# ЛАБОРАТОРНА РОБОТА № 4.

**ОДНОВИМІРНІ І ДВОВИМІРНІ МАСИВИ. ВКАЗІВНИКИ В C++**

# Мета роботи

Навчитися використовувати масиви.

# Теоретичні відомості

* + 1. **Масиви**

Масив – це послідовно розміщені у пам’яті елементи однакового типу. Кожен масив має ім’я. Доступ до окремих елементів масиву відбувається по імені масиву та індексу (порядковому номеру, зміщенню відносно першого) елемента. Основні властивості масивів:

* всі елементи масиву мають однаковий тип;
* всі елементи масиву розміщені у пам’яті послідовно один за одним;
* індекс першого елемента рівний нулю;
* ім’я масиву є вказівником-константою, що вказує на адресу в пам’яті першого елемента масиву.

Ознакою масиву при описі є наявність парних квадратних дужок “**[ ]**”. Константа або константний вираз у квадратних дужках задає число елементів. Розмір не може бути динамічним, тому не можна задавати його з допомогою змінних.

**Одновимірний масив**

Загальний вигляд стрічки оголошення одномірного масиву наступний:

# <тип> <ідентифікатор>[<кількість елементів>];

Наприклад:

# int x[10];

**?**

**int x**

**[**

**5**

**]**

**;**

**Stack**

**?**

**5**

**?**

**?**

**x**

**[**

**0**

**(**

**]**

**== \***

**x**

**+**

**0**

**)**

**x**

**[**

**1**

**(**

**== \***

**]**

**x**

**+**

**1**

**)**

**x**

**[**

**2**

**]**

**(**

**== \***

**x**

**+**

**2**

**)**

**x**

**[**

**3**

**== \***

**(**

**]**

**x**

**+**

**3**

**)**

**x**

**[**

**4**

**(**

**== \***

**]**

**x**

**+**

**4**

**)**

**x**

**[**

**2**

**]**

**=**

**5**

**;**

**//**

**еквівалентно**

**//**

**адреса**

**x**

**-**

**>**

**0**

**x**

**600**

**a**

**00**

**//**

**\*(**

**0**

**x**

**600**

**a**

**00**

**+**

**sizeof**

**(**

**int**

**)\***

**2**

**)=**

**5**

**0**

**x**

**600**

**a**

**10**

**0**

**x**

**600**

**a**

**0**

**c**

**0**

**x**

**600**

**a**

**08**

**0**

**x**

**600**

**a**

**04**

**0**

**x**

**600**

**a**

**00**

**...**

**...**

**0**

**x**

**600**

**a**

**1**

**4**

*Ідентифікатор*

**x**

*стає вказівником*

*-*

*константою на деяку дялнку в оперативній*

*пам*

*’*

*яті*

*,*

*визначаючи масив*

*Другому елементу масиву*

**x**

*,*

*що відповідає*

*комірці на яку вказує*

**x**

*плюс*

**sizeof**

**(**

**int**

**)\***

**2**

*присвоюється значення*

**5**

Індекси

Елементи

Назва масиву

*.*

*Рис. 4.1 Розміщення елементів масиву в пам’яті*

Загальний розмір масиву в байтах рівний розміру базового типу помноженого на кількість елементів. Для наведеного прикладу розмірність масиву рівна **sizeof(int)\*10**. Приклад:

**#include <stdio.h> int main()**

**{**

**int x[10];**

**for(int i = 0; i < 10; i++)**

**{**

**x[i] = i;**

**printf("%d ", x[i]);**

**}**

**return 0;**

**}**

Результат:

**0 1 2 3 4 5 6 7 8 9**

**Двовимірні масиви**

С++ дає змогу використовувати багатовимірні масиви. Найпростіший варіант

– двовимірний масив. Для оголошення двовимірного масиву типу **int twodim**

розмірністю **10** на **20** потрібно описати його наступним чином:

**int twodim[10][20];**

На відміну від інших мов програмування розмірності масиву відокремлені одна

від одної квадратними дужками. Останній запис можна розглядати як масив масивів, тобто масив з двадцяти елементів типу **(twodim[10])**, що в свою чергу є масивом з десяти елементів типу **(int)**. Звідси випливає, що елементи багатовимірних масивів в мові С++ у пам’яті зберігаються по рядках.

