Лекція 16

Вказівники на масиви

Тип даних вказівник

Найбільш цікаве, що є в мовах С і С ++ - 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.

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

Покажчик завжди є змінною величиною. Покажчики в 16-розрядної операційної системи MS DOS так само були 16-розрядними. В даний час зазвичай використовуються 32-розрядні операційні системи (Windows, Linux і ін.), В яких покажчики займають 4 байта (32-розрядний ціле без знака). І хоча це по внутрішньому поданню в точності відповідає типу unsigned int, але покажчик і змінна типу unsigned int – це дві великі різниці.

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

Формальний опис покажчика таке:

Тип_данных * Ім'я_покажчика;

де

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

Дамо опис покажчика для роботи з даними типу double:

```
double *u;
```

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

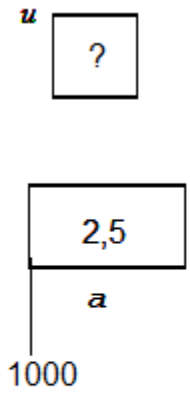
На який об'єкт налаштований цей покажчик? В даний момент ні на який. Значення покажчика ***u*** в момент виділення пам'яті не визначено.

Нехай є змінна a типу double:

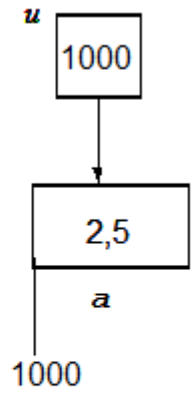
```
double a = 2.5;
```

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

```
u = &a;
```



а)до настройки указателя



б)после настройки указателя

Тут знак **&** означає обчислення адреси, тобто ми обчислюємо адреса 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 а.

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

```
*u = 11.3;
cout << a << endl;
```

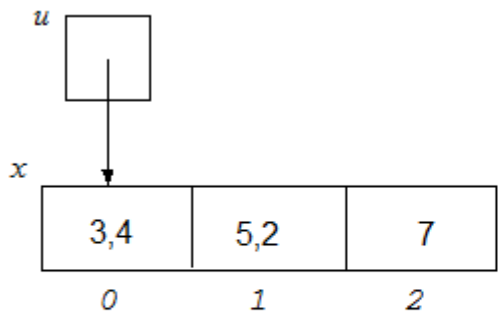
На екран буде виведено число 11.3, хоча раніше змінна ***a*** мала значення 2.5. Отже, користуючись покажчиком, ми дійсно змінили вміст змінної ***a***.

Операції для роботи з покажчиками

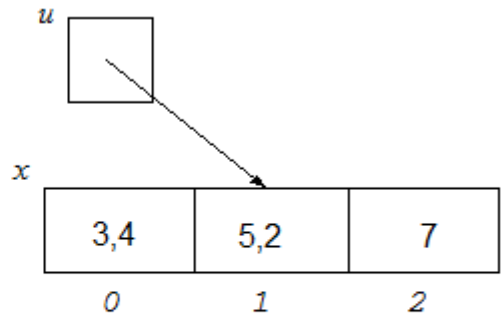
Хоча про всяк покажчик по внутрішньому поданню – це беззнакове ціле, але за характером операцій він помітно відрізняється від будь-яких цілих типів. Розглянемо всі операції, допустимі для покажчиків.

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. ***Разіменування*** (*) – 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. ***Автозбільшення*** (++).

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

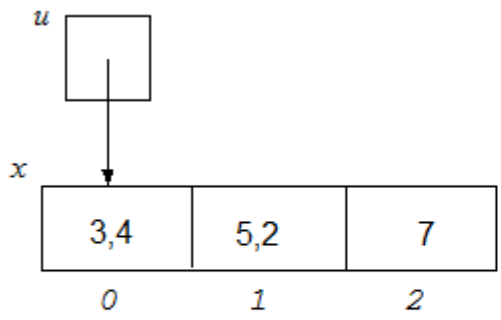
```
int i = 5;
i++; // Теперь i равно 6.
```

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**:

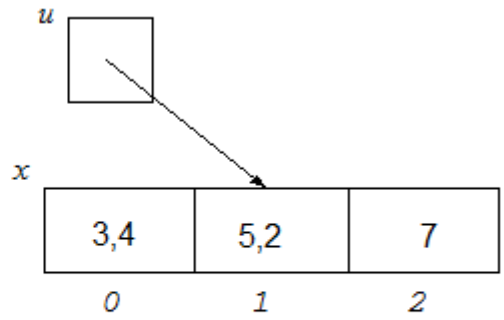
```
double x[3] = {3.4, 5.2, 7};
double *u;
```

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

Варіант №6 – «посилимо» ситуацію.

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

Звичайно, в програмах, призначених для загального огляду, треба використовувати тільки «правильні» конструкції виду **x[i]** або ***(x + i)**, щоб ні кого 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 убедится, 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 поводиться як змінна величина, а масив – як константа. Тому масив і не є повноцінним покажчиком. ***Масив*** – це ***константних покажчик***.

Динамічні вектори

За допомогою покажчиків 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 покажчик.

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.

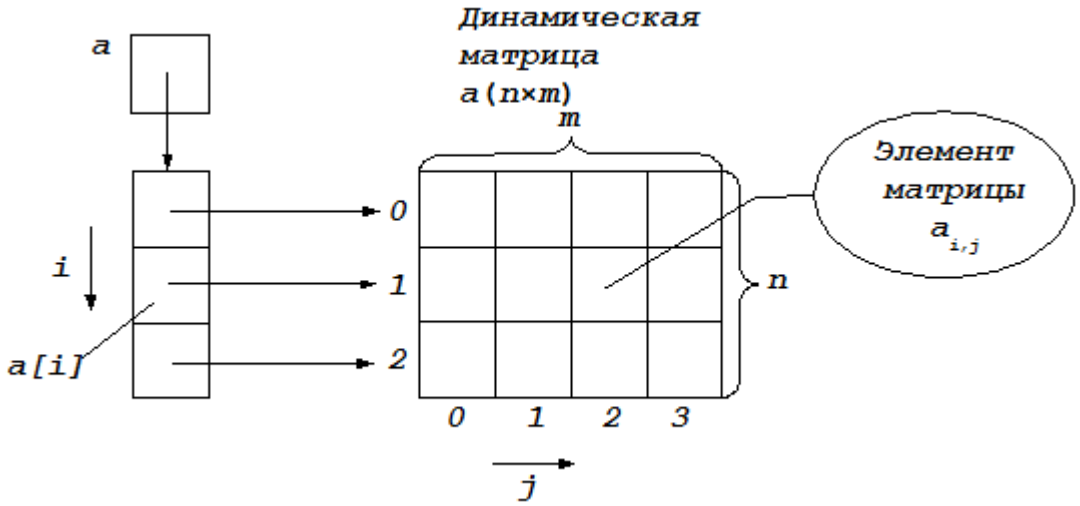
г) Як тільки об'єкт став непотрібним – звільняємо займану ним пам'ять:

```
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 зберігається дані, тобто елементи матриці ***a [i] [j]***

Порядок роботи з динамічної матрицею розглянемо на прикладі.

Приклад. У прямокутної 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 << "Matriza 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.