# Lab #01: Large Integer Arithmetic Expression

Tran Van Tan Khoi
ID: 24120010

May 13, 2025

**Attributions and References**

| Section | Percentage Understood | Content Understood | Percentage Referenced | Content Referenced | Source |
|---|---|---|---|---|---|
| Custom 'int' type | 80% | Constructor, operator overloading, data stream | 10% | Karatsuba algorithm | Wikipedia |
| Parsing expression | 100% | Parsing simple expression | 0% | | |
| Convert to postfix | 95% | Using stack for conversion, precendence, parenthesis grouping | 10% | Algorithm outline | TakeUForward |
| Postfix evaluation | 100% | Algorithm using stack | 0% | | |
| Exceptions handling | 90% | C++ built-in exception handling | 10% | Exception handling syntax | cppreference |
| Command-line Arguments | 0% | argc, argv | 0% | | |

## 1 Introduction

The purpose of this Lab is to show an elementary but nonetheless crucial application of one of the most commonly used data structure, the stack, in evaluating basic arithmetic expressions. Not only will we dive into the rabbit hole of resolving many types of expressions, but also show the pitfalls of how we typically write expressions in infix notation. This lab leverages the previously implemented 'stack' class in C++ and the newly written 'integer' class that handles whole numbers with nearly unlimited digits, similar to the 'int' data type in Python.

## 2 The Problem

**Statement**

Multiple arithmetic expressions are given in infix notation on multiple lines. Each expression consists of operands which are signed integers up 100 digits long, operators '+', '-', '*', and '/' (integer division), and parenthesis to group operations.
Write a program that can evaluate these expressions and output the results on multiple lines. Display appropriate error messages for zero-division or malformed expression.

## 3 The Approach

In a nutshell, there are 3 steps to evaluate expressions:

1. *Parse* the expression into symbols (operands, operators, and parenthesis)
2. *Convert* the parsed list of symbols into *postfix notation*.
3. *Evaluate* the given expression in postfix notation.

### Parser Class

To design a modular system, we combine the expression parser and postfix notation converter into a single component called *Expression Parser*, implemented as a class with two methods, parse_expression(expression) and convert_to_postfix(parsed_expression).

parse_expression(expression) takes an std :: string object which represents the original expression and seperate it into symbols, skipping any spaces.

convert_to_postfix(parsed_expression), as the name suggests, convert the list of symbols in infix ordering to postfix ordering. Create an empty stack. Traversing the list of symbols from left to right, each time we encounter a closing parenthesis, pop the top of the stack until we find the opening. If we encounter an operator instead, pop off the top of the stack until we find an operator with lower precendence or the stack is empty, then push the operator to the top of the stack. In other cases, push the symbol to the top of the stack. After traversing the list, simply pop off each item in the stack to receive the expression in postfix ordering.

### Evaluating Postfix Expression

Evaluating expressions in postfix notation is very simple by popping off the top two symbols in the stack each time we encounter an operator, calculating the result, then putting it back in. The final symbol in the stack will be the result of the expression.

## 4 The Caveats

### Large Integers

The operands can have up too 100 digits, exceeding the upper bound of the 'long long' data type of $2^{63} - 1$. That's why a custom type is needed. For purposes of this lab only, we need to maintain a vector of digits ordered from the least significant digit to the most significant digit and a seperate variable telling the sign.

### Handling Exceptions

Within the constraints of this lab, we also need to handle cases where division by 0 happens, or the given expression's syntax isn't correct. We can raise exceptions using C++ exceptions handling system when such cases arise.

## 5 Conclusion

The source code is available at this Github repo.