06.04.2024

Умные указатели

Спикер: Ибрагимов Булат Ленарович

Fast Track в Телеком, 2024

ИБРАГИМОВ БУЛАТ ЛЕНАРОВИЧ

- Преподаватель в МФТИ. Проводит курсы по C++ и алгоритмам, структурам данных
- Научный сотрудник Института Искусственного Интеллекта (AIRI)
- Работал разработчиком-исследователем в Яндекс и Сбербанк

Проблемы

Рассмотрим следующий код

```
int* Foo() {
  auto p = new int(42);
  WorkWithPointer(p);
  return p;
}
```

Какими проблемами он обладает?

Проблемы

```
int* Foo() {
  auto p = new int(42);
  WorkWithPointer(p);
  return p;
}
```

1. Неясно, кто должен освободить память после вызова `Foo`
2. Ручное управление памятью с помощью `new`
3. Утечка памяти в случае исключения в `WorkWithPointer`

Альтернатива

Чтобы избежать этих проблем, можно использовать умные указатели

```
std::unique_ptr<int> Foo() {
  auto p = std::make_unique<int>(42);
  WorkWithPointer(p.get());
  return p;
}
```

Теперь память будет освобождаться автоматически при вызове деструктора `std::unique_ptr`

std::unique_ptr

std::unique_ptr

`std::unique_ptr` - это умный указатель, который гарантирует, что только один указатель владеет объектом.

Это означает, что копирование `std::unique_ptr` запрещено, но можно передать владение с помощью перемещения.

```
std::unique_ptr<int> p1(new int(42));
auto p2 = p1;              // ошибка компиляции (копирование запрещено)
auto p3 = std::move(p1);  // перемещение владения (p1 == nullptr)
```

std::unique_ptr

- `operator*` - разыменование указателя
- `operator->` - доступ к членам объекта
- `get()` - получение сырого указателя
- `release()` - отказ от владения объектом
- `reset(ptr)` - смена владения объектом
- `swap(ptr)` - обмен объектами

std::make_unique

Чтобы избавиться от использования `new`, можно воспользоваться функцией `std::make_unique`:

```
auto p = std::make_unique<int>(42);
```

Эта шаблонная функция параметризуется типом объекта и принимает аргументы для конструктора объекта.

```
std::make_unique<A>(arg1, arg2, arg3);
// является аналогом
new A(arg1, arg2, arg3);
```

std::make_unique

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

До C++17 порядок вычисления аргументов функции не определен, поэтому возможна утечка памяти в случае исключения во время выделения памяти:

1. Вызывается `new A`
2. Вызывается `new B`, возникает исключение
3. Утечка памяти (результат `new A` не успевает быть передан в `std::unique_ptr`)

Поэтому рекомендовалось использовать `std::make_unique`:

```
Foo(std::make_unique<A>(), std::make_unique<B>());
```

std::make_unique

```
Foo(std::make_unique<A>(), std::make_unique<B>());
```

В современном C++ (C++17 и выше) подобной проблемы нет, однако использование `std::make_unique` все равно рекомендуется, так как это делает код более читаемым.

Умный указатель для массива

`std::unique_ptr` также поддерживает массивы:

```
std::unique_ptr<int[]> p(new int[10]);
p[5] = 11;
```

Главное отличие от указателя на одиночный объект - освобождает память с помощью `delete[]` и поддерживает `operator[]`.

Пользовательский удалитель

Удалитель (deleter) - это функция, которая вызывается при удалении объекта умным указателем.

Если необходимо использовать специфичесный метод удаления объекта, можно передать пользовательский удалитель:

```
std::unique_ptr<FILE, int(*)(FILE*)> p(std::fopen("file.txt", "r"), std::fclose);
```

либо

```
class Closer {
 public:
  void operator()(FILE* file) const {
    std::fclose(file);
  }
};

std::unique_ptr<FILE, Closer> p(std::fopen("file.txt", "r"));
```

std::shared_ptr

std::shared_ptr

`std::shared_ptr` - это умный указатель, который позволяет разделять владение объектом между несколькими указателями.

```
std::shared_ptr<int> p1(new int(42));
auto p2 = p1;  // p1 и p2 указывают на один и тот же объект
```

Как же `std::shared_ptr` понимает, кому и когда нужно удалить объект?

