Assignment 1: Journal

Sources:

For this assignment, I primarily used the Think C++ textbook for reference. In my procedure, I first read through and took notes on chapters 1-7, as indicated in the Study Guide to strengthen my understanding of C++ concepts and complete these assignments. My notes as well as links to other sources I have referenced are listed below.

Think C++ Notes [Chapters 1-7]

Outputting:
- Use cout statements to output multiple lines.
- Comments can be inline (///) or standalone for clarity.

String Values:
- Strings are sequences of characters enclosed in double quotes.
- They can include escape sequences for special characters.
- Adding endl or \n prints output on a new line.
- Space between statements affects output spacing.

Variables:
- Variables must be declared with a specific data type.
- Variable names should be meaningful for clarity.

Assignment Statements:
- Assign values to variables using the assignment operator (=).
- Variables must be initialized before use.
- Use cout to display variable values.

Keywords:
- Keywords are reserved and cannot be used as variable names.

Operators:
- C++ provides various operators for arithmetic, logical, and bitwise operations.
- Integer division truncates the fractional part.

Operators for Characters:
- Characters can be manipulated using arithmetic operators.
- Automatic type conversion may occur between characters and integers.

Floating-point Numbers:

- Used to represent fractions and integers with decimal points.
- C++ has two types: float and double, with double being preferred.
- Variables can be declared and initialized with floating-point values.
- C++ distinguishes between integers and floating-point numbers.
- Automatic conversion may occur, but strict typing applies for assignments.

Converting Types:

- Typecasting allows conversion between types.
- Syntax resembles a function call, e.g., int x = int(pi);.

Math Functions:

- C++ provides built-in math functions like sin, cos, log, etc.
- Functions are invoked using standard mathematical notation.
- Ensure to include the <cmath> header file for math functions.

Composition:

- Functions can be composed, with one function's result used as another's argument.
- Expressions can be nested within function calls for complex computations.

Functions:

- Define custom functions using the syntax void functionName(parameters) { statements }.
- Functions can simplify code by encapsulating complex operations.
- Ensure function definitions appear before their first use.
- Functions can have parameters to receive values.
- Parameters are local to the function and must match in type with the arguments.
- Arguments passed to functions can be variables or literal values.
- Parameters must be declared with their types.
- When invoking functions, type declarations for arguments are unnecessary.

Local Variables:

- Variables declared within functions are local to that function.
- Local variables and parameters only exist within the function's scope.

Modulus Operator:

- Works on integers, yielding the remainder when the first operand is divided by the second.
- Represented by the percent sign %.
- Example: int quotient = 7 / 3; yields 2, while int remainder = 7 % 3; yields 1.

Conditional Execution:

- Essential for checking conditions and altering program behavior.
- if statement syntax: if (condition) { /* code */ }.
- Condition can use comparison operators: ==, !=, >, <, >=, <=.

Alternative Execution:

- Offers two possibilities based on a condition.
- Syntax: if (condition) { /* code */ } else { /* code */ }.
- Only one alternative executes based on the condition's truthiness.

Chained Conditionals:

- Checking multiple related conditions.
- Syntax: if { /* code */ } else if { /* code */ } else { /* code */ }.

Nested Conditionals:

- Conditional within another conditional.

The Return Statement:

- Terminates function execution prematurely.
- Useful for error handling or stopping execution based on conditions.
- Syntax: return; exits the function immediately.

Recursion:

- Function calls itself.
- Requires a base case to avoid infinite recursion.

Infinite Recursion:

- Occurs when a recursive function lacks a base case.
- Leads to endless function calls and program termination errors.
- Generally avoided in programming.

Void vs. Fruitful Functions:

- Void functions return no value and are called for their side effects.
- Fruitful functions return values and are defined with a return type other than void.
- Examples of fruitful functions include those that compute area or perform mathematical operations.

Return Statements:

- Return statements exit a function and can optionally return a value.
- The syntax: return expression; is used to return a value immediately from a function.
- The expression provided must match the return type declared for the function.
- Functions can have multiple return statements, but only one will be executed.

- Typically used in conditional branches, where the function exits upon hitting a return statement.
- Ensure that all possible paths through a function include a return statement.
- Failure to do so can lead to undefined behavior or compiler warnings/errors.

Compiler Errors and Debugging:
- Compiler errors are seen as helpful in debugging, as they point out potential issues in the code.
- Enabling strict compiler options can catch more errors during compilation.

Overloading: Functions with the same name but different parameters coexist.
Main function should return an integer indicating success or failure, conventionally 0 for success and other values for errors.

