

# Вказівники



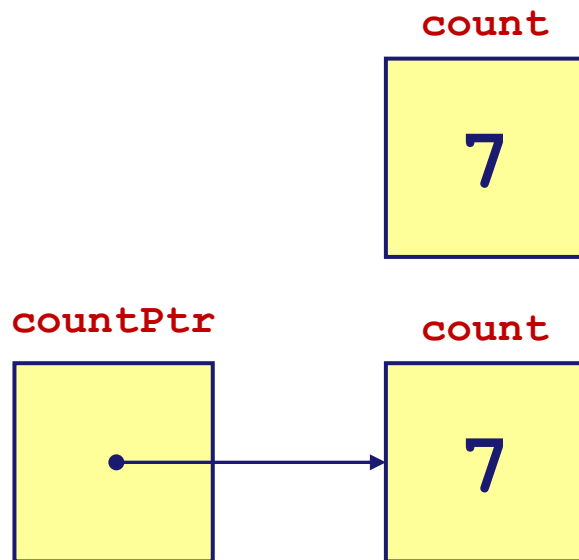
*Лекція №12*

*Дисципліна «Програмування»*



# Визначення вказівника

**Вказівник** – це змінна, значенням якої є адреса в пам'яті. Звичайні змінні містять безпосередні значення. Вказівник містить адресу змінної, яка зберігає безпосереднє значення. Ім'я змінної є безпосереднім посиланням на значення, а вказівник – непрямым. Доступ до значення за допомогою вказівника має назву **непрямого доступу**.



Ім'я `count` напряду посилається на змінну зі значенням `7`

Вказівник `countPtr` опосередковано посилається на змінну зі значенням `7`



# Оголошення вказівника

Визначення:

```
int *countPtr;
```

оголошує змінну типу **int \*** (тобто вказівник на ціле число) і читається як «**countPtr** – вказівник на значення типу **int**» або «**countPtr** вказує на об'єкт типу **int**».

Коли символ '**\***' використовується у визначенні таким чином, він повідомляє, що змінна є вказівником.

Вказівники можуть вказувати на об'єкти будь-яких типів.

Вказівники можуть ініціалізуватися в момент оголошення, а також їм можна присвоювати значення вже в ході виконання.

Вказівник може бути ініціалізований значенням **NULL**, **0** або **адресою**.



# Оператори вказівників

Амперсанд (&), або **оператор взяття адреси**, – це унарний оператор, що повертає адресу свого операнда.

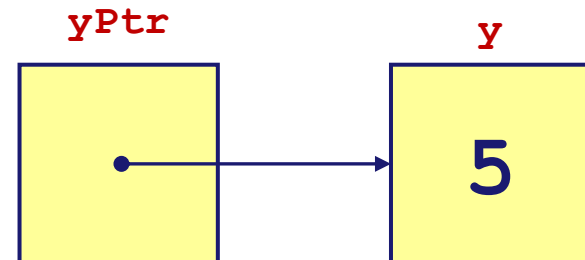
Наприклад, припустимо, що є наступні визначення:

```
int y = 5;  
int *yPtr;
```

Тоді інструкція

```
yPtr = &y;
```

присвоїть змінній-вказівнику **yPtr** адресу змінної **y**. Після цього про змінну **yPtr** можна сказати, що вона «вказує на» змінну **y**.



Схематичне представлення пам'яті  
після виконання присвоювання

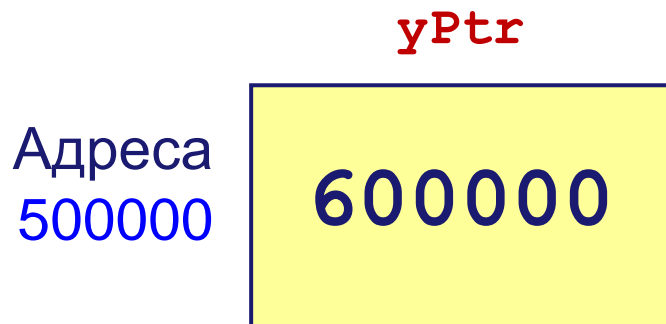


# Оператор непрямого звернення (\*)

В даному випадку передбачається, що цілочислова змінна **y** зберігається в пам'яті за адресою **600000**, а змінна-вказівник **yPtr** – за адресою **500000**.

Операнд оператора взяття адреси **&** обов'язково повинен бути змінною.

Оператор **&** не може застосовуватися до констант і виразів.





# Оператор непрямого звернення (\*)

Унарний оператор **\***, що часто має назву **оператора непрямого звернення** або **оператора розіменування**, повертає значення об'єкта, на який вказує його операнд (тобто вказівник). Наприклад, інструкція

```
printf("%d", *yPtr);
```

виведе значення змінної **y**, а саме число **5**. Такий спосіб використання унарного оператора **\*** має назву **розіменування вказівника**.

Розіменування неініціалізованого вказівника або вказівника, якому не була присвоєна реальна адреса змінної в пам'яті, є помилкою. Це може привести до аварійного завершення програми або до пошкодження важливих даних і отримання помилкових результатів.



# Використання операторів & і \*

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

int main(void)
{
    int a;          // a - цілочислова змінна
    int *aPtr;      // aPtr - вказівник на цілочислову змінну

    SetConsoleOutputCP(1251);

