Lecture 8
Bitwise Operations

Dr. Hacer Yalım Keleş

Ref: Programming in ANSI C, by Kumar

Operations on Bits

C provides four operators, called *bitwise logical operators*, for masking operations on bits, and two operators, called the *bitwise shift operators*, for shifting bits in a word. In addition, C allows you to partition a word into groups of bits, called *bit-fields*, and assign names to them.

Bitwise AND Operator

The bitwise AND operator & has the formation

intvalue & *intvalue*

When it is applied to two integral operands, the binary representations of the converted values of the operands are compared bit by bit. If *bl* and *b2* represent corresponding bits of the two operands, then the result of *bl* & *b2* is as shown in the following truth table:

b1	*b2*	*b1'& b2*
1	1	1
1	0	0
0	1	0
0	0	0

For example, given that

unsigned int el = Oxd, e2 = 0x7;

and that an integer is represented in 16 bits in the machine being used, the expression el & e2 has the value 0x5 as shown below:

Expression	Binary Representation	Value
el	0000 0000 0000 1101	Oxd
e2	0000 0000 0000 0111	0x7
el & e2	0000 0000 0000 0101	0x5

Bitwise & op. is often used to mask off some set of bits:

n = n & 0177

sets to zero all but the low order 7 bits of n

Bitwise **OR** operator **|** is used to turn bits on:

x = x | SET_ON

sets to obe in x the bits that are set to 1 in SET_ON.

1 | 1 = 1
1 | 0 = 1
0 | 1 = 1
0 | 0 = 0

One must distinguish the bitwise operators & and | from the logical operators && and || , which imply left to right evaluation of a truth value.

For example: if x is 1 and y is 2, then
x & y is 0, while x && y is 1.

Bitwise **exclusive OR** operator ^ sets a one in each bit position where its operamds have different bits, and zero where they are the same

1 ^ 1 = 0
0 ^ 0 = 0
1 ^ 0 = 1
0 ^ 1 = 1

The bitwise complement operator ~ is a unary operator, which yields the one's complement of an integer; that is, it converts each 1-bit into a 0 bit and vice-versa.

For example:

```
x = x & ~077
```

sets the last siz bits of x to 0.

Precedence and Associativity

Order: ~ > & > ^ > |

For example:

01|~01^01&01

is interpreted as:

01|((~01)^(01&01))

Precedence and Associativity

Except for the bitwise complement that associates from right to left, all others associate from left to right.

Note that: the precedence of the binary bitwise operators is lower than the equality operator == and the inequality operator !=. Thus use paranthesis for expressions such as:

(i&01)== 0 or (i|01)!=0

Left Shift Operator

The left shift operator shifts bits to the left, and has the formation

intvalue « intvalue

As bits are shifted toward high-order positions, 0 bits enter the low-order positions. Bits shifted out through the high-order position are lost. For example, given

```
unsigned int i = 5;
```

and 16-bit integers, that is,

i is binary 00000000 00000101,

then

i « 1 is binary 00000000 00001010, or decimal 10,

and

i << 15 is binary 10000000 00000000, or decimal 32768.

In the second example, the 1 originally in the third bit position has dropped off. Another left shift by one position will drop off the 1 in the sixteenth bit position, and the value of the expression will become zero.

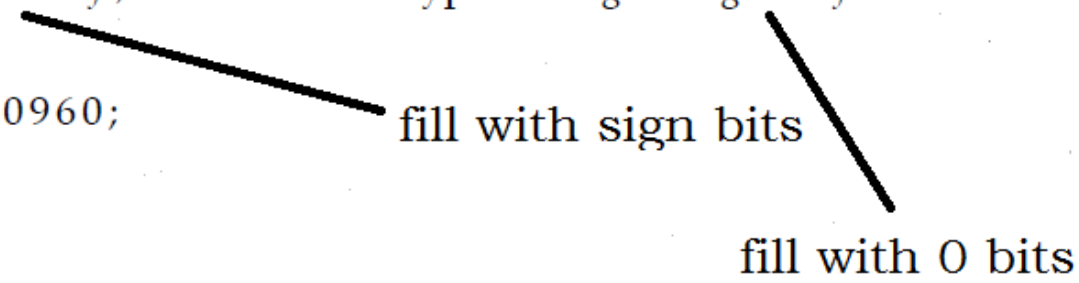
Right Shift Operator

The right shift operator shifts bits to the right, and has the formation

intvalue » *intvalue*

As bits are shifted toward low-order positions, 0 bits enter the high-order positions, if the data is `unsigned`. If the data is signed and the sign bit is 0, then 0 bits also enter the high-order positions. However, if the sign bit is 1, the bits entering high-order positions are implementation-dependent. On some machines Is, and on others Os, are shifted in. The former type of operation is known as the *arithmetic right shift*, and the latter type the *logical right shift*. For example, given

```
unsigned int i = 40960;
```



and 16-bit integers, that is,

i is 10100000 00000000,

then

i » 1 is binary 01010000 00000000, or decimal 20480,

and

i >> 15 is binary 00000000 00000001, or decimal 1.

