

Масиви. Застосування масивів.

Масив це структура даних, представлена у вигляді групи комірок одного типу, об'єднаних під одним єдиним ім'ям. Масиви використовуються для обробки великої кількості однотипних даних. Ім'я масиву є покажчиком, що таке покажчики розглянемо трохи пізніше. Окрема комірка даних масиву називається елементом масиву. Елементами масиву можуть бути дані будь-якого типу. Масиви можуть мати як одне, так і більш одного вимірювань. Залежно від кількості вимірювань масиви діляться на одномірні масиви, двовимірні масиви, тривимірні масиви і так далі до n -мірного масиву. Найчастіше в програмуванні використовуються одномірні і двовимірні масиви, тому ми розглянемо тільки ці масиви.

Одномірні масиви в C ++(вектори)

Одновимірний масив – масив, з одним параметром, що характеризує кількість елементів одновимірного масиву. Фактично одновимірний масив – це масив, у якого може бути тільки один рядок, і n -та кількість стовпців. Стовпчики в одновимірному масиві – це елементи масиву. Ще одномірні масиви називають векторами. На малюнку 1 показана структура цілочисельного одновимірного масиву a . Розмір цього масиву – 16 комірок.

5	-12	-12	9	10	0	-9	-12	-1	23	65	64	11	43	39	-15
$a[0]$	$a[1]$	$a[2]$	$a[3]$	$a[4]$	$a[5]$	$a[6]$	$a[7]$	$a[8]$	$a[9]$	$a[10]$	$a[11]$	$a[12]$	$a[13]$	$a[14]$	$a[15]$

Рисунок 1

Зауважте, що максимальний індекс одновимірного масиву a дорівнює 15, але розмір масиву 16 елементів, тому що нумерація елементів масиву завжди починається з 0. Індекс елементу – це ціле невід'ємне число, за яким можна звертатися до кожної клітинки масиву і виконувати будь-які дії з нею (елементом).

Синтаксис опису масиву:

тип ідентифікатор[кількість елементів];

Так масив цілих чисел a з 16 елементів можна описати:

int $a[16]$; // опис одного масиву

int $a[16]$, $c[16]$; // опис двох масивів однакової розмірності.

Завжди відразу після імені масиву йдуть квадратні дужки, в яких задається розмір одновимірного масиву, цим масив і відрізняється від всіх інших змінних.

Масиви можуть бути ініційовані при оголошенні:

int $a[16]$ = { 5, -12, -12, 9, 10, 0, -9, -12, -1, 23, 65, 64, 11, 43, 39, -15 };

Ініціалізація одновимірного масиву виконується в фігурних дужках після знака =, кожен елемент масиву відокремлюється від попереднього комою. В даному випадку компілятор сам визначить розмір одновимірного масиву. Розмір масиву можна не вказувати тільки при його ініціалізації, при

звичайному оголошенні масиву обов'язково потрібно вказувати розмір масиву.

Приклад. Розробимо просту програму на обробку одновимірного масиву.

```
#include <iostream>
using namespace std;

int main()
{
    cout << "Обробка масиву" << endl;

    // Оголошення та ініціалізація масиву "array1"
    int array1[16] = { 5, -12, -12, 9, 10, 0, -9, -12, -1, 23, 65, 64, 11, 43, 39, -15 };

    // Виведення заголовків стовпців
    cout << "Індекс" << "\t\t" << "Елемент масиву" << endl;

    // Початок циклу "for" для обходу масиву
    for (int counter = 0; counter < 16; counter++)
    {
        // Виведення на екран індексу елементу масиву та його значення
        cout << "array1[" << counter << "]" << "\t\t" << array1[counter] << endl;
    }

    system("pause");
    return 0;
}
```

Результат виконання програми:

```
array1[0]          5
array1[1]         -12
array1[2]         -12
array1[3]          9
array1[4]         10
array1[5]          0
array1[6]         -9
array1[7]        -12
array1[8]         -1
array1[9]         23
array1[10]        65
array1[11]        64
array1[12]        11
array1[13]        43
array1[14]        39
array1[15]       -15
Для продолжения нажмите любую клавишу . . .
```

Двовимірні масиви в C ++ (матриці)

До цього моменту ми розглядали одновимірні масиви, якими не завжди можна обмежитися. Припустимо, необхідно обробити деякі дані з таблиці. У таблиці є дві характеристики: кількість рядків і кількість стовпців. Також і в

двовимірному масиві, крім кількості елементів масиву, є такі характеристики як, кількість рядків і кількість стовпців двовимірного масиву. Тобто, візуально, двовимірний масив - це звичайна таблиця, з рядками і стовпцями. Фактично двовимірний масив - це одновимірний масив одновимірних масивів. Структура двовимірного масиву, з ім'ям *a*, розміром *m* на *n* показана нижче

<i>a</i> [0][0]	<i>a</i> [0][1]	<i>a</i> [0][2]	<i>a</i> [0][3]	. . .	<i>a</i> [0][<i>n</i>]
<i>a</i> [1][0]	<i>a</i> [1][1]	<i>a</i> [1][2]	<i>a</i> [1][3]	. . .	<i>a</i> [1][<i>n</i>]
<i>a</i> [2][0]	<i>a</i> [2][1]	<i>a</i> [2][2]	<i>a</i> [2][3]	. . .	<i>a</i> [2][<i>n</i>]
. . .	. . .	. . .	. . .	. . .	. . .
<i>a</i> [<i>m</i>][0]	<i>a</i> [<i>m</i>][1]	<i>a</i> [<i>m</i>][2]	<i>a</i> [<i>m</i>][3]	. . .	<i>a</i> [<i>m</i>][<i>n</i>]

Синтаксис оголошення двовимірного масиву:

тип ідентифікатор[кількість рядків] [кількість стовпчиків];

В оголошенні двовимірного масиву, також як і в оголошенні одновимірного масиву, в першу чергу, потрібно вказати:

- тип даних;
- ім'я масиву.

