Проектирование больших систем на C++

Коноводов В. А.

кафедра математической кибернетики ВМК

Лекция 7
19.10.2018

std::weak_ptr

Дополнение функциональности std::shared_ptr.

- не участвует в совместном владении,
- позволяет понять, не является ли указатель висячим,
- нельзя ни разыменовать, ни проверить на nullptr.

std::weak_ptr

Дополнение функциональности std::shared_ptr.

- не участвует в совместном владении,
- позволяет понять, не является ли указатель висячим,
- нельзя ни разыменовать, ни проверить на nullptr.

```
auto sp = std::make_shared<A>();
std::weak_ptr<A> wp(sp);

// sp.use_count() == 1, wp.expired() == false;
// нельзя написать *wp, можно написать *sp

sp = nullptr;
// sp.use_count() == 0, wp.expired() == true;
```

std::weak_ptr

Разыменование происходит через преобразование к std::shared_ptr<>:

- Через функцию lock() – удаленный объект соответствует nullptr.
- Прямая конвертация – удаленный объект вызывает исключение.

```
auto sp2 = wp.lock();
// sp2 нулевой, если wp expired

std::shared_ptr<A> sp3(wp);
// exception std::bad_weak_ptr, если wp expired
```

std::weak_ptr: зачем?

Пусть есть фабрика объектов:

```
std::unique_ptr<const TObject> BuildObject(int param);
```

std::weak_ptr: зачем?

Пусть есть фабрика объектов:

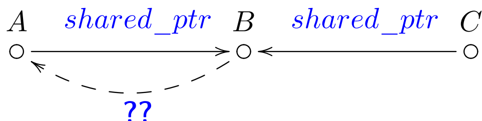
```
std::unique_ptr<const TObject> BuildObject(int param);
```

Кэш с удалением неиспользованных кэшированных значений.

```
std::shared_ptr<const TObject> FastBuildObject(int param) {
    static std::unordered_map<int,
                              std::weak_ptr<const TObject>>
                             cache;
    auto objPtr = cache[param].lock();
    if (!objPtr) {
        objPtr = BuildObject(param);
        cache[param] = objPtr;
    }
    return objPtr;
}
```

Предупреждение циклов std::shared_ptr

Рассмотрим такую структуру совместного владения:



Каким должен быть указатель?

- raw pointer
- shared_ptr
- weak_ptr

Счетчик слабых ссылок

Время между уничтожением последнего shared_ptr и weak_ptr значительно.

```
auto sp = std::make_shared<A>();
// создание shared_ptr, weak_ptr
// ..
// удаление всех shared_ptr
// ... (!)
// удаление последнего weak_ptr
```

Или через new?

std::make_shared

Где тут потенциальная проблема?

```
Do(std::shared_ptr<A>(new A), getItem());
```

std::make_shared

Где тут потенциальная проблема?

```
Do(std::shared_ptr<A>(new A), getItem());
```

Исправляем:

```
Do(std::make_shared<A>(), getItem());
```

+ единовременное выделение памяти под объект и счетчик ссылок

std::make_shared

Где тут потенциальная проблема?

```
Do(std::shared_ptr<A>(new A), getItem());
```

Исправляем:

```
Do(std::make_shared<A>(), getItem());
```

+ единовременное выделение памяти под объект и счетчик ссылок
Но в std::make_shared нельзя использовать custom-удалитель:

```
std::shared_ptr<A> sp(new A, customDeleter);
```

Как тогда исправить

```
Do(std::shared_ptr<A>(new A, customDeleter), getItem());
```

чтобы было безопасно?

std::make_shared

Где тут потенциальная проблема?

```
Do(std::shared_ptr<A>(new A), getItem());
```

Исправляем:

```
Do(std::make_shared<A>(), getItem());
```

+ единовременное выделение памяти под объект и счетчик ссылок
Но в std::make_shared нельзя использовать custom-удалитель:

```
std::shared_ptr<A> sp(new A, customDeleter);
```

Как тогда исправить

```
Do(std::shared_ptr<A>(new A, customDeleter), getItem());
```

чтобы было безопасно?

```
std::shared_ptr<A> sp(new A, customDeleter);
Do(sp, getItem());
```

Чего не хватает теперь для оптимальности?

std::make_shared

Применение make-функций может быть плохой идеей для объектов типов с перегруженными `operator new` и `operator delete`.
Тут может помочь

`std::allocate_shared<>`

и собственный аллокатор.

Другие умные указатели

- `intrusive_ptr` — облегченная версия shared_ptr для классов, имеющих встроенные механизмы подсчёта ссылок.
- `scoped_ptr` — аналог `const auto_ptr` с запрещенными конструктором копирования и оператором присваивания.

