CPT-281 Team Project 2A: Infix Expression Parser

Contributors: Athul Jaishankar, Timothy Huffman, Kathleen Dunn

Project Summary:

This project is an Infix expression parser system that helps parse an infix expression that supports arithmetic and logical operators with specified precedencies. The system utilizes stacks for efficient management of expression data.

Technical Requirements:

▪ The Infix expression parser system will support:

Operator	Precedence	Example
1) Power (‘^’)	7	2 ^ 8
2) Arithmetic (‘*’, ‘/’, ‘%’)	6	6 * 2
3) Arithmetic (‘+’, ‘-’)	5	6 - 2
4) Comparison (‘>’, ‘>=’, ‘<’, ‘<=’)	4	6 > 5
5) Equality Comparison (‘==’, ‘!=’)	3	6 != 5
6) Logical And (‘&&’)	2	6 > 5 && 4 > 5
7) Logical Or (‘||’)	1	1 || 0

▪ The infix expression parser is flexible with the given expressions. The user don’t need to worry about writing the spaces between operands and operators

▪ The file that keeps track of the infix expression is a plain text file. An original file input format is made based on this example:

```
((2 + 3) * 4) - (5 * (6 - 7))
(1 | | (0 && 1)) && (1^ ( 1 && 0 ))
(( 2 *3) ^ 2 ) + ( 4* 5) % 3
```

In the example above, each line stores a valid infix expression with appropriate suitable operators and operands.

System Design:

The Expression parser was made in order to efficiently convert infix expressions and evaluate postfix expressions. This system uses three classes which are Expression_Parser, Evaluate_Postfix, and Convert_to_Postfix.

- **Expression parser:**

  The Expression parser class is used in order to convert and evaluate an infix string into a postfix string returning the result by using the parse_and_evaluate() function. The function precedence() serves as the purpose of determining the precedence level of operators, which is crucial for correctly parsing and evaluating mathematical or logical expressions. The power_function() which will be called whenever the power operator is parsed.

- **Evaluate postfix class:**

  In the Evaluate postfix class is where the postfix expression will be evaluated . The prominent function in this class is the postfix_evaluator(), with the use of stacks, an input file is read containing the postfix expression. Digits are then stored in the operand stack while processing tokens appending and popping when needed.

- **Convert to postfix class:**

  In order to evaluate an infix expression, it first needs to be converted to a postfix expression. The Convert to postfix class converts an infix expression into a postfix expression with the use of its only class-member function, infix_to_postfix(), using a stack to iterate through the infix string appending and popping when needed.

Data Structures:

- **Stack:**

  The system developed uses the data structure stacks , the stacks data structure can be seen in the Evaluate_Postfix & Convert_to_Postfix classes within the postfix_evaluator() & infix_to_postfix() function.

  - **Postfix_evaluator():**

    The postfix_evaluator() function utilizes a stack of integers to store operands during evaluation of a postfix expression. It iterates over each token, pushing digits onto the stack and performing operations when encountering operators. It supports basic arithmetic operations as well as logical and comparison operators.