Для доступу до елемента з індексами **3**, **5** масиву **twodim** служить запис **twodim[3][5]**. Наступний приклад демонструє роботу з двовимірним масивом, а саме, присвоєння кожному елементу цього масиву суму індексів:

**#include <stdio.h> int main()**

**{ int num[2][4];**

**for(int i = 0; i < 2; i++)**

**{ for(int j = 0; j < 4; j++)**

**{ num[i][j] = i + j; printf("%d ", num[i][j]);**

**}**

**printf("\n");**

**}**

**return 0;**

**}**

Результат:

**0 1 2 3**

**1 2 3 4**

Двовимірні масиви можна розглядати як матриці, тобто у вигляді колонок та рядків. У пам’яті всі елементи багатовимірного масиву зберігаються підряд один за одним. Схематично це можна зобразити так (рис. 4.2).

Окремий рядок – елемент типу (**int[3])**

**?**

**Stack**

**?**

**?**

**?**

**5**

**x**

**[**

**0**

**][**

**0**

**] == \*(\*(**

**x**

**+**

**0**

**)+**

**0**

**)**

**0**

**x**

**600**

**a**

**24**

**0**

**x**

**600**

**a**

**20**

**0**

**x**

**600**

**a**

**1**

**c**

**0**

**x**

**600**

**a**

**18**

**0**

**x**

**600**

**a**

**14**

*Елементи розмішуються*

*послідовно по рядках*

**...**

**...**

**0**

**x**

**600**

**a**

**1**

**c**

Індекси

рядків

**?**

**0**

**x**

**600**

**a**

**28**

**x**

**[**

**0**

**][**

**1**

**] == \*(\*(**

**x**

**+**

**0**

**)+**

**1**

**)**

**x**

**[**

**0**

**][**

**2**

**] == \*(\*(**

**x**

**+**

**0**

**)+**

**2**

**)**

**x**

**[**

**1**

**][**

**0**

**] == \*(\*(**

**x**

**+**

**1**

**)+**

**0**

**)**

**x**

**[**

**1**

**][**

**1**

**] == \*(\*(**

**x**

**+**

**1**

**)+**

**1**

**)**

**x**

**[**

**1**

**][**

**2**

**] == \*(\*(**

**x**

**+**

**1**

**)+**

**2**

**)**

Індекси

стовпців

**int m[2][3];**

**/\* адреса m -> 0x600a14 sizeof(int)==4 sizeof(\*m)==sizeof(int[3])==12 sizeof(m)==sizeof(int[2][3])==24\*/**

Елементи типу (**int)**

**m[1][1] = 5;**

**/\* еквівалентно**

**\*(\*(0x600a14+sizeof(int[3])\*1))+ sizeof(int)\*1)) = 5 \*/**

*Рис. 4.2 Розміщення елементів двовимірного масиву в пам’яті*

**Багатовимірні масиви**

Загальний вигляд оголошення багатовимірного масиву наступний:

**<тип> <ідентифікатор>[<розмір1>][<розмір2>]…[<розмірN>];**

Наприклад :

**int three[4][10][5];**

**int four[4][10][5][7];**

Для масиву **four .**  необхідно **sizeof(int)\*4\*10\*5\*7.** байт.

**Ініціалізація масиву**

С++ дає змогу ініціалізувати масиви при їх оголошенні. Загальна форма такої ініціалізації подібна до ініціалізації інших змінних:

**<тип> <ім’я> [<розмір1>]…[<розмір N>] = {список значень};**

Cписок значень – це розділений комами список констант. Наприклад:

**int i[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};**

Це означає, що **i[0] = 1**, **i[1] = 2**, ... , **i[9] = 10**.

Багатовимірні масиви ініціалізуються так само, як і одновимірні:

**int s[4][2] = { 1, 1,**

**2, 4,**

**3, 9,**

**4, 16};**

Для такої попередньої ініціалізації необхідно знати точну кількість елементів. Щоб не задавати фіксовану розмірність масиву, ініціалізацію можна провести наступним чином:

# int i[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

а для багатомірного масиву:

**int s[][2] = {1, 1,**

**2, 4,**

**3, 9};**

Якщо кількість елементів для ініціалізації є завеликою, компілятор видасть повідомлення про помилку, якщо замалою – згідно стандарту, непроініціалізовані елементи заповняться нулями. Можлива ініціалізація окремих елементів масиву, наприклад:

# int k[5] = {[2]=1}; //{0, 0, 1, 0, 0};

Стандартом не визначено, що міститимуть елементи непроініціалізованих масивів, якщо тільки вони не оголошені, як статичні – у такому випадку вони будуть містити нулі.

