Міністерство освіти і науки України
Національний університет «Києво-Могилянська академія»
Факультет інформатики

Курсова робота

першого (бакалаврського) рівня вищої освіти

на тему: **«СТРАТЕГІЇ ІНТЕЛЕКТУАЛЬНИХ УКАЗНИКІВ»**

Виконав: студент 3-го року навчання,
Спеціальності: 121 Інженерія програмного забезпечення,
Освітньої програми: Інженерія програмного забезпечення,
Клепацький Олег Святославович

Керівник Бублик В.В.
доцент кандидат фіз.-мат. наук

Київ – 2023

Зміст

Зміст 2
Перелік скорочень 4
1 Вступ 5
2 Особливості узагальненого метапрограмування (templates) 7
2.1 Проблеми інстанціювання шаблонів 7
2.2 Часткова спеціалізація шаблонів 8
2.3 Концепти як засіб типізації шаблонів 8
2.4 Метод захоплення ресурсів 9
2.5 Інтелектуальні указники 11
3 Узагальнене програмування засобами стандартної бібліотеки шаблонів STL 13
3.1 Автоматичний указник (auto_ptr) 13
3.2 Одноосібний указник (unique_ptr) 13
3.3 Розподілений указник (shared_ptr) 14
3.4 Слабкий указник (weak_ptr) 15
3.5 Невладний указник (observer_ptr) 19
4 Дизайн програмних систем на основі стратегій (Policy-Based Design) ... 21
4.1 Бібліотека метапрограмування Loki 22
4.2 Стратегії за Александреску 22
4.2.1 Стратегія зберігання 22
4.2.2 Стратегія володіння 23
4.2.3 Стратегія перетворення (конвертації) 24
4.2.4 Стратегія перевірки 24
5 Рефакторинг успадкованого коду Александреску 26
5.1 Застарілий код 26
5.2 Відсутність вимог до типових параметрів шаблону (стратегій) 26
5.3 Перехід до семантики переміщень 27
5.4 Використання застарілих ідіом 28
5.4.1 Зведення типів для шаблонного конструктору 28
5.4.2 Ідіома конструктор переміщення 29
5.4.3 Анонімний перелік 29
5.5 Використання застарілих технік метапрограмування 30

5.6 Відсутність стратегії видалення 31
6 Демонстраційна програма випробування стратегій 34
6.1 Визначення стратегій 36
6.1.1 Спільно використані стратегії 37
6.1.2 Стратегії видалення 38
6.1.3 Стратегії володіння 39
6.2 Застосування стратегій 42
6.2.1 Одноосібний указник 42
6.2.2 Розподілений указник 45
6.2.3 Невладний указник 47
7 Висновок 49
8 Список використаних джерел 51
9 Додаток 53

Перелік скорочень

1. C++03, C++11, C++17, C++20 – стандарти ISO мови програмування C++
2. RAII - Resource Acquisition Is Initialization, або метод захоплення ресурсів
3. STL - Standrard Template Library, стандартна бібліотека шаблонів
4. SDL - Simple DirectMedia Layer

1 Вступ

Непряме використання пам’яті, іншими словами застосування указників, було наявне вже в перших мовах програмування. Досить згадати адресну мову програмування, створену К. Л. Ющенко і В. С. Королюком в середині 50-их років. Розвитком техніки указників стали інтелектуальні указники, які дозволили уникнути проблем, пов’язаних з використанням звичайних указників, шляхом управління використання пам’яті.

Для дослідження інтелектуальних указників використаємо дизайн на основі стратегій – прийом шаблонного метапрограмування, детально описаний Александреску у його книзі [1].

Актуальність теми зумовлена тим, що використання інтелектуальних указників є невід'ємною частиною сучасного програмування, а C++20 містить нові можливості, які можуть допомогти розробникам створювати більш лаконічні та оптимальні реалізації стратегій інтелектуальних указників.

Метою курсової роботи є дослідження прийомів метапрограмування та нових особливостей, доданих до стандарту C++20, опис стратегій інтелектуальних указників та їх реалізація на C++20.

Завдання роботи:

1. Дослідити та описати прийоми метапрограмування, які використовує дизайн на основі стратегій
2. Дослідити та застосувати особливості метапрограмування в новому стандарті C++20.
3. Порівняти стратегії інтелектуальних указників Александреску (на основі C++03) з STL (C++11).
4. Виявити та виправити недоліки дизайну на основі стратегій в C++03.
5. Визначити стратегії інтелектуальних указників на основі C++20.
6. Протестувати реалізацію нових стратегій у демонстраційному проекті.

Об'єктом дослідження є реалізації інтелектуальних указників на основі стратегій.

Предметом дослідження є шляхи реалізації інтелектуальних указників, їх порівняльний аналіз та вирішення проблем, які виникають з використанням шаблонного метапрограмування, підходами з нових стандартів C++.

Практичне значення роботи полягає у модернізації рішення, запропонованого Александреску, для можливості його використання у сучасних програмах як альтернатива уже існуючим реалізаціям інтелектуальних указників, які можуть не вирішувати усіх проблем, які виникають під час використання динамічної пам’яті та інших ресурсів.

2 Особливості узагальненого метапрограмування (templates)

2.1 Проблеми інстанціювання шаблонів

Використовуючи шаблони у C++, може скластись враження, що семантично у параметр шаблону можна підставити будь-який тип. Насправді це не зовсім так.

Програміст шаблонів класів і функцій робить припущення про операції, які можна виконувати над параметрами шаблону. Тому дуже важливо знати, які типи є допустимими для конкретного параметру шаблону, тобто відповідають усім неявним синтаксичним вимогам до нього.

Користувачі шаблонних функцій та класів є відповідальними за дотримання всіх вимог до параметрів шаблонів. Якщо синтаксична вимога була порушена, програміст міг побачити це, наприклад, у помилці компілятора про відсутню функцію.

На жаль, такі помилки часто виявляються глибоко всередині великого стеку викликів функцій при інстанціюванні деяких внутрішніх реалізацій технічних деталей, які не є справою програміста. У результаті, повідомлення про помилку не мають особливого значення ні для кого, окрім автора неінстанційованого внутрішнього коду. Те, що компілятор надає виклик та включення стеку у довжелезні повідомлення, більше лякає, ніж допомагає програмістам зрозуміти, в чому полягає помилка компіляції. Втім, синтаксичні помилки завжди виявлялися компілятором, хоч і не завжди відображені у корисній, читабельній формі. Семантичні помилки, з іншого боку, не виявляються компіляторами.

2.2 Часткова спеціалізація шаблонів

Часткова спеціалізація шаблонів – це прийом метапрограмування, який дозволяє налаштовувати та змінювати властивості шаблонів класів для заданої категорії параметрів шаблону [2].

До цього поняття також можна віднести псевдоніми шаблонів [3]. Використовуючи псевдоніми, ми можемо задати усі або декілька параметрів шаблону заздалегідь, спрощуючи використання шаблонного класу. На прикладі (Лістинг 1) псевдонім SecondInt заміняє шаблон класу TwoParams та має лише параметр шаблону замість двох, а другий визначає як int. Якщо ми підставимо у цей псевдонім double як параметр, тип об'єкту буде TwoParams<double, int>, оскільки другий параметр визначено псевдонімом.

Лістинг 1

```
template<typename T, typename U>
struct TwoParams {};

template <typename V>
using SecondInt = TwoParams<V, int>;

// ...

SecondInt<double> instance;
std::cout << typeid(instance).name(); // struct TwoParams<double, int>
```

2.3 Концепти як засіб типізації шаблонів

Для більш якісного використання шаблонів, ми повинні явно визначити набір вимог до потенційних параметрів та мати спосіб застосувати його до параметрів так, щоб функція чи клас не компілювались доти, доки усі вимоги до параметрів шаблону не були виконані. Такі набори вимог називаються концептами (concepts) і були додані до стандарту C++20.

Концепт може містити суто синтаксичні вимоги, такі як «певний параметр повинен мати копіювальний конструктор або визначений operator+». Також можливо задати семантичні вимоги, що визначають

обов'язкову властивість, наприклад, комутативність додавання: $x + y = y + x$ $\forall x, y$.

До C++20 концепти існували лише у документації (якщо взагалі існували), а користувачі шаблонних функцій та класів були відповідальні за дотримання всіх вимог до параметрів шаблонів. Якщо синтаксична вимога була порушена, програміст міг побачити це, наприклад, у помилці компілятора про відсутню функцію. Тепер, починаючи з C++20, ми можемо виражати наші концепти як частину наших програм.

Використання концептів має кілька переваг [4]:

- Недотримані вимоги виявляються вже на етапі виклику функції або інстаціювання шаблону.
    - A не глибоко всередині реалізації.
    - Це призводить до значно коротших повідомлень про помилки.
- Повідомлення про помилки часто є більш змістовними.
- Є можливість виявляти семантичні помилки.
- Функції можуть диспетчеризуватись за властивостями типу параметру.

Поки що, використовуючи концепти, можна виразити лише безпосередньо синтаксичні вимоги, але Петер Готшлінг демонструє також емуляцію семантичних концептів [4]. Диспетчеризація за властивостями типів вже була можлива за допомогою метапрограмування раніше, але концепти роблять реалізацію набагато простішою та більш читабельною.

2.4 Метод захоплення ресурсів

Метод захоплення ресурсів (RAII, або Resource Acquisition Is Initialization) – «це техніка програмування на C++, яка пов'язує життєвий цикл ресурсу, який повинен бути отриманий перед використанням (виділена динамічна пам'ять, потік виконання, відкритий сокет, відкритий файл, заблокований м'ютекс, дисковий простір, з'єднання з базою даних - все, що існує в обмеженій кількості), з життєвим циклом об'єкту» [5].

Детерміноване видалення є головною особливістю C++. Коли ім'я об'єкта програмованого типу виходить за межі області видимості, об'єкт знищується, тобто викликається його деструктор, і вся пам’ять об’єкта очищається. Не потрібно чекати, поки збирач сміття зробить свою справу і видалить об’єкти, як це зазвичай буває у мовах з автоматичним збиранням сміття (garbage collection), що може призвести до неприємних недетермінованих побічних ефектів, таких як вичерпання ресурсів у невідповідний момент, або до того, що очищення взагалі буде відкладено чи пропущено [6].

Однак, коли ім'я вказівника виходить за межі області видимості, знищується саме вказівник, а не об'єкт, на який він вказує, залишаючи об'єкт без імені та, очевидно, без засобів виклику його деструктора. Той же принцип стосується усіх ресурсів, лише замість звільнення пам’яті там відповідно відбуватиметься звільнення цього ресурсу (закриття сокету, розблокування м’ютексу тощо).

Насправді, це не обмежується просто механічною помилкою програміста, який може забути звільнити ресурс, як тільки він йому не потрібен, наприклад, у кінці блоку. По-перше, шлях виконання функції не завжди відомий: функція може мати декілька розгалужень, кожне з яких може закінчити її виконання, або виклик іншої функції може визвати виключну ситуацію, і ми не зможемо звільнити ресурс. По-друге, цикл життя ресурсу може виходити за рамки блоку його отримання (наприклад, ресурс може тримати змінна об’єкту або статична змінна), відтак ми маємо знати та пам’ятати, в який момент треба його звільнити.

