C-9 Bitwise Operators

Bit Manipulation in C

The C language has Bitwise Operators.

They allow us to manipulate bits.

May only be applied to **integers**:

- char
- short
- int
- long
- unsigned

Bitwise Operators:

<u>Logical operators</u>

~	bitwise *compliment* (unary)
&	bitwise *and*
^	bitwise *exclusive or*
|	bitwise *inclusive or*

<u>Shift operators</u>

<<	left shift
>>	right shift

Bitwise Complement ~

It inverts the bit string representation;
the 0s become 1s, and the 1s become 0s.

int a = 70707;

binary representation of **a** is:

```
        00000000  00000001  00010100  00110011
```

the expression **~a** results in:

```
        11111111  11111110  11101011  11001100
```

so the **int** value of the expression **~a** is -70708

Truth Table for Logical Bit Operators:

Values of:

a	b	a & b	a ^ b	a | b
0	0	0	0	0
1	0	0	1	1
0	1	0	1	1
1	1	1	0	1

Bitwise And **&**

The AND will return a 1 if both bits have a value of 1,
else it returns a 0;

int a = 33333, b = -77777;

a → 00000000 00000000 10000010 00110101 → 33333

b → 11111111 11111110 11010000 00101111 → -77777

a & b → 00000000 00000000 10000000 00100101 → 32805

Masking, using the AND (1 of 2)

Suppose **a** is an unsigned integer variable whose value is 0x6db7.
Extract the **right**most 6 bits of the value using an AND (&).
Assign them to unsigned integer variable **b.**
Assign 0s to the 10 **left**most bits of **b**.

b = a & 0x3f;

```
                                          Base 10
   a = 0110  1101  1011  0111             28087
mask = 0000  0000  0011  1111                63
   b = 0000  0000  0011  0111                55
     = 0x37
```

Masking, using the AND (2 of 2)

Suppose **a** is an unsigned integer variable whose value is 0x6db7.
Extract the **left**most 6 bits of the value using an AND (&).
Assign them to unsigned integer variable **b.**
Assign 0s to the 10 **right**most bits of **b**.

b = a & 0xfc00;

```
                                            Base 10
   a = 0110  1101  1011  0111               28087
mask = 1111  1100  0000  0000                -1024
   b = 0110  1100  0000  0000               27648
     = 0x6c00
```

Bitwise *EOR* ^

The EOR will return a 1 if both bits have opposing values (1 and 0, or 0 and 1), else it returns a 0;

int a = 33333, b = -77777;

a → 00000000 00000000 10000010 00110101 → 33333

b → 11111111 11111110 11010000 00101111 → -77777

a ^ b → 11111111 11111110 01010010 00011010 → -110054

Masking, using the EOR (1 of 2)

Suppose **a** is an unsigned integer variable whose value is 0x6db7.
Let us reverse the **right**most 8 bits using an OR (^),
and preserve the **left**most 8 bits.
This new bit pattern will be assigned to the unsigned integer **b**.

b = a ^ 0xff;

		Base 10
a =	0110 1101 1011 0111	28087
mask =	0000 0000 1111 1111	255
b =	0110 1101 0100 1000	27976
=	0x6d48	

Masking, using the EOR (2 of 2)

Suppose **a** is an unsigned integer variable whose value is 0x6db7. The expression **a ^ 0x4** will invert the value of the bit number 2 (the third bit from the right) within **a**.

