CS 420
PROGRAM 5P

Purpose

The purpose of these program is to gain familiarity with bitwise operations and structures, writing functions, unit testing, and working with makefiles.

Only one makefile should be submitted containing targets which work for each program as specified below.

Program 1

Details

Files to be submitted: **bitFunctions.c, bitTest.c, makefile**

Write the following functions all of which use bitwise operations. The signature (function arguments, return value) of each function is available in the header file bitFunctions.h which needs to be included in your files. This header file is available to you in the public folder.

(1) circular_shift_left()

- This function should left-shift by n positions, where the high-order bits are reintroduced as the low-order bits.
- Here are two examples of a circular shift operation shown for an unsigned integer:
  10000000 00000000 00000000 00000001 circular shift 1 yields:
  00000000 00000000 00000000 00000011

  11000000 11110000 00001111 00001011 circular shift 3 yields:
  00000111 10000000 01111000 01011110

(2) reverse_it()

- This function that will reverse the bit representation of an unsigned integer i.e. values of bits 0 and 31 will be swapped with each other, values of bits 1 and 30 will be swapped and so on (bits 15 & 16 will be swapped with each other).
- Here is an example of a reverse operation shown for an unsigned integer:

11000000 00000000 00000001 00000001 reversing it yields:
10000000 10000000 00000000 00000011

(3) every_other_bit()

- This function that will extract every other bit of a 32-bit number starting at bit 0, and returns a 16-bit number with the extracted even numbered bits of the original value
- Here is an example of extracting every other bit shown for an unsigned integer:

11000000 00000000 00000001 00000001 extracting even numbered bits (from bit 0) yields:
10000000 00010001

For functions (4) and (5) here's the background:

> *A twentieth century date can be written with integers in the form day/month/year. An example is 1/7/33, which represents 1 July 1933. Write a function that stores the day, month, and year compactly. Since we need 31 values for the day, 12 different values for the month, and 100 values for the year, we can use five bits to represent the day, four bits to represent the month, and seven bits to represent the year. Your function should take as input the day, month, and year as integers, and it should return the date packed into a 16-bit integer. Write another function that does the unpacking.*

(4) date_to_binary()

- This function that will pack the date into a 16-bit value (least significant 7 bits represent year, most significant 5 bits represent day, middle 4 bits represent month)
- Here is an example of date packing:
  Date 12/7/80 will be packed and returned as: 0110001111010000 → 0x63d0

(5) date_from_binary()

- This function that will unpack the 16-bit binary date value (bit pattern as follows: least significant 7 bits represent year, most significant 5 bits represent day, middle 4 bits represent month) and returns individual values for day, month and year
- Here is an example of date unpacking:
  Hex value 0x1184 which represents a binary value 0001000110000100 returns 2 for day, 3 for month, 4 for year

All your functions must be defined in bitFunctions.c. The header file bitFunctions.h contains the prototype declarations. bitTest.c contains your main() function and must only include bitFunctions.h (not bitFunctions.c).Your test driver must be submitted as bitTest.c and must include all the tests you write. You can use assert statements to show me that your tests are working correctly, or you can choose to use any other mechanism you wish. You can either write comments or submit a readme file to describe how you are testing your implementation. A portion of your grade is set aside for your test driver.

The makefile should have the following targets:
bitTest – to build your test driver bitTest.c with the functions defined in bitFunctions.c

bitTest.o – to build and create just an object file (.o file) after compiling your bitTest.c module

bitFunctions.o – to build and create just an object file (.o file) after compiling your bitFunctions.c module

clean – delete all object and executable files

Source file samples

```c
//this is bitFunctions.c
#include "bitfunctions.h"

unsigned int circular_shift_left(unsigned int value, int n) {
    ..............
    ..............
}
```

```c
//this is bitTest.c

#include <stdio.h>
#include <assert.h>
#include <stdlib.h>
#include "bitFunctions.h"

int main(int argc, char *argv[]) {

    ................
    ................

}
```

```
#makefile example
CXX=gcc -Wall -std=c99

bitFunctions.o: bitFunctions.c bitFunctions.h
    $(CXX) -c bitFunctions.c
```

Program 2

Files to be submitted: **ipAddress.c, ipAddress.h**

Details

You are given a data file that has several records, each record contains 2 items: an "IP address" and a corresponding "Subnet mask" both in octet dot notation. Scroll down to the bottom of this document if you want to know details about what they mean which will help you in understanding how to come up with the logic for your program.

