

ДИНАМІЧНА ПАМ'ЯТЬ. ДИНАМІЧНІ МАСИВИ. АЛГОРИТМИ СОРТУВАННЯ МАСИВІВ

Динамічна пам'ять

Динамічна пам'ять (heap, купа) – це пам'ять, в якій змінні розміщуються динамічно. Основна потреба у динамічному виділенні пам'яті виникає тоді, коли розмір або кількість даних, які необхідно виділити чи використати, наперед невідомі і визначаються у процесі виконання програми.

Засоби динамічного управління пам'яттю дають змогу розширити можливості програмування. Під час свого виконання програма отримує можливість самостійно визначати потрібні розміри пам'яті і за потреби в реальному часі створювати та знищувати динамічні об'єкти.

Основні *відмінності динамічного масива від статичного* такі:

- пам'ять для динамічного масива виділяється під час виконання програми за допомогою певних функцій;
- кількість елементів динамічного масива може бути задано змінною, але в програмі вона обов'язково має бути визначена до виділення пам'яті для масива.

Динамічні об'єкти не є локальними для певної функції (локальною може бути змінна, що містить адресу такого об'єкта). Вони можуть створюватись в одній функції, опрацьовуватись у другій (якщо перша повідомить їй адресу об'єкта), а знищуватись у третій. Таким чином, динамічний об'єкт доступний з будь-якої функції програми, якій відома його адреса.

Час життя динамічного об'єкта – це час від моменту його створення до знищення, тобто динамічно виділена пам'ять існує поки її не звільнити. Якщо динамічно виділену пам'ять не звільнити, то в операційній системі може виникнути ситуація нестачі вільної пам'яті.

У мові C++ *для виділення динамічної пам'яті використовується оператор new*, який має такий синтаксис:

```
<тип_даних> *<змінна> = new <тип_даних> (<змінна>);
```

Динамічна пам'ять, виділена за допомогою оператора **new**, *автоматично не звільняється*, тому її обов'язково *потрібно звільняти самостійно* за допомогою оператора **delete**, синтаксис якого такий:

```
delete [] <вказівник>
```

Параметр `[]` (квадратні дужки) в операторі `delete` необов'язковий. При цьому квадратні дужки повинні бути порожніми, оскільки операційна система контролює

кількість виділеної пам'яті для кожного об'єкта і їй відома кількість байтів, яку потрібно звільнити.

Наприклад, виділити пам'ять для цілого числа за допомогою оператора `new` можна так:

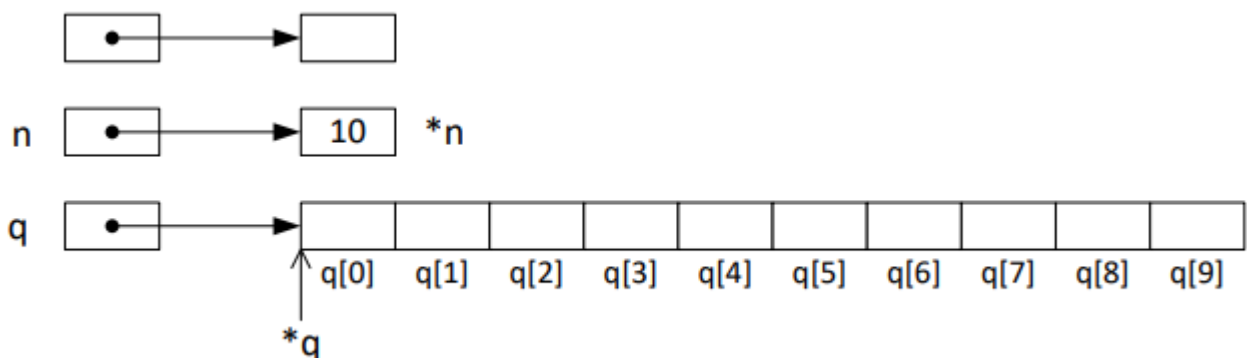
```
int* n = new int;      // виділення пам'яті
*n = 10; // ініціалізація динамічної змінної значенням 5
delete n; // звільнення пам'яті
```

Або так

```
int n = 10;
// виділення пам'яті та ініціалізація динамічної змінної числом 7
int *q = new int(n);
delete q; // звільнення пам'яті
```

Доступ до числа здійснюється через вказівник на нього.

