

Новшества C++

Амиран Мстоян

AI System Engineer (C++) в Huawei
Moscow Research Center



Проверка связи





Если у вас нет звука:

- убедитесь, что на вашем устройстве и на колонках включён звук
- обновите страницу вебинара (или закройте страницу и заново присоединитесь к вебинару)
- откройте вебинар в другом браузере
- перезагрузите компьютер (ноутбук) и заново попытайтесь зайти



Поставьте в чат:

-  если меня видно и слышно
-  если нет

Амиран Мстоян

О спикере:

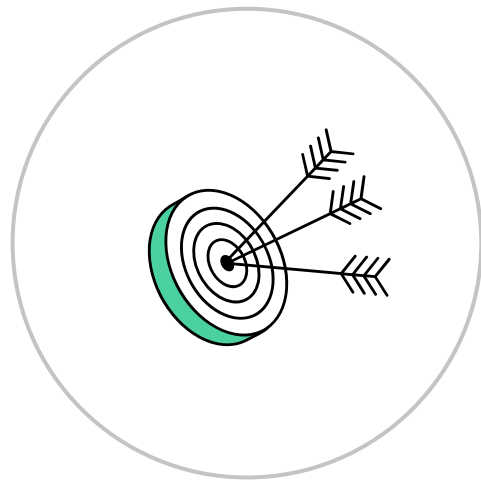
AI System Engineer (C++) в Huawei Moscow Research Center

Математик-алгоритмист лаборатории научного центра при МФТИ



Цели занятия

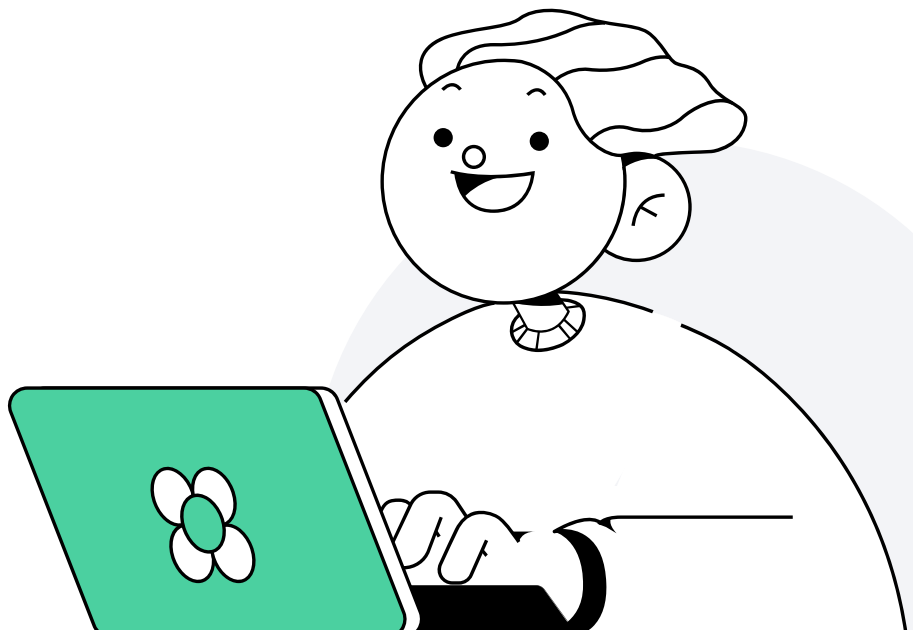
- Разберём основные новшества 11, 14, 17 стандартов
- Увидим, что C++ может быть красивым и понятным
- Познакомимся с мощностью современного C++



План занятия

- 1 Стандарты языка C++
- 2 Нововведения C++ 11
- 3 Нововведения C++ 14
- 4 Нововведения C++ 17
- 5 Итоги
- 6 Домашнее задание

*Нажми на нужный раздел для перехода



Стандарт языка C++



1

Откуда берутся стандарты

Существует Комитет ISO по стандартизации языка C++:

