Лекція 5. Вказівники

Оголошення вказівника, ініціалізація, використання.
Створення, використання та знищення динамічних змінних простих типів.
Зв'язок вказівників і масивів. Арифметика вказівників.
Створення та використання динамічних масивів.
Робота з двохвимірними масивами.
Побудова лінійного списку за допомогою вказівників.

Вказівники – це змінні, значеннями яких є адреси розташування в пам'яті інших значень. Вказівники часто використовують для передавання аргументів у функцію, для побудови структур і класів, вони мають вирішальне значення для динамічного створення простих змінних і структур даних – для керування купою[1] на етапі виконання програми.

Оголошення вказівника має вигляд

<базовий тип> * <ім'я>

Тут *ім'я* – це ім'я змінної-вказівника, *зірочка* – оператор оголошення вказівника, *базовий тип* – тип значення, адресу якого зберігатиме вказівник. Усі вказівники мають однакову природу: вони є адресами пам'яті, всі мають однаковий розмір – але значення, на які вони вказують, можуть бути різними, тому базовий тип потрібен для правильної інтерпретації адреси.

Оголошення можна записувати у різній манері:

```
int * ptr; // неініціалізований вказівник ptr на ціле
           // нейтральна манера оголошення: просто використали *
int* ptr;  // ptr – змінна типу «вказівник на ціле»
int *ptr;  // *ptr – розіменований вказівник – є змінною цілого типу
```

В одній інструкції можна оголосити декілька змінних, але треба пам'ятати, що оператори оголошення стосуються тільки того імені, біля якого записані:

```
int * a, b; // a - вказівник на ціле, b - ціле
int *a, *b; // обидві змінні є вказівниками
```

Ініціалізацію вказівника можна виконати (як і для всякої змінної) одразу в момент оголошення або пізніше в програмі. Для цього використовують оператор «&» взяття адреси, оператор «*new*» створення змінної. Вказівник можна задати порожнім, просто присвоївши йому значення 0 (нуль), та в сучасному стандарті рекомендовано використовувати *nullptr*:

```
int k = 10;        // звичайна змінна цілого типу
int * altK = &k;   // altK вказує на k,
                   // тепер *altK – альтернативний шлях до значення k
double * aReal;    // невизначений вказівник на дійсне
double * ptr = 0;  // порожній вказівник
         aReal = new double; // створили динамічну змінну *aReal
void * aPointer = nullptr;   // універсальний вказівник, порожній
```

Однотипні вказівники можна використовувати в операторі присвоєння: *ptr = aReal*, тобто значення одного присвоїти іншому, але присвоєння різнотипних: *ptr = altK* – є помилкою. Замість числового значення 0 для ініціалізації вказівника прийнято використовувати спеціальне слово *nullptr*. Вказівник універсального типу сумісний за присвоєнням з вказівниками довільних типів: *aPointer = altK* або *aPointer = aReal* але не навпаки.

---

[1] *Купа* (heap) – динамічна область пам'яті програми, розподілом якої керує програміст на етапі виконання програми (за допомогою операторів *new, new[], delete, delete[]*).

Значення вказівників можна *виводити на друк*:

```
cout << "altK = " << altK << "    aReal = " << aReal << "    ptr = " << ptr << '\n';
```

У результаті отримаємо:

```
altK = 0012FF60    aReal = 00385E40    ptr = 00000000
```

Це цілі значення, записані в шістнадцятковій системі, проте жоден вказівник не є цілим числом. Інструкція *altK = 0x0012FF60* була б помилкою. Присвоєння такого роду можна виконати хіба що за допомогою «жорсткого» приведення типу *altK = (int*)0x0012FF60*, але на практиці слід уникати таких потенційно небезпечних дій.

Створення динамічних змінних.

Просту змінну створюють за допомогою оператора *new*:

<ім'я вказівника на тип> = new <тип>;

Аргументом оператора new є базовий тип вказівника. Таким чином у попередньому прикладі було створено динамічну змінну **aReal*. Часто оголошення вказівника поєднують зі створенням змінної:

```
int * dynInt = new int;
double * dynDbl = new double;
```

Щоб отримати доступ до новоствореної змінної, використовують операцію розіменування вказівника:

```
*dynInt = 5; *dynDbl = sin(0.2 * *dynInt + 0.15);
cout << "k=" << *dynInt << "    S=" << *dynDbl << '\n';
```

Після використання динамічну змінну можна знищити оператором *delete*:

delete <ім'я вказівника>;

Для нашого прикладу це матиме вигляд

```
delete dynInt; delete dynDbl;
```

Використання операторів *new* і *delete* має бути збалансованим. Не можна двічі знищувати змінну, забута в пам'яті непотрібна змінна означатиме зайві витрати пам'яті. Помилкою є спроба знищити звичайну змінну (не динамічну), навіть якщо є вказівник з її адресою. Оператор *delete* з порожнім вказівником в якості аргумента не спричиняє ніяких дій.