Команда створює динамічну змінну цілого типу, налаштовує вказівник `n` на цю змінну та ініціалізує її значенням 10. Зауважимо, що ініціалізатор в цьому випадку має вигляд `(n)`, а не `= n`, оскільки команда з ініціалізатором `= n` — недопустима (вона приводить до помилки при компіляції).



Наступна команда

```
int* q = new int[10];
```

створює динамічний масив із 10 елементів цілого типу і налаштовує вказівник `q` на цей масив. Тобто, виділяється пам'ять для 10 величин типу `int` (динамічного масиву із 10 елементів), і записує адресу початку цієї області у змінну `q`, яка може трактуватися як ім'я масиву.

Динамічні одновимірні масиви

*Робота з динамічними масивами відбувається за допомогою змінних типу **вказівник**.* Після створення масива така змінна вказує на його початок, тобто містить адресу його першого елемента.

Розмір динамічного масива задається не константою, а *змінною*, значення якої під час виконання програми вводить з клавіатури користувач або вона обчислюється програмно за певними формулами.

Синтаксис оголошення динамічного одновимірного масива за допомогою оператора **new** такий:

```
<тип_даних> *<ім'я_вказівника> = new <тип_даних>
                               [кількість_елементів];
```

Оператор **new** виділяє місце в пам'яті для певної кількості елементів певного типу, а адреса цієї ділянки пам'яті записується у змінну-вказівник.

Наприклад

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

int main() {
    // кількість елементів масива
    int n;
    cout << "Enter number of elements: ";
    cin >> n; // зчитати кількість елементів масива
    // створення динамічного одновимірного масива
    int *b = new int [n];
    // перевірка, чи масив створився
    if (b == nullptr) {
        cout << "\nMemory error!\n";
        return -1; // якщо масив не створився – вихід із програми
    }
    // введення масива
    cout << "\n\nEnter elements:\n";
    for (int i = 0; i < n; i++) {
        cout << "b[" << i+1 << "]=";
        cin >> b[i];
    }

    // опрацювання масива
    // звільнення пам'яті, виділеної під масив
    delete [] b;

    return 0;
}
```

Для доступу до значень елементів такого масива використовується одна з наступних форм:

- індексна форма:

```
<ім'я_масива>[<індекс>]
```

- вказівникова форма:

```
*(<ім'я_масива> + <індекс>)
```

Динамічні двовимірні масиви

Динамічний двовимірний масив як одновимірний

Двовимірний динамічний масив займає в пам'яті сусідні комірки, тобто

зберігається, як одновимірний масив

Для звернення до довільного елемента такого масива використовують один із таких способів:

1) індексний спосіб:

```
<ім'я_масива> [ <індекс_рядка> * <кількість_стовпчиків> +  
                <індекс_стовпчика> ]
```

2) адресний спосіб:

```
*(<ім'я_масива> + <індекс_рядка> * <кількість_стовпчиків> +  
   <індекс_стовпчика>)
```

Синтаксис оголошення динамічного двовимірного масива як одновимірного за допомогою оператора `new` такий:

```
<тип_даних> *<ім'я_вказівника> = new <тип_даних>  
[кількість_елементів];
```

Виділення пам'яті під масив:

```
int *m = new int [rowCount * colCount];
```

Звільнення пам'яті:

```
delete [] m;
```

Динамічний двовимірний масив як двовимірний

Для того, щоб працювати з елементами двовимірного масива звичайним способом (як з двома вимірами) *спочатку виділяється пам'ять для стовпчика вказівників на рядки матриці, а потім окремо на кожний рядок*. Звільнення пам'яті потрібно виконувати у зворотному порядку.

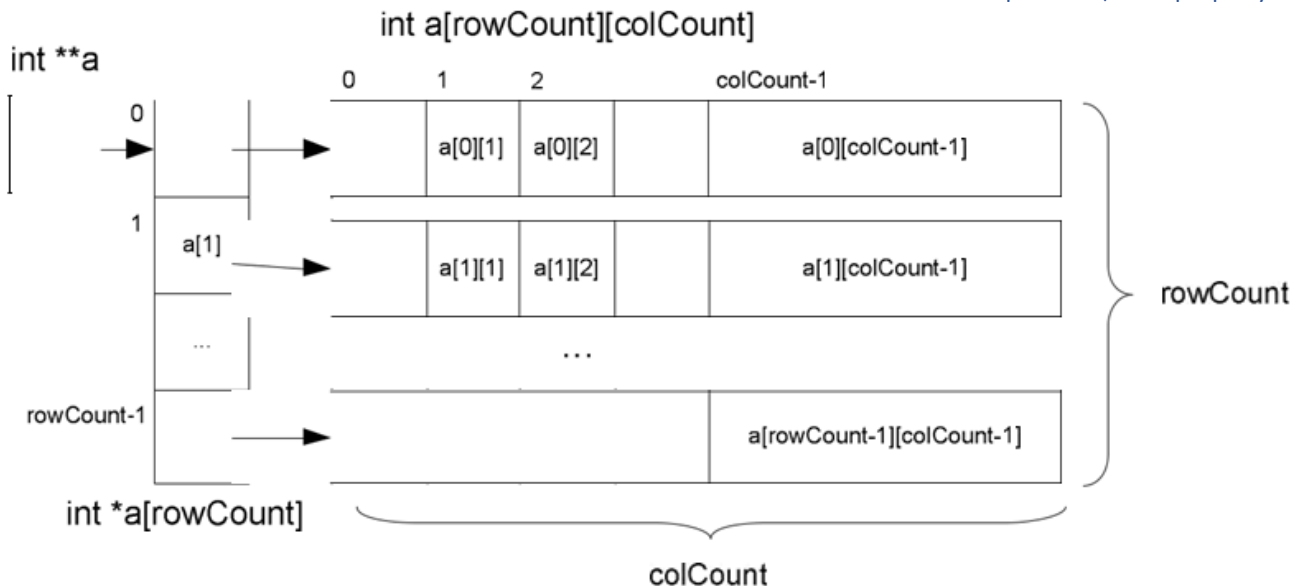
Адреси нульових елементів динамічного двовимірного масива зберігаються в допоміжному масиві, пам'ять під який потрібно виділяти завчасно. Елементами такого масива будуть адреси значень певного типу, а їхнім типом – “вказівник на значення певного типу”, тобто `<тип>*`. *Оголошуючи такий масив, слід записувати дві зірочки.*

З таким динамічним двовимірним масивом можна працювати як зі звичайним двовимірним масивом, звертаючись до кожного елемента за його індексом, вказаним у квадратних дужках (*індексна форма*):

```
<ім'я_масива> [<індекс_рядка>][<індекс_стовпчика>]
```

Крім того, для доступу до елементів такого масива інколи використовують адресний вираз (*вказівникова форма*):