- создает новые версии языка
- утверждает международные стандарты



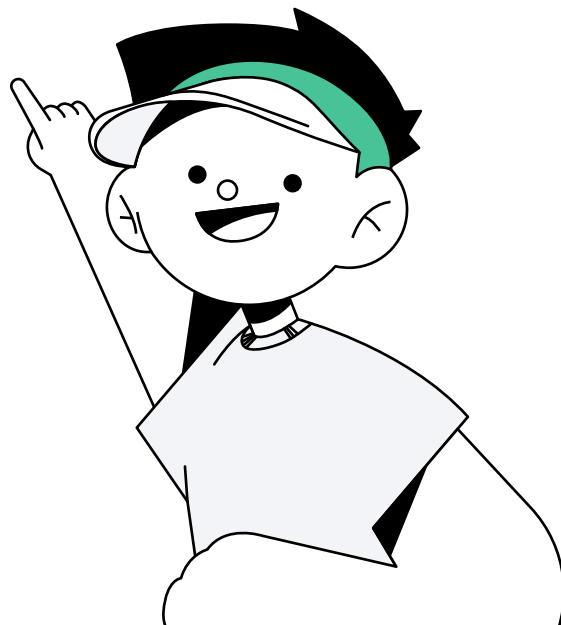
Нововведения C++ 11



2

Нововведения C++11

- Ключевое слово `auto`
- Range based for
- Лямбда-функции
- Спецификаторы `override` и `final`
- Семантика перемещения
- Умные указатели
- Кортежи
- `constexpr`



Ключевое слово auto

Можно явно не указывать тип переменной при инициализации - компилятор выведет тип за нас

```
auto integer_value = 5;           // y integer_value тип int
auto floating_point_value = 4.0;  // y floating_point_value тип double
auto my_object = new my_class();  // y my_object тип my_class*
```

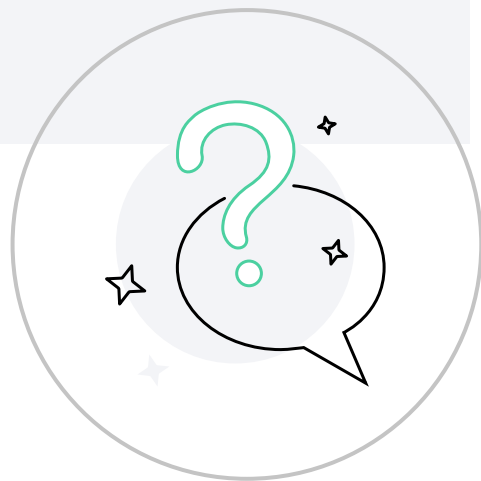
Будет работать и с возвращаемыми значениями функций:

```
int add(int first, int second) {
    return first + second;
}
auto result = add(3, 4);
```

Ключевое слово auto

Вопрос: а так корректно использовать?

```
void swap_function(auto lhs, auto rhs)
{
    auto tmp = lhs;
    lhs = rhs;
    rhs = tmp;
}
```



Ключевое слово auto

Вопрос: а так корректно использовать?

```
void swap_function(auto lhs, auto rhs)
{
    auto tmp = lhs;
    lhs = rhs;
    rhs = tmp;
}
```

Ответ: нет, так как компилятор не может определить типы переменных `lhs`, `rhs` во время компиляции



Ключевое слово auto

А зачем это нужно?

1. **Улучшает абстракцию:** принуждает думать в терминах интерфейсов, а не реализаций.
2. Заставляет **инициализировать** переменные
3. **Безопасность и быстрота:** нельзя опечататься в имени типа или случайно использовать другой, к которому есть неявное приведение
4. **Стабильность:** если тип функции изменится, то меньше случаев, когда код придется менять

Ключевое слово auto

А зачем это нужно?

