Bitwise Operations

Section 2.2

Introduction

- A single bit can represent a binary (or Boolean) value
  - On / off, yes / no, true / false, etc.
- But in C, a char (1 byte) is the smallest data type
- Could use an entire byte to represent such a state
  - Wasteful!

- Bitwise operations allow us to work with individual bits within a byte
- For example, each bit in char might be used to represent some on / off state
- Great for when memory is limited
  - Embedded environments

Binary Logic

- Familiar with AND, OR, and NOT logical operators for Boolean values
- C has logical operators for both Boolean expressions and bitwise operators
- Conceptually, both are the same, though their usage differs

AND

- 0 AND 0 = 0
- 0 AND 1 = 0
- 1 AND 0 = 0
- 1 AND 1 = 1

OR

- 0 OR 0 = 0
- 0 OR 1 = 1
- 1 OR 0 = 1
- 1 OR 1 = 1

XOR

- 0 XOR 0 = 0
- 0 XOR 1 = 1
- 1 XOR 0 = 1
- 1 XOR 1 = 0

NOT

- NOT 0 = 1
- NOT 1 = 0

Bit Operators

- C has
  - Four logical bitwise operators
  - Two shift operators
- Primarily intended for use on integer data types

Operator	Action
~	bitwise NOT
&	bitwise AND
|	bitwise OR
^	bitwise XOR (eXclusive OR)
>>	right shift
<<	left shift

Bitwise NOT

- Bitwise NOT symbol is ~
- Unary operator (operates on one thing)
- Flips all the bits in the variable
  - All 0s become 1s
  - All 1s become 0s

```
unsigned char a;
a = 17;
a = ~a;
printf("%d\n", a);
```

Code	Binary	Decimal
a = 17	**0 0 0 1 0 0 0 1**	17
a = ~a	**1 1 1 0 1 1 1 0**	238

Bitwise &

- Symbol for bitwise AND is **&**
  - && in C is the Boolean logical AND
- Binary operators (operates on two things)
- Performs an AND on the bits in two variables
  - First bit in both variables are ANDed together, second bit in both variables are ANDed together, etc.

```
unsigned char a;
unsigned char b;
a = 17;
b = 22;
a = a & b;
printf("%d\n", a);
```

Code	Binary	Decimal
a = 17	**0 0 0 1 0 0 0 1**	17
b = 22	**0 0 0 1 0 1 1 0**	22
a = a & b	**0 0 0 1 0 0 0 0**	16

Bitwise OR

- Symbol for bitwise OR is |
- Binary operators (operates on two things)
- Performs an OR on the bits in two variables
  - First bit in both variables are ORed together, second bit in both variables are ORed together, etc.

```
unsigned char a;
unsigned char b;
a = 17;
b = 22;
a = a | b;
printf("%d\n", a);
```

Code	Binary	Decimal
a = 17	**0 0 0 1 0 0 0 1**	17
b = 22	**0 0 0 1 0 1 1 0**	22
a = a | b	**0 0 0 1 0 1 1 1**	23

- Operands in bitwise operations don't have to be variables
- Can be constants

```
char x;
char y;
x = 7;
y = 6;
x = x & y;
y = x | 16;
printf("%d %d\n", x, y);
```

Code	Binary	Decimal
x = 7	**0 0 0 0 0 1 1 1**	7
y = 6	**0 0 0 0 0 1 1 0**	6
x = x & y	**0 0 0 0 0 1 1 0**	6
16	**0 0 0 1 0 0 0 0**	16
y = x | 16	**0 0 0 1 0 1 1 0**	22

Shift Operations

- Left shift moves bits into higher bit positions
  - Left shift symbol is **<<**
- Right shift moves bits into lower bit positions
  - Right shift symbol is >>

```
unsigned char a;
unsigned char b;
a = 17;
a = a << 2;
b = 64;
b = b >> 3;
printf("%d %d\n", a, b);
```

Code	Binary	Decimal
a = 17	**0 0 0 1 0 0 0 1**	17
a = a << 2	**0 1 0 0 0 1 0 0**	68
b = 64	**0 1 0 0 0 0 0 0**	64
b = b >> 3	**0 0 0 0 1 0 0 0**	8

- Sometimes shifting is used to efficiently multiply by 2 and to do integer division by 2 in magnitude only bit models
  - For a lot of CPUs, shifting corresponds to more efficient instructions than multiply and divide
- Left shifting is equivalent to multiplying by 2
- Right shifting is equivalent to integer division by 2

```
unsigned char a;
unsigned char b;
a = 17;
a = a << 1;
b = 64;
b = b >> 1;
printf("%d %d\n", a, b);
```

Code	Binary	Decimal
a = 17	**0 0 0 1 0 0 0 1**	17
a = a << 1	**0 0 1 0 0 0 1 0**	34
b = 64	**0 1 0 0 0 0 0 0**	64
b = b >> 1	**0 0 1 0 0 0 0 0**	32

- What happens with bits that are shifted off the edge depends on the bit model being used
  - For **magnitude only bit model**, new values added always **0**s
  - For **two's complement**, new values are
    - **0**s for left shifts
    - Copy of highest order bit for right shift
      - Maintains the sign

```
char a;
char b;
a = 17;
a = a >> 2;
b = -65;
b = b >> 2;
printf("%d %d\n", a, b);
```

Code	Binary	Decimal
a = 17	**0 0 0 1 0 0 0 1**	17
a = a >> 2	**0 0 0 0 0 1 0 0**	4
b = -65	**1 0 1 1 1 1 1 1**	-65
b = b >> 2	**1 1 1 0 1 1 1 1**	-17

Bitmask Operations

- **Bitmasks** are ways to refer to or access only specific bits in a variable
- For example,
  - Only change a particular bit in a variable
  - Or only retrieve a particular bit in a variable

- **Setting** a bit refers to giving it a value of 1
- **Clearing** a bit refers to giving it a value of 0
- **Reading** or getting a bit refers to accessing, but not modifying the bit
- Usually bitmasks have 0s in bit positions we don't care about and 1s in bit positions we do

Decimal	Bitmask	Bits Indicated
1	**0 0 0 0 0 0 0 1**	bit 0
16	**0 0 0 1 0 0 0 0**	bit 4
172	**1 0 1 0 1 1 0 0**	bits 2, 3, 5, and 7

- Most often interested on a single bit in a variable
- Set, clear, and read operations can be implemented logically..

Operation	Logic
Set Nth Bit	**x = x OR 2^N**
Clear Nth Bit	**x = x AND NOT(2^N)**
Read Nth Bit	**= x AND 2^N**

- In C code these could be implemented as..

Operation	Logic
Set Nth Bit	**x = x | (1 << N);**
Clear Nth Bit	**x = x & (~(1 << N));**
Read Nth Bit	**(x & (1 << N)) >> N**

```
char a;
int i;
a = 17;
a = a | (1 << 3);
printf("%d\n", a);

a = a & (~(1 << 4));
printf("%d\n", a);

for (i = 7; i >= 0; i--)
{
    printf("%d ", (a & (1 << i)) >> i);
}
```

Code	Binary	Decimal
a = 17	**0 0 0 1 0 0 0 1**	17
a = a | (1 << 3)	**0 0 0 1 1 0 0 1**	25
a = a & (~(1 << 4))	**0 0 0 0 1 0 0 1**	9