# ВКАЗІВНИКИ

# Оператор адреси &

При виконанні ініціалізації змінної, їй автоматично присвоюється вільна адреса в пам’яті, і, будь-яке значення, яке ми присвоюємо змінній, зберігається за цією адресою в пам’яті. Наприклад:

|  |  |
| --- | --- |
| **1** | **int b;** |

При виконанні цього стейтменту процесором, виділяється частина оперативної пам’яті. В якості прикладу припустимо, що змінній b присвоюється комірка пам’яті під номером 150. Всякий раз, коли програма зустрічає змінну b в виразі чи в стейтменті, вона розуміє, що для того, щоб отримати значення — їй потрібно зазирнути в комірку пам’яті під номером 150.

Хороша новина — нам не потрібно турбуватися про те, які конкретно адреси в пам’яті виділені для певних змінних. Ми просто посилаємося на змінну через присвоєний їй ідентифікатор, а компілятор конвертує цей ідентифікатор у відповідну адресу в пам’яті. Однак цей підхід має деякі обмеження, які ми обговоримо далі.

**Оператор адреси**& дозволяє дізнатися, яку адресу в пам’яті присвоєно певній змінній. Все просто:

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8**  **9**  **10** | **#include <iostream>**    **int main()**  **{**  **int a = 7;**  **std::cout << a << '\n'; // виводимо значення змінної a**  **std::cout << &a << '\n';// виводимо адресу в пам'яті**  **//змінної a**  **return 0;**  **}** |

Результат на комп’ютері:

**7  
0046FCF0**

Примітка: Хоча оператор адреси виглядає так само, як [оператор побітового І](https://acode.com.ua/urok-48-pobitovi-operatory/#toc-4), відрізнити їх можна по тому, що оператор адреси є [унарним оператором](https://acode.com.ua/urok-20-operatory/" \l "toc-1" \t "_blank), а оператор побітового І — бінарним оператором.

# Оператор розіменування \*

**Оператор розіменування \*** дозволяє отримати значення по вказаній адресі:

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8**  **9**  **10**  **11** | **#include <iostream>**    **int main()**  **{**  **int a = 7;**  **std::cout << a << '\n'; // виводимо значення змінної a**  **std::cout << &a << '\n'; // виводимо адресу змінної a**  **std::cout << \*&a << '\n';// виводимо значення комірки**  **// в пам'яті змінної a**  **return 0;**  **}** |

Результат на моєму комп’ютері:

**7  
0046FCF0  
7**

Примітка: Хоча оператор розіменування виглядає так само, як і оператор множення, відрізнити їх можна по тому, що оператор розіменування — унарний, а оператор множення — бінарний.

# Вказівники

Тепер, коли ми вже знаємо про операторів адреси і розіменування, ми можемо поговорити про вказівники.

**Вказівник**(або ***“покажчик”***) — це змінна, значенням якої є адреса комірки в пам’яті. Вказівники оголошуються так само, як і звичайні змінні, тільки із зірочкою між типом даних і ідентифікатором:

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7** | **int \*iPtr; // вказівник на значення типу int**  **double \*dPtr; // вказівник на значення типу double**    **int\* iPtr3;//коректний синтаксис (дозволено, але не**  **//бажано)**  **int \* iPtr4; // коректний синтаксис (не робіть так)**  **int \*iPtr5, \*iPtr6; // оголошуємо два вказівники для змінних**  **// типу int** |

Синтаксично мова C++ приймає оголошення вказівника, коли зірочка знаходиться поруч з типом даних, з ідентифікатором або навіть посередині. Зверніть увагу, ця зірочка НЕ є оператором розіменування. Це всього лише частина синтаксису оголошення вказівника.

Однак, при оголошенні кількох вказівників, зірочка повинна знаходитися біля кожного ідентифікатора. Це легко забути, якщо ви звикли вказувати зірочку біля типу даних, а не біля імені змінної. Наприклад:

|  |  |
| --- | --- |
| **1**  **2** | **int\* iPtr3, iPtr4; // iPtr3 - це вказівник на значення типу**  **//int, а iPtr4 - це звичайна змінна типу int!** |

З цієї причини, при **оголошенні** вказівника, рекомендується вказувати зірочку біля імені змінної. Як і звичайні змінні, вказівники не ініціалізуються при оголошенні. Вмістом неініціалізованого вказівника є звичайне сміття.

