**Как да принтираме едно число като двоично?**

int number = 77812313;

Console.WriteLine("number as Binary: {0}", Convert.ToString(number,2));

Което ще принтира:

number as Binary: 100101000110101001001011001

Ако искаме всичките 32 бита да ни бъдат принтирани ще стане с добавка .PadLeft(32, '0'), само ако числото е положително.

Console.WriteLine("number as Binary: {0}", Convert.ToString(number,2).PadLeft(32, '0'));

Което ще принтира:

number as Binary: 00000100101000110101001001011001

Ако искаме **отрицателно** .PadLeft(32, '1')

Имайте предвид, че .PadLeft() - добавя до 32рия знак от стринга, с char, който сме му подали. Т.е. от нас зависи дали ще допише 0 или 1, и не ви проверява дали числото е отрицателно или положително.

**Оператори << и >>?**

При shift-ване(<< и >>) на **положителни** числа съответно се допълва с **нули**/0/

Например

1 //00001

1<< 4 //10000

1>>4 //00001

Имайте предвид че докато в единия край добавяте нули в другия край се ТРИЯТ битовете! Например имаме следното десетично число:

13 //1101

13 >> 2 //0011

итриха се последните два бита (01)

При shift-ване(<< и >>) на **отрицателни числа** съответно се допълва с **единици** /1/. По същия начин се добавят и се трият излишните битове само, че се добавят 1ци! Прочетете 32рия бит какво значение има за двоичното представяне :)

**Какво е маска и как се създава?**

Маска е число което ползваме за някакви операци върху битовете на друго число. Например ако искаме да влияем по някакъв начин на бита на 5та позиция. Можем да си направим маска с числото 1 и да я изместим наляво до позиция 5/припомням битовете се броят от 0/

int mask = 1; //...001

mask = mask << 5; //100000

По-надолу ще обясняваме какво можем да правим с тази маска.

**Как се създава маска, ако искаме да влияем на последователни битове?**  
например на битовете на позиция 5,4,3. Т.е. ни трябва маска, която да е равна на 111 в двоично представяне. T.e. ще влияем върху три бита

Например:

Правим си маска, която я изместваме на ляво (броя битове върху които влияем), след което вадим десетичното число 1

int number = 1; //0001

number = number << 3; //..000 1000

number = number - 1; //...000 0111

Припомням, че ако shift-ването се осъществява върху отрицателно число то ще се допълва с единици. Когато се invert-не дадено положително число то става отрицателно.

Това се получава, защото разликата между десетичното число съответстващо на двоично 1000 и десетичното число съответстващо на двоично 111 е равна на десетично 1;

По същия начин

(двоично 100) - 1 = (двоично11);

**Как се взима бит на позиция p?**

int position = 4;

int number = 35;                 // 00000000 00100011

int mask = 1 << position;          // 00000000 00010000

int result = number & mask;      // 00000000 00010000

Ако result == 0 означава, че бита на позиция 4 е равен на 0!

Ако result != 0 означава, че бита на позиция 4 е равен на 1! Тук важно уточнение е, че проверката не трябва да е result == 1, защото според позицията на бита, който проверяваме result ще бъде равно на (2 на степен (позиция)). Ако това не ви е ясно прочетете как се превръща число от двоична в десетична бройна система.

**Как да запазите стойността на бита на позиция p?**

int position = 4;

int number = 35;                 // 00000000 00100011

int mask = 1 << position;          // 00000000 00010000

int bitAtPosition= number & mask;      // 00000000 00010000

След като запаметим бита на съответната позиция можем да го върнем на нулева позиция, за да го имаме готов за евентуална по-късна обработка.

bitAtPosition = bitAtPosition >> position // 00000000 00000001

В случая ще имаме десетично число 1, ако беше нула, щяхме да имаме десетично число 0.

**Как да занулите бита на дадена позиция p?**

int p = 5;

int n = 35;                 // 00000000 00100011

int mask = ~(1 << p);       // 11111111 11011111

int result = n & mask;      // 00000000 00000011

Когато & (and-ваме) едно число със друго, което е само единици (1111 1111 1111) ние не влияем по никакъв начин на първото! Например:

0000 1000 1101

&

0000 1111 1111

--------------------

0000 1000 1101

Както се вижда от примера по-горе всичко остава непроменено!

Ако искаме да занулим някой от битовете и да го сложим нула, трябва да вмъкнем 0. Например:

0000 1000 1101

&

0000 1111 0111

--------------------

0000 1000 0101 тук вече имаме промяна на третия бит/броим ги от нула, от дясно наляво/

Може би се питате какво ако беше вече там нула?! Няма значение! Защо?!

0000 1000 0101

&

0000 1111 0111

--------------------

0000 1000 0101

Бита, който ни трябва на тази позиция вече е нула и си остава нула!

**Въпроса е как да сложим тази нула там?**

Правим си маска, като позицията на която искаме нула трябва да има единица а на всички други позиции трябва да има нула! На този етап  е така. Например

Ако искаме нула на 3та позиция си правим следната маска

int mask = 1<<3 //0000 1000

Съответно нашата цел е от 0000 1000 да го превърнем в 1111 0111 Това можем да го постигнем чрез invert или ~

~(0000 1000) -> 1111 0111

т.е. можем да направим маската по следните няколко начина, който Ви е по-лесен за четене:

int mask = ~(1<<3); //  1111 0111

или

int mask = 1; //0000 0001

mask = mask << 3; //0000 1000

mask = ~mask; // 1111 0111

двете неща са едно и също :)

**Как да сетнем даден бит на определена позиция на определена стойност/1 или 0/ или включване и изключване на битове?**

Най-лесния начин е да занулим съответния бит!