Як бачимо, підхід до «ручного» управління ресурсами не тільки є схильним до помилок, він також може порушувати принцип єдиної відповідальності класу (Single Responsibility).

Отже, враховуючи необхідність вручну слідкувати за звільненням ресурсів та детермінований виклик деструктора у C++, бачимо необхідність у реалізації окремих типів для керування ресурсом. Ці типи матимуть

семантику володіння даним ресурсом. Одним з підходом буде отримувати ресурс при ініціалізації об’єкта такого класу (про що власне і говорить ідіома RAII – “Resource Acquisition Is Initialisation”) та автоматично звільняти його у деструкторі. Це гарантуватиме нам, що ресурс не буде втрачено.

2.5 Інтелектуальні указники

Загалом, сирі указники мають немало застосувань у C [7]:

1. Як дешеве, некопіююче представлення об'єкту, що належить тому, хто викликає функцію;
2. Як спосіб модифікації об'єкту, що належить тому, хто викликає функцію;
3. Як пари вказівник/довжина, що використовується для масивів
4. Як необов'язковий аргумент (або дійсний вказівник, або nullptr)
5. Як спосіб керування динамічною пам'яттю, виділеною на купі.

Для перших двох пунктів у C++ використовуються посилання (const& та &). Окрім того, семантика переміщень, що була додана у стандарті C++11, дозволяє передавати за значенням об’єкт, уникаючи його копіювання. Щодо повернення об’єкту з функції за значенням, компілятори оптимізують це, і по факту виконується переміщення об’єкта із функції у місце виклику (це називають Return Value Optimisation або RVO) [8].

Починаючи з C++17, ми можемо використовувати std::string_view для передачі стрічок як параметр без копіювання, що задовольняє 1 та 3 пункти. Для масиву довільних об’єктів (не тільки символів) у STL існують класи std::array та std::vector для статичних та динамічних масивів відповідно. Щодо 4 пункту, до C++17 стандарту також додався клас std::optional, який дозволяє явно вказати, що параметр необов’язковий.

Інтелектуальні указники вирішують проблему, згадану у 5 пункті.

Інтелектуальні указники (Smart pointers) – це окремий підтип класів, які реалізують ідіому RAII. Ресурс у цьому випадку – динамічна пам’ять,

виділена за допомогою операторів new/delete (або аналог у С – функції malloc/calloc/realloc та free).

Дехто може вважати таку заміну сирим указникам дещо накладною, оскільки замість звернення напряму до вказівника, ми маємо спочатку дістати його з обгортки. Це так, але плюси використання цих класів перевищують недолік у вигляді декількох зайвих викликів, а по пам'яті розмір інтелектуального указника не перевищує розмір звичайного указника в найпростішому випадку (звичайно, це не стосується окремих типів інтелектуальних указників, про що згодом).

Основна вимога до таких класів – реалізовувати семантику володіння, таким чином взявши на себе відповідальність звільнити зайняту пам'ять, як тільки вона не буде використовуватись. Як вже було сказано, дана операція відбуватиметься у деструкторі інтелектуального указника, оскільки він буде викликаний автоматично по завершенню життя об'єкта.

Додатково, ці класи мають симулювати сирі указники, підтримуючи їх синтаксис і семантику. Це реалізовується за допомогою перевантаження операторів розіменування: operator-> та унарний operator*. Також необхідно забезпечити неявну конвертацію до булевого значення, щоб мати змогу перевіряти указник на порожнє значення (nullptr) в умовному операторі, як це можна зробити зі звичайним указником. Чи дозволяти при цьому неявну конвертацію до типу сирого указника буде залежати від реалізації, але хорошою практикою буде уникати такої конвертації і натомість надавати метод для явного отримання указника, який перебуває у власності об'єкта інтелектуального указника.

Усе це потрібно, щоб використання інтелектуальних указників замість звичайних не ускладнювало код і вимагало мінімальних змін до коду. Таким чином, ми можемо синтаксично використовувати їх як звичайні указники, але з усіма перевагами таких класів.

3 Узагальнене програмування засобами стандартної бібліотеки шаблонів STL

3.1 Автоматичний указник (auto_ptr)

Перша відома реалізація інтелектуального указника у стандартній бібліотеці С++ - клас auto_ptr. Для забезпечення інваріанту і семантики єдиного володіння, під час копіювання об'єкту auto_ptr володіння передається лівій частині присвоєння, а інформація з правої частини видаляється, тобто копія не дорівнюватиме оригіналу. Через цю незвичну семантику копіювання auto_ptr не можна розміщувати у стандартних контейнерах [9].

3.2 Одноосібний указник (unique_ptr)

Натомість у С++11 було додано клас unique_ptr, який виправляє недоліки попередника. З підтримкою семантики переміщень стало можливо повністю заборони копіювання указника з єдиним правом на власність, але надати конструктор і присвоєння переміщення. Вагомим плюсом також є підтримка застосування цього класу у контейнерних класах, які надаються стандартною бібліотекою.

Шаблон класу unique_ptr має два параметри, перший з яких – обов'язковий, і є власне типом, указником на який буде володіти об'єкт. Через те, що указник може вказувати на динамічний масив, створений оператором new[], існує часткова спеціалізація шаблону для типу T[], який у деструкторі замість delete викликатиме коректний delete[].

Другий параметр шаблону – необов'язковий, і це стратегія видалення указника. Стратегія за замовчуванням – виклик delete. Якщо існує ситуація, коли указник ініціалізується і звільняється за допомогою інших функцій (наприклад, API багатьох C бібліотек), для коректного звільнення ресурсів ми мали б передати указник у функцію, яка відповідає за його видалення.

Таким чином, ми можемо перевикористати цей же клас для багатьох різних нестандартних випадків.

3.3 Розподілений указник (shared_ptr)

Виникають ситуації, коли нам необхідно «поділитись» указником, не віддаючи повне право на володіння тому, хто буде ним користуватись. Семантика володіння unique_ptr у такому разі не є підходящою. Тобто, ми хочемо, щоб право володіння мали усі, хто має цей указник, і останній, хто буде ним користуватись, і буде відповідати за його звільнення.

Такий указник називають shared_ptr. Схожу семантику мають майже усі мови з автоматичним очищенням сміття, такі як Java, C#, але вони мають дещо іншу реалізацію. C++ такої «розкоші» не має, тому маємо власноруч визначати, у яких місцях нам необхідна така семантика володіння указником.

До C++11 такий інтелектуальний указник надавала бібліотека Boost [10]. Реалізацію з цієї бібліотеки я детально не розглядатиму, оскільки за принципом роботи вона є дуже схожою на ту, яку надає стандартна бібліотека C++11.

Ця реалізація використовує принцип підрахування відсилок. Це означає, що клас матиме лише указник на контрольний блок, який триматиме власне сам указник на об’єкт та лічильник, який вказує на те, скільки об’єктів наразі володіють інформацією про цей блок (Рис. 1). При копіюванні об’єкта shared_ptr лічильник збільшується на 1 і копіюється указник на цей блок. У деструкторі shared_ptr зменшує лічильник на 1 та перевіряє, чи він не досяг нуля. У такому разі, він останній має посилання на цей блок і має видалити його.

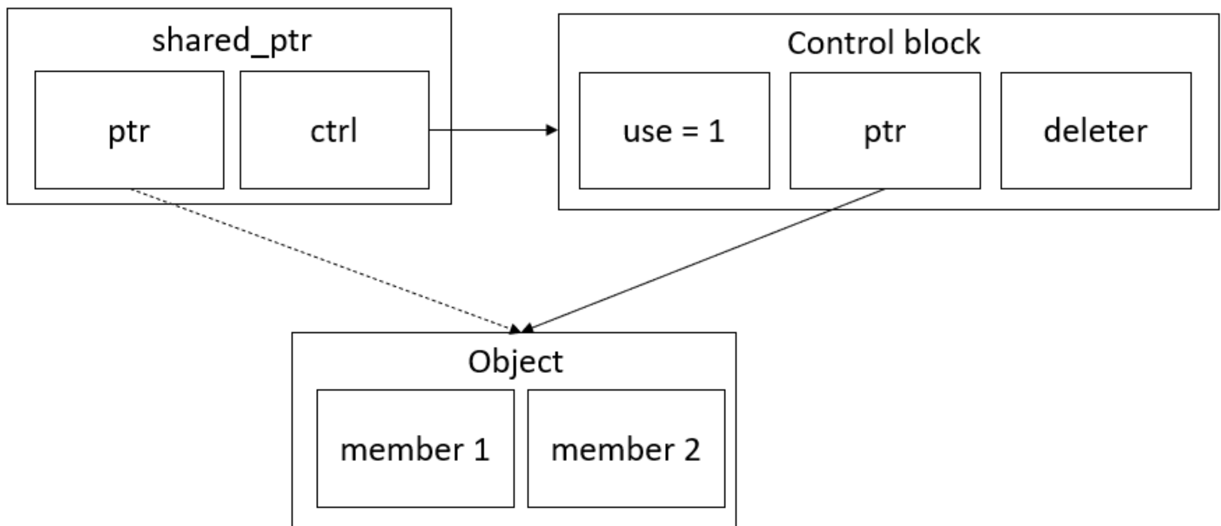


Рис. 1

3.4 Слабкий указник (weak_ptr)

Може виникнути прикра ситуація, коли два класи (A та B) мають циклічні посилання один на одного через shared_ptr (Лістинг 2). У такому разі, якщо ми створимо два shared_ptr з цими класами так, що один об'єкт матиме посилання на другий і навпаки, їхню пам'ять ніколи не буде звільнено!

