Лекция 1. Переход к современному C++11

ИУ8

September 1, 2016

Обзор курса

Что нас ждёт

- Дифференцируемый зачет
- ~~8~~ 17 лекций
- ~~8~~ 24 семинара
- 17 лабораторных работ
- Домашнее задание (подробности у семинаристов)

Обзор курса

Программа курса

В ходе лекций будут затронуты следующие темы:

- Введение в C++11
- Оптимизация возвращаемых значений
- Функторы, лямбда
- Умные указатели
- Многопоточность
- Структуры данных без блокировок
- и многое другое...

Рекомендованная литература

Дополнительные материалы для изучения

1. С. Мэйерс - Эффективное использование C++
2. С. Мэйерс - Эффективный и современный C++
3. Н. Джосаттис - C++ стандартная библиотека. Второе издание
4. Липпман - Язык программирования C++. Базовый курс
5. Бьерн Страуструп - Язык программирования C++
6. Б. Эккель, Ч. Эллисон - Философия C++
7. Уильямс - Параллельное программирование на C++ в действии
8. Александреску - Современное проектирование на C++
9. Саттер - Решение сложных задач на C++
10. Блог Алёны C++ - http://alenacpp.blogspot.ru/
11. ISO/IEC 14882:2011 C++11 или Working Draft, Standard for Programming Language C++
12. Standard C++ site - https://isocpp.org/

Переход к современному C++

На текущей лекции рассмотрим

- auto
- nullptr
- range-for
- override
- final
- static_assert
- initialize_list
- enum class
- constexpr
- etc.

Переход к современному C++11: auto

Определение типа при объявлении

```
1 auto b = true;
2 auto ch = 'x';
3 auto i = 123;
4 auto d = 1.2;
5 auto z = sqrt(y);
6 // std::unique<std::vector<int>> ptr =
7 //     std::make_unique<std::vector<int>>();
8 auto ptr = std::make_unique<std::vector<int>>();
```

Переход к современному C++11: auto

Есть три случая:

1. Спецификатор типа представляет собой ссылку или указатель, но не универсальную ссылку
2. Спецификатор типа представляет собой универсальную ссылку
3. Спецификатор типа не является ни ссылкой, ни указателем

Вывод типа auto

```
1 auto x = 42; // case 3
2 const auto cx = x; // case 3
3 const auto& rx = x; // case 1 (rx isn't universal reference)
4
5 // case 2:
6 auto&& uref1 = x;
7 auto&& uref2 = cx;
8 auto&& uref3 = 27;
```

Переход к современному C++11: auto

Без использования auto

```cpp
// C++98
int x1 = 27;
int x2(27);
// C++11
int x3 = {27};
int x4{27};
```

С использования auto

```cpp
auto x1 = 27;
auto x2(27);
auto x3 = {27};
auto x4{27};
// auto x5 = {1, 2, 3.0};
// error: impossible to deduce T
```

Различие вывода template и auto

```cpp
auto x = {11, 23, 9}; // x - std::initializer_list<int>

template<typename T>
void f(T param);
// f({11, 23, 19}); // error: impossible to deduce T

template<typename T>
void f(std::initializer_list<T> initList);

f({11, 23, 19});
// T is int.
// initList is std::initializer_list<int>
```

Переход к современному C++11: decltype

Для данного имени или выражения decltype сообщает вам тип этого имени или выражения

Использование decltype

```
template<typename Container, typename Index>
auto authAndAccess(Conteiter c, Index i)
  -> decltype(c[i]) {
  authenticate();
  return c[i];
}
```

C++14: decltype(auto)

```
Widget w;
const Widget& cw = w;
auto myWidget1 = cw; // myWidget1 - Widget
decltype(auto) myWidget2 = cw; // myWidget2 - const Widget &
int x = 0;
```

Полезный хак

```
1 template<typename T>
2 class TD;
3
4 TD<decltype(x)> xType; // error
5 TD<decltype(y)> yType; // error
```

