15.3 Bit-operatorer

C konstruerades från början för att vara ett alternativ till assembler när man skulle skriva s.k. maskinnära program, program som nära samarbetar med datorns operativsystem och hårdvara. I sådana sammanhang har man ofta behov av att kunna hantera enskilda bitar i en minnescell. I C finns därför ett antal operatorer med vars hjälp man kan hantera heltalsvariabler som inte innehåller heltal utan bitmönster.

När man arbetar med bitmönster brukar man oftast använda *hexadecimala siffror*. Det som gör att det är lämpligt att använda sådana för att beskriva binära mönster är att man kan gruppera de binära siffrorna fyra och fyra. Varje sådan grupp kan beskrivas med en enda hexadecimal siffra. En hexadecimal siffra skrivs med något av tecknen `0`–`9` och `a`–`f` (eller `A`–`F`). Här visas bitmönstren för de 16 hexadecimala siffrorna. De hexadecimala siffrorna visas med röd färg.

NOTIS som inte är från boken:

Vill man skriva binära tal i C så kan man skriva det som följande:

int talBinary = 0b00000011 //motsvarar 3 decimalt

eller som hexadecimalt:

int talHex = 0x3 //motsvarar också 3 decimalt

```
0 0000   1 0001   2 0010   3 0011
4 0100   5 0101   6 0110   7 0111
8 1000   9 1001   a 1010   b 1011
c 1100   d 1101   e 1110   f 1111
```

Om vi t.ex. har en 16-bitars variabel som innehåller bitmönstret

```
0010 1111 1000 1100
```

så kan variabelns värde anges med de fyra hexadecimala siffrorna `2f8c`.

När man skriver hexadecimala konstanter i ett C-program skall de alltid inledas med tecknen `0x` eller `0X`. Därefter kommer en följ av hexadecimala siffror. Både stora och små bokstäver är tillåtna. Här kommer några exempel.

```
0x7  0xa  0xF  0X1a  0X2B 0x2f8c
```

Om man vill tolka en följd av hexadecimala siffror som ett numeriskt värde kan man säga att varje hexadecimal siffra anger ett värde i intervallet 0 till 15 (decimalt). Den hexadecimala siffran `a` betecknar t.ex. det decimala värdet 10 och siffran `f` det decimala värdet 15. Det hexadecimala talet `2a` motsvarar då det decimala värdet 42 ($2 \times 16^1 + 10 \times 16^0$) och det hexadecimala talet `1eb` motsvarar värdet 491 ($1 \times 16^2 + 14 \times 16^1 + 11 \times 16^0$).

Funktionerna `scanf` och `printf` kan läsa in och skriva ut heltal som hexadecimala tal. Man använder då omvandlingsspecifikationerna `%x`, `%hx` och `%lx` för **`unsigned int`**, **`short int`** resp. **`long int`**. Vid inläsning och utskrifter skrivs hexadecimala tal utan tecknen `0x` först.

Vi skall nu studera de operatorer som hanterar bitmöster. En sammanställning av dem finns i faktarutan.

Bit-operatorer

Är bara definierade för heltalstyper.
Utför alla operationerna bit för bit.

~	byter 0 <-> 1
&	och, bit för bit
<	eller, bit för bit
^	exclusive or
<<	vänsterskift
>>	högerskift

Observera att bit-operatorerna bara är definierade för heltalstyper. Hur de fungerar visas bäst med några exempel. I de följande satserna kan du anta att alla variabler har typen **unsigned short int** och att de är 16 bitar långa. I kommentarerana visas de bitar variablerna innehåller.

Operatorn ~ är enklast. Den har bara en operand. Som resultat ger den ett bitmönster där alla nollor har bytts mot ettor och tvärt om.

```
a = 0x7;              // a = 0000 0000 0000 0111
c = ~a;               // c = 1111 1111 1111 1000
```

Operatorerna &, |och ^ utför *and*, *or* och *exclusive or* bit för bit. Exclusive or innebär att en viss bit i resultatet blir en etta om exakt en av operanderna innehåller en etta i motsvarande position. Skulle ingen eller båda operanderna innehålla en etta blir resultatbiten lika med 0.

```
b = 0x14;             // b = 0000 0000 0001 0100
d = a & b;            // d = 0000 0000 0000 0100
e = a | b;            // e = 0000 0000 0001 0111
f = a ^ b;            // f = 0000 0000 0001 0011
```

Operatorerna << och >> utför s.k. *vänsterskift* respektive *högerskift*, dvs. de förskjuter bitarna åt höger eller vänster. Den vänstra operanden innehåller det bitmönster som skall förskjutas och den högra operanden anger hur många steg bitarna skall förskjutas. (Om den högra operanden är negativ eller större än antalet bitar i den vänstra operanden, är resultatet odefinierat.) När man utför vänsterskift flyttas alltid nollor in från höger. Vid högerskift på ett värde som är positivt eller är **unsigned** flyttas nollor in från vänster. (Om man utför högerskift på ett negativt värde som är **signed** är det odefinierat om nollor eller ettor flyttas in från vänster.) Ett par exempel:

```
g = b << 5;           // g = 0000 0010 1000 0000
h = c >> 2;           // h = 0011 1111 1111 1110
```

Man använder ofta operatorn & för att utföra s.k. *maskning*, dvs. välja ut vissa bitar i ett bitmönster. Följande sats maskar t.ex. bort de åtta bitarna i mitten i det bitmönster variabeln `h` innehåller:

```
i = h & 0xf00f;       // i = 0011 0000 0000 1110
```

Operatorn | kan användas för att lägga in ettor i vissa bitar. Följande sats ser t.ex. till att det finns ettor i de åtta bitarna längst till höger:

```
j = i | 0x00ff;       // i = 0011 0000 1111 1111
```

Lägg märke till att ingen av de operatorer du sett här påverkar sina operatorer. Vill man ha operatorer som gör detta kan man använda de motsvarande tilldelningsoperatorer som också finns. Man kan t.ex. skriva

```
a &= 0x000f;          // samma som a = a & 0x000f;
```