Лістинг 2

```
struct B;
struct A {
	std::shared_ptr<B> b;
	~A() { std::cout << "~A()\n"; }
};
struct B {
	std::shared_ptr<A> a;
	~B() {std::cout << "~B()\n"; }
};
int main()
{
	std::shared_ptr<A> a = std::make_shared<A>();
	std::shared_ptr<B> b = std::make_shared<B>();
	a->b = b;
	b->a = a;
}
```

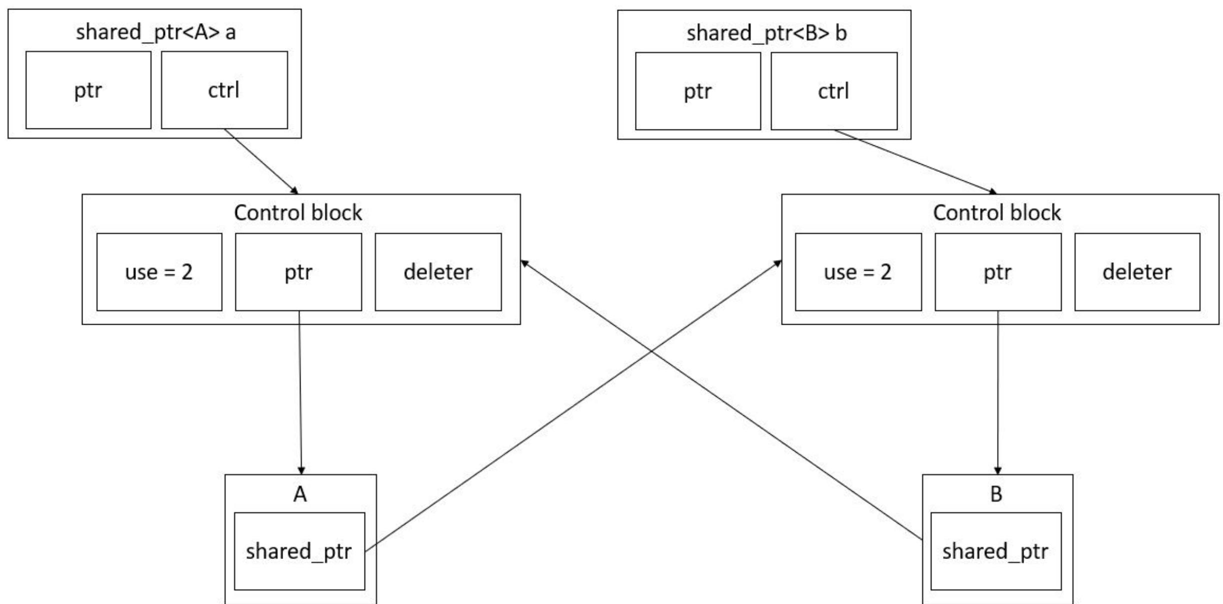
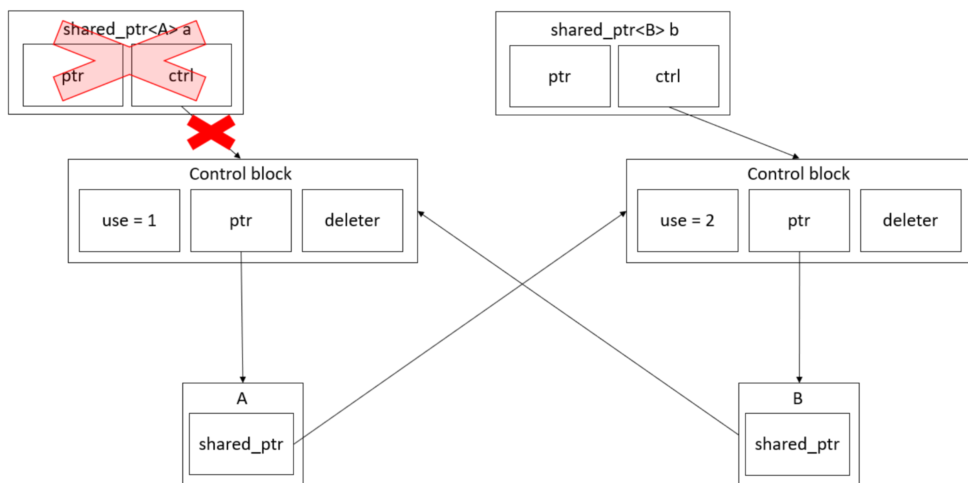


Рис. 2

Змоделюємо видалення цих shared_ptr, назвемо їх ptrA, ptrB. При видаленні ptrA матиме лічильник 2 (на нього ще вказує об’єкт у ptrB). При видаленні ptrA нічого не робитиме з об’єктом, яким він володіє та який посилається на об’єкт, яким володіє ptrB. Відповідно, на лічильнику



залишиться 1 (Рис. 3).

Рис. 3

Аналогічна ситуація відбудеться при видаленні указника ptrB. Тобто, після їх видалення самі об'єкти не буде звільнено, і вони триматимуть відсилку один на одного у пам'яті до кінця роботи програми. Маємо витік пам'яті. (Рис. 4)

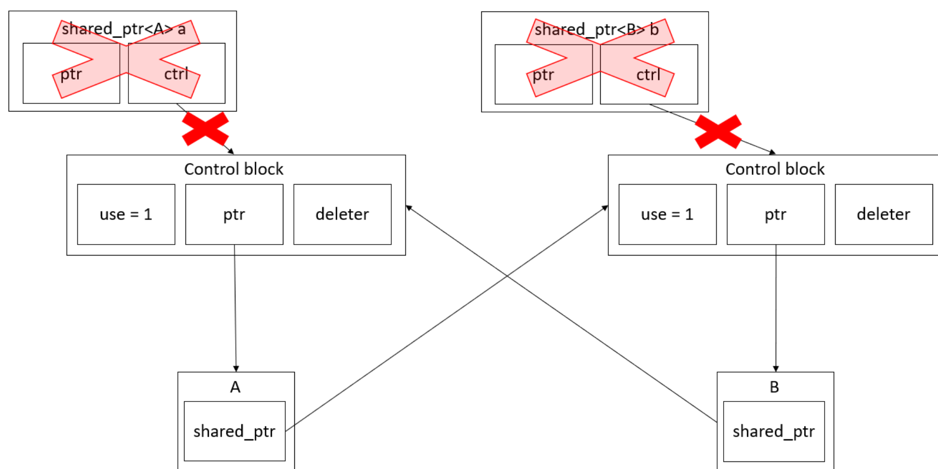


Рис. 4

Ця проблема вирішується додатковим класом weak_ptr. Він діє аналогічно до shared_ptr з єдиною різницею, що він не відповідає за звільнення пам'яті, відповідно при копіюванні він не додає себе до основного лічильнику у контрольному блоці. Якщо ми все ж хочемо розіменувати указник weak_ptr, ми маємо спочатку отримати екземпляр shared_ptr і працювати з ним.

Щоб вирішити попередню проблему, ми маємо вирішити, який клас є основним, а який є підпорядкованим. Підпорядкований клас не буде мати права володіння на об'єкт основного, але основний буде, і відповідатиме за видалення підпорядкованого об'єкту. Нехай A – основний клас, який володітиме B, тоді B матиме лише «слабку» відсилку на екземпляр A. Замість shared_ptr у клас B треба додати weak_ptr на клас A. (Рис. 5)

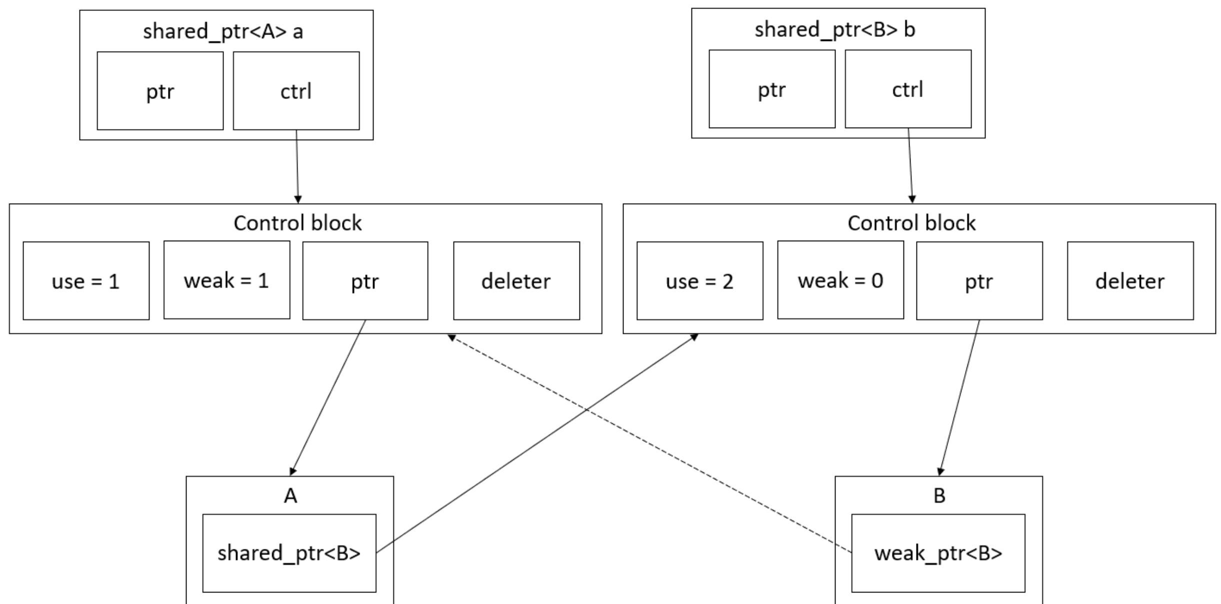
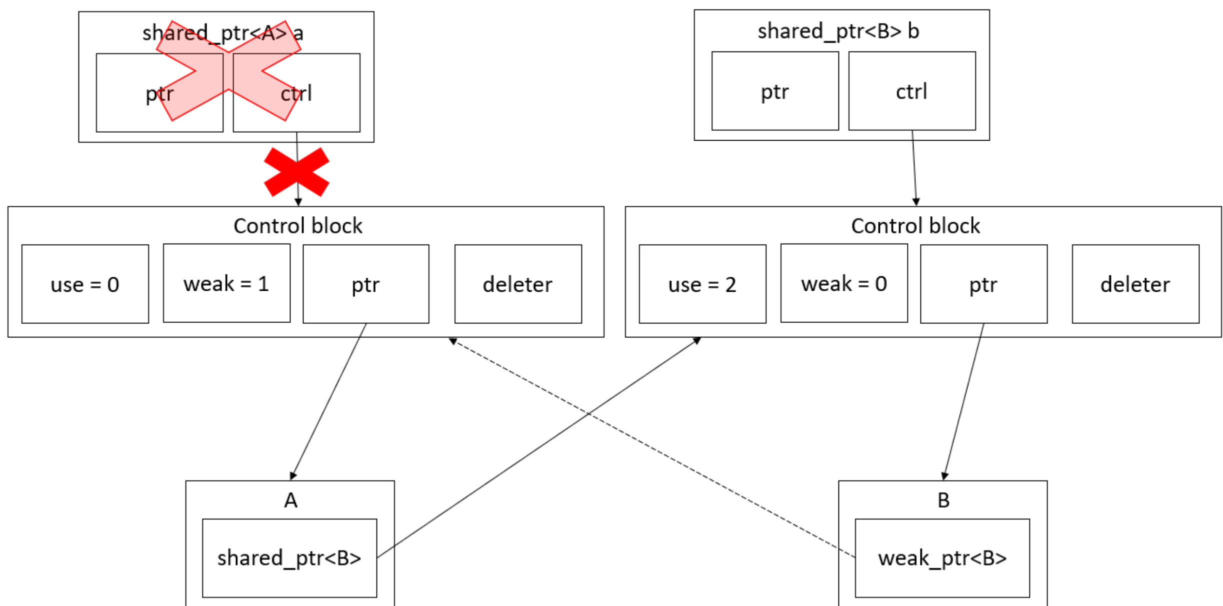


Рис. 5

Під час видалення вказівника А, контрольний блок буде видалено, не



зважаючи на існуючий слабкий указник. (Рис. 6)

Рис. 6

Після видалення об’єкту А, лічильник у контрольному блоці указника В зменшиться на 1, як і потрібно. (Рис. 7)

