COMP 3522 Lab 5

Object Oriented Programming in C++

Due Friday 11:59pm

1 Instructions

This week you will continue to explore C++'s flavor of OOP (specifically **inheritance** and **abstraction**) by implementing a command line reverse Polish notation calculator.

We've already talked about binary infix operators in lecture:

```
2 + 2 -> 4
4 - 2 -> 2
5 / 3 -> 1 (assuming we are using ints)
```

Reverse Polish notation (RPN) is a mathematical notation where operators follow their operands. Instead of using a binary infix operator, RPN uses a binary postfix operator:

```
2 2 + -> 4
4 2 - -> 2
5 3 / -> 1
```

Consider our usual notation. When we mix our operations using binary infix operators, we must implement rules of precedence. The use of parentheses can result in dramatically different results:

```
2 - 3 * 4 -> -10
(2 - 3) * 4 -> -4
```

RPN (postfix) notation removes the need for parentheses! RPN's greatest advantage is clear when we consider expressions that contain more than one operand. If we want an operation to take precedence, we just put the operator immediately to the right of the two operands:

```
2 3 4 * - -> -10
2 3 - 4 * -> -4
```

RPN doesn't just eliminate parentheses and keystrokes. It's flexible. If we were to employ a stack, and push operands onto the stack until we reached an operator, we could pop operands off the stack, use them, and push results back onto the stack as we go, letting us calculate complex partial results without having to save them in multiple locations.

That's exactly what we're going to do. We'll use a bit of inheritance and ab- straction to create a hierarchy of operations. Then we'll build an RPN calculator class that contains just a few methods.

We're also going to use the C++ stack. The C++ std::stack is defined in the header file <stack>. It is a container adapter class provided in the C++ standard library. It resides, like all standard library components, in the std namespace. The C++ stack is described here `http://www.cplusplus.com/reference/stack/stack/`. You will use the push, pop, and top methods.

2 Set up your lab

Start by creating a new project:

1. Clone your repo using github classroom: https://classroom.github.com/a/Yp8RBUP2
2. Fill out your **name** and **student number** at the top of **main.cpp**
3. Ensure you commit and push your work frequently. You will not earn full marks if you don't

3 Requirements

Please complete the following:

1. You must create an interface called operation. Recall that there is not really such a thing as an interface in C++ like Java. We must use an abstract class, i.e., a class with one or more purely virtual functions. In C++, we say an abstract class is an interface if it contains nothing but purely virtual methods:
   (a) Create a new header file called operation.hpp. operation does not need a cpp file.
   (b) operation's members are all public.
   (c) Declare a purely virtual member function called get_code that accepts no parameter and returns a char.
   (d) Declare a purely virtual member function called perform that accepts two ints and returns an int.
   (e) Declare a virtual destructor called ~operation, and provide an inline empty implementation.
2. You must create an abstract class called abstract_operation that implements our operation interface:
   (a) Create a new header file called abstract_operation.hpp. The abstract_operation class can go in this file, it does not need a cpp file.
   (b) abstract_operation contains a single private data member called operation_type which is a char.
   (c) Declare and define a one-parameter constructor which accepts a char representing a mathematical operation and assigns it to operation type.
   (d) Implement the get_code member function so that it returns operation_type.
   (e) Declare a virtual destructor called ~abstract_operation, and provide an inline empty implementation.
3. You must create four classes that extend abstract_operation:
   (a) Each class must be implemented in its own header file. There is no need for a cpp file.
   (b) Define an addition_operation class, a subtraction_operation class, a multiplication_operation class, and a division_operation class.
   (c) Provide each class with a public static constant char called OPERATION_CODE where OPERATION is the mathematical operation, and assign the correct value, i.e., +, -, *, or /. For example, the addition_operation class will contain a constant called ADDITION_CODE and with the value '+'.
   (d) Implement a zero-parameter constructor which passes the value stored in the static constant to the base class' one-parameter constructor.

(e) Implement the perform member function (this should be a single line) by performing the operation using the correct operator and returning the result.

(f) Declare a virtual destructor, and provide an inline empty implementation.

4. You must create a class called rpn_calculator:

(a) Declare the class in a header file called rpn_calculator.

(b) Remember to include the other header files at the top of this class (which can you omit?)

(c) Declare two private data members

i. an int called result

ii. an std::stack that contains ints. You will declare a variable of type std::stack<int>.

(d) Define a private member function called operation_type. This member function should accept an int (or char) called operation, and return a pointer to an operation. Inside the function, use a switch statement to examine the parameter. If it is a plus sign, return a pointer to a new addition_operation. If the operation is a minus sign, return a pointer to a new subtraction_operation. And so on. **Use the public constants from the respective operation implementations instead of literals in this function.**

i. HINT: Remember to delete dynamic memory when creating dynamic memory with the **new** operations,

(e) Define a private member function called perform. This member function should accept a parameter which is a pointer to an operation. The member function should return void. It must pop the top two numbers from the stack, apply the operation (use polymorphism, and make sure the operands are in the correct order!) and then push the result back on the stack.

(f) Here's the fun part. Define and implement a public member function called process_formula:

i. This function must accept a string called formula as its parameter, and return an int which is the final solution to the equation in the formula passed.

ii. This function must read the formula from left to right. Integers in the formula must be pushed to the stack. When an operation is encountered, the top two operands must be removed from the stack and used with the operator. The result must be pushed back on the stack. When we get to the end of the formula, we can return the final value.

iii. An interesting and straight-forward algorithm follows. Create a local istringstream object called iss, passing the formula string to the iss constructor. 'While' we can loop and use the extraction operator on iss to extract a second string called operand,

take operand and pass it as the parameter to a second nested istringstream object called iss2. Try to extract an integer from iss2. If we can, push the integer on the stack. If not, we must have hit an operation, so we can simply extract the first char from the second string (operand) and use it as our operation. This code snippet may help you get started:

```
perform(operation_type(operand[0]));
//the above code will likely lead to a
  memory leak, examine how to fix it
```

5. Here is the main method I wrote to help you get started. Put this in a file called main.cpp:

```
int main()
{
    std::cout << "Enter your formula:\n";
    std::string formula;
    std::getline(std::cin, formula);
    std::cout << "You entered " << formula <<
    std::endl;
    RPNCalculator calculator;
    int result =
    calculator.process_formula(formula);
    std::cout << "The result is:\n";
    std::cout << result <<
    std::endl;
    return 0;
}
```

Test your program with the following input: 1 2 3 4 * 5 - 6 / + 7 - *

The expected output is: -4

Another test case: 6 4 8 2 - + *

The expected output is: 60

Good luck, and have fun!