C++11 и выше: некоторые особенности языка

Коноводов Владимир

кафедра математической кибернетики ВМК

11.12.2017

C++11: auto и decltype

Автоматический вывод типа переменной сильно упрощает жизнь.

```
auto x = 0;
auto y = x, *z = &x; // int * y
auto& ref = x;  // int& ref
const auto & xcref = x; // const int&
auto* p = &x; //int* p;
const auto* cp = p; // const int * p;
const auto pc = &x ; // int * const pc
```

decltype используется всегда в одном и том же контексте — чтобы вернуть тип чего-то. А «чего-то» находится у него как аргумент в скобочках. Чаще всего хочется получить тип переменной или тип выражения. Очень удобно использовать decltype в шаблонах.

```
decltype(1 + 2) x = 1; // int
decltype(x) y = x; // int
decltype(1,x) z = x; // ???
```

C++11: фигурные скобки

Создаем контейнер, содержащий определенный набор значений.

```
std::vector<int> v {1, 3, 5};
TFoo a{};

// вектор с 10 элементами, каждый =20
std::vector<int> v1 (10, 20);

// конструктор с std::initializer_list:
// вектор из 2х элементов 10 и 20
std::vector<int> v2 {10, 20};
```

C++11: фигурные скобки

```
#include <iostream>
class T {
  public:
    T(int a, bool b) { std::cout << "int, bool" << std::endl;}
    T(int a, double b) { std::cout << "int, double" << std::endl;}
    T(std::initializer_list<long double> l) {
        std::cout << "init list" << std::endl;
    }
};

int main() {
    T t1(10, true);
    T t2 {10, true};
    T t3(10, 0.2);
    T t4 {10, 0.2};
}
```

C++11: фигурные скобки

```
#include <iostream>
class T {
  public:
    T(int a, bool b) { std::cout << "int, bool" << std::endl;}
    T(int a, double b) { std::cout << "int, double" << std::endl;}
    T(std::initializer_list<long double> l) {
        std::cout << "init list" << std::endl;
    }
};

int main() {
    T t1(10, true);  // int, bool
    T t2 {10, true};  // init list
    T t3(10, 0.2);  // int, double
    T t4 {10, 0.2};  // init list
}
```

C++11: range-based циклы

```
for (auto x : container) {
    // x - копия элемента в контейнере
}
for (auto& x : container) {
    // x - ссылка на элемент в контейнере
}
```

C++11: Псевдонимы

typedef:

```
typedef
    std::shared_ptr<std::map<std::string, std::string> >
    TMyPtr;
typedef bool (*FPtr)(int, int);
```

using (C++11):

```
using TMyPtr =
    std::shared_ptr<std::map<std::string, std::string> >;
using FPtr = bool (*)(int, int);
```

В чем отличие typedef от using?

Объявление псевдонимов поддерживает шаблонизацию.

C++11: scoped enumerations

```
enum Color {black, white, blue};
bool white; // error!
```

C++11:

```
enum class Color { red, green = 20, blue };
Color r = Color::blue;
switch (r) {
    case Color::red:  // ...
    case Color::green:  // ...
    case Color::blue:  // ...
}
int n = r; // ошибка
int n = static_cast<int>(r);
```

Базовый тип — int.

constexpr

```
const int a = 10;
const int b = std::numeric_limits<int>::max(); // <limits>
const int c = INT_MAX;
```

```
int a;
const int b = a; // ok
constexpr auto s = a; // error
```

```
constexpr int f() {return 1024;}
```

constexpr-функция должна состоять из одного return (C++11), возвращать константу или вызывать такую же функцию. Вычисление должно производиться во время компиляции (с аргументами, значения которых известны во время компиляции).

Пример: проверка простоты числа в compile-time

```
constexpr bool is_div(int a, int b) {
  return (b == 1) || (a % b != 0 && is_div(a, b - 1) );
}

constexpr bool is_prime(int number) {
  return number != 1 && is_div(number, number / 2);
}

int main() {
  static_assert(is_prime(29) , " 29 is not prime");
  static_assert(is_prime(36) , " 36 is prime");
  return 0;
}
```

C++11: лямбда-выражения

Быстрый способ создать такую структуру с оператором ():

```
struct T {
    bool operator()(int x){};
};
Do(T(), ...);
```

- [] — список переменных, которые захватывает лямбда-выражение;
- () — входные аргументы функции;
- {} — тело функции.