5. Использование auto **сокращает код**. Часто помогает при работе с std. Например, есть прототип некоторой функции:

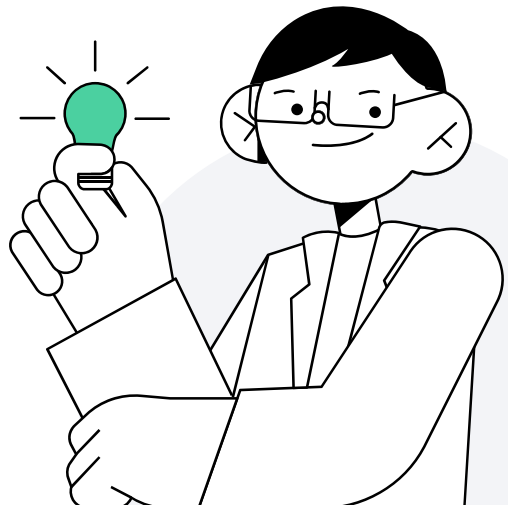
```
std::map<std::string, std::vector<int>> some_func()
```

И каждый раз результат выполнения функции нужно будет сохранять в переменную типа `std::map<std::string, std::vector<int>>`. А можно написать короче:

```
auto result = some_func()
```

Ключевое слово auto. Особенности

- Переменная auto должна быть проинициализирована
- Переменная auto не может быть членом класса
- Переменная auto не может быть параметром функции (до C++14)
- Тип auto не может быть возвращаемым типом функции (до C++11)



Range-based for

Теперь стандартный for можно использовать как foreach (цикл, основанный на диапазоне)

Синтаксис:

```
for (элемент : контейнер)  
    какое-то действие
```

Пример:

```
for (int number : my_array) // итерация по массиву math  
    std::cout << number << ' '; //доступ к элементу массива
```

Идеально использовать с auto:

```
for (const auto& number : my_array) // итерация по массиву math  
    std::cout << number << ' '; //доступ к элементу массива
```


Range-based for

Особенности

- Может работать не только с массивами, а с любыми коллекциями, которые имеют методы `begin`, `end`
- Объект итерации должен поддерживать операцию `++`
- Объект итерации должен поддерживать оператор `!=`
- Можно разыменовывать объект итерации (оператор `*`)



Лямбда-функции

Позволяет определить функцию внутри другой функции “на лету”, не засоряя глобальную область видимости.

Синтаксис:

```
① [=] ② () ③ mutable throw ④ () -> ⑤ int
{
    //do something
    ⑥ return x + y;
}
```

- ① Список захвата переменных
- ② Параметры функций
- ③ Изменяемая спецификация (необязательный параметр)
- ④ Спецификация исключений (необязательный параметр)
- ⑤ Возвращаемый тип функции (условно обязательный параметр)
- ⑥ Тело функции

Лямбда-функции

Зачем это нужно?

Часто бывает ситуация, когда необходимо некоторую функцию передать как параметр в другую функцию. Например, в стандартную функцию `std::sort` для сортировки ваших классов.

Старое решение: объявить функцию где-то и передать по указателю
Минусы: загромождает код и область видимости

Новое решение: использовать лямбда-функции

Пример без лямбда-функций.

```
#include <iostream>
#include <vector>
#include <algorithm>

struct exam_results {
    std::string name;
    int score;
};

bool comparator(const exam_results& lhs, const exam_results& rhs) {
    return lhs.score > rhs.score;
}

int main() {

    std::vector<exam_results> results{
        {"Alex", 55}, {"Igor", 87}, {"Alena", 69}
    };
    std::sort(results.begin(), results.end(), comparator);

    return 0;
}
```

Пример с лямбда-функциями

```
#include <iostream>
#include <vector>
#include <algorithm>

struct exam_results {
    std::string name;
    int score;
};

int main() {

    std::vector<exam_results> results{
        {"Alex", 55}, {"Igor", 87}, {"Alena", 69}
    };
    std::sort(
        results.begin(), results.end(),
        [](const exam_results& lhs, const exam_results& rhs){
            return lhs.score > rhs.score;
        }
    );

    return 0;
}
```

Лямбда-функции. Список захвата

Список захвата определяет переменные, которые будут доступны внутри лямбда-функции.

Простейшая лямбда-функция:

```
[=] {}
```

Пример:

```
int val = 0;
auto func = []() {
    return val*val;
};
```

Вопрос: скомпилируется ли программа?

