Операції з бітами

Операції з бітами

Усі дані представлені в комп'ютері як послідовність бітів.

Кожний біт може приймати значення 0 або 1.

У більшості комп'ютерних систем 8 бітів складається в один байт – стандартна одиниця пам'яті для змінної **char.**

У мові C/C++ операції над бітами використовуються для цілочисленних операндів (**char, short, int, long**).

Операції з бітами

Побітові операції – це операції, які передбачають прямі дії з бітами змінних, або визначеними бітами комірок пам’яті.

Порозрядні операції застосовуються тільки до цілочисельних операндів і "працюють" з їх двійковими представленнями.

Ці операції неможливо використовувати із змінними типу double, float, long double.

Операції з бітами

Дія в мові С	Опе-рація	Пояснення
Порозрядне "І" (Bitwise AND)	&	Біт результату дорівнює 1, якщо відповідні біти обох операндів дорівнюють 1
Порозрядне "АБО" (Bitwise OR)	|	Біт результату дорівнює 1, якщо хоча б один відповідний біт дорівнює 1
Порозрядне "виключне АБО" (Bitwise XOR)	^	Біт результату дорівнює 1, якщо біти операндів різні
Доповнення (Bitwise NOT)	~	Усі біти інвертуються: 0 → 1, 1 → 0
Зсув вліво	<<	Біти зсуваються вліво; праві біти заповнюються нулями
Зсув вправо	>>	Біти зсуваються вправо; ліві біти заповнюються залежно від типу даних

Операції з бітами

Порозрядне **І** (Bitwise **AND**) - кон'юнкція (**логічне множення**)

X	Y	X AND Y
0	0	0
0	1	0
1	0	0
1	1	1

Порозрядне **АБО** (Bitwise **OR**) - диз'юнкція (**логічне додавання**)

X	Y	X OR Y
0	0	0
0	1	1
1	0	1
1	1	1

Виключне **АБО** (альтернативна диз'юнкція)

X	Y	X XOR Y
0	0	0
0	1	1
1	0	1
1	1	0

Порозрядне **НЕ** (Bitwise **NOT**) - **логічне заперечення**

X	NOT X
0	1
1	0

Заперечення — це операція, яка **інвертує значення біта**

Особливість переведення чисел до двійкової системи

Продовжіть ряд:

1	2^0
2	2^1
4	2^2
8	2^3
16	2^4
32	2^5
…	…
1024	2^{10}

Особливість переведення чисел до та з двійкової системи

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
128	**64**	**32**	**16**	8	4	2	1
0	0	0	0	0	1	0	1

$$1*2^0+0*2^1+1*2^2+0*2^3+\ldots=5$$

0	1	0	1	1	1	1	1

$$1*2^0+1*2^1+1*2^2+1*2^3+1*2^4+0*2^5+1*2^6+0*2^7+\ldots=95$$

Обчислення побітових операцій

Побітове І:

5&4=4

7&3=3

5&2=???

&	0	1	0	1
0	0	0	1	0
0	0	0	0	0

5&2=0

Обчислення побітових операцій

Побітове І:

5&4=4

```
&   0 1 0 1
    0 1 0 0
    -------
    0 1 0 0
```

7&3=3

```
&   0 1 1 1
    0 0 1 1
    -------
    0 0 1 1
```

13&14=12

```
&   1 1 0 1
    1 1 1 0
    -------
    1 1 0 0
```

Особливість переведення чисел до двійкової системи

Продовжіть ряд:

1	2^1-1
3	2^2-1
7	2^3-1
15	2^4-1
31	2^5-1
63	2^6-1
…	…
1023	$2^{10}-1$

Особливість переведення чисел до двійкової системи

Трикутник з одиниць:

1
3
7
15
31
63

- …
- 1023

- 1
- 11
- 111
- 1111
- 11111
- 111111
- …
- 1111111111

Особливість переведення чисел до двійкової системи

Трикутник з одиниць:

0000000001 = 1

0000000011 = 3

0000000111 = 7

0000001111 = 15

0000011111 = 31

0000111111 = 63

- …
- **1111111111 = 1023**

Обчислення побітових операцій

Побітове І – виділення молодших розрядів:

&	x	x	x	x	x	x	x
	0	0	0	1	1	1	1
	0	0	0	x	x	x	x

5&7=5

&	0	1	0	1
	0	1	1	1
	0	1	0	1

5&3=1

&	0	1	0	1
	0	0	1	1
	0	0	0	1

5&1=1

&	0	1	0	1
	0	0	0	1
	0	0	0	1

Обчислення побітових операцій

Побітове АБО:

5|4=5

0	1	0	1
0	1	0	0
0	1	0	1

7|3=7

0	1	1	1
0	0	1	1
0	1	1	1

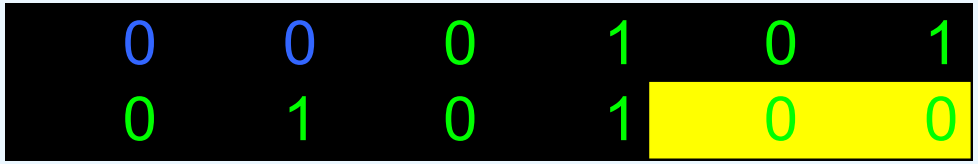
13|14=15

1	1	0	1
1	1	1	0
1	1	1	1

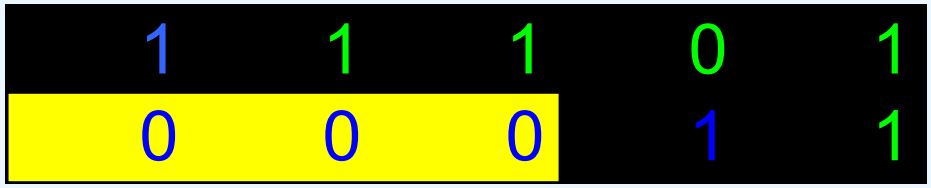
Обчислення побітових операцій

Побітові зсуви (<< та >>):

5<<2=101 << 2 = 10100 = 20



29>>3=11101>>3=00011 = 3



Операції з бітами

У побітових (bit-wise) операціях значення біта, рівне 1, розглядається як логічна істина, а 0 як хиба.

Побітове І (оператор **&**) бере два числа і логічно перемножує відповідні біти.

Наприклад, якщо логічно помножити 3 на 8, то отримаємо 0.

```
char a = 3;
char b = 8;
char c = a & b;
printf("%d", c);
```

Так як у двійковому вигляді **3** у вигляді однобайтного цілого представляє собою 00000011.

Операції порозрядного зсуву

Операції порозрядного зсуву можна використовувати замість операцій множення і ділення.

Наприклад, розглянемо такі оператори:

```
1  #include <stdio.h>
2
3  int main() {
4      int a=13, shift=3, res;
5      res=a<<shift;
6      printf ("%d", res);
7      return 0;
8  }
```

```
104

...Program finished with exit code 0
Press ENTER to exit console.
```

Операції порозрядного зсуву

число 13 (значення a) буде представлено таким чином:

00000000 00001101

Після зсуву вліво на 3 біти (a<<shift) ми отримаємо:

00000000 01101000

Останнє значення – це число 104 у двійковій системі.

Тобто, **зсув вліво числа 13 на 3 біти** –
це добуток 13 і 8 ($2*2*2=2^n$)

Узагальнюючи вищезазначене, маємо, що порозрядний зсув вліво будь-якого цілого числа на n бітів призведе до **збільшення цього числа у 2^n разів**.

Операції порозрядного зсуву

Аналогічним чином можемо впевнитися, що порозрядний зсув цілого числа вправо на один біт призведе до отримання нового значення, яке буде вдвічі менше за початкове.

І в загальному випадку порозрядний зсув вправо будь-якого цілого числа на n бітів призведе до зменшення цього числа у 2^n разів.

Решту операцій з бітами (&, |, ~, ^) використовують для отримання доступу до конкретного біта

Операції порозрядного зсуву


```
int a =0, bit=3;
a = a | bit;
printf (“%d”, a);
```

призведуть до виведення на екран числа '3', тобто присвоєнню '***a=3***‘.

Тобто

```
  0000000000000000
| 0000000000000011
_____________________
  0000000000000011
```

Хоча, насправді, ми практично встановили два останніх біта змінної ***a*** в 1.

Операції порозрядного зсуву

Для представлення від'ємних цілих чисел у пам'яті комп'ютера використовується так звана **нотація доповнення до двох**:

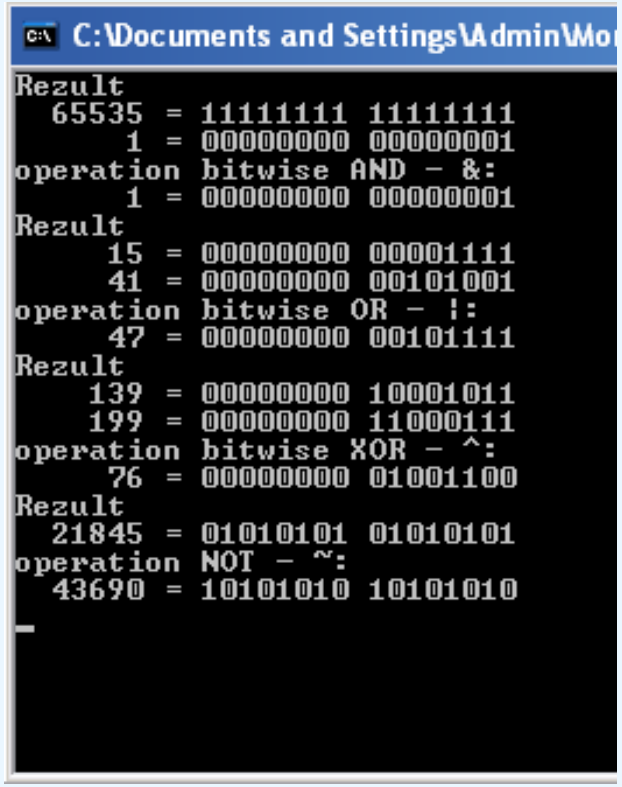
- ✓ обчислюється доповнення цього числа до 1 шляхом використання операції '~';
- ✓ значення, яке отримане, доповнюється до 2 шляхом простого додавання одиниці.

Операції порозрядного зсуву

```c
#include <stdio.h>
#include <conio.h>

void displayBits(unsigned);

void main(void) {

    unsigned n1, n2, m, setBits;
    n1=65535;     /*1111111111111111*/
    m=1;        /*0000000000000001*/
    printf("Rezult\n");
    displayBits(n1);
    displayBits(m);
    printf("operation bitwise AND - &:\n");
    displayBits(n1 & m);

    n1=15;
    setBits=241;
    printf("Rezult\n");
    displayBits(n1);
    displayBits(setBits);
    printf("operation bitwise OR - |:\n");
    displayBits(n1 | setBits);

    n1=139;
    n2=199;
    printf("Rezult\n");
    displayBits(n1);
    displayBits(n2);
    printf("operation bitwise XOR - ^:\n");

    displayBits(n1 ^ n2);
    n1=245;
    printf("Rezult\n");
    displayBits(n1);
    printf("operation NOT - ~:\n");
    displayBits(~n1);
    getch();
}

void displayBits(unsigned value) {
    unsigned c, displaym = 1 << 15;
    printf("%7u = ",value);
    for (c = 1; c<=16; c++) {
        putchar(value & displaym ? '1' : '0');
        value<<=1;
        if (c%8==0) putchar(' ');
   }
    putchar('\n');
}
```

Операції порозрядного зсуву

```
Rezult
  65535 = 11111111 11111111
      1 = 00000000 00000001
operation bitwise AND - &:
      1 = 00000000 00000001
Rezult
     15 = 00000000 00001111
    241 = 00000000 11110001
operation bitwise OR - |:
    255 = 00000000 11111111
Rezult
    139 = 00000000 10001011
    199 = 00000000 11000111
operation bitwise XOR - ^:
     76 = 00000000 01001100
Rezult
    245 = 00000000 11110101
operation NOT - ~:
4294967050 = 11111111 00001010


...Program finished with exit co
```



Приклад 1. Написати функцію, яка перевіряє, чи встановлений певний біт у числі


```c
#include <stdio.h>

int bit(int a, int pos) {
    return (a & (1 << pos)) != 0;
    // Зсув бітів вправо на вказану позицію та перевірка останнього біта
}

int main() {
    int a, pos;

    printf("Введіть число: ");
    scanf("%d", &a);
    printf("Введіть позицію біта (починаючи з 0): ");
    scanf("%d", &pos);

    if (bit(a, pos)) {
        printf("Біт на позиції %d встановлений.\n", pos);
    } else {
        printf("Біт на позиції %d не встановлений.\n", pos);
    }

    return 0;
}
```

```
Введіть число: 12
Введіть позицію біта (починаючи з 0): 2
Біт на позиції 2 в встановлений.


...Program finished with exit code 0
Press ENTER to exit console.
```

```
Введіть число: 25
Введіть позицію біта (починаючи з 0): 1
Біт на позиції 1 н встановлений.


...Program finished with exit code 0
Press ENTER to exit console.
```

```c
#include <stdio.h>

int bit(int a, int pos) {
    return (a & (1 << pos)) != 0;
    // Зсув бітів вправо на вказану позицію та перевірка останнього біта
}

void bin(int a) {
    for (int i = sizeof(a) * 8 - 1; i >= 0; i--) {
        printf("%d", (a >> i) & 1);
    }
    printf("\n");
}

int main() {
    int a, pos;

    printf("Введіть число: ");
    scanf("%d", &a);
    printf("Число в двійковому форматі: ");
    bin(a);
    printf("Введіть позицію біта (починаючи з 0): ");
    scanf("%d", &pos);


    if (bit(a, pos)) {
        printf("Біт на позиції %d встановлений.\n", pos);
    } else {
        printf("Біт на позиції %d не встановлений.\n", pos);
    }

    return 0;
}
```

```
Введіть число: 25
Число в двійковому форматі: 00000000000000000000000000011001
Введіть позицію біта (п?чинаючи з 0): 2
Біт на позиції 2 н? встановлений.


...Program finished with exit code 0
Press ENTER to exit console.
```

```
Введіть число: 25
Число в двійковому форматі: 00000000000000000000000000011001
Введіть позицію біта (п?чинаючи з 0): 3
Біт на позиції 3 в?тановлений.


...Program finished with exit code 0
Press ENTER to exit console.
```