Масив створюють оператором *new[]*. Пояснимо це на прикладі створення одновимірного масиву дійсних чисел заданого розміру.

```
size_t n;                // кількість елементів масиву – натуральне число
cout << "Enter the array size: "; cin >> n;
double * P = new double[n];       // оголосили вказівник і створили масив
```

Приклад демонструє звичайну практику виділення пам'яті для масиву, розмір якого стає відомим тільки під час виконання програми. Звільняють таку пам'ять оператором *delete[]*:

```
delete [] P;
```

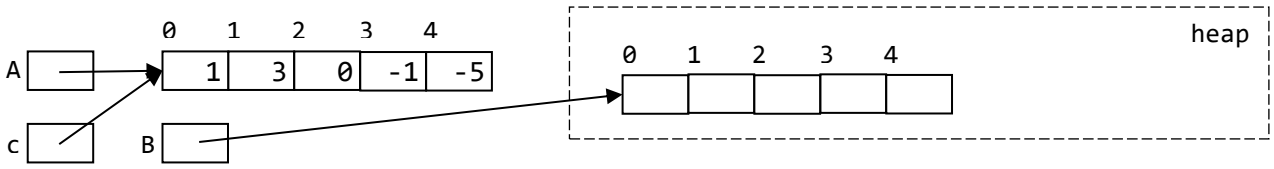
Як використати створений динамічний масив? Виявляється, що ніяких особливих засобів чи підходів не потрібно: динамічним масивом можна користуватися точно так само,

як і звичайним. Секрет такої чудової можливості криється у низькорівневій організації масивів. Адже ім'я масиву насправді є іменем комірки пам'яті, куди записано адресу першого елемента масиву. Так само є і з вказівником *P* з попереднього прикладу: після виконання оператора *new* він містить адресу виділеної для масиву ділянки пам'яті, тобто, першого елемента масиву.

Розглянемо приклад, що пояснює **зв'язок між масивом і вказівником**

```
int A[5] = {1,3,0,-1,-5};         // звичайним чином визначили масив
int * B = new int[5];  // створили динамічний масив такого ж розміру
int * c = A; // спорідненість: вказівник на ціле – майже масив цілих
```

Структури, що утворяться в пам'яті комп'ютера внаслідок виконання цих інструкцій, схематично можна зобразити так:



Завдяки такому влаштуванню масивів цілком правильними будуть наступні звертання:

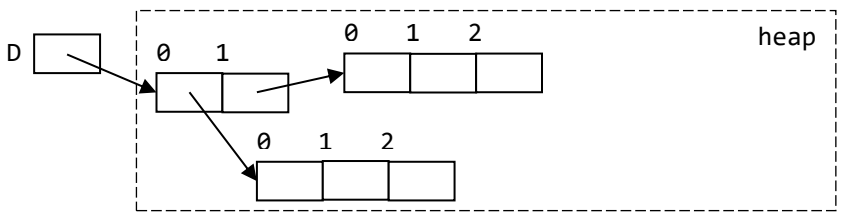
```
A[0], c[0], *c, *A – перший елемент масиву A
for (int i=0; i<5; ++i) cin >> B[i]; - ввести всі елементи масиву B
c, &A[0] – адреса першого елемента масиву A
B, &B[0] – адреса першого елемента маисву B
```

Оператор розіменування (*) та оператор індексування ([]) діють схожим чином. Вказівник і масив теж схожі поняття але не тотожні. Наприклад, розмір вказівника (sizeof) – завжди чотири байти, розмір масиву – кількість елементів у ньому, помножена на розмір елемента. Зрештою, вказівник не мусить вказувати на елементи масиву.

Двохвимірний масив можна створити різними способами. Наприклад, так

```
int n, m; cout << "Enter dimensions of the matrix: "; cin >> n >> m;
double * * D = new double* [n]; // масив вказівників на рядки матриці
for (int i=0; i<n; ++i) D[i] = new double[m]; // створили кожен рядок
```

Тепер у нашому розпорядженні є $n \times m$ матриця *D*. Звертання до її елементів можна виконувати як звичайно: *D*[*i*][*j*]. Все майже як у «справжньої» матриці. Так і буде, поки не матиме значення внутрішнє влаштування *D*. Воно відрізняється від того, яке має матриця оголошена за допомогою операторів [] [] – та займає неперервну ділянку пам'яті, а ми кожен рядок *D* створювали окремо. Структуру пам'яті такої динамічної матриці зображено нижче (для випадку $n = 2$, $m = 3$).