```
*(<ім'я_масива> + <індекс_рядка>) + <індекс_стовпчика>)
```



Наприклад, створимо динамічний двовимірний масив як двовимірний:

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

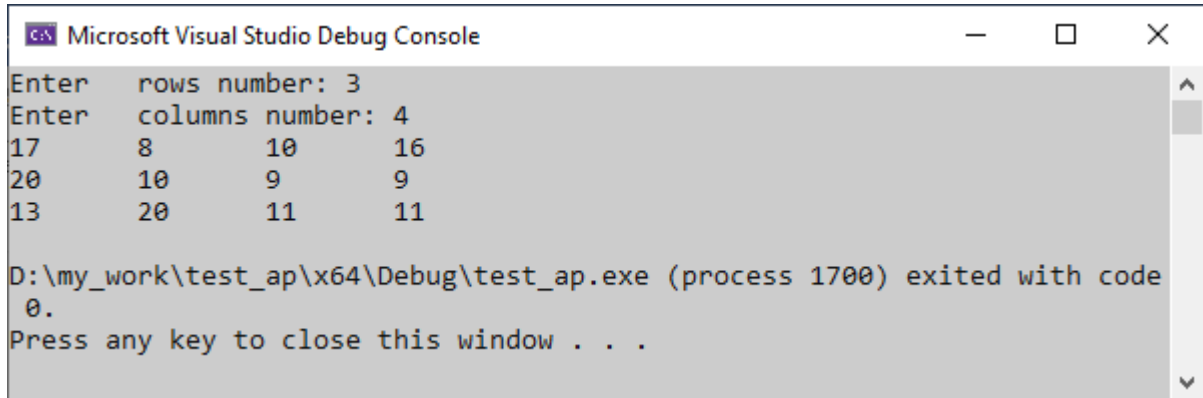
void main() {
    const int Low = 6, High = 20;
    int rowN, // кількість рядків
        colN; // кількість стовпчиків
    cout << "Enter rows number: ";
    cin >> rowN;
    cout << "Enter columns number: ";
    cin >> colN;
    // оголошення та розміщення в пам'яті допоміжного масива
    int** m = new int* [rowN];
    // оголошення та розміщення в пам'яті основного масива
    for (int i = 0; i < rowN; i++)
        m[i] = new int[colN];

    // ініціалізація елементів та виведення масива на екран
    for(int i = 0; i < rowN; i++) {
        for (int j = 0; j < colN; j++) {
            // ініціалізація
            m[i][j] = Low + rand() % (High - Low + 1);

            // індексний спосіб
            cout << m[i][j]<<"\t" ; // виведення
            // вказівниковий спосіб
            //cout << setw(5) << (*(m + i) + j);
        }
        cout << endl;
    }

    // звільнення пам'яті, виділеної під основний масив
    for (int i = 0; i < rowN; i++)
        delete[] m[i];

    // звільнення пам'яті, виділеної під допоміжний масив
    delete[] m;
}
```



```

Microsoft Visual Studio Debug Console

Enter    rows number: 3
Enter    columns number: 4
17       8       10      16
20       10      9       9
13       20      11      11

D:\my_work\test_ap\x64\Debug\test_ap.exe (process 1700) exited with code
0.
Press any key to close this window . . .
    
```

Опрацювання динамічних масивів у функціях

Динамічний масив, як і звичайний, можна передавати у функцію за допомогою *вказівника на масив, масива визначеного розміру або масива невизначеного розміру*. Найгнучкіший спосіб передачі масива в функцію є *вказівник на масив*. При передачі масива в функцію насправді передається не масив, а адреса його першого елемента. Тобто масив завжди передається за адресою, а не за значенням. При цьому інформація про кількість елементів масива втрачається, тому кількість елементів кожної розмірності потрібно передавати за допомогою окремих параметрів.

Всередині функції багатовимірний масив сприймається як одновимірний, а його індекс формується з набору індексів багатовимірного масива.

Якщо потрібно виключити можливість зміни масива в функції, то його необхідно передавати в неї як константу.

Синтаксис заголовку функції, що отримує динамічний двовимірний масив-константу як одновимірний, має такий вигляд:

```

<тип_даних> <ім'я_функції> (const <тип_даних>*, [const]
                             <тип_даних>, ...);
    
```

де `[]` позначають необов'язковий елемент конструкції.

Тоді в програмі ця функція *повинна викликатись* так:

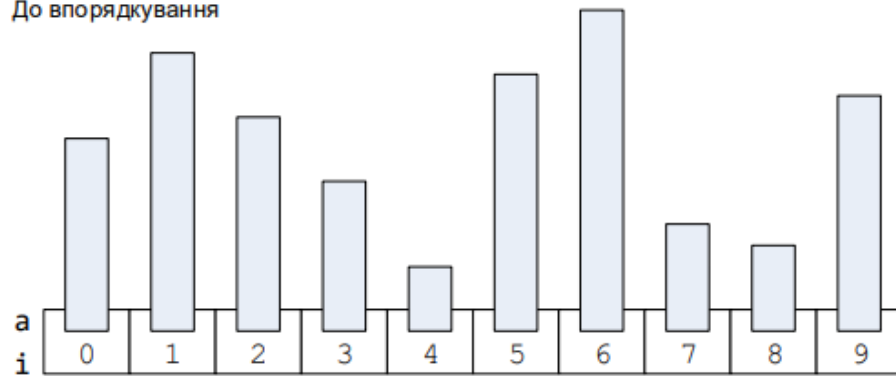
```

<ім'я_функції> (<ім'я_масива>, <ім'я_параметра>, ...);
    
```

