Язык C++

Мещерин Илья

Лекция 17

11.6) Виды value

```
void push_back(const T &x);
void push_back(T &&x);
```

В во второй реализации не будет происходить лишнего копирования.

11.7) Универсальные ссылки

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

Если функция имеет такую сигнатуру (как у *std::move()*), то, вызвав ее от *lvalue* объекта типа *type*, вместо T подставится *type*& (да, это костыль).

11.7$\frac{1}{2}$) Reference collapsing (сворачивание ссылок)

Если в процессе раскрытия шаблонных подстановок могут возникнуть следующие ситуации:
type&& & –> *type*&
type& && –> *type*&
type&& && –> *type*&&

```
void push_back(T &&x);
```

Это **не будет** универсальной ссылкой, т.к. T - шаблонный параметр класса.

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

Также **не является** универсальной ссылкой.

11.8) std::forward и прямая передача

Функция *std::forward()*, в отличие от *std::move()*, кастует объект к *rvalue* тогда и только тогда, когда приняла его по *rvalue* ссылке.

```
template<class ... Args>
void construct(T *p, Args &&... args){
    new (p) T(std::forward<Args>(args)...);
}
```

Правильная реализация *construct()* в аллокаторе (если объекты были типа *lvalue*, а при, например, использовании *std::move()*, вместо *std::forward()*, они стали бы типа *rvalue* и могли бы испортиться где-то дальше).
Похожим образом нужно изменить реализацию метода *emplace_back()* у вектора.

11.9) Реализация std::forward

```
template<class T>
T&& forward(typename std::remove_reference<T>::type &t){
    return static_cast<T&&>(t);
}
```

```
template<class T>
T&& forward(typename std::remove_reference<T>::type &&t){
    return static_cast<T&&>(t);
}
```

Вторая реализация отдельно, чтобы принимать *rvalue* объекты.

11.10) std::move_if_noexcept

Теперь стало понятно, что при реаллокации в векторе нужно делать перемещение, а не копирование. Но что если при перемещении вылетело исключение? Стандартная библиотека гарантирует безопасность относительно исключений. Но в нашей ситуации один массив испорчен, а другой не до конца создан. Делать перемещения обратно делать нельзя, т.к. может опять вылететь исключение. Решение - делать перемещение, только если конструктор перемещения является *noexcept*, в противном случае нужно делать копирование. Функция *std::move_if_noexcept* делает каст к *rvalue* если конструктор перемещения является *noexcept*. Реализация этой функции откладывается на несколько лекций.

11.11) Return Value Optimization

```
BigInteger operator+(const BigInteger &x){
    BigInteger ans = *this;
    ans += x;
    return ans;
}
```

В данном случае объект *ans* создается на том месте памяти, куда должен быть возвращен из функции, т.е. промежуточного объекта не создается. Из-за сложной логики функции компилятор не всегда может сделать такой трюк, в таком случае он сделает перемещение.

11.12) Reference qualifiers

```
void f() const & {}
void f() const && {}
```

Если это методы некоторого класса, то, в зависимости от того является объект этого класса в данный момент *rvalue* или *lvalue*, вызовутся разные версии этого этого метода. В частности можно таким способом запретить конструкцию вида $a + b = c$.