Пояснення

Операція (**a & (1 << pos)**) використовується для перевірки конкретного біта в числі a на позиції **pos**.
Давайте розберемо її детальніше:

1. 1 << pos
   Це операція зсуву вліво.
   Операція << зсуває біти числа вліво на задану кількість позицій.
   У випадку 1 << pos, одиниця (1) зсувається на pos позицій вліво.
   Наприклад:
   - ✓якщо pos = 0, то 1 << 0 дає 0001 (у двійковій системі числення).
   - ✓якщо pos = 3, то 1 << 3 дає 1000 (в двійковій системі).

   Таким чином, ми отримуємо маску, в якій є одиниця лише на позиції pos, а всі інші біти — нулі.

Пояснення

2. **a & (1 << pos):**

   Тут відбувається побітова операція AND між числом a та маскою, яка створена за допомогою (1 << pos).

   Операція AND порівнює відповідні біти обох чисел:

   - якщо обидва біти дорівнюють 1, результат буде 1.
   - в іншому випадку результат буде 0.

   Тобто, ми перевіряємо тільки один конкретний біт числа a на позиції pos.

   Якщо на цій позиції в числі a стоїть одиниця, результат операції буде ненульовим (якщо a має одиницю на позиції pos).

   Якщо на цій позиції стоїть нуль, результат буде 0.

Пояснення на прикладі

Припустимо, що ми маємо число a = 12 (в двійковій формі це 1100), і хочемо перевірити, чи встановлений біт на позиції pos = 2:

1 << pos:

Для pos = 2, ми маємо 1 << 2, що дає маску 0100 (в двійковій системі).

a & (1 << pos):

Тепер робимо операцію a & (1 << pos), тобто 1100 & 0100:

```
1100
& 0100
------
0100
```

Результат — це 0100, що не є нулем, тобто біт на позиції 2 встановлений.

Пояснення на прикладі

Якщо ми спробуємо перевірити біт на позиції, де є нуль, наприклад, pos = 1:

1 << pos	Для pos = 1, 1 << 1 дає маску 0010
a & (1 << pos)	Тепер робимо операцію a & (1 << pos), тобто 1100 & 0010 1100 & 0010 ------ 0000

Результат — це 0000, що означає, що біт на позиції 1 не встановлений.

Висновок

- (a & (1 << pos)) дозволяє перевірити, чи встановлений біт на певній позиції в числі.
- Якщо результат операції не дорівнює нулю, то біт на цій позиції встановлений (тобто його значення — 1).
- Якщо результат операції дорівнює нулю, то біт на цій позиції не встановлений (тобто його значення — 0).

Приклад 2. Написати функцію, яка знаходить максимальну ступінь двійки, що ділить число.


```c
#include <stdio.h>

int max2pow(int a) {
    if (a == 0) {   // Перевірка, чи число не є нулем
        return 0;
    }

    int p = 0;
    while ((a & 1) == 0) { // Поки число парне, зсуваємо його вправо
        a >>= 1;
        p++;
    }
    return p;
}
void bin(int a) {
    for (int i = sizeof(a) * 8 - 1; i >= 0; i--) {
        printf("%d", (a >> i) & 1);
    }
    printf("\n");
}

int main() {
    int a;
    printf("Введіть число: ");
    scanf("%d", &a);
    printf("Число в двійковому форматі: ");
    bin(a);
    int p = max2pow(a);

    if (p > 0) {
        printf("Максимальна ступінь двійки, що ділить число %d: 2^%d\n", a, p);
    } else {
        printf("Число %d не ділиться на жодну ступінь двійки (воно непарне).\n", a);
    }

    return 0;
}
```

```
Введіть число: 128
Число в двійковому форматі: 00000000000000000000000010000000
Максимальна ступінь дв♦йки, що ділить число 128: 2^7


...Program finished with exit code 0
Press ENTER to exit console.
```

```
Введіть число: 25
Число в двійковому форматі: 00000000000000000000000000011001
Число 25 не ділиться на жодну ступінь двійки (воно непарне).


...Program finished with exit code 0
Press ENTER to exit console.
```

Дякую за увагу!