Курс «Проектирование больших систем на языке C++»

Лекция

lambda

decltype

Определяет тип выражения, которые передается ему в качестве аргумента

```
1.  std::string operator+(const std::string& Lhs, int Rhs) {
2.      std::ostringstream ss(Lhs, std::ios_base::out | std::ios_base::ate);
3.      ss << Rhs;
4.      return ss.str();
5.  }

6.  template <class T, class U>
7.  auto Mix(T Lhs, U Rhs) -> decltype(Lhs + Rhs) {
8.      return Lhs + Rhs;
9.  }

10. int main() {
11.     std::string str("20");
12.     std::cout << Mix(str, 12) << " and the type is “
13.               << typeid(decltype(Mix(str, 12))).name();
14.     return 0;
15. }
```

Альтернативный синтаксис функций

шаблона функции, возвращаемым типом является результат некоторого выражения

```
1.  template <typename LHS, typename RHS>
2.  decltype(std::declval<const LHS &>() + std::declval<const RHS &>())
3.  AddingFunc(const LHS &lhs, const RHS &rhs) { //несколько громоздко
4.      return lhs + rhs;
5.  }
```

Альтернативный синтаксис функций

шаблона функции, возвращаемым типом является результат некоторого выражения

```
1.  template <typename LHS, typename RHS>
2.  decltype(std::declval<const LHS &>() + std::declval<const RHS &>())
3.  AddingFunc(const LHS &lhs, const RHS &rhs) { //несколько громоздко
4.      return lhs + rhs;
5.  }

6.  template <typename LHS, typename RHS>
7.  decltype(lhs + rhs) AddingFunc(const LHS &lhs, const RHS &rhs) { //недопустимо
8.      return lhs + rhs;
9.  }
```

Альтернативный синтаксис функций

шаблона функции, возвращаемым типом является результат некоторого выражения

```
1.  template <typename LHS, typename RHS>
2.  decltype(std::declval<const LHS &>() + std::declval<const RHS &>())
3.  AddingFunc(const LHS &lhs, const RHS &rhs) { //несколько громоздко
4.      return lhs + rhs;
5.  }

6.  template <typename LHS, typename RHS>
7.  decltype(lhs + rhs) AddingFunc(const LHS &lhs, const RHS &rhs) { //недопустимо
8.      return lhs + rhs;
9.  }

10. template <typename LHS, typename RHS>
11. auto AddingFunc(const LHS &lhs, const RHS &rhs) -> decltype(lhs + rhs) { //самое оно
12.     return lhs + rhs;
13. }
```

lambda

- **Лямбда-функция** является анонимной функцией способной хранить состояние. Реализация lambda есть функтор без имени.
- **Функтор** в языке C++ называется класс переопределяющий operator()

Пример

```
1.  int main() {
2.      int First = 10;
3.      auto Printer = [First](int Second) -> int {
4.          std::cout << First + Second;
5.          return First + Second;
6.      };
7.      Printer(5);
8.      return 0;
9.  }
```

lambda

Синтаксис

```
lambda ::=
“[“ capture_list ”]” , [ “(“ args ”)” ], [ “mutable” ], [ “noexcept” ], [ -> return_type ] ,
 “{” function body “}”
```

Capture list

- Содержит имена переменных, которые должны быть видимы внутри
- Сами эти переменные должны быть видимы в месте объявления лямбда-функции
- Переменные перечисленные с модификатором ‘&’ передаются по ссылке, время их жизни должно превышать время жизни лямбда-функции
- Переменные перечисленные без модификатора ‘&’ передаются по значению, копируются
- Нельзя захватывать поля и методы класса владельца, если лямбда-функция создаётся внутри одного из методов класса

Capture list

3 специальных символа

this – внутри метода класса лямбда-функция может захватить указатель this на объект владения метода, (всегда передаётся по значению, &this – запрещено, т.к. не имеет значения).