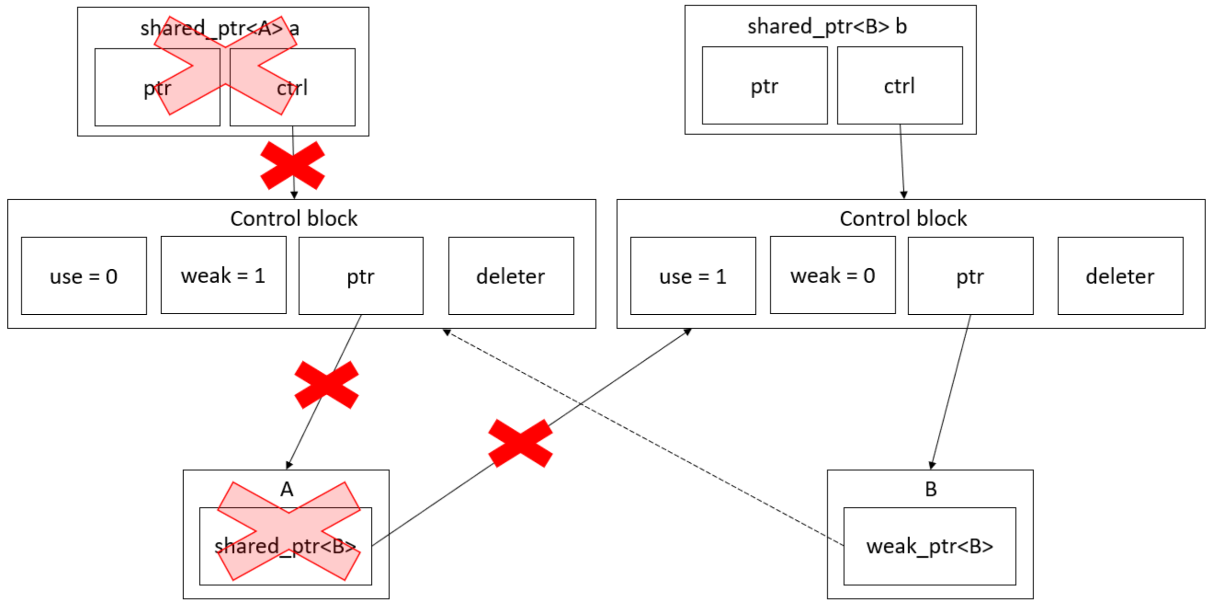
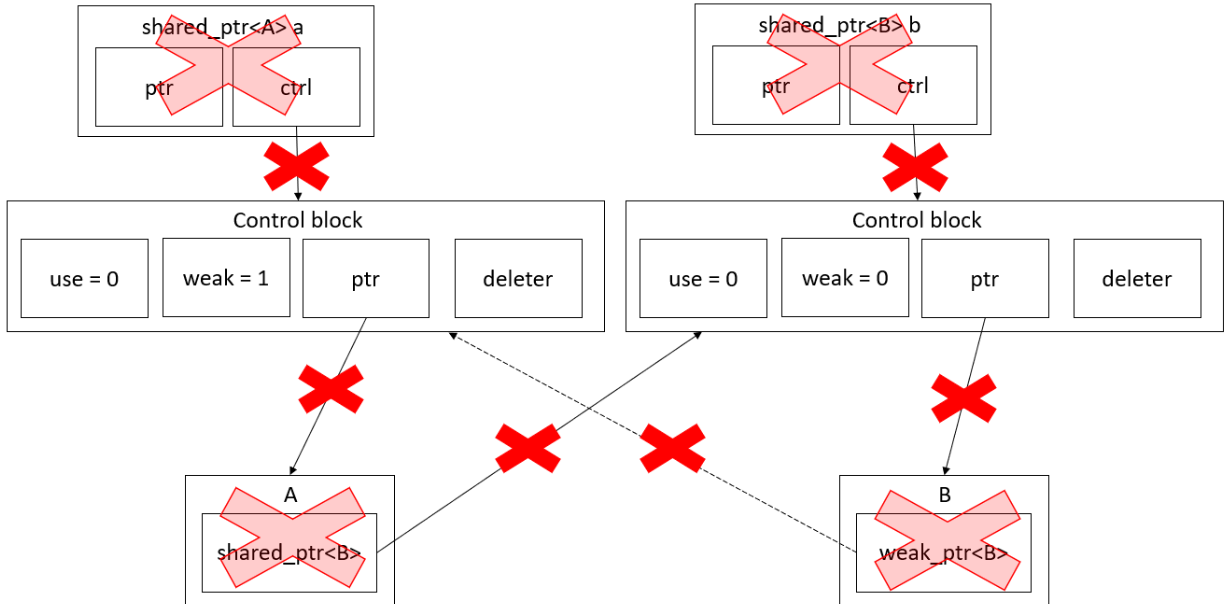


Рис. 7

Під час видалення указника B лічильник досягне нуля і пам’ять буде



звільнено. (Рис. 8)

Рис. 8

3.5 Невладний указник (observer_ptr)

Усі попередні типи інтелектуальних вказівників виражають семантику володіння в тому чи іншому вигляді. Але часто виникають ситуації, коли функція має приймати вказівник на існуючий об’єкт без наміру його змінювати. Звичайно, це може бути виведено з однієї лише сигнатури функції, оскільки цей параметр може також бути і вхідним, і вихідним.

Щоб позбавитись такої невизначеності, яка присутня у мові C, у C++ були додані відсилки. Вони вказують, що функція не буде мати відношення до управління пам’яттю, на яку вказує відсилка.

У разі, якщо уникнути використання вказівників неможливо, і необхідно, щоб параметр залишався вказівником, можемо визначити клас-обгортку, ім’я якого і буде вказувати нам, що ми не володіємо указником, який він зберігає.

Такий клас називається observer_ptr, але його не було додано до стандарту. На це є декілька причин, в тому числі – жахлива назва, як уважає Артур О'Двайер, яка не має нічого спільного з взірцем проектування спостерігач (observer) [7].

4 Дизайн програмних систем на основі стратегій (Policy-Based Design)

У книзі Банди чотирьох стратегія описується як взірець проектування, який дозволяє перевизначити поведінку класу в залежності від наданого нами типу [11]. Тобто, виконання методу деякого класу може делегуватись класу-стратегії, який власне і визначатиме, як саме буде виконуватись метод. Клас-стратегія, очевидно, має визначати інтерфейс, на який спирається клас, який її використовує. У класичній реалізації стратегії можна змінювати динамічно, на етапі виконання програми.

Александреску надає дещо складніший інтерфейс інтелектуального указника [1]. Його дизайн спирається на поєднанні метапрограмування (шаблонів) та множинного спадкування. Таким чином, ідея, описана Бандою Чотирьох, дещо модифікується: стратегії тепер визначаються на етапі компіляції за допомогою параметрів шаблону.

Множинне спадкування надає гнучкість обирати, які саме реалізації ми хочемо об’єднати та перевикористати у похідному класі. Але цей підхід матиме свої недоліки. Наприклад, це недостатня інформація про типи, яка може бути необхідна у випадку створення об’єктів (у C++ не існує віртуальних конструкторів).

Поєднавши шаблони та множинне спадкування ми отримуємо дуже потужний інструмент для створення гнучких дизайнів, які можуть перевикористовувати майже будь-які комбінації з визначених стратегій та, що головніше, дозволяти користувачам визначати власні і таким чином модифікувати поведінку шаблону.

При цьому самі стратегії не мають бути жодним чином пов’язані, інакше це може призвести до ще більш заплутаного дизайну. Ключ до успіху – це визначення «ортогональних» стратегій, які відповідають за кардинально різні аспекти поведінки даного класу.

4.1 Бібліотека метапрограмування Loki

Loki – це бібліотека C++, яка була написана Александреску як частина його книги Modern C++ Design. Геніальна на свій час, вона реалізовувала багато функціоналу, якого не існувало у тогочасному стандарті C++, а також деякі взірці проектування. В тому числі, в ній присутня реалізація інтелекутальних указників. Пізніше бібліотека продовжила розвиватись вже як проект з відкритим вихідним кодом за безпосередньої участі R. Sposato та P. Kuemmel [12].

4.2 Стратегії за Александреску

Дизайн Александреску передбачає один основний клас SmartPtr, який параметризуватиметься типом об'єкта, яким він володіє, та чотирма стратегіями:

1. Зберігання (StoragePolicy)
2. Володіння (OwnershipPolicy)
3. Перетворення (ConversionPolicy)
4. Перевірки (CheckingPolicy)

4.2.1 Стратегія зберігання

Найчастіше інтелектуальний вказівник зберігатиме звичайний указник на тип. Ця стратегія також визначатиме порядок доступу до ресурсу (указника), оператори розіменування operator->() та operator*() а також визначатиме типи, які повертатимуть відповідні методи (така гнучкість необхідна для того, щоб, наприклад, повертати не сам указник, а деяку обгортку над ним, т.зв. проксі-об'єкт).

Також ця стратегія визначатиме, яким чином треба звільнити тип, який зберігається у ній.

Найчастіше цим типом буде звичайний указник. Але можливі ситуації, коли це треба змінити. Наприклад, near та far указники, які використовувались для керування пам’яттю у старих процесорах.

Також Александреску дає у приклад використання цієї стратегії обгортку над іншим застарілим класом інтелектуального указника, який ми хочемо модифікувати. Успадковувати цей клас буде не найкращою ідеєю, а у такий спосіб можна легко перевизначити його поведінку. Це нагадує взірець адаптер [11].

Насправді, така стратегія не обмежує нас лише вказівниками. Існують випадки, коли у нас є деякі дескриптори, які надає ОС, щоб користувач не мав прямого доступу до ресурсів. Такі типи технічно не є указниками (наприклад, файлові дескриптори у Linux є цілими числами), тобто до них не застосовна семантика указників, але інтелектуальні указники надають зручний спосіб роботи з такими дескрипторами. Оскільки це такі самі ресурси, ми можемо вказати їх тип у стратегії, але просто не визначати оператори розіменування.

У своїй роботі я обмежився лише однією стратегією, яка зберігає указник.

4.2.2 Стратегія володіння

Стратегія володіння є найважливішою з-поміж усіх, оскільки інтелектуальний указник має визначати володіння над деяким указником, і його реалізація по більшості і спирається на дану стратегію. Дана стратегія має визначати умови, за яких стратегія зберігання буде очищати об’єкт, який зберігається.

Під час копіювання інтелектуального указника викликається метод Clone стратегії володіння, який має 1 параметр – відсилку на указник (чи

інший тип), який лежить у стратегії зберігання. Метод Clone може видалити указник після копіювання, якщо того вимагає стратегія.

Я вже розглянув деякі з типів володіння, які присутні в STL. У Александреску [1] є стратегії, деякі з них відповідають інтелектуальним указникам із стандартної бібліотеки:

- DeepCopy – при копіюванні указника копіюється об'єкт за допомогою метода Clone()
- RefCounted (або багатопоточна версія RefCounterMT) – shared_ptr
- COMRefCounted – стратегія, яка викорстовує Windows API для підрахунку відсилок
- RefLinked – двозв'язний список усіх, хто указує на контрольний блок; коли список стає пустим, відбувається видалення
- DestructiveCopy – auto_ptr
- NoCopy – забороняє будь-яке копіювання

4.2.3 Стратегія перетворення (конвертації)

Клас, який є стратегією перетворення, має лише визначати булеву константу, значення якої доступне на етапі компіляції, яка означатиме, чи може інтелектуальний указник бути явно приведений до указника типу, який він тримає.

