Проектирование больших систем на C++

Коноводов В. А.

кафедра математической кибернетики ВМК
vkonovodov@gmail.com

Лекция 7
31.10.2020

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.
- `copy_ptr`
- `linked_ptr`
- ...

Владение памятью

Что такое правильно спроектированное владение памятью?

Владение памятью

Что такое правильно спроектированное владение памятью?

- в каждой точке программы понятно, кто каким объектом владеет?

Владение памятью

Что такое правильно спроектированное владение памятью?

- в каждой точке программы понятно, кто каким объектом владеет?
- в каждой точке программы понятно, кто владеет данным объектом или что владение не меняется?

Владение памятью

Что такое правильно спроектированное владение памятью?

Ф.Пикус определяет так:

- Если некоторая функция или класс никак не изменяет владение памятью, то это понятно каждому клиенту и автору
- Если некоторая функция или класс принимает уникальное владение объектами, то это должно быть понятно клиенту
- Если некоторая функция или класс разделяет владение объектом, то это должно быть понятно клиенту
- Для любого созданного объекта в любой точке использования понятно, должен код удалить объект или нет

Владение памятью

Что такое неправильно спроектированное владение памятью?

- Нужна дополнительная информация. Например, кто владеет возвращаемым объектом в этом коде:

```
TObjectFactory factory;
TObject* p = factory.MakeObject();
```

Владение памятью

Что такое неправильно спроектированное владение памятью?

- Нужна дополнительная информация. Например, кто владеет возвращаемым объектом в этом коде:

```
TObjectFactory factory;
TObject* p = factory.MakeObject();
```

- А что с владением здесь:

```
TObject* p1 = Process(p);
```

Владение памятью

Что такое неправильно спроектированное владение памятью?

- Нужна дополнительная информация. Например, кто владеет возвращаемым объектом в этом коде:

```
TObjectFactory factory;
TObject* p = factory.MakeObject();
```

- А что с владением здесь:

```
TObject* p1 = Process(p);
```

- Функция принимает владение вектором, а зачем:

```
void Double(std::shared_ptr<std::vector<int>> v) {
    if (!v) return;
    for (auto& x: *v) x *= 2;
}
...
std::shared_ptr<std::vector<int>> sv(...);
Double(sv);
```

Владение памятью

В терминах синтаксиса умных указателей можно говорить про разные стратегии владения, передачи владения, преобразование владения.

Какой тип владения самый распространенный?

Владение памятью: невладение

```
void Process(TObject *p); // я не буду удалять p
void Do(TObject& p);      // я тоже
```

Невладеющие указатели, ссылки.

not null

У указателей есть `nullptr`. И это значение всегда нужно проверять. CppCoreGuidelines:

I.12: Declare a pointer that must not be null as `not_null`

Reason To help avoid dereferencing `nullptr` errors. To improve performance by avoiding redundant checks for `nullptr`.

Example

```
int length(const char* p);            // it is not clear whether length(nullptr) is valid

length(nullptr);                      // OK?

int length(not_null<const char*> p);  // better: we can assume that p cannot be nullptr

int length(const char* p);            // we must assume that p can be nullptr
```

By stating the intent in source, implementers and tools can provide better diagnostics, such as finding some classes of errors through static analysis, and perform optimizations, such as removing branches and null tests.

not null

- CompileTime: код не компилируется, если объекту `not_null<T*>` присвоить `nullptr`.
- RunTime: поведение программы переопределяется (исключение, игнор, terminate, ...)
- Это расширение языка

Владение памятью: уникальное владение

```
void f() {
    TObject o;
    Do(o);
    Process(o);
}
```

- Локальные переменные
- `std::unique_ptr`

Владение памятью: уникальное владение

Обычное использование — создание объектов фабриками:

```
std::unique_ptr<TObject> p(ObjectFactory());
```

Что должна возвращать `ObjectFactory`?

Владение памятью: уникальное владение

Обычное использование — создание объектов фабриками:

```
std::unique_ptr<TObject> p(ObjectFactory());
```

Что должна возвращать ObjectFactory?
Мы хотим, чтобы клиент этой фабрики был вынужден использовать её так, чтобы принимать владение.

```
std::unique_ptr<TObject> ObjectFactory() {
    return std::unique_ptr<TObject>(...);
}

void Process(TObject* p);
// ...
std::unique_ptr<TObject> p(ObjectFactory()); // ok
Process(ObjectFactory());                    // compile err
Process(p.get());                            // ok
```

Владение памятью: разделяемое владение

- Несколько сущностей владеют объектом на равных основаниях
- `std::shared_ptr`
- Совместное владение быть избыточно
- Если функция принимает `std::shared_ptr` в качестве параметра, она извещает клиента о том, что намеревается принять частичное владение
- Монопольное владение предпочтительнее — его проще отследить и оно более эффективно.

Где проблемы?

```
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;
}
```

Где проблемы?

```
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.

Решение

```
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 y;
};
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!