

Краткое описание стандарта C++11

Гусев Илья

Московский физико-технический институт

Москва, 2017

Содержание

- 1 Упрощение
 - Auto
 - Range-for
 - Array
 - Tuple
 - nullptr
 - In-class member initializers
 - Regex
- 2 Обобщение
 - Uniform initialization
- 3 Расширение
 - Inherited constructors
 - Delegating constructors
 - Static assertions
- 4 Исправление
 - Right-angle brackets

Список фич

- Упрощение:

`Auto`, `array`, `tuple`, `forward_list`, `unordered containers`,
`range-for` statement, `regex`, `nullptr`, `in-class member`
`initializers`, `raw string literals`

- Обобщение:

`Uniform initialization`, `inherited constructors`, `delegating`
`constructors`, `static assertions`, `threads`, `template alias`,
`variadic templates`

- Расширение:

`Lambdas`, `default and delete`, `suffix return type`, `smart`
`pointers`, `override-final`, `enum class`, `constexpr`, `noexcept`,
`user-defined literals`, `alignment`, `rvalue`, `TLS`, `async`,
`future-promise`

- Исправление:

`Right-angle brackets`, `unions`, `PODs`

Auto

Вывод типа в инициализаторе

Компилятор в состоянии предугадать тип переменной при её инициализации. Ключевое слово - auto. Пример:

```
auto i = 7;
```

Компилятор понимает, что i - переменная типа int. Более сложный пример:

```
template<class T, class U> void multiply( const vector<T>& vt,  
    const vector<U>& vu )  
{  
    auto tmp = vt[0] * vu[0];  
}
```

Auto

Вывод типа в инициализаторе

Было:

```
template<class T> void printall( const vector<T>& v )
{
    for( typename vector<T>::const_iterator p = v.begin();
        p!=v.end(); ++p )
    {
        cout << *p << "\n";
    }
}
```

Стало:

```
template<class T> void printall( const vector<T>& v )
{
    for( auto p = v.begin(); p!=v.end(); ++p ) {
        cout << *p << "\n";
    }
}
```

Auto

Вывод типа в инициализаторе

Здесь же нужно упомянуть decltype. То, что раньше называлось typeof.

```
void f( const vector<int>& a, vector<float>& b )
{
    typedef decltype( a[0] * b[0] ) Tmp;
    for( int i = 0; i < b.size(); ++i ) {
        Tmp* p = new Tmp( a[i] * b[i] );
    }
}
```

Range-for statement

Удобный цикл

Аналог foreach цикла в других языках. Можно использовать для большинства стандартных контейнеров и для всего, где определён begin() и end(), в том числе и для пользовательских классов.

```
void f( vector<double>& v )
{
    for( auto x : v ) cout << x << '\n';
    // По ссылке можем менять значение.
    for( auto& x : v ) ++x;
}
```

Array

Новый простой стандартный контейнер

- Элемент STL, `std::array`
- Как обычный статический массив, но с поддержкой всех итераторных методов STL
- Поддержка initializer list
- Знает свой размер
- Не спутать с указателем

Array

Новый простой стандартный контейнер

```
array<int, 6> a = { 1, 2, 3 };  
a[3] = 4;
```

```
// x равен 0, потому что по умолчанию элементы zero initialized.  
int x = a[5];
```

```
// Error: std::array неявно не преобразовывается у указателю.  
int* p1 = a;  
int* p2 = a.data(); // Ok: указатель на первый элемент.  
array<int> a3 = { 1, 2, 3 }; // Error: не указан размер.  
array<int, 0> a0; // Ok: пустой массив.  
int* p = a0.data(); // Unspecified; не делайте так.
```

