

Упражнение 4

Всички данни в паметта на компютъра са представени чрез **битове**. Всяко число е представено в **двоична** бройна система. **Побитовите** операции работят върху тези битове.

Ще разглеждаме примери с **8-битово** число. Както знаем **най-старшият бит** се използва за определяне на знака на числото. Числото **5** в паметта на компютъра е представено като **00000101**. Добавяйки **1** към **5** ще получим **00000110** == **6**. Добавяйки **1** към **6** ще получим **00000111** == **7**. Добавяйки **1** към **7** ще получим **00001000** == **8**. **Най-голямото** 8-битово **знаково** число е **2^7-1** представено в компютъра като **01111111**. Добавяйки **1** към най-голямото число ще получим **overflow** на типа данни (като на километража на колата), ще превърти и ще стигнем до **най-малкото** число. И така числото **10000000** е най-малкото **8-битово знаково** число == **-2^8** . Обратно ако от най-малкото число извадим **1** ще получим най-голямото - тогава получаваме **underflow**.

Побитови операции (&, |, ^, ~).

Работят точно както и логическите операции, но върху битовете на данните. (**и**, **или**, **изключващо или [xor]**, **негация [отрицание]**)

a	b	a & b	a b	a ^ b	~a
1010	1110	1010	1110	0100	0101
1001	0101	0001	1101	1110	0110

Примери:

Код	В паметта
<code>unsigned int num1 = 5;</code>	000...000101 // 5
<code>unsigned int num2 = 6;</code>	000...000110 // 6
<code>cout << (num1 & num2);</code>	000...000100 // 4

// Извършваме операцията побитово "И" над числата 5 и 6 и
// извеждаме на конзолата резултата. Програмата ще изведе 4.
// Забележка: **unsigned int** най-често има размер **32 бита**. Но това не
// винаги е така. За да разберем колко голям е даден тип можем да
// използваме оператора **sizeof(<тип/променлива>)**, който ни връща
// размера на подадения тип в **байтове**. `sizeof(unsigned int)` ще ни
// върне размера на типа `unsigned int` в байтове, ако искаме да
// получим колко бита е можем просто да умножим по 8 (и това

зависи

// от архитектурата, но най-често са 8 бита).

// sizeof(unsigned int)*8 -> брой битове на типа.

Код	В паметта
<code>unsigned int num1 = 5;</code>	000...000101 // 5
<code>unsigned int num2 = 6;</code>	000...000110 // 6
<code>cout << (num1 num2);</code>	000...000111 // 7

Код	В паметта
<code>unsigned int num1 = 5;</code>	000...000101 // 5
<code>unsigned int num2 = 6;</code>	000...000110 // 6
<code>cout << (num1 ^ num2);</code>	000...000011 // 3

Побитово отместване (<<, >>).

Отмества битовете наляво или надясно.

Примери:

Код	В паметта
<code>unsigned int num = 5;</code>	000...000101 // 5
<code>cout << (num << 2);</code>	000...010100 // 20

Код	В паметта
<code>unsigned int num = 5;</code>	000...000101 // 5
<code>cout << (num >> 2);</code>	000...000001 // 1

// Забележка: При отместването на битове, паметта се допълва с **нули**.

// Единствения частен случай е когато числото, което отместваме, е

// **знаково** (например **int**) и е отрицателно, т.е. най-старшият му бит е

// 1, тогава отместването надясно допълва с **единици**, а отместването

// наляво запазва старшия бит като 1.

Знакови примери:

10000...0001010110	<< 2	->	10000...0101011000
10000...0001010110	>> 2	->	11100...0000010101
11000...0001010110	<< 3	->	10000...1010110000
00000...0001010110	<< 2	->	00000...0101011000
00000...0001010110	>> 2	->	00000...0000010101

// Забележка: При отместването, << отговаря на умножение

// по 2, а >> отговаря на деление на 2.

0001 == 1,

0001 << 1 -> 0010 == 2

0010 << 1 -> 0100 == 4

0001 << 3 -> 1000 == 2³

Побитови маски.

Можем да си ги представяме като прозорчето на календара. Там, където се намира прозорчето, тази дата разглеждаме. Така и побитовите маски - там, където се намира "прозорчето", този бит разглеждаме.

Примери:

- Четене на бит

```
unsigned int mask = 1;    // 000...0001
```

// Маска, с която ще можем да разглеждаме най-младшия бит.

Прилагането на тази маска върху число с операцията **побитово "и"** ще ни даде възможността да **видим** какъв е най-младшият бит на числото.

```
unsigned int num = 5;     // 000...0101
```

```
cout << (num & mask);    // 000...0001 == 1
```

// Програмата ще изведе най-младшия бит на числото **num**

Разглеждането на други битове става чрез отместване на маската.

```
mask = mask << 2;        // 000...0100
```

// Така маската е на 2-рия бит отдясно наляво (започвайки да броим // от нулевия)

```
if (num & mask == 0) {    // 000...0100
```

```
    cout << 0;
```

```
} else {
```

```
    cout << 1;
```

```
}
```

// Ако резултатът от прилагането на маската с побитово "и" е 0, то

// бита на мястото, означено с маската, е 0. Ако е резултатът е

// различен от 0, то бита е 1.

- Вдигане на бит

Аналогично на четенето на бит - взимаме маска и числото и прилагаме **побитово “или”**.

```
unsigned int mask = 1;    // 000...0001 == 1
unsigned int num = 6;     // 000...0110 == 6
num = (num | mask);       // 000...0111 == 7
// Така битът на най-младша позиция в числото num беше вдигнат
// (set-нат на 1)
```

- Сваляне на бит

Тук преди да приложим маската, трябва да я **обърнем** (да използваме побитово отрицание). След това прилагаме **побитово “и”**

```
unsigned int mask = 1;    // 000...0001 == 1
mask = ~mask;            // 111...1110
unsigned int num = 5;     // 000...0101 == 5
num = (num & mask);       // 000...0100 == 4
// Така битът на най-младша позиция в числото num беше свален
// (set-нат на 0)
```

- Обръщане на бит

Операцията, която използваме за обръщане на бит е **XOR** (побитово изключващо “или”).

```
unsigned int mask = 1;    // 000...0001 == 1
unsigned int num = 6;     // 000...0110 == 6
num = (num ^ mask);       // 000...0111 == 7

unsigned int mask = 1;    // 000...0001 == 1
unsigned int num = 5;     // 000...0101 == 5
num = (num ^ mask);       // 000...0100 == 4
// Така битът на най-младша позиция в числото num беше обърнат
// (flip-нат)
```