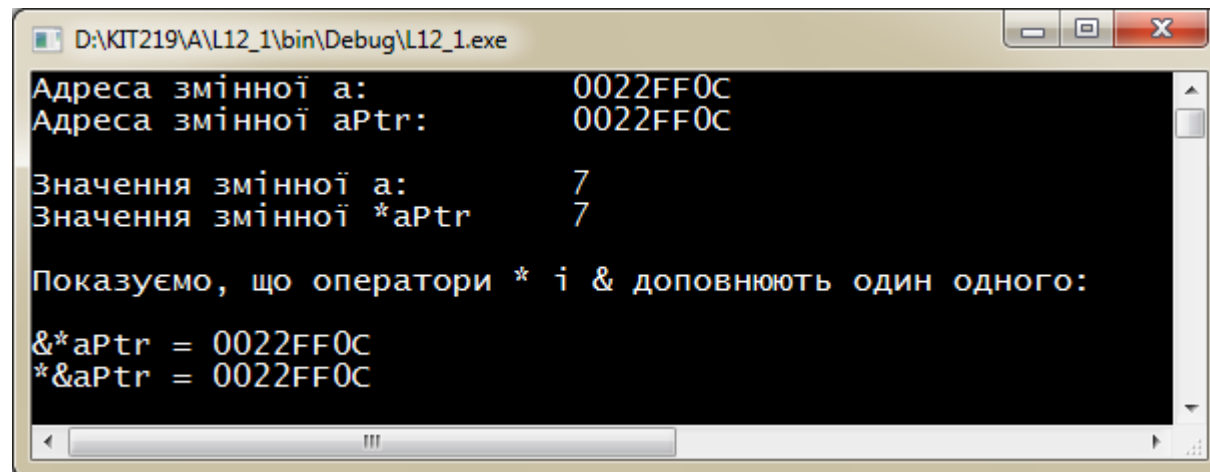
    a      = 7;
    aPtr = &a;      // присвоїти змінній aPtr адресу змінної a

    printf("Адреса змінної a:           %p\n"
           "Адреса змінної aPtr:        %p", &a, aPtr);

    printf("\n\nЗначення змінної a:           %d\n"
           "Значення змінної *aPtr          %d", a, *aPtr);
}
```

# Використання операторів & і \*

```
printf("\n\nПоказуємо, що оператори * і & доповнюють"  
      " один одного:\n\n&*aPtr = %p\n"  
      "&aPtr = %p\n", &aPtr, *&aPtr);  
  
return 0;  
}
```



```
D:\KIT219\A\L12_1\bin\Debug\L12_1.exe  
Адреса змінної a: 0022FF0C  
Адреса змінної aPtr: 0022FF0C  
Значення змінної a: 7  
Значення змінної *aPtr 7  
Показуємо, що оператори * і & доповнюють один одного:  
&*aPtr = 0022FF0C  
*&aPtr = 0022FF0C
```





# Демонстрація операторів & і \*

```
#include <stdio.h>
#include <windows.h>
int main(void)
{
    int a;          // a - цілочислова змінна
    int *aPtr;      // aPtr - вказівник на цілочислову змінну

    SetConsoleOutputCP(1251);
    a = 7;
    aPtr = &a;      // присвоїти змінній aPtr адресу змінної a

    printf("Адреса змінної a:          %p\n"
           "Адреса змінної aPtr:        %p", &a, aPtr);
    printf("\n\nЗначення змінної a:          %d\n"
           "Значення змінної *aPtr          %d", a, *aPtr);
    printf("\n\nПоказуємо, що оператори * і & доповнюють"
           " один одного:\n\n&*aPtr = %p\n"
           "*&aPtr = %p\n", &*aPtr, *&aPtr);
    return 0;
}
```



# Передавання аргументів функціям за посиланням

Існує два способи передавання аргументів функціям: **за значенням** і **за посиланням**.

Усі аргументи в мові **C** передаються за значенням.

Часто необхідно мати можливість модифікувати змінні, які оголошені в функції, яка викликає, або передавати вказівник на великий об'єкт даних, щоб уникнути накладних витрат на копіювання цього об'єкта при передачі за значенням (що вимагає часу і пам'яті для зберігання копії об'єкта).

У мові **C** вказівники та оператор непрямого доступу використовуються для імітації передачі аргументів за посиланням.



# Передавання аргументів функціям за посиланням

Коли функція повинна змінити значення аргументу, їй передається адреса цього аргументу.

Зазвичай це реалізується застосуванням оператора взяття адреси **&** до змінної, значення якої повинно бути змінено.

Масиви передаються до функцій без застосування оператора **&**, тому що компілятор мови **C** автоматично передає масив як адресу першого його елемента (ім'я масиву діє еквівалентно виразу **&arrayName[0]**).

Коли функції передається адреса змінної, всередині неї можна використовувати оператор непрямого доступу **\***, щоб змінити значення, яке зберігається за цією адресою.



# Передавання за значенням

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

int cubeByValue(int n);    // прототип функції

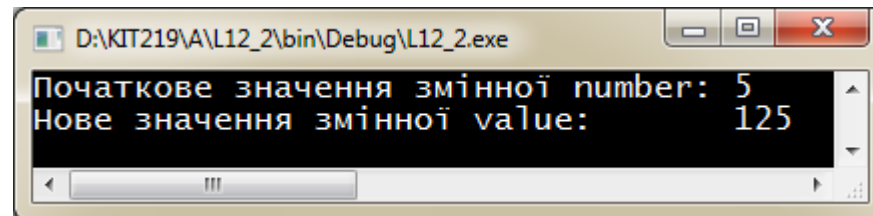
int main(void)
{
    int number = 5;        // ініціалізувати змінну number

    SetConsoleOutputCP(1251);

    printf("Початкове значення змінної number: %d", number);
    // передати number функції cubeByValue() по значенню
    number = cubeByValue(number);
    printf("\nНове значення змінної value:          %d\n",
           number);
    return 0;
}
```

# Передавання за значенням

```
int cubeByValue(int n)
{
    return n * n * n;    // піднести число до третьої степені
                        // та повернути результат
}
```





# Передавання за посиланням

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

void cubeByReference(int *nPtr); // прототип функції

int main(void)
{
    int number = 5;                // ініціалізуємо змінну number

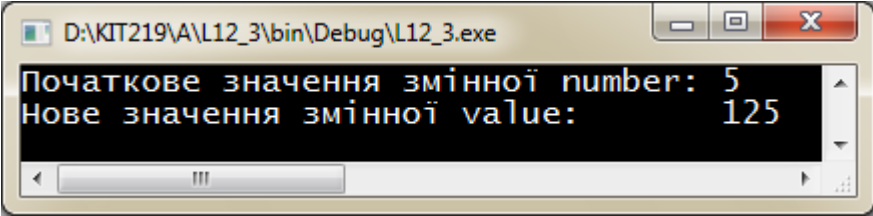
    SetConsoleOutputCP(1251);

    printf("Початкове значення змінної number: %d", number);
    // передати функції cubeByReference() адресу
    // змінної number
    cubeByReference(&number);
    printf("\nНове значення змінної value:      %d\n",
           number);
    return 0;
}
```

# Передавання за посиланням