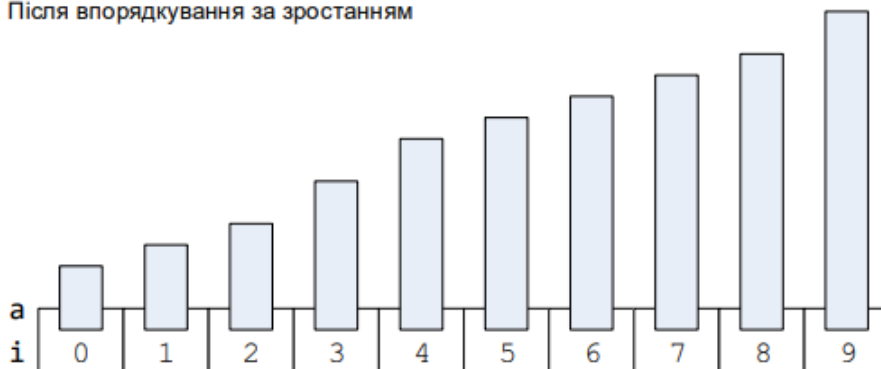
Сортування масивів

Сортування або впорядкування масиву – це перестановка місцями його елементів так, щоб після перестановки елементи розміщувалися в певному порядку, наприклад – зростали або спадали. Наступні рисунки показують масив до і після впорядкування за зростанням:

До впорядкування



Після впорядкування за зростанням



Всі способи впорядкування масивів поділяють на дві категорії:

- *методи внутрішнього сортування* – не використовують допоміжних масивів, всі дії відбуваються в одному і тому самому масиві. Ці методи зазвичай використовуються для масивів, які повністю розміщуються в оперативній пам'яті.

- *методи зовнішнього сортування* – використовують допоміжні масиви, зазвичай використовуються для сортування файлів (наборів даних, що розміщуються на зовнішніх носіях).

Далі будемо розглядати методи внутрішнього сортування, причому обмежимося випадком впорядкування масиву за зростанням. Для реалізації впорядкування за спаданням достатньо буде поміняти умову впорядкування.

Всі методи внутрішнього сортування можна поділити *на два види*:

- *елементарні (прямі) методи* – прості для розуміння але малоефективні;
- *вдосконалені методи*.

Є три основні способи прямого впорядкування масиву:

- 1) метод вибору;
- 2) метод вставки (включення);
- 3) метод обміну.

До вдосконалених методів належать: метод Хоара або швидке сортування (qsort, quickSort); метод Шелла або сортування включенням при спадному прирості;

пірамідальне сортування або сортування за допомогою дерева; сортування методом злиття.

Метод вибору

Розглянемо на прикладі наступного масиву цілих чисел:

a	9	3	7	5	8
i	0	1	2	3	4

На кожному кроці методу вибору масив вважається розділеним на дві частини: ліву, вже впорядковану, та праву, ще не впорядковану. На початку ліва частина має нульову довжину:

a	9	3	7	5	8
i	0	1	2	3	4

Починається перша ітерація. Вибираємо найменший елемент правої (не впорядкованої) частини \min та запам'ятовуємо його індекс imin (ці значення будуть потрібні для наступного кроку):

a	9	3	7	5	8
i	0	1	2	3	4
min	3				
imin	1				

Міняємо місцями цей мінімальний елемент з першим елементом правої частини, при цьому довжина впорядкованої частини масиву збільшується (а не впорядкованої – зменшується) на одиницю:

a	3	9	7	5	8
i	0	1	2	3	4

Тепер починається друга ітерація. Невпорядкована частина масиву складається з елементів від другого до останнього (з індексами від 1 до 4 для нашого прикладу). Знову шукаємо найменший елемент правої частини \min та запам'ятовуємо його індекс imin :

a	3	9	7	5	8
i	0	1	2	3	4
min	5				
imin	3				

В правій частині міняємо місцями щойно знайдений мінімальний елемент з її першим елементом (від початку масиву – другим за порядком елементом), при цьому знову довжина лівої частини масиву збільшується (а правої – зменшується) на одиницю:

a	3	5	7	9	8
i	0	1	2	3	4

Процес продовжується до тих пір, поки у невпорядкованій частині масиву не залишиться один елемент (бо його тоді не потрібно буде міняти місцями із ним же).

Алгоритм методу вибору складається з наступних дій:

1. Вважати весь масив – невпорядкованою частиною (індекс, з якого починається невпорядкована частина, дорівнює 0).

