Лекция 12 — Владение памятью

Содержание

”Восемь байт, помноженное на вечность — это дофига.”
— А. П. Хвастунов

RAII

Управление ресурсами в C++ — не самая простая задача, поскольку вместо привилегии делегировать свою работу сборщику мусора программист получает лишь обязанность самостоятельно следить за тем, чтобы вся выделенная память была полностью очищена. Это не говоря даже о том, что работа с динамической памятью происходит при помощи указателей, а указатели — это, как мы знаем, то ещё удовольствие.

Весь этот процесс — утомительный, трудоёмкий и безотрадный, и нам ну очень хотелось бы повесить ответственность за управление ресурсами на кого угодно другого, лишь бы не на себя. К счастью, мы в этом желании не одиноки, и из тех же соображений люди более догадливые пришли к концепции RAII — Resource Acquisition Is Initialization, дословно «*захват ресурса есть инициализация*».

RAII — это парадигма управления памятью, в основе которой лежит идея связывания жизненного цикла ресурса с жизненным циклом объекта. Суть проста: будем строить архитектуру так, чтобы выделение и освобождение памяти было привязано к объектам с автоматическим временем жизни, чтобы этими процессами управлял компилятор.

Идиома RAII гарантирует, что ресурс будет доступен любой функции, которая имеет доступ к объекту, и высвобожден в момент окончания времени жизни этого объекта или в случае, если захват ресурса привел к ошибке. Практически, её можно свести к следующему:

> 1. Каждый ресурс инкапсулируется в класс, где:
>   - конструктор захватывает ресурс или выбрасывает исключение, если захват ресурса невозможен;
>   - деструктор освобождает ресурс.

> 2. Любой ресурс всегда необходимо использовать через экземпляр RAII-класса, время жизни которого:
>   - автоматическое, либо
>   - привязано к другому объекту с автоматическим временем жизни.

RAII активно используется контейнерами стандартной библиотеки вроде `std::string` или `std::vector`, благодаря чему они не требуют ручного освобождения занятой ими памяти.

RAII исключает возможность утечки ресурсов, обеспечивает безопасную обработку исключений при работе с ними, делает код более читаемым и предотвращает неопределенное поведение. Автор находит идиому RAII прикольной.

Идиома Copy-and-swap

Правило трёх обязывает нас реализовать деструктор, конструктор копирования и оператор копирующего присваивания при работе с RAII-классами. Семантика первых двух из них в общем и целом очевидна, а вот с третьим могут возникнуть некоторые трудности.

Его наивная реализация имеет приблизительно следующий вид:

```
void operator = (const MyClass& other) {

     if (this != &other) {
          delete resource;
          resource = new T(*(other.resource));
     }

}
```

У неё есть несколько основных проблем.

Первая — проверка на самоприсваивание. Она логична и необходима, но усложняет логику достаточно простого по своей сути действия.

Вторая — дублирование кода. Почти вся логика, которую мы используем, уже была реализована где-либо ещё: в конструкторе, в деструкторе или в копирующем конструкторе, и если в данном общем примере это не критично, то в случае с более сложными объектами ситуация, скорее всего, будет не такой радужной.

Третья — отсутствие гарантии исключений. Аллокация `resource = new T(...)` необходима для возвращения класса в инвариантное состояние, однако если она провалится, инвариант будет неизбежно нарушен. Во-первых, это грустно. Во-вторых, это сложно нормально учесть. Это главная из наших проблем.

Идиома *Copy-and-swap* захватывает новый ресурс *до* того, как освободить старый, используя передачу по значению, после чего, если инициализация успешна, обменивается ресурсами с текущим объектом:

```
void operator = (MyClass other) {
    std::swap(resource, other.resource);
}
```

Разберемся чуть подробнее, что здесь происходит:

1. В оператор передаётся *временная копия* `other`.
2. `this` и `other` меняются данными, в результате чего текущий объект получает копию ресурсов присваиваемого, а временный «забирает» себе ресурсы текущего.
3. После завершения работы оператора временный объект уничтожается, и вместе с ним уничтожаются прежние ресурсы текущего объекта.

Эта идиома существует именно благодаря тому, что мы имеем дело с RAII-классами и автоматическим владением ресурсов — в противном случае пункты (1) и (3) не работали бы. Автор находит её прикольной.

Умные указатели

Выше упоминалось, что RAII достаточно активно используется в стандартной библиотеке, однако это использование выходит широко за рамки управления памятью контейнеров.

Главной реализацией этой идиомы являются определенные в заголовке `<memory>` шаблонные *умные указатели* — объекты, которые ведут себя как обычные указатели, но при этом автоматически управляют временем жизни динамически выделенного ресурса. Их можно рассматривать как примитивную реализацию сборки мусора. Для деаллокации они по умолчанию используют оператор `delete`, но при этом поддерживают и передачу пользовательских функций для удаления ресурса.

Всего существует три категории умных указателей:

> 1. `std::unique_ptr`

Это умный указатель, единолично владеющий динамическим ресурсом. Никакие другие умные указатели не могут указывать на объект, закрепленный за `unique_ptr` — иными словами, у `unique_ptr` удалён копирующий конструктор.

