DSBA Introduction to Programming // Workshops 23+

Spring semester 2022/23

Part 1. Bitwise operations.

Task 1.1. Get bit at index.

You are given an integer number **A** and an index **i**. Find and print the bit at position **i** in the number. The least significant bits have the lower indices.

Example input:

```
9 1
```

Example output:

```
0
```

Task 1.2. Number of ones.

You are given an integer number **A** in the range 0 <= A <= 4*10^18. Print the number of bits set to 1 in the binary representation of *A*.

Example input:

```
9
```

Example output:

```
2
```

Example input:

```
126
```

Example output:

```
6
```

Task 1.3. Output Binary Number.

Write a function that takes an unsigned integer number and prints its binary representation. Print all 32 bits, including leading zeros. Separate groups of 4 bits with apostrophes (').

Additional task: Implement a version of this function that works with signed integers.

Example input:

```
5
```

Example output:

```
0000'0000'0000'0000'0000'0000'0000'0101
```

Part 2. Tests

Study the structure of the test project, make sure you can run it.

Add a new file bitwise_test.cpp to the project. Implement EXPECT_EQ tests for tasks 1-3 from the previous workshop in this file.

1. Task 1, getting a bit at an index.
    i. Several checks for bits equal to 0 and 1. Use binary literals to define numbers for testing.
    ii. Checks for indices 0 and 31 both for numbers where they should be 0 and where they should be 1.
2. Task 2, counting bits equal to 1 in a number.
    i. Checks for a number with 64 ones, 0 ones, some random numbers.
3. Task 3, output.
    i. Rewrite the function to output the number as a string instead of printing it. (If you haven't finished this function during the last workshop, you may reverse a string instead of directly printing it in the reverse order).
    ii. Check 1-2 random numbers.
    iii. Check that numbers which have 1 as their first bit are outputted correctly.
    iv. If you implemented the signed version of this task, check that it works with negative numbers, including ones which have 0 as their second bit.

Part 3.

Exceptions. Reading incorrect data.

Task 3.1. Warnings.

You have a function that reads data and fills empty "ID", "Age", "SibSp" and "Parch" cells with 0. However, there are no checks implemented for some other columns, and if empty data appears in them, the program will not work.

- Try loading data from the file "titanic_empty_fare.csv".
- Fix the program so that it warns the user that the file was corrupted, similar to the existing check.

Task 3.2. Corrupted data.

The original version of the program assumes that if a line was read correctly, it contains a correct number of columns. However, this may not be the case.

- Try loading data from the file "titanic_no_columns.csv".
- Add a check to the cell reading function `extractData` to see if all the cells were read correctly. Give a warning and skip the row, if there was a mistake in that row.

To skip the row, move the line that adds a passenger to the vector to a `try` block like this:

```
        std::stringstream lineStream(buffer);
        try
        {
            Passenger newPass = extractData(lineStream);
            passengers.push_back(newPass);
        }
        catch(std::runtime_error const& ex)
        {
            // warn the user that a row was skipped
        }
```

Task 3.3. Asking the user.

Implement the following logic in your application.

When the program encounters incorrect data, it should ask the user for input. The user has 4 options.

1. Skip the row as corrupt.
2. Fill the row with default data, trying to "repair" it and add it to the vector of passengers.
3. Choose and remember option 1, so the user isn't asked anymore.
4. Choose and remember option 2, so the user isn't asked anymore.

Additional subtask

Add option 5: "Exit the program" by throwing another exception and catching it in `main()`.