

# Маніпулювання бітами



*Лекція №4*

*Дисципліна «Програмування»*

*2-й семестр*

# Побітові логічні операції

## Доповнення до одиниці або побітове заперечення (~)

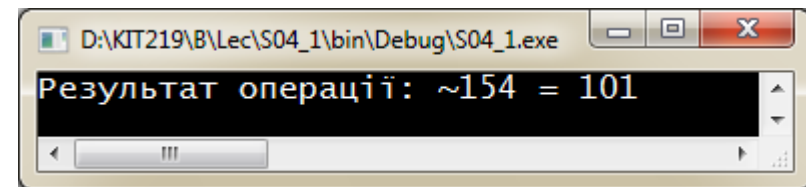
Унарна операція (~) перетворює кожну одиницю на нуль, а кожен нуль на одиницю:

$\sim 10011010_2$  // вираз  
 $01100101_2$  // результат  
 $\sim 154_{10} = 101_{10}$

Таблиця істинності

x	y
0	1
1	0

```
#include <stdio.h>
#include <windows.h>
int main(void) {
    unsigned char ch;
    SetConsoleOutputCP(1251);
    ch = ~154;
    printf("Результат операції: ~154 = %d\n", ch);
    return 0;
}
```





# Побітові логічні операції

## Побітова операція "І" (&)

Двійкова операція (&) створює нове значення за рахунок виконання побітового порівняння двох операндів. Для кожної позиції підсумковий біт буде дорівнювати **1**, тільки якщо обидва відповідні біти в операндах дорівнюють **1**.

$$\begin{array}{r} 10010011_2 \\ \& \\ 00111101_2 \\ \hline 00010001_2 \end{array} \quad 147_{10} \& 61_{10} = 17_{10}$$

Операція "**І**" може бути об'єднана з присвоюванням (**&=**).

```
val &= 0377;
```

дає такий самий результат, як і оператор

```
val = val & 0377;
```

Таблиця істинності

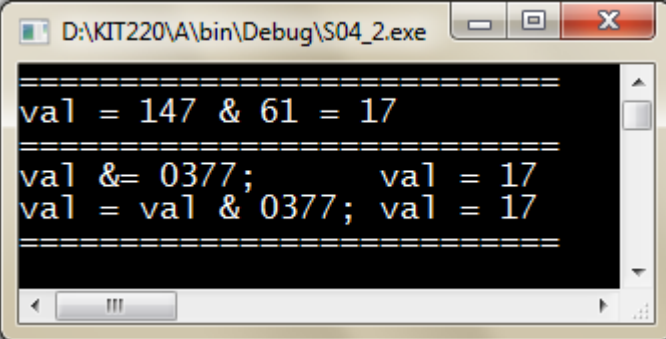
$x_1$	$x_2$	$y$
0	0	<b>0</b>
0	1	<b>0</b>
1	0	<b>0</b>
1	1	<b>1</b>

# Побітові логічні операції

```
#include <stdio.h>

int main(void)
{
    unsigned char val, val1;

    printf("=====\n");
    val = 147 & 61;
    printf("val = 147 & 61 = %d\n", val);
    printf("=====\n");
    val1 = val;
    val &= 0377;
    printf("val &= 0377;          val = %d\n", val);
    val = val1;
    val = val & 0377;
    printf("val = val & 0377; val = %d\n", val);
    printf("=====\n\n");
    return 0;
}
```



$$\begin{array}{r} 00010001_2 \\ \& 11111111_2 \\ \hline 00010001_2 \end{array}$$

$17_{10} \& 255_{10} = 17_{10}$



# Побітові логічні операції

## Побітова операція "АБО" (|)

Двійкова операція (|) створює нове значення за рахунок виконання побітового порівняння двох операндів. Для кожної позиції біт буде дорівнювати **1**, якщо будь-який з відповідних бітів в операндах дорівнює **1**.

$$\begin{array}{r} 10010011_2 \\ | \\ 00111101_2 \\ \hline 10111111_2 \end{array} \quad 147_{10} | 61_{10} = 177_{10}$$

Операція **"АБО"** може бути об'єднана з присвоюванням (|=).

```
val |= 0377;
```