Переход к современному C++11: Фигурная инициализация

Фигурная инициализация

```
1  int x{ 1 };
2  int x = {1};
3  std::vector v{1, 2, 3, 4, 5};
4
5  struct Foo
6  {
7      std::string str;
8      double x;
9      int y;
10 };
11
12 Foo foo {"C++11", 4.0, 42}; // {str, x, y}
13 Foo bar {"C++11", 4.0}; // {str, x}, y = 0
```

Переход к современному C++11: Фигурная инициализация

initializer_list<> в конструкторе

```
1 template<class T>
2 struct Foo
3 {
4 // ...
5     Foo(std::initializer_list<int> list) {
6       // ...
7   }
8 // ...
9 };
```

Переход к современному C++11: Фигурная инициализация

Если у компилятора есть любой способ истолковать вызов как конструктор с **std::initializer_list**, он использует эту возможность!

Странности фигурной инициализации

```
1  struct Widget {
2    Widget(int i, bool b); // first .ctor
3    Widget(std::initializer_list<double> il); // second .ctor
4    Widget(const Widget&); // copy .ctor
5    operator floct() const;
6  };
7  // client code:
8  Widget w1(10, true); // first .ctor is called
9  Widget w2{10, true}; // second .ctor is called
10 Widget w3(w1); // copy .ctor is called
11 Widget w4{w1}; // second .ctor is called
```

Переход к современному C++11: Фигурная инициализация

Странности фигурная инициализация

```
1 struct Widget {
2   Widget(int i, bool b);
3   Widget(int i, double d);
4   Widget(std::initializer_list<bool> il);
5   Widget(const Widget&);
6 };
7 // client code:
8 Widget w1(10, 514.1234); // ok
9 Widget w2{10, 514.1234}; // error: Requires narrowing conversion
```

Переход к современному C++11: Фигурная инициализация

Странности фигурная инициализация

```
1  struct Widget {
2    Widget(int i, bool b); // first .ctor
3    Widget(int i, double d); // second .ctor
4    Widget(std::initializer_list<std::string> il);
5    Widget(const Widget&);
6  };
7  // client code:
8  Widget w1(10, true); // first .ctor is called
9  Widget w2{10, true}; // first .ctor is called
10 Widget w3(10, 514.1234); // second .ctor is called
11 Widget w4{10, 514.1234}; // second .ctor is called
```

Переход к современному C++11: Фигурная инициализация

Проблемы интерфейсов класса с initializer_list

```
1 std::vector v{10, 10};
2 for(std::vector<int>::iterator it
      = v.begin();
3   it != v.end(); ++it)
4   std::cout << *it << ' ';
```

```
1 // 10 10
```

```
1 std::vector v(10, 10);
2 for(std::vector<int>::iterator it
      = v.begin();
3   it != v.end(); ++it)
4   std::cout << *it << ' ';
```

```
1 // 10 10 10 10 10 10 10 10 10 10
```

Переход к современному C++11: Новые спецификаторы

Новые спецификаторы в C++11

- override
- final
- delete
- default
- noexcept
- ссылочный квалификатор

Переход к современному C++11: override

Компилятор, обнаружив **override**, проверяет существование *виртуального* метода с данной сигнатурой в базовом классе. Если же такого метода нет — выдает ошибку.

problem code

```
1  struct A {
2    virtual void
3    m1(const char * str) {
4      std::cout << "A::" << str;
5    }
6  };
7  struct B : public A {
8    virtual void
9    m1(char * str) {
10     std::cout << "B::" << str;
11   }
12 };
13 int main() {
14   A * ptr = new B();
15   ptr->m1("abc");
16 }
1  // std::cout << A::abc
```

with override

```
1  struct A {
2    virtual void
3    m1(const char * str) {
4      std::cout << "A::" << str;
5    }
6  };
7  struct B : public A {
8    virtual void
9    m1(char * str) override {
10     std::cout << "B::" << str;
11   }
12 };
13 int main() {
14   A * ptr = new B();
15   ptr->m1("abc");
16 }
17 // compilation error
```

Переход к современному C++11: final

Спецификатор **final** позволяет запрещать в классах-наследниках переопределение определенных методов. Этот спецификатор также запрещает наследование от некоторого класса