Идиома CRTP

The curiously recurring template pattern. Класс отнаследован от шаблонного класса, в котором наследник – аргумент шаблона:

```
template <typename Derived>
class CuriousBase {
    // ...
};
class Curious : public CuriousBase<Curious> {
    // ..
};
```

Coplien, James O. (February 1995).

Идиома CRTP

The curiously recurring template pattern. Класс отнаследован от шаблонного класса, в котором наследник – аргумент шаблона:

```
template <typename Derived>
class CuriousBase {
    // ...
};
class Curious : public CuriousBase<Curious> {
    // ..
};
```

Coplien, James O. (February 1995).

Пример: `enable_shared_from_this`.

Идиома CRTP

Ограничиваем число объектов класса.

```
#include <stdexcept>
template <typename T, size_t maxN>
class LimitedInstances {
    static size_t counter;
  protected:
    LimitedInstances() {
        if (counter >= maxN) {
            throw std::logic_error("too many instances");
        }
        ++counter;
    }
    ~LimitedInstances() {
        --counter;
    }
};

template <typename T, size_t maxN>
size_t LimitedInstances<T, maxN>::counter(0);
```

Идиома CRTP

```
class oneInst: public LimitedInstances<oneInst, 1> {};
class twoInst: public LimitedInstances<twoInst, 2> {};

int main() {
    oneInst obj;
    try {
        oneInst();
    } catch (std::logic_error &e) {
        std::cerr << "Caught: " << e.what() << std::endl;
    }

    twoInst obj1;
    twoInst obj2;
    try {
        twoInst();
    } catch (std::logic_error &e) {
        std::cerr << "Caught: " << e.what() << std::endl;
    }
};
```

Идиома CRTP

- Часто заменяют динамический полиморфизм через статический: в базовом классе вызываем методы класса, которым он параметризован при инстанциировании

Идиома PImpl

Pointer to implementation.
Метод, при котором члены-данные класса заменяются указателем на класс реализации с этими данными.
`a.h`:

```
#include "myitems.h"
class A {
    TMyItem item1, item2;
  public:
    A();
    // ...
};
```

Break compilation dependencies!

a.h:

```
class A {
    struct Impl;
    Impl *pImpl;
  public:
    A();
    // ...
};
```

---

a.cpp:

```
#include "a.h"
#include "myitems.h"
struct A::Impl {
    TMyItem item1, item2;
};

A::A() : pImpl(new Impl) {}
A::~A() {delete pImpl;}
```

Идиома PImpl: C++11

a.h:

```
class A {
    struct Impl;
    std::unique_ptr<Impl> pImpl;
  public:
    A();
    // ...
};
```

---

a.cpp:

```
#include "a.h"
#include "myitems.h"
struct A::Impl {
    TMyItem item1, item2;
};

A::A() : pImpl(std::make_unique<A::Impl>()) {}
```

Идиома PImpl: C++11

a.h:

```
class A {
    struct Impl;
    std::unique_ptr<Impl> pImpl;
  public:
    A();
    // ...
};
```

a.cpp:

```
#include "a.h"
#include "myitems.h"
struct A::Impl {
    TMyItem item1, item2;
};

A::A() : pImpl(std::make_unique<A::Impl>()) {}
```

main.cpp:

```
#include "a.h"
A a;  // !error
```

Идиома PImpl

Нужно обеспечить полноту в точке уничтожения `std::unique_ptr<A::Impl>`.

a.h:

```
class A {
    struct Impl;
    std::unique_ptr<Impl> pImpl;
  public:
    A();
    ~A();
    // ...
};
```

---

a.cpp:

```
~A::A() = default;
```

Идиома PImpl

Нужны перемещающие функции:

a.h:

```
class A {
    struct Impl;
    std::unique_ptr<Impl> pImpl;
  public:
    A();
    ~A();
    A(A&& other) = default;
    A& operator=(const A& other) = default;
    // ...
};
```

И снова та же проблема!

Идиома PImpl

Объявляем в заголовочном файле, реализуем в файле реализации:

a.h:

```
class A {
    struct Impl;
    std::unique_ptr<Impl> pImpl;
  public:
    A();
    ~A();
    A(A&& other);
    A& operator=(const A& other);
    // ...
};
```

---

a.cpp:

```
A::A(A&& other) = default;
A& A::operator=(const A& other) = default;
```

Идиома PImpl

Потребуются копирующие операции:
a.h:

```
class A {
    struct Impl;
    std::unique_ptr<Impl> pImpl;
  public:
    A();
    ~A();
    A(A&& other);
    A& operator=(const A& other);
    A(const A& other);
    A& operator=(const A& other);
    // ...
};
```