You need to read each record into an array of structures set up to hold these two items (as character strings of appropriate sizes). You will then perform computations on these two items to obtain the following (each of these will need a placeholder in your structure along with the two octet dot notation strings coming from the input file):

1) 32-bit numeric value of IP address stored with each octet in a byte (unsigned integer)
2) 32-bit numeric value of subnet mask stored with each octet in a byte (unsigned integer)
3) Which Network Class does the IP address belong to? (a character containing 'A', 'B', or 'C')
4) How many subnets are possible? (a number)

5) How many hosts are possible per subnet? (a number)

- Both IP address and subnet mask items are available as octet numbers with "dot" notation in the input data file.
- An IP address of 192.168.1.10 should be stored in a 32-bit container where each octet value is stored in its corresponding byte position: 192 in the most significant byte, 168 in the second byte, 1 in the third byte and 10 in the least significant byte. 192.168.1.10 value stored in a 32-bit container turns out to be C0A8010A in hex or 3,232,235,786 in decimal.
- Similarly, a subnet mask value of 255.255.255.192 turns out to be stored as FFFFFFC0 in hex or 4,294,967,232 in decimal.
- Network classes are determined by the IP address value. Class A: Range for most significant byte: 0-127 (only first byte used for network addresses); Class B: Range for most significant byte: 128-191 (first two bytes are used for network addresses); Class C: Range for most significant byte: 192-223 (first three bytes are used for network addresses)
- Total number of subnets possible: $2^{\text{number-of-1-bits-in the-host-address portion}}$
- Total number of hosts per subnet possible: $(2^{\text{number-of-0-bits-in the-host-address portion}} - 2)$

For an IP address of 192.168.1.10 if the subnet mask is 255.255.255.192 it belongs to a Class C network, which means we have 8 total bits (only the least significant byte) allowed for creating subnets and hosts. 192_{10} is $C0_{16}$ which is 11000000_2 which means out of the 8 bits allowed, we have 2 one bits and 6 zero bits.
Total number of subnets possible: $2^{\text{number-of-1-bits-in the-host-address portion}}$ which is $2^2 = 4$
Total number of hosts per subnet possible: $(2^{\text{number-of-0-bits-in the-host-address portion}} - 2)$ which is $2^6 = 64-2 = 62$ (we subtract 2 from each subnet since they are reserved)

Other examples:

- IP address192.168.1.0 and subnet mask 255.255.255.252
  results in: Class C network, total subnets possible: $2^6 = 64$ and hosts per subnet $(2^2 – 2) = 2$

- IP address150.150.0.0 and subnet mask 255.255.255.240
  results in: Class B network, total subnets possible: $2^{12} = 4096$ and hosts per subnet $(2^4 – 2) = 14$

Structure to be defined (in ipAddress.h)

Names for all highlighted items must be the same in your file

```
typedef struct {
        ……..
        ……..
} ipInfo_t;
```

This structure must contain the exact same number of items mentioned below with the same names for variables identifying the fields within the structure:

ipAddressDot (character string read from data file)
subnetMaskDot (character string read from data file)
ipAddress (32-bit container – numeric value)
subnetMask (32-bit container – numeric value)
networkClass (character)
totalSubnets (numeric value)
totalHosts (numeric value)

Constant to be defined (in ipAddress.h)

```
#define MAX_RECORDS 100 //maximum records read from the input data file
```

Function prototype declarations needed (in ipAddress.h)

(note that array content can be changed within a function by using it like a typical input argument i.e. without the "&" to pass the address, since the address of the first element of the array is what gets passed in)

1) int readData(char [], ipInfo_t []);
(input arguments: filename, array of structures)
Function return value: actual size of array (i.e. number of records read)

This function opens the file whose name is sent as an argument and fills up the array with items for each record it reads from the file. It returns the actual size of the array after the input file is read or a maximum size of 100 was reached.

