Лямбда выражения

ЛЕКЦИЯ №7

Протечка абстракции

Неудобной составляющей работы с коллекциями объектов родительского типа является необходимость приведения родительского типа к типу-наследнику (для выполнения необходимых операций над элементом коллекции). Т.е. мы жертвуем статическим контролем типов.

Протечка абстракции
abstraction_leak_1.cpp

```
for (int i = 0; i < array.size(); i++)
        if (array[i] != nullptr) {
            std::cout << "Square:" << array[i]->Square() << std::endl;
            // abstraction leak
            Sphere *sphere = dynamic_cast<Sphere*> (array[i]);
            if (sphere != nullptr)
                std::cout << "Volume:" <<
                    sphere->Volume() << std::endl;
        }
```

Протечка абстракции
abstraction_leak_2.cpp

```
    for (int i = 0; i < array.size(); i++)
         if (array[i] != nullptr) {
             if(array[i]->Square())
                    std::cout << "Square:" << array[i]->Square() << std::endl;
             if(array[i]->Volume())
                    std::cout << "Volume:" << array[i]->Volume() << std::endl;
         }
```

Протечка абстракции
abstraction_leak_3.cpp

```
tuple<Circle, Circle, Sphere, Sphere>
      t(Circle(1), Circle(2), Sphere(1), Sphere(2));

    std::cout << "Square:" << get<0>(t).Square() << std::endl;
    std::cout << "Volume:" << get<3>(t).Volume() << std::endl;
```

Протечка абстракции

variant.cpp //c++17

```
struct FigureVisitor{
    void operator() ( Circle& value) const{
        std::cout << "square:" << value.Square() << std::endl;
    }
    void operator() ( Sphere& value) const{
        std::cout << "volume:" << value.Volume() << std::endl;
    }
};
```

```
std::variant<Circle, Sphere> array[]={Circle(1),Sphere(1),Circle(2)};


 for(auto a:array) std::visit(FigureVisitor(),a);
```

Лямбда

parameter_function.cpp

```cpp
#include <functional>



template< class R, class... Args > class function<R(Args...)>;
```

Lambda
lambda_1.cpp

Лямбда-выражения в C++ — это краткая форма записи анонимных функторов.

Например:

```
[](int _n)   {    cout << _n << " ";}
```

Соответствует:

```
class MyLambda
{
   public: void operator ()(int _x) const { cout << _x << " "; }
};
```

Лямбда функции могут возвращать значения

lambda_2.cpp

В случае, если в лямбда-функции только один оператор return то тип значения можно не указывать. Если несколько, то нужно явно указать.

```
[] (int i) -> double
    {
        if (i < 5)
            return i + 1.0;
        else if (i % 2 == 0)
            return i / 2.0;
        else
            return i * i;
    }
```

Lambda == Functor

```
[ captures ]   ( params ) -> ret   { statements; }
```

```
class functor {
private:
  CaptureTypes __captures;
public:
  __functor( CaptureTypes captures )

  auto operator() ( params ) -> ret
    { statements; }
```

Захват переменных из внешнего контекста

lambda_3.cpp

```
[]                      // без захвата переменных из внешней области видимости
[=]                     // все переменные захватываются по значению
[&]                     // все переменные захватываются по ссылке
[this]                  // захват текущего класса
[x, y]                  // захват x и y по значению
[&x, &y]                // захват x и y по ссылке
[in, &out]              // захват in по значению, а 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…
```

Генерация лямбда-выражений

lambda_4.cpp

Начиная со стандарта C++11 шаблонный класс std::function является полиморфной оберткой функций для общего использования. Объекты класса std::function могут хранить, копировать и вызывать произвольные вызываемые объекты - функции, лямбда-выражения, выражения связывания и другие функциональный объекты. Говоря в общем, в любом месте, где необходимо использовать указатель на функцию для её отложенного вызова, или для создания функции обратного вызова, вместо него может быть использован std::function, который предоставляет пользователю большую гибкость в реализации.