Счетчик ссылок

`std::shared_ptr` использует счетчик ссылок для отслеживания количества указателей на объект.

Когда создается новый `std::shared_ptr`, счетчик ссылок увеличивается на 1.

Когда `std::shared_ptr` уничтожается, счетчик ссылок уменьшается на 1.

Когда счетчик ссылок становится равным 0, объект удаляется.

И все это потокобезопасно!

std::make_shared

Чтобы избавиться от использования `new`, можно воспользоваться функцией `std::make_shared`:

```
auto p = std::make_shared<int>(42);
```

Однако в отличие от `std::make_unique`, следующие вызовы не эквивалентны!

```
std::shared_ptr<A>(new A(arg1, arg2, arg3));
// не эквивалентно!
std::make_shared<A>(arg1, arg2, arg3);
```

std::make_shared

```
std::shared_ptr<A>(new A(arg1, arg2, arg3));
// не эквивалентно!
std::make_shared<A>(arg1, arg2, arg3);
```

Дело в том, что первый вызов требует двух выделений памяти: одно для объекта `A`, другое для управляющего блока (счетчика ссылок) `std::shared_ptr`.

`std::make_shared` же выделяет память только один раз для объекта и управляющего блока.

Это позволяет уменьшить количество выделений памяти и улучшить производительность за счет уменьшения фрагментации памяти.

Однако у этого подхода есть свой недостаток, про который будет рассказано далее.

Пользовательский удалитель

Как и в `std::unique_ptr`, можно передать пользовательский удалитель, но в случае `std::shared_ptr` это делается ТОЛЬКО через конструктор:

```
std::shared_ptr<FILE> p(std::fopen("file.txt", "r"), std::fclose);
```

Объект удалителя при этом хранится в управляющем блоке.

Проблема циклических ссылок

Использование `std::shared_ptr` может привести к утечкам памяти в случае циклических ссылок.

```
class A {
 public:
  std::shared_ptr<A> p;
};

auto a1 = std::make_shared<A>();
auto a2 = std::make_shared<A>();
a1->p = a2;
a2->p = a1;
```

std::weak_ptr

std::weak_ptr

`std::weak_ptr` - это не владеющий умный указатель, который позволяет избежать циклических ссылок.

`std::weak_ptr` создается на основе `std::shared_ptr`:

```
auto p = std::make_shared<A>();
std::weak_ptr<A> wp = p;
// weak_ptr не является полноценным указателем:
// *wp
// wp->p
```

std::weak_ptr

Функциональность `std::weak_ptr` ограничена следующими методами:

- `use_count()` - возвращает количество `std::shared_ptr`, которые владеют объектом
- `expired()` - возвращает `true`, если объект был удален (`use_count() == 0`)
- `lock()` - возвращает `std::shared_ptr`, если объект существует, иначе `nullptr`

Решение проблемы циклических ссылок

```
class A {
 public:
  std::weak_ptr<A> p;
};

auto a1 = std::make_shared<A>();
auto a2 = std::make_shared<A>();
a1->p = a2;
a2->p = a1;
```

Как это работает?

`std::weak_ptr` хранит указатель на управляющий блок `std::shared_ptr`, но не увеличивает счетчик ссылок.

При этом дополнительно хранится счетчик слабых ссылок, который увеличивается при создании `std::weak_ptr` и уменьшается при уничтожении, так как удаление объекта не должно приводить к удалению управляющего блока в случае наличия слабых ссылок.

Проблема std::make_shared

В случае вызова `std::make_shared` память для объекта и управляющего блока выделяется единым куском.

Это означает, что в случае обнуления счетчика ссылок объекта при наличии слабых ссылок на него, память не будет освобождена!

Если объект достаточно большой, это будет означать, что память будет занята дольше, чем это необходимо.

Резюме

- `std::unique_ptr` - умный указатель, который гарантирует, что только один указатель владеет объектом
- `std::shared_ptr` - умный указатель, который позволяет разделять владение объектом между несколькими указателями
- `std::weak_ptr` - не владеющий умный указатель, который позволяет избежать циклических ссылок
- Если не требуется разделять владение объектом, стоит отдавать предпочтение `std::unique_ptr`