= – копирует по значению все переменные из области видимости внутрь лямбда-функции. Не может быть использован совместно с литералом “&”. Если присутствует литерал “=”, то если внутри *списка захвата* присутствуют переменные они должны быть с модификатором “&”.

& – создает константные ссылки на все переменные области видимости лямбда-функции. Не может быть использован совместно с литералом “=”. Если присутствует литерал “&”, то если внутри *списка захвата* присутствуют переменные они должны быть без модификатора “&”.

Фактически при использовании “=” или “&” происходит захват только тех локальных переменных из области видимости, которые затем непосредственно вызваны в этой лямбда-функции или же во вложенных в ней.

Capture list

Примеры

```
1.  int x, y, z;
2.  [x, y, &z]
3.  [=, &z]
4.  [&, y]
5.  [&, &y]
6.  [&, =]
7.  [=, x]
8.  [x, x] { std::cout << x; }
9.  [=, this] () { std::cout << m_x; }
10. [&, &x]() { x = 5; }
```

Capture list

Примеры

```
1.  int x, y, z;
2.  [x, y, &z]
3.  [=, &z]
4.  [&, y]
5.  [&, &y]
6.  [&, =]
7.  [=, x]
8.  [x, x] { std::cout << x; }
9.  [&, &x]() { x = 5; }        // error, explicit captures matched default
10. [&, x]() { x = 5; }
```

Capture list

Примеры

```
1.  int x, y, z;
2.  [x, y, &z]
3.  [=, &z]
4.  [&, y]
5.  [&, &y]
6.  [&, =]
7.  [=, x]
8.  [x, x] { std::cout << x; }
9.  [&, &x]() { x = 5; }        // error, explicit captures matched default
10. [&, x]() { x = 5; }         // error, хотим изменить значение по константной ссылке
11. [&, x]() mutable { x = 5; }
```

Capture list

Примеры

```
1.  int x, y, z;
2.  [x, y, &z]                  // скопировать x и y, создать ссылку на z
3.  [=, &z]                     // скопировать x и y, создать ссылку на z
4.  [&, y]                      // создать ссылки на z и x, и для y тоже!
5.  [&, &y]                     // ошибка, y должен быть без модификатора
6.  [&, =]                      // ошибка
7.  [=, x]                      // ошибка, x должен быть с модификатором &
8.  [x, x] { std::cout << x; }  // ошибка, x повторяется
9.  [&, &x]() { x = 5; }        // error, explicit captures matched default
10. [&, x]() { x = 5; }         // error, хотим изменить значение по константной ссылке
11. [&, x]() mutable { x = 5; } // ok
```

Capture list

Пример this

```
1.  class A {
2.      int i = 0;
3.      int x = 0;
4.  public:
5.      void foo() {
6.          [this] {
7.              std::cout << i << std::endl;
8.              this->i = 3;
9.              this->bar();
10.         }();
11.     }
12.     void bar() {
13.         [=, this] () { std::cout << x; }();
14.         std::cout << i;
15.     }
16. };
```

Capture list

Пример this

```
1.  class A {
2.      int i = 0;
3.      int x = 0;
4.  public:
5.      void foo() {
6.          [this] mutable { // ошибка
7.              std::cout << i << std::endl;
8.              this->i = 3;
9.              this->bar();
10.         }();
11.     }
12.     void bar() {
13.         [=, this] () { std::cout << x; }(); // ошибка = и this
14.         std::cout << i;
15.     }
16. };
```

Capture list

Конвертация в указатель на функцию

```
1.  int i = 3;
2.  auto Statefull = [i](int j) {
3.      std::cout << i - j;
4.  };

5.  auto Stateless = [](int j) {
6.      int i = 4;
7.      std::cout << i - j;
8.  };

9.   void(*Ptr1)(int j) = Statefull;
10.  void(*Ptr2)(int j) = Stateless;
```