# Присвоювання значень вказівнику

Оскільки вказівники містять тільки адреси, то при присвоюванні значення вказівнику — це значення повинно бути адресою. Для отримання адреси змінної використовується оператор адреси:

|  |  |
| --- | --- |
| **1**  **2** | **int value = 5;**  **int \*ptr = &value;//ініціалізуємо ptr адресою змінної** |

Ось чому вказівники мають таку назву: ptr містить адресу значення змінної value, і, можна сказати, ptr *вказує*на це значення.

Ще дуже часто можна побачити наступне:

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8**  **9**  **10**  **11**  **12**  **13**  **14** | **#include <iostream>**    **int main()**  **{**  **int value = 5;**  **int \*ptr = &value; // ініціалізуємо ptr адресою**  **//значення змінної**    **std::cout << &value << '\n';// виводимо адресу**  **// значення змінної value**  **std::cout << ptr << '\n'; // виводимо адресу, яку**  **// містить ptr**  **return 0;**  **}** |

Результат на комп’ютері:

**003AFCD4  
003AFCD4**

Тип вказівника повинен відповідати типу змінної, на яку він вказує:

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8**  **9** | **int iValue = 7;**  **double dValue = 9.0;**    **int \*iPtr = &iValue; // ок**  **double \*dPtr = &dValue; // ок**  **iPtr = &dValue; // неправильно: вказівник типу int не може**  **// вказувати на адресу змінної типу double**  **dPtr = &iValue; // неправильно: вказівник типу double не**  **// може вказувати на адресу змінної типу int** |

Наступне не є допустимим:

|  |  |
| --- | --- |
| **1** | **int \*ptr = 7;** |

Це пов’язано з тим, що вказівники можуть містити тільки адреси, а цілочисельний [**літерал**](https://acode.com.ua/urok-39-literaly-i-magichni-chysla/) 7 не має адреси в пам’яті. Якщо ви все ж зробите це, то компілятор повідомить вам, що він не може перетворити цілочисельне значення в **цілочисельний** вказівник.

Мова C++ також не дозволить вам напряму присвоювати адреси в пам’яті вказівнику:

|  |  |
| --- | --- |
| **1** | **double \*dPtr = 0x0012FF7C; // не ок: розглядається як**  **//присвоювання цілочисельного літералу** |

# Оператор адреси повертає вказівник

Варто зазначити, що оператор адреси & не повертає адресу свого операнда в якості літералу. Замість цього він повертає вказівник, що містить адресу операнда, тип якого отримано з аргументу (наприклад, адреса змінної типу int передається як адреса вказівника на значення типу int):

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8**  **9**  **10** | **#include <iostream>**  **#include <typeinfo>**    **int main()**  **{**  **int x(4);**  **std::cout << typeid(&x).name();**    **return 0;**  **}** |

Результат виконання програми:

**int \***

# Розіменування вказівників

Як тільки у нас є вказівник, який вказує на що-небудь, ми можемо його розіменувати, щоб отримати значення, на яке він вказує. Розіменований вказівник — це вміст комірки в пам’яті, на яку він вказує:

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8**  **9**  **10**  **11**  **12**  **13**  **14**  **15** | **#include <iostream>**    **int main()**  **{**  **int value = 5;**  **std::cout << &value << std::endl; // виводимо адресу value**  **std::cout << value << std::endl; // виводимо вміст value**    **int \*ptr = &value; // ptr вказує на адресу value**  **std::cout << ptr << std::endl; // виводимо адресу, яка**  **//зберігається в ptr (тобто &value)**  **std::cout << \*ptr << std::endl; // розіменовуємо ptr**  **//отримуємо значення, на яке вказує ptr)**  **return 0;**  **}** |

Результат:

**0034FD90  
5  
0034FD90  
5**

Ось чому вказівники повинні мати тип даних. Без типу вказівник не знав би, як інтерпретувати вміст, на який він вказує (при розіменуванні). Також, тому і повинні збігатися тип вказівника з типом змінної. Якщо вони не збігатимуться, то вказівник при розіменуванні може неправильно інтерпретувати біти (наприклад, замість типу double використати тип int).

