Лекція 5. Вказівники та операції з ними

1. Поняття вказівника, оголошення та використання вказівників у програмі.

2. Внутрішня організація масивів. Зв'язок масивів і вказівників.

3. Унарні операції з вказівниками.

4. Бінарні операції з вазівниками.

1. Поняття вказівника, оголошення та використання вказівників у програмі

Вказівники - це змінні спеціального типу, значеннями яких є адреси різних об'єктів програми, насамперед змінних. Якщо ми використовуємо ім'я того чи іншого об'єкта для вилучення його значення, або для зміни його значення, то прийнято говорити про ***безпосередній*** (прямий) доступ до об'єкта. У тому випадку, коли адреса об'єкта міститься у вказівнику, мова йде про ***непрямий доступі до об'єкта, на який "дивиться" вказівник***.

Ідеї такої непрямої адресації зародилися ще в архітектурі ЕОМ першого покоління, коли адресу потрібної комірки пам'яті містив спеціальний регістр (наприклад, *РА - регістр адреси* в ЕОМ типу М-20). Доступ по вмісту регістра надавав програмістам більш широкі можливості за рахунок машинних команд зміни вмісту такого регістра (автоматичне збільшення – інкремент, або зменшення – декремент на 1) та використання РА в командах циклів. Найбільш повне втілення ідеї непрямої адресації знайшли в проекті адресного мови, яку розробив професор О.Л. Ющенко (Інститут кібернетики АН УРСР, Київ).

У мовах С, С + + розрізняють три категорії вказівників. Перша категорія вказівників призначена для зберігання ***адрес даних певного типу***. При їх оголошенні вказується тип даних, на які ці вказівники можуть "дивитися". До другої категорії відносяться ***вказівники***, які можуть "дивитися" на ***дані будь-якого типу***. При їх оголошенні використовується службове слово void. Нарешті, третю групу складають вказівники, значеннями яких можуть бути тільки ***адреси точок входу в функції.***

Для оголошення одного вказівника з іменем p1 або кількох вказівників з іменами p1, p2, ..., які повинні будуть "дивитися" на об'єкти типу type1, використовується одна з наступних синтаксичних конструкцій:

```
type1 *p1;
type1 *p1,*p2,...;
```

Оголошені вказівники ще ***нікуди*** конкретно ***не дивляться***, одна з типових помилок програмістів-початківців – спроба записати що-небудь за вказівником, якому ще не присвоєно значення.

```
int x=2,y;

int *p1=&x;   //ініціалізація адресою змінної x

int *p2(&x);  //ініціалізація адресою змінної x

int *p3=p1;   //ініціалізація значенням іншого вказівника
```

Вказівник є змінною і його значення можна задати або змінити за допомогою оператора присвоєння:

```
p1 = &y; / / тепер значенням p1 є адреса змінної y
```

У програмах на мові С можна зустріти нагромадження символів "*". Лякатися не треба – це просто багаторівнева адресація:

```
int x, y;

int * p1 = &y;

int ** p2 = &p1;

x = ** p2; / / те ж, що x = * (* p2) = * (p1) = y
```

Оголошення не типізованого вказівника виглядає наступним чином:

```
void * pu;
```

Нетипізованому вказівнику може бути присвоєно значення вказівника будь-якого типу. Однак безпосередньо витягти або змінити значення змінної за адресою, яку містить нетипізований вказівник не можна, потрібно вдаватися до приведення типів:

```
#include <iostream.h>

#include <conio.h>

void main()

{ int x=5;

  void *p=&x;

  int *p1;

  p1=(int*)p; //приведення вказівника p до типу int*

  cout<<"x="<<*p1<<endl;

  getch();

}
```

Окрім вказівників з адресами змінних в пам'яті ПК пов'язані також ***посилання*** – особливий вид даних, що при оголошенні отримує адресу змінної

і далі може використовуватися в якості додаткового імені (псевдоніма) цієї змінної. Наприклад, внаслідок виконання коду:

```
int x;

int & rx = x; / / оголошення та ініціалізація посилання
```

отримаємо посилання ***rx***, що є повним еквівалентом ідентифікатору ***x***. Далі, наприклад, оператори ***x = 5*** та ***rx = 5*** будуть виконувати присвоєння значення 5 одній і ті й же змінній ***x***. В такому варіанті особливої користі від використання посилань небагато, проте в С++ часто доводиться мати справу з довгими позначеннями властивостей об'єктів, і застосування посилань в цьому контексті дозволяє значно скоротити код програми. Але основна перевага посилань проявляється при специфікації параметрів функцій. Якщо формальний параметр оголошений в заголовку функції як посилання, то спрощується його використання в тілі функції (на відміну від вказівників до імен посилань нічого додавати не треба) і стає більш природним виклик функцій (замість формальних параметрів-посилань вказуються імена змінних).

