LAB 5

CS 361: Systems Programming / Spring 2023

Description

In this lab session, you will:

- Practice pointer arithmetic
- Practice bitmasks

Please read this document carefully and follow the instructions on the last section to complete this lab session. When you answered all the questions, please show your work to the TA.

Guide

1. Accept the invitation for Lab 5 on Github classroom: https://classroom.github.com/a/R1npZODk
2. Import the Github repository created to your machine using vscode, as explained in Assignment 0
3. Make sure that you can launch a terminal inside vscode via menus: Terminal > New Terminal
4. Read this guide and answer the questions as they appear. You should answer a total of 6 questions.

Pointer Arithmetic

Basic operations with pointers

The C programming language features pointers, which allow you to navigate addresses in memory and their contents. This is a declaration of a pointer to an integer:

```
int * ptr;
```

The contents of a pointer are a memory address. A program can set a pointer to the memory address of a variable by using the & operator before the name of the variable:

```
int variable = 361;
ptr = &variable;
```

If we inspect the contents of the pointer, we can see that they match the address of the variable:

```
printf("The address of the variable is:\t\t0x%p\n", (void*) &variable);
printf("The contents of the pointer are:\t0x%p\n", (void*) ptr);
```

Question 1: Copy the samples above to file lab5-1.c, type make, and run it. What is the output?

Dereferencing a pointer

Once you have a pointer, you can read and write the memory that the pointer refers to through derefencing via the `*` operator. `*ptr` reads the memory pointed to by the pointer, and `*ptr = <val>` writes `<val>` to the memory pointed by the pointer:

```
printf("The integer pointed to is\t%d\n",  *ptr);
printf("The value of the variable is\t%d\n",  variable);
*ptr = 361361;
printf("The value of the variable is\t%d\n",  variable);
```

Question 2: Copy the samples above to file lab5-1.c, type make, and run it with command ./lab5-1. Does the variable change? Why?

malloc

Pointers are very useful to refer to memory dynamically allocated with `malloc`:

```
int * pointer = malloc(sizeof(int));
```

We can also allocate more memory than a single integer:

```
int * pointer = malloc(10*sizeof(int));
```

`malloc` will allocate a contiguous block of memory 10 times the size of an `int`. This is similar to an array of 10 integers, and C allows you to use array notation to read/write that memory:

```
pointer[5] = 361;
printf("The value is %d\n", pointer[5]);
```

Pointer arithmetic

Instead of using the array notation, you can also perform pointer arithmetic:

```
*(pointer+5) = 361;
printf("The value is %d\n", *(pointer+5));
```

In this case, adding a number to a pointer will advance the pointer the number times the size of the data pointed to. For instance, if an int is 4 bytes long, then `pointer+5` will advance the pointer 20 bytes.

You can also compare pointers by subtracting them, and get the number of elements of the pointer size between two pointers:

```
int * pointer1 = …;
int * pointer2 = pointer1 + 512;
printf("%d\n", pointer2-pointer1); // prints 512
```

Question 3: Change file lab5-2.c to use pointer arithmetic instead of array notation, type make, and run it with command ./lab5-2. When incrementing the pointer by one, how many bytes does the pointer advance?

Hint: You can use the code you copy/pasted for lab5-1.c to print addresses and values of pointers

Hint: The values printed are in hexadecimal, the answer we look for is in decimal.

Casting

The memory address inside a pointer is a number like any other. Casting operations turn a variable `var` into a type `type` and look like `(type) var`. For instance, you can obtain the pointer address as a number by casting it to a sufficiently large integer:

```
int * ptr = … ;
long address = (long) ptr;
```

Conversely, you can turn a number into a pointer by casting:

```
long address = … ;
int * ptr = (int*) address;
```

Combining the two code snippets above, you can obtain a pointer and move it in units of bytes instead of units of the size of the pointer's type:

```
int * ptr = …;
int * ptr2 = ptr + 1; // This pointer is many bytes ahead of ptr

long address = (long) ptr;
address += 1; // Advances the address one byte
int * ptr3 = (int*) address; // This pointer is one byte ahead of ptr
```

Question 4: In file lab5-3.c, how many bytes **in front** the given pointer can you find the value 361? You have to **increment** the pointer one byte at a time to find it. You can assume that the value you are looking for is smaller than 512 bytes. Remember to type make every time you change the file.

Question 5: In file lab5-4.c, how many bytes **behind** the given pointer can you find the value 361? You have to **decrement** the pointer one byte at a time to find it. You can assume that the value you are looking for is smaller than 512 bytes. Remember to type make every time you change the file.

void *

Pointers with type void* are pointers without any type, and they cannot be dereferenced. To dereference a pointer of type void*, you need to cast it first to another pointer that can be dereferenced:

```
void * pointer = … ;
printf("%d\n", *pointer); // Compilation error
*pointer = 361; // Compilation error
printf("%d\n", *((int*)pointer)); // OK, pointer was cast to int*
*((int*)pointer) = 361; // OK, pointer was cast to int*
```

Bitmasks

You can use bitmasks together with bitwise operations to test individual bits, and to set them to 1 or 0.

For instance, all even numbers have the least significant bit set to zero. Odd numbers have that bit set to 1. You can test the least significant bit with the bitmask 1 and the bitwise and operation as follows:

Number 3	0	0	1	1
Bitmask 1	0	0	0	1
&	0	0	0	1
3 & 1 == 1				

Number 10	1	0	1	0
Bitmask 1	0	0	0	1
&	0	0	0	1
10 & 1 == 0				

You can ignore bits by using a bitmask, the bitwise negation operation ~, and the bitwise AND operator &. For instance, the following ignores all bits except the least significant 2 bits (note that 3 is 11 in binary):

Number 11	1	0	1	1
Bitmask 3	0	0	1	1
~3	1	1	0	0
11 & ~3	1	0	0	0
11 & ~3 == 8				

Question 6: In file lab5-5.c what mask can you apply to the number to obtain 361? Remember to type make every time you change the file.

Hint: try powers of 2 (1, 2, 4, 8, 16, …)

Extra / Optional

The types `long` and `int` are not very portable. Header file `stdint.h` defines more portable types:

- `uint32_t`: unsigned integer with 32 bits (i.e., the size of a pointer in 32bit architectures)
  - Note that, on the systems machines: `sizeof(int) == sizeof(uint32_t)`
- `uint64_t`: unsigned integer with 64 bits (i.e., the size of a pointer in 64bit architectures)
  - Note that, on the systems machines: `sizeof(void*) == sizeof(uint64_t)`

Can you rewrite this lab using these two types instead of `int` and `long`, respectively?

Grading

Show your UIC card to the TA when you enter the lab, or type your UIN on the chat when joining remotely. Stay in the session until you show your work, or until the TA announces that the lab is over.

- You have to remain present for the whole lab to get attendance, which you can then use to resubmit Assignment 3.
- You can leave early after showing your work to the TA (answers to all questions). In this case, you will get a 5% bonus in Assignment 3.