Після чого, в перших квадратних дужках вказується кількість рядків двовимірного масиву, у других квадратних дужках – кількість стовпців двовимірного масиву. Двовимірний масив візуально відрізняється від одновимірного другою парою квадратних дужок. Розглянемо приклад оголошення двовимірного масиву. Припустимо нам необхідно оголосити двовимірний масив, з кількістю елементів, рівним 15. У такому випадку двовимірний масив може мати три рядки і п'ять стовпців або п'ять рядків і три стовпці:

int a[5][3];

Ініціалізація двовимірного масиву:

int a[5][3] = { {4, 7, 8}, {9, 66, -1}, {5, -5, 0}, {3, -3, 30}, {1, 1, 1} };

В даному масиві 5 рядків, 3 стовпці. після знака присвоїти ставляться фігурні дужки, всередині яких ставиться стільки пар фігурних дужок, скільки повинно бути рядків в двовимірному масиві, причому ці дужки розділяються комами. У кожній парі фігурних дужок записуються через кому елементи двовимірного масиву. У всіх фігурних дужках кількість елементів має збігатися. Так як в масиві п'ять рядків, то і внутрішніх дужок теж п'ять. У внутрішніх дужках записані по три елементи, так як кількість стовпців – три. Графічно наш масив буде виглядати, як двовимірна таблиця

		Напря́м зміни друго́го індексу		
		→		
		0	1	2
Напря́м зміни першого індексу	0	mas[0][0]	mas[0][1]	mas[0][2]
	1	mas[1][0]	mas[1][1]	mas[1][2]
	2	mas[2][0]	mas[2][1]	mas[2][2]
	3	mas[3][0]	mas[3][1]	mas[3][2]

Для обробки двовимірного масиву застосовують подвійний цикл, наприклад, зовнішній цикл управляє рядками, а внутрішній – стовпчиками.

Приклад.

Розробимо нескладну програму, на обробку двовимірного масиву, яка називається «Лабіринт». Лабіринт має бути побудований на основі двовимірного масиву.

```
#include "stdafx.h"
#include <iostream>
using namespace std;

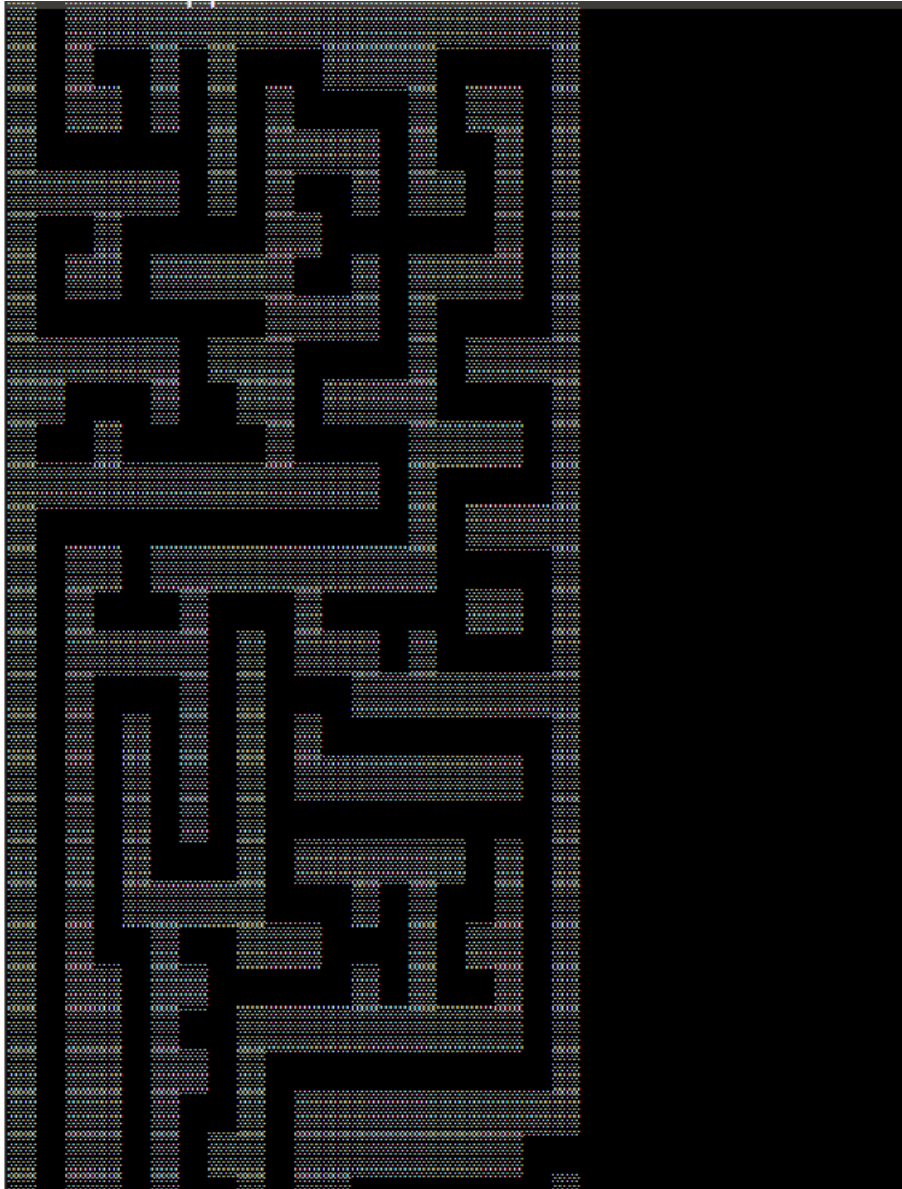
int main(int argc, char* argv[])
{
    // 1-умовно "стенки лабіринта"
    // 2-"правильный путь, выход из лабіринта"
    // 0-"ложный путь"
    int mas[33][20] = { {1,2,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1}, // инициализация
        {1,2,1,0,0,1,0,1,2,2,2,1,1,1,1,0,0,0,0,1},
        {1,2,1,1,0,1,0,1,2,1,2,2,2,1,0,1,1,0,1,},
        {1,2,2,2,2,2,2,1,2,1,1,1,1,2,1,0,0,1,0,1},
        {1,1,1,1,1,1,2,1,2,1,0,0,1,2,1,1,0,1,0,1},
        {1,0,0,1,0,0,2,2,2,1,1,0,0,2,0,0,0,1,0,1},
        {1,0,1,1,0,1,1,1,1,1,0,0,1,2,1,1,1,1,0,1},
        {1,0,0,0,0,0,0,0,0,1,1,1,1,2,1,0,0,0,0,1},
        {1,1,1,1,1,1,0,1,1,1,2,2,2,1,0,1,1,1,1,},
        {1,1,0,0,0,1,0,0,1,1,2,1,1,1,1,0,0,0,0,1},
        {1,0,0,1,0,0,0,0,0,1,2,2,2,2,1,1,1,1,0,1},
        {1,1,1,1,1,1,1,1,1,1,1,1,1,2,1,0,0,0,0,1},
        {1,2,2,2,2,2,2,2,2,2,2,2,2,1,0,1,1,1,1},
        {1,2,1,1,0,1,1,1,1,1,1,1,1,1,1,0,0,0,0,1},
        {1,2,1,0,0,0,1,2,2,2,1,0,0,0,0,0,1,1,0,1},
        {1,2,1,1,1,1,1,2,1,2,1,1,1,0,1,0,0,0,0,1},
        {1,2,1,2,2,2,1,2,1,2,2,2,1,1,1,1,1,1,1,},
        {1,2,1,2,1,2,1,2,1,0,1,2,2,2,2,2,2,2,2,1},
        {1,2,1,2,1,2,1,2,1,0,1,1,1,1,1,1,1,1,2,1},
        {1,2,1,2,1,2,1,2,1,0,0,0,0,0,0,0,0,0,2,1},
        {1,2,1,2,1,2,2,2,1,0,1,1,1,1,1,1,0,1,2,1},
        {1,2,1,2,1,1,1,1,1,0,0,0,1,0,1,0,0,1,2,1},
    }
```

```

        {1,2,1,2,2,1,0,0,1,1,1,0,0,0,1,0,1,1,2,1},
        {1,2,1,1,2,1,1,0,0,0,0,0,1,0,1,0,0,1,2,1},
        {1,2,1,1,2,1,0,0,1,1,1,1,1,1,1,1,1,1,2,1},
        {1,2,1,1,2,1,1,0,1,2,2,2,2,2,2,2,2,2,2,1},
        {1,2,1,1,2,1,0,0,1,2,1,1,1,1,1,1,1,1,1,1},
        {1,2,1,1,2,1,0,1,1,2,1,1,1,1,1,1,1,1,2,2},
        {1,2,1,1,2,1,0,0,1,2,1,1,2,2,2,2,2,2,2,1},
        {1,2,1,1,2,1,0,1,1,2,1,1,2,1,1,1,1,1,1,1},
        {1,2,1,1,2,1,0,0,1,2,1,1,2,1,0,0,0,1,0,1},
        {1,2,2,2,2,1,0,1,1,2,2,2,2,0,0,1,0,0,0,1},
        {1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1} };
// два цикла - осуществляют обращение к каждому элементу массива
for (int i = 0; i < 33; i++) //переключение по строкам
{
    for (int j = 0; j < 20; j++)// переключение по столбцам
        if (mas[i][j] == 1)
        { // вывести два раза символ (номер которого 176 в таблице аски)
            cout << static_cast<char>(176);
            cout << static_cast<char>(176);
        } else
            cout << " "; // вывести два пробела
    cout << endl;
}
system("pause");
return 0;}

