1. Area of a Triangle

Problem Statement

Write a C++ program to find the area of any triangle using Heron's formula.

How to Tackle the Problem

To calculate the area of a triangle given its three sides, Heron's formula is an efficient approach. The core idea is to first calculate the semi-perimeter (half the perimeter) and then use it in the area formula. It's crucial to validate the input to ensure that the side lengths are positive and that they can actually form a valid triangle (the sum of any two sides must be greater than the third).

1. **Understand Heron's Formula:**
   - Semi-perimeter `s = (a + b + c) / 2`
   - Area = `sqrt(s * (s - a) * (s - b) * (s - c))`
2. **Input Collection:** Prompt the user to enter the three side lengths.
3. **Input Validation (Enhanced Version):**
   - **Positive Sides:** Ensure all side lengths (`a`, `b`, `c`) are greater than zero. If not, re-prompt the user.
   - **Triangle Inequality Theorem:** Verify that the sum of any two sides is greater than the third side (e.g., `a + b > c`, `a + c > b`, `b + c > a`). If this condition isn't met, a triangle cannot be formed with the given sides.
4. **Calculation:** Once valid inputs are confirmed (or directly for the simple version), proceed with calculating `s` and then the `area`.
5. **Output:** Display the calculated area.

C++ Code (Just Working Version)

```cpp
#include <iostream>
#include <cmath>

using namespace std;

int main() {
  double a, b, c, s, area;

  cout << "Enter three sides of triangle: ";
  cin >> a >> b >> c;

  s = (a + b + c) / 2;
  area = sqrt(s*(s - a)*(s - b)*(s - c));

  cout << "Area of a triangle with sides " << a << ' ' << b << ' ' << c << " = "
↪  << area << '\n';

  return 0;
```

```
}
```

C++ Code (Enhanced Version with Validation)

```cpp
#include <iostream>
#include <cmath>

using namespace std;

int main() {
  double a, b, c, s, area;

  cout << "Enter three sides of triangle: ";
  cin >> a >> b >> c;

  // Validate for positive side lengths
  while( a <= 0 || b <= 0 || c <= 0) {
    cout << "Invalid Dimensions: " << a << ' ' << b << ' ' << c << '\n';
    cout << "Renter positive values only: ";
    cin >> a >> b >> c;
  }

  s = (a + b + c) / 2;

  // Validate for triangle inequality theorem
  if(a + b <= c || a + c <= b || b + c <= a) {
    cout << "Invalid Dimensions: The given sides cannot form a triangle.\n";
    return 1; // Indicate an error
  }

  area = sqrt(s*(s - a)*(s - b)*(s - c));

  cout << "Area of a triangle with sides " << a << ' ' << b << ' ' << c << " = "
 ↪  << area << '\n';

  return 0;
}
```

---

2. Even or Odd

Problem Statement

Write a Program to Check Even or Odd Integers.

How to Tackle the Problem

Determining if an integer is even or odd relies on the concept of divisibility by 2. An even number is perfectly divisible by 2 (leaves no remainder), while an odd number is not. The modulo operator (`%`) in C++ is perfect for this check.

1. **Input:** Get an integer from the user.
2. **Core Logic (Modulo Operator):** Use the expression `number % 2`.
   - If the result is `0`, the number is even.
   - If the result is `1` (or `-1` for negative odd numbers, depending on implementation), the number is odd.
3. **Conditional Output:** Use an `if-else` statement to print whether the number is even or odd based on the modulo result.

C++ Code

```cpp
#include <iostream>

using namespace std;

int main() {
  int number;

  cout << "Enter an integer to check whether it\'s even or odd: ";
  cin >> number;

  if(number % 2 == 0) {
    cout << number << " is an even number\n";
  } else {
    cout << number << " is an odd number\n";
  }

  return 0;
}
```

---

3. Factorial

Problem Statement

Find the Factorial of the number entered by the user.

How to Tackle the Problem

Calculating the factorial of a non-negative integer `n` (denoted as `n!`) involves multiplying all positive integers less than or equal to `n`. For example, `5! = 5 * 4 * 3 * 2 * 1 = 120`. Special cases include `0! = 1`. Factorials grow very rapidly, so choosing an appropriate data type (`long long unsigned` for larger numbers) and handling

potential overflows is important. Input validation for non-negative numbers is also essential.