2. Поки невпорядкована частина масиву містить більше одного елемента, виконувати дії:

2.1. Знайти мінімальний елемент невпорядкованої частини масиву і запам'ятати його індекс. Для цього:

2.1.1. Вибрати (запам'ятати) перший елемент невпорядкованої частини масиву, вважати його мінімальним.

2.1.2. Запам'ятати індекс цього елемента.

2.1.3. Для елементів від наступного після вибраного (другого елемента невпорядкованої частини масиву) і до останнього повторювати дії:

2.1.3.1. Порівняти вибраний елемент з поточним.

2.1.3.2. Якщо вибраний елемент більший поточного, запам'ятати поточний елемент як мінімальний, а його індекс – як індекс мінімального елемента.

2.2. Поміняти місцями мінімальний та вибраний на кроці 2.1.1 елементи масиву.

2.3. Перемістити початок невпорядкованої частини масиву на одну позицію праворуч (збільшити на 1 індекс, з якого починається невпорядкована частина).

Функція для сортування:

```
void Sort(int* a, const int size) // метод вибору
{
    for (int i = 0; i < size - 1; i++) // індекс початку невпорядкованої частини
    {
        int min = a[i]; // пошук мінімального елемента
        int imin = i; // невпорядкованої частини
        for (int j = i + 1; j < size; j++)
            if (min > a[j])
            {
                min = a[j];
                imin = j;
            }
    }
}
```

```

    a[imin] = a[i]; // обмін місцями мінімального та першого
    a[i] = min; // елементів невідсортованої частини
}
}
Виклик функції
int main()
{
    const int n = 5;
    int a[n] = { 9, 3, 7, 5, 8 };
    Sort(a, n);
    return 0;
}

```

Метод обміну (бульбашки)

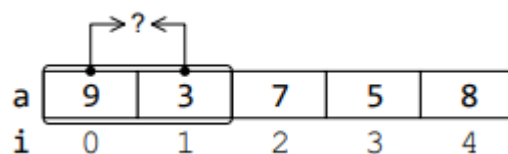
Розглянемо той самий масив цілих чисел:

a	9	3	7	5	8
i	0	1	2	3	4

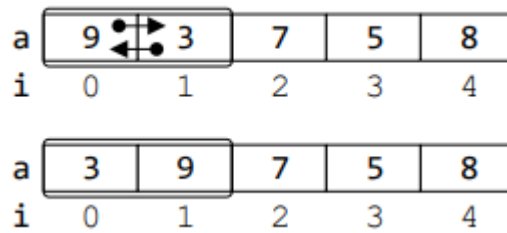
Цей метод оснований на порівнянні двох сусідніх елементів масиву. Якщо їх значення порушують умову впорядкування, то ці елементи міняються місцями. Знову розглянемо випадок впорядкування за зростанням. Якщо порівняння пар сусідніх елементів масиву проводити від початку (від першої пари) до кінця (до останньої пари), то після завершення всіх операцій послідовного порівняння та обміну місцями найбільший елемент буде встановлений в останню позицію масиву.

Тоді впорядкованою буде права частина масиву – після першого проходження її довжина стане рівною одиниці. Якщо ж порівняння пар сусідніх елементів проводити від останньої пари до першої, то це дозволить встановити найменший елемент у першу позицію масиву. При такому способі проходження масиву впорядкованою буде його ліва частина.

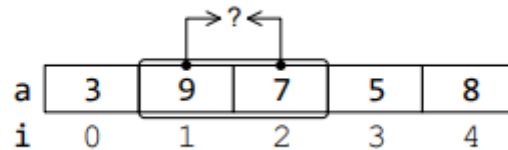
Оскільки і при першому (від початку до кінця) і при другому (в зворотному порядку) способі проходження масиву він стане впорядкованим – немає значення, який порядок проходження обрати. Для пояснення методу розглянемо впорядкування при проходженні масиву в прямому порядку. Перша ітерація. Впорядкована частина масиву має нульову довжину. Порівнюємо першу пару сусідніх елементів: елементи 9 (з індексом 0) та 3 (з індексом 1):