Он объявляется следующим образом:

```
std::unique_ptr<T> ptr(x); // x имеет тип T*
```

Его также можно создать при помощи функции `make_unique`:

```
std::unique_ptr<T> ptr = std::make_unique<T>(x);
```

`unique_ptr` освобождает память объекта, когда выходит из поля видимости. Он также имеет перегрузку для работы с массивами.

2. `std::shared_ptr`

Это умный указатель, разделяющий владение динамическим ресурсом. В отличие от `unique_ptr`, он поддерживает множественные ссылки.

Он объявляется следующим образом:

```
std::shared_ptr<T> ptr(x); // x имеет тип T*
```

Его также можно создать при помощи функции `make_shared`:

```
std::shared_ptr<T> ptr = std::make_shared<T>(x);
```

Внутри `shared_ptr` находится специальный счётчик, который отслеживает количество ссылок на объект. Получить информацию о количестве ссылок можно с помощью метода `use_count()`:

```
int main() {
     std::shared_ptr<int> ptr1(new int);
     std::shared_ptr<int> ptr2 = ptr1;
     std::cout << ptr1.use_count() << '\n'; // 2
}
```

`shared_ptr` также содержит в себе *контрольный блок*, который практически хранит метаинформацию об указателе. Контрольный блок выделяется динамически, причем

при использовании `make_shared()` он выделяется одной аллокацией сразу вместе с основным объектом, а при вызове конструктора — отдельно.

`shared_ptr` освобождает память объекта, когда на объект больше не указывает ни один указатель. До C++17 при работе с ним существуют некоторые трудности, связанные с массивами, поскольку он не имеет для этого отдельной перегрузки.

Другая проблема, возникающая при работе со `shared_ptr` — это циклические ссылки:

```
struct Poo {
     std::shared_ptr<Poo> ptr;
};

int main() {
     std::shared_ptr<Poo> ptr1(new Poo);
     std::shared_ptr<Poo> ptr2(new Poo);
     ptr1->ptr = ptr2;
     ptr2->ptr = ptr1;
}
```

Здесь по выходу из тела `main` указателть `ptr1` попытается освободить память объекта, и не сможет, ведь на него указывает ещё один указатель `ptr2->ptr`. Аналогично, указатель `ptr2` также не сможет освободить память своего объекта. Ура! Утечка!

Таких циклических зависимостей не стоит иметь.

3. `weak_ptr`

Это умный указатель, который содержит «слабую» ссылку на объект, управляемый `shared_ptr`. Это значит, что `weak_ptr` не владеет памятью этого объекта и, соответственно, не увеличивает счётчик ссылок в `shared_ptr`:

```
std::shared_ptr<int> ptr1(new int);
std::weak_ptr<int> ptr2 = ptr1;
std::cout << ptr1.use_count(); // 1
```

Отсюда сразу видно, что `weak_ptr` решает проблему циклических ссылок.
Он также моделирует *временное владение* в ситуации, когда объект должен быть доступен только если он существует и может быть удалён в любой момент. В этом случае `weak_ptr` используется для отслеживания объекта и может получить

временное владение, если будет преобразован в `shared_ptr` при помощи метода `lock()`:

```
int main() {
     std::shared_ptr<int> ptr1(new int);
     std::weak_ptr<int> ptr2 = ptr1;
     std::cout << ptr1.use_count() << '\n';
     std::shared_ptr<int> ptr3 = ptr2.lock();
     std::cout << ptr1.use_count() << '\n'
     }
```

Он также предоставляет метод `expired()`, который позволяет легко обнаруживать висячие ссылки:

```
std::shared_ptr<int> foo() {
     std::shared_ptr<int> ptr(new int);
     return ptr;
}

int main() {
     std::weak_ptr<int> wptr = foo();
     std::cout << std::boolalpha << wptr.expired(); // true
}
```

Стоит отметить, что использование `make_unique` и `make_shared` для создания объектов чаще более предпочтительно, чем объявление через `new`. Раньше это обосновывалось во многом устойчиостью к исключениям. Так, если в следующем вызове

```
foo(std::unique_ptr<A>(new A()), std::unique_ptr<B>(new B()));
```

аллокация `A()` будет успешна, а аллокация `B()` выдаст исключение, то ресурс, выделенный под `A()`, до стандарта C++17 не освобождался. Если же передавать их через `make_unique`:

```
foo(std::make_unique(A()), std::make_unique(B()));
```

то при этом создадутся временные объекты, которые валидно уничтожаются даже в случае исключений. Несмотря на упомянутые исправления в 17 стандарте, `make` - функции всё ещё остаются несколько более лаконичным и эффективным методом.

Источники

1. А. П. Хвастунов — Лекции по основам программирования на C++, 1 семестр, 2025
2. cppreference — RAII (https://en.cppreference.com/w/cpp/language/raii.html)
3. @badcasedaily1 — Концепция RAII (Resource Acquisition Is Initialization) (https://habr.com/ru/companies/otus/articles/778942/)
4. StackOverflow — What is the copy-and-swap idiom? (https://stackoverflow.com/questions/3279543/what-is-the-copy-and-swap-idiom)
5. Wikibooks — Copy-and-swap (https://en.wikibooks.org/wiki/More_C%2B%2B_Idioms/Copy-and-swap)
6. @ph_piter — Умные указатели в C++ с точки зрения новичка (https://habr.com/ru/companies/piter/articles/706866/)
7. cppreference.com — Smart pointers (https://en.cppreference.com/w/cpp/memory.html#Smart_pointers)