```

Перший цикл **for** переключає рядки двовимірного масиву. У тілі другого циклу **for** всередині оператора умовного вибору **if** виконується унарна операція перетворення типу даних – `static_cast <> ()`, яка друкує символ таблиці ASCII, під номером 176. операція перетворення типів даних дублюється для збільшення ширини лабіринту. Результат роботи програми



Застосування масивів

Сортування

Сортування масиву – це процес розподілу всіх елементів масиву в певному порядку. Дуже часто це буває корисним.

Сортування зазвичай виконується шляхом повторного порівняння пар елементів масиву і їх заміни, якщо вони відповідають певним критеріям. Порядок, в якому ці елементи порівнюються, залежить від того, який алгоритм сортування використовується. Критерії складаються з того, як буде сортуватися масив (наприклад, в порядку зростання або зменшення).

Щоб поміняти два елементи місцями, ми можемо використовувати функцію `std::swap()` зі стандартної бібліотеки C++, яка визначена в заголовки `<algorithm>`. У C++, з метою ефективності, `std::swap()` перенесено в заголовки `<utility>`.

```

1 #include <algorithm> // для std::swap. в C++11 используйте заголовок <utility>
2 #include <iostream>
3
4 int main()
5 {
6     int a = 3;
7     int b = 5;
8     std::cout << "Before swap: a = " << a << ", b = " << b << '\n';
9     std::swap(a, b); // меняем местами значения переменных a и b
10    std::cout << "After swap: a = " << a << ", b = " << b << '\n';
11 }

```

Результат:

Before swap: a = 3, b = 5

After swap: a = 5, b = 3

Сортування методом вибору

Існує безліч способів сортування масивів. Сортування методом вибору, мабуть, найпростіше для розуміння, хоч і одне з найбільш повільних.

Для сортування масиву від найменшого до найбільшого елемента методом вибору виконуються наступні кроки:

1. Починаючи з елемента під індексом 0, шукаємо в масиві найменше значення.

2. Знайдене значення міняємо місцями з нульовим елементом.

3. Повторюємо кроки 1 і 2 вже для наступного індексу в масиві.

Іншими словами, ми шукаємо найменший елемент в масиві і переміщаємо його на перше місце. Потім шукаємо другий найменший елемент і переміщаємо його вже на друге місце. Цей процес триває до тих пір, поки не закінчатся елементи в масиві.

Ось приклад роботи цього алгоритму в масиві з 5-ма елементами:

{ 30, 50, 20, 10, 40 }

Спочатку шукаємо найменший елемент, починаючи з індексу 0:

{ 30, 50, 20, **10**, 40 }

Потім міняємо місцями найменший елемент з елементом під індексом 0:

{ **10**, 50, 20, **30**, 40 }

Тепер, коли перший елемент відсортований, ми його ігноруємо.

Шукаємо наступний найменший елемент, але вже починаючи з індексу 1:

{ 10, 50, **20**, 30, 40 }

І міняємо його місцями з елементом під індексом 1:

{ 10, **20**, **50**, 30, 40 }

Тепер ми можемо ігнорувати перші два елементи. Шукаємо наступний найменший елемент, починаючи з індексу 2:

{ 10, 20, 50, **30**, 40 }

І міняємо його місцями з елементом під індексом 2:

{ 10, 20, **30**, **50**, 40 }

Шукаємо наступний найменший елемент, починаючи з індексу 3:

{ 10, 20, 30, 50, **40** }

І міняємо його місцями з елементом під індексом 3:

{ 10, 20, 30, 40, 50 }

Елемент з індексом 4 є останнім.

```
1 #include <algorithm> // для std::swap. в C++11 используйте заголовок <utility>
2 #include <iostream>
3
4 int main()
5 {
6     const int length = 5;
7     int array[length] = { 30, 50, 20, 10, 40 };
8
9     // Перебираем каждый элемент массива
10    // (кроме последнего, он уже будет отсортирован к тому времени, когда мы до него доберемся)
11    for (int startIndex = 0; startIndex < length - 1; ++startIndex)
12    {
13        // В переменной smallestIndex хранится индекс наименьшего значения, которое мы нашли в этой итерации
14        // Начинаем с того, что наименьший элемент в этой итерации - это первый элемент (индекс 0)
15        int smallestIndex = startIndex;
16
17        // Затем ищем элемент поменьше в остальной части массива
18        for (int currentIndex = startIndex + 1; currentIndex < length; ++currentIndex)
19        {
20            // Если мы нашли элемент, который меньше нашего наименьшего элемента
21            if (array[currentIndex] < array[smallestIndex])
22            {
23                // запоминаем его
24                smallestIndex = currentIndex;
25            }
26
27            // smallestIndex теперь наименьший элемент
28            // меняем местами наше начальное наименьшее число с тем, которое мы обнаружили
29            std::swap(array[startIndex], array[smallestIndex]);
30        }
31
32        // Теперь, когда весь массив отсортирован - выводим его на экран
33        for (int index = 0; index < length; ++index)
34            std::cout << array[index] << ' ';
35
36        return 0;
37    }
```