Після використання матриці її пам'ять можна звільнити стількома ж операторами *delete*, скільки було виконано *new*.

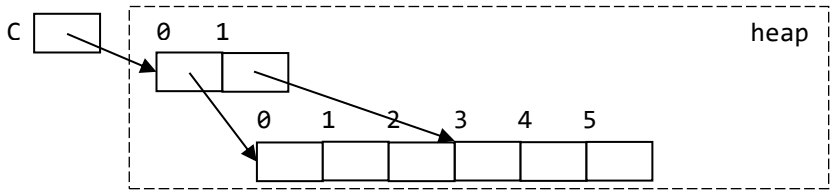
```
for (int i=0; i<n; ++i) delete[] D[i]; // вилучили кожен рядок
delete[] D;             // вилучили масив вказівників на рядки
```

Зверніть увагу на порядок звільнення пам'яті: він обернений до порядку створення структури матриці. Якщо оператор *delete*[] *D* виконати першим, то станеться помилка, і ми не зможемо звільнити пам'ять рядків матриці, оскільки вказівники на них буде втрачено.

Щоб не виникало тонких непорозумінь, створюйте матриці так:

```
int n, m; cout << "Enter dimensions of the matrix: "; cin >> n >> m;
double * * C = new double* [n];     // масив вказівників на рядки матриці
C[0] = new double[n*m];             // вся потрібна пам'ять «одним шматком»
for (int i=1; i<n; ++i) C[i]=C[i-1]+m;   // значення вказівників на рядки
```

Тоді структура пам'яті (також для випадку $n = 2$, $m = 3$) буде іншою:



Така матриця не підведе! Вона виготовлена «з дотриманням усіх вимог технології», її структура близька до структури статичної матриці. Але зверніть увагу на останній рядок коду. З ким там виконують арифметичні дії?

Звільнення динамічної пам'яті можна виконати двома операторами:

```
delete[] C[0]; // вилучили пам'ять елементів матриці
delete[] C;    // вилучили масив вказівників на рядки
```

Арифметика вказівників.

Будь-який вказівник можна збільшити чи зменшити на ціле, при цьому одиницями виміру є розмір базового типу вказівника:

```
double X[10]; double * p = X; // p вказує на перший елемент X
p++;             // означає збільшити p на sizeof(double),
                 // тепер p вказує на другий елемент X
p += 5;          // p вказує на X[6]
p--;             // а тепер – на X[5]
```

Для вказівників визначено оператори інкременту, декременту, + і –: якщо *T * p;* то (*p+k*), (*p–k*) означає зміну значення *p* на *k*sizeof*(*T*); ++*p*, *p*++, – –*p*, *p*– – означає зміну значення *p* «на один» – на *sizeof*(*T*). Не визначено додавання двох вказівників. Їх можна тільки відняти:

```
double X[10]; double * p = X;
double * q; q = &X[7];
int k = q-p; // k – це кількість елементів масиву між двома вказівниками
```

У сенсі арифметики вказівників *B*[*i*] означає *(*B*+*i*). Цими особливостями і пояснюється цікавий рядок коду нашого прикладу: *C* – це вказівник на вказівник на ціле, *C*[*i*] – вказівник на ціле, тому збільшення його на *m* означає відшукання адреси початку наступного рядка матриці.

Продовжимо приклади. Обчислити кількість входжень літери «о» в задану фразу:

```
char w[] = "How do you do?";
char * p = w;
int k = 0;
int n = 0;
for (int i = 0; w[i] != '\0'; ++i)
{
   if (w[i] == 'o') ++k;
}
```

```
while (*p)
{
   if (*p == 'o') ++n;
   ++p;
}
cout << "Occurrences of 'o' is " << k << " or " << n << '\n';
```

Обчислення виконано двома різними способами: за допомогою індексування елементів масиву та за допомогою вказівника. Немає причин для того, щоб котрийсь зі способів був ефективнішим. Хороший компілятор може згенерувати однаковий код для обох циклів.

Ще один цікавий приклад. Наступний фрагмент мав би копіювати один рядок в інший. Поясніть, як він це робить (*p* і *q* – вказівники на *char*):

```
while (*p++ = *q++) ;
```

Зауважимо, що інкремент має вищий пріоритет, ніж розіменування, тому вираз **q*++ означає «спочатку збільшити (вказівник), потім – розіменувати». Для того, щоб збільшити змінну, на яку вказує вказівник, використовуйте вираз (**q*)++.

