Лекція 8. Управління пам'яттю

Версія 21 травня 2023 р.

Анотація

Ця лекція говорить про одну з найважливіших основних тем, як для C++, так і для будь-якої іншої мови програмування.

Зміст

1 Основи **1**

2 Вказівники в C++ **2**
2.1 Вказівники, що не володіють пам'яттю (сирі) 4
2.2 Потенційні проблеми при використанні сирих вказівників . . . 4

3 Вказівники і класи **6**

4 Розумні вказівники **6**
4.1 std::unique_ptr 7
4.2 std::shared_ptr 8
4.3 std::weak_ptr 9

Література **10**

1 Основи

Оперативну пам'ять комп'ютера можна уявити як одновимірний масив байтів, які мають одну лінійну адресу – їх позицію, що часто записується в шістнадцятковій формі. Фізично, вона працює таким чином, що доступ до будь-якої адреси пам'яті займає однаковий час. Це дозволяє зберігати частини даних програми будь-де, не переймаючись завчасним виділенням блоку пам'яті. Для конкретного процесу, що виконується, ця пам'ять доступна

за адресою, яка зберігається в об'єкті спеціального типу, що зветься вказівником. Ця пам'ять виділяється під час виконання програми (runtime), тому зветься динамічною, на відміну від статичної пам'яті, яка виділяється операційною системою кожному процесу при старті і має фіксований розмір в вісім мегабайт на більшості Unix-like платформах. Проте, частіше, перша називається купою (heap memory), а друга – стеком (stack memory). Детальніше про це згодом.

2 Вказівники в C++

З одного боку, вказівники це звичайні об'єкти, зі своїм типом і розміром[1], а з іншого вони особливі, хоча б тим, що їх тип залежить від типу того самого об'єкта, яким може бути будь-який вбудований тип, або ваш тільки що створений клас. Маркером типа вказівника є зірка (∗). Наприклад:

- `MyType*` – вказівник на тип (клас) `MyType`;
- `const MyType*` – вказівник на `const MyType`;
- `MyType* const` – константний (незмінний) вказівник на `MyType`;
- `const MyType* const` – константний (незмінний) вказівник на `const MyType`;

Константність вказівника означає, як і для будь-якої іншої змінної, що ми не можемо змінити її значення, тобто адресу. Іншими словами, перенаправити його на інший об'єкт. У той час як вказівник на константний тип означає, що ми не можемо модифікувати сам екземпляр, або викликати його неконстантні методи.

Знову ж таки, якщо створити екземпляр вказівника без ініціалізації, то його "значенням" буде якесь довільне число, що буде невірно трактуватись як адреса, яка скоріше за все належить іншому процесу, або є вільною, якщо ми спробуємо його якимось чином використати. Вказівники є ще особливими тим, що вони можуть бути в явно неініціалізованому стані, не вказувати ні на що. Досягається це присвоюванням ключового слова `nullptr`.

Щоб ініціалізувати вказівник, необхідно створити екземпляр його відповідного типу та отримати її адресу за допомогою ключового слова `new`.[2] Наприклад,

Лістинг 1: "Сирі вказівники"

```
#include <iostream>
```

[1]Це фактично число, розмір не залежить від типу об'єкта на який він вказує, а радше від кількості пам'яті на комп'ютері, щоб було можливо зберігати індекс першого байта.

[2]Або `new[]`, якщо ми створюємо масив.

```
int main() {

    int* i; // unutilized
    std::cout << "This is a garbage address: " << i <<
        std::endl;

    i = new int(42);
    std::cout << "This is the address with my value: " << i <<
        std::endl;
    std::cout << "The value is: " << *i << std::endl;

    *i -= 2;
    std::cout << "Now the value is: " << *i << std::endl;

    return 0;
}
```

Якщо запустити скомпільований файл, то побачимо

```
This is a garbage address: 0x7ff7b7390d60
This is the address with my value: 0x7fde0af05b60
The value is: 42
Now the value is: 40
```

