Стандарт C++ 11: лямбда-выражения и много другое

Александр Смаль

CS центр
5 апреля 2017
Санкт-Петербург

Кортежи

```
std::tuple<double, char, std::string> get_student(int id) {
    if (id == 0) return std::make_tuple(3.8,'A',"Lisa");
    if (id == 1) return std::make_tuple(2.9,'C',"Milhouse");
    if (id == 2) return std::make_tuple(1.7,'D',"Ralph");
    throw std::invalid_argument("id");
}
int main() {
    auto st0 = get_student(0);
    std::cout << "ID: 0, " << "GPA: "   << std::get<0>(st0) << ", "
                           << "grade: " << std::get<1>(st0) << ", "
                           << "name: "  << std::get<2>(st0) << '\n';

    double gpa1; char grade1; std::string name1;
    std::tie(gpa1, grade1, name1) = get_student(1);
    std::cout << "ID: 1, "  << "GPA: "   << gpa1   << ", "
                            << "grade: " << grade1 << ", "
                            << "name: "  << name1  << '\n';
}
```

Явное переопределение и финальность

```
struct B {
    virtual void some_func();
    virtual void f(int);
    virtual void g() const;
};
struct D1 : B {
    void sone_func() override;           // error
    void f(int) override;                // OK
    virtual void f(long) override;       // error
    virtual void f(int) const override; // error
    virtual int f(int) override;         // error
    virtual void g() const final;        // OK
    virtual void g(long);                // OK
};
struct D2 final : D1 {
    virtual void g() const;              // error
};
struct D3 : D2 {};                       // error
```

Делегация конструкторов

```
struct SomeType  {
    SomeType(int new_number) : number(new_number) {}
    SomeType() : SomeType(42) {} // делегация конструктора
private:
    int number;
};
struct SomeClass {
    SomeClass() {} // value = 42
    explicit SomeClass(int new_value) : value(new_value) {}
private:
    int value = 42; // значение по-умолчанию
};
struct BaseClass {
    BaseClass(int value);
};
struct DerivedClass : public BaseClass {
    using BaseClass::BaseClass; // конструкторы базового класса
};
```

Новые строковые литералы

```
u8"I'm a UTF-8 string."          // char[]
u"This is a UTF-16 string."      // char_16_t[]
U"This is a UTF-32 string."      // char_32_t[]
L"This is a wide-char string."   // wchar_t[]

u8"This is a Unicode Character: \u2018."
u"This is a bigger Unicode Character: \u2018."
U"This is a Unicode Character: \U00002018."

R"(The String Data \ Stuff " )"
R"delimiter(The String Data \ Stuff " )delimiter"

LR"(Raw wide string literal \t (without a tab))"
u8R"XXX(I'm a "raw UTF-8" string.)XXX"
uR"*(This is a "raw UTF-16" string.)*"
UR"(This is a "raw UTF-32" string.)"
```

Изменения в стандартной библиотеке

1. Исправлен смысл хинта при вставке в `set/map`.
2. Метод `emplace` для контейнеров.

```
template< class... Args >
iterator emplace( const_iterator pos, Args&&... args );
```

3. Методы `cbegin` и `cend` (для метапрограммирования, для задания типов через auto).
4. Метод `shrink_to_fit` для `vector`-а.
5. В `list` `splice` — за $O(n)$, `size` — за $O(1)$.
6. В `vector` добавился прямой доступ к памяти через `data()`
7. Запрет нескольким `string` ссылаться на одну память.
8. Добавлены `unordered_set` и `unordered_map`.
9. Добавлены `unique_ptr`, `shared_ptr`, `weak_ptr`.