More on variables:
- It's legal to make more than one assignment to the same variable.
- The second assignment replaces the old value of the variable with a new value.
- Variables in C++ are described as containers for values.
- When you assign a value to a variable, you change the contents of the container.
- In C++, = is used for assignment, not equality.
- Equality is commutative, but assignment is not.
- Constantly changing variable values in different parts of the program can make the code difficult to read and debug.

Encapsulation and Generalization:
- Encapsulation involves wrapping a piece of code in a function, making the program easier to read and debug.
- Generalization involves making specific code more abstract and applicable to a wider range of scenarios.

Local and Global Variables:
- Variables declared inside a function are local to that function.
- Multiple variables with the same name can exist in different functions without causing conflicts.
- Changing the value of a local variable in one function does not affect the value of a variable with the same name in another function.
- The scope of variables is limited to the block or function in which they are declared.
- Variables with the same name can exist in different scopes without conflict.

Variable Types:
- C++ supports various types of variables, including bool, char, int, and double.
- Strings are not directly supported as a variable type

C Strings vs. C++ Strings:

- C++ has both C-style strings and its own string class.
- C-style strings have an ugly syntax and require manual memory management, while the string class is part of the C++ Standard Library and is easier to use.

String Variables:

- String variables can be declared and initialized using the string class.
- Automatic conversion from C-style strings to C++ strings is supported.
- Strings can be outputted using the standard output stream (cout).
- Characters can be extracted from strings using square brackets ([]) notation.
- The length() function returns the number of characters in a string.

String Traversal:

- Traversal involves iterating through each character of a string.
- Typically done using a loop with an index variable.
- Indexing outside the bounds of a string results in a run-time error.

String Search:

- The find() function locates the index of a specified character or substring within a string.
- It's possible to write a custom find() function with additional parameters to specify the starting index.
- Individual characters within a string can be modified.

Operators:

- Increment (++) and decrement (--) operators can be used to modify variables by one.
- The + operator can concatenate strings.
- Comparison operators (==, <, >, etc.) can be used to compare strings.

Character Classification:

- Functions like isalpha(), isdigit(), isspace(), etc., classify characters based on certain criteria.
- toupper() and tolower() functions convert characters to uppercase and lowercase, respectively.

Other sources referenced: https://www.learncpp.com/cpp-tutorial/introduction-to-stdstring/

Journal Entry 1: Multiplication Table [Problem 1]

Program Purpose:

The primary objective of this program is to generate a neatly formatted multiplication table up to 12x12. This task serves as an exercise to reinforce understanding and application of core programming concepts, including loops, functions, and output formatting techniques. By implementing this program, I aim to develop proficiency in loop implementation and code optimization while adhering to best coding practices.

Reflection on Assignment:

The implementation of the multiplication table program offered a valuable opportunity to translate theoretical knowledge into practical application. It enabled me to deepen my understanding of algorithm design, modular programming, and debugging methodologies. Moreover, it underscored the importance of meticulous attention to detail in ensuring program correctness and readability.

Activities Undertaken:

Understanding Requirements:

- Analyzed the assignment specifications to identify the required functionalities, input parameters, and output format.
- Recognized the necessity for two primary functions: one to calculate the number of digits (digs) and another to generate and print the multiplication table (mult)
  .

Design Decisions:

- Opted for a modular design approach to enhance code readability and maintainability.
- Decided to utilize a nested loop structure to iterate through multiplicands and multipliers efficiently, generating the multiplication table dynamically.

Implementation:

- Developed the program logic iteratively, focusing on writing clean, self-explanatory code.
- Ensured robust error handling and input validation to handle potential edge cases gracefully and prevent runtime errors.

Testing and Debugging:

- Conducted testing multiple times to ensure correct formatting and correct values.
- Debugged encountered issues promptly, employing systematic debugging strategies to identify and rectify logical errors.

Difficulties Encountered:

Formatting Challenges:

- Initially struggled with achieving consistent alignment and spacing in the output multiplication table.
- Overcame this challenge by implementing dynamic formatting logic based on the number of digits in each product.

Optimizing Performance:

- Faced difficulties in optimizing the program's efficiency, particularly regarding reducing redundant computations.
- Addressed this issue by refining loop structures and minimizing unnecessary calculations to improve runtime performance.

Learning Outcomes Fulfilled:

Consolidation of Core Concepts:

- Reinforced understanding of fundamental programming concepts, including loops, functions, and conditional statements.
- Applied these concepts effectively to design and implement a solution to the assignment requirements.