дає такий самий результат, як і оператор

```
val = val | 0377;
```

Таблиця істинності

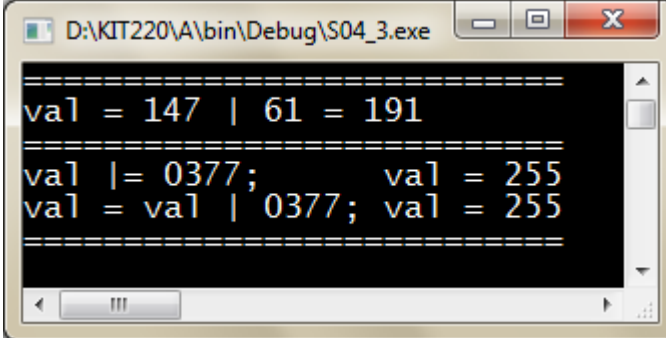
$x_1$	$x_2$	$y$
0	0	<b>0</b>
0	1	<b>1</b>
1	0	<b>1</b>
1	1	<b>1</b>

# Побітові логічні операції

```
#include <stdio.h>

int main(void)
{
    unsigned char val, val1;

    printf("=====\n");
    val = 147 | 61;
    printf("val = 147 | 61 = %d\n", val);
    printf("=====\n");
    val1 = val;
    val |= 0377;
    printf("val |= 0377;          val = %d\n", val);
    val = val1;
    val = val | 0377;
    printf("val = val | 0377; val = %d\n", val);
    printf("=====\n\n");
    return 0;
}
```



$$\begin{array}{r} 10111111_2 \\ | \\ 11111111_2 \\ \hline 11111111_2 \end{array}$$

$191_{10} | 255_{10} = 255_{10}$

# Побітові логічні операції

## Побітова операція "виключне АБО" (^)

Двійкова операція (^) виконує побітве порівняння двох операндів. Для кожної позиції підсумковий біт буде дорівнювати **1**, якщо один або інший (але не обидва) з відповідних бітів в операндах дорівнює **1**.

$$\begin{array}{r} 10010011_2 \\ \wedge \\ 00111101_2 \\ \hline 10101110_2 \end{array} \quad 147_{10} \wedge 61_{10} = 174_{10}$$

Операція "виключне АБО" може бути об'єднана з присвоюванням (^=).

```
val ^= 0377;
```

дає такий самий результат, як і оператор

```
val = val ^ 0377;
```

Таблиця істинності

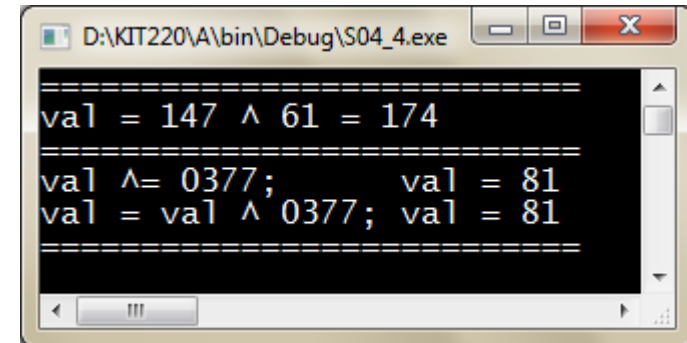
$x_1$	$x_2$	$y$
0	0	<b>0</b>
0	1	<b>1</b>
1	0	<b>1</b>
1	1	<b>0</b>

# Побітові логічні операції

```
#include <stdio.h>

int main(void)
{
    unsigned char val, val1;

    printf("=====\n");
    val = 147 ^ 61;
    printf("val = 147 ^ 61 = %d\n", val);
    printf("=====\n");
    val1 = val;
    val ^= 0377;
    printf("val ^= 0377;          val = %d\n", val);
    val = val1;
    val = val ^ 0377;
    printf("val = val ^ 0377; val = %d\n", val);
    printf("=====\n\n");
    return 0;
}
```