1. **Understand Factorial Definition:** `n! = n * (n-1) * ... * 1` for `n > 0`, and `0! = 1`.
2. **Input:** Get a non-negative integer from the user.
3. **Input Validation (Enhanced Version):** Ensure the number is non-negative. Also, consider practical limits for `long long unsigned` to prevent overflow (e.g., factorials beyond 65 might exceed its capacity).
4. **Base Case:** If the number is 0, the factorial is 1.
5. **Iterative Calculation:** Use a loop (e.g., `for` loop) to multiply a running product (initialized to 1) by each integer from 1 up to the input number.
6. **Output:** Display the calculated factorial.

C++ Code (Just Working Version)

```cpp
#include <iostream>

using namespace std;

int main() {
  int number;
  long long unsigned fact = 1;

  cout << "Enter a non-negative integer: ";
  cin >> number;

  if(number == 0) {
    cout << "Factorial of " << number << " = " << 1 << '\n';
    return 0;
  }

  for(int i = 1; i <= number; i++) {
    fact *= i;
  }

  cout << "Factorial of " << number << " is " << fact << '\n';
  return 0;
}
```

C++ Code (Enhanced Version with Validation)

```cpp
#include <iostream>

using namespace std;

int main() {
  int number;
```

```
  long long unsigned fact = 1; // Use long long unsigned for larger factorials

  cout << "Enter a non-negative integer to get its factorial: ";
  cin >> number;

  // Input validation for non-negative numbers and practical limits
  while (number < 0 || number > 65) { // 65! is roughly the max for long long
   ↪  unsigned
    cout << "Invalid input. Factorials are defined for non-negative integers.\n";
    cout << "Also, for practical purposes, please enter a number less than or
 ↪  equal to 65: ";
    cin >> number;
  }

  if(number == 0) {
    cout << "Factorial of " << number << " = " << 1 << '\n';
    return 0;
  }

  for(int i = 1; i <= number; i++) {
    fact *= i;
  }

  cout << "Factorial of " << number << " is " << fact << '\n';
  return 0;
}
```

---

4. Armstrong Number

Problem Statement

Find if the entered number is Armstrong or not?

How to Tackle the Problem

An Armstrong number (also known as a narcissistic number) is a number that is equal to the sum of its own digits each raised to the power of the number of digits. For example, 153 is an Armstrong number because it has 3 digits, and `1^3 + 5^3 + 3^3 = 1 + 125 + 27 = 153`. The strategy involves extracting each digit, determining the total number of digits, and then performing the sum of powers.

1. **Input:** Get a positive integer from the user.
2. **Handle Single-Digit Numbers:** Single-digit numbers are trivially Armstrong numbers.
3. **Determine Number of Digits:** Convert the number to a string to easily find its length, which represents the number of digits. Alternatively, use a loop with integer division to count digits.

4. **Extract Digits and Calculate Sum of Powers:** Iterate through the digits. For each digit:
   - Extract the last digit using the modulo operator (`% 10`).
   - Raise this digit to the power of the total number of digits.
   - Add this result to a running sum.
   - Remove the last digit from the number using integer division (`/ 10`).
5. **Comparison:** Compare the calculated sum with the original number. If they are equal, it's an Armstrong number.

C++ Code (Just Working Version)

```cpp
#include <iostream>
#include <string>
#include <cmath>

using namespace std;

int main() {
  int number, originalNumber, remainder, n = 0, result = 0;

  cout << "Enter a positive integer: ";
  cin >> number;

  originalNumber = number;

  // Count the number of digits
  string str_num = to_string(originalNumber);
  n = str_num.length();

  // Calculate sum of powers of digits
  originalNumber = number; // Reset originalNumber for calculation
  while (originalNumber != 0) {
    remainder = originalNumber % 10;
    result += pow(remainder, n);
    originalNumber /= 10;
  }

  if(result == number) {
    cout << number << " is an Armstrong number.\n";
  } else {
    cout << number << " is not an Armstrong number.\n";
  }
  return 0;
}
```