Звичайно, можна отримувати і маніпулювати адресами не тільки динамічної пам'яті, а також і локальної:

```
int i = 42;
int* ptr = &i;
*ptr -= 2;
std::cout << "The value of i is: " << i << std::endl;
```

На виході побачимо 40. На практиці існують випадки, коли це доцільно, але скоріше за все, якщо ви бачите спокусу маніпулювати адресами стекової пам'яті, то вам варто ретельніше подумати над реалізацією, тому що зазвичай це призводить до великої кількості проблем, які важко відстежити, і до коду, який важко зрозуміти. В цих прикладах ви побачили використання оператора розіменування посилань, $*$, що дає доступ до екземпляру, який знаходиться за цією адресою, а також оператора взяття адреси, $\&$, що повертаю адресу екземпляра, до якого він застосовується. Якщо маємо вказівник на екземпляр якогось класу, то його треба розіменовувати перед тим, як викликати якісь методи, тобто `(*ptr).begin()`. Оскільки це доволі часта операція, в C++ для цього існує спеціальний синтаксис, оператор `->`,

який автоматично здійснює розіменування, тобто `ptr->begin()`.

Цікавим є той факт, що оскільки вказівники є у якомусь розумінні звичайними об'єктами, на них також можна посилатись за допомогою своїх вказівників. У такому випадку додається відповідно ще одна[3] зірка до типу, та оператор розіменування застосовується двічі, оскільки першого разу ми отримаємо нову адресу. Наприклад

```
int i = 42;
int* ptr = &i;
int** ptr_to_ptr = &ptr;
**ptr_to_ptr *= 2;
std::cout << "The new value is: " << **ptr_to_ptr <<
    std::endl;
```

2.1 Вказівники, що не володіють пам'яттю (сирі)

Вказівники, які ви бачили до цього моменту, звуться сирими (іноді ще голими), тобто такими, що ніяк не управляють пам'яттю, на яку вказують, а просто тримають до неї адресу, тим самим даючи доступ. Управління або володіння тут не означають, що такий вказівник повинен бути принципово іншим, а радше, що він мусить мати додатковий інтерфейс поверх звичайного вказівника, який ставить умови для роботи з пам'яттю. Але про це згодом. Сирі вказівники не викликають інструменти управління пам'яттю C++, а саме оператори `new` та `delete`. Якщо ми вже бачили роботу оператора `new` вище, то про оператор `delete` варто сказати, що він використовується для звільнення отриманої пам'яті.[4] За великим рахунком, на кожен виклик `new` повинен бути один, і тільки один, виклик `delete`,[5] чого ми не побачили в лістингу 1, що може призвести до так званого витоку пам'яті, про який ми поговоримо в наступній секції.

2.2 Потенційні проблеми при використанні сирих вказівників

Ситуація, зображена в лістингу 1 – це приклад витоку пам'яті (memory leak), де чотири байти, що були виділені операційною системою для проце-

[3] Або більше, якщо це вказівник на вказівник на вказівник...

[4] Так само, як і для `new`, наряду зі звичайним `delete` існує також `delete[]` для вказівників на масиви.

[5] За великим рахунком, не є великою проблемою, якщо `delete` не буде явно викликатись на ту пам'ять, яка використовуються на протязі всього часу виконання програми, тому що, при її завершені, вся пам'ять вивільняється автоматично, але на це не варто розраховувати.

су нашої програми на число, вказівник до якого ми зберігали в локальній змінній i, не вивільнились після того, як ми завершили роботу з i, і ми опинились в ситуації, коли ні цей процес, ні будь-який інший на цьому комп'ютері, не може їх використати.

