Проектирование больших систем на C++

Коноводов В. А.

кафедра математической кибернетики ВМК

Лекция 7
20.10.2017

std::shared_ptr

- Реализует семантику совместного владения.
- Использует метод подсчета ссылок. Счетчик ссылок хранится в динамически выделяемой памяти. Объект про счетчик ссылок ничего не знает.
- Тип удалителя не является частью типа указателя.
- Может работать только для указателей на объекты.

Что происходит здесь?

```
sp1 = sp2;
```

Некоторые методы std::shared_ptr

- `use_count` — количество shared ptr'ов, ссылающихся на этот же объект;
- `reset` — заменяет объект;
- `unique` — проверяет, что объект контролируется единственным указателем shared ptr;
- `get` — возвращает указатель на объект, которым владеет;

std::shared_ptr

- Перемещение быстрее копирования.
- Счетчик ссылок хранится в динамически выделяемой памяти.
- Пользовательский удалитель не является частью типа указателя.

Управляющий блок

- Функция `std::make_shared` всегда создает управляющий блок.
- Управляющий блок создается, когда указатель `std::shared_ptr` создается из указателя с исключительным владением
- Когда конструктор. `std::shared_ptr` вызывается с обычным указателем — он создает управляющий блок.

Типичная ошибка

```
A* p = new A;
std::shared_ptr<A> sptr1(p);
std::shared_ptr<A> sptr2(p);
```

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:

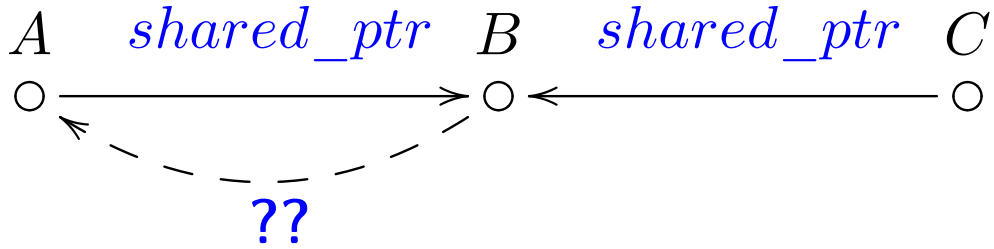
```
std::unique_ptr<const TObject> BuildObject(int param);
```

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?

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

Исправляем:

```
void 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

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

чтобы было безопасно?

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