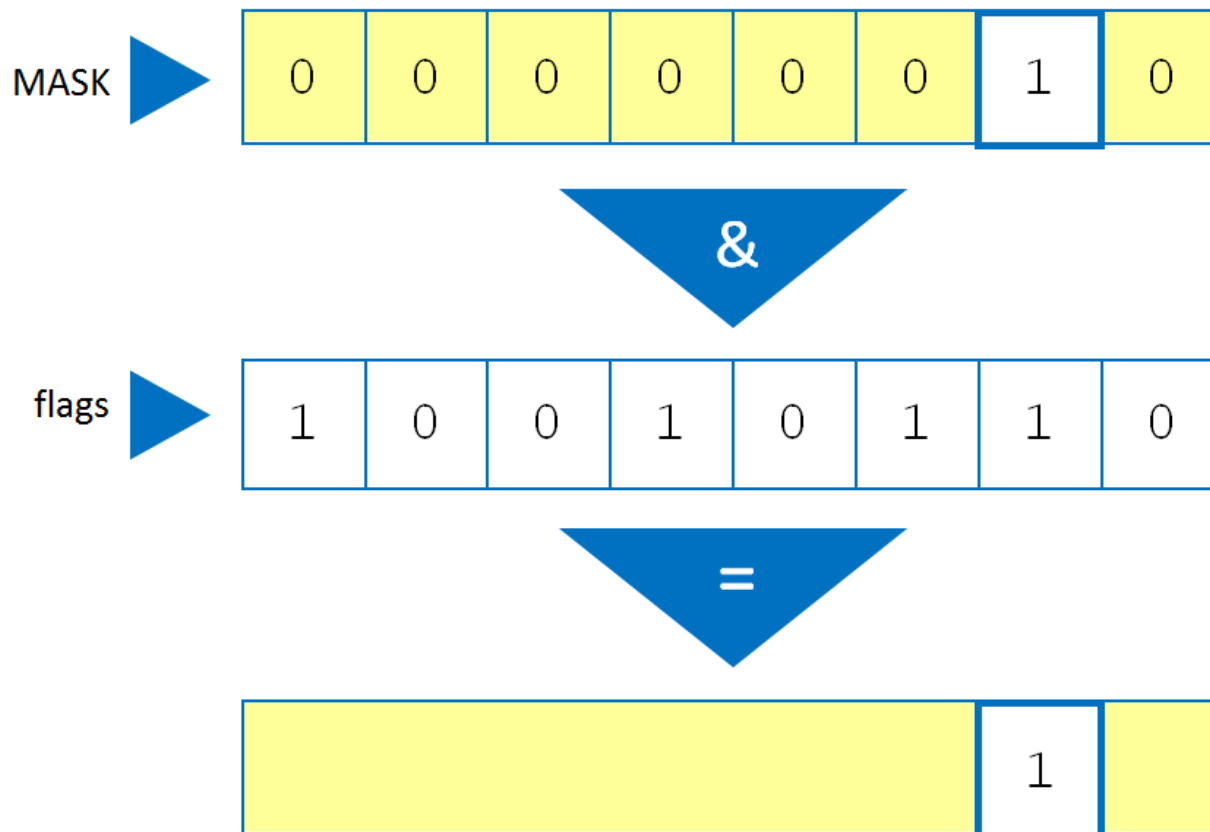
$$\begin{array}{r} \phantom{174_{10} \wedge 255_{10} = 81_{10}} \\ \phantom{174_{10} \wedge 255_{10} = 81_{10}} \phantom{0} 10101110_2 \\ \phantom{174_{10} \wedge 255_{10} = 81_{10}} \phantom{0} 11111111_2 \\ \hline \phantom{174_{10} \wedge 255_{10} = 81_{10}} \phantom{0} 01010001_2 \\ \phantom{174_{10} \wedge 255_{10} = 81_{10}} 174_{10} \wedge 255_{10} = 81_{10} \end{array}$$





# Використання маски

**Маска** – це комбінація бітів, в якій деякі біти дорівнюють 1, а деякі 0.





# Використання маски

```
#include <stdio.h>
#include <windows.h>

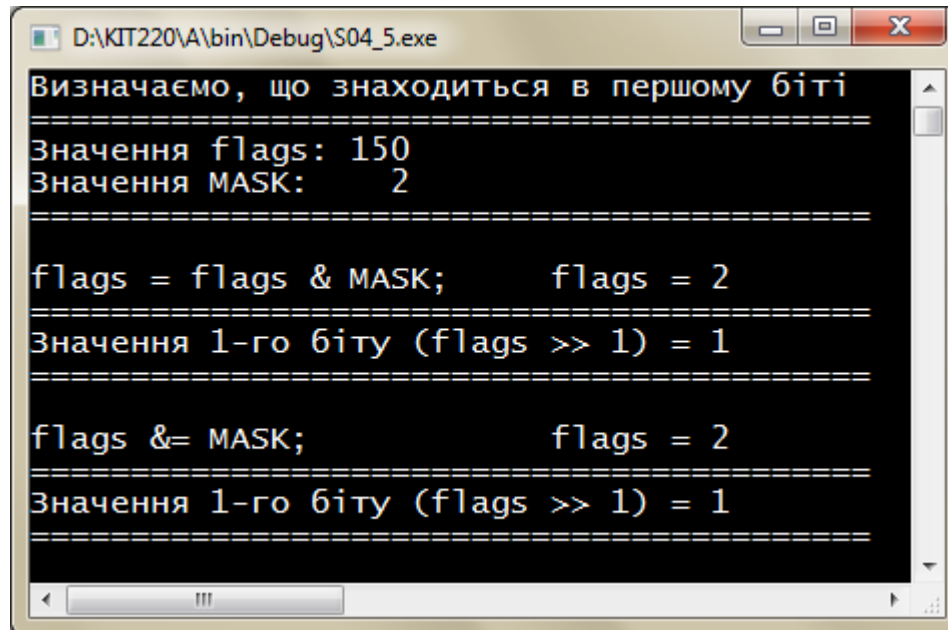
int main(void)
{
    unsigned char flags = 150;    // 10010110
    unsigned char MASK  = 2;      // 00000010
    unsigned char flags1;         // для дублювання flags

    SetConsoleOutputCP(1251);

    printf("Визначаємо, що знаходиться в першому біті\n");
    printf("=====\n");
    printf("Значення flags: %3d\n", flags);
    printf("Значення MASK:  %3d\n", MASK);
    printf("=====\n");
    flags1 = flags;
    flags = flags & MASK;
    printf("\nflags = flags & MASK;      flags = %d\n", flags);
    printf("=====\n");
```