Друга типова проблема, це так званий висячий вказівник (dangling pointer). Це, можна сказати, зворотна ситуація, коли пам'ять вже вивільнилась, а вказівник ще доступний користувачеві. Небезпечно це тим, що розіменування такого вказівника призведе до невизначеної поведінки: для простих вбудованих типів, ми можемо отримати значення за замовченням, або якесь довільне значення (шум) і навіть не зрозуміти, реальне це значення, чи ні, або, і це насамперед для класів, це може призвести до термінової зупинки роботи програми. Щоб не трапилось, той факт, що до цього дійшло, є логічною помилкою. Наприклад,

Лістинг 2: "Висячий вказівник"

```
#include <iostream>
int main() {
    int* i = new int(42);
    // use data
    (*i)++;
    // release data
    delete i;
    // further use is undefined behavior
    std::cout << "The value is: " << *i << std::endl;

    return 0;
}
```

Остання найбільш поширена ситуація, це так зване подвійне вивільнення пам'яті (double free corruption), тобто ситуація, коли оператор `delete` (або `delete[]`) викликається другий раз на одну і ту саму адресу. Коли програма має більше однієї логічної одиниці (що охоплює за великим рахунком їх всі), то іноді автор програми може збитися з пантелику, при намаганні самотужки відстежити де саме повинна вивільнятися пам'ять.

Також, через те, як побудовані масиви в C, не завжди може бути зрозуміло чи ми працюємо зі вказівником на один елемент, чи на масив, що важливо, щоб розуміти треба викликати `delete` або `delete[]`. Використання неправильного оператора призводить до невизначеної поведінки.

Для запобігання цих ситуацій, важливо поєднати час життя вказівників на стеку, та об'єктів, на які вони вказують, в купі. Це називається принципом RAII в C++ (Resource allocation is initialization). В стандартній бібліотеці C++ його реалізовано у вигляді так званих розумних вказівників, про

яких ми поговоримо на наступних секціях.

3 Вказівники і класи

Окремо зупинимось на тому, яку роль вказівники грають при роботі з класами.

Ми вже бачили, що коли в нас є вказівник на екземпляр будь-якого класу, ми можемо викликати його метод, або звернутись до елементу даних за допомогою оператора `->`, який поєднує розіменування (оператор `*`) з викликом (оператор `.`).

Коли ми розробляємо код якогось методу класу, часто там необхідно звернутись до якогось іншого методу, або елементу даних об'єкта, що викликав цей метод.[6] В багатьох інших мовах програмування це досягається тим, що методи функцій мають перший обов'язковий аргумент, який є (вказівником, посиланням тощо) екземпляром, що їх викликав. Оскільки часто це є надлишковим, в C++ вказівник на екземпляр передається неявно і є доступним через зарезервовано ключове слово `this`. Сам вказівник є завжди `const`, а об'єкт, на який він вказує, є таким тільки для методів, що помічаються `const`.

Також, вказівники є тим інструментом, яким досягається динамічний поліморфізм.[7] Тобто, через вказівник на базовий клас, зокрема на інтерфейс, ми можемо писати функції, або інші інтерфейси, які будуть працювати з будь-якою його конкретною реалізацією. Викликатися буде правильна[8] версія окремо взятого віртуального методу завдяки віртуальний таблиці.

4 Розумні вказівники

Розумні вказівники – це звичайні об'єкти-обгортки, або надбудови, над сирими вказівниками, що зберігають інтерфейс взаємодії з ними,[9] вводячи певні обмеження та автоматизації, щоб мінімізувати випадки будь-якої з перелічених, і потенційно інших, проблем. Тобто, вони "керують" часом життя екземпляра, та запобігають або управляють множинним володінням. [10]

[6]Інакше цей метод скоріше за все був би статичним, тобто таким, який не прив'язаний до конкретного екземпляру.

[7]Посилання також є поліморфними, тому знову може постати вибір – використовувати посилання чи вказівники. Всі ті самі аргументи, що і раніше, є валідними. Найчастіше на практиці це саме вказівники, тому фокус на них і тут.

[8]Тобто реалізація класу екземпляра, на не вказівника.

