18.2.1: Bitwise Operators

[This section corresponds to K&R Sec. 2.9]

The bitwise operators operate on numbers (always integers) as if they were sequences of binary bits (which, of course, internally to the computer they are). These operators will make the most sense, therefore, if we consider integers as represented in binary, octal, or hexadecimal (bases 2, 8, or 16), not decimal (base 10). Remember, you can use octal constants in C by prefixing them with an extra `0` (zero), and you can use hexadecimal constants by prefixing them with `0x` (or `0X`).

The `&` operator performs a bitwise AND on two integers. Each bit in the result is 1 only if both corresponding bits in the two input operands are 1. For example, `0x56 & 0x32` is `0x12`, because (in binary):

```
  0 1 0 1 0 1 1 0
& 0 0 1 1 0 0 1 0
  ---------------
  0 0 0 1 0 0 1 0
```

The `|` (vertical bar) operator performs a bitwise OR on two integers. Each bit in the result is 1 if either of the corresponding bits in the two input operands is 1. For example, `0x56 | 0x32` is `0x76`, because:

```
  0 1 0 1 0 1 1 0
| 0 0 1 1 0 0 1 0
  ---------------
  0 1 1 1 0 1 1 0
```

The `^` (caret) operator performs a bitwise exclusive-OR on two integers. Each bit in the result is 1 if one, but not both, of the corresponding bits in the two input operands is 1. For example, `0x56 ^ 0x32` is `0x64`:

```
  0 1 0 1 0 1 1 0
^ 0 0 1 1 0 0 1 0
  ---------------
  0 1 1 0 0 1 0 0
```

The `~` (tilde) operator performs a bitwise complement on its single integer operand. (The `~` operator is therefore a unary operator, like `!` and the unary `-`, `&`, and `*` operators.) Complementing a number means to change all the 0 bits to 1 and all the 1s to 0s. For example, assuming 16-bit integers, `~0x56` is `0xffa9`:

```
~ 0 0 0 0 0 0 0 0 0 1 0 1 0 1 1 0
  --------------------------------
  1 1 1 1 1 1 1 1 1 0 1 0 1 0 0 1
```

The `<<` operator shifts its first operand left by a number of bits given by its second operand, filling in new 0 bits at the right. Similarly, the `>>` operator shifts its first operand right. If the first operand is `unsigned`, `>>` fills in 0 bits from the left, but if the first operand is signed, `>>` might fill in 1 bits if the high-order bit was already 1. (Uncertainty like this is one reason why it's usually a good idea to use all `unsigned` operands when working with the bitwise

operators.) For example, `0x56 << 2` is `0x158`:

```
   0 1 0 1 0 1 1 0 << 2
------------------
0 1 0 1 0 1 1 0 0 0
```

And `0x56 >> 1` is `0x2b`:

```
0 1 0 1 0 1 1 0 >> 1
---------------
  0 1 0 1 0 1 1
```

For both of the shift operators, bits that scroll ``off the end'' are discarded; they don't wrap around. (Therefore, `0x56 >> 3` is `0x0a`.)

The bitwise operators will make more sense if we take a look at some of the ways they're typically used. We can use `&` to test if a certain bit is 1 or not. For example, `0x56 & 0x40` is `0x40`, but `0x32 & 0x40` is `0x00`:

```
  0 1 0 1 0 1 1 0        0 0 1 1 0 0 1 0
& 0 1 0 0 0 0 0 0      & 0 1 0 0 0 0 0 0
  ---------------        ---------------
  0 1 0 0 0 0 0 0        0 0 0 0 0 0 0 0
```

Since any nonzero result is considered ``true'' in C, we can use an expression involving `&` directly to test some condition, for example:

```
if(x & 0x04)
        do something ;
```

(If we didn't like testing against the bitwise result, we could equivalently say `if((x & 0x04) != 0)` . The extra parentheses are important, as we'll explain below.)