Сортування методом бульбашки

Алгоритм бульбашкового сортування – це досить простий в реалізації алгоритм для сортування масивів. Можна зустріти й інші назви: бульбашкове сортування, Buble sort або сортування простими. Алгоритм названий так, тому що більше чи менше значення «спливає» (зсувається) до краю масиву після кожної ітерації.

Сутність алгоритму полягає у попарному порівнянні двох сусідніх чисел і при необхідності вони міняються місцями на кожному прогоні. Прогони завершуються коли вже жодної перестановки не відбувається.

Розглянемо попередній приклад: { 30, 50, 20, 10, 40 }

Спочатку порівнюються числа 30 і 50. Так як 50 більше тридцяти нічого не відбувається. Далі порівнюється 50 і 20. 20 менше, тоді ці числа міняються місцями: { 30, 20, 50, 10, 40 }. Далі порівнюється 50 і 10. знову виконується перестановка: { 30, 20, 10, 50, 40 }. Потім порівнюємо останню пару чисел 50 і 40 і знову виконуємо перестановку. В результаті першого прогону отримаємо: { 30, 20, 10, 40, 50 }.

Виконаємо другий прогін і запишемо його покроково:

1. { 20, 30, 10, 40, 50 }.

2. { 20, 10, 30, 40, 50 }.

Так як відбувались перестановки виконаємо наступний прогін: { 10, 20, 30, 40, 50 }. Якщо виконаємо ще один прогін, то перестановок не буде, що означає, що масив відсортований.

Приклад реалізації методу сортування за спаданням наведено нижче.

```
#include <iostream>
using namespace std;

int main()
{
    /* Установим размер массива */
    int n; // Кол-во элементов
    cout << "Количество элементов: ";
    cin >> n;

    /* Заполним массив значениями */
    int mass[n];
    for(int i = 0; i < n; ++i)
    {
        cout << i+1 << "-ый элемент: ";
        cin >> mass[i];
    }

    /* Выведем исходный массив */
    cout << "Исходный массив: ";
    for(int i = 0; i < n; ++i)
    {
        cout << mass[i] << " ";
    }
    cout << endl;

    /* Отсортируем массив по убыванию */
    for(int i = 1; i < n; ++i)
    {
        for(int r = 0; r < n-i; r++)
        {
            if(mass[r] < mass[r+1])
            {
                // Обмен местами
                int temp = mass[r];
                mass[r] = mass[r+1];
                mass[r+1] = temp;
            }
        }
    }

    /* Выведем отсортированный массив */
    cout << "Отсортированный массив: ";
    for(int i = 0; i < n; ++i)
    {
        cout << mass[i] << " ";
    }
    cout << endl;

    return 0;
}
```

4

5

6

7

8

9

0

1

2

3

Цей метод теж повільно працює для великих масивів, тому, на практиці, для сортування застосовують спеціальні методи «швидкого» сортування

Оскільки сортування масивів настільки поширено, то стандартна бібліотека C ++ надає вбудовану функцію сортування - std::sort. Вона знаходиться в заголовку <algorithm> і викликається наступним чином:

```
1 #include <algorithm> // для std::sort
2 #include <iostream>
3
4 int main()
5 {
6     const int length = 5;
7     int array[length] = { 30, 50, 20, 10, 40 };
8
9     std::sort(array, array+length);
10
11     for (int i=0; i < length; ++i)
12         std::cout << array[i] << ' ';
13
14     return 0;
15 }
```

Дії з матрицями

Дано квадратні матриці A і B порядку n . Отримати матрицю, $A(B-E)+C$ де E – одинична матриця порядку n , а елементи матриці C обчислюються за формулою: $C_{ij} = \frac{1}{i+j}$

Реалізуємо зберігання матриці у вигляді двовимірної масиви. Операції додавання і віднімання матриць виконуються поелементно. Множення вимагає наявності у лівого співмножника такого ж числа стовпців, як і число рядків у правого співмножника. Поелементно множимо кожен рядок першої матриці на кожен стовпець другої накопичуючи суму значень в елементах результуючої матриці.

```
A705
1  #include <iostream>
2  using namespace std;
3
4  int main() {
5      int n;
6      cin >> n;                                //ввод из стандартно
7      int A[n][n];
8      int B[n][n];
9      int Z[n][n];
10     int H[n][n];
11     int E[n][n];
12     float C[n][n];
13     float Ans[n][n];
14
15     for (int i = 0; i < n; i++){                //матрица A
16         for (int j = 0; j < n; j++){
17             cin >> A[i][j];
18         }
19     }
20
21     for (int i = 0; i < n; i++){                //матрица B
22         for (int j = 0; j < n; j++){
23             cin >> B[i][j];
24         }
25     }
26
27     for (int i = 0; i < n; i++){                //единичная матриц
28         for (int j = 0; j < n; j++){
29             if (i == j) E[i][j] = 1;
30             else E[i][j] = 0;
31         }
32     }
```

```

33
34     for (int i = 0; i < n; i++){                //разность матриц B и E
35         for (int j = 0; j < n; j++){
36             Z[i][j] = 0;
37             Z[i][j] = B[i][j] - E[i][j];
38         }
39     }
40
41     for (int i = 0; i < n; i++){                //умножение матриц A и (B - E)
42         for (int j = 0; j < n; j++){
43             H[i][j] = 0;
44             for (int t = 0; t < n; t++){
45                 H[i][j] += A[i][t] * Z[t][j];
46             }
47         }
48     }
49
50     for (int i = 0; i < n; i++){                //матрица C
51         for (int j = 0; j < n; j++){
52             C[i][j] = 0;
53             C[i][j] = 1.0/(i+1 + j+1);
54         }
55     }
56
57     for (int i = 0; i < n; i++){                //матрица A(B-E)+C
58         for(int j = 0; j < n; j++){
59             Ans[i][j] = H[i][j] + C[i][j];
60             printf("%1.2f", Ans[i][j]);
61             cout << " ";
62         }
63         cout << endl;
64     }
65
66     return 0;
67 }

```

Вставка елементів у масив

Перш за все треба розуміти, що при додаванні даних в масив їх кількість буде зростати. А це значить, що треба заздалегідь подбати про виділення потрібної кількості пам'яті під масив.