[9]Тобто мають відповідні імплементації для операторів `*`, `->`, `=`, що зокрема дозволяє присвоювати їм `nullptr`, та інші.

[10]Іншими словами, ситуація, коли у вказівника є декілька користувачів.

Так само як і сирі вказівники, вони поліморфні.

На практиці, ми використовуємо фабричні методи, які викликають оператор `new` (або `new[]`), отримують вказівник та передають його новоствореному розумному вказівнику, який і повертається. Пам'ять вивільняється деструктором розумного вказівника при його видаленні на стеку, або при роботі окремих його методів, коли треба видалити або перезаписати сирий вказівник, що знаходиться всередині. Через автоматичне вивільнення пам'яті, розумні вказівники не можна використовувати для вказування на екземпляри зі стеку.[11]

В стандартній бібліотеці є три види розумних вказівників: `std::unique_ptr`, `std::shared_ptr`, та `std::weak_ptr`. Це шаблони класів, головним параметром яких є тип, на який він вказує.[12] Про кожен ми поговоримо окремо далі.

4.1 std::unique_ptr

Якщо вирішення задачі потребує використання вказівників, то першим кандидатом на цю роль повинен бути `std::unique_ptr`. Конструктор його приймає голий вказівник на об'єкт, який він приймає під своє управління. Проте, як було зазначено, чистіше і безпечніше для таких цілей використовувати фабричні методи, а саме `std::make_unique`. Це шаблон функції, яка приймає ті самі аргументи, що і конструктор, який ми хочемо викликати, для типу, за яким він параметризується. Наприклад,

```
auto myClassPtr = std::make_unique<MyClass>(1, "A", true);
// auto is resolved to std::unique_ptr<MyClass>
// equivalent to
// std::unique_ptr<MyClass> myClassPtr(new MyClass(1, "A",
    true));
```

Як можна зрозуміти з його назви, цей розумний вказівник реалізує кон-

[11] Брати адресу стекових об'єктів взагалі може бути ризикованою справою, але іноді це необхідно або корисно. Для цього доречно використовувати сирі вказівники – як мінімум ми не зіштовхнемося з витоком пам'яті, бо стекові об'єкти видаляються автоматично при виході зі своєї області, а пам'ять залишається належати програмі на весь час її виконання у будь-якому разі.

[12] Другим параметром є функтор (клас, екземпляри якого мають оператор виклику, як функції), який задає логіку видалення об'єкту. У переважній більшості випадків ми нічого не передаємо і користуємось видаленням за замовченням, що реалізована в класі default_delete, який використовує оператор `delete`. Але розумні вказівники можна використовувати не тільки для об'єктів в купі, а для багато іншого також. Наприклад, файли чи мережеві з'єднання. Коли ми відкриваємо файл чи налаштовуємо з'єднання, їх важливо закрити в кінці. Цю логіку можна імплементувати в окремому класі в передати розумному вказівнику.

цепт, коли він є єдиним власником пам'яті. Тобто, його не можна скопіювати, тільки перемістити. Наприклад,

```
auto ptr = std::make_unique<std::string>("text string");
// we can use it
std::cout << "The text is: " << *ptr << std::endl;
// we can't copy it
// auto ptr2 = ptr;
// we can move it
auto ptr2 = std::move(ptr);
// and use the new object from now on
std::cout << "ptr2 points to: " << *ptr2 << std::endl;
// ptr is now nullptr
std::cout << "Is ptr null? " << (ptr ? "false" : "true") <<
    std::endl;
// as with normal nullptr, we cannot dereference it
// std::cout << "ptr points to: " << *ptr << std::endl; //
    this will crash
```

