 3: Imperative Languages—Control Flow

Case Study

An Expression Interpreter

The case study for this module incorporates two of the language features that we discussed—expressions and assignments. The program interprets fully parenthesized arithmetic expressions that contain either literal values or variables. The variables must then subsequently be assigned values.

The grammar for the language that this interpreter accepts is defined by the following grammar:

```
    <program> → <exp> , <assigns> ;
    <exp> → ( <operand> <op> <operand> )
    <operand> → <literal> | <variable> | <exp>
    <assigns> → <assigns> , <assign> | <assign>
    <assign> → <variable> = <literal>
```

The regular expressions defining the three tokens are the following:

```
    <op>        [+-*/]
    <variable>  [a-zA-Z][a-zA-Z0-9]*
    <literal>   [0-9]+
```

So, if you were to enter the following expression:

```
    (x + (y * 3)), x = 2, y = 6;
```

the interpreter would respond:

```
    Value = 20
```

The interpreter itself is written in C++. The complete program consists of 10 classes. We will present 7 of them. Your instructor may ask you to complete this program, perhaps enhance it, and add some error checking as one of the programming projects.

We begin with the `main` function and one subordinate function, which are contained in `module3.cpp`. The `main` function reads in the program, calls upon the static function `parse` of the `SubExpression` class to parse it, and builds an arithmetic expression tree. It then calls the subordinate function `parseAssignments` to parse the assignments and enter them into the symbol table, and then evaluates the expression and displays the result. That code is shown below:

```
    #include <iostream>
    #include <string>
    #include <vector>
```

```
using namespace std;

#include "expression.h"
#include "subexpression.h"
#include "symboltable.h"
#include "parse.h"

SymbolTable symbolTable;

void parseAssignments();

int main()
{
    Expression* expression;
    char paren, comma;
    cout << "Enter expression: ";
    cin >> paren;
    expression = SubExpression::parse();
    cin >> comma;
    parseAssignments();
    cout << "Value = " << expression->evaluate() << endl;
    return 0;
}

void parseAssignments()
{
    char assignop, delimiter;
    string variable;
    double value;
    do
    {
        variable = parseName();
        cin >> ws >> assignop >> value >> delimiter;
        symbolTable.insert(variable, value);
    }
    while (delimiter == ',');
}
```

The arithmetic expression tree is built using an inheritance hierarchy. At the root of the hierarchy is the abstract class `Expression`. The class definition for `Expression` is contained in the file `expression.h`, shown below:

```
class Expression
{
public:
    virtual double evaluate() = 0;
};
```

This abstract class has two subclasses. The first of these is `SubExpression`, which defines the node of the binary arithmetic expression tree. The class definition for `SubExpression` is contained in the file`subexpression.h`, shown below:

```
class SubExpression: public Expression
{
public:
    SubExpression(Expression* left, Expression* right);
    static Expression* parse();
protected:
    Expression* left;
    Expression* right;
};
```

As is customary in C++, the bodies of the member functions of that class are contained in the file `subexpression.cpp`, shown below:

```
#include <iostream>
using namespace std;

#include "expression.h"
#include "subexpression.h"
#include "operand.h"
#include "plus.h"
#include "minus.h"
#include "times.h"
#include "divide.h"

SubExpression::SubExpression(Expression* left, Expression* right)
{
    this->left = left;
    this->right = right;
}

Expression* SubExpression::parse()
{
    Expression* left;
    Expression* right;
    char operation, paren;

    left = Operand::parse();
    cin >> operation;
    right = Operand::parse();
    cin >> paren;
    switch (operation)
    {
        case '+':
            return new Plus(left, right);
        case '-':
            return new Minus(left, right);
        case '*':
            return new Times(left, right);
        case '/':
            return new Divide(left, right);
    }
    return 0;
}
```

The `SubExpression` class has four subclasses. We show one of them—`Plus`. The class definition for `Plus` is contained in the file `plus.h`, shown below:

```
class Plus: public SubExpression
```

```
{
public:
    Plus(Expression* left, Expression* right):
        SubExpression(left, right)
    {
    }
    double evaluate()
    {
       return left->evaluate() + right->evaluate();
    }
};
```

Because the bodies of both member functions are inline, no corresponding `.cpp` file is required.