Enhanced Debugging Skills:

- Improved proficiency in debugging techniques, including systematic error diagnosis and resolution.
- Developed the ability to identify and rectify logical errors and edge cases through rigorous testing and debugging processes.

Sources Used:

No external sources were consulted for this assignment.

Reflection on Programming Process:

This assignment underscored the significance of systematic problem-solving approaches, effective collaboration, and continuous learning. Through this experience, I not only deepened my technical expertise but also cultivated a mindset of adaptability and resilience in the face of programming challenges.

Journal Entry 2: Temperature Converter [Problem 2]

Program Purpose:

The primary objective of this program is to implement a temperature converter that allows users to convert temperatures between Fahrenheit and Celsius units. The program gets input from the user, performs the conversion based on the specified unit, and provides the converted temperature as output. It serves as an exercise in implementing basic control structures, user input handling, and functions in C++.

Reflection on Assignment:

This assignment presented an opportunity to develop a practical utility program catering to everyday needs. By implementing a temperature converter, I gained insights into the importance of user interaction, input validation, and functional decomposition. Additionally, it allowed for exploration of different strategies for error handling.

Activities Undertaken:

Understanding Requirements:

- Carefully reviewed the assignment specifications to comprehend the expected behavior and functionality of the temperature converter.
- Identified key components such as user input prompts, conversion logic, and output formatting requirements.

Design Decisions:

- Opted for a modular design approach, encapsulating temperature conversion logic within separate functions (fahrenheitToCelsius and celsiusToFahrenheit).
- Designed an interface leveraging a do-while loop to facilitate multiple conversions until the user opts to exit or enters invalid data.

Implementation:

- Implemented the program logic in adherence to the outlined design decisions, focusing on clarity, modularity, and robust error handling.
- Utilized appropriate data types and control structures to ensure accurate temperature conversions and user-friendly interaction.

Testing and Debugging:

- Conducted testing across various input scenarios to validate the correctness and reliability of the program.
- Debugged any encountered issues promptly, emphasizing error detection and graceful handling.

Difficulties Encountered:

Input Validation:

- Faced challenges in designing effective input validation mechanisms to handle invalid user inputs gracefully.
- Addressed through iterative refinement of input processing logic and incorporation of informative error messages to guide user interactions.

Learning Outcomes Fulfilled:

This problem reinforced my understanding of core programming concepts, including functions, loops, conditional statements, and data manipulation, through practical application in a real-world context.

Reflection on Programming Process:

The programming process for this assignment was both enlightening and rewarding. It provided valuable hands-on experience in software development, highlighting the iterative nature of problem-solving and the significance of systematic design and implementation.

Journal Entry 3: Temperature Table [Problem 3]

Program Purpose:

The primary objective of this program is to generate a temperature conversion table displaying conversions between Celsius and Fahrenheit temperatures. The program systematically calculates and displays temperature conversions within a specified temperature range, providing users with a convenient reference for converting temperatures between the two scales.

Reflection on Assignment:

This assignment provided an opportunity to develop a practical utility program focusing on temperature conversion. By implementing a temperature conversion table generator, I enhanced my understanding of data manipulation, output formatting, and iterative processing in C++. Additionally, it allowed for exploration of precision handling and formatting techniques to ensure readability and accuracy of the generated table.

Activities Undertaken:

Understanding Requirements:

- Thoroughly reviewed the assignment specifications to comprehend the expected behavior and format of the temperature conversion table.
- Identified key components such as temperature range, conversion functions, and output formatting requirements.

Implementation:

- Implemented the program logic according to the outlined design decisions, ensuring accurate temperature conversions and proper output formatting.
- Utilized loop structures to systematically generate temperature conversions within the specified range, incrementing temperature values with each iteration.

Testing and Debugging:

- Conducted comprehensive testing to validate the correctness and reliability of the temperature conversion table.
- Debugged any encountered issues promptly, focusing on precision handling, formatting inconsistencies, and boundary conditions.

Difficulties Encountered:

Output Formatting Consistency:

- Encountered difficulties in ensuring consistent and aesthetically pleasing output formatting across different temperature values and conversion pairs.
- Mitigated by fine-tuning formatting parameters and utilizing setw manipulators to align output columns uniformly.

Learning Outcomes Fulfilled:

Advanced Data Manipulation Skills:

- Enhanced proficiency in manipulating numerical data and performing arithmetic operations, particularly in the context of temperature conversions.
- Developed a deeper understanding of precision handling and formatting nuances in numerical computations.