Константные выражения

Для констант и функций времени компиляции.

```
constexpr double accelerationOfGravity = 9.8;
constexpr double moonGravity = accelerationOfGravity / 6;

constexpr int pow(int x, int k)
    { return k == 0 ? 1 : x * pow(x, k - 1); }

std::bitset<pow(3, 5)> bs;
```

```
struct Point {
   double x, y;
   constexpr Point(double x = 0, double y = 0) : x(x), y(y) {}
   constexpr double getX() const { return x; }
   constexpr double getY() const { return y; }
};
constexpr Point p(moonGravity, accelerationOfGravity);
constexpr auto x = p.getX();
```

Списки инициализации

```
// constructors
struct SequenceClass {
      SequenceClass(std::initializer_list<int> list);
};
SequenceClass someVar = {1, 4, 5, 6};
// functions
void FunctionName(std::initializer_list<float> list);

FunctionName({1.0f, -3.45f, -0.4f});

// containers
vector<string> v = { "xyzzy", "plugh", "abracadabra" };
vector<string> v{ "xyzzy", "plugh", "abracadabra" };
```

`std::initializer_list<>` может быть создан только статически с использованием синтаксиса { }, неизменяем.

Универсальная инициализация

```
struct BasicStruct {
    int x;
    double y;
};

struct AltStruct {
    AltStruct(int x, double y) : x_(x), y_(y) {}
    int x_;
    double y_;
};

BasicStruct var1{5, 3.2}; // инициализация структуры
AltStruct   var2{2, 4.3}; // вызов конструктора

BasicStruct GetString() { return {6, 4.2}; } // тип не обязателен

std::vector<int> theVec{4}; // [4], std::initializer_list приоритетнее
```

Range-based for

Синтаксическая конструкция для работы с контейнерами.

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

vector<int> my_vector = {1, 2, 3, 4, 5};
for(int &x : my_vector) {
  x *= 2;
}
```

Применимо к C-массивам, спискам инициализаторов и любым другим типам, для которых определены функции begin() и end(), возвращающие итераторы.

Reference wrapper

Позволяет обернуть ссылку для передачи в шаблон.

```
void foo (int & r)  { r++; }

template<class F, class P>
void bar(F f, P t)  {
    f(t);
}

int main() {
    int i = 0;
    // bar<void (int & r), int>
    bar(foo, i);
    std::cout << i << std::endl; // 0

    // bar<void(int & r),reference_wrapper<int>>
    bar(foo, std::ref(i));
    std::cout << i << std::endl; // 1
}
```

std::function

Класс для хранения указателей на функции и функторов.

```
struct int_div {
  float operator()(int x, int y) const { return float(x)/y; };
};

void test() {
    std::function<float (int, int)> f = int_div();
    std::cout << f(5, 3) << std::endl;
}
```

Позволяет работать и с указателями на методы.

```
struct X { int foo(int i) {return i * i}; };

void test() {
    std::function<int (X*, int)> f = &X::foo;
    X x;
    f(&x, 5);
}
```

Лямбда-выражения

```
std::function<int (int, int)> f = [](int x, int y) { return x + y; }
// то же но с указанием типа возвращаемого значения
f = [](int x, int y) -> int { int z = x + y; return z; }
// C++14
f = [](auto x, auto y) { return x * y; }
```

Можно захватывать *локальные* переменные.

```
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 захватывать this
auto lambdaFun = [this]() { this->privateMethod(); };
```

Различные виды захвата

Могут быть разные типы захвата, в т.ч. смешанные:
`[]`, `[x, &y]`, `[&]`, `[=]`, `[&, x]`, `[=, &z]`
Не стоит использовать захват по-умолчанию (`[&]` или `[=]`).

```
std::function<bool(int)> create_filter(int v1, int v2) {
    auto d  = v1 / v2;
    // захватывает ссылку на локальную переменную
    return [&] (int i) { return i % d == 0; }
}
```

```
struct C {
    std::function<bool(int)> create_filter() const {
        // захватывает this, а не d
        return [=] (int i) { return i % d == 0; }
    }
    int d;
};
```

Move-захват доступен только в C++14 (можно реализовать с помощью `std::bind`).