Наприклад, якщо ставиться завдання «перед кожним числом», що відповідає якійсь умові, «додати новий елемент», то скільки необхідно виділяти пам'яті? У масиві може не виявитися жодного елемента, що відповідає заданій умові, а можуть і все елементи підходити під цю умову. Значить, треба враховувати найгірший варіант. В даному випадку необхідно виділити пам'яті під $2n$ елементів масиву.

Друге. Дані можна додавати перед елементом, що відповідає заданій умові, а можна – після такого елемента. Це визначається умовою розв'язуваної задачі.

Приклад. Дан масив з n цілих чисел. Перед кожним позитивним числом додати число 0.

До вставки									
x									
2	-11	-5	7	4					
0	1	2	3	4	5	6	7	8	9

После вставки									
x									
0	2	-11	-5	0	7	0	4		
0	1	2	3	4	5	6	7	8	9

```
#include <iostream>
using namespace std;
int main()
{ int n = 5, i, j, k;
  int x[2*n];
  cout << "Input " << n << " numbers:" << endl;
  for(i = 0; i < n; i++)
    cin >> x[i];

  k = n; // Первоначально в массиве все элементы.
        // Затем k будет означать количество после добавления.
  for(i = 0; i < k; i++)
  { if(x[i] > 0) // условие на вставку элемента массива
    { for(j = k; j > i; j--) // сдвиг в право
      x[j] = x[j-1];
      x[i] = 0; // вставка нового элемента
      k++;     // увеличиваем количество
      i++;     // делаем обход уже обработанного элемента
    }
  }

  cout << "Otvet:" << endl;
  for(i = 0; i < k; i++)
    cout << x[i] << endl;
  return 0;}
```

Тип даних вказівник

Найбільш цікаве, що є в мовах C і C ++ - це покажчики. Без розуміння принципів роботи з покажчиками можна навчитися працювати на цих двох мовах.

Будь-яка програма, як відомо, призначена для обробки даних. Всі дані умовно можна поділити на власне дані (числа, рядки і т.д.) і на адреси в оперативній пам'яті, де зберігаються ці дані. Ми оперуємо в програмі іменами змінних, але для виконання програми процесором комп'ютера всі імена ще на етапі компіляції замінені на адреси. Тому комп'ютер працює або з деякими даними, або з адресою, де зберігається це дане. Для виконання різних дій з адресами служать покажчики.

Показчик – це беззнакове ціле, яке використовується для зберігання адреси будь-якого ділянки пам'яті.

Показчик завжди є змінною величиною. Показники в 16-розрядній операційній системі MS DOS так само були 16-розрядними. В даний час зазвичай використовуються 32-розрядні операційні системи (Windows, Linux і ін.), В яких показники займають 4 байта (32-розрядний ціле без знака). І хоча це по внутрішньому подання в точності відповідає типу unsigned int, але показчик і змінна типу unsigned int – це дві великі різниці.

Всякий показчик використовується для роботи з даними, які мають якийсь свій тип і, відповідно, свій розмір, наприклад, double або int. Отже, при описі показчика необхідно сказати, на об'єкти якого типу він буде налаштований.

Формальний опис показчика таке:

Тип_данных * Ім'я_показчика;

де

- Тип_данных – будь-який певний тип (стандартний або користувацький);
- * – знак «зірочка» тут є ознакою того, що мова йде про показчику;
- Ім'я_показчика – визначається за загальними правилами (як для будь-якого ідентифікатора користувача).

Дамо опис показчика для роботи з даними типу double:

double *u;

Цей запис необхідно розуміти так: «**u** – це показчик на об'єкт типу double».

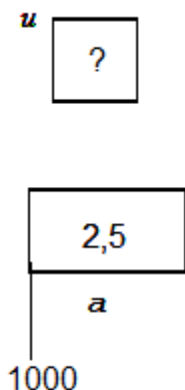
На який об'єкт налаштований цей показчик? В даний момент ні на який. Значення показчика **u** в момент виділення пам'яті не визначено.

Нехай є змінна **a** типу double:

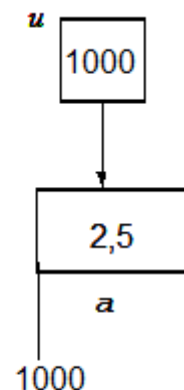
double a = 2.5;

Налаштуємо показчик **u** на цю змінну **a**:

u = &a;



а)до
настроюки
указателя



б)после
настроюки
указателя

Тут знак **&** означає обчислення адреси, тобто ми обчислюємо адресу змінної **a** і записуємо його в комірку **u**. Нехай **a** знаходиться за адресою 1000. Ось це число-адреса і стане вмістом **u**. Тепер покажчик **u** налаштований на початкову адресу змінної **a**. Що це нам дає? Чи не багато - не мало, а доступ до вмісту змінної **a**, якщо застосувати операцію розіменування:

```
cout << *u << endl;
```

Цим оператором ми виводимо на екран монітора вміст змінної **a**, тому що покажчик **u** налаштований на цю ж змінну **a**.

Можна і змінити значення змінної **a**, використовуючи покажчик **u**:

```
*u = 11.3;
```

```
cout << a << endl;
```

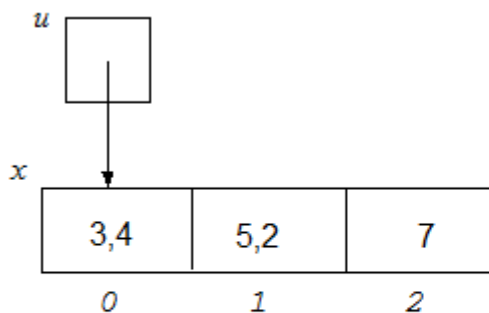
На екран буде виведено число 11.3, хоча раніше змінна **a** мала значення 2.5. Отже, користуючись покажчиком, ми дійсно змінили вміст змінної **a**.

Операції для роботи з покажчиками

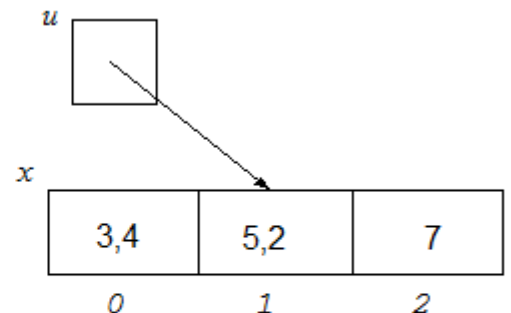
Хоча про всяк покажчик по внутрішньому подання – це беззнакове ціле, але за характером операцій він помітно відрізняється від будь-яких цілих типів. Розглянемо всі операції, допустимі для покажчиків.