C++ Code (Enhanced Version with Validation)

```cpp
#include <iostream>
#include <string>
#include <cmath>

using namespace std;

int main() {
  int number, originalNumber, remainder, n = 0, result = 0;

  cout << "Enter a positive integer to check if it\'s an Armstrong number: ";
  cin >> number;

  // Input validation
  if (number < 0) {
    cout << "Invalid input: Please enter a positive integer.\n";
    return 1;
  }

  originalNumber = number;

  // Count the number of digits
  // For single digit numbers, they are Armstrong numbers by definition
  if (originalNumber == 0) {
      n = 1;
  } else {
      string str_num = to_string(originalNumber);
      n = str_num.length();
  }

  // Calculate sum of powers of digits
  originalNumber = number; // Reset originalNumber for calculation
  while (originalNumber != 0) {
    remainder = originalNumber % 10;
    result += pow(remainder, n);
    originalNumber /= 10;
  }

  if(result == number) {
    cout << number << " is an Armstrong number.\n";
  } else {
    cout << number << " is not an Armstrong number.\n";
  }
  return 0;
}
```

5. Largest of Three Numbers

Problem Statement

Write a Program to Find the Largest Among 3 Numbers.

How to Tackle the Problem

Finding the largest among three numbers involves comparing them systematically. While a series of `if-else if` statements can work, C++ provides convenient functions like `std::max` (from the `<algorithm>` header) that can simplify this task, especially when dealing with multiple values.

1. **Input:** Get three integers from the user.
2. **Comparison Logic:**
   - **Manual Comparison:** You could use nested `if` statements or a chain of `if-else if` to compare `a` with `b`, and then the larger of those with `c`.
   - **Using `std::max`:** The `std::max` function can take an initializer list (e.g., `{a, b, c}`) to find the maximum value among them directly. This is generally cleaner and less error-prone.
3. **Output:** Display the largest number. For an enhanced version, you might also consider finding the smallest and middle numbers.

C++ Code (Just Working Version: Finds Largest)

```cpp
#include <iostream>
#include <algorithm> // Required for std::max

using namespace std;

int main() {
  int a, b, c;

  cout << "Enter three numbers: ";
  cin >> a >> b >> c;

  int largest = max({a, b, c});

  cout << "Largest = " << largest << '\n';
  return 0;
}
```

C++ Code (Enhanced Version: Finds Largest, Smallest, Middle)

```cpp
#include <iostream>
#include <algorithm> // Required for std::max and std::min

using namespace std;
```

```cpp
int main() {
  int a, b, c;

  cout << "Enter three numbers a, b and c: ";
  cin >> a >> b >> c;

  // Check if all numbers are equal
  if(a == b && a == c) {
    cout << "All three numbers are equal.\n";
  } else {
    // Use std::max and std::min for efficient comparison
    int largest = max({a, b, c});
    int smallest = min({a, b, c});
    // Calculate middle by summing all and subtracting largest and smallest
    int middle = a + b + c - largest - smallest;

    cout << "Largest = " << largest << '\n';
    cout << "Smallest = " << smallest << '\n';
    cout << "Middle = " << middle << '\n';
  }
  return 0;
}
```

---

6. Prime or Composite

Problem Statement

Write a Program to Check Whether a Number Is Prime or composite.

How to Tackle the Problem

A prime number is a natural number greater than 1 that has no positive divisors other than 1 and itself. A composite number is a natural number greater than 1 that is not prime. The most straightforward way to check is to try dividing the number by all integers from 2 up to `number - 1`. However, an important optimization is that you only need to check for divisors up to the square root of the number. If no divisors are found up to that point, the number is prime.

1. **Input:** Get an integer from the user.
2. **Handle Special Cases:**
   - Numbers less than or equal to 1 are neither prime nor composite (or specifically, 1 is not prime).
   - 2 is the only even prime number.
3. **Iterative Division (Simple Version):** Loop from 2 up to `number - 1`.
   - Inside the loop, use the modulo operator (`%`) to check if the input number is divisible by the current loop variable.

- If it is divisible, the number is composite, and you can stop checking.

4. **Optimization (Enhanced Version):** Calculate the integer part of the square root of the number. You only need to check for divisibility up to this value. Also, you can skip even numbers as divisors after checking for 2.
5. **Output:** If the loop completes without finding any divisors, the number is prime.