If this operation is carried out repeatedly, the value of **a** will alternate between 0x6db7 and 0x6db3

Thus, using this operation repeatedly will toggle the third bit from the right on and off.

```
                                        Base 10
0x6db7 = 0110  1101  1011  0111          28087
  mask = 0000  0000  0000  0100              4
0x6db3 = 0110  1101  1011  0011          28083
  mask = 0000  0000  0000  0100              4
0x6db7 = 0110  1101  1011  0111          28087
```

Bitwise *OR* |

The EOR will return a 0 if both bits have a value of 0 ,
else it returns a 1;

int a = 33333, b = -77777;

a → 00000000 00000000 10000010 00110101 → 33333

b → 11111111 11111110 11010000 00101111 → -77777

a | b → 11111111 11111110 11010010 00111111 → -77249

Masking, using the OR **|** (1 of 2)

Suppose **a** is an unsigned integer variable whose value is 0x6db7. Transform the corresponding bit pattern into another bit pattern in which:

the **right**most 8 bits are all 1s, and

the **left**most 8 bits retain their original value.

Assign this new bit pattern to the unsigned integer **b**.

```
b = a | 0xff;
```

```
                                        Base 10
     a = 0110  1101  1011  0111          28087
  mask = 0000  0000  1111  1111            255
     b = 0110  1101  1111  1111          28159
       = 0x6dff
```

Masking, using the OR **|** (2 of 2)

Suppose **a** is an unsigned integer variable whose value is 0x6db7. Transform the corresponding bit pattern into another bit pattern in which:

the **left**most 8 bits are all 1s, and

the **right**most 8 bits retain their original value.

Assign this new bit pattern to the unsigned integer **b**.

b = a | 0xff00; } Both accomplish the same thing.

b = a | ~0xff; } (2nd is independent of word size)

```
                                      Base 10
   a = 0110  1101  1011  0111         28087
mask = 1111  1111  0000  0000          -256
   b = 1111  1111  1011  0111           -73
     = 0xffb7
```

Two Examples using both *Complement* and *OR*

int a = 33333, b = -77777;

a → 00000000 00000000 10000010 00110101 → 33333

b → 11111111 11111110 11010000 00101111 → -77777

~(a | b) → 00000000 00000001 00100010 11010000 → 74448

(~a | ~b) → 11111111 11111111 11111111 11111010 → -6

Left Shift Operator:

Both operands must be integers of some sort.

expr1 << expr2

causes the bit representation of ***expr1***
to be shifted to the left
by the number of places specified by ***expr2****.*

Left Shift Operator Examples: (1 of 2)

char c = ‘Z’;

Expression	*Representation*	*Action*
c	00000000 00000000 00000000 01011010	un-shifted
c << 1	00000000 00000000 00000000 10110100	left-shifted 1
c << 4	00000000 00000000 00000101 10100000	left-shifted 4
c<< 31	00000000 00000000 00000000 00000000	left-shifted 31

Another Left Shift Operator Example: (1 of 2)

Suppose **a** is an unsigned integer variable whose value is 0x6db7.
The expression **b = a << 6;** will shift all bits
of variable **a** six places to the left and
assign the resulting bit pattern to the unsigned integer variable **b**.

lost bits

a = 0110 11$\underline{\text{01 1011 0111}}$

shift left

____

a << 6 = 0110 1101 11$\underline{\text{00 0000}}$ = 0x6dc0

filled with 0s

The leftmost 6 bits are lost.
The six rightmost bits are zero-filled.

<u>Right Shift Operator</u>:

Both operands must be integers of some sort.

expr1 >> expr2

Not symmetric to the left shift operator.

For *unsigned* integral expressions,
    0s are shifted at the high end.

For *signed* types, some machines shift in 0s,
    while others shift in sign bits.

The sign bit is the high-order bit;
It is 0 for nonnegative integers
It is 1 for negative integers

Right Shift Operator Examples: (1 of 2)

```
int      a = 1 >> 31;  /* shift 1 to the high bit */
unsigned b = 1 >> 31;
```

Expression	*Representation*	*Action*
a	00000000 00000000 00000000 01011010	unshifted
a >>3	00000000 00000000 00000000 00001011	right-shifted 3
b	00000000 00000000 00000101 10100000	unshifted
b >> 3	00000000 00000000 00000000 10110100	right-shifted 3

Another Right Shift Operator Example: (2 of 2)