2. Внутрішня організація масивів. Зв'язок масивів і вказівників

Масиви - це однорідні дані (тобто дані одного типу), розташовані в послідовних комірках оперативної пам'яті. Так як всі елементи таких даних мають однакову довжину, то для ідентифікації будь-якого елемента достатньо знати його порядковий номер у послідовності та адресу першого елемента масиву. Це дозволяє використовувати загальну групову назву для всіх елементів масиву і виділяти кожен з них лише індексом (або індексами), відповідним порядковому номеру елемента.

У простому випадку для так званих одновимірних масивів (векторів) індекс масиву і його порядковий номер фактично співпадають. Так як в мовах С, С + + прийнято відраховувати індекси від 0, то позначення ху [2] відповідає третьому елементу масиву з іменем ху. Якщо елементи цього масиву мають тип double і початковий елемент ху [0] розташований в оперативній пам'яті з адресою 0x02000000, то для обчислення адреси елемента ху [2] достатньо виконати кілька операцій - 0x02000000 +2 * 8. Природно, що такого роду операції перекладаються на компілятор, а програміст може записувати алгоритм обробки елементів масиву, маніпулюючи індексами його елементів.

Позначення елементів масиву, більш звичні для програміста, компілятор перетворює в вирази з вказівниками за правилами приведення індексу:

```
a[6]      еквівалентно  *(a+6)
```

Ці перетворення засновані на наступних угодах мови С. Ім'я одновимірного масиву ***a*** одночасно є вказівником на його перший елемент, тобто значенням, доступним за адресою ****a***, є елемент масиву ***a[0]***. Ім'я двовимірного масиву ***b*** одночасно є вказівником на вказівник його першого рядка, тобто значенням, доступним за адресою *****b***, є елемент масиву

b[0][0]. Вказівник ***b+1*** "дивиться" на вказівник, що визначає адресу першого елемента другого рядка масиву ***b***. Сенс подібного роду перетворень полягає в підвищенні ефективності програми, бо операції з вказівниками виконуються набагато швидше. Іноді до такого роду перетворень вдаються і програмісти, наприклад, зводячи обробку двовимірного масиву до одновимірних приведених індексів.

```
int a[10];
int *p1=a;       //p1 вказує на початок масиву a
int *p2=&a[0];   //p2 також вказує на початок масиву a
int *p3=(int *)&a;
//p3 також вказує на початок масиву a
```

Коли вказівник ***p2*** "дивиться" на початок масиву ***q***, то доступ до елементів цього масиву можна організувати одним із таких способів:

```
int q[20];
int *p2=q;
y=*(p2+5);     //тепер y=q[5]
x=p2[3];       //тепер x=q[3]
*(p2+1)=7;     //тепер q[1]=7
```

3. Унарні операції з вказівниками.

Основні операції, що найчастіше застосовують до вказівників – додавання до вказівника цілого числа та віднімання від вказівника цілого числа. Обидві вони широко застосовуються у випадку, якщо вказівник пов'язаний з масивом. По суті, ці операції еквівалентні аналогічним процедурам над індексами елементів масиву, так само як додавання цілого числа до індексу означає перехід до наступного елементу масиву. Арифметичні операції за кількістю операндів поділяються на унарні (одно) та бінарні (двоопераднді).

Над вказівниками можна виконувати операції інкременту та декременту як постфіксного, так і префіксного. Проте, якщо операції інкременту чи декременту для цілочисельних даних визначали зміну їх величини на одиницю, то у випадку вказівника зміна відбувається на одиницю обсягу пам'яті, яку займає дане, що на нього вказує вказівник. Тобто, якщо операція інкременту здійснюється над вказівником на цілий тип, то нове значення адреси збільшиться на стільки байт, скільки займає відповідний цілий тип. Якщо над вказівником на тип double, тоді на 8 байт, а на тип float – на 4 байти. При операції декременту адреса відповідно зменшиться. Нагадаємо, що ці операції можна виконувати тільки над змінними, тому виконання такої дії над ідентифікатором статичного масиву (назвою масиву) спричинить синтаксичну помилку.

