NPRG041 – 2024/25 Winter – Labs MS – **Small Assignment C05**

Arithmetic Expressions I

The topic of the next pair of assignments is arithmetic expressions and working with them. In this task, we first focus on the ability of representing such expressions, their printing in the form of postfix and infix notations, and also their evaluation. In the following task, we will then deal with their parsing from input strings written in infix notation.

In particular, we assume simple arithmetic expressions over integer values. More specifically, the only input values (or simple operands, i.e., factors) can be non-negative integers (therefore natural numbers or zero, without specifying the unary sign). When evaluating, however, intermediate results of individual operations can be negative, but always integer.

As for the supported operations, we only consider addition, subtraction, multiplication, and division. We will use the following operator symbols for them: `+`, `-`, `*`, and `/`. According to the usual convention, we perceive all the mentioned operations as binary left-associative operations. This means that, for example, an expression `3+5+7-9` is understood implicitly parenthesized as `((3+5)+7)-9`.

The use of auxiliary parentheses `(` and `)` may be necessary to enforce a specifically intended evaluation order of individual involved operations if the default operator priorities do not suit us. This happens, e.g., with an expression `3*(7-4)`. In this sense, we assume the usual behavior again, i.e., that multiplication and division operations have a higher priority (precedence) than addition and subtraction operations. It is essential for us to realize that, within the scope of this task, we will not consider the parentheses themselves in any way when representing the expressions, since it will be enough for us just to correctly capture the internal tree structure of the expressions. Potential parenthesizing will then result from it implicitly.

For the purpose of the mentioned representation, we will use the concept of inheritance and propose the following hierarchy of classes, with which we will be able to represent individual tree nodes, from the root node, through internal nodes, to leaf nodes. Specifically, we will assume an abstract class `Node` as a common ancestor for all nodes, a derived final class `NumberNode` for numeric factors (leaf operands), an abstract derived class `OperationNode` for a shared representation of all our binary operations, and final classes derived from it for individual operations, i.e., `AdditionNode`, `SubtractionNode`, `MultiplicationNode`, and `DivisionNode` for addition, subtraction, multiplication, and division, respectively.

The class for leaf numeric values is expected to have a parameterized constructor `NumberNode(int number)`. As for the individual operation nodes, it will then be `OperationNode(Node* left, Node* right)`. Via both the parameters `left` and `right`, we obtain valid pointers to the already correctly created nodes representing subtrees of the left and right operands of a given operation. We expect that both of these nodes were created using dynamic allocation, hence we store the provided pointers and become the owners of such nodes. This means that we take full responsibility for their subsequent deallocation. In other words, and in addition to the enumerated constructors, it is also necessary to correctly implement the corresponding destructors.

Names of the listed classes of nodes and the interfaces of their constructors must be respected, all other methods can be designed as you wish. Following other requirements, and for practical reasons, it might be useful for us to propose an enumeration class with two different values (e.g., `NUMBER` and `OPERATION`), via which we would be able to easily distinguish between leaf numeric nodes and nodes of operations.

When designing the necessary methods, we will make full use of the inheritance mechanisms, and so primarily virtual functions. The goal is to achieve optimal implementation in the sense of not repeating similar or even the same fragments of code. In the same way, we will try to store only genuinely necessary information within the individual nodes, not anything that would be easily derivable.

Having any complicated expression, it would, of course, be sufficient to hold a pointer to its root node for the purpose of its representation. From the practical point of view, however, it would be unfortunate if we were to force the users to work with the expressions as a whole just in this form of traditional and not entirely friendly pointers. Especially since we are working with dynamic allocation. In other words, we will deem our tree structure only as internal and so we want the end users to be shielded from it as much as possible, just as we want to offer them an interface that will be sufficiently elegant.

We will thus propose an `Expression` class for this reason. Its goal will be the encapsulation of a single particular arithmetic expression. Only one constructor will be offered, namely `Expression(Node*` root). We once again assume that we will be provided with a valid pointer to the root node of a correct tree constructed using dynamic allocation. This means we take responsibility for its future deallocation (including its entire subtree, if any), and so we must also provide a corresponding destructor.

When it comes to the expression class, we will also offer a member function `void print_postfix(std::ostream& stream = std::cout) const`, through which we will be able to print the serialization of our expression in postfix (reverse Polish) notation to the provided output stream. Since several numbers may occur right after each other in this notation, we will always consistently print one separating space between the numbers and operators. Let us have a look at a particular example to illustrate the intended behavior: for an expression `1*2+3*(4+5)-6` in the infix notation, we will print `1 2 * 3 4 5 + * + 6 -` in the postfix notation.

Analogously, we will also offer a method `void print_infix(std::ostream& stream = std::cout) const` for printing the expression in the infix notation. We will not include any spaces around the numbers, operators, or parentheses this time, because that is not necessary. As for parentheses, they are necessary in this case, unlike in the postfix notation. However, we will print them only in situations where they are necessary in the sense of faithfully preserving the intended internal structure of the expression and its subsequent correct evaluation.