```
// підносить *nPtr до третьої степені  
// фактично змінює number
```

```
void cubeByReference(int *nPtr)  
{  
    // піднести *nPtr до третьої степені  
    *nPtr = (*nPtr) * (*nPtr) * (*nPtr);  
}
```



```
D:\KIT219\A\L12_3\bin\Debug\L12_3.exe  
Початкове значення змінної number: 5  
Нове значення змінної value: 125
```



# Використання одновимірних масивів в якості аргументів

Якщо функція приймає в якості аргументу одновимірний масив, в списку параметрів у прототипі та в заголовку функції можна використовувати форму запису вказівників, як у заголовку функції `cubeByReference()`. Компілятор **C** не розрізняє ситуації, коли функції приймають вказівники та одновимірні масиви. Однак сама функція повинна «знати», що вона приймає – одновимірний масив чи адресу єдиної змінної. Коли компілятор зустрічає визначення параметра у формі одновимірного масиву `int b[ ]`, він перетворює його на визначення параметра вказівника `int *b`. Ці дві форми є взаємопов'язані.

Використовуйте передавання аргументів за значенням, якщо функції, яка викликає, явно не треба, щоб функція, яка викликається, змінювала значення змінної, яка передається.





# Використання кваліфікатора `const` з вказівниками

Кваліфікатор `const` дає можливість повідомити компілятор про те, що значення позначеної ним змінної не повинне змінюватися.

Застосування кваліфікатора `const` допоможе скоротити час на відлагодження програми, усунути небажані побічні ефекти та спростити супровід і подальший розвиток програми.

## Застосування `const` до параметрів функцій

Існує шість варіантів застосування (або незастосування) кваліфікатора `const` до параметрів функцій – два варіанти до параметрів, які передаються за значенням, і чотири варіанти до параметрів, які передаються за посиланням.



# Передавання змінюваного вказівника на змінювані дані

Кваліфікатор **const** дає можливість повідомити компілятор про те, що значення позначеної ним змінної не повинне змінюватися.

Застосування кваліфікатора **const** допоможе скоротити час на відлагодження програми, усунути небажані побічні ефекти та спростити супровід і подальший розвиток програми.

## Застосування **const** до параметрів функцій

Існує шість варіантів застосування (або незастосування) кваліфікатора до параметрів функцій – два варіанти до параметрів, які передаються за значенням, і чотири варіанти до параметрів, які передаються за посиланням.



# Передавання змінюваного вказівника на змінювані дані

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

void convertToUppercase(char *sPtr);    // прототип функції

int main(void)
{
    char string[] = "cHaRaCters and $32.98";

    SetConsoleOutputCP(1251);

    printf("Рядок до перетворення:    %s", string);
    convertToUppercase(string);
    printf("\nРядок після перетворення: %s\n", string);
    return 0;
}
```



# Передавання змінюваного вказівника на змінювані дані

// перетворює символи рядка у верхній регістр

```
void convertToUppercase(char *sPtr)
{
    while(*sPtr != '\0')
    {
        *sPtr = toupper(*sPtr);    // перетворити у верхній
                                   // регістр
        ++sPtr;                    // перейти до наступного
                                   // символу
    }
}
```

The screenshot shows a command prompt window with the title bar 'D:\KIT219\A\L12\_4\bin\Debug\L12\_4.exe'. The window contains two lines of text: 'Рядок до перетворення: cHaRaCters and \$32.98' and 'Рядок після перетворення: CHARACTERS AND \$32.98'. The first line shows the original string with mixed case, and the second line shows the result after the function has been applied, where all letters are now uppercase.



# Передавання змінюваного вказівника на константні дані

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

void printCharacters(const char *sPtr);

int main(void)
{
    // ініціалізація масиву символів
    char string[] = "print characters of a string";

    SetConsoleOutputCP(1251);

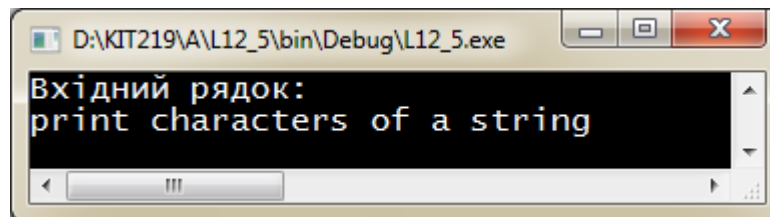
    puts("Вхідний рядок:");
    printCharacters(string);
    puts("");
    return 0;
}
```



# Передавання змінюваного вказівника на константні дані

```
// вказівник sPtr не може використовуватися  
// для зміни символу, на який він вказує,  
// тобто sPtr - вказівник "тільки для читання"
```

```
void printCharacters(const char *sPtr)  
{  
    for( ; *sPtr != '\0'; ++sPtr)  
        printf("%c", *sPtr);  
}
```





# Аргументи за посиланням

```
#include <stdio.h>
#include <windows.h>
#define SIZE 10

void bubbleSort(int *const array, size_t size); // прототип

int main(void)
{
    int a[SIZE] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };
    size_t i; // лічильник

    SetConsoleOutputCP(1251);
    puts("Порядок елементів вхідного масиву");
    for(i = 0; i < SIZE; i++) printf("%4d", a[i]);
    bubbleSort(a, SIZE); // відсортувати масив
    puts("\n\nПорядок елементів вихідного масиву");
    for(i = 0; i < SIZE; i++) printf("%4d", a[i]);
    puts("");
    return 0;
}
```



# Аргументи за посиленням

```
// сортує масив чисел з використанням алгоритму
// бульбашкового сортування

void bubbleSort(int *const array, size_t size)
{
    void swap(int *element1Ptr, int *element2Ptr);

    unsigned int pass;           // лічильник проходів
    size_t j;                     // лічильник порівнянь