1. **Присвоювання (=)**. Вказівником можна привласнити адресу якоїсь змінної або значення іншого покажчика. Наведемо приклад:

```
double a = 2.5;
```



а) до выполнения операции автоувеличения



б) после выполнения операции автоувеличения

```
double *u;
```

```
double *v;
```

```
u = &a; // Указатель u настроен на переменную a
```

```
v = u; // Теперь и указатель v настроен на переменную
```

a

2. **Розіменування (*)** – це унарна операція дозволяє звернутися до комірки пам'яті, на яку попередньо налаштований покажчик. наприклад:

```
*u = a + 2;
```

Тепер значення змінної **a** дорівнює 4.5.

3. **Обчислення адреси (&)**

```
double b = -1.5;
```

```
v = &b; // указатель v настроен на переменную b.
```

4. **Автомобільшення (++)**.

Для змінної будь-якого числового типу автозбільшення означає збільшення на 1 значення того, що було в комірці, наприклад:

```
int i = 5;  
i++; // Тепер i равно 6.
```

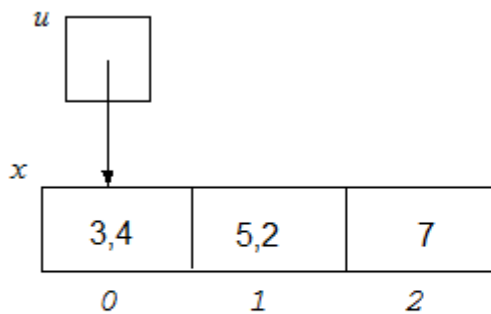
Для покажчика операція автозбільшення означає переключення на наступний об'єкт того ж типу, що і тип, на який був налаштований покажчик.

Нехай є масив **x** з трьох чисел типу **double** і покажчик **u** на тип **double**:

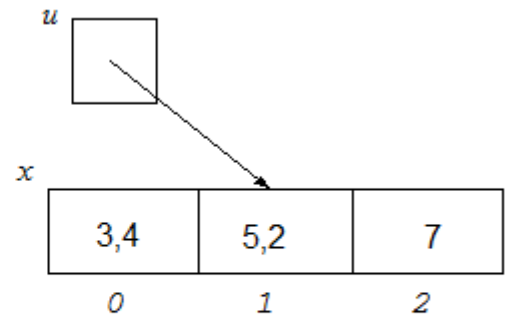
```
double x[3] = {3.4, 5.2, 7};  
double *u;
```

А тепер попрацюємо з масивом через покажчик (див. малюнок нижче):

```
u = &x[0]; // настроим указатель u на начало массива  
cout << *u << endl; // будет напечатано число 3.4  
u++;       // переключаемся на 1 объект вправо, т.е. на  
x[1]  
cout << *u << endl; // будет напечатано число 5.2
```



а) до выполнения
операции
автоувеличения



б) после выполнения
операции
автоувеличения

Як бачимо, автозбільшення дає зміщення на 1 один об'єкт, а в байтах для покажчика на тип **double** це дорівнюватиме 8 (тобто зміщується на 8 байт). Таким чином, адреса, що зберігається в покажчику **u**, за рахунок виконання операції автозбільшення (++), збільшиться на 8.

Подібна картина буде і для покажчиків на інші типи, наприклад: для покажчика на тип **int** крок зсуву при автозбільшенні дорівнює 4.

5. Автозменшення (-).

Тут все аналогічно тому, що було сказано про автозбільшенн.

6. Додавання (+).

До покажчика можна додати ціле число. Якщо складання зіставити з автозбільшенням, то нескладно зрозуміти, що оператори

```
double *v;  
v = &x[0];  
v = v + 2;  
cout << *v << endl;
```

призводять до зміщення покажчика *v* на 2 об'єкти вправо, тобто адреса, записана в *v*, збільшиться на 16 ($2 * 8 = 16$). буде надруковано число 7 (див. рис. вище).

Зауваження. Не можна складати два покажчика (що може означати сума значень двох покажчиків?), Тому вона просто заборонена.

7. Віднімання (-)

Для покажчиків допустимо як віднімання числа з покажчика, так і визначення різниці двох покажчиків.

Приклад 1:

```
double *u = &x[2]; // Настроить указатель
                        // на 2-й элемент массива
int k = 2;           // Пусть это будет смещением
u = u - k;           // Смещаем влево значение указателя
                        // на k объектов типа double
```

Приклад 2:

```
double *u, *v;
// Далее какие-то действия
// .....
int k = u - v;
```

Тепер *k* визначає відстань між об'єктами в пам'яті, на які були попередньо налаштовані покажчики.

8. *Множення, ділення і обчислення залишку* для покажчиків безглузді, а тому заборонені.

9. *Операції порівняння.* Для покажчиків допустимі всі без винятку операції порівняння. Найчастіше використовується перевірка на рівність (==) і на нерівність (!=). приклад:

```
double *u;
.....
if(u != NULL) {
    // Можно работать с указателем. К примеру, так:
    u++; }
else {
    //Работа с указателем невозможна
}
```

10. *Логічні операції* для покажчиків не застосовують, але оператор типу `if(u && v) { операторы }`

помилки не викликає, хоча важко надати сенс такому запису.

Область застосування покажчиків

Використовуючи механізм покажчиків, можна вирішувати найрізноманітніші завдання. Найчастіше покажчики використовують для:

- роботи з масивами і особливо з масивами типу `char`;
- передачі даних в функцію за адресою;
- побудова складних динамічних структур даних.

Подробиці використання покажчиків по перерахованих темах будуть приведені в наступних темах.

Вказівники та масиви

Взаємозв'язок між масивами і покажчиками

Між покажчиками і масивами існує дуже тісний зв'язок. Щоб її зрозуміти, розглянемо простий приклад.

Завдання. Знайти суму значень елементів масиву, що складається з n дійсних чисел.

Вирішимо основну частину завдання (підсумовування елементів масиву) декількома способами.

Можливий 1-й варіант рішення (наводимо повний текст програми):

```
#include <iostream>
using namespace std;
int main()
{ int n = 5;
  double x[n];
  double *u, s;
  int i;
  cout << "Input " << n << " numbers:" << endl;
  for(i = 0; i < n; i++)
    cin >> x[i];
  cout << "1 variant:" << endl;
  //-----
  for(s = 0, i = 0; i < n; i++)
    s += x[i];
  cout << "s=" << s << endl;
  return 0;}
```

Цей 1-й варіант – звичайне використання масиву. Саме так ми вирішували завдання з масивами досі.