Tuple

Кортеж в C++

- Элемент STL, `std::tuple`
- Удобен для хранения разнородных объектов
- Частный случай - `std::pair`

```
std::tuple<string,int> t2( "Word", 123 );  
// У t будем mun tuple<string,int,double>.  
auto t = make_tuple( string( "Letter" ), 10, 1.23 );  
string s = get<0>(t);  
int x = get<1>(t);  
double d = get<2>(t);
```

nullptr

Выделенный нулевой указатель

```
char* p = nullptr;
int* q = nullptr;
char* p2 = 0; // 0 работает, p==p2.

void f(int);
void f(char*);
f(0); // Вызов f(int).
f(nullptr); // Вызов f(char*).

void g(int);
g(nullptr); // Error: nullptr не int.
int i = nullptr; // Error: nullptr не int.
```

In-class member initializers

Инициализируем поля непосредственно в классе

```
class A {  
public:  
    int a = 7;  
};
```

ЭКВИВАЛЕНТНО

```
class A {  
public:  
    int a;  
    A(): a(7) {}  
};
```

Если есть и то, и то - constructor initialization > in-class initialization.

Regex

Регулярные выражения в C++

Семейство функций `regex_replace`, `regex_search`, `regex_match`.

```
std::string text = "Quick brown fox";  
std::regex vowel_re( "a|o|e|u|i" );  
std::regex_replace( text, vowel_re, "[X]" );  
// Q[X][X]ck br[X]wn f[X]x
```

Uniform initialization

Одинаковая инициализация в разных ситуациях

Новый тип - `std::initializer_list<T>`. Любая длина списка, но один и тот же тип элементов.

```
void f( initializer_list<int> );  
f( {1, 2} );  
f( {23, 345, 4567, 56789} );  
f({}); // Пустой список инициализации.  
f{ 1,2 }; // Error: нет () при вызове функции  
years.insert( { {"Bjarne", "Stroustrup"}, {1950, 1975, 1985} } );
```

Uniform initialization

Одинаковая инициализация в разных ситуациях

Раньше:

```
// Ok: можно использовать для стат. массивов
string a[] = { "foo", " bar" };
// Error: для std::vector уже нельзя
vector<string> v = { "foo", " bar" };
void f( string a[] );
f( { "foo", " bar" } ); // Syntax error: block as argument

int a = 2; // Обычное присваивание
int aa[] = { 2, 3 }; // Обычное присваивание массиву
complex z( 1, 2 ); // Инициализация в функциональном стиле
// Инициализация в функциональном стиле с преобразованием типа
x = Ptr(y);

int a(1); // Определение переменной.
int b(); // Объявление переменной.
int b(foo); // Определение переменной или объявление функции.
```

Uniform initialization

Одинаковая инициализация в разных ситуациях

Теперь можно так:

```
X x1 = X{ 1, 2 };
```

```
X x2 = { 1, 2 };
```

```
X x3{ 1, 2 };
```

```
X* p = new X{1,2};
```

```
struct D : X {  
    D( int x, int y ) :X{ x, y } {};  
};
```

```
struct S {  
    int a[3];  
    S( int x, int y, int z ) :a{ x, y, z } {};  
};
```

```
X x{a};
```

```
X* p = new X{a};
```

```
z = X{a}; // Преобразование типа.
```

```
f( {a} ); // Аргумент функции (типа X).
```

```
return {a}; // Возвращаемое значение функции (типа X)
```


Inherited constructors

Конструкторы базового класса в области видимости наследника

Была проблема:

```
struct B {  
    void f(double);  
};  
struct D : B {  
    void f(int);  
};  
B b; b.f(4.5);           // Ок.  
// Неочевидная особенность: вызов f(int) с аргументом 4.  
D d; d.f(4.5);
```

Inherited constructors

Конструкторы базового класса в области видимости наследника

И было решение:

```
struct B {  
    void f(double);  
};  
struct D : B {  
    // Вносим все f() базового класса в область видимости.  
    using B::f;  
    void f(int);  
};  
  
B b;    b.f(4.5);           // Ок.  
// Ок: вызов D::f(double), которая на самом деле B::f(double).  
D d;    d.f(4.5);
```

Inherited constructors

Конструкторы базового класса в области видимости наследника

Раньше подобный трюк не работал с конструкторами, теперь работает.

```
class Derived : public Base {  
public:  
    using Base::f;      // Работает в C++98.  
    void f(char);  
  
    using Base::Base;   // Работает только с C++11.  
    Derived(char);  
};
```