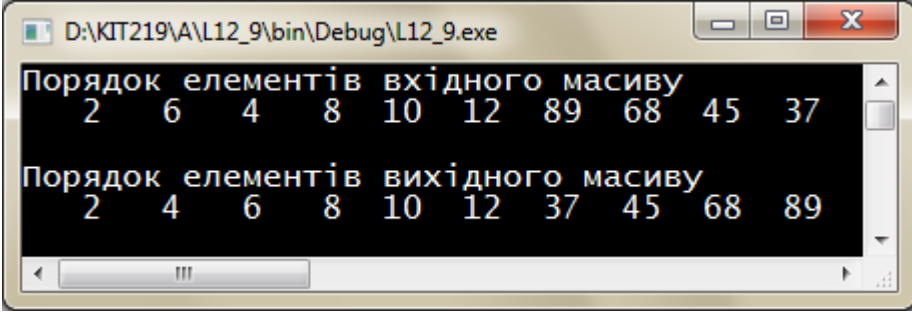
    for(pass = 0; pass < size - 1; pass++)
    {
        for(j = 0; j < size - 1; j++)
        {
            if(array[j] > array[j+1])
                swap(&array[j], &array[j+1]);
        }
    }
}
```



# Аргументи за посиланням

```
// міняє місцями значення в пам'яті, які адресуються  
// вказівниками element1Ptr и element2Ptr
```

```
void swap(int *element1Ptr, int *element2Ptr)  
{  
    int hold = *element1Ptr;  
  
    *element1Ptr = *element2Ptr;  
    *element2Ptr = hold;  
}
```



```
D:\KIT219\A\L12_9\bin\Debug\L12_9.exe  
Порядок елементів вхідного масиву  
2 6 4 8 10 12 89 68 45 37  
Порядок елементів вихідного масиву  
2 4 6 8 10 12 37 45 68 89
```



# Арифметика вказівників

Вказівники є допустимими операндами в арифметичних виразах, виразах присвоювання та порівняння. Однак далеко не всі оператори, що зазвичай застосовуються у виразах, можуть використовуватися у виразах з вказівниками.

- 1) Вказівники можуть збільшуватися за допомогою оператора інкремента (**++**) або зменшуватися за допомогою оператора декремента (**--**).
- 2) До вказівників можна додавати цілі числа (**+** і **+=**).
- 3) З вказівників можна віднімати цілі числа (**-** і **-=**).
- 4) Один вказівник може відніматися від іншого.

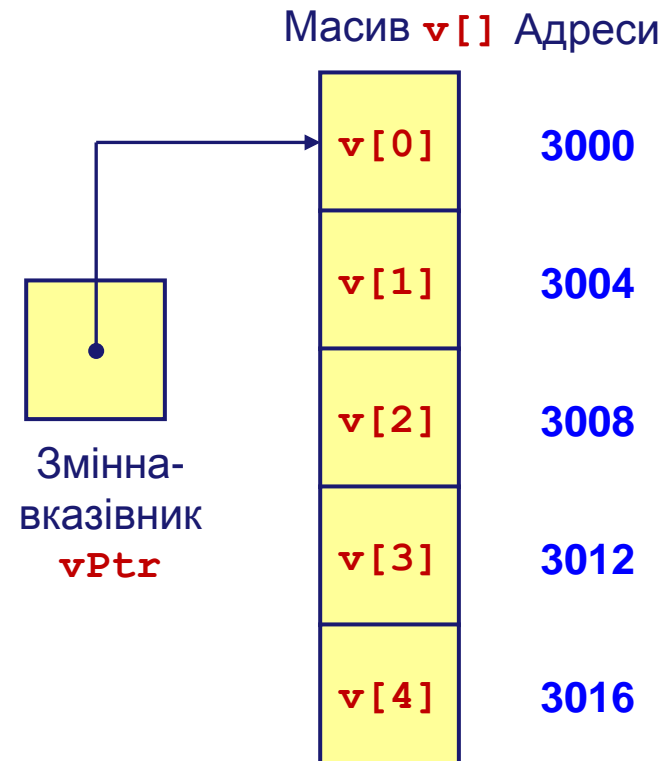
Остання операція має сенс, тільки якщо обидва вказівника вказують на елементи одного й того ж самого масиву.

# Арифметика вказівників

Припустимо, що визначено масив `int v[5]`, а його перший елемент зберігається у пам'яті за адресою `3000`. Припустимо також, що вказівник `vPtr` ініціалізований адресою першого елемента `v[0]`, тобто `vPtr` має значення `3000` і під цілі числа відводиться `4` байти. Тоді змінна `vPtr` може бути ініціалізована вказівником на масив `v` однією з наступних інструкцій:

```
vPtr = v;
```

```
vPtr = &v[0];
```





# Арифметика вказівників

У звичайній арифметиці вираз  $3000 + 2$  дає в результаті  $3002$ .

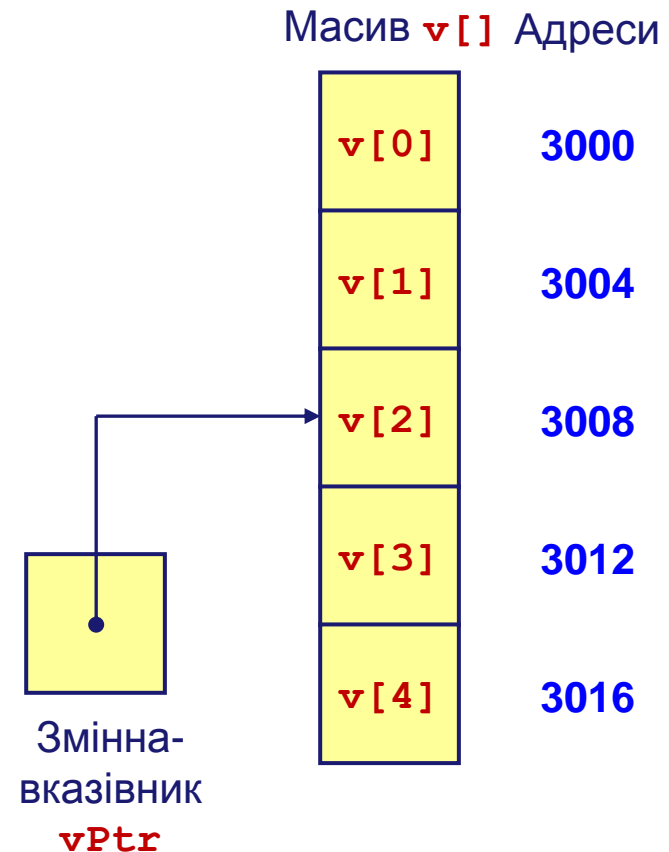
В арифметиці вказівників вказівник збільшується або зменшується на число помножене на розмір об'єкта, на який посилається вказівник.

Наприклад, інструкція