The other subclass of `Expression` is `Operand`, which defines the leaf nodes of the arithmetic expression tree. The class definition for `Operand` is contained in the file `operand.h`, shown below:

```
class Operand: public Expression
{
public:
    static Expression* parse();
};
```

The body of its only member function is contained in `operand.cpp`, shown below:

```
#include <cctype>
#include <iostream>
#include <list>
#include <string>

using namespace std;

#include "expression.h"
#include "subexpression.h"
#include "operand.h"
#include "variable.h"
#include "literal.h"
#include "parse.h"

Expression* Operand::parse()
{
    char paren;
    double value;

    cin >> ws;
    if (isdigit(cin.peek()))
    {
        cin >> value;
        Expression* literal = new Literal(value);
        return literal;
    }
    if (cin.peek() == '(')
    {
        cin >> paren;
        return SubExpression::parse();
    }
    else
        return new Variable(parseName());
    return 0;
}
```

The `Operand` class has two subclasses. The first is `Variable`, which defines leaf nodes of the tree that contain variables. The class definition for `Variable` is contained in the file `variable.h`, shown below:

```
    class Variable: public Operand
    {
    public:
        Variable(string name)
        {
            this->name = name;
        }
        double Variable::evaluate();
    private:
        string name;
    };
```

The body of its member function `evaluate` is contained in `variable.cpp`, shown below:

```
    #include <strstream>
    #include <vector>
    using namespace std;

    #include "expression.h"
    #include "operand.h"
    #include "variable.h"
    #include "symboltable.h"

    extern SymbolTable symbolTable;

    double Variable::evaluate()
    {
        return symbolTable.lookUp(name);
    }
```

The other subclass of `Operand` is `Literal`, which defines leaf nodes of the tree that contain literal values. The class definition for `Literal` is contained in the file `literal.h`, shown below:

```
    class Literal: public Operand
    {
    public:
        Literal(int value)
        {
            this->value = value;
        }
        double evaluate()
        {
            return value;
        }
    private:
        int value;
    };
```

This interpreter uses a symbol table that is implemented with an unsorted list defined by the class `SymbolTable`. Its class definition is contained in the file `symboltable.h`, shown below:

```
    class SymbolTable
    {
    public:
        SymbolTable() {}
        void insert(string variable, double value);
        double lookUp(string variable) const;
    private:
        struct Symbol
        {
            Symbol(string variable, double value)
            {
                this->variable = variable;
                this->value = value;
            }
            string variable;
            double value;
        };
        vector <Symbol> elements;
    };
```

The bodies of its member functions are in the file `symboltable.cpp`, shown below:

```
    #include <string>
    #include <vector>
    using namespace std;

    #include "symboltable.h"


    void SymbolTable::insert(string variable, double value)
    {
        const Symbol& symbol = Symbol(variable, value);
        elements.push_back(symbol);
    }

    double SymbolTable::lookUp(string variable) const
    {
        for (int i = 0; i < elements.size(); i++)
            if (elements[i].variable == variable)
                 return elements[i].value;
        return -1;
    }
```

Finally, one utility function, `parseName`, is needed by this program. Its function prototype is the file `parse.h`, shown below:

```
string parseName();
```

Its body is in `parse.cpp`, shown below:

```
#include <cctype>
#include <iostream>
#include <string>
using namespace std;

#include "parse.h"

string parseName()
{
    char alnum;
    string name = "";

    cin >> ws;
    while (isalnum(cin.peek()))
    {
        cin >> alnum;
        name += alnum;
    }
    return name;
}
```