Лямбда-функции. Список захвата

```
int val = 0;
auto func = []() {
    return val*val;
};
```

Ответ: нет, так наша лямбда-функция ничего не знает про переменную val.

Нужно написать так:

```
int val = 0;
auto func = [val]() {
    return val*val;
};
```

Лямбда-функции. Список захвата

Можно захватить все переменные из области видимости, где объявлена лямбда-функция, по ссылке [&] или значению [=].

Можно часть переменных захватывать по ссылке, часть по значению

```
int val= 0;
auto func = [&val]() {
    val++;
};
func(); //вызов лямбда-функции
std::cout << val;
```

При этом значение val будет равно 0.

Для того, чтобы захватить методы класса, нужно захватить this.

Лямбда-функции. Параметры

Здесь все также как в обычных функциях. Сколько параметров укажем при создании лямбды, столько обязаны использовать при её вызове.

```
int val = 3;
auto add_to_val= [val](double x, double y) {
    return val + x + y;
};

std::cout << add_to_val(4, 5); //вызов лямбда-функции
```

Значение val будет равно 12

Лямбда-функции. Спецификатор mutable

Данный спецификатор говорит о том, что внутри лямбда-функции можно изменять переменные, захваченные не по ссылке.

```
int val1 = 0, val2 = 0
auto func = [val1, &val2]() mutable {
    val1++;
    val2 = val1;
};
func(); //вызов лямбда-функции
std::cout << val1 << val2;
```

При этом значение val1 будет равно 0, а val2 = 1, так как val1 захватили по значению.

Вопрос на засыпку

Какая функция называется
виртуальной?

Напишите в чат

Спецификатор override

Рассмотрим следующий код

```
class base {
public:
    virtual void some_func(int val) { std::cout << val << " in base class" << std::endl; }
};

class derived : public base{
public:
    void some_func(double val) { std::cout << val << " in derived class" << std::endl; }
};

int main() {

    base* obj = new derived();
    obj->some_func(2.0);
    delete obj;
    return 0;
}
```

Что выведет программа?

Спецификатор `override`

Хочется ответить, что **“2 in derived class”**. Но это не так
На экране появится **“2 in base class”**

Все дело в том, что мы опечатались и на самом в классе `derived` создали новую функцию (перепутав тип входного аргумента у `some_func`), а не переопределили функцию базового класса.


В больших программах такие ошибки будет довольно сложно найти.

Спецификатор override

Решение: использовать спецификатор override - компилятор предупредит нас

```
class base {
public:
    virtual void some_func(int val) { std::cout << val << " in base class" << std::endl; }
};

class derived : public base{
public:
    void some_func(double val) override { std::cout << val << " in derived class" << std::endl; }
};
```



Получим ошибку компиляции:

member function declared with 'override' does not override a base class member


Спецификатор final

Если вдруг мы захотим, чтобы функцию нельзя было переопределить, то можно написать следующий код

```
class base {
public:
    virtual void some_func(int val) { std::cout << val << " in base calss"<< std::endl; }
};

class derived1 : public base{
public:
    void some_func(int val) override final{ std::cout << val << " in derived calss" <<
std::endl; }
};

class derived2 : public derived1 {
public:
    void some_func(int val) override { std::cout << val << " in derived calss" << std::endl; }
};
```




Спецификатор final

Получим ошибку компиляции: cannot override 'final' function "derived1::some_func"

```
class base {
public:
    virtual void some_func(int val) { std::cout << val << " in base calss"<< std::endl; }
};

class derived1 : public base{
public:
    void some_func(int val) override final{ std::cout << val << " in derived calss" <<
std::endl; }
};

class derived2 : public derived1 {
public:
    void some_func(int val) override { std::cout << val << " in derived calss" << std::endl; }
};
```