# Використання маски

```
printf("Значення 1-го біту (flags >> 1) = %d\n", flags >> 1);
printf("=====\n");
flags1 = flags;
flags &= MASK;
printf("\nflags &= MASK;                flags = %d\n", flags);
printf("=====\n");
printf("Значення 1-го біту (flags >> 1) = %d\n", flags >> 1);
printf("=====\n");
return 0;
}
```



```
D:\KIT220\A\bin\Debug\S04_5.exe
Визначаємо, що знаходиться в першому біті
=====
Значення flags: 150
Значення MASK: 2
=====
flags = flags & MASK;    flags = 2
=====
Значення 1-го біту (flags >> 1) = 1
=====
flags &= MASK;           flags = 2
=====
Значення 1-го біту (flags >> 1) = 1
=====
```



# Встановлення бітів в 1

Іноді треба встановити окремі біти в одиничні значення, залишивши інші без змін. Наприклад, комп'ютер **IBM PC** керує обладнанням, відправляючи потрібні значення в порти. Для активізації, скажімо, динаміка, необхідно встановити в одиницю біт **1**, а інші біти залишити незмінними. Це можна зробити за допомогою побітової операції "**АБО**".

```
000011112  flags
| 101101102  MASK
-----
101111112      1510 | 18210 = 19110
```

Операція "**АБО**" може бути об'єднана з присвоюванням (**|=**).

```
flags |= MASK;
```

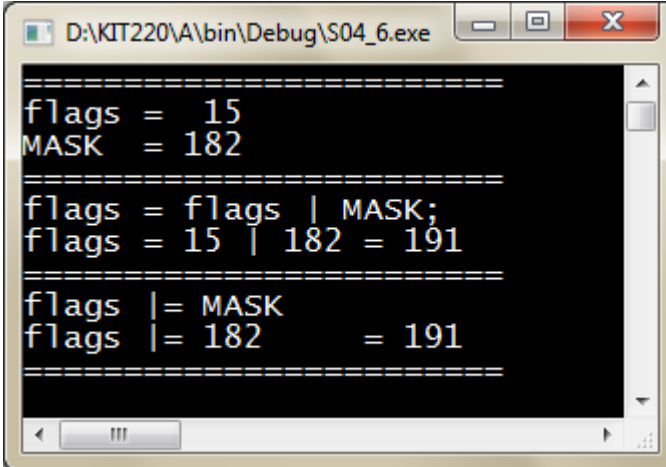
дає такий самий результат, що і оператор

```
flags = flags | MASK;
```

```
010011012  flags
| 000000102  MASK
-----
010011112
```

# Встановлення бітів в 1

```
#include <stdio.h>
int main(void) {
    unsigned char flags = 15;
    unsigned char MASK = 182;
    unsigned char flags1;
    printf("=====\n");
    printf("flags = %3d\n", flags);
    printf("MASK = %3d\n", MASK);
    printf("=====\n");
    flags1 = flags;  flags = flags | MASK;
    printf("flags = flags | MASK;\n");
    printf("flags = 15 | 182 = %d\n", flags);
    flags = flags1;  flags |= MASK;
    printf("=====\n");
    printf("flags |= MASK\n");
    printf("flags |= 182      = %d\n", flags);
    printf("=====\n\n");
    return 0;
}
```



```
=====  
flags = 15  
MASK = 182  
=====  
flags = flags | MASK;  
flags = 15 | 182 = 191  
=====  
flags |= MASK  
flags |= 182      = 191  
=====
```



# Встановлення бітів в 0

Дуже зручно мати можливість встановлювати окремі біти в нульові значення, тобто очищувати їх.