При оперуванні зі вказівниками можна вказувати більше однієї унарної операції, зокрема можна поєднувати арифметичні операції та операцію непрямого доступу, пам'ятаючи, що унарні операції мають однаковий пріоритет, але виконуються справа наліво, а також слід враховувати постфіксний чи префіксний характер відповідної арифметичної операції.

Якщо буде записано ***cout<<*ptr_ar_1++***; то це означає, що виконається операція інкременту першою, але оскільки це постфіксний варіант, тоді фактично виведеться вміст за адресою, на яку вказує вказівник, а тоді адресована комірка збільшить своє значення на 4 (якщо вказівник вказує на дане цілого типу). Якщо є префіксний варіант – ***cout<<*++ptr_ar_1***, тоді спершу змінюється адреса, а тоді виводиться вміст за новою адресою. Якщо ж треба змінити вміст за адресою, то це слід виконувати з використанням дужок ***cout<<(*ptr_ar_1)++*** або міняти місцями операції ***cout<<++*ptr_ar_1***.

При такому виводі буде змінюватися лише вміст комірок (постфіксно або префіксно), що розташовані за відповідною адресою. Для того, щоб змінити і адресу і вміст треба двічі скористатися з операції зміни значення. Для прикладу виведення масиву подамо різні варіанти відповідних інструкцій. Вираз ***(*ptr_ar_1++)++*** читається так: при постфіксній зміні адреси беремо вміст за цією адресою, а тоді значення за цією адресою збільшується на 1. Префіксне збільшення вмісту з постфіксною зміною адреси дозволить поточний елемент масиву збільшити на 1 при використання операції розадресації, а тоді постфіксна зміна адреси дасть можливість перйти до наступного елемента масиву.

```
#include <iostream>

using namespace std;

void main()

     {int ar_int[]={2, 4, 0, 8, 9, -4};

      int *ptr_ar_1=ar_int;

      int n=sizeof(ar_int)/sizeof(ar_int[0]);

      cout<<"ar_int[i++]\t";

      for(int i=0; i<n; cout<<ar_int[i++]<<'\t');

      cout<<endl<<"(*ptr_ar_1++)++\t";

      for(int i=0; i<n; i++)

          cout<<(*ptr_ar_1++)++<<"\t";

          cout<<endl<<"ar_int[i++]\t";

     for(int i=0; i<n; cout<<ar_int[i++]<<'\t');

     cout<<endl<<"++*ptr_ar_1++\t";ptr_ar_1=ar_int;
```

```
for(int i=0; i<n; i++)
     cout<<++*ptr_ar_1++<<"\t";
     cout<<endl<<"ar_int[i++]\t";
for(int i=0; i<n; cout<<ar_int[i++]<<'\t');
     cout<<endl;
}
```

4. Бінарні операції з вказівниками.

Серед бінарних арифметичних операції, дозволених над вказівниками є лише операції додавання чи віднімання цілого числа ($ptr \pm i$), а також різниці двох вказівників (віднімання вказівників: $ptr1 - ptr2$). Додавання вказівників відсутнє, інших двомісних арифметичних операцій для вказівників немає, але для вказівників визначено операції порівняння.

Додавання до вказівника цілого числа означає, що відбувається зміна адреси на величину, що рівна кількості байт пам'яті, яку займає вказане число даних того ж типу, на який вказує вказівник. Якщо це вказівник типу double, то це означає, адреса пам'яті зміниться на величину кратну 8: якщо додається число n, то адреса збільшиться на 8n байт, якщо віднімається число p, тоді адреса зменшиться на 8p байт. Результатом віднімання двох вказівників буде число елементів даного типу, які можуть бути записані в області між вказаними двома адресами (вказівниками на той самий тип даних), результат може виявитися як додатним так і від'ємним залежно від того, який операнд є більшим.

Вказівник – це дане, що містить адресу. Значення цієї адреси може співпадати чи не співпадати зі значенням іншої адреси, тому вказівники можна порівнювати, тобто виконувати над ними операції порівняння. Якщо змінні локальні, то змінна, яка оголошена швидше, має більшу адресу, про що вказує результат порівняння:

```
int k,n;int *ptr_k=&k, *ptr_n=&n;
(ptr_k<ptr_n)?cout<<"<"<<endl:cout<<">"<<endl;
```

Вказівники можуть входити у вирази, і над результатами їх порівняння можна виконувати логічні операції (!, &&, ||), результат виконання яких буде типу bool.