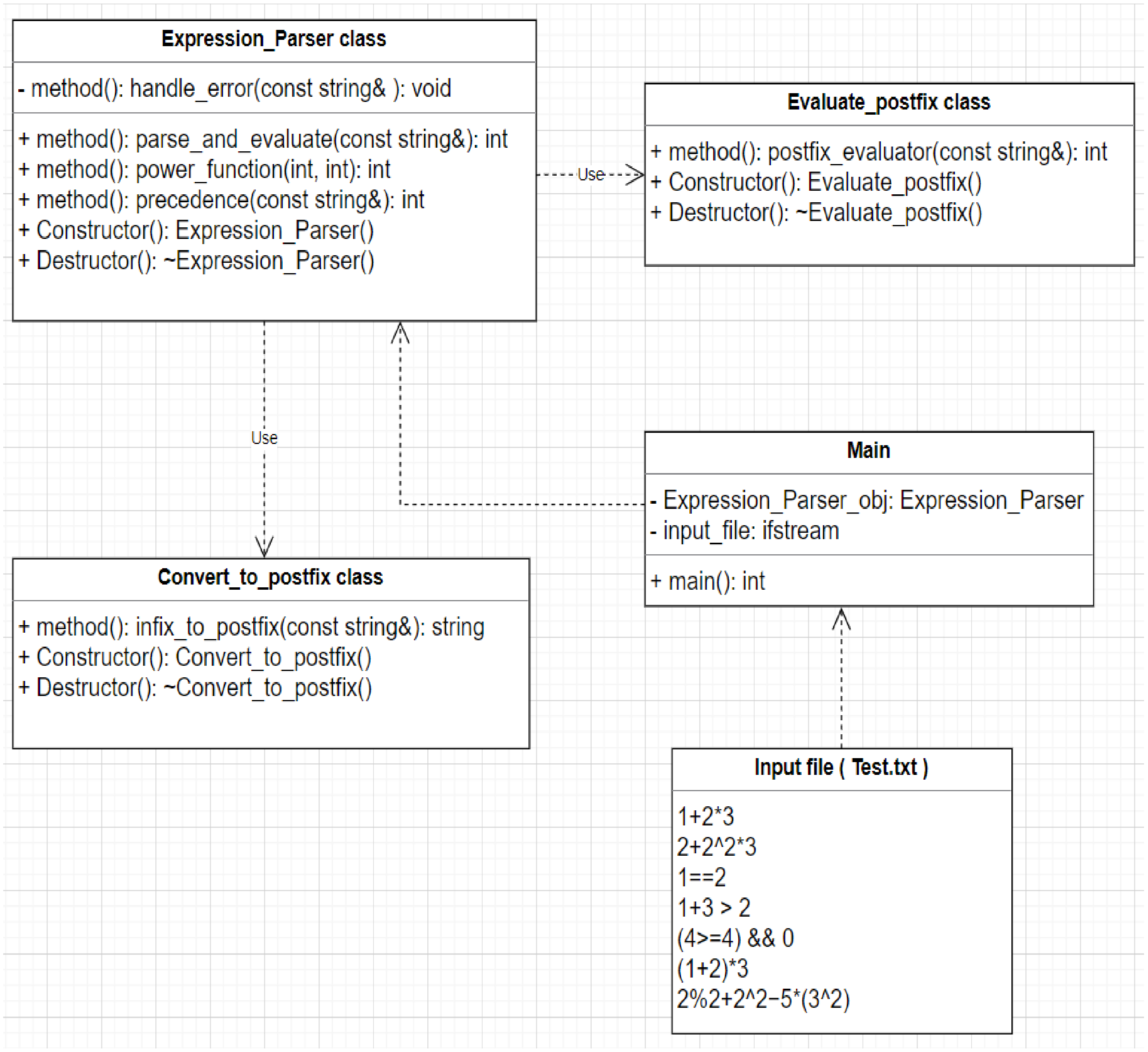
  - **Infix_to_postfix():**

    The infix_to_postfix() function uses a stack of strings to store operands while converting an infix expression to a postfix expression. The infix string is parsed pushing operations onto the stack based on the rules given and popping the elements when needed. After processing, remaining elements in the stack are appended to an output string.

- **String:**

  The String data type was used to store the infix expressions, when converting to postfix expressions.

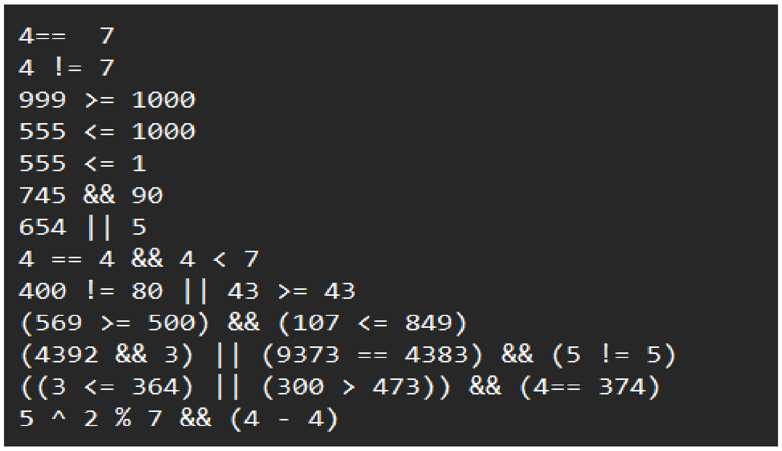
UML:


Expression_Parser class
- method(): handle_error(const string&): void
+ method(): parse_and_evaluate(const string&): int
+ method(): power_function(int, int): int
+ method(): precedence(const string&): int
+ Constructor(): Expression_Parser()
+ Destructor(): ~Expression_Parser()
Use
Evaluate_postfix class
+ method(): postfix_evaluator(const string&): int
+ Constructor(): Evaluate_postfix()
+ Destructor(): ~Evaluate_postfix()
Use
Main
- Expression_Parser_obj: Expression_Parser
- input_file: ifstream
+ main(): int
Convert_to_postfix class
+ method(): infix_to_postfix(const string&): string
+ Constructor(): Convert_to_postfix()
+ Destructor(): ~Convert_to_postfix()
Input file (Test.txt)
1+2*3
2+2^2*3
1==2
1+3 > 2
(4>=4) && 0
(1+2)*3
2%2+2^2−5*(3^2)