```
vPtr += 2;
```

присвоїть змінній **vPtr** адресу  $3008$  ( $3000 + 2 * 4$ ), якщо виходити з припущення, що значення типу **int** займає **4** байти.

В нашому випадку **vPtr** буде вказувати на елемент масиву **v[2]**.





# Масиви та вказівники

```
#include <stdio.h>
#include <windows.h>
#define ARRAY_SIZE 4

int main(void)
{
    // створити та ініціалізувати масив b
    int b[] = { 10, 20, 30, 40 };
    // створити вказівник і присвоїти йому адресу масиву b
    int *bPtr = b;
    size_t i;                // лічильник
    size_t offset;           // лічильник
    SetConsoleOutputCP(1251);
    // вивести масив b з використанням індексів
    puts("Масив b:\n\nЗапис за допомогою індексів");
    // цикл по елементам масиву b
    for(i = 0; i < ARRAY_SIZE; ++i)
        printf("b[%u] = %d;\n", i, b[i]);
```



# Масиви та вказівники

```
// використання форми запису вказівник/зміщення
puts("\nЗапис вказівник/зміщення, де вказівником"
     " є ім'я масиву");

for(offset = 0; offset < ARRAY_SIZE; offset++)
    printf("(b + %u) = %d;\n", offset, *(b + offset));
// вивести масив b з використанням bPtr і індексів
puts("\nЗапис вказівник/індекс");

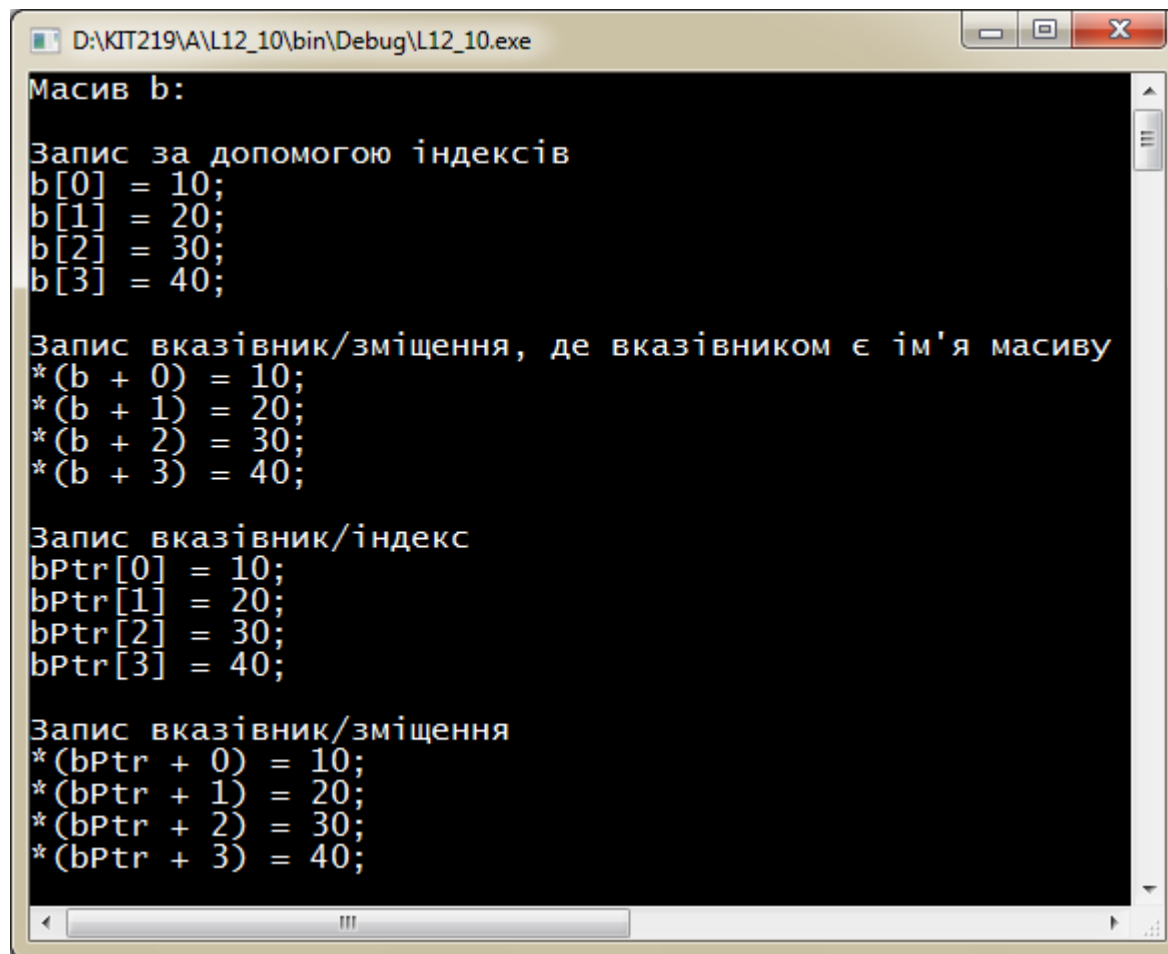
for(i = 0; i < ARRAY_SIZE; ++i)
    printf("bPtr[%u] = %u;\n", i, bPtr[i]);
// використання bPtr і форми запису вказівник/зміщення
puts("\nЗапис вказівник/зміщення");

for(offset = 0; offset < ARRAY_SIZE; ++offset)
    printf("(bPtr + %u) = %d;\n",
           offset, *(bPtr + offset));

return 0;