Запрет наследования

```
1  struct A final {
2    virtual void m1() {
3      std::cout << "A::";
4    }
5  };
6
7  // a 'final' class type cannot be
       used as a base class
8  struct B : public A {
9
10   void m1() override {
11     std::cout << "B::";
12   }
13 };
14 int main() {
15   A * ptr = new B();
16   ptr->m1("abc");
17 }
```

Запрет переопределения

```
1  struct A {
2    virtual void m1() final {
3      std::cout << "A::";
4    }
5  };
6
7  struct B : public A {
8
9  // function declared as 'final'
       can't be overridden by B::m1
10   void m1() override {
11     std::cout << "B::";
12   }
13 };
14 int main() {
15   A * ptr = new B();
16   ptr->m1("abc");
17 }
```

Переход к современному C++11: delete

Спецификатор **delete** призван пометить те методы, работать с которыми нельзя. То есть, если программа ссылается явно или неявно на эту функцию — ошибка на этапе компиляции. Запрещается даже создавать указатели на такие функции.

C++98 : хаки

```
template <class charT, class traits = char_traits<chatT> >
class basic_ios: public ios_base{
// ...
private:
  basic_ios(const basic_ios&); // undefined
  basic_ios& operator=(const basic_ios&); // undefined
};
```

C++11 : delete

```
template <class charT, class traits = char_traits<chatT>>
class basic_ios: public ios_base{
// ...
private:
  basic_ios(const basic_ios&) = delete;
  basic_ios& operator=(const basic_ios&) = delete;
};
```

Переход к современному C++11: default

Суть **default** заключается в том, что пользователь может указать компилятору реализовать ту или иную функцию-член класса по умолчанию.

Использование default

```
1 class Foo {
2 public:
3     Foo() = default;
4     Foo(Foo &&) = default;
5     Foo& operator(Foo &&) = default;
6     Foo(const Foo &) = default;
7     Foo& operator(const Foo &) = default;
8     ~Foo() = default;
9 };
```

Переход к современному C++11: default

Специальные функции-члены

- конструктор по-умолчанию
- конструктор копирования
- оператор присваивания
- деструктор
- конструктор перемещения (введен в C++11)
- оператор перемещения (введен в C++11)

Переход к современному C++11: default

Генерация членов класса по-умолчанию

- Перемещающие операции генерируются только для классов, в которых нет явно объявленных перемещающих операций, копирующих операций и деструктора.
- Копирующий конструктор генерируется только для классов, в которых нет явно объявленного копирующего конструктора, и удаляется, если объявляется перемещающая операция. Копирующий оператор присваивания генерируется только для классов, в которых нет явно объявленного копирующего оператора присваивания, и удаляется, если объявляется перемещающую операция. Генерация копирующих операций в классах с явно объявленным деструктором является устаревшей и может быть отменена в будущем.
- Шаблоны функций-членов не подавляют генерацию специальных функций-членов.

using vs typedef

uisng используется для определения псевдонима типа. Псевдоним типа является именем, ссылающимся на ранее определённый тип. Псевдоним шаблона является именем, ссылающимся на семейство типов

Использование using

```
1 // typedef std::unique_ptr<std::unordered_map<std::string, std::string>>
      UPtrHashStrStr;
2 using UPtrHashStrStr =
3     std::unique_ptr<std::unordered_map<std::string, std::string>>;
4
5 //typedef void (*FPtr)(int, const UPtrHashStrStr&);
6 using FPtr = void (*)(int, const UPtrHashStrStr&);
```

using vs typedef

Преимущество using

```
1  template<typename T>
2  struct MyAllocVec {
3    typedef std::vector<T, MyAlloc<T>> type;
4  };
5  // client code: MyAllocVec<T>::type myVec;
6  template<typename T>
7  class Widget {
8    typename MyAllocVec<T>::type vec;
9    // ...
10 };
11
12 template<typename T>
13 using MyAllocVec = std::vector<T, MyAlloc<T>>;
14 // client code: MyAllocVec<T> myVec;
15 template<typename T>
16 class Widget {
17   MyAllocVec<T> vec;
18   // ...
19 };
```

