CS 24000 - Programming In C

Week Eight: arithmetic and bit operations on chars ; C Programming Tools: GDB, C preprocessor, Makefile

Zhiyuan Li
Department of Computer Science
Purdue University, USA

This week's lectures

- Arithmetic and bit-wise operations on characters
  - related to Lab 6
- Tools for C program development
  - GDB for debugging,
  - C preprocessors, and
  - Makefile

Char input and output

- In Lab 6, we deal with arbitrary files, not just text files.
- A non-text file may be unformatted, and therefore may not be suitable for formatted I/O functions such as scantf() and printf().
- Since we are doing byte-level encryption/decryption in Lab 6, we perform character I/O by either UNIX read/write calls or getchar()/putchar() C standard lib routines

UNIX read/write

- In the first week, we looked at how to do char I/O using the UNIX read() and write() system calls.
- For Lab 6, we can do unbufferred read/write
  - Not efficient, but simple:

```
#include <stdio.h>

main()
{
        char buf[1];
        int n;
        unsigned char code, key = '\x0f';
        while ((n = read(0, buf, 1)) > 0) {
                code = *buf;
                *buf = code ^ key;    /* perform an xor cipher */
                write(1, buf, 1);
        }
        return 0;
}
```

Use getchar() and putchar()

- Alternatively, we can use C standard library functions getchar() putchar()
  - Must be careful with typecast

```
#include <stdio.h>
main()
{
    unsigned char msg, code, key = '\x0f';
    int input;
    while ((input = getchar()) != EOF) {
        msg = (unsigned char) input;
        code = msg ^ key;
        input = (int) code;
        putchar(input);
    }
    return 0;
}
```

Sign Extension

- The main reason for **getchar()** to return an integer (typecast from unsigned char) is because the **EOF** number is greater than what a char can represent
- The leading bits of the typecast integer will be all zeros because the read char is unsigned
  - Note: By default a char is signed. Casting it to **int** will result in sign extension, **‘\xff’** will be come **0xffffffff**

- We want to perform byte-level cipher
  - Therefore we must recast the integer to unsigned char
  - Otherwise, we will be rotating four bytes, e.g.
- Function **putchar()** takes an integer as parameter, hence we recast the unsigned char result to integer, x, before calling **putchar(x)**
  - Again the leading bits will be zero in this integer
  - **Putchar(x)** will automatically convert **x** back to **unsigned char** before writing to the output.

Type Promotion

- The arithmetic unit on the processor hardware operates on integers.
- Therefore to perform *add, subtract,* and other arithmetic operations on bytes
  - The operands are promoted to integers first
  - If the lvalue (i.e. the variable holding the result) in the assignment statement is a **char** **(or unsigned char)**
  - The integer result will be automatically recast to char (or unsigned char)
  - Next, we compare the result of adding unsigned numbers versus signed numbers
    - Pay attention to sign extension and type casting

Adding unsigned chars (byteadd.c)

```
#include <stdio.h>
main() {
	unsigned char ua = 0, ub = -1, ux;
	unsigned char uc = 0, ud =  1, uy;
	int w, z;
	unsigned char ubig = '\xff', uoverflow,
ucast;
	int ioverflow;
/* difference between adding a negative
	byte versus subtracting a positive byte
when writing
	back to an integer */
	ux = ua + ub;
	uy = uc - ud;
	w  = ua + ub;
	z  = uc - ud;
 printf("ux is \t %#X\n", ux);
	printf("uy is \t %#X\n", uy);
	printf("w is \t %#X\n", w);
	printf("z is \t %#X\n", z);

/* The following shows what happens with
'overflow' when adding
	bytes together */

	uoverflow = ubig + ubig;
	ioverflow = ubig + ubig;
	ucast = (unsigned char) ioverflow;
	printf("uoverflow is \t %#X\n", uoverflow);
	printf("ioverflow is \t %#X\n", ioverflow);
	printf("icast is \t %#X\n", ucast);

	return 0;}
```

Results

- ux is 0XFF
- uy is 0XFF
- w is 0XFF
- z is 0XFFFFFFFF
- uoverflow is 0XFE
- ioverflow is 0X1FE
- icast is 0XFE

Adding signed chars (signedadd.c)

```
#include <stdio.h>
main() {
	char a = 0, b = -1, x;
	char c = 0, d =  1, y;
	int w, z;
	char big = '\xff', overflow, bytecast;
	int ioverflow;

	x = a + b;
	y = c - d;
	w  = a + b;
	z  = c - d;

	printf("x is \t %#X\n", x);
	printf("y is \t %#X\n", y);
	printf("w is \t %#X\n", w);
	printf("z is \t %#X\n", z);

/* The following shows what happens with
'overflow' when adding
	bytes together */

	overflow = big + big;
	ioverflow = big + big;
	bytecast = (char) ioverflow;
	printf("overflow is \t %#X\n", overflow);
	printf("ioverflow is \t %#X\n", ioverflow);
	printf("bytecast is \t %#X\n", bytecast);

	return 0;
```

Results

- x is 0XFFFFFFFF
- y is 0XFFFFFFFF
- w is 0XFFFFFFFF
- z is 0XFFFFFFFF
- overflow is 0XFFFFFFFE
- ioverflow is 0XFFFFFFFE
- bytecast is 0XFFFFFFFE

The Unix “od” command

- Let us run command:
  - od -t x1 text
- Displays
  - 0000000 61 62 63 64 65 66 67 0a 0000010
- We see that the seven letters are displayed seven chars in hexadecimal representation: 61 to 67.

Text:

abcdefg

- Next, we will examine the “od” display of the result of bit-wise xor, char addition, and bit-rotation
  - These are three cipher operations used in Lab 6

Review the example on bit rotation

```
/* Purpose: showing result of bit
rotation */
#include <stdio.h>

main() {

    char a = '\x0f';
    char b, c;

    unsigned char ua = '\x0f';
    unsigned char ub, uc;

    printf("a is \t %#X\n", a);

    b = a >> 2;
    printf("b is \t %#X\n", b);
    c = a << 6;
    printf("c is \t %#X\n", c);
    a = b | c;
    printf("a is \t %#X\n", a);
    printf("ua is \t %#X\n", ua);
    ub = ua >> 2;
    printf("ub is \t %#X\n", ub);
    uc = ua << 6;
    printf("uc is \t %#X\n", uc);
    ua = ub | uc;
    printf("ua is \t %#X\n", ua);
    ua = '\x0f';
    ua = ua >> 2 | ua << 6;
printf("rotation of ua is \t%#X\n",  (unsigned
char) ua >> 2 | ua << 6);
}
```