Семестр 2. Лекция 7. C++11. auto. Разное. lambda.

Евгений Линский

28 Апреля 2017

Вывод типов: auto и decltype

```
// тип x будет выведен по expression
auto x = expression;

// тип x будет такой же, как у expression
decltype(expression) y;
```

```
//reverse_iterator: идет от end (rbegin) к begin
for(std::vector<string>::const_iterator
  i  = v.cbegin(); i!= v.cend(); ++i){
    std::vector<string>::const_reverse_iterator j = i;
}
// C++11
for(auto i = v.cbegin(); i != v.cend(); ++i) {
  decltype(v.crbegin()) j = i;
}
```

Вывод типов: auto и decltype

```
// тип x будет выведен (без & и const) по expression
auto x = expression;

// тип x будет такой же, как у expression
decltype(expression) y;
```

```
const std::vector<int> v(1);
auto a = v[0];              // a - int
decltype(v[0]) b = 1;       // b - const int&

//auto можно уточнить:
string s = "hello";
auto& s1 = s;               // s1 - string&
const auto& c = v[0];       // c - const int&
```

```
// decltype можно использовать в typedef
typedef decltype(v[0]) new_type;
```

Вывод типов: auto и decltype

Увлекаться не надо!

```
std::map <KeyClass, ValueClass> m;
// OK, все знают, что find возвращает iterator.
auto I = m.find(something);

MyClass myObj;
// NOT OK, большинству нужно пойти и проверить,
// что возвращает FindRecord.
auto ret = myObj.findRecord(something);
```

nullptr

```
libc: #define NULL 0
```

```
void f(int* ptr) { ... }
void f(int n) { ... }

f(NULL); // error:  call of overloaded ‘f(NULL)’ is ambiguous
f(nullptr); // OK
```

Вычисления в compile time

- static_assert (как assert, только в compile time)

```
static_assert(sizeof(unsigned int) * 8 == 32,
  "16bit CPU is not supported.");
```

- constexpr — функция может быть вычислена в compile time (компилятор это проверит!) и использована в const выражении.
- Функция constexpr должна быть “single statement” (однострочная). Есть и другие ограничения.

```
constexpr unsigned fibonacci(unsigned i) {
  return (i <= 1u) ?
    i : (fibonacci(i-1) + fibonacci(i-2));
}
int array[fibonacci(3)];
```

begin(), end()

```
//begin() в отличии от v.begin() работает для массивов
template <class T>
void foo(T& v) {
  auto i = begin(v);
  auto e = end(v);
  for(; i != e; i++) { ... }
}
```

```
template<typename T, size_t N>
T* end(T (&a)[N]) { return a + N; }

// параметр N выводится компилятором
int a[4] = {0, 1, 2, 3};
auto i = end(a);
```

for(x:)

Реализован на begin() и end().

```
void f(vector<int>& v) {
    for (auto x : v) cout << x << '\n';
    // чтобы изменить x, нужна ссылка
    for (auto& x : v) ++x;
}

int array[5] = {1, 2, 3, 4, 5};
for(int &x : array) {
  x *= 2;
}
```

lambda

```
vector<int> v = {50, -10, 20, -30};
// До C++11:
struct abs_cmpr {
  bool operator()(int a, int b) const {
    return abs(a) < abs(b);
  }
};
std::sort(v.begin(), v.end(), abs_cmpr());


//C++11:
std::sort(v.begin(),
          v.end(),
          [](int a, int b) { return abs(a)<abs(b); });

// [](int a, int b) { return abs(a)<abs(b); } -- lambda function
```

lambda, capture list

```
vector<int> lst = {1,2,3,4,5};
int total = 0;
// захват по ссылке
for_each(lst.begin(),
         lst.end(),
         [&total](int x) { total += x; });

// захват по значению
for_each(lst.begin(),
         lst.end(),
         [total](int & x) { x -= total ; });
```

- Можно “захватить” (capture) локальные переменные из области видимости.
- Могут быть разные типы захвата, в т.ч. смешанные: [] (ничего), [&] (все по ссылке), [=] (все по значению), $[x, \&y]$, $[\&, x]$, $[=, \&z]$.
- Не стоит использовать захват по-умолчанию ([&] или [=]).

lambda, capture list

C++11:

```
vector<int> lst = {1,2,3,4,5};
int total = 0;
// захват по ссылке
for_each(lst.begin(),
         lst.end(),
         [&total](int x) { total += x; });
```

До C++11:

```
vector<int> lst = {1,2,3,4,5};
int total = 0;

struct summator {
  int& sum;

  summator(int &s) : sum(s) {};
  void operator()(int x) { sum += x; }
};

for_each(lst.begin(), lst.end(), summator(total));
```