

академия
больших
данных

mail.ru
group

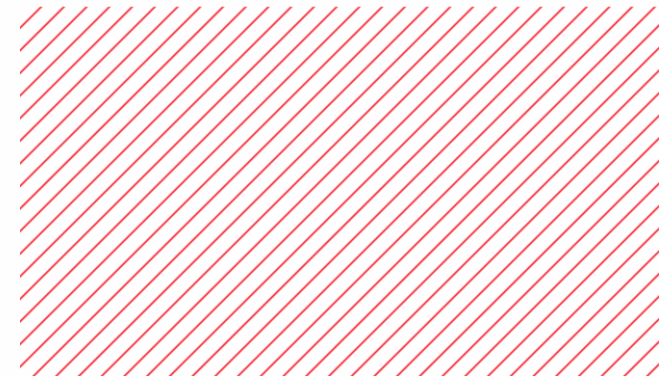


Smart pointers

Лекция 7

Гуров Роман

Advanced C++





Проблема

Нужно всегда следить за сырыми указателями
и не забывать вызывать **delete**

```
T* f() {  
    T* a = new T;  
    g();  
    return a;  
}  
// ...  
T* obj = f();  
h(obj);  
delete obj;
```

А если в *g* или *h* бросится исключение?

Проблема

```
struct C {  
    int* p1;  
    int* p2;  
    C() {  
        p1 = new int;  
        p2 = new int;  
    }  
    ~C() {  
        delete p1;  
        delete p2;  
    }  
};
```

Если при аллокации `p2` вылетит исключение, то `p1` не будет удален



Проблема

```
struct C {  
    int* p1;  
    int* p2;  
    C() {  
        p1 = new int;  
        try {  
            p2 = new int;  
        } catch (...) {  
            delete p1;  
            throw;  
        }  
    }  
    ~C() {  
        delete p1;  
        delete p2;  
    }  
};
```

Обрабатывать руками каждый случай тяжело,
и код становится нечитаемым



auto_ptr

В старых версиях стандарта проблема решалась классом `auto_ptr`:

```
template <typename T>
struct auto_ptr {
    T* ptr;
    explicit auto_ptr(T* ptr) : ptr(ptr) {};
    ~auto_ptr() {
        delete ptr;
    }
};
```

Но что делать с копированием такого класса?



std::auto_ptr

В старых версиях стандарта проблема решалась классом `auto_ptr`:

```
template <typename T>
struct auto_ptr {
    T* ptr;
    explicit auto_ptr(T* ptr) : ptr(ptr) {};
    ~auto_ptr() { delete ptr; }

    auto_ptr(auto_ptr& other) : ptr(other.ptr) {
        other.ptr = NULL;
    }
};
```

Вместо копирования `auto_ptr` делает перемещение,
что может привести к неявным ошибкам в коде



std::unique_ptr

В C++11 добавили `unique_ptr`,
использующий преимущества move-семантики:

```
unique_ptr(const unique_ptr& other) = delete;  
unique_ptr& operator=(const unique_ptr& other) = delete;  
  
unique_ptr(unique_ptr&& other);  
unique_ptr& operator=(unique_ptr&& other);
```

Теперь указатель не получится случайно переместить и инвалидировать



std::unique_ptr

У умных указателей есть перегрузки операторов, позволяющие работать с ними, как с сырыми указателями

```
T& operator* () {  
    return *ptr;  
}
```

```
T* operator-> () {  
    return ptr;  
}
```




std::unique_ptr

Что делать, если хотим передать указатель на массив,
который надо удалять с помощью `delete []`?

Для этого есть специальная перегрузка:

```
std::unique_ptr<int[]> a(new int[10000]);
```



std::unique_ptr и исключения

Можно исправить прошлый пример с помощью `unique_ptr`:

```
struct C {  
    std::unique_ptr<int> p1;  
    std::unique_ptr<int> p2;  
    C() : p1(new int), p2(new int) {}  
};
```

Теперь, если при аллокации объектов бросится исключение,
память всегда корректно очистится



std::shared_ptr

А если всё-таки хотим уметь копировать умные указатели?

Все упирается в реализацию такого поведения деструктора:

```
~shared_ptr() {  
    if (/* мы последняя копия */) {  
        delete ptr;  
    }  
}
```




std::shared_ptr

Нужно как-то считать количество копий указателя, чтобы вызвать **delete** только в последней

Значит, должен быть какой-то блок данных, который не хранится непосредственно внутри указателя, общий для всех копий

```
struct ControlBlock {  
    T* ptr;  
    size_t ref_count;  
};
```

Количество shared_ptr,
которые владеют данными





std::shared_ptr

```
explicit shared_ptr(T* ptr) {  
    cb = new ControlBlock;  
    cb->ptr = ptr;  
    cb->ref_count = 1;  
};
```

```
~shared_ptr() {  
    if (cb->ref_count == 1) {  
        delete cb->ptr;  
        delete cb;  
    } else {  
        --cb->ref_count;  
    }  
}
```

```
shared_ptr(const shared_ptr& other) : cb(other.cb) {  
    ++cb->ref_count;  
}
```

Теперь можем реализовать упрощённый
аналог `shared_ptr`

std::shared_ptr

```
explicit shared_ptr(T* ptr) {  
    cb = new ControlBlock;  
    cb->ptr = ptr;  
    cb->ref_count = 1;  
};
```

← А что, если тут бросится исключение?