Multiplication and Division

The left shift of a value by one position has the effect of multiplying the value by two, unless an overflow occurs due to a 1 falling off from the high-order position. Similarly, the right shift of a value by one position has the effect of dividing the value by two, provided the value is nonnegative.

`unsigned int i`	i as each statement executes	
	Binary Representation	**Decimal Value**
`i .`	00000000 00000011	3
`i « 1`	00000000 00000110	6
`i << 4`	00000000 01100000	96
`i « 9`	11000000 00000000	49152
`i << 1`	1000000000000000	32768
`i » 1`	01000000 00000000	16384
`i >> 9`	00000000 00100000	32
`i » 4`	00000000 00000010	2
`i » 1`	00000000 00000001	1
`i » 1`	00000000 00000000	0

Precedence and Associativity

Left and right shift operators have equal precedence and they associate from left to right. Thus, the expression

```
1 « 1 » 2
```

is interpreted as

```
. (1 « 1) » 2 .
```

Precedence and Associativity

The precedence of the shift operators is lower than that of any arithmetic operator, but higher than that of any bitwise logical Operator except the unary bitwise complement operator. Thus, the expression

```
1 « 2 - 1
```

is interpreted as

```
1 « (2 - 1)
```

and the expression

```
01 | ~ 01 « 1
```

is interpreted as

```
01 | ( ( ~ 01 ) « 1 ) .
```

Write a function getbits(x,p,n) that returns the (right adjusted) n-bit field of x that begins at position p.

Assume bit position 0 is at the right end. ex: getbits(x,4,3) returns the 3 bits in posn 4,3 and 2. right adjusted.

Write a function getbits(x,p,n) that returns the (right adjusted) n-bit field of x that begins at position p.

Assume bit position 0 is at the right end. ex: getbits(x,4,3) returns the 3 bits in posn 4,3 and 2. right adjusted.

```
unsigned getbits(unsigned x, int p, int n)
{
    return (x>>(p-n+1)) & ~(~0<<n);
}
```

Bit Fields

C provides a more convenient method for defining and accessing fields within a word than the use of the bit-manipulation operators. This method uses a special syntax in the structure definition to define bit-fields and assign names to them. A *bit-field* is a set of adjacent bits within an implementation-dependent storage unit, called a "word." The syntax for defining a bit-field is

type field-name : bit-length;

where *bit-length* is the number of bits assigned to the bit-field variable *field-name* of the type *type*. Bit-fields can only be of type `int`; for portability, they should explicitly be specified to be `signed` or `unsigned`. Thus, the variable `ax`, containing the encoding of the equipment list, can be defined using bit-fields as

```
struct
  {
    unsigned diskette        : 1;
    unsigned unused          : 1;
    unsigned sysboard_ram    : 2;
    unsigned video           : 2;
    unsigned disks           : 2;
    unsigned dma_chip        : 1;
    unsigned rs232_ports     : 3;
    unsigned game_adapter    : 1;
    unsigned serial_printer  : 1;
    unsigned printers        : 2;
  } ax;
```

A structure that contains bit-fields may also include non-bit-field members. Thus, we may have

```
struct node
  {
    char *name;
    unsigned keyword_flag : 1;
    unsigned extern_flag  : 1;
    unsigned static_flag  : 1;
    char *value;
  } table[MAXSYMBOLS];
```

Non-bit-field members are aligned in an implementation-dependent manner appropriate to their types.

Individual bit-fields are accessed as any other structure members. Thus, we can write

```
if (ax.video == 03) printf("monochrome\n");
```

Bit-fields behave like small integers and may participate in arithmetic expressions just like other integers. We can, therefore, write:

```
devices = ax.disks + ax.rs232_ports + ax.printers;
```

or

```
for (i =0; i < MAXSYMBOLS; i++)
    table[i].keyword_flag =
        table[i].extern_flag =
            table[i].static_flag = 0;
```

A bit-field cannot be dimensioned; that is, an array of bit-fields, such as

```
flag : 1[5];
```

cannot be formed. Moreover, the address operator & cannot be applied to a bit-field object, and hence we cannot define a variable of type "pointer to bit-field."