Capture list

Конвертация в указатель на функцию

```
1.  int i = 3;
2.  auto Statefull = [i](int j) {
3.      std::cout << i - j;
4.  };

5.  auto Stateless = [](int j) {
6.      int i = 4;
7.      std::cout << i - j;
8.  };

9.   void(*Ptr1)(int j) = Statefull;
10.  void(*Ptr2)(int j) = Stateless;

11. class Functor {
12.     int i;
13. public:
14.     void operator()(int j) { /* ... */ }
15. };
```

std::function

функциональный объект имеет ТИП

```
1.  int main() {
2.      auto g = [](int x) -> std::function<int(int)> {
3.          return [=](int y) { return x + y; };
4.      };

5.      auto h = [](const std::function<int(int)> &f, int z) {
6.          return f(z) + 1;
7.      };

8.      auto a = h(g(7), 8);

9.      std::cout << a << std::endl;
10. }
```

std::function & std::mem_fn

Метод класса

```
1.  #include <functional>
2.  #include <iostream>
3.  struct Foo {
4.      int greeting() {
5.          std::cout << "Hello!\n";
6.          return (data = 11);
7.      }
8.      void print(int i) const { std::cout << i << std::endl; }
9.      int data = 7;
10. };

11. int main() {
12.     std::function<void(const Foo&, int)> Fprint = &Foo::print;
13.     Fprint(Foo{}, 1);

14.     Foo foo;
15.     auto greet = std::mem_fn(&Foo::greeting);
16.     greet(foo);

17.     std::function<int(const Foo&)> access_data = std::mem_fn(&Foo::data);
18.     std::cout << access_data(foo);
19. }
```

Устаревшие базовые классы

Классы обозначены, как устаревшие

1. `std::unary_function<>{}`
2. `std::binary_function<>{}`

3. `std::ptr_fun`
4. `std::pointer_to_unary_function`
5. `std::pointer_to_binary_function`

Устаревшие базовые классы

Классы обозначены, как устаревшие

1. ~~**std::unary_function<>{}**~~
2. ~~**std::binary_function<>{}**~~

3. ~~**std::ptr_fun**~~
4. ~~**std::pointer_to_unary_function**~~
5. ~~**std::pointer_to_binary_function**~~

6. Адаптивный функциональный протокол, базирующийся на наследовании, вводили эти классы, был заменён лямбда-функциями и **std::bind**

7. избавление от наследования разрешало некоторые неоднозначности

8. **std::bind1st**
9. **std::bind2nd**

10. функцию от двух аргументов преобразовать в функцию от одного аргумента

std::bind

Связыватель

```
1.  void f(int n1, int n2, int n3, const int& n4, int n5) {
2.      std::cout << n1 << ' ' << n2 << ' ' << n3 << ' ' << n4 << ' ' << n5 << '\n';
3.  }
4.  int g(int n1) { return n1; }
5.  struct Foo {
6.      void print_sum(int n1, int n2) { std::cout << n1 + n2 << '\n'; }
7.      int data = 10;
8.  }

9.  int main() {
10.     using namespace std::placeholders;  // for _1, _2, _3...

11.     int n = 7;
12.     auto f1 = std::bind(f, _2, _1, 42, std::cref(n), n);
13.     n = 10;
14.     f1(1, 2, 1001);

15.     auto f2 = std::bind(f, _3, std::bind(g, _3), _3, 4, 5);
16.     f2(10, 11, 12);




17. }
```

std::bind

Связыватель

```
1.  void f(int n1, int n2, int n3, const int& n4, int n5) {
2.      std::cout << n1 << ' ' << n2 << ' ' << n3 << ' ' << n4 << ' ' << n5 << '\n';
3.  }
4.  int g(int n1) { return n1; }
5.  struct Foo {
6.      void print_sum(int n1, int n2) { std::cout << n1 + n2 << '\n'; }
7.      int data = 10;
8.  }

9.  int main() {
10.     using namespace std::placeholders;  // for _1, _2, _3...
11.     std::default_random_engine e;
12.     std::uniform_int_distribution<> d(0, 10);
13.     auto rnd = std::bind(d, e);
14.     std::cout << rnd();

15.     Foo foo;
16.     auto f3 = std::bind(&Foo::print_sum, &foo, 95, _1);
17.     f3(5);

18.     auto f4 = std::bind(&Foo::data, _1);
19.     std::cout << f4(foo) << f4(std::make_shared<Foo>(foo));
20. }
```

