Язык C++

Мещерин Илья

Лекция 16

10.6) Insert iterator

```
template< class InputIt, class OutputIt >
OutputIt copy(InputIt first , InputIt last , OutputIt d_first);
```

Копирует элементы диапазона [*first, last*) в диапазон, начинающийся с *d_first*.

```
template<class Container>
class back_insert_iterator{};
```

Методы *operator*, operator++* ничего не делают.

```
vector<int> a;
std :: back_insert_iterator<vector<int>> it(a);
(*it ) = 42;
```

В данном случае произойдет *push_ back()* в вектор.

```
std :: back_insert_iterator<Container> back_inserter(Container& c);
```

Принимает контейнер и возвращает *std::back_ insert_ iterator* от него.

Move-семантика и rvalue-ссылки

11.1) Мотивировка

```
template<class T>
void swap(T &x, T &y){
    T t = x;
    x = y;
    y = t;
}

vector<int> v;
v.push_back(a);
```

Хочется делать операции *swap(), push_ back()* без лишних копирований

11.2) std::move

```
template<class T>
void swap(T &x, T &y){
     T t = move(x);
     x = move(y);
     y = move(t);
}

vector<int> v;
v.push_back(move(a));
```

Здесь эта проблема решается.

11.3) move-конструктор и move-assigment оператор

```
struct C{
    C(C&& x){...}
    C& operator=(C&& x){...}
};
```

Сигнатуры конструктора перемещения и оператора перемещения. В предыдущих примерах *std::move* преобразует тип объекта в такой, чтобы вызывался конструктор перемещения. Конструктор перемещения создается по умолчанию, и отличается от конструктора копирования тем, что каждое поле инициализируется объектом, пропущенным через *std::move*. Аналогично по умолчанию работает оператор перемещения. Если написать свой нетривиальный конструктор копирования, оператор присваивания или деструктор, то конструктор перемещения и оператор перемещения генерироваться компилятором не будет. В этом заключается **правило пяти**, т.е. если хоть какой-то из пяти озвученных ранее методов реализуется нетривиально, то и остальные, наверняка, реализоваться должны не по умолчанию.

11.4) Реализация std::move

```
template<class T>
std::remove_reference_t<T>::type&& move(T &&x){
    return static_cast<typename std::remove_reference<T>::type&&>(x);
}
```

Пример реализации.

```
template<class T>
remove_reference_t<T>&& move(T &&x){
    return static_cast<typename std::remove_reference<T>::type&&>(x);
}
```

Или вот так, начиная с *C++14*.
Функция принимает объект именно такого типа, чтобы возможно было использовать ее и для lvalue объектов, и для rvalue объектов (если написать один амперсанд, то функция не сможет принимать временные объекты).

11.5) rvalue ссылки

```
int x = 42;
int &&y = x;
```

Ошибка компиляции. Такую ссылку можно связывать только с временными объектами.

```
int &&z = 42;
int &t = z;
```

А так писать можно.

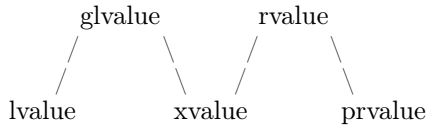
```
int &&p = z;
```

Ошибка компиляции.

```
int &&p = move(z);
```

Так уже можно.
При работе с *const* логика остается прежней: все, что не *const*, нельзя проинициализировать через *const*, а инициализировать *const* через не *const* можно.

11.6) виды value



prvalue (pure rvalue) - настоящие временные только что созданные объекты.
xvalue (expired value) - объекты, которые не являются временными объектами, но трактуются как таковые. (например, результат вызова *std::move()* или каста *static_ cast<type&&>*)
lvalue - именованные объекты, результат вызова функций *operator++, operator=, operator** и другое.
rvalue - результат вызова *operator++(int)* и другое.
glvalue (generalize lvalue).
Хорошая эвристика, позволяющая отличать эти типы - у *lvalue* объекта можно взять адрес. Если объект - именованная сущность, то это точно объект типа *lvalue*.