Ще раз про оголошення масивів і вказівників:

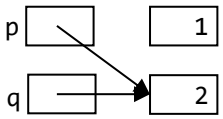
```
int* ap[5];   // масив п'яти вказівників на int
int (*pa)[5]; // вказівник на масив п'яти int'ів
```

Приклад нагадує, що суфіксний оператор оголошення має вищий пріоритет ніж префіксний.

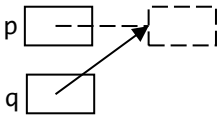
Помилки

Використання динамічних змінних потребує від програміста особливої уважності. Іноді синтаксично правильні, однак некоректні з погляду логіки програми дії з вказівниками можуть призвести до виникнення помилок, які буває важко виявити.
Розглянемо дві характерні помилкові ситуації, що можуть виникати в процесі некоректної роботи з динамічними змінними за допомогою вказівників.

```
int *p, *q;
  // фрагмент 1
p = new int; *p = 1; // створили першу змінну
q = new int; *q = 2; // і другу
   p = q;            // першу втратили !
      .......
  // фрагмент 2
p = new int; *p = 1; // створили динамічну змінну
q = p;               // до неї є два шляхи
delete p;            // змінну знищили, а що таке *q ?
      .......
```



фрагмент 1



фрагмент 2

Результати виконання цих фрагментів схематично зображено на рисунку праворуч.

У першому випадку відбулася втрата адреси першої динамічної змінної. Відведена для неї ділянка пам’яті стала недоступною. Вона залишатиметься зайнятою аж до завершення виконання програми. "Засмічення" пам’яті такими недоступними динамічними змінними може призвести до завчасного вичерпання всієї купи і неможливості подальшого правильного виконання програми.

У другому випадку *q* вказує на неіснуючу змінну і робота з * *q* може призвести до несподіваних результатів, якщо ділянка пам’яті, на яку вказує *q*, буде використана для розміщення нових динамічних змінних. У такій ситуації поведінка програми залежатиме не від її семантики, а від системних особливостей функціонування, тобто від способу реалізації мови C++ у конкретній системі програмування.

Зв'язні структури

На завершення покажемо, як за допомогою вказівників ***побудувати лінійний однозв'язний список***. Відомо, що список складається з ланок, кожна з яких має інформаційну частину та поле зв'язку з наступною ланкою. Ланку зручно моделювати за допомогою відповідної структури. Для визначеності припустимо, що наш список зберігатиме цілі числа. Наведена нижче програма створює список чисел від 0 до заданого *n*, друкує його та знищує. Ланки списку – динамічно створені структури. Для доступу до полів динамічної структури замість **p.поле* зручно використати оператор опосередкованого вибору «–>», тобто *p–>поле*, де *p* – вказівник на структуру.

```
#include <iostream>
using std::cin; using std::cout;

struct section // ланка списку
{
	int elem;
	section* link;
};
int main()
{
	// List – голова списку, b – робоча змінна
	section *List;
	section *b;
    // створення списку List
	int n;
	cout << "Enter n: "; cin >> n;
	List = new section; List->elem = 0;  // створили першу ланку
	b = List;
	for (int i = 1; i <= n; i++)
	{
		b->link = new section;          // та всі інші ланки
		b = b->link;          // для доступу до полів структури
		b->elem = i;          // всюди використовуємо оператор ->
	}
	b->link = nullptr;        // список закінчується порожнім вказівником
    // виведення елементів списку List
	cout << "List: ";
	b = List;    // вказівник b - для перебирання ланок
	while (b)                     // поки є продовження
	{
		cout << '\t' << b->elem; // друкуємо число
		b = b->link;              // беремо наступну ланку
	}
	cout << '\n';
	// знищення ланок списку List
	while (List)
	{
		b = List;
		List = List->link;
		delete b;
	}
	cout << "Done\n";
	system("pause");
	return 0;
}
```

Отримані результати

```
Enter n: 5
List:   0       1       2       3       4       5
Done
```

Література

1. Бублик В.В. Об'єктно-орієнтоване програмування мовою С++.
2. Стивен Прата Язык программирования С++.
3. Дудзяний І.М. Програмування мовою С++.
4. Брюс Эккель, Чак Эллисон Философия С++.
5. Бьерн Страуструп Язык программирования С++.
6. Герберт Шилдт С++ Базовый курс.