Notice that the value `0x40` has exactly one 1 bit in its binary representation, which makes it useful for testing for the presence of a certain bit. Such a value is often called a *bit mask*. Often, we'll define a series of bit masks, all targeting different bits, and then treat a single integer value as a set of *flags*. A ``flag'' is an on-off, yes-no condition, so we only need one bit to record it, not the 16 or 32 bits (or more) of an `int`. Storing a set of flags in a single `int` does more than just save space, it also makes it convenient to assign a set of flags all at once from one flag variable to another, using the conventional assignment operator `=`. For example, if we made these definitions:

```
#define DIRTY   0x01
#define OPEN    0x02
#define VERBOSE 0x04
#define RED     0x08
#define SEASICK 0x10
```

we would have set up 5 different bits as keeping track of those 5 different conditions (``dirty,'' ``open,'' etc.). If we had a variable

```
unsigned int flags;
```

which contained a set of these flags, we could write tests like

```
if(flags & DIRTY)
        { /* code for dirty case */ }
```

```c
double uniform(void); // Returns a number in [0,1)

void OnePointCrossover(unsigned int p1, unsigned int p2,
                       unsigned int *f1, unsigned int *f2){
/* d \in [1, 8*sizeof(unsigned int)-1] */
     unsigned char d = uniform()*(8*sizeof(unsigned int)-1) + 1;
/* d 0's at the beginning and (8*sizeof(unsigned int) - d) 1's at the end */
     unsigned int mask = 0xFFFFFFFFU << d;

     *f1 = (p1 & mask) | (p2 & ~mask);
     *f2 = (p2 & mask) | (p1 & ~mask);
}

void OnePointCrossover(unsigned int p1, unsigned int p2,
                       unsigned int *f1, unsigned int *f2){
     unsigned char len = 8*sizeof(unsigned int);
     unsigned char d = uniform()*(len-1) + 1, di = len - d; /* d \in [1, len-1] */

     *f1 =((p1>>d)<<d) | ((p2<<di)>>di);
     *f2 =((p2>>d)<<d) | ((p1<<di)>>di);
}


void TwoPointCrossover(unsigned int p1, unsigned int p2,
                       unsigned int *f1, unsigned int *f2){
     unsigned char len = 8*sizeof(unsigned int);
     unsigned char p = uniform()*(len-2) + 1; /* p \in [1, len-2] */
/* We want a mask with p 0's at the beginning; len-q zeros at the end;
 *                 and 1's between the positions p and q inclusive
 * where q \in [p+1, len-1].
 * CONSISTENCY NOTE: p <= len-2 ==> p+1 <= len-1
 * THEN: mask = (0xFFFFFFFFU >> (len-q+p)) << p;
 * OBSERVE THAT: p+1 <= len-q+p <= len-1 is a random number
 * THUS: len-q+p = uniform()*(len-p-1) + p+1; */
    unsigned int mask = (0xFFFFFFFFU >> (((unsigned char) uniform()*(len-p-1)) + p+1)) << p;

     *f1 = (p1 & mask) | (p2 & ~mask);
     *f2 = (p2 & mask) | (p1 & ~mask);
}

void UniformCrossover(unsigned int p1, unsigned int p2,
                      unsigned int *f1, unsigned int *f2,
                      double prob){
     unsigned char len = 8*sizeof(*f);
     unsigned int mask = 0U;
     register unsigned char i;

     for(i=0; i < len; i++) if(uniform() < prob) mask = mask | (1U << i);

     *f1 = (p1 & mask) | (p2 & ~mask);
     *f2 = (p2 & mask) | (p1 & ~mask);
}


void mutation1(unsigned int *f, double prob){
  if(uniform() < prob) *f = (*f)^(1U << ((unsigned char) uniform()*8*sizeof(*f)));
}

void BitFlipMutation(unsigned int *f, double prob){
     unsigned char len = 8*sizeof(*f);
     unsigned int mask = 0U;
     register unsigned char i;

     for(i=0; i < len; i++) if(uniform() < prob) *f = (*f)^(1U << i);
}
```