Suppose **a** is an unsigned integer variable whose value is 0x6db7. The expression **b = a >> 6;** will shift all bits of **a** six places to the right and assign the resulting bit pattern to the unsigned integer variable **b**.

lost bits

a = 0110 1101 1011 0111

shift right

a << 6 = 0000 0001 1011 0110 = 0x1b6

filled with 0s

The rightmost 6 bits are lost.
The six leftmost bits are zero-filled.

```
#include <stdio.h>              // right_shift.c
#include <stdlib.h>
int main (void)
{
    unsigned a = 0xf05a;
    int b = a;

    printf("\nOriginal numbers: \n\n");
    printf("Unsigned %u. Integer %d.\n", a, b);
    printf("Both in Hex %x. %x.\n", a, b);

    printf("\nAfter the right shift: \n");
    printf("a in Hex %x.\n", a >> 6);
    printf("b in Hex %x.\n", b >> 6);
    return EXIT_SUCCESS;
}
        /* the output on next page */
```

/* rightshift.c output*/

```
[bielr@athena ClassExamples]> rightshift

Original numbers:
Unsigned 61530. Integer 61530.
Both in Hex f05a. f05a.

After the right shift:
a in Hex 3c1.
b in Hex 3c1.

[bielr@athena ClassExamples]>
```

Just like we can do a += 5; or a = a + 5;
the same works for the bit operators.

unsigned a = 0x6db7;

Expression	Equivalent Expression	Final Value
a &= 0x7f	a = a & 0x7f	0x37
a ~= 0x7f	a = a ~ 0x7f	0x6dc8
a |= 0x7f	a = a | 0x7f	0x6dff
a << 5	a = a << 5	0xb6e0
a >>= 5	a = a >> 5	0x36d

sizeof()

- The **sizeof** unary operator is used to obtain the **size of** a variable or datatype
- Used in Lab 9
- Reminder: there are 8 bits in a byte.

```
/*---------------------------------------------------(1 of 3)-----*/
/* Your Name */
/* Lab 8         */
#include <stdio.h>
#include <stdlib.h>

/* Function Prototypes */
void bitprint (unsigned num);
int circular_shift(unsigned num, int n);
/*-------------------------------------------------------------*/
int main (void)
{
     int left_count;
    unsigned num;              /* the starting number */
    unsigned shifted_num;
```

(2 of 3)

```
   do {
       /* read a unsigned integer */
       printf("\n\nEnter an unsigned integer value (0 to stop):  ");
       scanf("%d", &num);

       if (num != 0) {
           printf("\n\nEnter an unsigned integer value for the left shift:  ");
           scanf("%d", &left_count);
           printf("\n\nOriginal is %i \n\n", num);
           bitprint(num);
           shifted_num = circular_shift(num, left_count);
           bitprint(shifted_num);
           printf("Shifted it is %i \n", shifted_num);
       }                      //end of if
   } while (num != 0);        //end of do-while
   printf("\n\n");
   return EXIT_SUCCESS;
}
/*-------------------------------------------------------------*/
```

```
void bitprint (unsigned num)                                                    (3 of 3)
{
   unsigned mask;
   int bit, count, nbits;
   /* determine the word size in bits and set the initial mask */
   nbits = 8 * sizeof(int);                  /* finds number of bytes in an unsigned
                                                  and changes it to bits */
   mask = 0x1 << (nbits - 1);                /* place 1 in left most position
                                                  starting place for the mask */
   for(count = 1; count <= nbits; count++)
   {
            bit = (num & mask) ? 1: 0;    /* set display bit on or off */
            printf("%x", bit);            /* print display bit */
            if(count %4 == 0)
               printf(" ");          /* blank space after every 4th digit */
            mask >>= 1;              /* shift mask 1 position to the right */
   }
   printf("\n\n");
   return;
}
/*----------------------------------------------------------------*/
```

C-9 Bitwise Operators

Bit Manipulation in C

THE END