Лекція 14
Вказівники

Оператор адреси &

При виконанні ініціалізації змінної, їй автоматично присвоюється вільна адреса в пам’яті, і, будь-яке значення, яке ми присвоюємо змінній, зберігається за цією адресою в пам’яті. Наприклад:

```
int b;
```

При виконанні цього оператора процесором, виділяється частина оперативної пам’яті. В якості прикладу припустимо, що змінній `b` присвоюється комірка пам’яті під номером 150. Всякий раз, коли програма зустрічає змінну `b` в вона розуміє, що для того, щоб отримати значення — їй потрібно зазирнути в комірку пам’яті під номером 150.

Хороша новина — нам не потрібно турбуватися про те, які конкретно адреси в пам’яті виділені для певних змінних. Ми просто посилаємося на змінну через присвоєний їй ідентифікатор, а компілятор конвертує цей ідентифікатор у відповідну адресу в пам’яті. Однак цей підхід має деякі обмеження, які ми обговоримо пізніше.

Оператор адреси `&` дозволяє дізнатися, яку адресу в пам’яті присвоєно певній змінній. Все дуже просто:

```
 1  #include <iostream>
 2
 3  int main()
 4  {
 5      int a = 7;
 6      std::cout << a << '\n'; // виводимо значення змінної a
 7      std::cout << &a << '\n'; // виводимо адресу в пам'яті змінної a
 8
 9      return 0;
10  }
```

Результат на моєму комп’ютері:

```
7
0046FCF0
```

Оператор розіменування *

Оператор розіменування `*` дозволяє отримати значення по вказаній адресі:

```
 1  #include <iostream>
```

```
2
3  int main()
4  {
5      int a = 7;
6      std::cout << a << '\n'; // виводимо значення змінної a
7      std::cout << &a << '\n'; // виводимо адресу змінної a
8      std::cout << *&a << '\n'; /// виводимо значення комірки в пам'яті змінної a
9
10     return 0;
11 }
```

Результат на моєму комп’ютері:

```
7
0046FCF0
7
```

Вказівники

Тепер, коли ми вже знаємо про операторів адреси і розіменування, ми можемо поговорити про вказівники.

Вказівник (або ***“покажчик”***) — це змінна, значенням якої є адреса комірки в пам’яті. Вказівники оголошуються так само, як і звичайні змінні, тільки із зірочкою між типом даних і ідентифікатором:

```
1 int  *iPtr; // вказівник на значення типу int
2 double  *dPtr; // вказівник на значення типу double
3
4 int*  iPtr3; // коректний синтаксис (дозволено, але не бажано)
5 int  *  iPtr4; // коректний синтаксис (не робіть так)
6
7 int *iPtr5, *iPtr6; // оголошуємо два вказівники для змінних типу int
```

Синтаксично мова C++ приймає оголошення вказівника, коли зірочка знаходиться поруч з типом даних, з ідентифікатором або навіть посередині. Зверніть увагу, ця зірочка НЕ є оператором розіменування. Це всього лише частина синтаксису оголошення вказівника.

Однак, при оголошенні кількох вказівників, зірочка повинна знаходитися біля кожного ідентифікатора. Це легко забути, якщо ви звикли вказувати зірочку біля типу даних, а не біля імені змінної. Наприклад:

```
int* iPtr3, iPtr4;    // iPtr3 - це вказівник на значення типу int, а iPtr4 - це звичайна змінна типу int!
```

З цієї причини, при оголошенні вказівника, рекомендується вказувати зірочку біля імені змінної. Як і звичайні змінні, вказівники не ініціалізуються при оголошенні. Вмістом неініціалізованого вказівника є звичайне сміття.

Присвоювання значень вказівнику

Оскільки вказівники містять тільки адреси, то при присвоюванні значення вказівнику — це значення повинно бути адресою. Для отримання адреси змінної використовується оператор адреси:

```
1 int value = 5;
2 int *ptr = &value; // ініціалізуємо ptr адресою значення змінної
```

Ось чому вказівники мають таку назву: `ptr` містить адресу значення змінної `value`, і, можна сказати, `ptr` ***вказує*** на це значення.

Ще дуже часто можна побачити наступне:

```
 1  #include <iostream>
 2
 3  int main()
 4  {
 5      int value = 5;
 6      int *ptr = &value; // ініціалізуємо ptr адресою значення змінної
 7
 8      std::cout << &value << '\n'; // виводимо адресу значення змінної value
 9      std::cout << ptr << '\n'; // виводимо адресу, яку містить ptr
10
11      return 0;
12 }
```

Результат на моєму комп'ютері:

```
003AFCD4
003AFCD4
```

Тип вказівника повинен відповідати типу змінної, на яку він вказує:

```
   int iValue = 7;
1 double dValue = 9.0;
2
3 int *iPtr = &iValue; // ок
4 double *dPtr = &dValue; // ок
5 iPtr = &dValue; // неправильно: вказівник типу int не може вказувати на адресу змінної
6 типу double
7 dPtr = &iValue; // неправильно: вказівник типу double не може вказувати на адресу
   змінної типу int
```

Наступне не є допустимим:

```
int *ptr = 7;
```

Це пов’язано з тим, що вказівники можуть містити тільки адреси, а цілочисельний **літерал** 7 не має адреси в пам’яті. Якщо ви все ж зробите це, то компілятор повідомить вам, що він не може перетворити цілочисельне значення в цілочисельний вказівник.