Применение в алгоритмах

сортировка

```
1.  template <typename T>
2.  void sort(T* begin, T* end, bool (*cmp)(const T&, const T&)) {
3.      for (T* p1 = begin; p1 != end; ++p1)
4.          for (T* p2 = p1 + 1; p2 != end; ++p2)
5.              if (cmp(*p2, *p1))
6.                  std::swap(*p1, *p2);
7.  }

8.  int main() {
9.      int junk[] = { 3, 5, 2, 8, 8, 1, 0, 15, 12, -1, 3, 4, 7 };
10.     auto *junk_end = (junk + sizeof(junk) / sizeof(junk[0]));

11.     sort(junk, junk_end, &[](const int &a, const int &b) { return a > b; });

12.     for (const auto &v : junk) std::cout << v << " ";
13.     std::cout << std::endl;
14. }
```

Функторы

сортировка

```
1.  #include <functional>

2.  template <typename T>
3.  void sort(T* begin, T* end, bool (*cmp)(const T&, const T&)) {
4.      for (T* p1 = begin; p1 != end; ++p1)
5.          for (T* p2 = p1 + 1; p2 != end; ++p2)
6.              if (cmp(*p2, *p1))
7.                  std::swap(*p1, *p2);
8.  }

9.  int main() {
10.     int junk[] = { 3, 5, 2, 8, 8, 1, 0, 15, 12, -1, 3, 4, 7 };
11.     auto *junk_end = (junk + sizeof(junk) / sizeof(junk[0]));

12.     sort(junk, junk_end, std::greater<int>{});

13.     for (const auto &v : junk) std::cout << v << " ";
14.     std::cout << std::endl;
15. }
```

Функторы

сортировка

```
1.  template <typename T>
2.  bool CompareTLess(const T&a, const T&b) { return a < b; }
```

```
3.  template <typename T>
4.  void sort(T* begin, T* end, bool (*cmp)(const T&, const T&)) {
5.      for (T* p1 = begin; p1 != end; ++p1)
6.          for (T* p2 = p1 + 1; p2 != end; ++p2)
7.              if (cmp(*p2, *p1))
8.                  std::swap(*p1, *p2);
9.  }

10. int main() {
11.     int junk[] = { 3, 5, 2, 8, 8, 1, 0, 15, 12, -1, 3, 4, 7 };
12.     auto *junk_end = (junk + sizeof(junk) / sizeof(junk[0]));

13.     sort(junk, junk_end, &CompareTLess<int>);

14.     for (const auto &v : junk) std::cout << v << " ";
15.     std::cout << std::endl;
16. }
```

Функторы

сортировка

```
1.  class TC {
2.      bool IsLess = true;
3.  public:
4.      bool operator() (int a, int b) const {  return (IsLess) ? (a < b) : (a > b); }
5.  };

6.  template <typename T>
7.  void sort(T* begin, T* end, bool (*cmp)(const T&, const T&)) {
8.      for (T* p1 = begin; p1 != end; ++p1)
9.          for (T* p2 = p1 + 1; p2 != end; ++p2)
10.             if (cmp(*p2, *p1))
11.                 std::swap(*p1, *p2);
12. }

13. int main() {
14.     int junk[] = { 3, 5, 2, 8, 8, 1, 0, 15, 12, -1, 3, 4, 7 };
15.     auto *junk_end = (junk + sizeof(junk) / sizeof(junk[0]));

16.     sort(junk, junk_end, TC{});

17.     for (const auto &v : junk) std::cout << v << " ";
18.     std::cout << std::endl;
19. }
```