Одному вказівнику можна присвоювати різні значення:

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8**  **9**  **10** | **int value1 = 5;**  **int value2 = 7;**    **int \*ptr;**    **ptr = &value1; // ptr вказує на &value1**  **std::cout << \*ptr; // виведеться 5**    **ptr = &value2; // ptr тепер вказує на value2**  **std::cout << \*ptr; // виведеться 7** |

Коли адреса значення змінної присвоєна вказівнику, то виконується наступне:

**ptr**— це те ж саме, що і **&value**;

**\*ptr** обробляється так само, як і **value**.

Оскільки **\*ptr** обробляється так само, як і **value**, то ми можемо присвоювати йому значення так, наче це звичайна змінна. Наприклад:

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6** | **int value = 5;**  **int \*ptr = &value; // ptr вказує на value**    **\*ptr = 7; // \*ptr - це те ж саме, що і value, якому ми**  **//присвоїли значення 7**  **std::cout << value; // виведеться 7** |

# Розіменування некоректних вказівників

Вказівники в мові С++ по своїй суті є небезпечними, а їх неправильне використання — один з найкращих способів отримати збій в програмі.

При розіменуванні вказівника, програма намагається перейти в комірку в пам’яті, яка зберігається в вказівнику і “витягнути” вміст цієї комірки. З міркувань безпеки сучасні операційні системи (скор. “ОС”) запускають програми в пісочниці для запобігання їх неправильної взаємодії з іншими програмами і для захисту стабільності самої операційної системи. Якщо програма спробує отримати доступ до комірки в пам’яті, не виділеної для неї операційною системою, то ОС відразу завершить виконання цієї програми.

Наступна програма добре ілюструє вищесказане. При запуску ви отримаєте збій (спробуйте, нічого страшного з вашим комп’ютером не відбудеться):

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8**  **9**  **10**  **11**  **12**  **13**  **14**  **15**  **16**  **17** | **#include <iostream>**    **void foo(int \*&p)**  **{**  **}**    **int main()**  **{**  **int \*p; // створюємо неініціалізований вказівник (вмістом якого є сміття)**  **foo(p); // вводимо компілятор в оману, ніби ми**  **//збираємося присвоїти**  **// вказівнику коректне значення**    **std::cout << \*p; // розіменовуємо вказівник зі сміттям**  **return 0;**  **}** |

# Розмір вказівників

Розмір вказівника залежить від архітектури, на якій скомпільовано виконуваний файл: 32-бітний виконуваний файл використовує 32-бітні адреси в пам’яті. Відповідно, вказівник на 32-бітному пристрої займає 32 біти (4 байти). З 64-бітним виконуваним файлом вказівник займатиме 64 біти (8 байтів). І це незалежно від того, на що вказує вказівник:

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8**  **9**  **10**  **11**  **12**  **13** | **char \*chPtr; // тип char займає 1 байт**  **int \*iPtr; // тип int займає 4 байти**    **struct Something**  **{**  **int nX, nY, nZ;**  **};**    **Something \*somethingPtr;**    **std::cout << sizeof(chPtr) << '\n'; // виведеться 4**  **std::cout << sizeof(iPtr) << '\n'; // виведеться 4**  **std::cout << sizeof(somethingPtr) << '\n'; // виведеться 4** |

Як ви можете бачити, розмір вказівника завжди один і той же. Це пов’язано з тим, що вказівник — це всього лише адреса в пам’яті, а кількість біт, необхідна для доступу до адреси в пам’яті на певному пристрої, — завжди постійна.

# Чим корисні вказівники?

Зараз ви можете подумати, що вказівники є непрактичними і взагалі непотрібними. Навіщо використовувати вказівник, якщо ми можемо використати вихідну змінну?

**Вказівники корисні в наступних випадках**:

***Випадок №1:* Масиви реалізовані за допомогою вказівників**. Вказівники можуть використовуватися для ітерації по масиву.

***Випадок №2:* Вони є єдиним способом динамічного виділення пам’яті в С++**. Це, безумовно, найбільш поширений варіант використання вказівників.

***Випадок №3:* Вони можуть використовуватися для передачі великої кількості даних в функцію без копіювання цих даних.**

***Випадок №4:* Вони можуть використовуватися для передачі однієї функції в якості параметру іншій функції.**

***Випадок №5:* Вони використовуються для досягнення поліморфізму при роботі зі спадкуванням.**

***Випадок №6:* Вони можуть використовуватися для представлення однієї структури/класу в іншій структурі/класі, формуючи, таким чином, “ланцюжки”.**

Вказівники застосовуються в багатьох випадках. Тепер, коли ми розібралися зі вказівниками на базовому рівні, ми можемо почати заглиблюватися в окремі випадки, в яких вони корисні.

# Висновки

Вказівники — це змінні, які містять адреси в пам’яті. Їх можна розіменувати за допомогою оператора розіменування \* для доступу до значень, які містяться за адресою в пам’яті. Розіменування вказівника, значенням якого є сміття, призведе до збою в вашій програмі.