Идиома PImpl

a.cpp:

```
A::A(const A& other) : pImpl(nullptr) {
    if (other.pImpl) {
        pImpl = std::make_unique<Impl>(*other.Impl);
    }
}

A& A::operator=(const A& other) {
    if (!other.pImpl) {
        pImpl.reset();
    } else if (!pImpl) {
        pImpl = std::make_unique<Impl>(*other.Impl);
    } else {
        *pImpl = *other.pImpl;
    }
    return *this;
}
```

Идиома PImpl

a.cpp:

```
A::A(const A& other) : pImpl(nullptr) {
    if (other.pImpl) {
        pImpl = std::make_unique<Impl>(*other.Impl);
    }
}

A& A::operator=(const A& other) {
    if (!other.pImpl) {
        pImpl.reset();
    } else if (!pImpl) {
        pImpl = std::make_unique<Impl>(*other.Impl);
    } else {
        *pImpl = *other.pImpl;
    }
    return *this;
}
```

В случае `std::shared_ptr` всё проще!

Паттерны проектирования

- Способ построения кода для решения часто встречающихся проблем проектирования
- successful stories
- Готовые абстракции для решения классов проблем + унификация деталей и названий
- Не нужно их употреблять везде, не нужно им строго следовать

Э. Гамма, Р. Хелм, Р. Джонсон, Дж. Влиссидес. Приемы объектно-ориентированного проектирования. Паттерны проектирования.

Паттерны проектирования

- Способ построения кода для решения часто встречающихся проблем проектирования
- successful stories
- Готовые абстракции для решения классов проблем + унификация деталей и названий
- Не нужно их употреблять везде, не нужно им строго следовать

Э. Гамма, Р. Хелм, Р. Джонсон, Дж. Влиссидес. Приемы объектно-ориентированного проектирования. Паттерны проектирования. [contains a lot of «ancient» C++ code]

Пример: паттерн Bridge

Цель: разделить абстракцию и реализацию на две отдельные иерархии классов так, что их можно изменять независимо друг от друга.

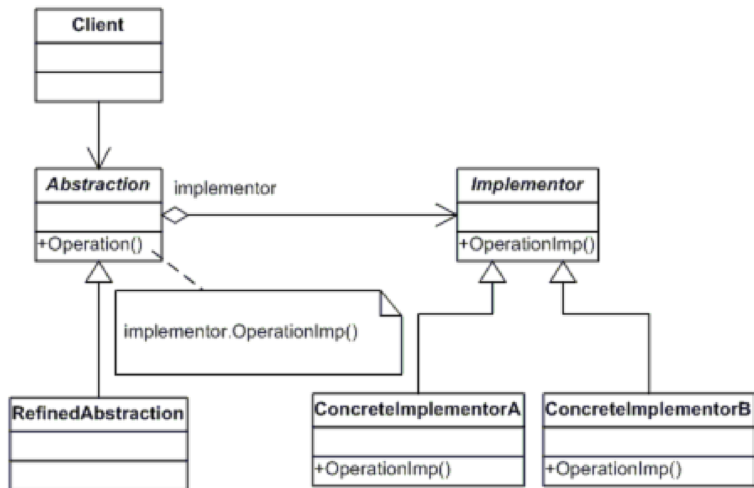
Почему не наследование: наследование жестко привязывает реализацию к абстракции. Это затрудняет расширение и повторное использование абстракции и ее реализации.

Пример: паттерн Bridge

Цель: разделить абстракцию и реализацию на две отдельные иерархии классов так, что их можно изменять независимо друг от друга.
Почему не наследование: наследование жестко привязывает реализацию к абстракции. Это затрудняет расширение и повторное использование абстракции и ее реализации.

Первая иерархия определяет интерфейс абстракции, доступный пользователю. Основной класс содержит указатель на реализацию `pimpl`, который используется для перенаправления пользовательских запросов в неё.
Все детали реализации, связанные с какими-либо особенностями находятся во второй иерархии.

Пример: паттерн Bridge



Abstraction перенаправляет объекту Implementation запросы клиента.

Пример: паттерн Bridge

Когда: когда нужно часто изменять реализацию какого-нибудь метода с сохранением API

Когда: когда используется постоянно изменяющаяся внешняя библиотека

Когда: когда нужно добиться разделения ответственности между классами

Пример: паттерн Bridge

В чем отличие от PIMPL?

- PIMPL — способ скрыть реализацию, в основном для того, чтобы убрать зависимости
- Bridge — поддержка множественных реализаций, а в PIMPL обычно не изменяется реализация, это отдельно компилируемый класс
- PIMPL — идиома проектирования на уровне файлов с кодом, Bridge — паттерн объектно-ориентированного проектирования.