Варіант №2 - використовуємо покажчик **u** для доступу до елементів масиву. У заголовку циклу операцією **u ++** кожен раз перемикаємо покажчик на наступний елемент масиву **x**. Наведемо для стислості не весь текст програми, а тільки цикл з підсумовуванням:

```
for(s = 0, u = &x[0], i = 0; i < n; i++, u++)
  s += *u;
```

Якщо змінений варіант програми запустити на виконання, то можна переконатися, що все відмінно працює. І відповідь вийде той же, що і для 1-го варіанту.

Варіант №3 – покажчик **u** залишається налаштованим на початок масиву, а перемикання на наступні елементи масиву робиться за рахунок зміни змінної циклу **i**:

```
for(s = 0, u = &x[0], i = 0; i < n; i++)
  s += *(u + i);
```

Також все нормально. Відповідь збігається з колишнім.

Варіант №4 - а що буде, якщо написати так:

```
for(s = 0, i = 0; i < n; i++)
  s += *(x + i);
```

Результат знову виходить правильним. Звернення до елементу масиву $x[i]$ замінено на розіменування для адресного виразу $*(x + i)$. Чому це працює? Справа в тому, що всі вирази типу $x[i]$ компілятор розглядає як $*(x + i)$. Напрошується питання: масив – це покажчик? Відповідь: так, але з деякими обмеженнями, які розберемо трохи пізніше.

Варіант №5 - продовжимо експеримент:

```
for(s = 0, i = 0; i < n; i++)  
    s += i[x];
```

Виглядає абсурдно (проста змінна i стала як би масивом, а масив – як би індексом), але працює! Причина – дивись варіант 4. Вираз $i[x]$ для компілятора означає те ж, що $*(i + x)$, яке, природно, рівносильно $*(x + i)$.

Варіант №6 – «посилимо» ситуацію.

Змінімо початкове значення змінної циклу $i = -2$, а цю зміну компенсуємо додаванням числа 2 в індексний вираз, тобто «Нормальне» звернення до елементу масиву виглядало б як $x[i + 2]$, у нас же все набагато «веселіше»:

```
for(s = 0, i = -2; i < n-2; i++)  
    s += 2[i+x];
```

А тепер i ціле число «прикидається» масивом. Причина правильної роботи такої ненормальності та ж, що i для варіантів 4, 5.

Звичайно, в програмах, призначених для загального огляду, треба використовувати тільки «правильні» конструкції виду $x[i]$ або $*(x + i)$, щоб нікого не ставити в глухий кут своєю оригінальністю. А ось для експериментів можна (і потрібно!) Пробувати все. Це корисно для кращого розуміння мов програмування.

Варіант №7 – у варіанті 4 ми переконалися, що наш звичайний масив x – це покажчик, але деякі сумніви залишилися? Спробуємо працювати з масивом x так, як працювали з покажчиком u в другому варіанті, тобто застосуємо до x операцію автозбільшення:

```
for(s = 0, i = 0; i < n; i++,  
    x++) // Выражение x++ записано отдельной строкой  
        // специально для того, чтобы убедиться, что  
        // именно это выражение «не нравится»  
        // компилятору  
    s += *x;
```

І тільки зараз компілятор фіксує помилку і саме для вираження, що показано окремим рядком, тобто $x++$. В чому причина?

Справа в тому, що пам'ять для масиву виділяється при компіляції. Початкова адреса масиву, кількість елементів і як наслідок – обсяг пам'яті, що відводиться під масив, фіксуються на етапі компіляції і продовжують залишатися незмінними в процесі виконання програми. Для покажчика настройка на потрібну ділянку пам'яті виконується на етапі виконання програми, що дозволяє перенастроювати покажчик по ходу роботи програми багаторазово і на різні ділянки пам'яті. Таким чином, покажчик поводить

як змінна величина, а масив – як константа. Тому масив і не є повноцінним показником. *Масив* – це *константних показчик*.

Динамічні вектори

За допомогою показчиків можна створювати динамічні масиви – це масиви, пам'ять для яких виділяється не на етапі компіляції, а під час виконання програми. Це дуже зручно. Адже програміст далеко не завжди може заздалегідь уявити, якого розміру має бути той чи інший масив. Вказувати розмір масиву з великим запасом (на всякий випадок!) Не розумно. Простіше на етапі виконання запропонувати користувачеві задати самому необхідну кількість елементів масиву, або необхідний розмір масиву повинен обчислюватися в програмі з якихось формулах.

Для динамічного виділення пам'яті служить операція **new**, звільнення стала непотрібною пам'яті виконується операцією **delete**.

Пам'ять виділяється не в межах області програми, а в так званій «кучі» – області за межами програми.

Наведемо формальний запис оператора, за допомогою якого виділяється пам'ять під динамічний масив:

Указатель = new Тип [Количество_элементов] ;

де **Тип** – це той же тип даних, що і тип, на який розрахований показчик.

Запис оператора для звільнення динамічної пам'яті ще простіше:

delete [] Указатель ;

Тут порожні квадратні дужки – це ознака того, що звільняється пам'ять, відведена для масиву, а не для одиночного об'єкту.

Приклад. Дан масив цілих чисел з n елементів. Інвертувати цей масив, тобто поміняти місцями перший елемент і останній, другий – і передостанній і так далі. Пам'ять під масив виділяти динамічно.

Можливий текст програми:

```
#include <iostream>
using namespace std;
int main()
{
    int i, j, n;
    int *x, c;

    // Задаём размер будущего массива
    do {cout << "n=";
        cin >> n;
    } while(n < 1);

    // Динамически выделяем память под массив
    x = new int[n];

    // Ввод массива с клавиатуры
    cout << "Input " << n << " numbers:" << endl;
    for(i = 0; i < n; i++)
```

```

        cin >> x[i];

    // Перестановка элементов массива
    for(i = 0, j = n - 1; i < j; i++, j--)
    {   c = x[i];
        x[i] = x[j];
        x[j] = c;
    }

    // Вывод массива на экран монитора
    cout << "Otvet:" << endl;
    for(i = 0; i < n; i++)
        cout << x[i] << endl;

    // Освобождение динамической памяти, отводимой под
массив
    delete [] x;

    return 0;}

```

Як видно з прикладу, після того, як пам'ять під динамічний масив виділена, з ним можна працювати так само, як і зі звичайним масивом.