Примеры применения (STL)

Размотка цикла

```
1.  std::for_each(std::begin(eList), std::end(eList), [](const Employee& e) {
2.      std::cout << e.firstName << " "
3.                << e.secondName << " "
4.                << "has salary: " << e.salary << std::endl;
5.  });


6.  size_t totalCosts = std::accumulate(std::begin(eList), std::end(eList), 0,
7.                                      [](size_t s, const Employee& e) {
8.        return s + e.salary;
9.  });


10. std::list<Employee> HonorList;

11. std::transform(std::begin(eList), std::end(eList),
12.                std::back_inserter(HonorList),
13.                [&](const Employee& e) -> Employee {
14.     Employee honored(e);
15.     honored.FirstName = std::string("Mrs. ") + honored.FirstName;
16.     return honored;
17. });
```

Примеры применения

Active Object pattern

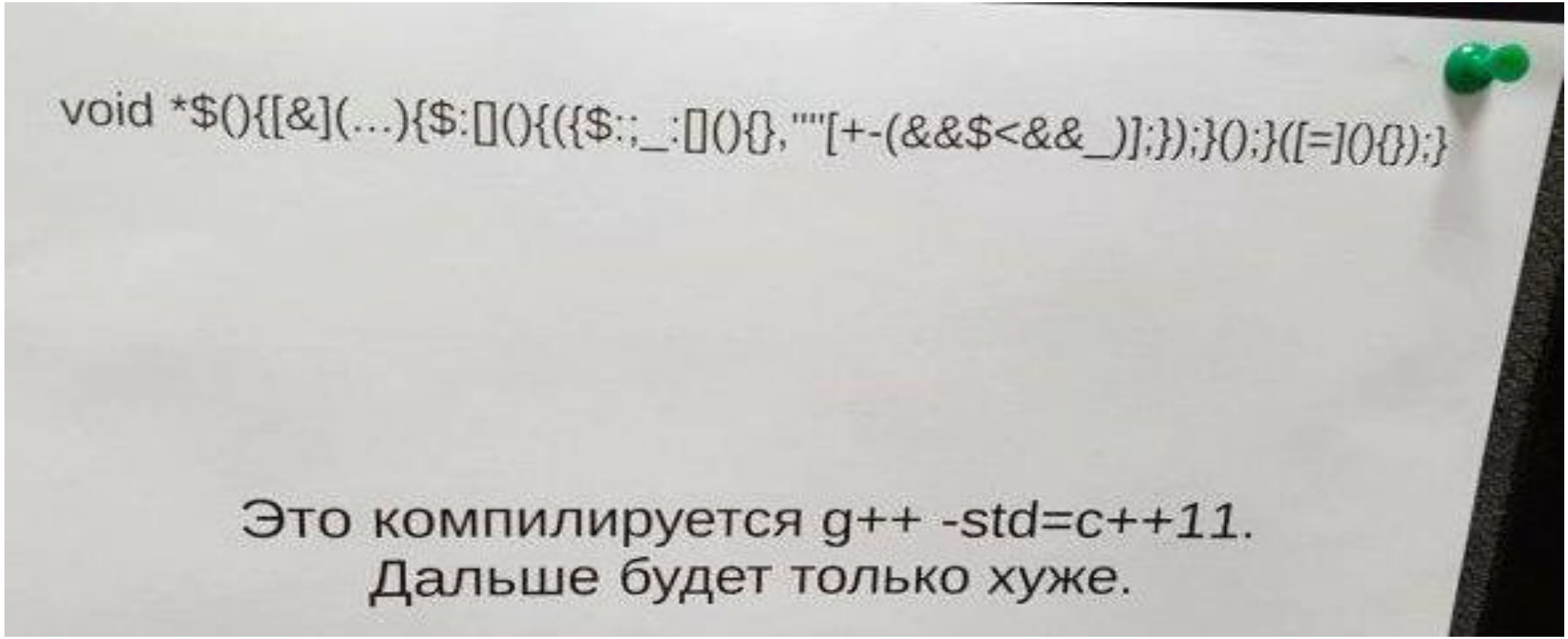
```
1.  struct ActiveObject {
2.      using Message_t = std::function<void()>;
3.      ActiveObject() :
4.          is_done(false),
5.          thread(new std::thread([&]() { run(); }))
6.      {}
7.      ~ActiveObject() {
8.          sendMessage([&]() { is_done = true; });
9.          thread->join();
10.     }
11.     void sendMessage(const Message_t &Msg) ) {
12.         std::lock_guard<std::mutex> lock(mutex);
13.         messages.push(Msg);
14.     }
15. private:
16.     std::atomic<bool> is_done;
17.     std::unique_ptr<std::thread> thread;
18.     std::queue<Message_t> messages;
19.     std::mutex mutex;
20.     void run();
21. };
22. // see next -->
```

```
22. // see prev <--

23. void ActiveObject::run() {
24.     Message_t msg;
25.     bool received = false;
26.     while (!is_done) {
27.         if (!messages.empty()) {
28.             std::lock_guard<std::mutex> lock(mutex);
29.             msg = messages.front();
30.             messages.pop();
31.             received = true;
32.         }
33.         if (received) { msg(); received = false; }
34.     }
35. }

36. int main() {
37.     std::cout << "Main start" << std::endl;
38.     {
39.         ActiveObject Object;
40.         for (auto i = 0u; i < 30u; ++i)
41.             Object.sendMessage([i]() {
42.                 std::this_thread::sleep_for(std::chrono::seconds(1));
43.                 std::cout << "Lambda finished " << i << std::endl;
44.             });
45.         std::this_thread::sleep_for(std::chrono::seconds(50));
46.     }
47.     std::cout << "Main finished" << std::endl;
48.     return 0;
49. }
```