Припустимо, що треба встановити в 0 біт номер 1 в змінній `flags`. Будемо використовувати `mask`, що має встановлений в одиницю тільки біт 1.

$$\begin{array}{rcl} & 00001111_2 & \text{flags} \\ \& & \\ \sim & 10011001_2 & \text{MASK} \\ \hline & 00001001_2 & \\ & 15_{10} \& \sim 182_{10} = 9_{10} & \end{array}$$

$$\begin{array}{rcl} & 01001111_2 & \text{flags} \\ \& & \\ \sim & 00000010_2 & \text{MASK} \\ \hline & 01001101_2 & \end{array}$$

Операція "**I**" може бути об'єднана з присвоюванням (**&=**).

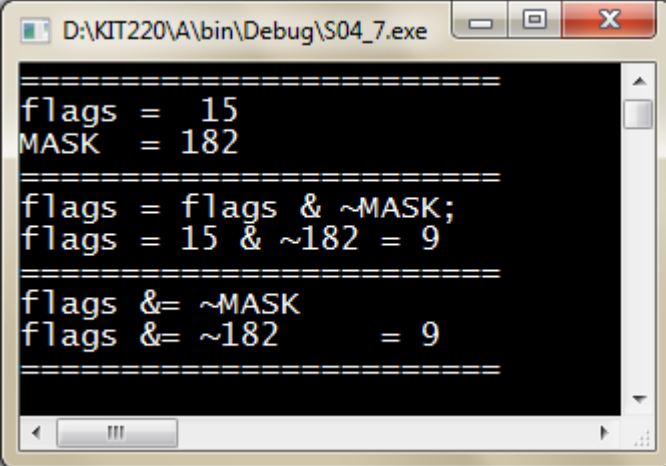
```
flags &= ~MASK;
```

дає такий самий результат, що і оператор

```
flags = flags & ~MASK;
```

# Встановлення бітів в 0

```
#include <stdio.h>
int main(void) {
    unsigned char flags = 15;
    unsigned char MASK = 182;
    unsigned char flags1;
    printf("=====\n");
    printf("flags = %3d\n", flags);
    printf("MASK = %3d\n", MASK);
    printf("=====\n");
    flags1 = flags;  flags = flags & ~MASK;
    printf("flags = flags & ~MASK;\n");
    printf("flags = 15 & ~182 = %d\n", flags);
    flags = flags1;  flags &= ~MASK;
    printf("=====\n");
    printf("flags &= ~MASK\n");
    printf("flags &= ~182      = %d\n", flags);
    printf("=====\n\n");
    return 0;
}
```



```
=====  
flags = 15  
MASK = 182  
=====  
flags = flags & ~MASK;  
flags = 15 & ~182 = 9  
=====  
flags &= ~MASK  
flags &= ~182      = 9  
=====
```



# Перемикання бітів

Перемикання біта означає зміну його значення на протилежне. Для перемикання бітів можна застосовувати побітову операцію **"виключне АБО"** (^). В результаті об'єднання значення та маски з використанням операції (^) біти, що відповідають 1 в масці, перемикаються, а біти, що відповідають 0 в масці, залишаються незмінними.

$$\begin{array}{r} \wedge \quad 0000111_2 \text{ flags} \\ \quad 1011011_2 \text{ MASK} \\ \hline 1011100_2 \end{array} \quad 15_{10} \wedge 182_{10} = 185_{10}$$

$$\begin{array}{r} \wedge \quad 0100111_2 \text{ flags} \\ \quad 0010101_2 \text{ MASK} \\ \hline 0110010_2 \end{array}$$

Операція **"виключне АБО"** може бути об'єднана з присвоюванням (^=).

```
flags ^= MASK;
```

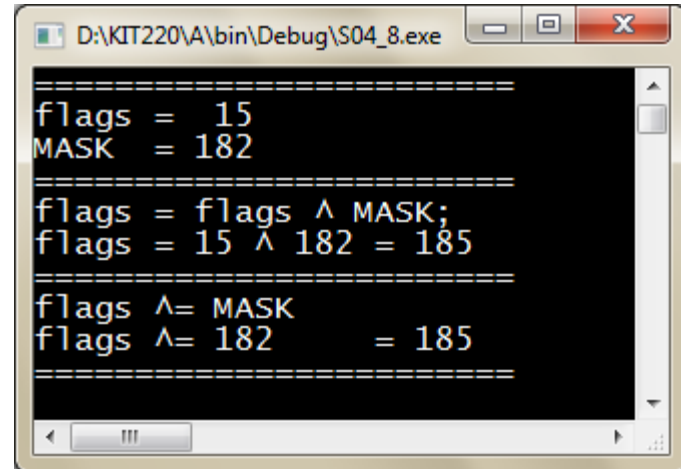
дає такий самий результат, як і оператор

```
flags = flags ^ MASK;
```



