# Побитови операции

Побитовите операции са операции, които ни позвляват да променяме данни чрез директен достъп до паметта – буквално „местим единиците и нули“. Ще ги прилагаме върху числа. Повечето от тези операции действат по подобие на аритметичните – между две числа пишем знак и връщаме резултата от действието.

В паметта числата се представят със записа си в двоична бройна система. Например 21 = 10101 и се записва в паметта като 0000 0000 0000 0000 0000 0000 0001 0101. Многото нули отпред (не гарантирам, че са точният брой) показват, че числото е малко (но за записване на големи числа като 1 милион са необходими много цифри и тогава тези отпред ще се използват). В примерите ще разглеждаме само числа с по-малко от 8 съществени двоични цифри и за удобство ще пишем само тях. И така, ва паметта на компютъра представянето на 21 е

21 = 0001 0101

Всяка цифра в двоичното представяне се нарича бит. Удобно е да броим битовете отзад напред, започвайки от нула – така номерът на всеки бит съответства на степента на 2. В случая с $21 = 2^0 + 2^2 + 2^4$ имаме три бита с единици – битовете с номера 0, 2 и 4.

## Побитово изместване наляво <<

Пример:

```
int a = 60;          //  60 = 0011 1100
int c = 60 << 2;     // 240 = 1111 0000
```

Побитовото изместване наляво взема битовете на първото числото (започвайки от първата единица) и ги измества няляво с толкова позиции, колкото е второто. По този начин всъщност се увеличават степените на 2 в двоичното представяне на числото. Можем да мислим за тази операция като за умножение със степен на 2.

n << k е същото като n * K, където $K = 2^k$.

В случая 60 се умножава по $2^2$ , т.е. 4.
Ако k = 0, n не се променя.

## Побитово изместване надясно >>

То е подобно на изместването наляво, само че този път местим битовете в другота посока. Възможно е да изгубим някоя цифра 1 при местенето, например когато имаме 21 >> 1 ще

получим 0000 1010, което е равно на 10. Изместването надясно всъщност е целочислено деление със степен на 2.

n >> k е същото като n / K, където $K = 2^k$.

В случая 21 се дели целочислено на $2^1$ , т.е. 2.
Ако k = 0, n не се променя.

## Побитово И – &

Пример:

```
int a = 21;          // 21 = 0 0 0 1  0 1 0 1
int b = 25;          // 25 = 0 0 0 1  1 0 0 1
int c = a & b;       // 17 = 0 0 0 1  0 0 0 1
```

Ако бит номер X на a е 1 и бит номер X на b е 1, то бит номер X на a & b също ще е 1. Ако обаче съответният бит не е 1 и в двете числа, той ще е 0 в резултата.

## Побитово Или и Изключващо Или – | , ^

Пример:

```
int a = 21;          // 21 = 0 0 0 1  0 1 0 1
int b = 25;          // 25 = 0 0 0 1  1 0 0 1
int c = a | b;       // 29 = 0 0 0 1  1 1 0 1

int aa = 21;             // 21 = 0 0 0 1  0 1 0 1
int bb = 25;             // 25 = 0 0 0 1  1 0 0 1
int cc = aa ^ bb;        // 12 = 0 0 0 0  1 1 0 0
```

При побитовото Или ( | ), ако бит номер X е 1 в поне едно от числата a и b , то бит номер X на a | b ще е 1, иначе ще е 0.
При побитовото Изключващо Или ( ^ ), ако бит номер X е различен в двете числа, той ще е 1 в резултата, а ако е еднакъв, той ще е 0 в резултата.

## Как да разберем какъв е бит номер X

Достъчно е да приложим & върху въпросното число и $2^X$. Ако получим 0, значи е 0, а ако резултатът е различен от 0, значи е 1.

```
 0001 0101      // 21                0001 0101      // 21
&                                   &
 0000 0100      // 4 = 2^2           0000 1000      // 8 = 2^3
=                                   =
 0000 0100      // !=  0             0000 0000      // == 0
```

## Скоби и cout

Не е никак трудно да напишем програма с побитови операции, която работи, но дава грешен резултат. Символите, които използваме за тези операции, имат и други значения в C++ и това понякога обърква компилатора. Има и особени случаи, в които други оператори са с по-висок приоритет.

Най-сигурно е (поне в началото) да ограждаме побитовите операции в скоби. Така става ясно както какви точно действия се извършват, така и в какъв ред. А при употреба на cout няма как да по друг начин да уточним, че става дума за побитово изместване, а не за извеждане на две числа.

Примери:

```
cout  <<   6  <<  2;          // Отпечатва се: 62   (цифрите 6 и 2)
cout  <<  (6  <<  2);         // Отпечатва се: 24   (което е 6 * 4)

if ( 21 & 8 == 0)
        cout << “true”;
else
        cout << “false”;

   //  По-рано видяхме, че  21 & 8  е 0.
   //  Ще се отпечата  false  обаче, защото компилаторът тук се обърква и
   //  първо проверява дали 8 е равно на 0.

if ( (21 & 8) == 0)
        cout << “true”;
else
        cout << “false”;

   // Този път ще се отпечата  true.
```