Bitwise Operations

- In some situations, a single bit can store all the information needed for a computation.
- For example, a file can be write-protected, or not. A user can be logged in, or not. A program can be currently running, or not.
- In many advanced algorithms, bits are frequently used in computations and data structures. For example, compression/decompression routines (codecs) commonly manipulate bit patterns to reduce the necessary storage size. Sorting methods commonly use trees and other binary structures during operation. Bitwise operations are also common in device drivers and graphics programming.
- In the C language, the smallest data type available is the char, which is 1 byte (8 bits). To manipulate a single bit, you can
  - use an entire char to store the single bit. This is of course wasteful and, if used indiscriminately, would seriously raise the amount of memory necessary to operate a modern computing system.
  - figure out how to manipulate just 1 bit within a char, or within any other data type. In the C language, this is accomplished using bitwise operations.

Binary Logic Operations

- a single bit can have a value of either 1 or 0, which can also represent true or false.
- The three most basic logic operations are AND, OR, and NOT. The operations work as follows:

AND	OR	NOT
0 AND 0 = 0	0 OR 0 = 0	NOT 0 = 1
0 AND 1 = 0	0 OR 1 = 1	NOT 1 = 0
1 AND 0 = 0	1 OR 0 = 1	
1 AND 1 = 1	1 OR 1 = 1	

- The result of an AND operation is true only if both input values are true.
- The result of an OR operation is true if either input value is true.
- The result of a NOT operation is to reverse the bit value.

- Logic operations are independent of any particular programming language used to implement them.
- There are several other logic operations, for example NOR and NAND, which are commonly used in circuits.

Bit Operators

- Bit operators are the symbols or syntax used in a programming language to affect individual bits within a variable.
- In the C programming language, there are six bit operators:

Operator Symbol	name	Action
~	tilde	bitwise NOT
&	ampersand	bitwise AND
|	vertical bar	bitwise OR
^	caret bitwise	XOR
>>	greater-than greater-than	right-shift
<<	less-than less-than	left-shift

- The bit operators are intended to be used only on the whole number data types, char and int, and the related extensions (e.g., unsigned char).
- Although bits can be manipulated in any variable, it is rare to see bit operators applied to the real number data types.

- The bitwise NOT operator inverts every bit in a variable. It is written using the tilde symbol (~). For example:

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

- The result of executing this code is:

  238

- At the bit level, the result of the code is:

C code	**Bits in variable** a	**Base 10 value**
a=17;	0 0 0 1 0 0 0 1	17

- The bitwise AND operator performs an AND between two variables, independently at every bit. It is written using the ampersand symbol (&). For example:

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

- The result of executing this code is:

16

- At the bit level, the result of the code is:

C code	**Bits in variable** a	**Base 10 value**
a=17;	0 0 0 1 0 0 0 1	17
b=22;	0 0 0 1 0 1 1 0	22

- NOTE:Prior to the bitwise AND, only one bit position had a value of 1 in both variables a and b. Therefore, after the bitwise AND, this is the only bit position with a value of 1.

- The bitwise OR operator performs an OR between two variables, independently at every bit. It is written using the vertical bar symbol (|). For example:

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

- The result of executing this code is:

  23

- At the bit level, the result of the code is:

C code	**Bits in variable** a	**Base 10 value**
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

NOTE: After the bitwise OR, the variable a has a value of 1 in any bit position in which either of the two input variables a or b had a value of 1.

- Like the arithmetic or logic operators in C, the bit operators can be applied to either variables or constants. For example:

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

- The result of executing this code is:

6 22

- The left-shift and right-shift bit operators move bits into higher-order and lower-order bit positions, respectively.
- A left-shift is written in C using two consecutive less-than symbols (<<) and a right-shift is written using two consecutive greater-than symbols (>>).
- The value following the shift operator indicates how many bit positions to move.
- For example:

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

- The result of executing this code is:

  68 8

- Any bits that move beyond the highest or lowest available bit are discarded.

- The new bit values that take up residence in the now vacated bit positions are given values that depend upon the bit model used by the variable.
- For a magnitude-only bit model, the new bit values are always zero, as shown above.
- For a two's complement bit model, the new bit values are zero for left-shifts, and copies of the original highest-order bit for right-shifts.
- The latter maintains the original sign of the value, while shifting the negative number in a manner synonymous with positive numbers. For example:

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

The result of executing this code is:

4 -17

- At the bit level, the result of the code is:

C code	**Bits in variable** a	**Base 10 value**
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

- Bitmasking is perhaps the most common type of bitwise operation.
- It involves using a *bitmask* to change or query one or more designated bits within a variable.
- The bitmask indicates which bits are to be affected by the operation. The idea is to operate on a variable, changing or affecting only the bits indicated by the bitmask:

variable→bitmask (bit N)→variable (only bit N changed)

- The bitmask indicated that bit N should be changed (N is the bit position, which is also the power of 2 of the bit).

- The following are examples of bitmasks for an 8-bit variable (e.g., unsigned char):

Bitmask	Base 10 value	Indicated bits to work on
0 0 0 0 0 0 0 1	1	bit 0
0 0 0 1 0 0 0 0	16	bit 4
1 0 1 0 1 1 0 0	172	bits 2, 3, 5, and 7

- A bitmask can indicate that any number of bits are to be affected.
- The three most common bitmask operations work on a single bit: set the bit, clear the bit, or query the value of the bit. Each of these operations can be accomplished through the following logic:

Operation	Logic
set Nth bit	$x = x \text{ OR } 2^N$
clear Nth bit	$x = x \text{ AND NOT}(2^N)$
read Nth bit	$= x \text{ AND } 2^N$

- The act of setting a bit gives the bit a value of 1, regardless of its initial value. it leaves all the other bits unchanged.
- The act of clearing a bit gives the bit a value of 0, regardless of its initial value, and leaves all the other bits unchanged.
- The act of querying a bit determines the current value of a bit, leaving all bit values unchanged.

- Each of these operations can be implemented in C as follows:

Operation	**C code**
set Nth bit	x = x | (1<<N);
clear Nth bit	x = x & (~(1<<N));
read Nth bit	(x & (1<<N)) >>N

- When reading the Nth bit, the final right-shift operation results in a value of 1 or 0 regardless of which bit is being read. Without that final right-shift, the read value will be equal to the value of the bit position (the power of 2 of the bit place).

- The following code demonstrates setting, clearing, and reading bits:

```
char a;
int i;
a=17;
a=a | (1 << 3); /* set 3rd bit */
printf("%d\n",a);
a=a & (~(1<<4)); /* clear 4th bit */
printf("%d\n",a);
for (i=7; i>=0; i--)
printf("%d ",(a&(1<<i)) >> i); /* read i'th bit */
printf("\n");
```

- The result of executing this code is:

```
25
9
0 0 0 0 1 0 0 1
```

- At the bit level, the result of the code is:

C code	**Operation**	**Bits in variable** a	**Base 10 value**
a=17;		0 0 0 1 0 0 0 1	17
a=a | (1 << 3);	set bit 3	0 0 0 1 1 0 0 1	25
a=a & (~(1 << 4));	clear bit 4	0 0 0 0 1 0 0 1	9