Існує лише дві стратегії конвертації: AllowConversion та DisallowConversion, які дозволяють чи забороняють неявну конвертацію відповідно.

4.2.4 Стратегія перевірки

Стратегії перевірки потрібні, щоб впевнитись, що інтелектуальний указник тримає ненульове значення. Ще одне можливе використання, це

впевнитись, що ми розіменовуємо коректний (ненульовий) указник. Можуть бути різні способи реакції на це: або аварійна зупинка програми (assert), або створення виключної ситуації тощо. У разі assert, цієї перевірки не буде у версії релізу, тому на етапі відлагодження програми можна скористатись перевагами перевірки, а коли помилки буде виправлено, уникнути витрат на ці перевірки.

Стратегія має визначати три методи:

- onDefault – перевірка у конструкторі за замовчуванням
- onInit – перевірка у конструкторі, який приймає указник
- onDereference – перевірка перед розіменуванням

Варіантами таких стратегій можуть бути:

- NoCheck – жодної перевірки
- AssertCheck – використовує assert для перевірки перед розіменуванням (Лістинг 3) [1].

Лістинг 3

```
template <class P>
struct AssertCheck
{
	AssertCheck() {}

	template <class P1>
	AssertCheck(const AssertCheck<P1>&) {}

	template <class P1>
	AssertCheck(const NoCheck<P1>&) {}

	static void onDefault(const P&) {}

	static void onInit(const P&) {}

	static void onDereference(const P& val)
	{
		assert(val);
	}

	static void swap(AssertCheck&) {}
};
```

- AssertCheckStrict – також перевіряє указник у конструкторі.

5 Рефакторинг успадкованого коду Александреску

Хоч ідеї Александреску були проривні на час випуску його книги, зараз його реалізація може здатись дещо застарілою. Це пов’язано в першу чергу з тим, що стандарт мови не стоїть на місці, а комітет додає новий, часто корисний функціонал до стандартної бібліотеки. В тому числі це вже згадані класи інтелектуальних указників.

5.1 Застарілий код

Почати варто з того, що сучасні компілятори взагалі не можуть зібрати SmartPtr, який є у бібліотеці Loki. Це може спрацювати зі старішими версіями (до C++17), але через деякі залежності усередині бібліотеки її треба модифікувати під новий стандарт для нормальної працездатності.

5.2 Відсутність вимог до типових параметрів шаблону (стратегій)

Для кращої читабельності програмного коду та явного задання вимог до стратегій та інших параметрів шаблону, необхідно використати концепти (concepts) зі стандарту C++20, як на це вказує Страуструп у основних рекомендаціях до C++ [13].

Для усіх стратегій були додані вимоги щодо їх реалізацій. Ці вимоги загальні для усього шаблону SmartPtr. Також були додані допоміжні концепти, які є необов’язковими в усіх випадках, а лише необхідні для деяких методів (Лістинг 4).

Лістинг 4

```
template <typename T>
concept Swappable = requires(T& l, T& r) {
	{ l.swap(r) } -> std::same_as<void>;
};

template <typename T>
concept ConversionPolicyConcept = requires {
	{ T::allow } -> std::same_as<const bool&>;
};

namespace {
	struct M {};
```

```
}

template<template <typename> class T, typename P>
concept CheckingPolicyConcept = requires(T<P>&t, const P & p, const T<M>& t1) {
	{ T<P>()} -> std::same_as<T<P>>;
	{ T<P>(t1)} -> std::same_as<T<P>>;
	{ T<P>::onDefault(p)} -> std::same_as<void>;
	{ T<P>::onInit(p)} -> std::same_as<void>;
	{ T<P>::onDereference(p)} -> std::same_as<void>;
};

template <template <class> class T, typename P>
concept OwnershipPolicyConcept = requires(T<P>& t, const P& p, const T<M>& t1) {
	{ t.release(p) } -> std::same_as<bool>;
};

template<template <class> class OP, typename T>
concept CopyCloneable = requires (OP<T> op, const T& t) {
	{op.clone(t)} -> std::same_as<T>;
};

template<template <class> class OP, typename T>
concept MoveCloneable = requires (OP<T> op, T&& t) {
	{op.moveClone(std::move(t))} -> std::same_as<T>;
};

template<template <class> class T, typename P>
concept StoragePolicyConcept = requires(T<P> t, const T<P> ct) {
	typename T<P>::StoredType;
	typename T<P>::PointerType;
	typename T<P>::ReferenceType;
	{ct.getPointer() } -> std::same_as<typename T<P>::PointerType>;
	{t.getPointerRef() } -> std::same_as<typename T<P>::StoredType&>;
	{ct.getPointerRef() } -> std::same_as<const typename T<P>::StoredType&>;
	{ct.defaultValue()} -> std::same_as<typename T<P>::StoredType>;
};

template<template <class> class T, typename P>
concept MovableStoragePolicyConcept = StoragePolicyConcept<T,P> and requires(T<P> t) {
	{ t.getPointerRRef() } -> std::same_as<typename T<P>::StoredType&&>;
};

template<template <class> class T, typename P>
concept Dereferencable = StoragePolicyConcept<T, P>
and requires(T<P> t, const T<P> ct) {
	{ t.operator->() } -> std::same_as<typename T<P>::PointerType>;
	{ ct.operator->() } -> std::same_as<const typename T<P>::PointerType>;
	{ t.operator*() } -> std::same_as<typename T<P>::ReferenceType>;
	{ ct.operator*() } -> std::same_as<const typename T<P>::ReferenceType>;
};
```

5.3 Перехід до семантики переміщень

У C++11 були додані семантики переміщення, що дозволяє коректно реалізувати семантику єдиного володіння. Так, auto_ptr було позначено застарілим та замінено на unique_ptr, а у C++17 auto_ptr було видалено із стандартної бібліотеки. Саме через відсутність семантики переміщень у стратегіях Александреску не могло існувати прямого відповідника unique_ptr.

Мною були додані конструктор і оператор присвоєння переміщення, а також метод moveClone стратегії володіння, який використовується під час переміщення. До конструкторів копіювання та переміщення були додані додаткові концепти до стратегії, які вимагають наявність методів clone чи moveClone відповідно (Лістинг 5).

Лістинг 5

```
template<template <class> class OP, typename T>
concept CopyCloneable = requires (OP<T> op, const T& t) {
	{op.clone(t)} -> std::same_as<T>;
};

template<template <class> class OP, typename T>
concept MoveCloneable = requires (OP<T> op, T&& t) {
	{op.moveClone(std::move(t))} -> std::same_as<T>;
};
```

5.4 Використання застарілих ідіом

Для обходу деяких обмежень C++ було використано ідіоми або реалізовано іншими шляхами деякий необхідний фунціонал.

5.4.1 Зведення типів для шаблонного конструктору

Наприклад, для конструктора копіювання деяких стратегій володіння, коли вони інстанційовані з різними параметрами, було використано reinterpret_cast для зведення до однакового типу (Лістинг 6). Така проблема виникає, оскільки шаблон, інстанційований різними типами, згенерує два різні класи, які не матимуть доступу до приватних членів класу один одного. Зведення було необхідно, бо на той час не існувало friend шаблонних класів.

Лістинг 6

```
template <class P>
class RefCounted
{
public:
	//...
	template <typename P1>
	RefCounted(const RefCounted<P1>& rhs)
	: pCount_(reinterpret_cast<const RefCounted&>(rhs).pCount_)
	{}
	//...
private:
	uintptr_t* pCount_;
```

```
};
```

Натомість reinterpret_cast було видалено і додано визначення friend класу (Лістинг 7).

Лістинг 7

```
template <class P>
class RefCounted
{
public:
	//...
	template<class P1> friend class RefCounted;

	template<class P1>
	RefCounted(const RefCounted<P1>& rhs) : pCount_(rhs.pCount_) {}
	//...
private:
	uintptr_t* pCount_;
};
```

5.4.2 Ідіома конструктор переміщення

Для симуляції семантики переміщення був використаний т.зв. «Colvin-Gibbons trick» [14], який реалізований за допомогою класу RefToValue з бібліотеки Loki. Було замінено на вже згадану семантику переміщень.

5.4.3 Анонімний перелік

Анонімний перелік (anonymous enum) – це тип переліку без імені, часто використовувався для того, щоб обійти обмеження компіляторів старих версій, які не дозволяли визначати константи на етапі компіляції усередині класів.

Такі константи використовуються у стратегіях, наприклад, у стратегії конвертування, для позначення того, чи можна конвертувати інтелектуальний указник у звичайний неявно. (Лістинг 8)

Лістинг 8

```
struct AllowConversion
{
	enum { allow = true };
	// ...
};
```

Проблема полягає у тому, що наявність такої константи у класі набагато важче перевірити, ніж звичайну константу. До того ж, починаючи з C++11 стало доступним ключове слово constexpr, яке дозволяє визначати

константи часу компіляції. Це було замінено в усіх стратегіях, де присутні константи (Лістинг 9).

Лістинг 9

```
struct AllowConversion
{
	static constexpr bool allow = true;
	//...
};

template <typename T>
concept ConversionPolicyConcept = requires {
	{ T::allow } -> std::same_as<const bool&>;
};
```

5.5 Використання застарілих технік метапрограмування

В залежності від обраних стратегій, нам необхідно видалити той чи інший метод з класу SmartPtr. Наприклад, якщо стратегія конвертування забороняє неявну конвертацію у тип указника, нам необхідно замінити тип оператора конвертування на такий, щоб при його виклику він не зміг скомпілюватись, наприклад, на закритий вкладений тип класу SmartPtr. Для цього використовується шаблон Select, який за допомогою спеціалізації шаблону обирає тип на етапі компіляції (Лістинг 10).

Лістинг 10

```
template <bool flag, typename T, typename U>
struct Select { typedef T Result; };

template <typename T, typename U>
struct Select<false, T, U> { typedef U Result; };

template</* other params ... */ class CP>
class SmartPtr
{
	// ...
private:
	struct Insipid
	{
		Insipid(PointerType) {}
	};

	typedef typename Select<CP::allow, PointerType, Insipid>::Result AutomaticConversionResult;

public:
	operator AutomaticConversionResult() const
```

```
	{ return GetImpl(*this); }
};
```

Ця техніка використовує часткову спеціалізацію шаблону і дозволяє обрати тип в залежності від константи у стратегії. Коли стратегія забороняє неявну конвертацію, тип, який буде повертати оператор приведення до указнику буде недоступний до користувача класу, а відтак не буде компілюватись, і компілятор не буде неявно приводити клас до указника.

Але, як бачимо, для досягнення цього нам необхідно немало супроводжуючого коду, який не є читабельним і не є самодокументованим. Моє рішення використовує ключове слово requires, яке перед компіляцією функції перевірить вимоги до параметрів (Лістинг 11). Це рішення є більш декларативним і максимально лаконічним.

Лістинг 11

```
template</* other params ... */ class CP, class SP>
class SmartPtr
{
	// ...
public:
	operator StoredType()
	requires CP::allow
	{
		return SP::getPointer();
	}
};
```

5.6 Відсутність стратегії видалення

Хоч у цілому усі стратегії визначені коректно і серед них немає зайвих, під час безпосереднього використання цього класу я зрозумів, що, на відміну від інтелектуальних указників із стандартної бібліотеки, тут немає зручного способу надати свою функцію для видалення указника.