Сам по собі `std::unqiue_ptr` достатньо простий клас (який не так важко написати самому!), а його використання фактично не додає ніяких додаткових навантажень в скомпільованому коді, в порівнянні з використанням сирих показників. Тому, навіть якщо ви знаєте, що окремо взятий об'єкт необхідний впродовж всього часу виконання програми і його не потрібно буде видаляти, використання `std::unique_ptr` все ще може допомогти запобігти потенційним негараздам. Наприклад, ситуації, коли все ж таки хтось його видалив. Або, якщо логіка програми зміниться, і його час життя скоротиться. На додачу, іншим, хто буде дивитись ваш код не потрібно бути думати і аналізувати, чи не є помилкою використання сирого вказівника в цьому конкретному випадку, чи може це призвести до потенційних проблем.[13]

4.2 std::shared_ptr

Існують обґрунтовані причини для того, щоб вказівник використовували різні процеси одночасно,[14] або ж він зберігався в різних місцях одночасно. Для таких випадів стандартна бібліотека має готове рішення в вигляді `std::shared_ptr`.

З точки зору реалізації, він набагато складніший за `std::unique_ptr`,

[13]Звичка думати про тих, хто може працювати з вашим кодом у майбутньому не тільки позитивно впливатимете на його якість, а ще і збереже час всіх причетних.

[14]Тут треба говорити про цілу окрему дисципліну багатопоточного використання спільних ресурсів, але вона виходить за рамки цього курсу.

тому застосовувати його рекомендовано тільки якщо це дійсно треба, а використовується він так само. `std::shared_ptr` використовує концепцію спільного володіння пам'яттю. А саме, пам'ять виділяється при створенні першої (оригінальної) копії, а видалиться, коли остання копія закінчить своє існування. Їй може бути будь-яка з існуючих. При створенні першої копії `std::shared_ptr`, його конструктор також створює пов'язаний з цим екземпляром об'єкт-лічильник. Потім, при кожному копіюванні через оператор `=`, або конструктор копіювання, цей лічильник зростає на одиницю, і аналогічно зменшується завдяки деструктору, аж поки він не буде дорівнювати нулю, коли він[15] сам видаляється і пам'ять вивільняється. З тих самих міркувань, що і у випадку з `std::unique_ptr`, варто використовувати фабричний метод `std::make_shared` для створювання екземплярів `std::shared_ptr`.

4.3 std::weak_ptr

Наостанок, варто розглянути третій різновид розумних вказівників, який на справді не є вказівником, `std::weak_ptr`. Чим він є, так це способом дістати `std::shared_ptr`, якщо це можливо. Він зручний, коли ми хочемо мати вказівник на ресурс, але не володіти ним, не впливати на його лічильник, щоб його час життя не залежав від цієї копії. Іноді, це трапляється з точки зору якомога швидшого вивільнення пам'яті, або якщо вона логічно невід'ємна від іншого процесу, який мусить її вивільнити по закінченню, або у випадках, коли це технічно необхідно при кругових залежностях. З цього опису повинно бути зрозуміло, що конструктор `std::weak_ptr` приймає відповідний `std::shared_ptr`.

Як вже булл зазначено, `std::weak_ptr` не є вказівником. Він не реалізує відповідний інтерфейс, і сам по собі не дасть нам доступ до ресурсу. Щоб достукатись до пам'яті, треба спершу отримати на неї `std::shared_ptr`, що можливо зробити за допомогою метода lock. На виході, ми отримуємо справжній `std::shared_ptr`, що призведе до збільшення лічильника, але час життя цього `std::shared_ptr` буде меншим за `std::weak_ptr`, що цього створив, тому потенційна "затримка" оригінального об'єкту буде мінімальною. Якщо `std::shared_ptr`, на який дивиться наш екземпляр `std::weak_ptr` пустий, тобто ресурс вивільнено, `lock` повертає `std::shared_ptr` створений за замовченням, тобто такий, який дорівнює `nullptr`. Перевірити цей випадок можна за допомогою метода expired.

[15] Який до речі також існує в купі.

Література

- Stroustrup, Bjarne. The C++ programming language. Pearson Education, 2013 (Глави 7, 34).
- CPlusPlus, Pointer tutorial.
- CPPReference, Dynamic memory management.