Важливо не забувати звільняти пам'ять, займану динамічними об'єктами. Інакше можуть початися проблеми в роботі комп'ютера в цілому. Невивільнені програмою ділянки пам'яті можуть виявитися недоступними після завершення роботи програми. Це явище називають «витік пам'яті». Звичайно, в сучасних операційних системах завжди реалізується «прибирання сміття», тобто є якась службова програма, яка наводить порядок, відшукує «безгоспні» ділянки пам'яті і повертає їх у спільне користування. Але не треба надмірно покладатися на них, тому що будь-які проблеми операційної системи через невдалу оптимізації ОС, через віруси і тому подібне, можуть відбитися на роботі «збирача сміття». У цьому випадку саме ваша програма буде регулярно «підвішувати» систему.

Поодинокі динамічні об'єкти

Виділяти пам'ять динамічно можна не тільки під масиви, а й під поодинокі об'єкти. Звичайно, якщо одиночний об'єкт має стандартний тип (double, int і т.д.), то простіше використовувати звичайні змінні. При роботі з одними типами (структури, класи) в ряді випадків є сенс в динамічному виділенні пам'яті під поодинокі об'єкти, а для побудови динамічних структур типу списки, черги, дерева і т.д. це просто необхідно.

Нехай є якийсь клас (або структура) **Alfa** (про класи і структурах мова піде пізніше. Зараз можна ставиться до них просто як до типів даних). Тоді можна зробити наступне:

а) Створюємо покажчик x на об'єкт типу **Alfa**:

Alfa *x;

б) Виділяємо пам'ять під динамічний об'єкт:

```
x = new Alfa;
```

в) Далі працюємо з об'єктом класу через покажчик *x*. Як це робиться – будемо розбиратися пізніше, при вивченні структур і класів.

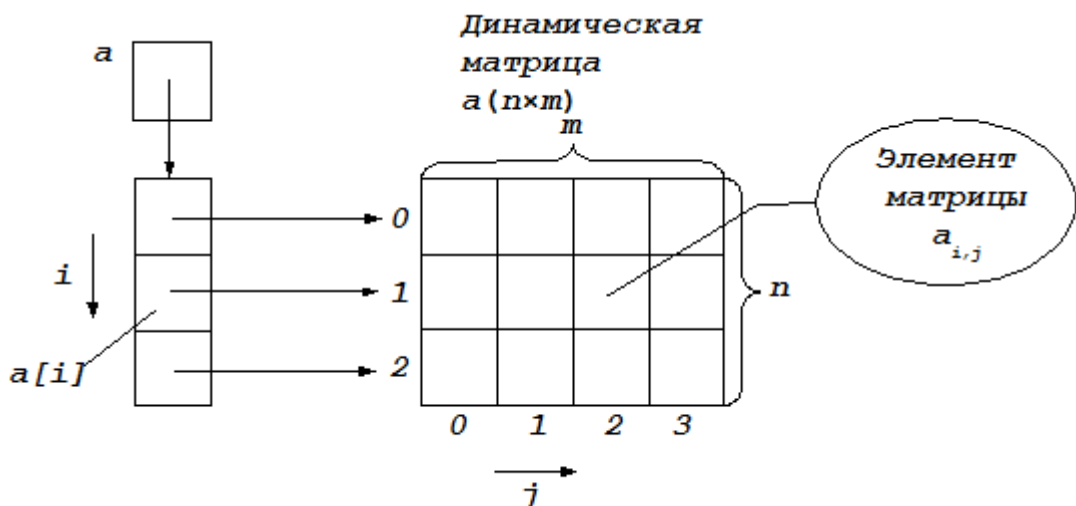
г) Як тільки об'єкт став непотрібним – звільняємо займану ним пам'ять:

```
delete x;
```

Загалом, все робиться за аналогією з динамічними масивами, але тільки ще простіше. Не треба вказувати кількість елементів при виділенні пам'яті, не потрібні квадратні дужки при її звільненні.

Двовимірні динамічні масиви

Як ми вже знаємо з теми «Матриці», матриця – це масив з одновимірних масивів, кожен з яких представляє один рядок матриці. Напрошується ідея: завести для кожного рядка матриці покажчик і динамічно виділити пам'ять під задану кількість елементів. Так як рядків в матриці може бути досить багато, а покажчики, які використовуються для виділення пам'яті під кожен рядок матриці, мають один і той же тип, то ці покажчики є сенс об'єднати в одновимірний масив покажчиків. Природно, пам'ять під масив покажчиків також будемо виділяти динамічно. А адресу цього масиву покажчиків будемо зберігати в покажчику на покажчик. На малюнку покажемо, як це може виглядати:



Тут *a* – це покажчик на масив покажчиків. Формальний опис для нього наступний:

Тип `** имя_указателя;`

a[i] - покажчики на рядки матриці, в яких будуть в подальшому зберігатися дані, тобто елементи матриці *a[i][j]*

Порядок роботи з динамічною матрицею розглянемо на прикладі.

Приклад. У прямокутній матриці підрахувати кількість негативних елементів. Пам'ять під матрицю виділяти динамічно.

Можливий текст програми:

```
#include <iostream>  
using namespace std;  
int main()  
{   int n = 3; // число строк  
      int m = 4; // число столбцов  
}
```

```

double **a; // указатель на указатель на тип double
int i, j;
// Динамическое выделение памяти под матрицу:
// -----
// 1) создаём массив указателей на тип double
a = new double* [n];

// 2) выделяем память (построчно) под матрицу
for(i = 0; i < n; i++)
    a[i] = new double[m];
//-----
// Ввод матрицы с клавиатуры
cout << "Matriz A("<< n << "*" << m << "):" <<
endl;
for(i = 0; i < n; i++)
    for(j = 0; j < m; j++)
        cin >> a[i][j];
// Подсчёт количества отрицательных чисел в матрице
int k = 0;
for(i = 0; i < n; i++)
    for(j = 0; j < m; j++)
        if(a[i][j] < 0)
            k++;
cout << "k=" << k << endl;
// Освобождение динамической памяти:
// -----
// 1) вначале освободим память, отведённую
собственно под матрицу, т.е. под данные
for(i = 0; i < n; i++)
    delete []a[i];
// 2) теперь освобождаем память, занятую массивом
указателей
delete []a;
// -----
return 0;}

```

Як бачимо, спочатку виділяється пам'ять під масив покажчиків, потім під дані. Звільнення пам'яті робиться в зворотному порядку: спочатку звільняють пам'ять, зайняту даними матриці, а потім – масивом покажчиків.

Коли пам'ять виділена, дії з елементами динамічної матриці виконуються точно так же, як і зі звичайною матрицею, тобто всюди, де необхідно, використовуємо звичайну форму звернення до елементу матриці.