C++11: лямбда-выражения

```
[capture] (params) mutable exception_attribute -> ret {body}
[capture] (params) -> ret {body}
[capture] (params) {body}
[capture] {body}
```

Пример:

```
std::vector<int> v = {-1, -2, -3, -4, -5, 1, 2, 3, 4 ,5};
std::sort(v.begin(), v.end(), [](int l, int r) {
    return l * l < r * r;
});
```

C++11: лямбда-выражения

- `[]` — без захвата переменных
- `[=]` — все переменные захватываются по значению
- `[&]` — все переменные захватываются по ссылке
- `[x]` — захват `x` по значению
- `[&x]` — захват `x` по ссылке
- `[x, &y]` — захват `x` по значению, `y` по ссылке
- `[=, &x, &y]` — захват всех переменных по значению, но `x`,`y` – по ссылке
- `[&, x]` — захват всех переменных по ссылке, кроме `x`
- `[this]` — для доступа к переменной класса

C++11: move-семантика

```
template <typename T>
void Swap(T& a, T& b) {
    T t(a);
    a = b;
    b = t;
}
```

Много лишних копирований. Неплохо бы сказать компилятору, что они необязательны. Сообщим компилятору, что копия не нужна, и можем значение, которое лежит в a полностью переместить в объект t . А что там оставить — что угодно, например, содержимое пустого вектора.

```
template <typename T>
void Swap(T& a, T& b) {
    T t(std::move(a)); // move constructor
    a = std::move(b);  // move operator =
    b = std::move(t);  // move operator =
}
```

C++11: move-семантика

```
TFoo a, b;
// TFoo &&c = b; - that's not ok
TFoo &&c = std::move(b);
a = c;  // copy constructor
a = std::move(b);  // move constructor


void f(T&& x);  // rvalue-ссылка

T&& x = T();  // rvalue-ссылка

template<typename T>
void f(T&& x);  // универсальная ссылка

template<typename T>
void f(std::vector<T>&& x);  // rvalue-ссылка
```

C++11: универсальные ссылки

Шаблон с универсальной ссылкой

```
template <typename T>
void call(T&& obj);
```

- Если в качестве аргумента передается lvalue, то T выводится как lvalue-ссылка.
- Если в качестве аргумента передается rvalue, то T не является ссылкой.

```
int x;
call(x);  // T - int&
call(std::move(x));  // T - int
```

Проблемы new и delete

1. Можно забыть написать delete.
2. Можно написать лишний delete.
3. Утечки памяти при исключениях и т.п.
4. delete / delete[].

Как решать?

- Оставить delete умным указателям.
- Оставить new make-функциям.

Smart pointers

Чем плохи обычные встроенные указатели?

- Указывают на массив или на объект?
- Владеет ли указатель тем, на что указывает?
- Трудно обеспечить уничтожение ровно один раз.
- Обычно сложно определить, является ли указатель висячим.
- Нельзя предоставить информацию компилятору о том, могут ли два указателя указывать на одну область памяти.

«Умный» («интеллектуальный») указатель притворяется обычным указателем с дополнительными функциями.
Обертка над обычными указателями.

C++11: std::unique_ptr

- Реализует семантику исключительного владения
- Перемещение передает владение от исходного указателя целевому, целевой при этом обнуляется.
- Копирование не разрешается.
- При деструкции освобождает ресурс, которым владеет.

Обычное использование – возвращаемый тип фабричных функций для объектов иерархии:

```
template <typename T>
std::unique_ptr<Base> makeObject(T&& params);
```

Как избавиться от new?

Написать обертку!

```
template <typename T, typename... Ts>
std::unique_ptr<T> make_unique(Ts&&... params) {
    return std::unique_ptr<T>(
        new T(std::forward<Ts>(params)...)
    )
}
```

Чего не хватает? Массивов, пользовательских удалителей.

Функция уже есть — `std::make_unique` в C++14.

```
std::unique_ptr<Base> p(new Base); // дважды пишем Base
auto p1(std::make_unique<Base>()); // make
```

C++11: std::shared_ptr

- Реализует семантику совместного владения.
- Использует метод подсчета ссылок. Счетчик ссылок хранится в динамически выделяемой памяти. Объект про счетчик ссылок ничего не знает.
- Тип удалителя не является частью типа указателя.
- Может работать только для указателей на объекты.

Что происходит здесь?

```
sp1 = sp2;
```

C++11: std::shared_ptr

- Перемещение быстрее копирования.
- Счетчик ссылок хранится в динамически выделяемой памяти.
- Пользовательский удалитель не является частью типа указателя.

C++14

- Расширенный вывод возвращаемого значения функций.
- Менее ограниченные `constexpr`-функции.
- Захват выражений в лямбдах.
- ...

C++17

- `std::optional`
- `template<auto>`
- `std::variant`
- `...`

C++17: std::variant

Это такой union, который знает, какой именно тип он хранит.

```
std::variant<int, char, double> v;
v = 5;
std::cout << std::get<int>(v);

// std::cout << std::get<double>(v);
// исключение std::bad_variant_access

// std::cout << std::get<float>(v);
// не компилируется

auto p = std::get_if<char>(&v); // nullptr
```