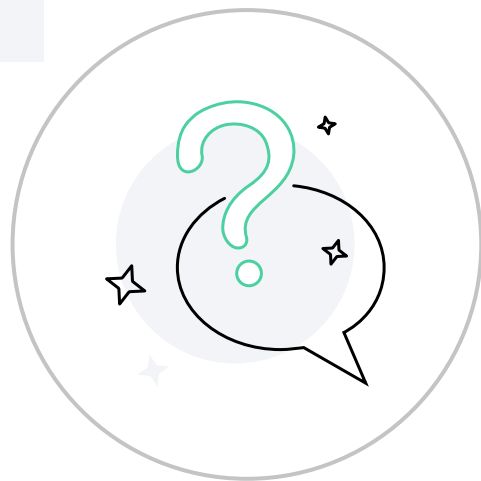
Семантика перемещения - набор правил и средств языка C++, предназначенных для перемещения объектов, время жизни которых скоро истекает, вместо их копирования

Семантика перемещения

Рассмотрим функцию, которая принимает 2 массива и меняет их местами

```
void swap(std::vector<int>& lhs, std::vector<int>& rhs)
{
    std::vector<int> tmp = lhs;
    lhs = rhs;
    rhs = tmp;
}
```

Вопрос: Что не так с этой функцией ?



Семантика перемещения

```
void swap(std::vector<int>& lhs, std::vector<int>& rhs)
{
    std::vector<int> tmp = lhs;
    lhs = rhs;
    rhs = tmp;
}
```

Ответ: Если передать в эту функцию большие массивы, например, по 1'000'000 элементов, то в силу особенностей реализации `std::vector` будет 3'000'000 лишних операций копирования



Семантика перемещения

```
void swap(std::vector<int>& lhs, std::vector<int>& rhs)
{
    std::vector<int> tmp = std::move(lhs);
    lhs = std::move(rhs);
    rhs = std::move(tmp);
}
```

Решение: использовать `std::move`

Мы указываем компилятору, что эти объекты могут быть перемещены, и он преобразует эти объекты к специальной ссылке `rvalue` (узнаете позже)



Умные указатели

Рассмотрим следующий код. Какие недостатки?

```
int some_func() {  
    int* my_data = new int[600];  
    // Проверка какого-либо условия  
    if (some_condition) {  
        // что-то делаем  
        return 1;  
    }  
    // Работаем дальше  
    if (another_condition) {  
        test_func();  
        return 2;  
    }  
    // Удаляем память  
    delete[] my_data;  
}
```

Умные указатели

- Если выполнится какое-либо из условий *some_condition, another condition*, то выделенная память не удалится. Нужно в каждом условии писать удаление - это дублирование кода
- Если вдруг в функции *test_func* возникнет исключение, то опять память не удалится, и возникнут утечки памяти.

Что хочется?

Иметь такой объект, который самостоятельно заботится о высвобождении своих ресурсов

Умные указатели

Для этого в std введены следующие классы

`auto_ptr`

Старый умный
указатель - **не
использовать!**
(Удалён в C++17)

`unique_ptr`

Предоставляет
уникальное
владение объектом

`shared_ptr`

Предоставляет
совместное
владение объектом

`weak_ptr`

Решает некоторые
проблемы, которые
возникают с
`shared_ptr`

Кортежи

Коллекция фиксированного размера, которая позволяет хранить элементы различных типов

Создание кортежа:

```
std::tuple<std::string, int, char, double> test("str", 4, 'a', 3.0) // создание через конструктор  
auto simple_tuple = std::make_tuple("str", 4, 'a', 3.0) // создание с помощью make_tuple
```

Получение элементов

```
std::string str = std::get<0>(simple_tuple); // "str"  
int num = std::get<1>(simple_tuple); // 4  
char symbol = std::get<2>(simple_tuple); // a  
double num_double = std::get<3>(simple_tuple); // 3.0  
  
// использовать функцию std::tie  
std::tie(str, num, symbol, num_double) = simple_tuple;
```


constexpr

С помощью этого спецификатора можно создавать переменные, функции и объекты, которые будут рассчитаны на этапе компиляции!

constexpr - переменные

```
// Синтаксис: constexpr тип = expression // expression должно быть известно на этапе компиляции  
constexpr double pi = 3.14159265; // значение переменной будет вычислено на этапе компиляции
```

Плюсы:

- часть вычислений можно переложить на компиляцию

Минусы:

- невозможно дебажить такие выражения
- много ограничений

constexpr

constexpr - функции

```
// Синтаксис: constexpr тип_возвращаемого_значения имя_функции
constexpr int sum (int v1, int v2) {
    return v1 + v2;
}
constexpr int val = sum (5, 12); // значение переменной будет посчитано на этапе компиляции

int new_sum (int v1, int v2){
    return v1 + v2;
}
constexpr int val_new = new_sum(4, 5) // ошибка: функция new_sum не является constexpr-выражением
```

constexpr ограничения

constexpr - переменные

- Тип должен быть литеральным (скалярный тип, указатель, массив скалярных типов)
- Должно быть сразу присвоено значение или вызван constexpr конструктор



constexpr ограничения

constexpr - функции

- Не может быть виртуальной
- Должна возвращать литеральный тип (В c++14 можно вернуть void)
- Все параметры должны иметь литеральный тип
- Тело функции должно содержать ровно один return, который может содержать литералы или constexpr функции



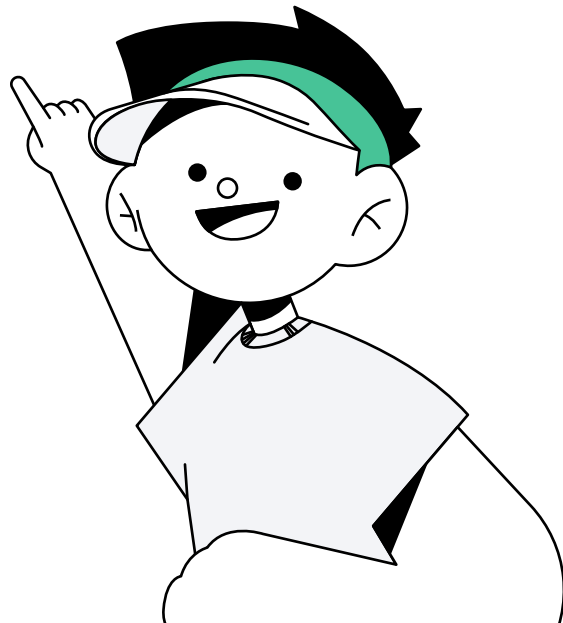
Нововведения C++ 14



3

Нововведения C++14

- Обобщённые лямбда-функции
- Инициализаторы лямбда захвата
- Автоматический вывод типов для функций
- Разделители разрядов



Обобщённые лямбда-функции

Рассмотрим лямбда-функцию, которая используется для сортировки контейнера, хранящего целые числа:

```
[](int lhs, int rhs) { return lhs > rhs; }
```

А если хотим добавить тоже самое, но для контейнера, хранящего вещественные числа? Придется добавить еще одну функцию, отличающуюся только входными типами

```
[](double lhs, double rhs) { return lhs > rhs; }
```

Дублирование кода - это плохо

Обобщённые лямбда-функции

Решение: использовать шаблоны (о них узнаете позже) или использовать auto

```
[](auto lhs, auto rhs) { return lhs > rhs; }
```

Код будет работать для всех типов данных, у которых определён оператор >

Таким образом, мы избежали дублирования кода

Инициализаторы лямбда-захвата

Как мы знаем, лямбда функции C++ 11 могут захватывать переменные путем передаче по ссылке или по значению. C++14 позволяет захватывать переменные, инициализировав их произвольным значением:

```
auto value {0};  
auto function = [value = 5] {return value;}; // вернёт 5
```

Благодаря этой возможности, можно использовать захват с перемещением

```
auto tmp = std::vector<int>(100000, 0);  
auto func = [val = std::move(tmp)]{ return val[0]; };
```