Iterative Processing Techniques:

- Gained insights into designing iterative processing algorithms to systematically generate and present tabular data, emphasizing efficiency and readability.

- Strengthened problem-solving abilities through iterative refinement of program logic and output presentation.

Enhanced Output Formatting Expertise:

- Expanded expertise in output formatting techniques, including setw manipulators and precision settings, to achieve consistent and visually appealing output layouts.

Reflection on Programming Process:

The programming process for this assignment provided valuable insights into numerical computations, output formatting, and iterative processing. Through this endeavor, I not only deepened my technical skills but also gained a deeper appreciation for the importance of precision, consistency, and attention to detail in software development.

Journal Entry 4: C++ Menu [Problem 4]

Program Purpose:

The primary objective of this program is to provide a help menu for C++ language constructs, including 'if' statements, 'switch' statements, 'for' loops, 'while' loops, and 'do-while' loops. The program allows users to select a topic of interest from the menu and displays detailed information about the selected construct.

Reflection on Assignment:

By implementing a help menu, I explored strategies for user interface design, input handling, and content presentation. Additionally, it underscored the importance of clear communication and user-centric design in educational software development.

Activities Undertaken:

Understanding Requirements:

- Carefully reviewed the assignment specifications to ascertain the functionality and content requirements for the help menu.
- Identified key components such as menu options, input handling, content presentation, and exit conditions.

Design Decisions:

- Opted for a loop-driven menu system, allowing users to select topics of interest iteratively until choosing to exit.
- Designed a switch statement to display detailed information about each selected construct, enhancing modularity and readability.

Implementation:

- Implemented the program logic according to the outlined design decisions, focusing on clarity and content accuracy.

Testing and Debugging:

- Conducted thorough testing to validate the correctness and usability of the help menu across various user interactions.

Difficulties Encountered:

Input Handling Challenges:

- Faced challenges in handling user input efficiently and gracefully, particularly regarding validation and error handling.
- Addressed through iterative refinement of input processing logic and incorporation of informative error messages to guide user interactions.

Learning Outcomes Fulfilled:

Effective Communication Skills:

- Developed proficiency in articulating complex programming concepts in a clear and accessible manner, catering to diverse user audiences.
- Enhanced ability to convey technical information effectively through concise and coherent language.

Iterative Design and Refinement:

- Gained insights into the iterative nature of software design and development, emphasizing continuous refinement based on user feedback and testing results.
- Cultivated a mindset of adaptability and flexibility in addressing evolving requirements and improving overall user experience.

Sources Used:

https://cplusplus.com/doc/tutorial/control/

Reflection on Programming Process:

The programming process for this assignment provided valuable insights into control flow systems in C++, and further honed my technical skills.

Journal Entry 5: Prime Numbers [Problem 5]

Program Purpose:

The primary objective of this program is to identify and print all prime numbers within the range of 1 to 9999. It employs a brute-force approach, iterating through each number within the specified range and checking for divisibility to determine primality. The program serves as an exercise in algorithmic thinking, loop structures, and boolean logic in C++ programming.

Reflection on Assignment:

This assignment provided an opportunity to delve into the concept of prime numbers and explore various approaches to identifying them algorithmically. By implementing a prime number finder, I gained insights into the efficiency considerations associated with different algorithms. Additionally, it facilitated practical experience in loop optimization and problem-solving strategies.

Activities Undertaken:

Understanding Requirements:

- Analyzed the assignment specifications to grasp the desired functionality and constraints of the prime number finder.
- Identified the need for nested loop structures to iterate through numbers and determine primality.

Design Decisions:

- Opted for a straightforward algorithmic approach, utilizing nested loops to systematically evaluate each number's primality within the specified range.
- Chose to implement a boolean flag (checkPrime) to track the primality status of each number efficiently.

Implementation:

- Implemented the program logic according to the outlined design decisions, focusing on clarity, simplicity, and efficiency.
- Employed appropriate loop termination conditions and conditional statements to optimize performance and minimize unnecessary computations.

Testing and Debugging:

- Conducted thorough testing across various input scenarios to validate the correctness and completeness of the prime number identification process.

Learning Outcomes Fulfilled:

Algorithmic Thinking Skills:

- Developed proficiency in algorithmic thinking and problem-solving strategies, particularly in devising efficient algorithms for identifying prime numbers.

Loop Optimization Techniques:

- Enhanced understanding of loop optimization techniques, including loop termination conditions, early exit strategies, and iteration control mechanisms.

Reflection on Programming Process:

The programming process for this assignment provided valuable hands-on experience in algorithmic problem-solving and performance optimization.