Delegating constructors

Вызов одного конструктора в другом

Раньше нужно было извернуться, чтобы 2 конструктора разделяли какую-то функциональность:

```
class X {  
    int a;  
    void init(int x) { /* ... */ }  
public:  
    X(int x) { init(x); }  
    X() { init(42); }  
};
```

Теперь всё просто:

```
class X {  
    int a;  
public:  
    X(int x) { { /* ... */ }  
    X(): X{42} { }  
};
```

Static assertions

Проверки на этапе компиляции

Сутья ясна, но не сразу понятно, зачем это нужно. Варианты использования - проверка платформы при компиляции и контроль целостности enum'ов.

```
static_assert( sizeof(long) >= 8,  
    "64-bit code generation required for this library." );  
struct S { X m1; Y m2; };  
static_assert( sizeof(S) == sizeof(X) + sizeof(Y),  
    "unexpected padding in S" );  
  
int f(int* p, int n)  
{  
    // Error: static_assert() не работает для runtime-проверок.  
    static_assert( p == 0, "p is not null" );  
}
```

Lambdas

Лямбда-функции

Безымянные функции, которые могут захватывать переменные из внешней по отношению к ним области видимости.

```
vector<int> v = {50, -10, 20, -30};
std::sort(v.begin(), v.end());
// v: { -30, -10, 20, 50 }
std::sort(v.begin(), v.end(),
    [](int a, int b) { return abs(a)<abs(b); });
// v: { -10, 20, -30, 50 }
```

Lambdas

Лямбда-функции. Захват переменных

[] - нет захвата переменных.

[&] - захват всех переменных по ссылке.

[=] - захват всех переменных по значению.

[&a, &b] - захват a и b по ссылке.

[a, b] - захват a и b по значению.

[in, &out] - захват in по значению, a out — по ссылке

[=, &out1, &out2] - захват всех переменных по значению, кроме out1 и out2, которые захватываются по ссылке.

[&, x, &y] - захват всех переменных по ссылке, кроме x.

```
void f( vector<Record>& v )
{
    vector<int> indices( v.size() );
    int count = 0;
    generate( indices.begin(), indices.end(),
              [&count]() { return count++; } );
    std::sort( indices.begin(), indices.end(),
              [&](int a, int b) { return v[a].name < v[b].name; } );
}
```

Lambdas

Лямбда-функции. Тип возвращаемого значения

По умолчанию лямбда возвращает `void`. При наличии одного `return`, компилятор вычисляет тип возвращаемого значения. Если же в лямбде присутствует ветвление, то на компилятор полагаться уже нельзя:

```
[] (int n)
{
    if (n % 2 == 0)
        return n / 2.0;
    else
        return n * n;
});
```

Компилятор не может самостоятельно вычислить тип возвращаемого значения, поэтому мы должны его указать явно:

```
[] (int n) -> double // suffix return type
{
    if (n % 2 == 0)
        return n / 2.0;
    else
        return n * n;
});
```


Default and delete

Контроль поведения по умолчанию

Мы можем запрещать использование функций, а также заставлять использовать версии по умолчанию.

```
class X {  
public:  
    X& operator=(const X&) = delete; // Запретили копирование.  
    X(const X&) = delete;  
};  
  
class Y {  
public:  
    // Явно заставили использовать копирование по умолчанию.  
    Y& operator=(const Y&) = default;  
    Y(const Y&) = default;  
};  
  
struct Z {  
    Z(long long); // Может инициализироваться с long long,  
    Z(long) = delete; // но ни с чем больше.  
};
```

Smart pointers

Умные указатели - владение

`Unique_ptr` - контейнер с семантикой владения.

- 1 Владеет объектом, на который указывает.
- 2 Запрещает копирование, но разрешает перемещение (`move`).
- 3 При уничтожении указателя, уничтожается и объект, которым он владеет.
- 4 Позволяет возвращать из функции объекты, выделенные на динамической памяти.
- 5 Кроме того корректно уничтожает такие объекты при раскрутке стека во время проталкивания исключений.
- 6 Опирается на `rvalue` семантику.

