Project 2 specifications

September 2024

1 Overview and purpose

This project has you implement a number of functions that work with integers. Here we describe where each of the functions you will be implementing are used in real-world applications, but you are welcome to skip this section and go immediately to the project description in Section 2 below.

Nested for loops are frequently employed in real-world applications across various domains.

- In **matrix operations** like multiplication and using the transpose, these loops iterate over rows and columns to compute the result, fundamental in computer graphics and scientific computing.
- **Image processing** tasks such as applying filters or converting images to gray-scale use nested loops to traverse pixel grids.
- In **game development**, nested loops manage game maps for updating states, checking collisions, or rendering scenes, essential in tile-based games and simulations.
- Scientific computing often involves **multi-dimensional data manipulation**, such as performing tensor operations.
- Engineering applications include **finite element analysis** where nested loops iterate over grid points to approximate solutions to partial differential equations, and digital signal processing where they apply convolution operations to signals.

These loops are pivotal in efficiently handling complex, structured data across diverse fields.

Integer logarithms are used in the following cases: In trading systems, they compute approximate exponential moving averages and other financial indicators that are faster than using floating-point computations. In computer graphics and game development, they help determine the appropriate level of detail for 3D models based on distance. In network protocols, they adjust congestion window sizes in algorithms like TCP Cubic. In data compression, they optimize the encoding process in Huffman coding. In signal processing, they design efficient digital filters. In cryptography, they facilitate rapid computations in hash functions and security algorithms. In machine learning, they speed up information gain calculations in decision trees. In database indexing, they maintain balanced B-trees for quick data retrieval. In audio processing, they calculate gains for equalization. In embedded systems, they efficiently scale and transform sensor data for timely responses.

Counting the number of 0s or 1s in an unsigned int is useful for various reasons. In error detection and correction, it helps identify parity bits and ensure data integrity. In cryptography, it assesses the randomness and distribution of bits, contributing to the security of encryption algorithms. In network protocols, it helps manage bitwise operations and optimize data transmission. In graphics and image processing, it aids in pixel manipulation and compression techniques. In embedded systems, it supports efficient sensor data processing and control logic. In machine learning, it assists in feature extraction and binary classification tasks. In

performance optimization, it enables efficient bitwise algorithms and hardware implementations. In digital signal processing, it facilitates modulation and encoding schemes. Overall, counting 0s or 1s is a fundamental operation that enhances efficiency, security, and accuracy across various computational tasks.

Swapping two bytes within a larger unsigned int or unsigned long has several practical applications. In data serialization and deserialization, it ensures correct endianness, allowing data to be accurately interpreted across different systems. In network protocols, it facilitates efficient data transmission by converting between network and host byte orders. In cryptography, it enhances security algorithms by randomizing data patterns and complicating analysis. In graphics and image processing, it optimizes pixel data manipulation and color space conversions. In embedded systems, it aids in memory management and peripheral communication, ensuring data is correctly formatted for hardware interfaces. In performance optimization, it allows for efficient manipulation of data structures and buffer management. In digital signal processing, it supports efficient encoding and decoding schemes. Overall, byte swapping is a crucial operation that ensures data integrity, compatibility, and efficiency in various computational tasks.

2 Project description

The first function `void pattern( unsigned int n )` displays a pattern based on the value of n. These patterns are shown in the Appendix. At the end of each line containing at least on asterisks, you must print an end-of-line (`std::endl`). This includes the last line.

The second function `unsigned int log10( unsigned int n )` returns the largest exponent m such that $10^m \leq n$. You will assert that the argument $n \neq 0$.

The third function `unsigned int count( unsigned int n, unsigned int bit )` will return the number of times that the bit `b` appears in the argument `n`. You will assert that `bit` is either equal to 0 or 1.

The fourth function `unsigned int swap_bytes( unsigned int n, unsigned int b0, unsigned int b1 )` will swap bytes `b0` and `b1` in the unsigned integer `n`. You will assert that the arguments `b0` and `b1` are integers between 0 and 3 inclusively. If these integers are equal, you will return `n` unchanged. Otherwise, you will swap the two bytes leaving the other two bytes unchanged. By convention, Byte 0 is the least significant byte and Byte 3 is the most significant byte, as indicated here:

```
33333333222222221111111100000000
```

2.1 Testing

In Project 1, we described many of the tests that could be used to verify that your project is working correctly. In this project, you will be required to author your own tests. We will, however, give you some hints.

1. If the integer logarithm of n is m, then if you perform an integer calculation dividing n by 10^m, the result should be an integer between 1 and 9.
2. You can initialize an `unsigned int` with a binary number. Also, the C++ 2020 function `std::popcount` returns the number of bits equal to `1`, so you can test your output on an argument `n` with the output of `std::popcount( n )` and `std::popcount( ~n )`. You should convince yourself that the second counts the number of bits set to `0` in `n`. The function `std::popcount` is in the `bit` library, which you can read about at cpp-reference. While you can test your code with `popcount(...)`, your submission cannot use the `popcount(...)` function.

3. You can initialize an `unsigned int` using a value in hexadecimal using `0x`, but to tell the compiler that this is unsigned, you need to append a `U` to the literal. Thus, `0x03020100U` has the bytes `03`, `02`, `01` and `00`. If you swap bytes 1 and 3, the result should be `0x01020300U`. It may be a little dangerous to have one of the bytes be zero, so you might want to instead have four random bytes like `0x6f1b5a04U`, and then swapping bytes 1 and 3 should result in `0x5a1b6f04U`.

Appendix

If $n = 0$, print one asterisks. If $n = 1, 2, 3, 4, 5$, the output should be as follows below. You must implement this for an arbitrary value of n and this must be implemented using nested for loops.

```
n = 1:
***
 *
***

n = 2:
*****
 ***
  *
 ***
*****

n = 3:
*******
 *****
  ***
   *
  ***
 *****
*******

n = 4:
*********
 *******
  *****
   ***
    *
   ***
  *****
 *******
*********

n = 5:
***********
 *********
  *******
   *****
    ***
     *
    ***
   *****
  *******
 *********
***********
```

This output is generated with the loop, where the end-of-line character is only printed between patterns to separate them.

```
std::cout << "n = 1:" << std::endl;
pattern( 1 );

for ( unsigned int n{ 2 }; n <= 5; ++n ) {
  std::cout << std::endl;
  std::cout << "n = " << n << ":" << std::endl;
  pattern( n );
}
```