Programming Assignment 1, TCSS 333 A, Winter 2021

OBJECTIVE

The objective of this assignment is to give you practice with constructing a program in C (written in Linux/Unix) that deals with the binary operations and representations, and with basic interactive reading and writing.

ASSIGNMENT SUBMISSION

To get credit for this assignment, you must

- ✓ write a program in C that you wrote on your own (no copying or help outside of CSS mentors or TLC or instructor allowed)
- ✓ write a test plan (optional)
- ✓ submit your files through Canvas exactly as instructed (naming, compatibility, etc.)
- ✓ submit your assignment by the due date

INTRODUCTION

Machines process information in fixed-size units called words, with most modern processors typically processing words of 32 or 64 bits. The machine has no insight as to what the bits actually represent and the interpretation is the layer that is put on top of the actual data, e.g. *unsigned int* means the bits should be interpreted using magnitude-only bit model, *int* means that the negative values should be interpreted using two's complement. For our assignment, we will examine a couple of additional ways bits and the 32-bit words could be used or interpreted.

IP and RGB representations

The Internet Protocol (IP) is the principal communications protocol for all devices connected to the Internet. Each device that participates in a computer network that uses IP for communication is identified by its IP address, with most URLs using a numerical label consisting of 32 bits. These 32-bit addresses are canonically represented in dot-decimal notation, which consists of four integer numbers (octets), each ranging from 0 to 255, separated by dots, e.g., 128.208.252.2 (copy and paste this address into the Web browser's address field, if you are curious what this particular location is). When a 32-bit URL is sent, then as bits, the leftmost 8 bits are used for the first octet, the next 8 bits are used for the second one, and so forth, while the dots are not represented (they are implied), e.g.

IP address	10000000 11010000 11111100 00000010	128.208.252.2

In fact, pixel RGB values may follow the same representation scheme, where the first octet is the intensity of the Red hue, the second octet of Green, the third octet of Blue, and the fourth one is used for an alpha channel that defines the degree of transparency (where 0 is fully transparent). By the way, 128.208.252.2 is a shade of almost transparent blue:

RGB-alpha		128.208.252.2

Characters

Characters in ASCII are represented by 7-bits only, so when sending a message of a 32-bit size instead of sending several characters padded with 0s, one could send text more efficiently when 4 subsequent characters are appended together into a 32-bit unit.

Error detection

Error detection is the process of detecting data corruption that may occur during the transmission process. The general idea used in the process is to add some redundancy to the message being sent over some communications channel. Among many techniques used to detect issues, probably the two most basic ones involve the usage of parity bits and the usage of checksums – we will focus here on parity bits. A parity bit is simply a bit added to the message that indicates whether the number of bits in the message with the value of 1 is even or odd. Let's assume odd parity encoding, in which case the number of 1s in the message should always be odd. Basically, we count the number of 1s in the original data, and if the result is an odd number, then we append a 0 at either end of the message (we will append in front) as we already have an odd number of 1s; else we append a 1 to make the total number of 1s an odd number. We will use an 8-bit model for our examples:

original data (7 bits)	**count of 1 bits**	**odd parity 8-bit message**

0000000	0	10000000
1010001	3	01010001
1000001	2	11000001

When a receiver gets the message, they know that the leftmost bit is used for parity purposes only. They count the number of 1s and if that number happens to be an odd number, then they assume the transmission went well and they decode the remaining bits (in our case 7 bits are the actual message). If, however, the receiver finds the even number of 1s, then they know the error occurred and may ask the sender to send the message again.

Data obfuscation and filtering

When considering a piece of data consisting of 32-bits one could use a technique to obfuscate data, where specific bits are turned to 0s for the transmission purposes, so that if anyone intercepts the transmission, their data is malformed, as only the sender and the actual receiver know the obfuscation scheme. The same scheme could be used when filtering information. For example, if we have an RGB value in which we want to remove all red color, we simply need to turn all red bits to 0s (using a mask and bitwise operators).

There are many more uses of bit-level knowledge but we will stop right here.

PROBLEM STATEMENT

Create a program that uses interactive input to explore some of the operations described above using primitive data types and bitwise operators only (i.e. arrays and strings are NOT allowed). The program should start by giving the user a choice as to which operation they are interested in, by providing a list of numerical options (use *short* data type for this).

If a user selects 1, then it means that a person is interested in knowing how a 32-bit integer would be understood as an IP address / RGB value:

a) prompt for the integer to be entered and read it into an unsigned int (assume the user will enter a valid integer)
b) echo print the number
c) print the binary representation of the number
d) use the bitwise operations and the masking to put the 32 bits you have just received into 4 unsigned chars
e) print the chars as positive numbers with dots in-between octets in the proper order

If a user selects 2, then it means that a person is interested in knowing how 4 characters could be appended into a 32-bit entity:

a) prompt for 4 printable characters (assume the user will enter 4 chars without any spaces and will make no mistakes)
b) echo print the numerical value of each character
c) print the binary representations of each character
d) use the bitwise operations and the masking to put these 4 characters into a 32-bit unit in the same order they were typed in, e.g. if ABCD was typed in, then the first 8 bits of a 32-bit unit are ASCII for A, next 8 bits are ASCII for B, etc.
e) print the resulting value as an unsigned decimal and its binary representation

If a user selects 3, then it means that a person is interested in knowing a parity of a character and adjusting it to an odd one, if needed:

a) prompt for the character to be entered and read it in (assume the user will enter a valid char)
b) echo print the character as ASCII number
c) print the binary representation of the character
d) calculate the number of 1 bits and print a message indicating the count
e) make the character follow the odd-parity logic by modifying the leftmost bit, if necessary
f) print the resulting character in unsigned format and its binary representation

