Lecture 10 - bit logical operators

Attention: It is not allowed to apply them to data of types *float* or *double*.

&	bit conjunction
|	bit alternative
^	bit symmetric difference
«	shifting to the left
»	shifting to the right
~	ones complement

Bit conjunction &

& is often used to "mask" some set of bits, e.g.

$$c = n\&0177;$$

converts to zero all except of seven lowest bits of variable n.

Differences between & and &&

```
E.g.  If          x=1,   y=2            then

    x & y         has value    0

    x && y        ----"----    1
```

Bit alternative operator |

Used to "set" bits. E.g.

$$x = x|MASK;$$

sets 1 on these bits in variable x which in $MASK$ are equal to 1.

Bit symmetric diffderence ^

Sets one at each bit position such that bits in both arguments are different and zero at these positions where they are the same. E.g.

$$x = 011\hat{}022;$$

gives as the result value 0.

Shift operators « and »

Used to shift argument argument staying on the left-hand side of operator. The number of shifting positions is determined by the argument staying on the right-hand side. E.g.

$$y = x << 2;$$

shifts x two positions to the left; bits which are free afterwards are filled with zeros (operation equivalent to the multiplication by 4.

$$y = x >> 2;$$

for a value of *unsigned* type, the released bits are always filled with zeros. For signed values however, on some machines these places shall be filled with the sign bit (arithmetic shift) and on the others with zeros (logical shift).

Complement to one ~

Reverses each bit 1 to 0 and vice-versa. Typical use:

$$y = x\&\tilde{}077;$$

last six bits of variable x are masked with zeros.

Attention: The above construction does not depend on the machine, while:

$$y = x\&0177700;$$

assumes 16-bit machine word and is therefore machine dependent.

Example:

```
/*  getbits: give n bits  x  from position  p
    Right end of  x  is the zero bit position;
    n, p  are reasonable integer quantities
*/
unsigned getbits(unsigned x, int p, int n)
{
  return (x >> (p+1-n)) & ~(~0 << n);
}
```

Bit operations on characters

ASCII codes								
Nr bitu	7 7	6 6	5 5	4 4	3 3	2 2	1 1	0 0
		0 - digits	0 - capital letters	0 - a-o				
		1 - letters	1 - digits/ small letters	digits/p-z				

```
char toupper(char c)
/*  Function converts small letters to capital  */
{
  char mask=223;   /*  223    -    11011111   */
  return c & mask;
}
```

```
char tolower(char c)
/*  Function converts capital letters to small  */
{
  return c | 32;    /*  32    -    00100000    */
}
```

```
char swapcase{char c)
/*  Conversion of capital letters
    to small and vice-versa  */
{
  return c ^ 32;
}
```

```
int multiply10(int n)
/*  Multiplication of an integer number by 10
    by means of the shift operators      */
{
  int m,p;
  m=n<<1;
  p=m<<2;
  return m+p;
}
```

Operators and assignement statements *op=*

i=i+3; is equivalent to i+=3;

```
op  jest jednym z operator\' ow:

  +  -  *  /  %  <<  >>  &  ^  |
```

Attention: If expr1 and expr2 are statements, then

```
expr1 op= expr2;           is equivalent to
expr1=(expr1) op (expr2);

Therefore assignement

x *= y+1;        corresponds to     x=x*(y+1);
```

Example:

```
/*  Count argument bits of value   1  */
int bitcount(unsigned x)
{
  int b;
  for (b=0; x!=0; x >> 1)
    if (x & 01)
      b++;
  return b;
}
```

Conditional statements

Statement realizing the function `z=max(a,b)`

```
if (a>b)

  z=a;

else

  z=b;

is equivalent to     z=(a>b) ? a:b;

or                   z= a>b ? a:b;
```

Priority of operator ?: is very low.

Conditional statement often positively influence program compactness. E.g.

```
1)
for (i=0; i<n; i++)
  printf("%6d%c", a[i],
          (i%10 == 9 || i== n-1) ? '\n':' ');
2)
printf("You've got  %d  part%s.\n",n, n==1 ? "":"s");
```

Bit fields

Definition of a set of masks

```
#define KEYWORD  01  /*  keyword           */
#define external 02  /*  external object   */
#define STATIC   04  /*  static object     */

or
enum { KEYWORD = 01, EXTERNAL = 02, STATIC = 04 };
```

Often appearing expressions:

```
flags |= EXTERNAL | STATIC;    /* sets bits    */
flags &= ~(EXTERNAL & STATIC); /* cancels bits */
if ((flags & (EXTERNAL | STATIC)) == 0) ...
/* condition true, when both bits are cancelled */
```

Bit fields – alternative mechanism in C language.

Bit field is the set of neighbour bits placed in the same memory unit called "word" (its size depends on implementation).

The above defined set of symbols may be replaced by the definition of three fields:

```
struct {
   unsigned int is_keyword : 1;  /*  1 - size of  */
   unsigned int is_extern  : 1;  /*      the field*/
   unsigned int is_static  : 1;  /*      in bits  */
} flags;
```

which defines variable *flags* with three one-bit fields. We recall them like ordinary variables. They behave like small integer variables and appear in arithmetic expressions equally with other integer objects.

Another form of the above formulated examples:

```
flags.is_extern = flags.is_static = 1;
flags.is_extern = flags.is_static = 0;
if (flags.is_extern == 0 && flags.is_static == 0) ...
```

Restrictions:

- on one machines fields are stored from the right to the left, on the other in the opposite way,
- fields are integer objects without sign,
- on many machines they can be placed exclusively inside the words,
- they are not arrays,
- they haven't got any address (it is not allowed to use upersand operator `&`).