NumberCrafters

C++ Implemented Arithmetic Expression Evaluator Software Requirements Specifications

Version <1.4>

Revision History

Date	Version	Description	Author
<11/10/23>	<1.0>	<Updated SRS Introduction>	<Kyle Spragg>
<14/10/23>	<1.1>	<Added functionality features and use case examples>	<Aidan Ingram>
<15/10/23>	<1.2>	<Added product perspective examples and info>	<Blake Smith>
<15/10/23>	<1.3>	<Added supplementary requirements and appendices>	<Hale Coffman>
<15/10/23>	<1.4>	<Added product functions, user characteristics, restraints, and subsets>	<Landon Bever>

Table of Contents

Software Requirements Specifications

1. Introduction

1.1 Purpose

The purpose of this SRS is to implement a functional calculator that receives input from a user and outputs a correct calculation. The application should only ask the user for what type of operator the user wants to use, the numerical values they are using for the equation, and whether the user wants to exit the calculator or continue doing another calculation. The non-functional requirements want to be able to handle at least one user. Design constraints include scalability, efficiency with large amounts of traffic on the application, and constraints on the functionality of the calculator whether that is through specific variable allocation, complex mathematical problems such as calculus, and other high-level calculations.

1.2 Scope

The scope of this software application primarily focuses on a singular user at a time using the application. This user requires calculations to be made and expects an answer from their calculations. This software application will be implemented using the C++ coding language. This SRS refers to our C++ application that will include all functionality of the software application, i.e., all mathematical operations and formatting to make the application fully functional. The Use-Case model closely related to this project would be a System Use Case Diagram that focuses on the system's interactions with external actors and the different interfaces that could be seen. A few other things that will be influenced by this document include the GitHub Repository and C++ file(s).

1.3 Definitions, Acronyms, and Abbreviations

Number-crafters is the company name used to develop software applications. Besides that, there are no definitions, acronyms, or abbreviations that will not be able to be interpreted by any external actors or people reviewing this software application.

1.4 References

There are no current documents that are referenced within this SRS at the moment. If this changes, the pathways will be updated in this area to determine what they were and where they were referenced from.

1.5 Overview

The rest of the SRS goes over several different areas. To start, from the project perspective, there are interfaces: user, hardware, software, and communication. It also acknowledges parts such as memory constraints and operations. Next, it talks briefly about other items like project functions, user characteristics, constraints, assumptions and dependencies, and requirements subsets. Taking a deeper dive into the requirements, we described the purpose of user-case modeling and further down in the SRS, there are use case specifications. The other focus is on all the software requirements, and other specifications. The functionality of the software application is also included in SRS including its requirements and description of it. Lastly, we write out some details about supplementary requirements, classification of functional requirements, and appendices.

2. Overall Description

2.1 Product perspective

2.1.1 System Interfaces

The software interfaces with the command-line interface for user input and output. It may also have system-level interactions with the operating system for tasks such as I/O operations and error handling.

2.1.2 User Interfaces

The user interface will allow users to enter arithmetic expressions and view the calculated results.

2.1.3 Hardware Interfaces

Since the software operates at the software level, there are no hardware dependencies.

2.1.4 Software Interfaces

The software makes use of C++ and its libraries for specific functionalities like string manipulation.

2.1.5 Communication Interfaces

The program will communicate with the user through a number of ways. The first is how the results of the evaluated expressions will be displayed to the user. Another is how errors will be reported to users, and how a user can recover from these errors.

2.1.6 Memory Constraints

Possible memory constraints include Memory Usage limits, where we consider defining maximum memory usage limits to ensure that the software doesn't consume excessive memory. This can be done by limiting the amount of RAM during execution through efficient code.

2.1.7 Operations

Expression Parsing: The software will be able to parse arithmetic expressions entered by the user.
Operator Precedence: Define operator precedence according to PEMDAS rules and implement logic to evaluate expressions.
Parenthesis Handling: Develop a mechanism to identify and evaluate expressions within parentheses.
Numeric Constants: Recognize and calculate numeric constants in the input, initially assuming integers only
User Interface: Create a user-friendly command-line interface for entering expressions and displaying results.
Error Handling: Implement error handling to manage scenarios such as division by zero or invalid expressions
Operations we will be working on include: addition, subtraction, multiplication, division, exponentiation, modulo, numeric constants.

2.2 Product functions

There are several functions of this implementation of an arithmetic expression evaluator. These functions include:

- Addition - the ability to add two values together.
- Subtraction – the ability to subtract one value from another.
- Multiplication – the ability to multiply two values together.
- Division – the ability to divide one value from another.
- Modulo – the ability to return the remainder of a division problem.
- Exponentiation – the ability to return the result of raising a value to the power of another value.