# Перемикання бітів

```
#include <stdio.h>
int main(void) {
    unsigned char flags = 15;
    unsigned char MASK = 182;
    unsigned char flags1;
    printf("=====\n");
    printf("flags = %3d\n", flags);
    printf("MASK = %3d\n", MASK);
    printf("=====\n");
    flags1 = flags;  flags = flags ^ MASK;
    printf("flags = flags ^ MASK;\n");
    printf("flags = 15 ^ 182 = %d\n", flags);
    flags = flags1;  flags ^= MASK;
    printf("=====\n");
    printf("flags ^= MASK\n");
    printf("flags ^= 182      = %d\n", flags);
    printf("=====\n\n");
    return 0;
}
```



```
D:\KIT220\A\bin\Debug\S04_8.exe
=====  
flags = 15  
MASK = 182  
=====  
flags = flags ^ MASK;  
flags = 15 ^ 182 = 185  
=====  
flags ^= MASK  
flags ^= 182      = 185  
=====
```



# Побітові операції зсуву

## Операція "зсув вліво" (<<)

Операція (<<) зсуває біти значення лівого операнда вліво на кількість позицій, яка задається правим операндом. Позиції, що звільняються, заповнюються 0, а біти, що виходять за межі значення лівого операнда, втрачаються.

$$10001010_2 \ll 2_{10} = 00101000_2 \quad ( 138_{10} \ll 2_{10} = 40_{10} )$$

Щоб змінити значення змінної, можна скористатися операцією зсуву вліво з присвоюванням (<<=). Ця операція зсуває біти змінної вліво на кількість позицій, яка вказана в правому операнді.

```
int stonk = 1;
```

```
int on;
```

```
on = stonk << 2;
```

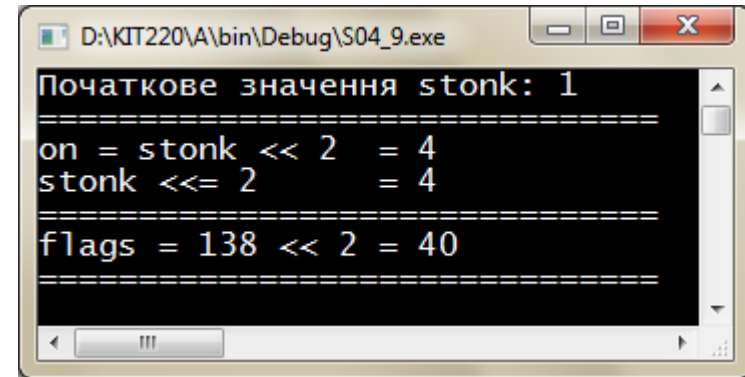
```
// присвоює 4 змінній on
```

```
stonk <<= 2;
```

```
// змінює значення stonk на 4
```

# Побітові операції зсуву

```
#include <stdio.h>
#include <windows.h>
int main(void) {
    unsigned char stonk = 1;
    unsigned char on;
    unsigned char flags;
    SetConsoleOutputCP(1251);
    printf("Початкове значення stonk: %d\n", stonk);
    printf("=====\n");
    on = stonk << 2;    // присвоює 4 змінній on
    printf("on = stonk << 2  = %d\n", on);
    stonk <<= 2;
    printf("stonk <<= 2      = %d\n", stonk);
    flags = 138 << 2;
    printf("=====\n");
    printf("flags = 138 << 2 = %d\n", flags);
    printf("=====\n\n");
    return 0;
}
```



```
D:\KIT220\A\bin\Debug\S04_9.exe
Початкове значення stonk: 1
=====
on = stonk << 2  = 4
stonk <<= 2      = 4
=====
flags = 138 << 2 = 40
=====
```



# Побітові операції зсуву

## Операція "зсув вправо" ( $\gg$ )

Операція ( $\gg$ ) робить зсув бітів значення лівого операнда вправо на кількість позицій, яка вказується в правому операнді. Біти, які виходять за праву межу лівого операнда, втрачаються. Для типів даних без знаку, позиції, які звільняються зліва, заповнюються 0. Для типів даних зі знаком, результат залежить від системи. Позиції, що звільняються, можуть заповнюватися 0 або бітом знаку.

$$10001010_2 \gg 2_{10} = 00100010_2 \quad ( 138_{10} \gg 2_{10} = 34_{10} )$$

Операція ( $\gg=$ ) робить зсув вправо бітів лівого операнда на задану в правому операнді кількість позицій.

```
int sweet = 16;
```