Упражнение



C++14: обобщённая lambda

Только для определённых типов данных

```
1.  auto func = [=, &b] (const auto& a) -> decltype(a() + b) {
2.       return a() + b;
3.  };
```

C++14: обобщённая lambda

Тип аргумента лямбда-функции известен на этапе компиляции

```
1.  auto print = [](auto value) {
2.      std::cout << value << "\n";
3.  };
```

C++ уже достаточно близок к “идеальной” конструкции

```
1.  auto auto = [](auto auto) {
2.      auto;
3.  };
```

C++14: обобщённая lambda

компилятор также преобразует лямбда-функцию в класс-функтор с шаблонным operator()

```
1.  class GenereicLambdaClass {
2.  public:
3.      template<typename T>
4.      auto operator()(T value) const {
5.          std::cout << value << "\n";
6.      }
7.  };
```

C++14: обобщённая lambda

квалификаторы, смешивание с явными типами

```
1.  auto print = [](auto&& first,
2.                  double second,
3.                  volatile auto third) {
4.      std::cout << first << second << third << "\n";
5.  };

6.  print(1, 2.5, 3);
7.  print("one", 2.5, "three");
```

C++14: обобщённая lambda

квалификаторы, смешивание с явными типами

```
1.  auto print = [](auto&& first,
2.                  double second,
3.                  volatile auto third) {
4.      std::cout << first << second << third << "\n";
5.  };

6.  print(1, 2.5, 3);
7.  print("one", 2.5, "three"); // error
```

C++14: обобщённая lambda

- необходимо передать в лямбда-функцию объект, который не имеет оператора копирования, но имеет оператор перемещения

теперь можно создавать объекты в списке захвата

```
1.  //...
2.  std::promise<void> ppromise;
3.  auto future = ppromise.get_future();
4.  auto asyncAction = [promise = std::move(ppromise)](int value)
5.  {
6.      //...
7.      promise.set_value(value);
8.  };
9.  //...


10. class A {
11. public:
12.     int x;
13. };
14. A a;
15. [x = a.x](){ return x * 17; }
```