}
```

# Масиви та вказівники



```
D:\KIT219\A\L12_10\bin\Debug\L12_10.exe

Масив b:

Запис за допомогою індексів
b[0] = 10;
b[1] = 20;
b[2] = 30;
b[3] = 40;

Запис вказівник/зміщення, де вказівником є ім'я масиву
*(b + 0) = 10;
*(b + 1) = 20;
*(b + 2) = 30;
*(b + 3) = 40;

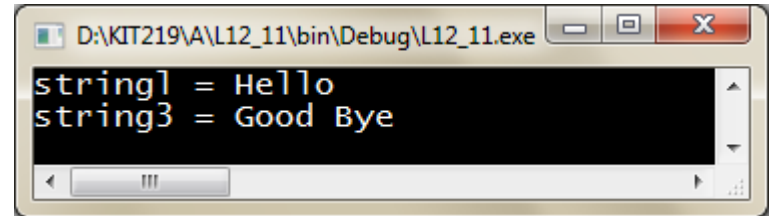
Запис вказівник/індекс
bPtr[0] = 10;
bPtr[1] = 20;
bPtr[2] = 30;
bPtr[3] = 40;

Запис вказівник/зміщення
*(bPtr + 0) = 10;
*(bPtr + 1) = 20;
*(bPtr + 2) = 30;
*(bPtr + 3) = 40;
```



# Передавання масивів і вказівників

```
#include <stdio.h>
#define SIZE 10
```



```
void copy1(char * const s1, const char * const s2); // прототип
void copy2(char *s1, const char *s2); // прототип
```

```
int main(void)
{
    char string1[SIZE]; // створити масив string1
    char *string2 = "Hello"; // створити вказівник на рядок
    char string3[SIZE]; // створити масив string3
    char string4[] = "Good Bye"; // створити вказівник на рядок
    copy1(string1, string2);
    printf("string1 = %s\n", string1);
    copy2(string3, string4);
    printf("string3 = %s\n", string3);
    return 0;
}
```





# Застосування масивів і вказівників

// копіює s2 в s1 з використанням форми звернення до масиву

```
void copy1(char * const s1, const char * const s2)
```

```
{
```

```
    size_t i;        // лічильник
```

```
    for(i = 0; (s1[i] = s2[i]) != '\0'; ++i)
```

```
    {
```

```
        ; // порожнє тіло циклу
```

```
    }
```

```
}
```

// копіює s2 в s1 з використанням форми звернення до вказівника

```
void copy2(char *s1, const char *s2)
```

```
{
```

```
    for( ; (*s1 = *s2) != '\0'; ++s1, ++s2)
```

```
    {
```

```
        ; // порожнє тіло циклу
```

```
    }
```

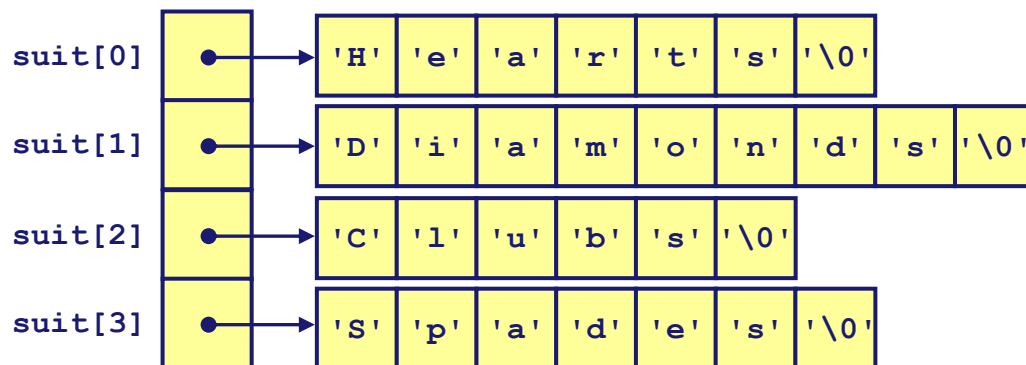
```
}
```



# Масиви вказівників

Масиви можуть зберігати не тільки звичайні значення, але й вказівники. Часто можливість створення масивів вказівників використовується для визначення масивів рядків. Кожен елемент такого масиву є рядком, але в мові **C** рядок фактично є вказівником на перший символ цього рядка. Тобто кожен елемент масиву рядків в дійсності є вказівником.

```
const char *suit[4] = {"Hearts", "Diamonds", "Clubs", "Spades"};
```





# Вказівники на функції

```
#include <stdio.h>
#include <windows.h>
#define SIZE 10
void bubble(int work[], size_t size,
            int (*compare)(int a, int b));
int ascending(int a, int b);
int descending(int a, int b);

int main(void)
{
    int order;           // 1 - за зростанням, 2 - за зменшенням
    size_t counter;     // лічильник

    SetConsoleOutputCP(1251);

    while (1)
    {
        // ініціалізувати неупорядкований масив a
        int a[SIZE] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };
```



# Вказівники на функції

```
puts("=====");  
printf("1 - сортувати за зростанням\n"  
       "2 - сортувати за зменшенням\n"  
       "3 - завершити програму\n");  
puts("=====");  
scanf("%d", &order);  
if(order == 3)  
    exit(0);  
puts("\nЕлементи вхідного масиву");  
// Вивести вхідний масив  
printf("=====\n");  
for(counter = 0; counter < SIZE; counter++)  
    printf("%5d", a[counter]);  
printf("\n=====\n");  
  
// сортувати масив за зростанням  
// передати функцію ascending() в аргументі,  
// щоб забезпечити сортування за зростанням
```



# Вказівники на функції