например имаме

int number = 11 //1011; и искаме да 3тия бит да го направим на НУЛА ИЛИ ЕДНО, който в момента е 1/броим от позиция нула/.

Вариант, който винаги ще работи в нашия случай е следния:

- правим си маска с единица за позиция 3

- invert-ваме маската, така че тя да ни стане 0 и всичко друго 1ци, за да не влияем на нито един бит ОСВЕН третия.

- &(and-ваме) маската и нашето число чрез което зануляваме бита на позицията 3 и НЕ влияем по никакъв начин на другите

- създаваме си маска със съответната стойност на бита, която искаме да придадем на нашето число за същата позиция ТРИ

- | (or-ваме) нашето число с занулен трети бит и маската която съдържа желаната от нас стойност. Използва се |, защото само по този начин може да сложим нашата стойност. Никой друг от другите оператори няма да работи в двата случая, когато искаме да сложим 1 или 0;

Сега малко код:

int number = 11 //1011;

int mask = 1; // 0001

mask = mask << 3; // 1000

mask = ~(mask) // ...11111 01111;

number = number & mask; //..000 0011

int bitValueOne= 1; // 0001

int bitValueZero = 0; // 0000

//Нарочно давам и пример с нула, защото искам да покажа, че това ще работи и при двата случая и съответно после може да го ползвате при смяна на местата на битовете.

bitValueOne = bitValueOne << 3; //1000

//number  в момента ни е // 0011 и ние искаме да му дадем на трета позиция 1ца

number = number | bitValueOne; //1011 по този начин сет-ваме на единица

Пример с bitValueZero, с зануления number //0011:

bitValueZero = bitValueZero << 3; //00000

//number = 0011

number = number & bitValueZero; //0011

**Как да сменим местата на битове на дадени позици P и N?**

Нека помислим малко преди да започнем задачата как бихме постъпили в реалния живот?

- ще си запазим бита на позиция P

- ще си запазим бита на позиция N

- ще занулим бита на позиция P

- ще занулим бита на позиция N

- ще shift-нем бита от позиция P на позиция N и ще го OR-нем със зануленото число

- ще shift-нем бита от позиция N на позиция P и ще го OR-нем със зануленото число

Сега малко в действие :)

Например искаме да сменим местата на битовете на позиция closePosition = 3 и farPosition = 5; и Имаме следното число

int number = 39; //100111 желания резултат като сменим местата е 001111;

//Създаваме маска, за да вземем стойността на бита на позиция closePosition;

int maskClose = 1;

//преместваме я на съответната позиция

maskClose = 1<<closePosition; // 1000

//взимаме бита

int closeBitValue = number & maskClose; //000000

// връщаме бита на първа позиция, защото в случая е нула, но ако беше единица???

// по този начин гарантираме, че на нулева позиция в closeBitValue ни е стойността на бита от 3та позиция ;)

closeBitValue = closeBitValue >> closePosition;

//По същия начин правим за далечния бит.

int maskFar = 1;

maskFar = 1<<farPosition; //100000;

int farBitValue = number & maskFar; //000 100000;

farBitValue = farBitValue >> farPosition: //000001;

farBitValue

closeBitValue = closeBitValue >> closePosition;

Така вече имаме запаметените съотвените битове, сега нека занулим 3тия и 5тия бит! За целта ще invert-нем вече ползваните маски, за да спестим малко код. Ако ви е по-удобно си направете нови маски

int invertedMaskClose = ~maskClose; //11110111

number  в момента ни е равен на /100111

Зануляваме близкия бит

number = number & maskClose; //100111  в случая той си е нула, но ако беше единица трябваше да го занулим???

int invertedMaskFar = ~maskFar; //111011111

//number ни е вече 100111

number = number & invertedMaskFar; //000111

Сега вече сме занулили тези битове на тези позиции 3 и 5 и можем да ги Or-нем със съответните стойности на битовете.

Местим стойността на БЛИЗКИЯ бит на ПОЗИЦИЯТА на ДАЛЕЧНИЯ бит

closeBitValue = closeBitValue << farPosition; //000000

тук с нулата може да не ви стане много ясно!

Сега | OR-ваме зануленото число с стойнсотта, на близкия бит, който като маска е преместен на далечната позиция

number = number | closeBitValue // тук влияем на 5тия бит вече !!!! //000111

След което правим същото за далечния. Местим стойността на ДАЛЕЧНИЯ на позицията на БЛИЗКИЯ

farBitValue = farBitValue << closePosition; // 1000

OR-ваме със зануленото число

number = number | farBitValue; //00010111

**Как се сменят позицията на поредица от битове, напрмер 3,4,5 и 24,25,26?**

Това го оставям на вас с малки подсказки. Стъпките са същите както за смяната на единични битове, само че маските които създавате са за броя на поредицата от битове. В случая например 3. По-горе съм обяснил как се създават такива маски!

**Съжалявам за правописните грешки, по-късно и ще се опитам да го оцветя малко, за да добиете представа! Някой, ако открие грешки да пише, защото съм го писал на един дъх без VS и може да съм изпуснал/сбъркал нещо.**