Тогда при таком использовании
произойдет утечка памяти:

```
shared_ptr<int> (new int)
```

Поэтому правильная реализация конструктора должна вызывать
delete ptr при бросании исключения



std::make_shared, std::make_unique

Рассмотрим данный вызов:

```
f(g(), shared_ptr<int>(new int));
```

Стандарт C++11 не имеет строгих требований на порядок обработки аргументов при вызове функции, и может быть так, что сначала вызовется **new**, потом `g()`, и только в конце конструктор `shared_ptr`

Если `g()` бросит исключение, то произойдет утечка памяти



std::make_shared, std::make_unique

Попробуем избежать явных вызовов **new** при помощи отдельной функции, которая аллоцирует объект за нас

```
shared_ptr<int> sp = make_shared<int>(5);
```

```
template <class T, class... Args>  
shared_ptr<T> make_shared(Args&&... args) {  
  
}
```




std::make_shared, std::make_unique

Попробуем избежать явных вызовов **new** при помощи отдельной функции, которая аллоцирует объект за нас

```
shared_ptr<int> sp = make_shared<int>(5);
```

```
template <class T, class... Args>
shared_ptr<T> make_shared(Args&&... args) {
    return shared_ptr<T>(new T(std::forward<Args>(args)...));
}
```



std::make_shared, std::make_unique

Аналогично реализуется и make_unique

```
template <class T, class... Args>
unique_ptr<T> make_unique(Args&&... args) {
    return unique_ptr<T>(new T(std::forward<Args>(args)...));
}
```



std::make_shared

На самом деле, часто применяется оптимизация `make_shared`, вызывающая оператор `new` только один раз, аллоцируя память сразу для `ControlBlock` и `T`



std::allocate_shared, std::allocate_unique

Бывает нужно, чтобы аллокация происходила на нашем аллокаторе

```
template <class T, class Alloc, class... Args>  
shared_ptr<T> allocate_shared(const Alloc& alloc, Args&&... args);
```

```
template <class T, class Alloc, class... Args>  
unique_ptr<T> allocate_unique(const Alloc& alloc, Args&&... args);
```

Проблема с std::shared_ptr

Может показаться, что `shared_ptr` позволяет избежать любых утечек памяти, но это не так.

```
struct B;  
struct A {  
    shared_ptr<B> b;  
};  
struct B {  
    shared_ptr<A> a;  
};  
  
shared_ptr<A> a(new A);  
shared_ptr<B> b(new B);  
a->b = b;  
b->a = a;
```

`shared_ptr` не может сам узнать, что образовалась замкнутость



std::weak_ptr

Есть дополнительный вид указателя, который не владеет данными

```
struct B;  
struct A {  
    shared_ptr<B> b;  
};  
struct B {  
    weak_ptr<A> a;  
};  
  
shared_ptr<A> a(new A);  
shared_ptr<B> b(new B);  
a->b = b;  
b->a = a;
```

Так утечка памяти не произойдет

std::weak_ptr

`weak_ptr` позволяет спросить, жив объект или нет,
и создать `shared_ptr`, владеющий им

`bool expired();`



Проверяет, не уничтожен
ли объект

`shared_ptr<T> lock();`



Создает `shared_ptr` на
объект



Deleter'ы

Не всегда выделенный указатель нужно удалять с помощью **delete**

```
FILE* file = fopen("filename", "r");  
fclose(file);
```




Deleter'ы

Умные указатели в C++11 поддерживают передачу кастомных delete'ов, которые будут вызваны при уничтожении объекта

```
struct FileCloser {  
    void operator()(FILE* file) {  
        fclose(file);  
    }  
};  
  
std::shared_ptr<FILE> file(fopen("filename", "r"), FileCloser());
```

По умолчанию используется функтор `std::default_delete`, который просто вызывает **delete**

std::enable_shared_from_this

Иногда может возникнуть желание уметь получать внутри какого-то класса `shared_ptr` на самого себя

```
shared_ptr<Foo> shared_from_this() {  
    return shared_ptr<Foo>(this);  
}
```

Так сделать не получится.
Нами уже владеет указатель
снаружи

Для этого существует специальный класс:

```
template <typename T>  
class enable_shared_from_this;
```



std::enable_shared_from_this

Нужно унаследовать свой класс от него

```
class Foo : public std::enable_shared_from_this<Foo> {};
```

Теперь при создании `shared_ptr` `foo` на класс `Foo`, в него запишется `weak_ptr`, связанный с `foo`, позволяющий вернуть валидную копию в любой момент

```
std::shared_ptr<Foo> foo = std::make_shared<Foo>();  
foo->shared_from_this();
```

Тут вернется честная копия `foo`