```
switch (order)
{
    case 1: bubble(a, SIZE, ascending);
            puts("\nЕлементи масиву за зростанням");
            break;
    case 2: bubble(a, SIZE, descending);
            puts("\nЕлементи масиву за зменшенням");
            break;
    default: break;
}
// вивести відсортований масив
printf("=====\n");
for(counter = 0; counter < SIZE; counter++)
    printf("%5d", a[counter]);
printf("\n=====\n");
printf("\n\n");
}
return 0;
}
```



# Вказівники на функції

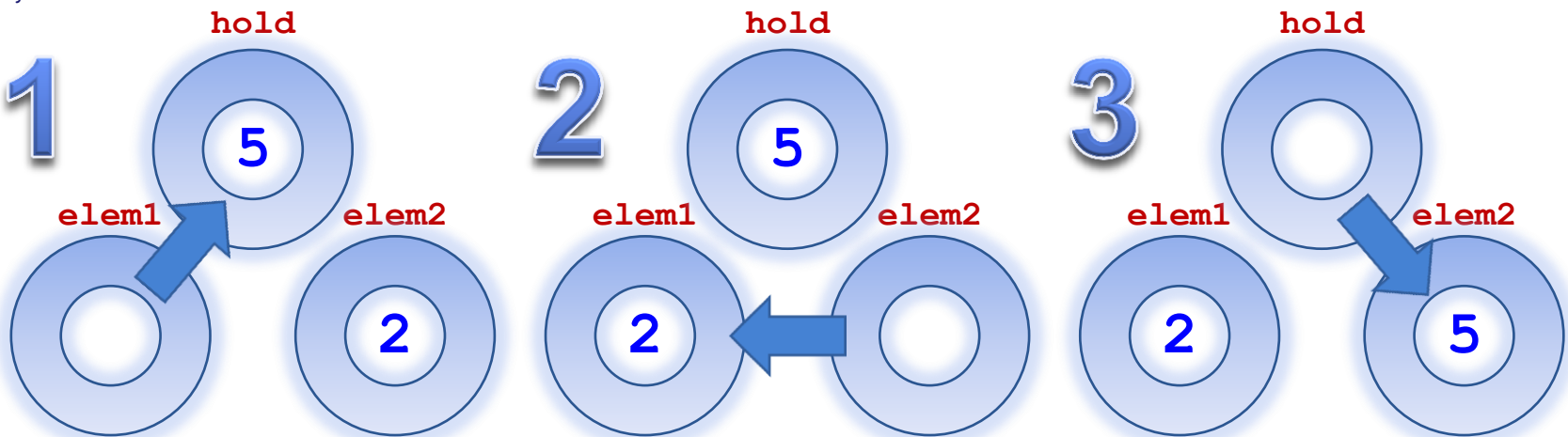
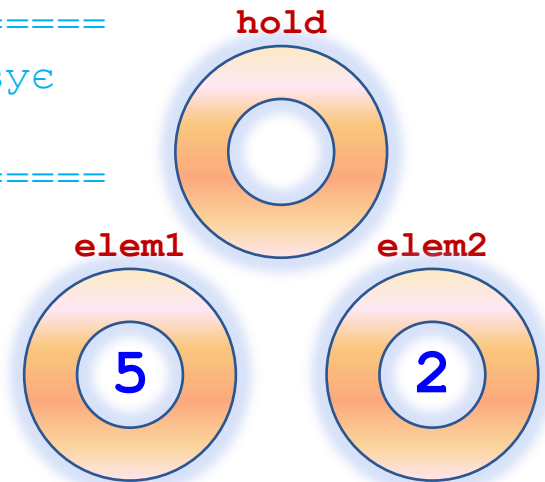
```
//=====
// Універсальна функція сортування параметр. compare - це вказівник
// на функцію порівняння, яка визначає порядок сортування
//=====
void bubble(int work[], size_t size, int (*compare)(int a, int b))
{
    unsigned int pass;           // лічильник проходів
    size_t count;                // лічильник порівнянь
    void swap(int *element1Ptr, int *element2ptr); // прототип
    // цикл, який керує кількістю проходів
    for(pass = 1; pass < size; pass++)
    {
        // цикл, який керує кількістю порівнянь в одному проході
        for(count = 0; count < size - 1; count++)
        {
            // якщо сусідні елементи стоять не в тому порядку,
            // поміняти їх місцями
            if((*compare)(work[count], work[count+1]))
                swap(&work[count], &work[count+1]);
        }
    }
}
```

# Вказівники на функції

```
//=====
// Міняє місцями значення елементів, на які вказує
// element1Ptr і element2Ptr
//=====
```

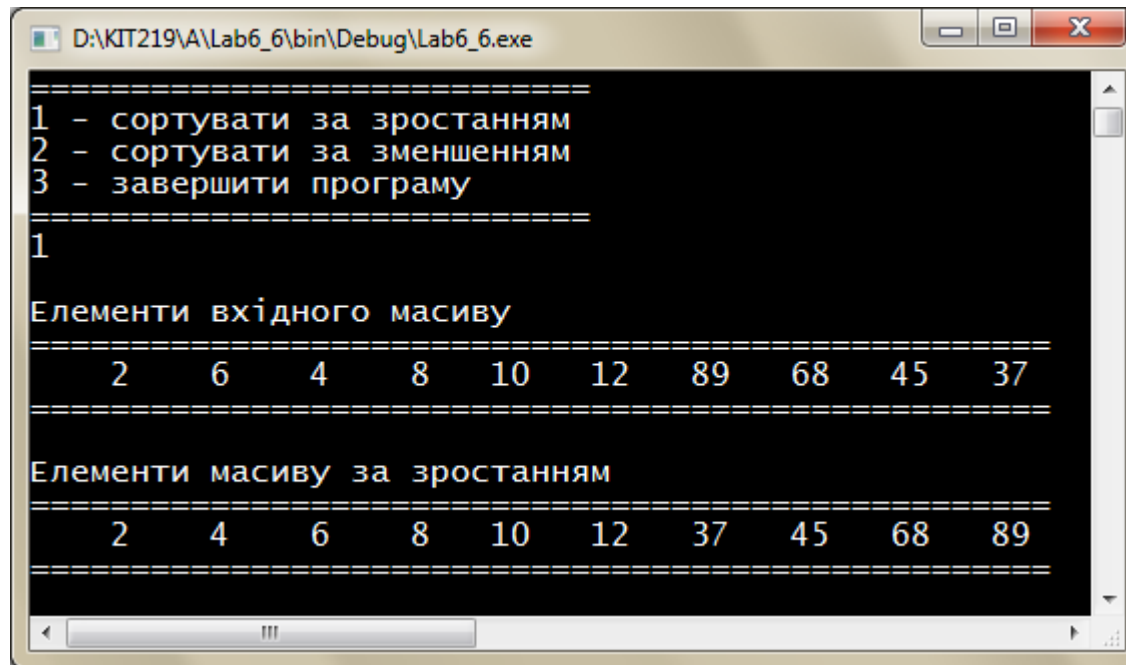
```
void swap(int *element1Ptr, int *element2Ptr)
{
    int hold;    // тимчасова змінна

    hold = *element1Ptr;    // 1
    *element1Ptr = *element2Ptr;    // 2
    *element2Ptr = hold;    // 3
}
```



# Вказівники на функції