Автоматический вывод типов для функций

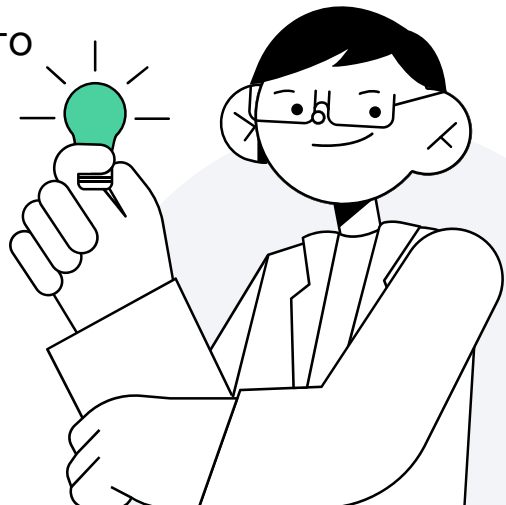
Можно объявить тип функции, как `auto`, он будет выведен позже.

```
auto Correct(int i) {  
    if (i == 1)  
        return i; // в качестве типа возвращаемого значения выводится int  
    else  
        return Correct(i - 1) + i; // теперь можно вызывать  
}  
  
auto Wrong(int i) {  
    if(i != 1)  
        return Wrong(i - 1) + i; // неподходящее место для рекурсии. Нет предшествующего возврата.  
    else  
        return i; // в качестве типа возвращаемого значения выводится int  
}
```

Автоматический вывод типов для функций.

Особенности

- Если несколько точек возврата, то возвращаемое значение должно иметь общий тип
- Можно использовать рекурсию, но рекурсивный вызов должен выполняться после одного возврата функции
- Использовать можно после определения, предварительного объявления недостаточно



Разделители разрядов

Можно использовать апостроф для разделения разрядов в числах
Это улучшает читаемость кода

```
auto big_integer = 1'000'000'000;  
auto floating_point = 0.00'021'395'74;  
auto binary_literal = 0b00100'0110'0110;
```

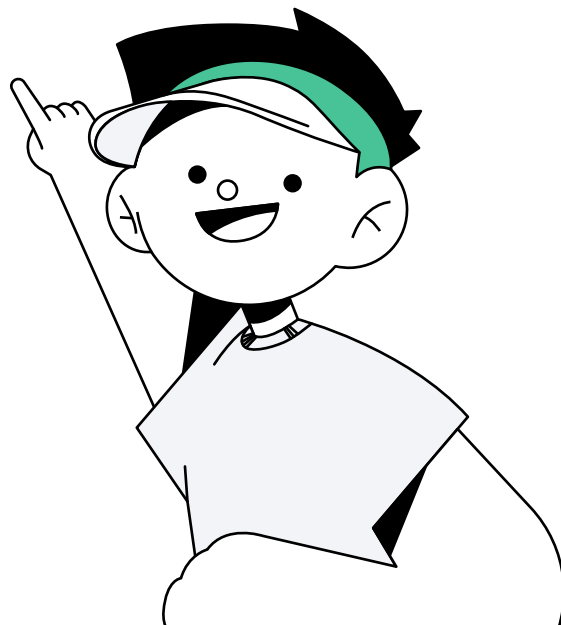
Нововведения C++ 17



4

Нововведения C++17

- Структурированные привязки
- `std::variant`
- `std::optional`
- `std::any`



Структурированные привязки

Можно обеспечить множественное присваивание от структурированных типов. Раньше нужно было каждую переменную сохранять по-отдельности, теперь это не нужно

Однако вложенные структуры так не получится декомпонировать.

```
struct Person {  
    std::string name;  
    unsigned score;  
    std::string birthday;  
};  
  
Person p1{"Alex", 70, "03.11.1995"};  
// было  
auto name = p1.name;  
auto score = p1.score;  
auto birthday = p1.birthday;  
  
// стало  
auto [name, score, birthday] = p1; // код стал компактнее
```

std::variant

Это такой класс, экземпляр которого в данный момент времени содержит **одно значение** из альтернативных типов.

```
std::variant<unsigned, std::string> age; // возраст – можем хранить как число или как строку с датой рождения
```

```
age = 51u; // работаем как с числом  
auto age_int = std::get<unsigned>(age); // получаем значение нужного типа с помощью функции get
```

```
age = "03.05.2014"; // работаем как со строкой  
auto age_string = std::get<std::string>(age);
```

```
// здесь выбросится исключение bad_variant_access  
// так как сейчас в переменной age находится тип string  
auto tmp = std::get<unsigned>(age);
```