2.3 User characteristics

Users of this software are people looking to solve an arithmetic problem. Users only need to know what the product functions do and users do not require technical expertise to solve arithmetic expressions. Arithmetic problems are present in many ways in the world, therefore, this software has many different types of users. One group of users may include students. Students would need to use an arithmetic expression evaluator in order to calculate values for a mathematics class. Another group of users could be people in an occupation that requires mathematics often, such as accountants. Accountants, among many other occupations, require arithmetic expression evaluations in order to complete tasks in their line of work.

2.4 Constraints

Multiple constraints are present in this software project. One constraint is the language used to implement the arithmetic expression evaluator. C++ must be used to program the project and no other languages may be used. The project must also be completed before the December deadline. The logical expression calculator can only calculate specific types of logical expressions. The project must follow legal and ethical guidelines regarding data privacy.

2.5 Assumptions and dependencies

One factor that affects the requirements is the availability of the system that the program will run on. If the system becomes unavailable or changes, then this SRS will change accordingly. Another factor that could affect the requirements is the number of operations provided by the expression evaluator. If more operations are added, the project, along with the SRS, will need to change accordingly to accommodate the new operations.

2.6 Requirements subsets

The software has many requirements and some of the requirements need broken down into subset categories. The functional requirements subsets have been previously defined to describe what operations will be included in our calculator software. For user interface requirements, the input methods should be decided, whether it includes a keyboard, mouse, on-screen buttons, voice recognition, etc. The output display will also need to be configured for result displaying, precision, formatting, and error handling. Finally, the subsets of platform and compatibility requirements would include the operating systems the software should run and any hardware prerequisites, such as memory or processor speed.

3. Specific Requirements

3.1 Functionality

3.1.1 Functional Requirement One: Input Handling

The software should take user input using the "cin>>" functionality provided by C++'s standard library to determine what value's need to be parsed for operations. It will take only one input at a time, and upon an incorrect input being entered will handle the error.

3.1.1.1 Functional Requirement Sub One: Error Handling

The software should not parse or attempt to take in any input provided by the user which is not a number, or an impossible operation on two numbers (like divide by zero). If the program detects user input which could cause an error, the "throw" and "catch" statements will be utilized to handle the exception. This will be a broad error handling, covering all possible inputs given to the program.

3.1.2 Functional Requirement Two: PEMDAS Operation

The software should successfully pass the numbers and operations into a stack-based system that handles PEMDAS operations like general mathematics. This specifically means that a given operation will perform it's tasks in the following order: Parenthesis -> Exponentials -> Multiplication -> Division -> Addition -> Subtraction. While some of these could be interchangeable, it is important to make a clear division between those not grouped together (addition and subtraction, or multiplication and division). The stack-based implementation of PEMDAS should handle things in this set order, using the given '(' and ')' symbols.

3.1.3 Functional Requirement Three: Addition Operation

The handling of the addition operation should utilize C++'s native '+' operator to add values in a mathematical way assuming valid input. If input is invalid or the number is not within given constraints, as will be left assumed for the rest of the requirements, the program should not parse or solve the equation given. It should be parsed according to PEMDAS but should be handled as normal addition when parsed into the program.

3.1.4 Functional Requirement Four: Subtraction Operation

The handling of the subtraction operation should utilize C++'s native '-' operator to subtract values in a mathematically valid way and parsed according to PEMDAS. All previous error handling and operations should be covered, but the subtraction should be normal operations and results.

3.1.5 Functional Requirement Five: Multiplication Operation

The multiplication handling of the function will follow all previous PEMDAS and error-handling functions. It will be a stack-allocated decision on when the multiplication will happen, and it will use C++'s native '*' operator to signify it is being multiplied. Alternatively, you should be able to place a number on the immediate left of a parenthetical operation and assume it will be multiplied to the result without a *. Ex: 2(3+1) = 8 is a valid construct. Other than ensuring PEMDAS is followed, it should multiply as it is deemed correct on the stack.

3.1.6 Functional Requirement Six: Division Operation

The division operation is one which could potentially cause more issues than most, and thus must be handled carefully. Dividing by zero is one of the easiest ways to break a program, and thus this operator, which will be C++'s native '/' character, should be ran through our error handling system to ensure that the input given is valid. Assuming valid input, the operation should then proceed as normal and return a decimal to two decimal places, unless given a constraint stating otherwise.

3.1.7 Functional Requirement Seven: Modulo Operation

The modulo operation will return the remainder of a given division problem and will utilize the '%' which is found natively in C++'s mathematical symbol catalogue. A given modulo operation will fall under the division operation in PEMDAS, and thus will receive allocation on the stack similarly as needed, as it is just division and returning a remainder. It should only be able to return values that are not infinite and should have all operations handled with all prior error handling operations.

3.1.8 Functional Requirement Eight: Exponentiation Operation

The exponentiation operation must be able to produce a given number or result multiplied by that number or result to n times. This can be achieved in several ways, but we will accomplish it by creating a function, and adding it as functionality when the system detects a "^" input into the IO system. This functionality should follow the PEMDAS format as all others and all error-handling is still valid. A user cannot input an invalid character to be exponentiated, or be that which is the n, in any given scenario.