C++ Code (Just Working Version)

```cpp
#include <iostream>

using namespace std;

int main() {
  int number;

  cout << "Enter a number: ";
  cin >> number;

  // Handle special cases for numbers <= 1
  if (number <= 1) {
    cout << number << " is neither prime nor composite.\n";
    return 0;
  }

  for(int i = 2; i < number; i ++) {
    if(number % i == 0) {
      cout << number << " is composite.\n";
      return 0;
    }
  }

  cout << number << " is prime.\n";
  return 0;
}
```

C++ Code (Enhanced Version with Optimization)

```cpp
#include <iostream>
#include <cmath>

using namespace std;

int main() {
  int number;

  cout << "Enter a number to check whether it\'s prime or composite: ";
  cin >> number;
```

```cpp
  // Handle special cases
  if (number <= 1) {
    cout << number << " is neither prime nor composite.\n";
    return 0;
  }
  if (number == 2) {
    cout << number << " is prime.\n";
    return 0;
  }
  if (number % 2 == 0) {
    cout << number << " is composite (divisible by 2).\n";
    return 0;
  }

  // Optimization: check divisibility only up to the square root
  int limit = round(sqrt(number));

  for(int i = 3; i <= limit; i += 2) { // Start from 3 and increment by 2 (skip
  ↪  even numbers)
    if(number % i == 0) {
      cout << number << " is composite as it has a smallest factor " << i <<
↪  '\n';
      return 0;
    }
  }

  cout << number << " is prime as it has no factors other than 1 and itself.\n";
  return 0;
}
```

---

7. Reverse a 3-Digit Number

Problem Statement

Display the reverse of a three-digit entered number.

How to Tackle the Problem

Reversing a number, especially a fixed-digit one like a 3-digit number, involves extracting its individual digits and then reconstructing the number in reverse order. The modulo operator (`%`) is used to get the last digit, and integer division (`/`) is used to remove the last digit.

1. **Input:** Get a 3-digit integer from the user.
2. **Input Validation (Enhanced Version):** Ensure the entered number is indeed a 3-digit number (e.g., between 100 and 999, or -999 and -100 for negative numbers).

3. **Extract Digits:**
   - **Last Digit (Units Place):** `first = number % 10;`
   - **Middle Digit (Tens Place):** `second = (number / 10) % 10;`
   - **First Digit (Hundreds Place):** `third = number / 100;`
4. **Reconstruct Reversed Number:** Combine the extracted digits in reverse order, multiplying by appropriate powers of 10: `reversed_number = first * 100 + second * 10 + third;`
5. **Output:** Display the reversed number.

C++ Code (Just Working Version)

```cpp
#include <iostream>

using namespace std;

int main() {
  int number, first, second, third;
  cout << "Enter a 3-digit number: ";
  cin >> number;

  first = number % 10; // Extracts the units digit
  second = (number / 10) % 10; // Extracts the tens digit
  third = number / 100; // Extracts the hundreds digit

  int result = first * 100 + second * 10 + third;
  cout << "The reversed number is: " << result << '\n';
  return 0;
}
```

C++ Code (Enhanced Version with Validation)

```cpp
#include <iostream>

using namespace std;

int main() {
  int number, first, second, third;
  cout << "Enter a 3-digit number to reverse it: ";
  cin >> number;

  // Input validation for a 3-digit number
  while(number < -999 || number > 999 || (number > -100 && number < 100)) {
    cout << "Invalid input: Only three-digit numbers are allowed (e.g., 123 or
↪  -456).\n";
    cout << "Please enter a 3-digit number: ";
    cin >> number;
  }
```

```
  // Extract digits
  first = number % 10; // Extracts the units digit
  second = (number / 10) % 10; // Extracts the tens digit
  third = number / 100; // Extracts the hundreds digit

  // Reconstruct the number in reverse order
  int result = first * 100 + second * 10 + third;
  cout << "The reversed number is: " << result << '\n';
  return 0;
}
```

---

8. Swap Two Numbers

Problem Statement

Write a C++ program that swaps two numbers.

How to Tackle the Problem

Swapping the values of two variables means that the value originally held by `a` should now be in `b`, and the value originally held by `b` should now be in `a`. The most common and straightforward method involves using a temporary variable to hold one of the values during the swap process.