This means that we will use the following three rules when serializing the nodes of our operations: 1) operands with leaf numbers will never be wrapped by parentheses, 2) left operation operand will be wrapped by parentheses if and only if it is an operation with a lower precedence than ours, and 3) right operation operand will always be wrapped by parentheses unless it has a higher precedence than we have. For example, having an input expression `(7+(9-(3*1))/3)-(5-1)`, we would serialize it as `7+(9-3*1)/3-(5-1)`. Exactly for the purpose of distinguishing the individual situations, the ability to easily separate nodes of numbers from nodes of operations will come in handy at this point. We will also need retyping, using, e.g., the traditional approach `(OperationNode*)ptr`, where `ptr` is a pointer to a general `Node`.

The very last method of the `Expression` class will be `int evaluate() const`. Its goal is to evaluate the entire arithmetic expression, and so calculate its resulting value. This will be passed via the return value. Let us recall that we perceive all the operations (especially division) as integer operations. To achieve a fully correct solution, we should probably detect possible value overflows, but that would be technically tedious, and so we will not treat such situations. Within this task, we will even not yet deal with potential division by zero.

Submit all source files you created (`*.cpp` and `*.h`) except for the main file `Main.cpp` with the `main` function. This will again be a part of the already prepared test. Assume that it contains only one `#include` directive for a header file `Expression.h` (through which everything needed must be directly or indirectly available).

The main goal of the assignment is to demonstrate the ability to design a more complex hierarchy of classes with inheritance and virtual functions, including the work with final and abstract classes, purely virtual methods, constructors, and destructors. Another main goal is to work with dynamic allocation for individually allocated objects with a non-trivial life cycle. It is also about working with pointers, enumeration classes, or applying the general idea of a wrapper class to encapsulate other structures in order to offer a more friendly interface and conceal internal, technical, and other implementation details.

Again, the usual requirements for our assignments are expected to be adhered to. As for specific requirements, we can hard-wire all our operations, their operators, and precedence in the code, even in multiple places. This is because our code is tied only to our arithmetic expressions, and so will not be directly applicable to other situations (types of expressions) anyway. However, we still need to define constants for the operator, parentheses, and space symbols using global named constants. This also applies to constants (or even better values of some other enumeration class) of individual precedence levels, if we potentially decide to use them. Such an approach is actually recommended. Just make sure that number nodes do not have any precedence, since it is only relevant regarding operations.

Within the nodes of our tree, we will store the necessary information and offer the corresponding functionality exactly in those nodes to which they logically relate. This especially applies to the abstract node `OperationNode`, which is meant to encapsulate everything that is common to all of our particular operations under consideration. On the contrary, solving the evaluation of individual operations at the level of this node using the `switch` construct (or otherwise analogously) would be inappropriate and against the inheritance principles.

Printing the expressions in postfix or infix notations can be achieved by implementing the corresponding tree traversal, i.e., post-order or in-order depth-first search. However, this tree traversal itself needs to be performed implicitly, recursively, and directly by the individual nodes as such, not by some algorithm that would work with the entire expression tree externally and as a whole. In our case, this would again be against the purpose of inheritance and virtual methods. The same analogously applies to the evaluation of expressions, too.

Although all these three functions have a prescribed interface only at the level of the `Expression` class and not the internal tree nodes, they should avoid excessive number of parameters. None is needed within the evaluation function, and only a reference to the stream and nothing else is enough for the printing functions. Since we further only assumed valid trees, do not test, e.g., invalid pointers to subtrees or other error situations in these functions, they simply cannot occur and such conditions would only slow down the code. Regardless of the fact that it would not actually be obvious what should happen in such situations.

Avoid repeated writing of identical-looking constructors for particular operation nodes, use the `using` construct. It is necessary to work with the concept of virtual destructors throughout the entire hierarchy, we would otherwise not be able to achieve correct behavior. As for the deallocation of our dynamically created nodes, it must always be carried out in the places that it logically belongs to, i.e., in the already discussed destructors, in no case elsewhere.

Just to be sure, let us add that the deallocation of each node must be called exactly once, hence we must not forget it, just as we must not call it multiple times by mistake. With respect to good programming practice, this means that we will follow the idea where we first verify that a given pointer is not zero before proceeding to its actual deallocation, and, on the contrary, we deliberately zero it out after the performed deallocation.

This is a preventive measure to help us defend against our own mistakes in the code, since any subsequent improper attempt to use such an already invalidated pointer will cause the program to crash immediately. When working with null pointers as such, we will only use a special literal `nullptr`, not the older approaches based on the `NULL` macro, let alone numeric `0`.

Just for the avoidance of any doubts, let us add that considering the goals of the task, it is really necessary to use traditional C-style pointers to nodes, not smart, i.e., most likely unique smart `std::unique_ptr` pointers.