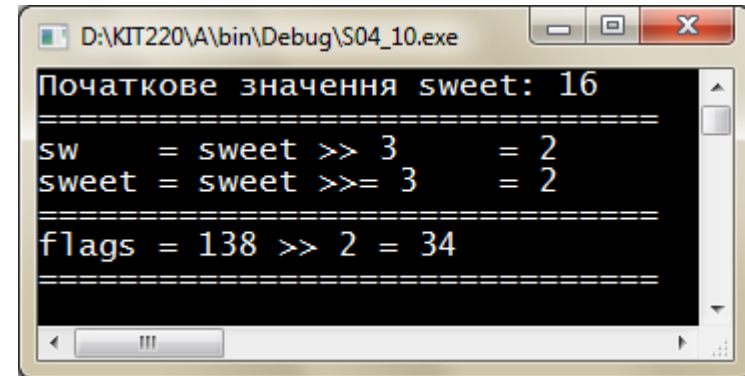
```
int sw;
```

```
sw = sweet >> 3;           // sw дорівнює 2, sweet як і раніше 16
```

```
sweet >>= 3;              // значення sweet змінилося на 2
```

# Побітові операції зсуву

```
#include <stdio.h>
#include <windows.h>
int main(void) {
    unsigned char sweet = 16;
    unsigned char sw;
    unsigned char flags;
    SetConsoleOutputCP(1251);
    printf("Початкове значення sweet: %d\n", sweet);
    printf("=====\n");
    sw = sweet >> 3; // sw дорівнює 2, sweet як і раніше 16
    printf("sw      = sweet >> 3      = %d\n", sw);
    sweet >>= 3;
    printf("sweet = sweet >>= 3      = %d\n", sweet);
    flags = 138 >> 2;
    printf("=====\n");
    printf("flags = 138 >> 2 = %d\n", flags);
    printf("=====\n\n");
    return 0;
}
```



```
D:\KIT220\A\bin\Debug\S04_10.exe
Початкове значення sweet: 16
=====
sw      = sweet >> 3      = 2
sweet = sweet >>= 3      = 2
=====
flags = 138 >> 2 = 34
=====
```



# Побітові операції зсуву

Побітові операції зсуву є зручним та ефективним (в залежності від обладнання) засобом виконання **множення** та **ділення** на степінь 2:

```
number << n      // Помножує number на 2 в степені n
number >> n      // Ділить number на 2 в степені n,
                 // якщо значення number ≥ 0
```

Операції зсуву можуть також використовуватися для **отримання груп бітів** з більш великих конструкцій.

```
#define BYTE_MASK 0xff

unsigned long color = 0x002a162f;
unsigned char blue, green, red;

red   = color & BYTE_MASK;
green = ( color >> 8 ) & BYTE_MASK;
blue  = ( color >> 16 } & BYTE_MASK;
```



# Побітові операції зсуву

```
#include <stdio.h>
#include <windows.h>
#include <limits.h> // для CHAR_BIT кількість бітів на символ

char *itobs(int, char *);
void show_bstr(const char *);

int main(void)
{
    char bin_str[CHAR_BIT * sizeof(int) + 1];
    int number;

    SetConsoleOutputCP(1251);
    puts("Введіть ціле число.");
    puts("Якщо ввести не число, програма завершиться.\n");
    while(scanf("%d", &number) == 1)
    {
        itobs(number, bin_str);
        printf("Двійкове представлення: ");
        show_bstr(bin_str);
    }
}
```



# Побітові операції зсуву

```
        putchar('\n');
    }
    puts("Програма завершена.");
    return 0;
}

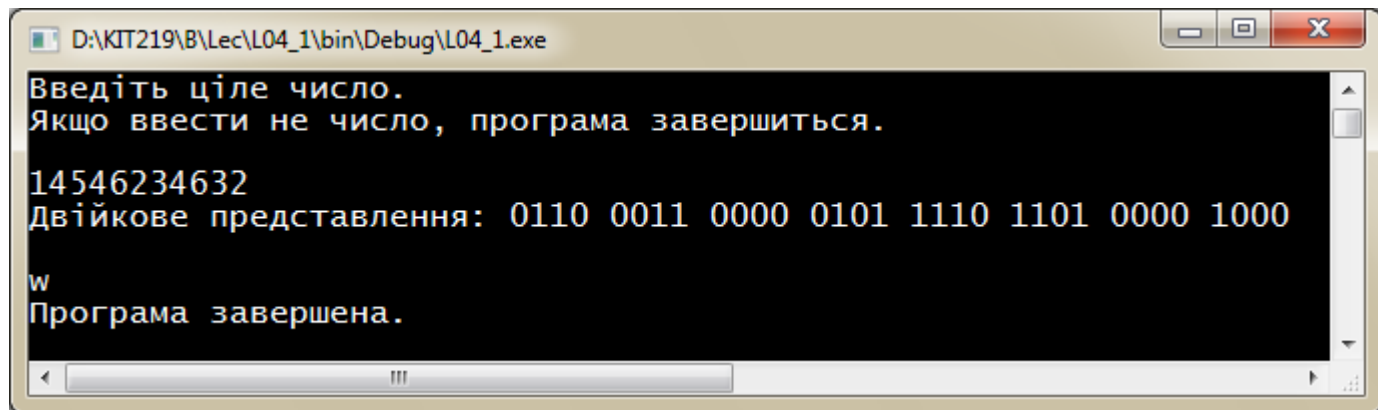
char *itobs(int n, char *ps)
{
    int i;
    const static int size = CHAR_BIT * sizeof(int);

    for(i = size - 1; i >= 0; i--, n >>= 1)
        // передбачається кодування ASCII або схоже
        ps[i] = (01 & n) + '0';
    ps[size] = '\0';
    return ps;
}
```



