Проектирование больших систем на C++

Коноводов В. А.

кафедра математической кибернетики ВМК
vkonovodov@gmail.com

Лекция 6
23.10.2019

std::shared_ptr

Что напечатает программа?

```
#include <iostream>
struct TObject {
    TObject() { std::cout << "construct" << std::endl; }
    ~TObject() { std::cout << "destruct" << std::endl; }
    std::shared_ptr<TObject> link;
};

void func() {
    std::shared_ptr<TObject> sp1 = std::make_shared<TObject>();
    std::shared_ptr<TObject> sp2 = std::make_shared<TObject>();
    sp1->link = sp2;
    sp2->link = sp1;
}

int main() {
    func();
}
```

std::shared_ptr

Что напечатает программа?

```
#include <iostream>
struct TObject {
    TObject() { std::cout << "construct" << std::endl; }
    ~TObject() { std::cout << "destruct" << std::endl; }
    std::shared_ptr<TObject> link;
};

void func() {
    std::shared_ptr<TObject> sp1 = std::make_shared<TObject>();
    std::shared_ptr<TObject> sp2 = std::make_shared<TObject>();
    sp1->link = sp2;
    sp2->link = sp1;
}

int main() {
    func();
}
```

```
construct construct
```

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:

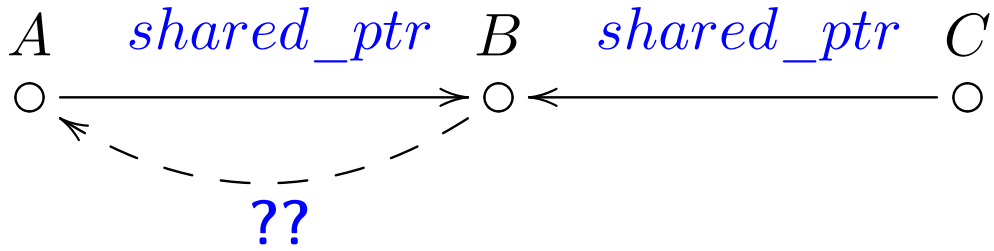
```
std::unique_ptr<const TObject> BuildObject(int param);
```

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?

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.

Задача с теста

```
class A;
using VecPtr = std::vector<std::shared_ptr<A>>;

class A {
  public:
    void process(VecPtr& done) {
        done.emplace_back(this);
    }
};

int main() {
    std::shared_ptr<A> p(new A);
    VecPtr done;
    p->process(done);
    return 0;
}
```

Задача с теста

```
class A;
using VecPtr = std::vector<std::shared_ptr<A>>;

class A {
  public:
    void process(VecPtr& done) {
        done.emplace_back(this);
    }
};

int main() {
    std::shared_ptr<A> p(new A);
    VecPtr done;
    p->process(done);
    return 0;
}
```

Конструируемый shared_ptr создает новый управляющий блок.

Задача с теста

```
class A;
using VecPtr = std::vector<std::shared_ptr<A>>;

class A : public std::enable_shared_from_this<A>{
  public:
    void process(VecPtr& done) {
        done.emplace_back(shared_from_this());
    }
};

int main() {
    std::shared_ptr<A> p(new A);
    VecPtr done;
    p->process(done);
    return 0;
}
```

enable_shared_from_this

Вариант реализации:

```
template<typename T>
class enable_shared_from_this {
    std::weak_ptr<T> wp;
public:
    std::shared_ptr<T> shared_from_this() {
        std::shared_ptr<T> p( wp );
        return p;
    }
};
```

enable_shared_from_this

Вариант реализации:

```
template<typename T>
class enable_shared_from_this {
    std::weak_ptr<T> wp;
public:
    std::shared_ptr<T> shared_from_this() {
        std::shared_ptr<T> p( wp );
        return p;
    }
};
```

- Что делать, если нужно в конструкторе?

enable_shared_from_this

В конструкторе:

```
struct B: public enable_shared_from_this<B> {
    B() {
        cout << shared_from_this() << endl;
    }
};
```

enable_shared_from_this

В конструкторе:

```
struct B: public enable_shared_from_this<B> {
    B() {
        cout << shared_from_this() << endl;
    }
};
```

private-конструктор

```
class A: public enable_shared_from_this<A> {
  public:
    template <typename... Ts>
    static std::shared_ptr<A> create(Ts&&... params) {
        // вызывает private-конструктор
    }
};
```

Но вообще лучше избегать вызов shared_from_this из конструкторов и деструкторов.

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

```
template <typename Derived>
class CuriousBase {
  public:
    void f(int x) { static_cast<Derived*>(this)->f(x);}
  protected:
    int a;
};
class Curious : public CuriousBase<Curious> {
  public:
    void f(int x) { y += x; }
};

// ...
CuriousBase<Curious>* b = ...;
b->f(10); // без механизма виртуальных функций!
```

Идиома CRTP

Для любого класса, использующего оператор равенства, сделать автоматически оператор неравенства (как инверсию первого):

```
template <typename D>
struct not_eq {
    bool operator != (const D& rhs) const {
        return !static_cast<const D*>(this)->operator==(rhs);
    }
};

class C: public not_eq<C> {
  public:
    bool operator == (const C& rhs) const {
        // ...
    }
};
```

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
- Техника реализации, при которой общая функциональность предоставляется нескольким производным базовым классам, и каждый расширяет и настраивает интерфейс шаблона базового класса.

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=(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=(A&& other);
    // ...
};
```

---

a.cpp:

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