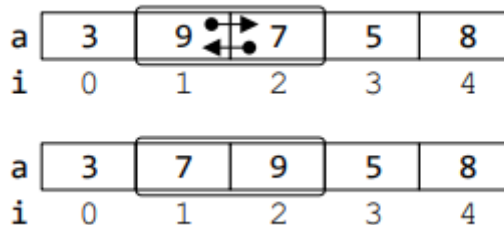
Виявляється, що $a[0] > a[1]$ – порушено умову впорядкування. Тому елементи $a[0]$ та $a[1]$ міняємо місцями:



Тепер порівнюємо другу пару – елементи $a[1]$ (9) та $a[2]$ (7)

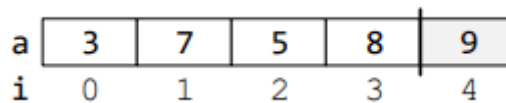


Умову впорядкування знову порушено, тому ці елементи міняємо місцями:



Аналогічно все відбувається для третьої ($a[2]$ та $a[3]$) і четвертої ($a[3]$ та $a[4]$) пар сусідніх елементів.

Після завершення першої ітерації (першого проходження масиву) елемент 9 переміститься в останню позицію масиву і стане першим елементом правої (впорядкованої) частини масиву:

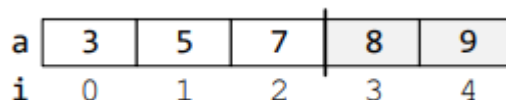


Друга ітерація

Тепер послідовно порівнюються пари елементів, які належать лівій (невпорядкованій) частині масиву:

$a[0]$ та $a[1]$, $a[1]$ та $a[2]$, $a[2]$ та $a[3]$.

Після другого проходження найбільший елемент лівої частини переміститься в передостанню позицію і стане новим елементом правої частини:



Процес продовжується до тих пір, поки в лівій частині не залишиться один елемент (бо тоді його не потрібно буде міняти місцями із ним же!).

Цей метод називають методом бульбашки тому, що подібно до бульбашок спочатку найбільший елемент «випливає на поверхню» – переміщується в останню позицію. Потім «випливає» наступний за величиною елемент – переміщується у передостанню позицію; і так далі – поки не будуть пройдені всі ітерації (причому в кожній наступній ітерації переглядається на один елемент менше)

Як видно із останнього рисунку, після другої ітерації масив вже став впорядкованим, проте ітерації ще будуть продовжуватися. Це – недолік прямого методу бульбашки.

Алгоритм методу обміну (бульбашки) містить такі дії:

1. Встановити лічильник ітерацій рівним одиниці.
2. Поки значення лічильника ітерацій менше кількості елементів масиву, виконувати дії:
 - 2.1. Встановити номер поточного елемента рівним нулю.
 - 2.2. Поки номер поточного елемента менший різниці кількості елементів масиву та значення лічильника ітерацій, виконувати дії:
 - 2.2.1. Якщо поточний елемент більший наступного – поміняти ці елементи місцями.
 - 2.2.2. Збільшити на одиницю номер поточного елемента (перейти до наступного елемента).
 - 2.3. Збільшити на одиницю лічильник ітерацій (перейти до наступної ітерації).

Функція, що реалізує впорядкування методом обміну :

```
void SortB(int* a, const int size) // метод обміну (бульбашки)
{
    for (int i = 1; i < size; i++) // лічильник ітерацій
        for (int j = 0; j < size - i; j++) // номер поточного елемента
            if (a[j] > a[j + 1]) // якщо порушено умову впорядкування
                { // - обмін елементів місцями
                    int tmp = a[j];
                    a[j] = a[j + 1];
                    a[j + 1] = tmp;
                }
}
```

Виклик функції

```
int main()
{
    const int n = 5;
    int a[n] = { 9, 3, 7, 5, 8 };
    SortB(a, n);
    return 0;
}
```

