Язык C++

Мещерин Илья

Лекция 19

Умные указатели

12.6) Weak_ptr
Такой умный указатель предназначен для разрешения циклов в указателях.

```
bool expired() const;
std::shared_ptr<T> lock() const;
```

Функция *expired()* проверяет, был ли удален объект, на который ссылается tiweak_ptr. Функция *lock()* создает *shared_ptr*, который управляет объектом, на который ссылается *weak_ptr*.
Из *unique_ptr* можно конструировать *shared_ptr*, а, наоборот, нельзя.
12.7) Deleter
Умных указатели идеологически помогают решить не только проблему с утечками памяти. С их помощью можно поддерживать, например, открытый файл или сетевое соединение. Достаточно передать свой кастомный *Deleter*.

```
template<
    class T,
    class Deleter = std::default_delete<T>
> class unique_ptr;
```

Такие параметры имеет класс *unique_ptr* на самом деле.

```
void operator()(T *ptr){
   delete ptr;
}
```

Примерно это делает *std::default_delete<T>*.
Функции *make_unique()* и *make_shared()* не поддерживают эту фичу.
12.8) std::enable_shared_from_this

```
template<class T>
class enable_shared_from_this;
class C : public std::enable_shared_from_this<C>{};
shared_from_this();
```

Последняя функция возвращает *shared_ptr* на *this*. Это является решением проблемы создания *shared_ptr* от одного и того же указателя (*this*) несколько раз (что делать нельзя). Примерная реализация: внутри класса *std::enable_shared_from_this* хранится *weak_ptr* на *this*, который делает *lock()* при необходимости (тут все сложнее, лучше погуглить).

Вывод типов (type deduction)

12.1) Range-based for loop

```
for (const std::pair<MyType, std::string> &x : m)
```

Пробег по контейнеру (*std::map*) без использования итераторов.

```
for (std :: pair<MyType, std::string> x : m)
```

Так писать плохо, т.к. происходит копирование. На самом деле в первом случае тоже произойдет копирование, т.к. в *std::mape* объектом является *std::pair<const MyType, std::string>*, поэтому произойдет каст и копирование.

12.2) Auto

```
auto x = y;
```

Тут компилятор может сам определить тип *x*.

```
for (const auto &x : m)
```

Удобный *for*.

```
template<typename T>
void f(const T &x)
```

Подстановка типа идет по тем же правилам, как и в случае шаблонного вывода.

```
auto &&x = y;
```

Второй и последний случай, когда ссылка является универсальной.

```
const auto &&x = y;
```

Тут универсальность теряется.

12.3) std::initializer_list

```
std :: vector<int> v = {1, 2, 3};
int x{5};
auto x = {1, 2, 3}
```

В последнем случае *x* имеет тип *std::initializer_list*.