За безпосереднє видалення указника відповідає стратегія зберігання. Як було сказано вище, вона реалізує окремий метод, який викликається за необхідних умов з класу SmartPtr. Для того, щоб якимось чином перевизначити цей метод, необхідно переписувати усю стратегію, що призводить до копіювання коду і нівелює усі плюси даної стратегії.

Класи unique_ptr та shared_ptr також відрізняються тим, як вони дозволяють користувачам перевизначати їх стратегію видалення за замовчуванням. Ми можемо легко перевизначити її для shared_ptr, передавши об'єкт, який можна викликати при видаленні указника, як параметр конструктора. З іншої сторони, як було зазначено вище, стратегія видалення є частиною типу об'єкта unique_ptr. Користувачі повинні надавати цей тип як явний шаблонний аргумент, коли вони визначають unique_ptr. Як наслідок, користувачам unique_ptr складніше змінювати стратегію видалення.

Зв'язуючи стратегію видалення на етапі компіляції, unique_ptr уникає витрат під час виконання, пов'язаних з непрямим викликом цієї стратегії. Прив'язуючи її під час виконання, shared_ptr полегшує користувачам перевизначення стратегії [15].

У випадку класу SmartPtr, вважаю, що варто обрати варіант unique_ptr і додати ще один параметр до шаблону, який і буде стратегією видалення. Щодо стратегії за замовчуванням, це можна виконати у 2 шляхи:

1. Надавати параметр за замовчуванням у класі SmartPtr.

З цим варіантом можуть бути невизначеності, оскільки неможливо заздалегідь обрати правильну стратегію за замовчуванням для типу, який визначає стратегія зберігання. Очевидним кандидатом може бути оператор delete, але, як було вказано раніше, стратегія зберігання може мати не лише вказівник у якості типу, який вона зберігає.

2. Параметр за замовчуванням задає псевдонім шаблону SmartPtr.

Очевидно, щоб використовувати такий клас ефективно, нам необхідно для обраної множини стратегій виокремити їх у шаблон і дати псевдонім, використовуючи using. Такий псевдонім може мати лише декілька параметрів шаблону, наприклад, сам тип і його стратегія видалення, а усі інші стратегії обрані заздалегідь. Таким чином, ці псевдоніми класу можуть самі визначати

стратегію видалення за замовчуванням, враховуючи обрану стратегію зберігання.

Єдина вимога до стратегії видалення – це мати operator() з одним параметром – типом, який міститься у стратегії зберігання. Як вже було сказано, нам не достатньо, щоб він приймав указник на типовий параметр, оскільки стратегія зберігання може визначати інший тип. Такий концепт існує у стандарті і називається std::invocable (Лістинг 12).

Лістинг 12

```
template
<
	typename T,
	class DeletePolicy,
	template <class> class StoragePolicy
	/* other policies ... */
>
requires StoragePolicyConcept<StoragePolicy, T>
    and std::invocable<DeletePolicy, typename StoragePolicy<T>::StoredType>
class SmartPtr
{
	~SmartPtr()
	{
		// ...
		DP()(SP::getPointer());
	}
};
```

У концепті для стратегії (StoragePolicyConcept) зберігання ми обов'язково маємо перевіряти, чи визначає вона необхідний тип (StoredType) та чи має селектори до значення, яке у ній зберігається (Лістинг 13).

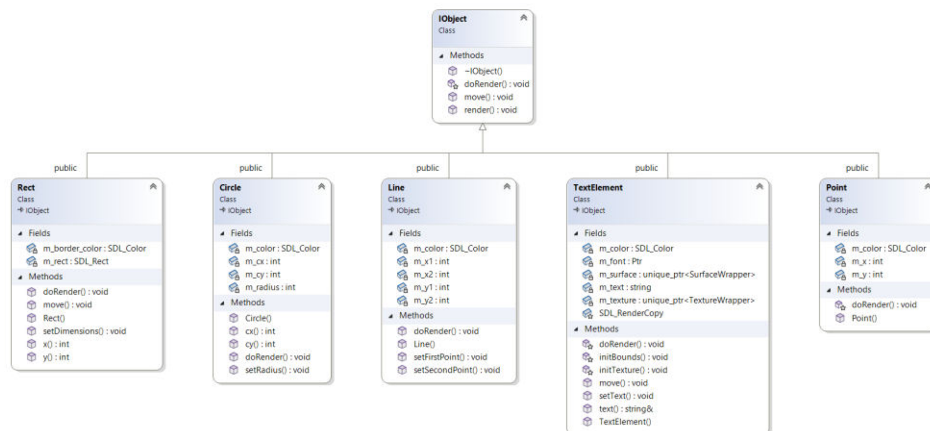
Лістинг 13

```
template<template <class> class T, typename P>
concept StoragePolicyConcept = requires(T<P> t, const T<P> ct) {
	typename T<P>::StoredType;
	typename T<P>::PointerType;
	typename T<P>::ReferenceType;
	{ct.getPointer() } -> std::same_as<typename T<P>::PointerType>;
	{t.getPointerRef() } -> std::same_as<typename T<P>::StoredType&>;
	{ct.getPointerRef() } -> std::same_as<const typename T<P>::StoredType&>;
	{ct.defaultValue()} -> std::same_as<typename T<P>::StoredType>;
};
```

6 Демонстраційна програма випробування стратегій

Демонстраційною програмою випробування стратегій служить графічний редактор. У його функціонал входить декілька простих інструментів, включаючи малювання курсором, розміщення фігур та тексту. Для відображення графічних об’єктів на екрані була використана бібліотека SDL (Simple DirectMedia Layer) [16], яка написана мовою C.

Для застосунку було необхідно використання динамічної пам’яті для динамічного поліморфізму та відображення текстур. Абстрактний клас IObject визначає віртуальний метод doRender(), який похідні класи мають



реалізувати відповідно до вимог того чи іншого графічного об’єкту (Рис. 9).

Рис. 9

Аналогічна ситуація для інструментів, кожен з яких визначає поведінку застосунку під час взаємодії користувача з програмою, реалізовуючи відповідні методи класу Tool (Рис. 10).

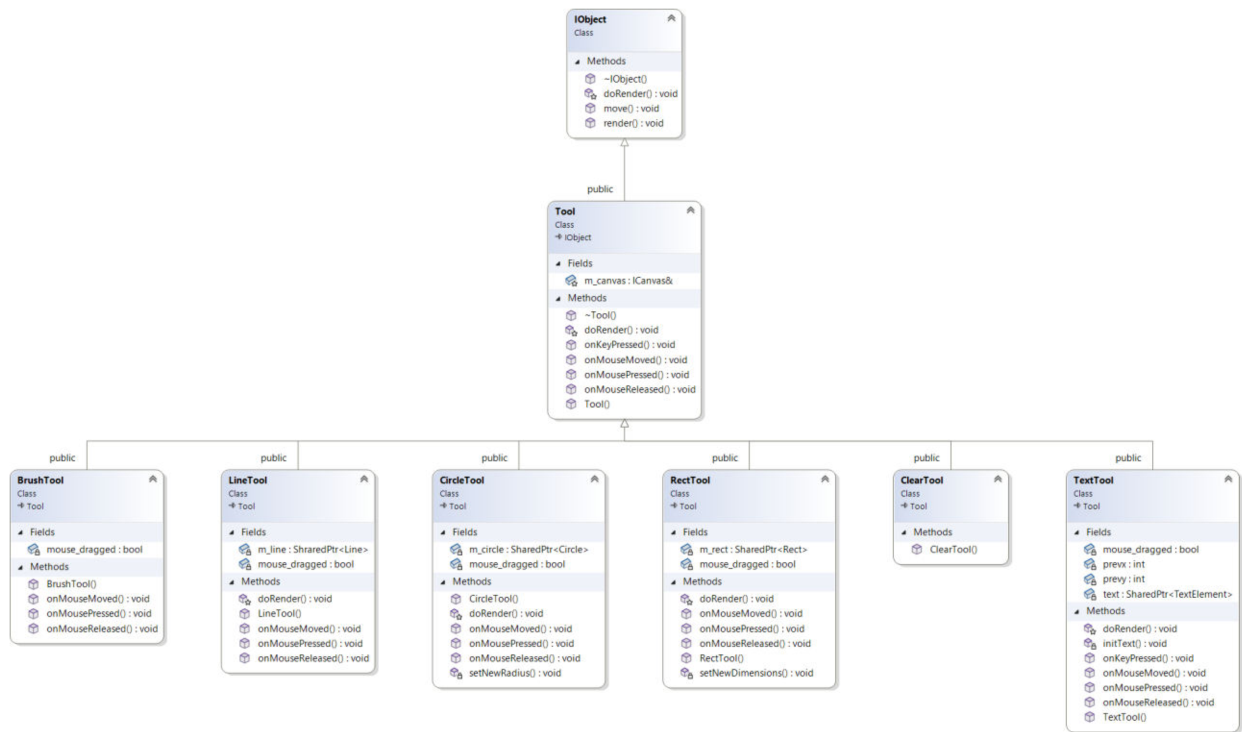


Рис. 10

Абстрактний клас ICanvas інкапсулює у собі стан полотна, а саме поточний інструмент, та має віртуальний метод renderCanvas, який відповідає за відображення усіх намальованих фігур на екрані. Похідний від нього клас Canvas визначає, як саме зберігатимуться і відображатимуться фігури (Рис. 11).

Клас Tool збергіає відсилку на полотно, до якого відноситься, і на яке відповідно додаватимуться зміни.

Класи ICanvas та Tool є похідними від IObject, оскільки мають визначати як будуть відображатись дочірні елементи через метод doRender. У випадку інструменту – це тимчасова фігура, яка з'являється у процесі малювання, але ще не додана на полотно. У випадку полотна це поточний інструмент та усі елементи, які були додані на полотно.

Клас ToolController реалізує прив'язку інструментів до клавіш і має мапу клавіш та відповідних фабрик інструментів (клас IToolFactory). Метод chooseTool викликається щоразу під час натискання будь-якої клавіші. У разі,

якщо у мапі знайдеться така клавіша, відповідна фабрика передається у метод changeTool класу ICanvas, де буде створено новий інструмент.

Створення нового інструменту з нуля необхідно тому, що похідні класи Tool можуть мати якийсь стан, і тому під час зміни інструментів, щоб уникнути будь-яких артефактів, необхідно видалити старий екземпляр.

Рис. 11

6.1 Визначення стратегій

Було використано три стратегії володіння інтелектуальних указників: одноосібний (UniquePtr), розподілений (SharedPtr) та невладний

(ObserverPtr). Ці класи було визначено як синоніми шаблону основного класу SmartPtr (Лістинг 14).

