Лекция от 21.03

Лектор: Илья Мещерин

Вывод типов. Продолжение

`auto` в качестве типа возвращаемого значения функции

```
auto f() {
    return 1;
}
```

Нельзя иметь несколько разнотипных `return`, поскольку две функции с одинаковой сигнатурой не могут различаться типом возвращаемого значения. Однако, несколько однотипных `return` допустимы.

---

decltype

Если написать так, то не происходит создание нового объекта:

```
const auto& x = f();
```

Иногда вызывает неудобства: если в большом проекте слишком много `auto`, то тяжелее разобраться в коде. С другой стороны, если изменить тип возвращаемого значения у функции `f`, то строка выше будет по-прежнему работать.

Если важно является ли значение `lvalue` или `rvalue`:

```
auto&& x = f();
```

Вернёмся к предыдущей строке:

```
const auto& x = f();
```

Допустим, нам нужно определить тип `x`. Для этого мы можем воспользоваться `decltype` (на этапе компиляции разворачивается в название типа):

```
c<decltype(x)> c;
```

Что сделает `decltype` в таком случае?

```
int&& y = 5;
c<decltype(y)> c;
```

Оказывается, он сохраняет как одиночный амперсанд, так и двойной. Также он сохраняет `const`.

А сейчас что он сделает?

```
decltype(y++) z;
```

Внутри `decltype` можно написать любое выражение (пока опустим здесь тернарные операторы). Компилятор не будет его вычислять, он посмотрит, что было бы в результате его вычисления.

`decltype` позволяет навешивать на себя модификаторы типов:

```
decltype(throw 1)* p = new int;
```

Существует несколько странных правил: 1. Если `decltype` имеет внутри себя `prvalue`, то он возвращает `T` 2. Если `decltype` имеет внутри себя `lvalue`, то он возвращает `T&` 3. Если `decltype` имеет внутри себя `xvalue`, то он возвращает `T&&`

```
struct A { double d; }
const A* p;
decltype(p->d);
// Но тут будет просто double, к этому не применяются странные
```

Если взять `x` в круглые скобки, то получится нетривиальное выражение. `decltype` навесит `&`:

```
decltype((x)) y;
```

Примеры применения decltype

Если вместо типа возвращаемого значения написать `auto`, то `auto` отбросит амперсанд (будет происходить копирование):

```
template<class Container>
    auto getByIndex(const Container& c, int i) {
        ...
        return c[i];
}
```

Если же сделать так, то проблем меньше не станет (к примеру, в

`vector<bool> x` обращение по индексу возвращает `rvalue`):

```
template<class Container>
    auto& getByIndex(const Container& c, int i) {
        ...
        return c[i];
}
```

До `C++14` проблема решалась таким образом:

```
template<class Container>
    auto getByIndex(const Container& c, int i) -> decltype(c[i]
        ...
        return c[i];
}
```

В `C++14` появился классный парень `decltype(auto)`:

```
template<class Container>
    decltype(auto) getByIndex(const Container& c, int i) {
        ...
        return c[i];
}
```

Что будет если сделать так?

```
decltype(x ? 1 : "abc") y;
```

Не будет ничего интересного, кроме ошибки компиляции. Более того, ошибка компиляции возникнет в принципе, если в тернарном операторе `x ? y : z` типы `y` и `z` несовместимы (полный список правил находится на `cppreference`).

Рассмотрим следующую ситуацию (нужно определить тип `x`):

```
template<typename T>
class C { C() = delete; } // Работающий трюк

template<typename T>
void f(const T&& x) {
    C<decltype(x)> t;
    // Компилятор скажет, что не может создать объект такого ти
    // И назовёт тип
}
```