Composite & Strategy Patterns

Author: Jimmy Tran, Brian Crites

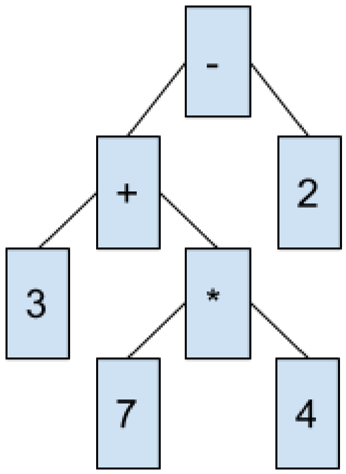
You must work in a group of two for this lab

Composite Pattern

You will start this lab by creating a composite pattern for representing an expression tree. An expression tree is a tree representation of an equation, so the expression

```
3 + 7 * 4 - 2
```

would be represented with the following tree



The first expression to be executed (in this case, 7 * 4) will be at the deepest level. The result of that calculation then becomes an input to the addition (which becomes 3 + 28), and that result becomes an input to the subtraction (31 - 2). The tree is evaluated recursively from the bottom to the top, and is constructed in this particular way because of the order of operations

You will write a composite pattern for representing these expression trees. You are required to use the following base class:

```
class Base {
    public:
        /* Constructors */
        Base() { };
        /* Pure Virtual Functions */
        virtual double evaluate() = 0;
};
```

Note that the main function in the base class is `evaluate()`, which will be used to return the value of the tree.

You will have one type of leaf node which will represent a number (`class Op`), and two types of composite nodes. There will be four types of nodes that have two children, capable of expressing the operations multiply (`class Mult`), divide (`class Div`), add (`class Add`), and subtract (`class Sub`). There also be one type of node that only has one child, which expresses squaring a value (`class Sqr`). Notice that any parentheses that would be in the expression can be implemented in the trees structure rather than explicitly with a node.

You are not required to implement functionality for parsing an expression, but can build the trees by instantiating nodes individually and adding them as children.

Strategy Pattern

Now that you have created your expression tree classes, we will create a strategy pattern for sorting these trees by their `evaluate()` value. You will start this by creating two containers, one that uses a vector to hold your trees (`class VectorContainer`), and one that uses a list (`class ListContainer`). Both of these classes hold the top node pointers of the trees, so the list or vector would be of the type `<Base*>`. You will implement them both using the following Container base class.

```
class Container {
    protected:
        Sort* sort_function;

    public:
         /* Constructors */
        Container() : sort_function(NULL){};
        Container(Sort* function) : sort_function(function) {};

        /* Non Virtual Functions */
        void set_sort_function(Sort* sort_function); // set the type of
sorting algorithm

        /* Pure Virtual Functions */
        // push the top pointer of the tree into container
        virtual void add_element(Base* element) = 0;
        // iterate through trees and output values
        virtual void print() = 0;
        // calls on the previously set sorting-algorithm. Checks if
      sort_function is not null, throw exception if otherwise
        virtual void sort() = 0;

        /* Essentially the only functions needed to sort */
        //switch tree locations
        virtual void swap(int i, int j) = 0;
        // get top ptr of tree at index i
        virtual Base* at(int i) = 0;
        // return container size
        virtual int size() = 0;
};
```

Notice that our Container abstract base class does not have any actual STL containers because it leaves it to derived classes to implement its own version of containers.

You must use the homogeneous interface above for your sort functions, and you are only allowed to manipulate the containers through this interface, not directly. This will allow you to extend and change the underlying functionality without having to change anything that interfaces with it.

You will also implement two sort functions for sorting these containers, one that uses the selection sort algorithm and one that uses the bubble sort algorithm (you may directly adapt this code when writing your sort functions). They should both inherit from the base class below

```
class Sort {
      public:
           /* Constructors */
            Sort();

           /* Pure Virtual Functions */
            virtual void sort(Container* container) = 0;
};
```

Notice that your container class requires a reference to your sorting class (and vice-versa). This will require you to use forward declarations and object file compilation. The easiest way to compile with object files (without learning more specific g++ commands) is to use the makefile I have provided you with. You can get a good overview of makefiles here (and you may want to learn more about them on your own). You can get a good overview of how forward declarations work here.

You should test all the combinations of containers and sorting objects together. The following code serves as a basic test you should use to extend your own test cases.

```
#include <iostream>
// #include necessary classes

using namespace std;

int main() {
    Op* op7 = new Op(7);
    Op* op4 = new Op(4);
    Op* op3 = new Op(3);
    Op* op2 = new Op(2);
    Mult* A = new Mult(op7, op4);
    Add* B = new Add(op3, A);
    Sub* C = new Sub(B, op2);
    Sqr* D = new Sqr(C);

    VectorContainer* container = new VectorContainer();
    container->add_element(A);
    container->add_element(B);
    container->add_element(C);
    container->add_element(D);

    cout << "Container Before Sort: " << endl;
    container->print();

    cout << "Container After Sort: " << endl;
    container->set_sort_function(new SelectionSort());
    container->sort();
    container->print();
};
```

Submission

To receive credit for this lab you must show an example program to your TA that demonstrates the full functionality of these two patterns, and must explain to your TA the structure of both your composite pattern and strategy pattern.

Once you have demo'd to your TA, submit a tarball of your code to the lab submission on iLearn.