Лістинг 14

```
template <typename T, std::invocable<T*> DeletePolicy = StandardDeleter<T>>
using UniquePtr = SmartPtr<T,
        DeletePolicy,
        Unique,
        AllowConversion,
        AssertCheck,
        PointerStorage
>;

template <typename T, std::invocable<T*> DeletePolicy = StandardDeleter<T>>
using SharedPtr = SmartPtr<T,
        DeletePolicy,
        RefCounted,
        AllowConversion,
        AssertCheck,
        PointerStorage
>;

template <typename T>
using ObserverPtr = SmartPtr<T,
        NoDeletion<T>,
        Observe,
        AllowConversion,
        AssertCheck,
        PointerStorage
>;
```

6.1.1 Спільно використані стратегії

Усі три класи використовують однакові стратегії для конвертації, перевірки та зберігання, а саме: AllowConversion – дозволяє неявну конвертацію до типу указника (Лістинг 9), AssertCheck – перевірка на порожнє значення перед розіменуванням за допомогою assert (Лістинг 3), PointerStorage – стратегія зберігання, яка тримає звичайний вказівник на типовий параметр (Лістинг 15).

Лістинг 15

```
template <class T>
class PointerStorage
{
public:
        typedef T* StoredType;    // the type of the pointee_ object
        typedef T* PointerType;   // type returned by operator->
        typedef T& ReferenceType; // type returned by operator*

        PointerStorage() : pointee_(defaultValue())
        {}
```

```
	PointerStorage(const PointerStorage&) : pointee_(defaultValue())
	{}

	template <class U>
	PointerStorage(const PointerStorage<U>&) : pointee_(defaultValue())
	{}

	PointerStorage(const StoredType& p) : pointee_(p) {}

	PointerType operator->() const {
		return pointee_;
	}

	ReferenceType operator*() const {
		return *pointee_;
	}

	void swap(PointerStorage& rhs)
	{
		std::swap(pointee_, rhs.pointee_);
	}

	inline PointerType getPointer() const {
		return pointee_;
	}

	inline const StoredType& getPointerRef() const {
		return pointee_;
	}

	inline StoredType& getPointerRef() {
		return pointee_;
	}

	inline StoredType&& getPointerRRef() {
		return std::move(pointee_);
	}

	constexpr static StoredType defaultValue()
	{
		return nullptr;
	}

private:
	// Data
	StoredType pointee_;
};
```

6.1.2 Стратегії видалення

UniquePtr та SharedPtr визначають два параметри шаблону, перший – сам тип, який зберігатиметься, та другий – стратегія видалення. За замовчуванням обирається клас StandardDeleter, який викликає оператор delete (Лістинг 16).

Лістинг 16

```
template<typename T>
```

```
struct StandardDeleter {
	void operator()(T* ptr) {
		delete ptr;
	}
};
```

Натомість ObserverPtr має пусту стратегію видалення NoDeletetion, оскільки не володіє указником, який зберігає (Лістинг 17).

Лістинг 17

```
template<typename T>
struct NoDeletion {
	void operator()(T*) {}
};
```

6.1.3 Стратегії володіння

Використовуються три різні стратегії володіння. Одноосібний указник використовує стратегію Unique, яка забороняє копіювання (Лістинг 18).

Лістинг 18

```
template<class P>
class Unique {
public:
	Unique() = default;

	template<class P1>
	Unique(const Unique<P1>& rhs) = delete;

	template<class P1>
	Unique(Unique<P1>&& rhs) {}

	P clone(const P& val) = delete;

	P moveClone(P&& val) {
		return std::move(val);
	}

	bool release(const P&)
	{
		return true;
	}

	void swap(Unique& rhs)
	{}

};
```

Розподілений указник використовує стратегію RefCounted, яка працює за принципом підрахування відсилок (Лістинг 19). Потреби мати слабкі указники не було, тому стратегія додатково не веде підрахунок слабких

відсилок. Передбачено багатопоточне використання стратегії, для чого було використано клас Atomic (Лістинг 20).

Лістинг 19

```
template <class P>
class RefCounted
{
	typedef Atomic<unsigned long long> ReferenceCountType;
	typedef ReferenceCountType* ReferenceCountTypePtr;
public:
	RefCounted() : pCount_(new ReferenceCountType())
	{
		assert(pCount_ != nullptr);
	}

	RefCounted(const RefCounted& rhs)
		: pCount_(rhs.pCount_)
	{}

	template<class P1> friend class RefCounted;

	template<class P1>
	RefCounted(const RefCounted<P1>& rhs) : pCount_(rhs.pCount_) {}

	template<class P1>
	RefCounted(RefCounted<P1>&& rhs) : pCount_(rhs.pCount_) {
		rhs.pCount_ = new ReferenceCountType();
	}

	P clone(const P& val)
	{
		++*pCount_;
		return val;
	}

	P moveClone(P&& val)
	{
		return std::move(val);
	}

	bool release(const P&)
	{
		if (!--*pCount_)
		{
			delete pCount_;
			pCount_ = nullptr;
			return true;
		}
		return false;
	}

	void swap(RefCounted& rhs)
	{
		std::swap(pCount_, rhs.pCount_);
	}
private:
	ReferenceCountTypePtr pCount_;
};
```

Лістинг 20

```
template<typename T>
class Atomic {
private:
	T val = 1;
	std::mutex mutex_;
public:
	T operator++() {
		std::lock_guard<std::mutex> lg(mutex_);
		return ++val;
	}

	T operator--() {
		std::lock_guard<std::mutex> lg(mutex_);
		return --val;
	}

	operator T() {
		std::lock_guard<std::mutex> lg(mutex_);
		return val;
	}

};
```

Невладний указник використовує стратегію Observe (Лістинг 21), яка забороняє переміщення, оскільки за переміщення передбачається передача володіння на указник, а також визначає конвертувальні конструктори зі стратегій Unique та RefCounted для можливості неявного приведення будь-якого указнику до невладного.

Лістинг 21

```
template<class P>
class Observe {
public:
	Observe() {};

	Observe(const Observe&) {};

	template<typename P1>
	Observe(const Unique<P1>&) {}

	template<typename P1>
	Observe(const RefCounted<P1>&) {}

	template<class P1>
	Observe(const Observe<P1>& rhs) {};

	P clone(const P& val)
	{
		return val;
	}

	P moveClone(P&& val) = delete;

	bool release(const P&)
	{
		return true;
```

```
        }
        void swap(Observe& rhs) {}
};
```

6.2 Застосування стратегій

6.2.1 Одноосібний указник

Очевидно, одноосібний указник використовується лише тоді, коли лише один об’єкт повинен мати повне право на указник.

Клас ICanvas має таке право на інструмент, який зберігає. Відтак, фабрика, яка створюватиме інструмент, може повертати UniquePtr<Tool> замість Tool* (Лістинг 22).

Лістинг 22

```
class IToolFactory
{
public:
	virtual UniquePtr<Tool> createTool(ICanvas& canvas) = 0;
};
```

Оскільки усі похідні класи Tool мають однаковий конструктор, можемо створити шаблон для генерації сімейства таких фабрик (Лістинг 23).

Лістинг 23

```
template<typename ToolDerived>
class ToolFactory : public IToolFactory
{
public:
	UniquePtr<Tool> createTool(ICanvas& canvas) override {
		return UniquePtr<ToolDerived>(new ToolDerived(canvas));
	}
};
```

Розглянемо клас ToolController (Рис. 11). Фабрики, що зберігаються у мапі, створюються один раз при запуску програми, тому маємо зберігати їх як UniquePtr (Лістинг 24).

Лістинг 24

```
template<typename EnumType>
class ToolController {
public:
	using ToolBindingMap = std::unordered_map<EnumType, UniquePtr<IToolFactory>>;
	// ...
private:
	ToolBindingMap m_toolmap;
};

ToolController<SDL_Scancode>::ToolBindingMap initMap() {
	ToolController<SDL_Scancode>::ToolBindingMap map;
	map.insert({ SDL_SCANCODE_1,
				UniquePtr<IToolFactory>(new ToolFactory<BrushTool>()) });
	map.insert({ SDL_SCANCODE_2,
```

```
				UniquePtr<IToolFactory>(new ToolFactory<LineTool>()) });
	map.insert({ SDL_SCANCODE_3,
				UniquePtr<IToolFactory>(new ToolFactory<CircleTool>()) });
	map.insert({ SDL_SCANCODE_4,
				UniquePtr<IToolFactory>(new ToolFactory<RectTool>()) });
	map.insert({ SDL_SCANCODE_5,
				UniquePtr<IToolFactory>(new ToolFactory<TextTool>()) });
	map.insert({ SDL_SCANCODE_TAB,
				UniquePtr<IToolFactory>(new ToolFactory<ClearTool>()) });
	return map;
};
```

Клас Font виконує функцію кешування ініціалізованих шрифтів. Це можливо, тому що SDL дозволяє, створивши об'єкт TTF_Font, перевикористовувати його надалі. SDL має функцію для ініціалізації та знищення об'єктів TTF_Font, тому маємо створити відповідну стратегію видалення для цього інтелектуального вказівника (Лістинг 25).

Лістинг 25

```
struct FontDeleter {
	void operator()(TTF_Font* font) {
		if (font) {
			TTF_CloseFont(font);
		}
	}
};

using FontUniquePtr = UniquePtr<TTF_Font, FontDeleter>;

static FontUniquePtr openFont(const std::string& path, int ptsize) {
	auto font = FontUniquePtr(TTF_OpenFont(path.c_str(), ptsize));
	if (!font) {
		throw std::runtime_error("Font was not opened.");
	}
	return font;
}
```

Клас TextElement наслідує IObject та реалізує відображення тексту (Рис. 10). Для цього необхідно, маючи текст і шрифт (TTF_Font), створити поверхню (SDL_Surface) та нарешті створити текстуру (SDL_Texture) і передати у функцію, відповідальну за рендер (Лістинг 26). Для оптимізації можливий варіант кешування текстур, але доцільно буде кешувати самі об'єкти TextElement, які будуть відповідальні за звільнення усіх об'єктів з бібліотеки.

Лістинг 26

```
void initTexture(Renderer renderer) {
	m_surface = SurfaceUniquePtr(TTF_RenderText_Solid(
```

```
			m_font,
			m_text.c_str(),
			m_color
		));

		m_texture = TextureUniquePtr(SDL_CreateTextureFromSurface(
			renderer,
			m_surface
		));

		if (m_surface) {
			m_bounds.w = m_surface->w;
			m_bounds.h = m_surface->h;
		}
	}

	void doRender(Renderer renderer) override {
		if (m_surface == nullptr) {
			initTexture(renderer);
		}

		SDL_RenderCopy(
			renderer,
			m_texture,
			NULL,
			&m_bounds
		);
	}
```

Оскільки, як і у випадку з класом TTF_Font, бібліотека має власні функції для створення та видалення об’єктів SDL_Surface та SDL_Texture, маємо визначити відповідні стратегії видалення (Лістинг 27).

Лістинг 27

```
struct SurfaceDeleter {
	void operator()(SDL_Surface* surface) {
		if (surface) {
			LOGDEBUG("Freeing surface: ", surface);
			SDL_FreeSurface(surface);
		}
	}
};

using SurfaceUniquePtr = UniquePtr<SDL_Surface, SurfaceDeleter>;

struct TextureDeleter {
	void operator()(SDL_Texture* texture) {
		if (texture) {
			LOGDEBUG("Freeing texture: ", texture);
		}
	}
};

using TextureUniquePtr = UniquePtr<SDL_Texture, TextureDeleter>;
```

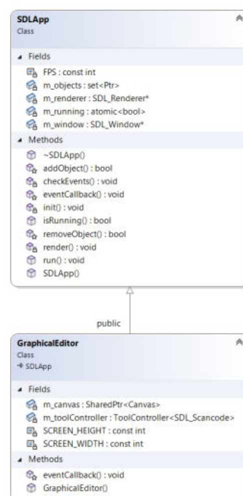
Цей варіант виглядає лаконічніше, ніж створювати окремі класи-обгортки для кожного класу з SDL, які б реалізовували ідіому RAII (Лістинг 28).