2) void computeValues(ipInfo_t [], int);
(input arguments array of structures, size of array)

This function looks at each array element of the structure and computes the rest of the values for each record/element with the given octet dot notation for the IP address and Subnet mask values.

3) void printResults(char [], ipInfo_t [], int);
(input arguments: filename, array of structures, size of array)

This function opens the file whose name is sent as an argument in write mode, and prints the records to the file, one array element at a time. Output sample is provided in the public folder.

The above functions are mandatory and must be implemented in ipAddress.c. You do not include a main() function in this file. You can have any other helper function that you wish to use. Since your functions will be tested by my main function you will need to use the same function names and argument types as indicated.

You cannot use the inet library or any related inet functions for conversion. You can use string manipulation functions (like strtok() which extracts a substring from another string, and atoi() which converts a string to an integer).

The makefile used in Program 1 should be updated with the following target:

ipAddress.o – to build and create just an object file (.o file) after compiling your ipAddress.c module

Additional Reading for those who want it! (some of the information above is repeated)

IP address and Subnet mask

An IP address is a unique address that identifies a device on the internet or a local network. It is generally seen as a a unique string of characters that identifies each computer using the Internet Protocol to communicate over a network. A typical address in IPv4 format looks like this: 192.168.1.10 where each numeric value is a octet (8 bits, 1 byte) identifying certain portions of the address stored as a 32-bit number.

Part of the IP address specifies the "network address" and the rest is the "host address" specific to the device connected to the internet. The network address is determined by InterNIC an organization that is responsible for domain name allocations.

Along with an IP address, there is also a subnet mask that splits the IP address into the host and network addresses, thereby defining which part of the IP address belongs to the device and which part belongs to the network. An example of a subnet mask would b 255.255.255.0 which is also a 32-bit number where the 1s in each octet indicates that it is part of the IP address is reserved for the network address and the 0s indicate the part that is available to be used as the host address. Subnet masks are typically enforced by each entity or organization that has a local network for the use of its consumers.

Since the internet must accommodate networks of all sizes, an addressing scheme for a range of networks exists based on how the octets in an IP address are broken down. You can determine based on the three high-order or left-most bits in any given IP address which of the three different classes of networks, A to C, the address falls within (there are actually five classes but we will just work with three).

Network Classes

A class A network number uses the first eight (most significant) bits of the IP address as its "network part." The remaining 24 bits comprise the host part of the IP address. The values assigned to the first byte of class A class A network numbers fall within the range 0-127. Examples: 75.4.10.4, 10.1.20.3

A class B network number uses the most significant16 bits for the network number and 16 bits for host numbers. The first byte of a class B network number is in the range 128-191. The second (most significant byte) covers the range 1- 255 Examples: 129.144.50.56, 191.132.14.10.

Class C network numbers use the most significant 24 bits for the network number and the least significant 8 bits for host numbers. The first byte of a class C network number covers the range 192-223. The second and third (most significant bytes) each cover the range 1- 255. Examples: 192.5.2.5, 220.5.7.1

Subnet masks have 1s in their network address and can have 0s or 1s in the host address part which determines how many subnets we have and how many hosts can be supported per subnet.

For an IP address that is 192.168.1.10 if the subnet mask is 255.255.255.192 we know that it belongs to a Class C network, we have 8 total bits allowed for creating subnets and hosts. 192_{10} is $C0_{16}$ which is 11000000_2 which means out of the 8 bits allowed, we have 2 one bits and 6 zero bits.
Total number of subnets possible: $2^{\text{number-of-1-bits-in the-host-address portion}}$ which is $2^2 = 4$
Total number of hosts per subnet possible: ($2^{\text{number-of-0-bits-in the-host-address portion}}$ - 2) which is $2^6 = 64-2 = 62$
(we subtract 2 from each subnet since they are reserved)

Other examples:

- IP address192.168.1.0 and subnet mask 255.255.255.252
  results in: Class C network, total subnets possible: $2^4 = 16$ and hosts per subnet $(2^4 – 2) = 14$

- IP address150.150.0.0 and subnet mask 255.255.255.240
  results in: Class C network, total subnets possible: $2^{14} = 16384$ and hosts per subnet $(2^2 – 2) = 2$