If a user selects 4, then it means a person is interested in filtering a 32-bit unit:

a) prompt for the integer to be entered and read it into an unsigned int (assume the user will enter a valid integer)
b) echo print the number
c) print the binary representation of the number
d) prompt for the value that will denote which bit to flip to a 0 (where 1 represents the rightmost bit and 32 the leftmost bit)
    a. if a user enters a bit value > 32 or < 0, display an error message and prompt again until a valid value is entered
    b. if a user enters a 0, quit the filtering subsection
e) change the original value by making the required bit a zero
f) print the result in both unsigned decimal and binary formats

If a user enters an integer value different than 1-4 for a menu selection, display error message and prompt again for a valid choice until one is entered. Once the particular section is processed, the programs prompts the user, if they want to repeat the program (use lowercase *r* as the indicator) and repeats the process as required. Sample program run is given at the end of the description.

Bit printing function
We have not discussed the usage of functions yet and you can write your program as one big chunk of code. However, since you will be printing a bit representation over and over again and the code for doing so was discussed in class, you can put the code into a function as in:

```
void bitPriting (int value)  {  // if you want to pass the size in bytes, then it could have int size as the second param)
    // function body
}
```

Write this function above main – although this is not an entirely correct syntactical approach, it should work for this assignment.

Specs

- Your program has to use the same choice values and follow the same order of operations as shown in the sample run above, although your actual prompt wording could be different
- You have to use bitwise operators in your program
- You are only allowed to use basic primitive types, i.e. arrays and strings are NOT allowed
- Your program is to be written as a single c file named ***pr1.c***
- **Your program must compile in gcc gnu 90 – programs that do not compile will receive a grade of 0**
- If you use a math library, then you will need to compile your program with the following command:
  `gcc -std=gnu90 pr1.c -lm`
- Your program has to follow basic stylistic features, such as proper indentation (use whitespaces, not tabs), whitespaces, meaningful variable names, etc.
- Your program should include the following comments:
    - Your name at the top
    - Whether you tested your code on the cssgate server or Ubuntu 16.04 LTS Desktop 32-bit
    - Comments explaining your logic

TEST PLAN
A test plan is optional but if you submit one, you will be showing us how you tested your code and how it ran for you – this may prove helpful while grading. A test plan in this class is a document that shows the test cases you performed on the final version of your program (values entered to test the outcomes). A test plan in this case should include full code coverage, should list the actual values you entered to test your program, and should be submitted as the following table, in a pdf document named ***test.pdf***:

Test case	Reason	Expected Outcome
3	Testing the menu for the parity selection	Prompts that ask for a value and ultimately print parity
ABCD	Testing the char to int operations	1,094,861,636

		01000001 01000010 01000011 01000100
…		

GRADING
Remember, the programming assignments are graded as pass / no pass only and 85 points (out of 100 are needed to pass). If you do not pass this assignment, you can turn it in again with assignment 2, 3, 4, or 5.
Each user selection implemented according to instructions and running perfectly constitutes 15 points
Proper coding style, meaningful identifiers, etc. is worth 15 points
Program that runs properly (menus displayed, selections read it, prompts and reading and writing functioning, loops implemented) is worth 25 points

PROGRAM SUBMISSION

On or before the due date for assignment 1, use the link posted in *Canvas* next to *Programming Assignment 1* to submit your C code and the test plan as a pdf file (if you choose to write one). Make sure you know how to do that before the due date since late assignments will not be graded until the next grading period. Valid program format: a single file named *pr1.c*

SAMPLE RUN
you are to follow the format shown in this sample run, where bold indicates user input:

What type of display do you want?
Enter 1 for IP/RGB, 2 for character concatenation, 3 for character parity, 4 for filtering : ***5***
Invalid input, try again

What type of display do you want?
Enter 1 for IP/RGB, 2 for character concatenation, 3 for character parity, 4 for filtering : ***1***

Enter a positive integer for the IP/RGB value: ***2161179650***
You entered: 2161179650, Bit representation: 10000000 11010000 11111100 00000010
The dot-decimal represented by this number is: 128.208.252.2

Enter r to repeat, and anything to quit: ***r***

What type of display do you want?
Enter 1 for IP/RGB, 2 for character concatenation, 3 for character parity, 4 for filtering : ***2***

Enter 4 characters: ***ABCD***
65, Bit representation: 01000001
66, Bit representation: 01000010
67, Bit representation: 01000011
68, Bit representation: 01000100
32-bit concatenation is: 1094861636, Bit representation: 01000001 01000010 01000011 01000100

Enter r to repeat, and anything to quit: ***r***

What type of display do you want?
Enter 1 for IP/RGB, 2 for character concatenation, 3 for character parity, 4 for filtering : ***3***

Enter a character for parity calculation: ***Z***
Character: 90, Bit representation: 01011010
Number of ones: 4
Odd 1 parity for the character is: 4294967258, Bit representation: 11011010

Enter r to repeat, and anything to quit: ***r***

What type of display do you want?
Enter 1 for IP/RGB, 2 for character concatenation, 3 for character parity, 4 for filtering : ***4***

Enter an integer to filter: ***16778240***
You entered: 16778240, Bit representation: 00000001 00000000 00000100 00000000
Enter which bit you want to mute (enter 0 to quit): 25
The new value is: 1024, Bit representation: 00000000 00000000 00000100 00000000
Enter which bit you want to mute (enter 0 to quit): 11
The new value is: 0, Bit representation: 00000000 00000000 00000000 00000000
Enter which bit you want to mute (enter 0 to quit): 50
Invalid bit, try again
Enter which bit you want to mute (enter 0 to quit): 0

Enter r to repeat, and anything to quit: ***q***