std::variant

Дополнительные функции, которые делают работу удобнее

```
std::variant<unsigned, std::string> age;

age = 51u; // работаем как с числом

// holds_alternative
bool is_string = std::holds_alternative<std::string>(age); // false
bool is_unsigned = std::holds_alternative<unsigned>(age); // true

// get_if
auto try_string = std::get_if<std::string>(&age) // nullptr
auto tmp = std::get_if<unsigned>(age); // unsigned int *
```

std::optional

Это класс, который может содержать значение, а может и нет. С помощью него удобно показать, что функция может не возвращать значение.

```
std::optional<double> my_sqrt(double val) {  
    if (val < 0) // корень из отрицательного числа не вычисляем  
        return {};  
    else  
        return sqrt(val);  
}  
  
// value_or(arg) - возвращает значение, если хранится, либо число в arg  
double v = my_sqrt(-5).value_or(0); // 0  
double v1 = my_sqrt(4).value_or(0); // 2  
  
// has_value - проверяет, содержится ли значение  
bool has_value = my_sqrt(-5).has_value(); // false  
bool has_value1 = my_sqrt(4).has_value(); // true
```

std::any

Это класс, который может хранить значения разных копируемых типов (не путать с auto)!

- Можно воспользоваться функцией *type*, которая вернёт объект *typeid*, у которого можем узнать тип
- Получить значение можно с помощью *any_cast*. Если значение преобразовываем не к тому типу, который находится, то получим исключение *bad_cast*

```
std::any a = 5;
std::cout << a.type().name() << ": " << std::any_cast<int>(a) << '\n'; // int: 5
a = 3.14;
std::cout << a.type().name() << ": " << std::any_cast<double>(a) << '\n'; // double: 3.14
a = false;
std::cout << a.type().name() << ": " << std::any_cast<bool>(a) << '\n'; // bool: false

// можно проверить содержит ли значение
bool has_value = a.has_value()

// можно хранить значения любых типов в одном контейнере
std::vector<any> v { "test_string", 4.12, 5, false};
```

Итоги



Итоги занятия

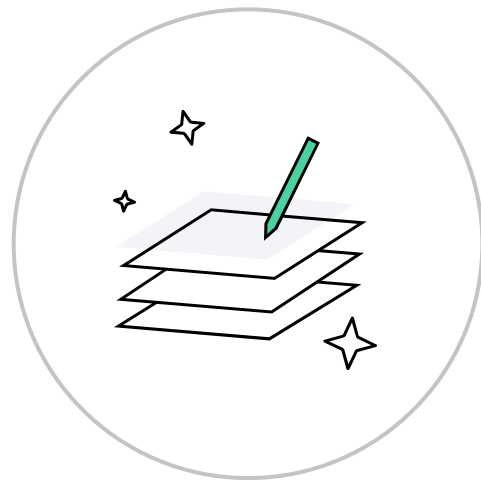
- 1 Познакомились с основными возможностями 11 стандарта (умные указатели, ключевое слово `auto`, лямбда-функции, семантика перемещения)
- 2 Познакомились с основными возможностями 14 стандарта (обобщенные лямбда функции, вывод типов для функций, разделители разрядов)
- 3 Познакомились с основными возможностями 17 стандарта (структурированные привязки, `std::variant`, `std::any`, `std::optional`)



Домашнее задание

Давайте посмотрим ваше [домашнее задание](#).

- 1 Вопросы по домашней работе задавайте в чате группы
- 2 Задачи можно сдавать по частям
- 3 Зачёт по домашней работе ставят после того, как приняты все задачи



Дополнительные материалы

- [Рекомендации по C++](#)
- Мейерс Скотт. Эффективный и современный C++: 42 рекомендации по использованию C++11 и C++14
- Яцек Галовиц. C++17 STL. Стандартная библиотека шаблонов
- [Таблица поддержки нововведений стандартов C++11, C++14, C++17 в разных компиляторах](#)



Задавайте вопросы и пишите отзыв о лекции

Амиран Мстоян

AI System Engineer (C++) в Huawei
Moscow Research Center

