Умные указатели — конспект темы

unique_ptr

Возможные ошибки при работе с памятью:

- утечки памяти,
- использование неинициализированного указателя,
- повторный вызов `delete` с указателем на ранее удалённый объект,
- использование непарной версии оператора `delete`.

Умный указатель — класс, синтаксически похожий на обычный указатель: у него есть операторы `*` и `->` для доступа к объекту или ресурсу. Умный указатель управляет временем жизни объекта, своевременно удаляя его. Это упрощает работу с объектами в динамической памяти.

`unique_ptr` — умный указатель, который единолично владеет объектом в динамической памяти. В программе в один момент времени может быть только один экземпляр `unique_ptr`, ссылающийся на конкретный объект в динамической памяти. Поэтому указатель называют **уникальным**. Внутри `unique_ptr` содержится обычный указатель на объект в динамической памяти:

Чтобы создать `unique_ptr`, подключите заголовочный файл `<memory>` и передайте конструктору `unique_ptr` указатель на объект в куче:

```
#include <iostream>
#include <memory>
#include <string>

using namespace std;

struct Cat {
    Cat(const string& name, int age)
        : name_(name)
        , age_(age)  //
    {
        cout << name_ << " cat was created"s << endl;
    }
    const string& GetName() const noexcept {
```

```
        return name_;
    }
    int GetAge() const noexcept {
        return age_;
    }
    ~Cat() {
        cout << name_ << " cat was destroyed"s << endl;
    }
    void Speak() const {
        cout << "Meow!"s << endl;
    }

private:
    string name_;
    int age_;
};

int main() {
    Cat* cat = new Cat("Tom"s, 2);
    unique_ptr<Cat> p{cat};
    p->Speak();
    // Деструктор p удалит кота
}
```

`unique_ptr` всегда проинициализирован — он либо ссылается на существующий объект, либо равен `nullptr`:

```
int main() {
    unique_ptr<Cat> p;
    assert(!p);
}
```

Шаблонная функция `std::make_unique` создаёт объект в куче, передавая конструктору объекта свои аргументы, и возвращает `unique_ptr`. Ключевое слово `auto` помогает более кратко объявить переменную-указатель:

```
int main() {
    // Создаёт экземпляр класса Cat в куче и возвращает владеющий unique_ptr
    auto cat = make_unique<Cat>("Tom"s, 2);
    cat->Speak();
}
```

`unique_ptr` позволяет создать объект в куче и управлять временем его жизни, не используя операторы `new` и `delete`.

Чтобы получить значение сырого указателя на объект, есть метод `get`:

```
Cat* raw_cat_ptr = new Cat("Tom"s, 2);
unique_ptr<Cat> cat_ptr{raw_cat_ptr};

// get() возвращает сырой указатель на объект
assert(cat_ptr.get() == raw_cat_ptr);
```

`unique_ptr` предназначен только для перемещения. Вместо копирования разрешается перемещать указатель на объект от одного `unique_ptr` к другому.

После перемещения старый указатель теряет право владения объектом и начинает указывать на `nullptr`. Право владения переходит к новому указателю:

```
int main() {
    // Создаёт экземпляр Cat в куче и возвращает unique_ptr
    auto cat1 = make_unique<Cat>("Tom"s, 2);
    // Следующая строчка не скомпилируется - экземпляры unique_ptr копировать нельзя
    unique_ptr<Cat> cat2 = cat1;

    // Зато можно перемещать.
    auto cat2 = std::move(cat1);

    assert(!cat1);
    assert(cat2);
}
```

Часто значение `unique_ptr` перемещается из одной области видимости в другую.

Обычно `unique_ptr` передаётся по rvalue-ссылке.

Класс, который содержит поле типа `unique_ptr` по умолчанию становится move-only.

shared_ptr

`shared_ptr` — умный указатель, обеспечивающий совместное владение динамически выделенным ресурсом. Несколько экземпляров `shared_ptr` могут владеть одним и тем же объектом. `delete` разрушает объект, а память освобождается, когда:

- вызывается деструктор последнего экземпляра `shared_ptr`, владевший этим объектом;

- последнему экземпляру `shared_ptr`, владевшему этим объектом, было присвоено значение другого указателя. Для этого использована операция присваивания или вызов метода `reset`.

Можно создать `shared_ptr` на основе сырого указателя на объект в куче:

```
shared_ptr<Object> obj_ptr{new Object()};
```

Либо сразу создать объект в куче и получить владеющий этим объектом `shared_ptr`, вызвав функцию `make_shared`:

```
auto obj_ptr = make_shared<Object>();
```

`shared_ptr` позволяет динамически продлевать время жизни объекта в куче, пока у этого объекта есть хотя бы один владелец. Как только все владеющие объектом указатели разрушаются или перестают указывать на него, объект удаляется.

`shared_ptr` полезен, когда несколько объектов с разным временем жизни совместно используют некоторый ресурс. `shared_ptr` позволяет автоматически управлять временем жизни такого ресурса.

weak_ptr

`std::weak_ptr` — умный указатель, хранящий невладеющую, или слабую, ссылку на объект, на который ссылается `shared_ptr`.

`weak_ptr` позволяет безопасно узнать, существует ли объект, на который он ссылается, и получить к нему временный доступ, если объект всё ещё жив.

Наличие слабой ссылки не продлевает объекту жизнь. Он будет удалён, как только на него перестанут ссылаться `shared_ptr`:

```
#include <cassert>
#include <memory>

using namespace std;

struct Object {
    void DoSomething() {
    }
};
```

```
int main() {
    auto sp = make_shared<Object>();

    weak_ptr wp{sp};  // Аналог weak_ptr<Object> wp{sp};

    // Пока объект жив, слабые ссылки не устаревают
    assert(!wp.expired());
    // После обнуления shared_ptr объект удаляется
    sp.reset();

    // С этого момента все слабые ссылки становятся устаревшими
    assert(wp.expired());
}
```

`weak_ptr` не имеет операторов `*` и `->`, ведь обращаться к объекту, который может быть в любой момент удалён, небезопасно.

Чтобы безопасно обратиться к объекту, на который ссылается `weak_ptr`, получи́те временный объект `shared_ptr`. Используйте метод `lock`. Если объект удалён, метод `lock` вернёт нулевой указатель:

```
int main() {
    auto sp = make_shared<Object>();
    weak_ptr wp{sp};

    if (auto sp2 = wp.lock()) {
        // В этом блоке можно безопасно обращаться к объекту, используя сильную ссылку
        sp2->DoSomething();
    } else {
        // Сюда мы не попадём, так как на объект всё ещё ссылается sp
        assert(false);
    }

    sp.reset();
    // Объект уже разрушен, создать сильную ссылку не выйдет
    assert(!wp.lock());
}
```

`weak_ptr` умеет отслеживать, жив ли некоторый объект, и получать доступ к нему.