Мова C++ також не дозволить вам напряму присвоювати адреси в пам’яті вказівнику:

```
1 double *dPtr = 0x0012FF7C; // не ок: розглядається як присвоювання цілочисельного літералу
```

Оператор адреси повертає вказівник

Варто зазначити, що оператор адреси `&` не повертає адресу свого операнда в якості літералу. Замість цього він повертає вказівник, що містить адресу операнда, тип якого отримано з аргументу (наприклад, адреса змінної типу int передається як адреса вказівника на значення типу int):

```
 1 #include <iostream>
 2 #include <typeinfo>
 3
 4 int main()
 5 {
 6         int x(4);
 7         std::cout << typeid(&x).name();
 8
 9         return 0;
10 }
```

Результат виконання програми:

```
int *
```

Розіменування вказівників

Як тільки у нас є вказівник, який вказує на що-небудь, ми можемо його розіменувати, щоб отримати значення, на яке він вказує. Розіменований вказівник — це вміст комірки в пам’яті, на яку він вказує:

```
 1 #include <iostream>
 2
 3 int main()
 4 {
 5         int value = 5;
 6         std::cout << &value << std::endl; // виводимо адресу value
 7         std::cout << value << std::endl; // виводимо вміст value
 8
 9         int *ptr = &value; // ptr вказує на value
10         std::cout << ptr << std::endl; // виводимо адресу, яка зберігається в ptr (тобто
11 &value)
```

```
12          std::cout << *ptr << std::endl; // розіменовуємо ptr (отримуємо значення, на яке
13 вказує ptr)
14
            return 0;
    }
```

Результат:

```
0034FD90
5
0034FD90
5
```

Ось чому вказівники повинні мати тип даних. Без типу вказівник не знав би, як інтерпретувати вміст, на який він вказує (при розіменуванні). Також, тому і повинні збігатися тип вказівника з типом змінної. Якщо вони не збігатимуться, то вказівник при розіменуванні може неправильно інтерпретувати біти (наприклад, замість типу double використати тип int).

Одному вказівнику можна присвоювати різні значення:

```
 1  int value1 = 5;
 2  int value2 = 7;
 3
 4  int *ptr;
 5
 6  ptr = &value1; // ptr вказує на value1
 7  std::cout << *ptr; // виведеться 5
 8
 9  ptr = &value2; // ptr тепер вказує на value2
10 std::cout << *ptr; // виведеться 7
```

Коли адреса значення змінної присвоєна вказівнику, то виконується наступне:

`ptr` — це те ж саме, що і `&value`;

`*ptr` обробляється так само, як і `value`.

Оскільки `*ptr` обробляється так само, як і `value`, то ми можемо присвоювати йому значення так, наче це звичайна змінна. Наприклад:

```
1 int value = 5;
2 int *ptr = &value; // ptr вказує на value
3
4 *ptr = 7; // *ptr - це те ж саме, що і value, якому ми присвоїли значення 7
5 std::cout << value; // виведеться 7
```

Розмір вказівників

Розмір вказівника залежить від архітектури, на якій скомпільовано виконуваний файл: 32-бітний виконуваний файл використовує 32-бітні адреси в пам'яті. Відповідно, вказівник на 32-бітному пристрої займає 32 біти (4 байти). З 64-бітним виконуваним файлом вказівник займатиме 64 біти (8 байтів). І це незалежно від того, на що вказує вказівник:

```
1  char *chPtr; // тип char займає 1 байт
2  int *iPtr; // тип int займає 4 байти
3
4  struct Something
5  {
6      int nX, nY, nZ;
7  };
8
9  Something *somethingPtr;
10
11 std::cout << sizeof(chPtr) << '\n'; // виведеться 4
12 std::cout << sizeof(iPtr) << '\n'; // виведеться 4
13 std::cout << sizeof(somethingPtr) << '\n'; // виведеться 4
```

Як ви можете бачити, розмір вказівника завжди один і той же. Це пов'язано з тим, що вказівник — це всього лише адреса в пам'яті, а кількість біт, необхідна для доступу до адреси в пам'яті на певному пристрої, — завжди постійна.

Чим корисні вказівники?

Зараз ви можете подумати, що вказівники є непрактичними і взагалі непотрібними. Навіщо використовувати вказівник, якщо ми можемо використати вихідну змінну?

Вказівники корисні в наступних випадках:

Випадок №1: **Масиви реалізовані за допомогою вказівників**. Вказівники можуть використовуватися для ітерації по масиву.

Випадок №2: **Вони є єдиним способом динамічного виділення пам'яті в C++**. Це, безумовно, найбільш поширений варіант використання вказівників.

Випадок №3: **Вони можуть використовуватися для передачі великої кількості даних в функцію без копіювання цих даних.**

Випадок №4: **Вони можуть використовуватися для передачі однієї функції в якості параметру іншій функції.**

Випадок №5: **Вони використовуються для досягнення поліморфізму при роботі зі спадкуванням.**

Випадок №6: **Вони можуть використовуватися для представлення однієї структури/класу в іншій структурі/класі, формуючи, таким чином, "ланцюжки".**

Вказівники застосовуються в багатьох випадках. Тепер, коли ми розібралися зі вказівниками на базовому рівні, ми можемо почати заглиблюватися в окремі випадки, в яких вони корисні, що ми і зробимо на наступних уроках.

Висновки

Вказівники — це змінні, які містять адреси в пам’яті. Їх можна розіменувати за допомогою оператора розіменування * для вилучення значень, які містяться за адресою в пам’яті. Розіменування вказівника, значенням якого є сміття, призведе до збою в вашій програмі.