Test Cases:

Test Case #1:

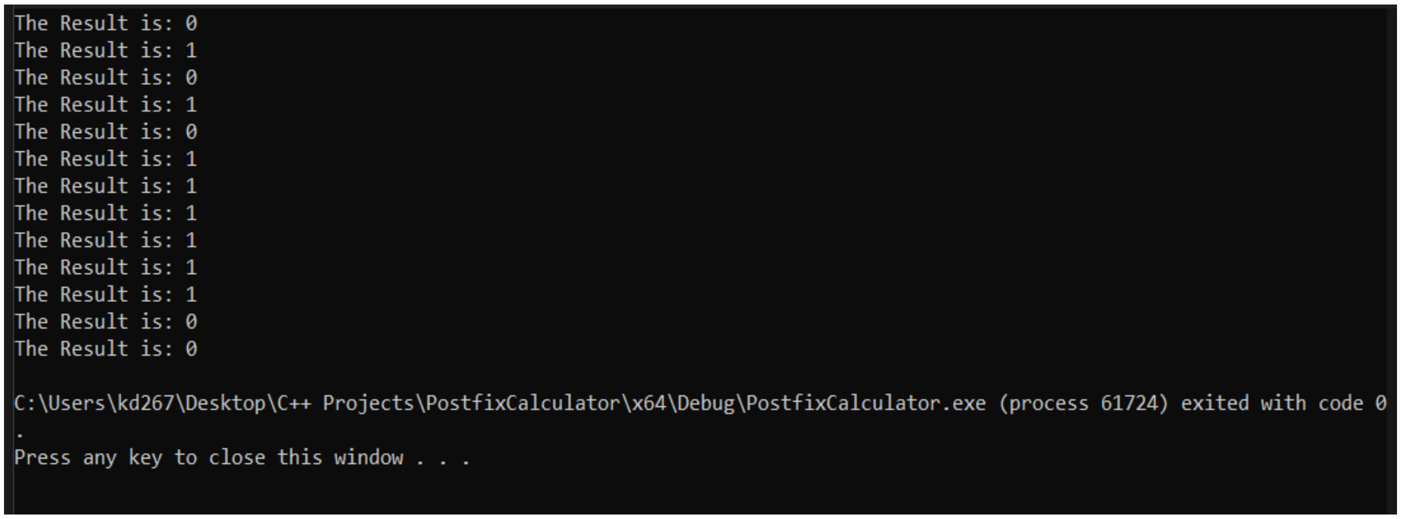
The first input file is shown below:



```
4==  7
4 != 7
999 >= 1000
555 <= 1000
555 <= 1
745 && 90
654 || 5
4 == 4 && 4 < 7
400 != 80 || 43 >= 43
(569 >= 500) && (107 <= 849)
(4392 && 3) || (9373 == 4383) && (5 != 5)
((3 <= 364) || (300 > 473)) && (4== 374)
5 ^ 2 % 7 && (4 - 4)
```

The expected output is a list of the evaluation results of these infix expressions. For the comparisons and logical and/or, the evaluation result is 1 for true, and 0 for false. In the first example of the expression (4 == 7), the expected result in the output should be 0 because 4 is not equal to 7, making it false (0).

The output from the first test case is shown below:

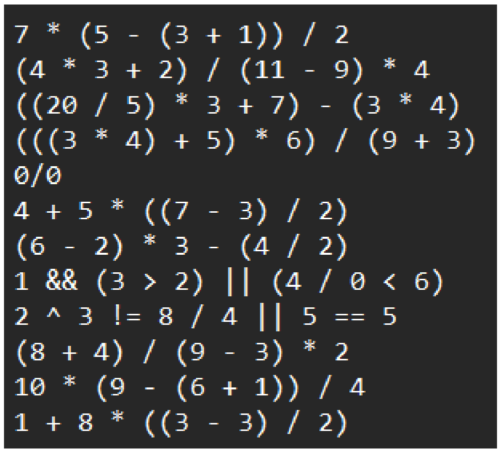


```
The Result is: 0
The Result is: 1
The Result is: 0
The Result is: 1
The Result is: 0
The Result is: 1
The Result is: 1
The Result is: 1
The Result is: 1
The Result is: 1
The Result is: 1
The Result is: 0
The Result is: 0

C:\Users\kd267\Desktop\C++ Projects\PostfixCalculator\x64\Debug\PostfixCalculator.exe (process 61724) exited with code 0
.
Press any key to close this window . . .
```

