Язык C++

Мещерин Илья

Лекция 18

Умные указатели

12.1) Мотивировка

```
void f(){
    int *p = new int;
    g();
    delete p;
}
```

Если функция *g()* бросает исключение, то возможна утечка памяти.

12.2) Auto_ptr

Класс представляет скорее историческую ценность, а не практическую.

```
template<class T>
class auto_ptr{
    T* ptr;
public:
    auto_ptr(T* ptr): ptr(ptr) {}
    ~auto_ptr() { delete ptr; }
};
```

Объекты такого класса нельзя копировать и присваивать, то есть, например, нельзя создать вектор объектов такого класса.

12.3) Unique_ptr

Класс реализует концепцию единоличного владения

```
template<class T>
class unqiue_ptr{
    T* ptr;
public:
    unqiue_ptr(T* ptr): ptr(ptr) {}
    ~unqiue_ptr() { delete ptr; }
    unqiue_ptr(const unqiue_ptr<T>&) = delete;
    unqiue_ptr<T>& operator=(const unqiue_ptr<T>&) = delete;
    unqiue_ptr(unqiue_ptr<T> &&p) noexcept {
        ptr = p.ptr;
        p.ptr = nullptr;
    }
    unqiue_ptr<T>& operator=(unqiue_ptr<T> &&p) noexcept {
        ptr = p.ptr;
        p.ptr = nullptr;
    }
};
```

Проблема предыдущего класса решается с помощью move-семантики. Тут может возникнуть проблема, если тип T является, например, массивом, тогда реализация не является корректной.

```
class unqiue_ptr <T[ ]> { ... };
```

Но в стандартной библиотеке написана специализация для решения этой проблемы, но вообще не рекомендуется создавать *unique_ptr* на массив, лучше перед этим обернуть массив в контейнер.
Еще нужно определить *operator*, operator->*, а у *unique_ptr* от массива только *operator*[].

```
void f(){
    std::unique_ptr<int> p = std::unique_ptr<int>(new int);
    std::unique_ptr<int> pp = std::move(p);
    g();
}
```

Так решается исходная проблема, поставленная в начале лекции.

12.4) Shared_ptr

Нужно научиться создавать такой умный указатель, который мог бы правильно обрабатывать ситуацию с хранением обычного указателя на один и тот же объект в разных умных указателях.

```
template<class T>
class shared_ptr{
private:
    struct Helper{
        T *ptr;
        int count;
    };
    Helper *h;
public:
    shared_ptr(T *ptr){
        h = new Helper;
        h.ptr = ptr;
        h.count = 1;
    }
};
```

Такой способ решения проблемы запрещает создавать несколько *shared_ptr* от одного и того же обычного указателя. Оператор и конструктор копирования должны увеличивать счетчик внутри структуры *Helper* на один, оператор и конструктор перемещения забирают объект типа *Helper* себе, деструктор уменьшает счетчик внутри структуры *Helper* на один, когда счетчик обнуляется происходит освобождение обычного указателя.

12.5) Make_shared, make_unique

```
shared_ptr<T> p = make_shared<T>(args);
```

Способ использования.

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

Реализация функции *make_shared()*.

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

При таком вызове функции *f()* сначала может создаться указатель на *int*, затем выполниться функция *g()*, которая может бросить исключение, а только потом сконструироваться *shared_ptr*, т.е. произойдет утечка памяти.

```
void *p = new char[sizeof(T) + sizeof(Helper)];
new (p + sizeof(Helper)) T(std::forward<Args>(args)...);
```

Если реализовать функцию *make_shared()* таким образом, то можно сэкономить на одном вызове оператора *new* и одном вызове оператора *delete*, т.к. объект и счетчик лежат рядом в памяти.

12.5$\frac{1}{2}$) Allocate_shared

```
std::allocator<int> alloc;
std::shared_ptr<int> p = std::allocate_shared<int> (alloc, 42);
```

Таким образом можно выделять память не через оператор *new*, а с помощью своего аллокатора.

```
template<class T, class Alloc, class ... Args>
shared_ptr<T> allocate_shared(const Alloc &alloc, Args &&... args){
    void *p = std::allocator_traits<Alloc>(alloc, /*...*/);
    /*...*/
}
```

Начало реализации функции *allocate_shared()*.