Метод вставки

Розглянемо той самий масив цілих чисел:

a	9	3	7	5	8
i	0	1	2	3	4

Як і для методу вибору, на кожному кроці методу вставки масив вважається розділеним на дві частини: ліву, вже впорядковану, та праву, ще не впорядковану. На початку ліва частина містить лише один елемент:

a	9	3	7	5	8
i	0	1	2	3	4

Перша ітерація. Беремо перший елемент невідсортованої (правої) частини масиву і вставляємо його у ліву частину таким чином, щоб ліва частина зберегла впорядкованість. Для цього потрібно буде знайти позицію вставки:

a	9	3	7	5	8
i	0	1	2	3	4

При вставці впорядкована частина масиву збільшується (а не впорядкована – зменшується) на один елемент. Сама вставка містить зсув частини впорядкованих елементів на одну позицію праворуч (в нашому прикладі – елемент 9 зсувається праворуч для того, щоб звільнити місце для елемента 3). Це приводить до того, що вказаний зсув «зітре» перший елемент невідсортованої частини (елемент 3) – замінить його значення останнім елементом впорядкованої частини. Після вставки отримаємо:

a	3	9	7	5	8
i	0	1	2	3	4

Невідсортована частина масиву починається з третього елемента (з індексом 2).

Друга ітерація. Знову вибираємо (і запам'ятовуємо) перший елемент невідсортованої частини масиву. Знову виконуємо операції: 1) пошук позиції для вставки вибраного елемента у ліву частину масиву:

a	3	9	7	5	8
i	0	1	2	3	4

2) зсув праворуч на одну позицію тих елементів лівої частини масиву, які розміщені після позиції вставки:

a	3	7	9	5	8
i	0	1	2	3	4

3)) вставка (присвоєння) вибраного елемента у знайдену позицію, довжина впорядкованої частини масиву збільшується на одиницю:

a	3	7	9	5	8
i	0	1	2	3	4

Тепер невідсортована частина масиву починається з елемента, який має індекс 3. Третя ітерація. Запам'ятовуємо перший елемент невідсортованої частини масиву (елемент 5) і шукаємо для нього позицію вставки (це – позиція з індексом 1):

Зсуваємо праворуч на одну позицію елементи 7 та 9, які містяться в позиціях з індексами 1 та 2:

a	3	7	9	5	8
i	0	1	2	3	4

Зрозуміло, що зсув праворуч виконується справа наліво: спочатку для крайнього правого з діапазону зсуву елемента (9), потім – для наступного ліворуч елемента (7):

a	3	7	9	5	8
i	0	1	2	3	4

a	3	7	7	9	8
i	0	1	2	3	4

Бо якщо зсув виконувати зліва направо: спочатку крайній ліворуч з діапазону зсуву елемент (7), потім – наступний праворуч елемент, і т.д., – то це приведе до того, що елемент 7 буде скопійований на весь діапазон зсуву:

a	3	7	7	9	8
i	0	1	2	3	4

a	3	7	7	5	8
i	0	1	2	3	4

Після вставки елемента 5 в позицію з індексом 1 довжина впорядкованої частини масиву знову збільшується на одиницю:

a	3	5	7	9	8
i	0	1	2	3	4

Процес продовжується до тих пір, поки права (невпорядкована) частина масиву містить хоча би один елемент.

Алгоритм методу вставки:

1. Вважати, що перший елемент масиву належить до впорядкованої частини, а індекс, з якого починається невлпорядкована частина, дорівнює 1

2. Поки невлпорядкована частина масиву містить хоча би один елемент, виконувати дії:

2.1. Зберегти перший елемент невлпорядкованої частини масиву в допоміжній змінній.

2.2. Визначити позицію вставки збереженого елемента у впорядковану частину масиву. Для цього:

2.2.1. Вважати перший елемент масиву поточним.

2.2.2. Поки збережений елемент більший поточного, збільшувати індекс поточного елемента.

2.3. Зсунути на одну позицію праворуч елементи відсортованої частини, які починаються з позиції вставки. 2.4. Вставити збережений на кроці 2.1 елемент у знайдену позицію вставки.

2.5. Збільшити на 1 індекс початку невлпорядкованої частини.

Функція, що реалізує впорядкування:

```
void Sort(int* a, const int size) // метод вставки
{
    for (int i = 1; i < size; i++) // індекс початку невлпорядкованої частини
    {
        int tmp = a[i]; // вибрали елемент для вставки
        int j = 0; // пошук позиції вставки
        while (tmp > a[j])
            j++;
        for (int k = i - 1; k >= j; k--) // зсув
            a[k + 1] = a[k];
        a[j] = tmp; // вставка
    }
}
```

Виклик функції

```
int main()
{
    const int n = 5;
    int a[n] = { 9, 3, 7, 5, 8 };
    Sort(a, n);
    return 0;
}
```