Smart pointers

Умные указатели - владение

Примеры:

```
unique_ptr<X> f()
{
    unique_ptr<X> p( new X );
    // Можно кидать исключения, всё будет ок.
    return p; // передали право владения наружу f()
}

void g()
{
    // Перемещаем с помощью move constructor.
    unique_ptr<X> q = f();
    q->memfct(2); // Используем q.
    X x = *q; // Можео скопировать сам объект.
} // q и объект, которым он владеет уничтожаются при выходе.
```

Smart pointers

Умные указатели - разделяемое владение

- 1 Shared_ptr - контейнер с семантикой разделяемого владения.
- 2 Weak_ptr - контейнер, дополняющий shared_ptr, нужен, чтобы не было циклических зависимостей и, соответственно, неумирающих объектов.
- 3 Shared_ptr реализован через счётчик ссылок, каждый shared_ptr, указывающий на объект даёт +1 к этому счётчику.
- 4 Если счётчик становится равен нулю, объект уничтожается.
- 5 Weak_ptr не меняет счётчик ссылок.
- 6 При неправильном использовании возможно возникновение циклических зависимостей.
- 7 Shared_ptr дорого стоит, особенно в параллельной среде.

Smart pointers

Умные указатели - разделяемое владение

Иллюстрация к счётчику ссылок:

```
void test()
{
    shared_ptr<int> p1(new int); // Счётчик равен 1.
    {
        shared_ptr<int> p2(p1); // Счётчик равен 2.
        {
            shared_ptr<int> p3(p1); // Счётчик равен 3.
        } // Счётчик равен 2.
    } // Счётчик равен 1.
} // Счётчик равен 0, int уничтожается.
```

Suffix return type

Помощь в выводе типа возвращаемого значения

Иногда мы должны помочь компилятору вывести тип возвращаемого значения, без нас он этого сделать не может. Мы не можем написать `decltype(x * y)` где обычно (вместо `auto`), так как `x` и `y` там ещё не определены. Таким образом, эта штука нужна в основном именно для обхода области видимости. Используется в лямбдах.

```
template<class T, class U>
auto mul( T x, U y ) -> decltype( x * y )
{
    return x * y;
}
```

Override-final

Явные виртуальные функции - override

```
struct B {  
    virtual void f();  
    virtual void g() const;  
    virtual void h(char);  
    void k(); // Не виртуальная.  
};  
struct D : B {  
    void f(); // Перезаписывает B::f().  
    void g(); // Не перезаписывает B::g() (не та сигнатура).  
    virtual void h(char); // Перезаписывает B::h().  
    void k(); // Не перезаписывает B::k() (B::k() не виртуальна).  
};  
// Сейчас можно так  
struct D : B {  
    void f() override; // OK: overrides B::f().  
    void g() override; // Error: не та сигнатура.  
    virtual void h(char); // Перезаписывает B::h(), warning.  
    void k() override; // Error: B::k() не виртуальна.  
};
```

Override-final

Явные виртуальные функции - final

```
struct B {  
    // Говорим, что перезаписывать нельзя.  
    virtual void f() const final;  
    virtual void g();  
};  
  
struct D : B {  
    // Error: D::f попытка перезаписи финальной B::f.  
    void f() const;  
    void g(); // OK.  
};
```



Right-angle brackets


Пробел не нужен!

```
list<vector<string>> lvs;
```


Раньше была синтаксическая ошибка, теперь нет.

Полезные ссылки I

 C++11 - the new ISO C++ standard
Bjarne Stroustrup
<http://www.stroustrup.com/C++11FAQ.html>

 C++ Super-FAQ
Bjarne Stroustrup, Marshall Cline
<https://isocpp.org/faq>

 N3337 2012 Draft (C++11)
ISO/IEC
<http://open-std.org/jtc1/sc22/wg21/docs/papers/2012/n3337.pdf>

 C++0x (C++11). Лямбда-выражения
Сергей Олендаренко
<https://habrahabr.ru/post/66021/>