3.1.9 Functional Requirement Nine: Numeric Constraints

The functionality of the program should allow it so that the program can handle both integers, and floating-point values. Although initially assuming that the given input will be an integer, we will ensure the program is able to take floating-point values by making all functionality floating-point friendly, so that if required, a user could even input which type of input they were giving, and how they wanted to see the output.

3.2 Use-Case Specifications

3.2.1 Valid User Input

All base user input and use cases is derived from the assumption that user input is valid, as this is where the initial use case of the program being activated will leave. If user input is not valid, this use case will be unutilized and not proceed to another case. If valid, the user will then enter their expression and enter another use case.

3.2.2 Invalid User Input

Invalid input is functionally determined when the user enters information, characters, or numbers that are not able to be read in C++ or are programmed to be detected as an error by the "throw" and "catch" statements. If this use case is entered, the program will end and display a message to the user explaining why it did so. There is no use case following this, assuming invalid input is entered.

3.2.3 Numeric Constraint Change

A change to the numeric constraint (assuming int to floating-point from instructions) would be a non-functional use case where the program should be able to handle a change from integer input and output to floating-point numbers, or a combination of both. This assumes all functionality is compatible with floating-point numbers and rounds to a reasonable number of decimal places in the event of an infinite answer.

3.2.4 Addition and Subtraction

The addition and subtraction use cases are seen to be generalized to one functional use case, as they are covered in a similar fashion by the PEMDAS stack operation. A user, assuming valid input has been entered, can utilize the addition and subtraction features, and then either move to another use case or receive an output.

3.2.5 Multiplication and Division/Modulo

Multiplication, Division, and Modulo operations are grouped together according to how they are processed on the stack during PEMDAS operations and thus are seen to be a generalized functional use case. The user, assuming valid input, should be able to enter an expression and receive its product, dividend, or remainder. From here they should move to another use case for an operation or receive output.

3.2.6 Extraneous Parenthesis Usage

Extraneous parenthesis usage assumes valid user input, but also that there is a lot of parsing parentheses occurring. This is a non-functional use case, and as such, our program should be able to differentiate between the last opening parenthesis '(', the first number in an expression, and the first closing parenthesis ')' that matches with the initial opening. This should be done by initiating a count of opening parenthesis starting with the first before an expression, and end when a similar count of closing parenthesis is reached.

3.2.7 Exponentiation

Exponentiation assumes valid user input, and accordingly, must be either initiated next to a number, or an expression. Invalid exponentiation would be a blank input and just the '^n' entered. Assuming this is valid, exponentiation is a functional requirement that should return an int or floating-point value, depending on numerical constraint.

3.2.8 Multiple PEMDAS Operations

Assuming one use case has been previously utilized and the program has moved to its next functional use case, this case assumes that there are multiple expressions being completed at once and ensures compatibility between different features stated in section 3.1. From this use case, the program should either move to output or end in an invalid user input if only one expression is invalid.

3.2.9 Problem Size/Scale

C++ as a language can handle an int up to size 2147483647, and thus, may need help if the size of a problem exceeds the number that can be handled. This can be handled in two ways, but to keep this as a non-functional use case, we will assume that it must be able to handle a program of any size without crashing due to a space limitation, and thus will be handled by an outside header file that will be determined at a later time. This file will assist in minimizing input to manage space effectively.

3.2.10 Receiving Output

The output provided by the program is the final use case, assuming valid user input, and thus must be presented in a way that is not impossible to read to humans and is not excessive in size. Numbers over what C++ can handle should be presented in scientific notation to minimize them, but to ensure this use case is functional, we will once again assume output is an int or floating-point number to an expression and not an intentionally large number meant to hog memory. The display should be visually pleasing and easily justifiable as being the last thing you see before a user exits the program.

3.3 Supplementary Requirements

- *Speed*: The output of the expression entered by the user must be produced in a timely manner. All functions within our software must be with reasonable time complexity. The user realistically should not have to wait more than one second, after entering the input, to receive the output.
- *Portability*: Our software should be able to be utilized from any device. The application should work with no differences or issues between devices.
- Compactivity: Our software should be compatible between operating systems. If someone is using our product on an android device it should function no different that someone using it on IOS, etc..
- *Capacity*: Our product will have a maximum size of expression to negate the worry of capacity. The capacity, with how small our software is, should never be an issue.

4. Classification of Functional Requirements

Functionality	**Type**
Input Handling	Essential
Error Handling	Essential
PEMDAS Operation	Essential
Addition Operation	Essential
Subtraction Operation	Essential
Multiplication Operation	Essential
Division Operation	Essential
Modulo Operation	Essential
Exponentiation Operation	Essential
Numeric Constraints	Essential

5. Appendices

- Refer to EECS_348_Project_Plan found in the "Project Documents" folder on the git hub found here.