Определение класса

```
template<class> class function; // undefined

template<class R, class... ArgTypes> class
function<R(ArgTypes...)>;
```

Функция, создающая функцию

Caller	Closure	C++11

	• a,b lives inside c	void caller()
		{
		int a = 42;
		int b = 15;
		auto c = [=]() mutable{a = 33;++b; int z = a + b; return z; };
		c();
		c();
		c();
		}

▪ a=42		
▪ b=15		
▪ a=42 b=15	• a=33 b=16 z=49	
▪ a=42 b=15	• a=33 b=17 z=50	
▪ a=42 b=15	• a=33 b=18 z=51	

Лямбда в C++14

C++11

```
auto lambda = [](int x, int y)
   {return x + y;}
struct unnamed_lambda {
  auto operator()(int x, int y)
   const {return x + y;}
};
```

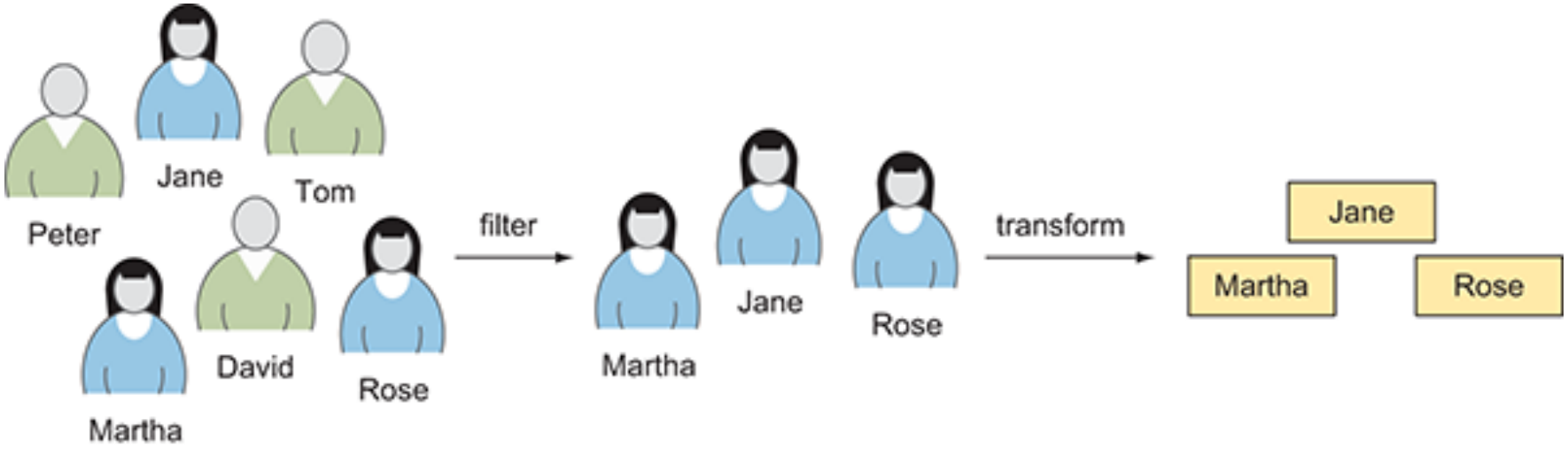
C++14

```
auto lambda = [](auto x,
   auto y) {return x + y;}
struct unnamed_lambda {
   template<typename T,
   typename U> auto
   operator()(T x, U y) const
   {return x + y;}
};
```

Лямбды + variadic template
lambda_5.cpp

```
template <typename T, typename ...Ts>
auto concat(T t, Ts ...ts)
{
if constexpr (sizeof...(ts) > 0)
    return [=](auto ...parameters)
        {
         return t(concat(ts...)(parameters...));
        };
 } else {return t;}
}
```

functional_1.cpp



for_each.cpp
#include <algorithm>

```
std::for_each(
             begin(vehicles),
             end(vehicles),
             [](const Vehicle &vehicle){
                 cout << vehicle.name << endl;
            });
```

transform.cpp
#include <algorithm>

```
      std::transform(
              std::begin(vect),
              std::end(vect),
              std::begin(vect2),
              [](int n) {
                return n * n;
            });
```

Спасибо!

ВСЕ ИДЕМ НА ПЕРЕРЫВ

Подсчет вхождений числа 42 в векторе

```
// C++03

struct functor {

int &a;
functor(int& _a) : a(_a) { } bool operator()(int x)
const {

return a == x; }

};

int a = 42;
count if(v.begin(),v.end(),functor(a));
```

```
// C++11

int a = 42; count_if(v.begin(), v.end(),

[&a](int x){ return x == a;});
```