```
//=====
// Сортуння за зростанням
//=====
int ascending(int a, int b)
{
    return b < a;    // треба поміняти місцями,
                    // якщо b < a
}
```



```
D:\KIT219\A\Lab6_6\bin\Debug\Lab6_6.exe

=====
1 - сортувати за зростанням
2 - сортувати за зменшенням
3 - завершити програму
=====
1

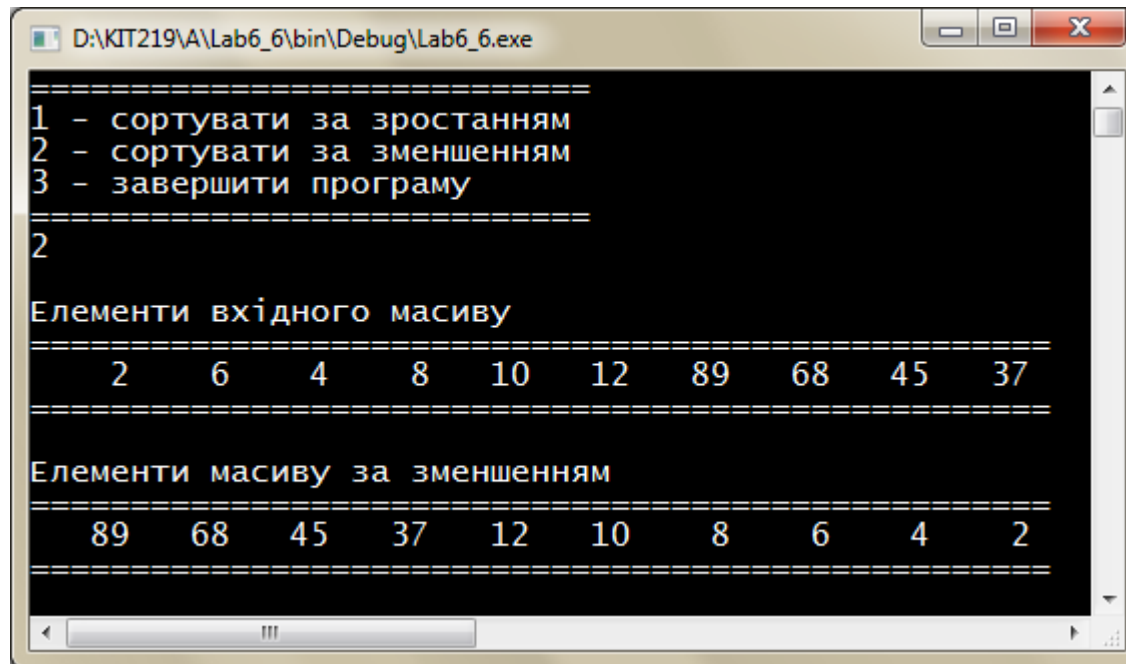
Елементи вхідного масиву
=====
  2   6   4   8  10  12  89  68  45  37
=====

Елементи масиву за зростанням
=====
  2   4   6   8  10  12  37  45  68  89
=====
```



# Вказівники на функції

```
//=====
// Сортуння за зменшенням
//=====
int descending(int a, int b)
{
    return b > a;    // треба поміняти місцями,
                    // якщо b > a
}
```



```
D:\KIT219\A\Lab6_6\bin\Debug\Lab6_6.exe

=====
1 - сортувати за зростанням
2 - сортувати за зменшенням
3 - завершити програму
=====
2

Елементи вхідного масиву
=====
  2   6   4   8  10  12  89  68  45  37
=====

Елементи масиву за зменшенням
=====
 89  68  45  37  12  10   8   6   4   2
=====
```



# Вибір пункту меню

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

// прототипи
void function1(int a);
void function2(int b);
void function3(int c);

int main(void)
{
    // ініціалізувати масив з 3 вказівниками на функції,
    // кожна з яких приймає ціле число і нічого не повертає
    void (*f[3])(int) = { function1, function2, function3 };
    size_t choice; // змінна для вибору користувача

    SetConsoleOutputCP(1251);

    printf("%s", "Введіть число між 0 і 2,3 - для виходу: ");
    scanf("%u", &choice);
```



# Вибір пункту меню

```
// обробити вибір користувача
while(choice >= 0 && choice < 3)
{
    // викликати функцію за вказівником в елементі масиву f
    // з індексом choice і передати їй значення choice
    (*f[choice])(choice);
    printf("%s", "Введіть число між 0 і 2,3 - для виходу: ");
    scanf("%u", &choice);
}
puts("Виконання програми завершено.");
return 0;
}

void function1(int a)
{
    printf("Ви ввели %d, тому була викликана function1\n\n", a);
}
```



# Вибір пункту меню

```
void function2(int b)
{
    printf("Ви ввели %d, тому була викликана function2\n\n", b);
}

void function3(int c)
{
    printf("Ви ввели %d, тому була викликана function3\n\n", c);
}
```

```
D:\KIT219\A\L6_7\bin\Debug\L6_7.exe
Введіть число між 0 і 2, 3 - для виходу: 0
Ви ввели 0, тому була викликана function1

Введіть число між 0 і 2, 3 - для виходу: 1
Ви ввели 1, тому була викликана function2

Введіть число між 0 і 2, 3 - для виходу: 2
Ви ввели 2, тому була викликана function3

Введіть число між 0 і 2, 3 - для виходу: 3
Виконання програми завершено.
```