Порада: При оголошенні вказівника вказуйте зірочку біля імені змінної

* 1. **Контрольні запитання**

1. Що таке масив?
2. Як обчислити розмір масиву?
3. В якій послідовності розміщуються в пам’яті елементи двовимірного масиву?
4. Що таке потоковий ввід-вивід?
5. Які існують стандартні потоки?
6. Для чого використовують вказівники?
7. Що означає розіменування вказівника?
8. Як присвоїти значення вказівнику?
9. Що означає некоректне розіменування?
10. У яких випадках вказівники корисні?

# Лабораторне завдання

1. Навчитися використовувати масиви та файли при розв’язанні задач.
2. Одержати індивідуальне завдання.
3. Побудувати блок-схеми алгоритмів відповідно до завдання.
4. Скласти програми на алгоритмічній мові C++ згідно завдання.
5. Відлагодити програми, виконати обчислення, проаналізувати отримані результати.

# Зміст звіту

1. Титульний аркуш.
2. Мета роботи.
3. Індивідуальне завдання.
4. Блок-схема алгоритмів у відповідності до завдання.
5. Тексти програм у відповідності до завдання.
6. Результати обчислень.
7. Аналіз результатів, висновки.

# Індивідуальні завдання

Завдання 1

Дано натуральне число *N* (задати довільно, як константу препроцесора) і одновимірний масив *A*0, *A*1, …, *AN*-1 цілих чисел (згенерувати додатні та від’ємні елементи випадковим чином, за допомогою функції бібліотеки **<stdlib.h> rand()**). Виконати наступні дії:

1. Визначити найбільше з непарних і кількість парних чисел.
2. Одержати масив, що відрізняється від оригінального тим, що всі непарні за порядком елементи подвоєні, а парні отримані додаванням власного значення з початковим значенням наступного елемента.
3. Визначити три максимальних і два мінімальних значення цього масиву.
4. Визначити найменше додатне.
5. Визначити число пар двох однакових додатних чисел, наприклад, чотири числа утворюють дві пари.
6. Визначити число пар двох однакових чисел різного знаку, наприклад, чотири числа утворюють дві пари.
7. Для кожного елемента визначити число його входжень у даний масив.
8. Одержати всі елементи, що входять у даний масив тільки по одному разі.
9. Одержати всі елементи, що входять у даний масив більше одного разу.
10. Визначити найбільше й найменше значення.
11. Замінити елементи, розташовані в парних позиціях першої половини масиву, подвоєними значеннями елементів, розташованих у непарних позиціях другої половини масиву.
12. Знайти найменший елемент у найбільш довгій безперервній послідовності додатних значень.
13. Виконати сортування елементів по зростанню.
14. Замінити всі нульові елементи масиву найменшим по модулю, але відмінним від нуля елементом.
15. Поміняти місцями елементи з мінімальним і максимальним значеннями.
16. Замінити від’ємні елементи масиву на середнє арифметичне значення чисел цього масиву.

Завдання 2

Записати матрицю цілих чисел *А*(*N*,*N*) (згенерувати випадковим чином, *N* задати довільно, як константу препроцесора).

1. Переставити стовпці матриці у зворотному порядку.
2. Поміняти місцями елементи головної і бічної діагоналей.
3. Визначити найменші у своєму рядку елементи матриці та їх індекси.
4. Знайти номер рядка, всі елементи якого є парними числами (якщо не відається згенерувати випадково відповідну матрицю, то задати її явно).
5. Транспонувати матрицю*.*
6. Відсортувати кожен рядок по зростанню*.*
7. Знайти суму елементів розміщених над головною діагоналлю*.*
8. Обчислити суму від’ємних елементів кожного рядка*.*
9. Визначити чи є ця матриця симетричною відносно головної діагоналі.
10. Обчислити скалярний добуток рядка в якому міститься найбільший елемент матриці на стовпець з найменшим елементом*.*
11. Пронормувати кожен рядок цієї матриці поділивши на найбільший по модулю елемент.
12. Знайти суму додатних елементів кожного рядка.
13. Обчислити середнє арифметичне додатних елементів кожного стовпця.
14. Знайти суму додатних елементів, які розміщені під головною діагоналлю і суму від’ємних елементів, які розміщені над головною діагоналлю.
15. Знайти максимальний елемент матриці і обнулити рядок та стовпець, в якому він знаходиться*.*
16. Обчислити середнє арифметичне всіх додатних елементів, що знаходяться під головною діагоналлю.