1. **Input:** Get two numbers (e.g., `a` and `b`) from the user.
2. **Introduce a Temporary Variable:** Declare a third variable, `temp`, of the same data type as `a` and `b`.
3. **Perform the Swap:**
   - Store the value of `a` into `temp`: `temp = a;`
   - Assign the value of `b` to `a`: `a = b;`
   - Assign the value stored in `temp` (which was `a`'s original value) to `b`: `b = temp;`
4. **Output:** Display the values of `a` and `b` after the swap to confirm the operation.

C++ Code

```
#include <iostream>

using namespace std;

int main () {
  int a, b, temp;

  cout << "Enter values for a and b: ";
  cin >> a >> b;
```

```
  cout << "Values before swapping: a = " << a << " and b = " << b << '\n';

  // Perform the swap using a temporary variable
  temp = a;
  a = b;
  b = temp;

  cout << "Values after swapping: a = " << a << " and b = " << b << '\n';

  return 0;
}
```

---

9. Multiplication Table

Problem Statement

Display the table of a number.

How to Tackle the Problem

Generating a multiplication table for a given number involves repeatedly multiplying that number by a sequence of integers (typically from 1 to 10). A `for` loop is the ideal control structure for this task, as it allows you to iterate a fixed number of times and perform the multiplication in each iteration.

1. **Input:** Get an integer from the user for which the table is to be generated.
2. **Define Range:** Decide the range for the multiplication (e.g., from 1 to 10). A `const int` can be used for the upper limit for clarity.
3. **Iterative Multiplication:** Use a `for` loop that iterates from the starting multiplier (e.g., 1) up to the ending multiplier (e.g., 10).
   - Inside the loop, calculate the product of the input number and the current multiplier.
   - Print the multiplication expression and its result in a clear format (e.g., `number X i = product`).

C++ Code

```
#include <iostream>

using namespace std;

int main() {
  const int MAX_MULTIPLIER = 10; // Define the upper limit for the table
  int number;

  cout << "Enter an integer to get its multiplication table (up to " <<
↪  MAX_MULTIPLIER << "): ";
```

```
  cin >> number;

  for(int i = 1; i <= MAX_MULTIPLIER; i++) {
    cout << number << "  X  " << i << "  =  " << number * i << '\n';
  }
  return 0;
}
```

---

10. Nested If-Else (Electricity Bill Calculation)

Problem Statement

Scenario based question of Nested if statement program - electricity bill calculations or salary calculation.

How to Tackle the Problem

This problem involves calculating an electricity bill based on a tiered pricing structure, which is a classic use case for `if-else if` (or nested `if-else`) statements. The key is to process the units consumed in stages, applying different rates to different consumption blocks. Input validation is important to ensure that the units consumed are non-negative.

1. **Input:** Get the number of electricity units consumed from the user.
2. **Input Validation:** Ensure the units consumed are not negative.
3. **Tiered Calculation Logic:** Use a series of `if-else if` statements to determine the bill:
   - **First Tier:** If units are within the first block (e.g., 0-100), apply the rate for that block.
   - **Subsequent Tiers:** If units exceed a block, calculate the cost for the full preceding blocks at their respective rates, and then apply the current block's rate to the remaining units.
   - This approach ensures that each unit is charged at its correct tier rate.
4. **Output:** Display the total calculated electricity bill.

C++ Code

```
#include <iostream>

using namespace std;

int main() {
    float units, bill;

    cout << "Enter the number of electricity units consumed: ";
```

```
    cin >> units;

    // Input validation
    if (units < 0) {
        cout << "Invalid input: Units cannot be negative." << endl;
        return 1;
    }

    if (units <= 100) {
        bill = units * 0.50; // Rate for first 100 units
    } else if (units <= 200) {
        // Cost for first 100 units + cost for units in 101-200 range
        bill = (100 * 0.50) + ((units - 100) * 0.75);
    } else if (units <= 300) {
        // Cost for first 100 + cost for next 100 + cost for units in 201-300
        ↪ range
        bill = (100 * 0.50) + (100 * 0.75) + ((units - 200) * 1.20);
    } else { // For units > 300
        // Cost for first 100 + cost for next 100 + cost for next 100 + cost for
        ↪ remaining units
        bill = (100 * 0.50) + (100 * 0.75) + (100 * 1.20) + ((units - 300) *
↪ 1.50);
    }

    cout << "Your electricity bill is: " << bill << endl;

    return 0;
}
```