# Побітові операції зсуву

```
// відображення двійкового рядка блоками по 4
void show_bstr(const char *str)
{
    int i = 0;
    while(str[i])    // поки не буде отриманий нульовий символ
    {
        putchar(str[i]);
        if( ++i % 4 == 0 && str[i])
            putchar(' ');
    }
    printf("\n");
}
```



```
D:\KIT219\B\Lec\L04_1\bin\Debug\L04_1.exe
Введіть ціле число.
Якщо ввести не число, програма завершиться.
14546234632
Двійкове представлення: 0110 0011 0000 0101 1110 1101 0000 1000
w
Програма завершена.
```



# Засоби вирівнювання (C11)

Операція `_Alignof` висуває вимоги до вирівнювання вказаного типу. Для її використання необхідно після ключового слова `_Alignof` помістити ім'я типу в круглих дужках:

```
size_t d_align = _Alignof(float);
```

За допомогою специфікатора `_Alignas` можна робити запит про конкретне вирівнювання для змінної або типу. Однак ви не повинні робити запит на вирівнювання, яке менше фундаментального вирівнювання, прийнятого для типу.

Цей специфікатор застосовується як частина оголошення, і за ним йде пара круглих дужок, яка містить або значення вирівнювання, або тип:

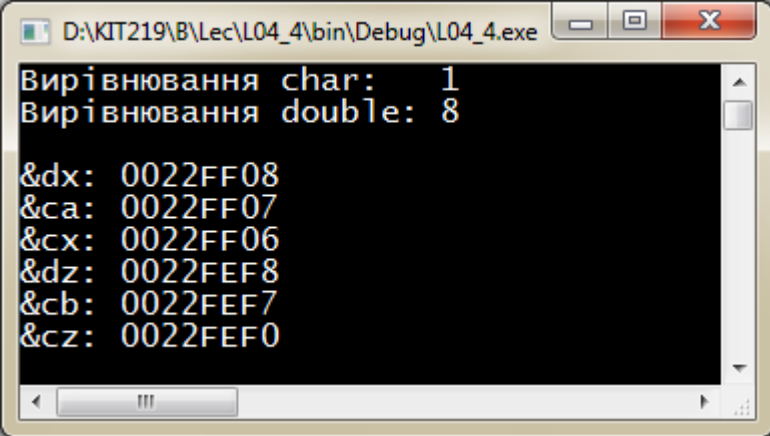
```
_Alignas(double) char c1;
```

```
_Alignas(8) char c2;
```

```
unsigned char _Alignas(long double) c_arr[sizeof(long double)];
```

# Засоби вирівнювання (C11)

```
#include <stdio.h>
#include <windows.h>
int main(void)
{
    double dx;
    char ca, cx;
    double dz;
    char cb, _Alignas(double) cz;
    SetConsoleOutputCP(1251);
    printf("Вирівнювання char: %d\n", _Alignof(char));
    printf("Вирівнювання double: %d\n\n", _Alignof(double));
    printf("&dx: %p\n", &dx);
    printf("&ca: %p\n", &ca);
    printf("&cx: %p\n", &cx);
    printf("&dz: %p\n", &dz);
    printf("&cb: %p\n", &cb);
    printf("&cz: %p\n", &cz);
    return 0;
}
```



```
D:\KIT219\B\Lec\L04_4\bin\Debug\L04_4.exe
Вирівнювання char: 1
Вирівнювання double: 8

&dx: 0022FF08
&ca: 0022FF07
&cx: 0022FF06
&dz: 0022FEF8
&cb: 0022FEF7
&cz: 0022FEF0
```