Лістинг 28

```
class SurfaceWrapper {
public:
	SurfaceWrapper(SDL_Surface* surface = nullptr) : m_surface(surface) {}

	~SurfaceWrapper() {
		if (m_surface != nullptr) {
			SDL_FreeSurface(m_surface);
		}
	}

	SDL_Surface* get() {
		return m_surface;
	}

	operator SDL_Surface* () {
		return m_surface;
	}

private:
	SDL_Surface* m_surface;
};
```

6.2.2 Розподілений указник

Усі графічні об’єкти, що мають бути відображені на даному циклі виконання, зберігаються у класі SDLApp (Рис. 12). Цей клас також виконує ініціалізацію бібліотеки SDL та відповідає за обробку подій, таких як натискання клавіш чи рух мишкою. Клас GraphicalEditor розширює цей клас



та перевизначає метод eventCallback(), який викликається для обробки подій.

Рис. 12

Список графічних об'єктів може складатись і з одноосібних указників, і це буде коректно з точки зору семантики володіння. Але є декілька мінусів такого підходу:

- Найчастіше об'єкти не будуть зберігатись лише у класі SDLApp, а й у інших класах
- Якщо ми захочемо приховати об'єкт, але не видаляти його, ми маємо повністю передати право власності об'єкту іншого класу

Тому для зберігання графічних об'єктів у класі SDLApp я використав розподілений указник (Лістинг 29).

Лістинг 29

```
class SDLApp {
	// ...
protected:
	virtual void eventCallback(SDL_Event& event) {}

	bool addObject(SharedPtr<IObject> object) {
		auto [_, inserted] = m_objects.insert(object);
		return inserted;
	}

	bool removeObject(SharedPtr<IObject> object) {
		return m_objects.erase(object);
	}

private:
	std::set<SharedPtr<IObject>> m_objects;
};
```

Прикладом одночасного володіння одним графічним об'єктом буде об'єкт класу Canvas у класі GraphicalEditor. У конструкторі класу цей об'єкт ініціалізується і передається у метод addObject, щоб автоматично відображати усі дочірні об'єкти, які додаються на полотно (Лістинг 30). Нам необхідно зберігати указник на цей об'єкт через те, що обробка усіх подій відбувається у класі GraphicalEditor і делегується цьому об'єкту.

Лістинг 30

```
class GraphicalEditor : public SDLApp {
public:
```

```
        GraphicalEditor();
protected:
        void eventCallback(SDL_Event& event) override {
                if (event.type == SDL_MOUSEBUTTONDOWN) {
                        m_canvas->onMousePressed(event.button);
                }
                // ...
        }
private:
        SharedPtr<Canvas> m_canvas;
};

GraphicalEditor::GraphicalEditor() :
        SDLApp("Graphical editor",
                SDL_WINDOWPOS_UNDEFINED,
                SDL_WINDOWPOS_UNDEFINED,
                SCREEN_WIDTH,
                SCREEN_HEIGHT),
        m_canvas(new Canvas()) // ...
{
        addObject(m_canvas);
        // ...
}
```

Наразі проект не передбачає функціоналу, коли необхідно видалити цей об’єкт з класу SDLApp, але задля гнучкості цього абстрактного класу нам необхідно мати розподілене володіння усіма графічними об’єктами.

6.2.3 Невладний указник

Невладний указник потрібний у випадках, коли ми хочемо явно вказати, що фунція, яка приймає його параметром, не отримує право володіння на нього, а отже не має турбуватись за його видалення.

Усі функції з SDL, які реалізують відображення графічних об’єктів, приймають обов’язковий параметр указник на об’єкт SDL_Renderer. Абстрактний клас IObject визначає метод render, який так само прийматиме указник на цей клас (Лістинг 31). Можливим рішенням було б замінити параметр відсилкою, але, враховуючи С-подібні сигнатури бібліотеки SDL, використовувати указник буде набагато зручніше, до того ж, ми уникаємо зайвого виклику оператору адресування (&).

Лістинг 31

```
class IObject {
public:
        using Renderer = ObserverPtr<SDL_Renderer>;
```

```
	void render(Renderer renderer) {
		if (renderer != nullptr) {
			doRender(renderer);
		}
	}
protected:
	virtual void doRender(Renderer renderer) = 0;
};
```

Досить схожа ситуація у класі Font, який є відповідальним за кешування об’єктів TTF_Font. Статична фунція getFont, яка за шляхом файлу повертає ініціалізований об’єкт шрифту, перевіряє, чи не існує у мапі уже закешованого об’єкту за таким шляхом, а якщо ні, то створює його (Лістинг 25).

Ми могли б законно використати розподілений указник у цій ситуації, але оскільки шрифт завжди зберігається у класі Font, тобто жоден інший клас не може перейняти на себе зобов’язання звільнити цей об’єкт, функція getFont має повертати невладний указник на клас TTF_Font (Лістинг 32). Тут використовується неявна конвертація з класу UniquePtr до ObserverPtr.

Лістинг 32

```
class Font {
public:
	using Ptr = ObserverPtr<TTF_Font>;

	static Ptr getFont(const std::string& path, int ptsize) {
		auto it = fonts.find(path);
		if (it != fonts.end()) {
			return it->second;
		}

		fonts.insert({ path, openFont(path, ptsize) });
		return fonts.find(path)->second;
	}
	// ...
};
```

7 Висновок

Указники, як і робота з пам'яттю, – одна з найважливіших особливостей мов програмування C та C++. З великою силою приходить велика відповідальність, і управління динамічною пам'яттю не є винятком.

«Структуроване програмування повністю витіснило з практики вживання оператора переходу goto. Так само, як оператор переходу призводить до важко контрольованих передавань керування в програмі, використання указників може спричинити виникнення несистематизованих зв'язків між даними» [17]. Це також може призвести до витоків пам'ятті, підвислих указників та інших проблем. Інтелектуальні указники вирішують цю проблему, беручи на себе повне (чи розподілене) володіння указником. Мотивація їх використання у сучасному коді очевидна.

Александреску пропонує гнучкий дизайн для побудови інтелектуального указнику як комбінації стратегій, які відповідають за різні аспекти реалізації цього класу. Через використання шаблонів код виглядає громіздким і вимагає ретельної документації для того, щоб ефективно ним користуватись. Мені вдалось уникнути цих проблем за допомогою концептів, які дозволяють обмежувати параметри шаблонів, і тим самим писати самодокументований код. Також були переглянуті підходи Александреску: застарілий код замінено сучасними конструкціями, а також додано нову стратегію видалення для більшої гнучкості шаблону.

Була розглянута і реалізація інтелектуальних указників із STL. Вона також використовує дизайн на основі стратегій, але не настільки широко, як у попередньому варіанті. Порівняно із C++03, де був присутній лише автоматичний указник (auto_ptr), у стандарт було додано достатній набір класів, який визначає 2 стратегії володіння. Їх було інтегровано у бібліотеку, дозволяючи використовувати ці класи у контейнерах STL. Моя реалізація

теж безпечна для використання у контейнерах із стандартної бібліотеки, що було продемонстровано у демонстраційному проекті.

Використовуючи підхід Александреску на основі стратегій, за рахунок підвищення рівня абстракції можна досягти максимальної гнучкості, створюючи власні стратегії і перевикористовуючи вже існуючі. З іншого боку, бібліотека вже пропонує, хоч і не настільки гнучкий, але досить зручні інтелектуальні указники.

Моя робота демонструє мотивацію для використання інтелектуальних указників. Було визначено стратегії інтелектуальних указників, використовуючи C++20, а саме: зберігання, володіння, перетворення, перевірки та видалення. Стратегії застосовано до різних типів інтелектуальних указників. Виконано рефакторинг стратегій Александреску до C++20.

Отже, замість використання сирих указників, я рекомендую використовувати інтелектуальні указники з STL для повсякденних задач, а коли необхідно модифікувати зберігання, володіння чи інші аспекти інтелектуальних указників, звертатись до дизайну Александреску. У поєднанні з сучасними техніками метапрограмування – це ідеальне рішення для широкого класу задач, які пов’язані з динамічною пам’яттю чи ресурсами у цілому.

8 Список використаних джерел

1. Alexandrescu A. Modern C++ Design: Generic Programming and Design Patterns Applied. Addison-Wesley Professional, 2001. 352 c.
2. Partial template specialization - cppreference.com. cppreference.com. URL: https://en.cppreference.com/w/cpp/language/partial_specialization
3. Type alias, alias template (since C++11) - cppreference.com. cppreference.com. URL: https://en.cppreference.com/w/cpp/language/type_alias
4. Gottschling P. Discovering Modern C++. Pearson Education, Limited, 2021. 576 c.
5. RAII - cppreference.com. cppreference.com. URL: https://en.cppreference.com/w/cpp/language/raii
6. Guy D. J. Beautiful C++: 30 Core Guidelines for Writing Clean, Safe, and Fast Code / D. J. Guy, K. Gregory., 2021. – 352 c. – (1st Edition).
7. O'Dwyer A. Mastering the C++17 STL: Make full use of the standard library components in C++17 / Arthur O'Dwyer., 2017. – 384 c.
8. Copy elision - cppreference.com. cppreference.com. URL: https://en.cppreference.com/w/cpp/language/copy_elision
9. std::auto_ptr - cppreference.com. cppreference.com. URL: https://en.cppreference.com/w/cpp/memory/auto_ptr
10. Boost.SmartPtr: The Smart Pointer Library - 1.82.0 / G. Colvin та ін. Boost C++ Libraries. URL: https://www.boost.org/doc/libs/1_82_0/libs/smart_ptr/doc/html/smart_ptr.html#shared_ptr.
11. Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. 1995. Design Patterns: Elements of Reusable Object-Oriented Software. Reading, MA: Addison-Wesley.
12. Loki | Main / HomePage. Loki | Main / HomePage. URL: https://loki-lib.sourceforge.net/

13. Stroustrup B. C++ Core Guidelines. GitHub Pages. URL: https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#t10-specify-concepts-for-all-template-arguments
14. More C++ Idioms/Move Constructor - Wikibooks, open books for an open world. Wikibooks. URL: https://en.wikibooks.org/wiki/More_C++_Idioms/Move_Constructor
15. Lippman S. C++ Primer / Stanley Lippman. – Boston: Addison Wesley Professional, 2012. – 976 c. – (5th Edition).
16. Simple DirectMedia Layer - Homepage. Simple DirectMedia Layer - Homepage. URL: https://www.libsdl.org/
17. Бублик В.В. Об’єктно-орієнтоване програмування: [Підручник] / В.В. Бублик. – К.: ІТ-книга, 2015. – 624 с.

9 Додаток

Код демонстраційного проекту додається у архіві.