Управление ошибками: static assertion

Если ошибка обнаружима до выполнения, ее лучше выявлять на этапе компиляции. Для этого используется **static_assert**

static_assert(A, S) печатает сообщение S как ошибку компиляции, если A **НЕ** выполняется.

```
1  // Exp 1. check integer sizec
2  static_assert(4<=sizeof(int), "integers are too small");
1
2  // Exp 2. using static_assert
3  constexpr double C = 299792.458; // km/s
4  void f(double speed) {
5    // 160 km/h == 160.0/(60*60) km/s
6    const double local_max = 160.0/(60*60);
7    // error: speed must be a constant
8    static_assert(speed < C, "can't go that fast");
9    // OK
10   static_assert(local_max < C, "can't go that fast");
11   // ...
12 }
```

Переход к современному C++11: nullptr

Ключевое слово **nullptr** обозначает нулевой указатель буквально.

Недействительный указатель

```
1 double* pd = nullptr;
2 Link* lst = nullptr; // pointer to a Link
3 int x = nullptr; // error: nullptr is not an integer
```

Переход к современному C++11: range-for

Ох уж этот C++98

```
std::vector<int> v(10, 10); // or any container
// ...
for (std::vector<int>::iterator it = v.begin();
  it != v.end(); ++it) {
// std::cout << *it << ' ';
}
```

А если использовать auto

```
std::vector<int> v(10, 10); // or any container
// ...
for (auto it = v.begin(); it != v.end(); ++it) {
// std::cout << *it << ' ';
}
```

Да здравствует C++11!

```
std::vector<int> v(10, 10); // or any container
// ...
for (auto i : v) {
// std::cout << i << ' ';
}
```

Переход к современному C++11: range-for

Цикл по набору значений

```
int v[] = {0,1,2,3,4,5,6,7,8,9};
for (int x : v) // for each x in v
  cout << x << '\n';
```

```
for (auto& x : v)
  ++x;
```

```
for (auto it = v.begin(); it != v.end(); ++it)
  ++(*it);
```

Для работы с массивами появились функции **begin()** и **end()**

Переход к современному C++11: Enumerations

enum class: строгая типизация

```
enum class Color {
  red,
  green };
enum class Traffic_light {
  green,
  red };
Color col = Color::red;
auto light = Traffic_light::red;
```

```
// error : which red?
Color x = red;
// error: that red is not Color
Color y = Traffic_light::red;
// error: this is not int
int z = Color::red;
```

```
Color z = Color::red; // OK
if(z == Color::green)
  cout<<"green";
if(z < Color::green)
  cout<<"it's red";
```

Сравнение: c-style enums

```
enum Color {
  red,
  green };
```

```
enum Traffic_light {
// error:
// redefinition of 'green'
  green,
  red };
```

```
enum Traffic_light {
  t_l_green, // ok
  t_l_red };
```

```
int color = blue; // ok
int t_l_color = t_l_green; // ok
if (t_l_color == color)
  printf("Same color!"); // WTF?
```

Переход к современному C++: const и constexpr

const

"Обещаю, что не буду изменять эту переменную" (контролируется компилятором)

constexpr

"Значение должно быть вычислено во время компиляции"

```
1 const int dmv = 17; // dmv is a named constant
2 int var = 17; // var is not a constant
3 constexpr double m1 = 1.4*square(dmv); // OK if square(17) is a constexpr
4 constexpr double m2 = 1.4*square(var); // error: var is not const
5 const double max3 = 1.4*square(var); // OK, evaluated at run time
6 double sum(const vector<double>&); // sum will not modify its argument
7 constexpr double s2 = sum(v); // error: sum(v) not constant expression
```

constexpr функция

```
1   constexpr double square(double x) { return x*x; }
```

The end

Самостоятельное изучение

- Ссылочный квалификатор
- Информация о типе во время компиляции **type_traits**
- Шаблоны с переменным числом аргументов
- Операторы явного преобразования
- Новые строковые литералы
- Пользовательские литералы
- Управление выравниванием объектов и запросы на выравнивание

Список литературы

- С. Мэйерс - Эффективный и современный C++
- C++11 - the new ISO C++ standard
  http://www.stroustrup.com/C++11FAQ.html
- http://archive.kalnitsky.org/