Test Case #2:

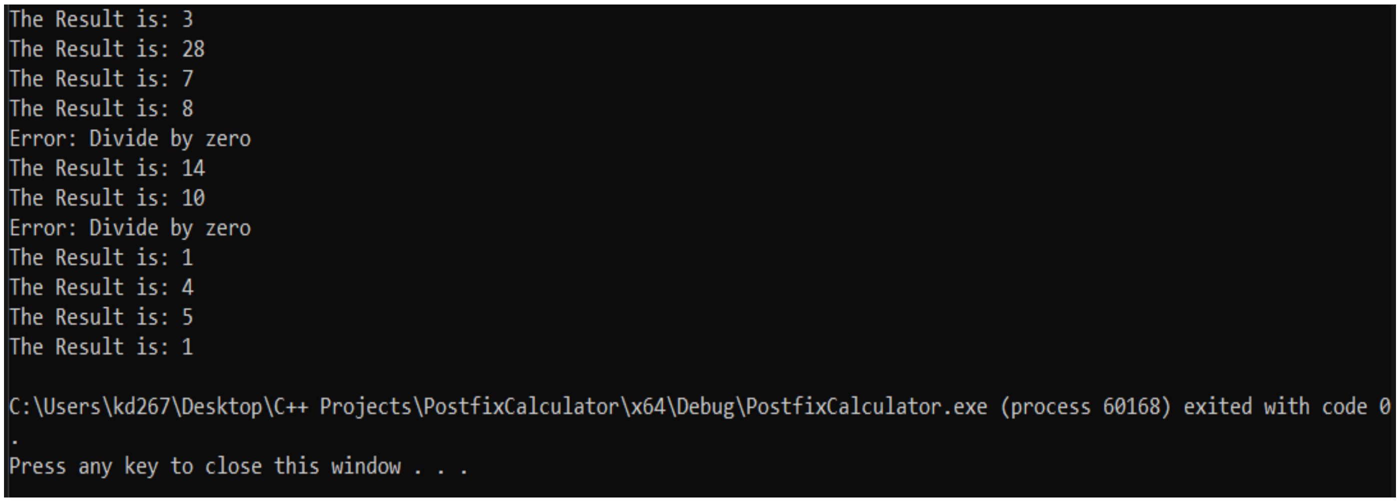
The second input file is shown below:



```
7 * (5 - (3 + 1)) / 2
(4 * 3 + 2) / (11 - 9) * 4
((20 / 5) * 3 + 7) - (3 * 4)
(((3 * 4) + 5) * 6) / (9 + 3)
0/0
4 + 5 * ((7 - 3) / 2)
(6 - 2) * 3 - (4 / 2)
1 && (3 > 2) || (4 / 0 < 6)
2 ^ 3 != 8 / 4 || 5 == 5
(8 + 4) / (9 - 3) * 2
10 * (9 - (6 + 1)) / 4
1 + 8 * ((3 - 3) / 2)
```

The expected output is a list of the evaluation results from the infix expressions above. For the division expressions, it will output an integer result. If the infix expression involves a division by zero, the output writes "Error: Divide by zero" to the console.

The output from the second test case is shown below:



```
The Result is: 3
The Result is: 28
The Result is: 7
The Result is: 8
Error: Divide by zero
The Result is: 14
The Result is: 10
Error: Divide by zero
The Result is: 1
The Result is: 4
The Result is: 5
The Result is: 1

C:\Users\kd267\Desktop\C++ Projects\PostfixCalculator\x64\Debug\PostfixCalculator.exe (process 60168) exited with code 0
.
Press any key to close this window . . .
```

Team Member Contributions:

- **Athul Jaishankar:**

  - **Expression_Parser.h:** Implemented the Expression_Parser class, responsible for parsing infix expressions and evaluating the result. Defined method for parsing and evaluating infix expressions, handling operator precedence. Also, created the handle_error method for handling exceptions.
  - **Expression_Parser.cpp:** Implemented all methods of the Expression_Parser class. Developed algorithms for parsing infix expressions while maintaining their format for efficiency. Ensured error handling by throwing exceptions and handling them.
  - **Convert_to_postfix.h:** Defined a class called Convert_to_postfix with a method infix_to_postfix to convert infix expression to postfix notation.
  - **Convert_to_postfix.cpp:** Implemented the functionalities declared in the header file for the Convert_to_postfix class. This implementation includes the constructor and destructor for the class, as well as the infix_to_postfix method.
  - **Evaluate_postfix.h:** Defined a class called Evaluate_postfix with a method postfix_evaluator to evaluate postfix expression.
  - **Evaluate_postfix.cpp:** Implemented the functionalities specified in the Evaluate_postfix header file. This implementation includes the constructor and destructor for the class, along with the postfix_evaluator method, which evaluates postfix expression to produce numeric outcomes.
  - **Main.cpp:** Implemented the logic to read infix expressions from an input file, parser and evaluate each expression using the Expression_Parser class, and display the result to the console. Integrated file I/O operations for input file handling and collaborated with team members to create and execute test cases. Addressed questions regarding program design and functionality.
  - **Bug Fixes:** Addressed issues related to incorrect output for expressions like ‘2 ^ 3 ^ 2’ by fixing the power function to calculate the exponents correctly. Ensured that the program generates the expected output for all test cases.
  - **Project Management:** Took the initiative to lead the project by designing the overall structure and goals of the infix expression parser system. Scheduled

and organized team meetings to facilitate communication and collaboration among team members, ensuring smooth progress throughout the project.

- **Task Division:** Effectively divided tasks among team members, assigning responsibilities for coding, testing and documentation.
- **Testing:** Collaborated with team members to create test cases covering various expressions and scenarios. Verified the correctness of the program by comparing the actual output with the expected output.
- **Quality Assurance:** Ensured code quality by writing clean, well-commented code with meaningful variable names and function names. Maintained an organized repository structure and adhered to coding standards to facilitate code review and future maintenance. Effectively divided tasks among team members, assigning responsibilities for coding, testing and documentation.

- **Timothy Huffman:**

  - **System Design Explanation:** Provided insights into the overall system design in the project report. Explained the architecture and structure of an infix expression parser system, ensuring clarity and coherence in the documentation.
  - **Data Structures Explanation:** Detailed the role of data structures used in the infix expression parser system. Explained how each data structure contributed to efficient expression parsing and evaluation.
  - **In-Line Comments**: Added in-line comments to the convert_to_postfix and evaluate_postfix files, improving code readability and comprehension of team members.
  - **Meeting Attendance and Questions**: Actively attended team meetings, contributing to discussions on project progress and asking follow-up questions to clarify requirements or resolve issues effectively.

- **Kathleen Dunn:**

  - **Test Cases:** Responsible for creating test cases to validate the correctness of the infix expression parser program. Ensured that the test cases covered various expressions and scenarios, documenting them in the project report for future reference.

- **Program Correctness:** Verified the correctness of the program by executing the test cases and comparing the actual output with the expected output.
- **Future Requirements**: Contributed four ideas for future improvements to the infix expression parser system. These ideas were aimed at enhancing the functionality and usability of the system. Documented these suggestions in the project report to guide future development efforts.
- **Meeting Attendance and Questions**: Actively participated in team meetings, providing valuable input on system design, discussing project progress and asking follow-up questions to clarify requirements or resolve issues effectively.

Future Improvements:

- **Support for more operators and functions:**

  Improve the parser to support additional operators and functions, this could mean implementing more logic in order for the program to receive various new mathematical, logical, or bitwise operators.

- **Memory Management:**

  Make sure memory usage is optimized, especially when dealing with larger expressions. Techniques like object pooling or even smart pointers can efficiently manage memory while avoiding any unnecessary allocations or deallocations.

- **Interactive Mode:**

  Allow users to only enter expressions one at a time for immediate feedback or results that can inform the user step by step what procedures or occurring.

- **Error Recovery:**
  Having an in-depth error recovery mechanism in order to gracefully handle syntax errors in expressions and provide valuable feedback to users without instantly terminating parsing.

- **Database Integration:**
  Integrating a database system into the application to store and manage expression data, user preferences and evaluation results. This integration can provide advantages such as persistence, scalability and data management capabilities.

- **Graphical User Interface (GUI):**
  Developing a GUI application to provide a user-friendly interface for inputting expressions, displaying the results and possibly visualizing the parsing and evaluating process.

- **Support for floating-point numbers:**
  Enhancing the parser to support floating-point numbers would make it more powerful and more useful for a wider range of applications.

- **Variable Support:**
  Extend the parser to support variables. This would allow users to define and use variables within expressions and enhance the utility of the parser.

- **History Session:**
  Implementing a history and session management would allow the users to refer back to previous calculations.