C++0x (C++11). Лямбда-выражения tutorial

C++*

Буквально на днях случайно наткнулся на Хабре на статью о лямбда-выражениях из нового (будущего) стандарта C++. Статья хорошая и даёт понять преимущества лямбда-выражений, однако, мне показалось, что статья недостаточно полная, поэтому я решил попробовать более детально изложить материал.

Вспомним основы

Лямбда-выражения — одна из фич функциональных языков, которую в последнее время начали добавлять также в императивные языки типа C#, C++ etc. Лямбда-выражениями называются безымянные локальные функции, которые можно создавать прямо внутри какого-либо выражения.

В прошлой статье лямбда-выражения сравнивали с указателями на функции и с функторами. Так вот первое, что следует уяснить: **лямбда-выражения в C++ — это краткая форма записи анонимных функторов**. Рассмотрим пример:

```
// Листинг 1
#include <algorithm>
#include <cstdlib>
#include <iostream>
#include <vector>

using namespace std;

int main()
{
	vector<int> srcVec;
	for (int val = 0; val < 10; val++)
	{
		srcVec.push_back(val);
	}

	for_each(srcVec.begin(), srcVec.end(), [](int _n)
	{
		cout << _n << " ";
	});
	cout << endl;

	return EXIT_SUCCESS;
}
```

Фактически данный код целиком соответствует такому:

```
// Листинг 2
#include <algorithm>
#include <cstdlib>
#include <iostream>
#include <vector>

using namespace std;

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

int main()
{
        vector<int> srcVec;
        for (int val = 0; val < 10; val++)
        {
                srcVec.push_back(val);
        }

        for_each(srcVec.begin(), srcVec.end(), MyLambda());
        cout << endl;

        return EXIT_SUCCESS;
}
```

Вывод соответственно будет следующим:

```
0 1 2 3 4 5 6 7 8 9
```

На что здесь стоит обратить внимание. Во-первых, из *Листинга 1* мы видим, что лямбда-выражение всегда начинается с **[]** (скобки могут быть непустыми — об этом позже), затем идет необязательный список параметров, а затем непосредственно тело функции. Во-вторых, тип возвращаемого значения мы не указывали, и по умолчанию лямбда возвращает **void** (далее мы увидим, как и зачем можно указать возвращаемый тип явно). В-третьих, как видно по *Листингу 2*, по умолчанию генерируется константный метод (к этому тоже еще вернемся).

Не знаю, как вам, но мне **for_each**, записанный с помощью лямбда-выражения, нравится

гораздо больше. Попробуем написать немного усложненный пример:

```
// Листинг 3
#include <algorithm>
#include <cstdlib>
#include <iostream>
#include <vector>

using namespace std;

int main()
{
	vector<int> srcVec;
	for (int val = 0; val < 10; val++)
	{
		srcVec.push_back(val);
	}

	int result =
		count_if(srcVec.begin(), srcVec.end(), [] (int _n)
		{
			return (_n % 2) == 0;
		});

	cout << result << endl;

	return EXIT_SUCCESS;
}
```

В данном случае лямбда играет роль *унарного предиката*, то есть тип возвращаемого значения **bool**, хотя мы нигде этого не указывали. При наличии одного **return** в лямбда-выражении, компилятор вычисляет тип возвращаемого значения самостоятельно. Если же в лямбда-выражении присутствует **if** или **switch** (или другие сложные конструкции), как в приведенном ниже коде, то на компилятор полагаться уже нельзя:

```
// Листинг 4
#include <algorithm>
#include <cstdlib>
#include <iostream>
#include <iterator>
#include <vector>

using namespace std;
```

```
int main()
{
	vector<int> srcVec;
	for (int val = 0; val < 10; val++)
	{
		srcVec.push_back(val);
	}

	vector<double> destVec;
	transform(srcVec.begin(), srcVec.end(),
		  back_inserter(destVec), [] (int _n)
	{
		if (_n < 5)
			return _n + 1.0;
		else if (_n % 2 == 0)
			return _n / 2.0;
		else
			return _n * _n;
	});

	ostream_iterator<double> outIt(cout, " ");
	copy(destVec.begin(), destVec.end(), outIt);
	cout << endl;

	return EXIT_SUCCESS;
}
```

Код из *Листинга 4* не компилируется, а, к примеру, Visual Studio пишет ошибку на каждый **return** такого содержания:

```
«error C3499: a lambda that has been specified to have a void return type cannot return a v
```

Компилятор не может самостоятельно вычислить тип возвращаемого значения, поэтому мы должны его указать явно:

```
// Листинг 5
#include <algorithm>
#include <cstdlib>
#include <iostream>
#include <iterator>
#include <vector>

using namespace std;
```

```
int main()
{
	vector<int> srcVec;
	for (int val = 0; val < 10; val++)
	{
		srcVec.push_back(val);
	}

	vector<double> destVec;
	transform(srcVec.begin(), srcVec.end(),
		  back_inserter(destVec), [] (int _n) -> double
	{
		if (_n < 5)
			return _n + 1.0;
		else if (_n % 2 == 0)
			return _n / 2.0;
		else
			return _n * _n;
	});

	ostream_iterator<double> outIt(cout, " ");
	copy(destVec.begin(), destVec.end(), outIt);
	cout << endl;

	return EXIT_SUCCESS;
}
```

Теперь компиляция проходит успешно, а вывод, как и ожидалось, будет следующим:

> 1 2 3 4 5 25 3 49 4 81

Единственное, что мы добавили в *Листинге 5*, это тип возвращаемого значения для лямбда-выражения в виде **-> double**. Синтаксис немного странноват и смахивает больше на Haskell, чем на C++. Но указывать возвращаемый тип «слева» (как в функциях) не получилось бы, потому что лямбда должна начинаться с **[]**, чтобы компилятор смог её различить.

Захват переменных из внешнего контекста

Все лямбда-выражения, приведенные выше, выглядели как анонимные функции, потому что не хранили никакого промежуточного состояния. Но лямбда-выражения в C++ — это анонимные функторы, а значит состояние они хранить могут! Используя лямбда-выражения, напишем программу, которая выводит количество чисел, попадающих в заданный

пользователем интервал [lower; upper):

```
// Листинг 6
#include <algorithm>
#include <cstdlib>
#include <iostream>
#include <iterator>
#include <numeric>
#include <vector>

using namespace std;

int main()
{
	vector<int> srcVec;
	for (int val = 0; val < 10; val++)
	{
		srcVec.push_back(val);
	}

	int lowerBound = 0, upperBound = 0;
	cout << "Enter the value range: ";
	cin >> lowerBound >> upperBound;

	int result =
		count_if(srcVec.begin(), srcVec.end(),
			[lowerBound, upperBound] (int _n)
		{
			return lowerBound <= _n && _n < upperBound;
		});
	cout << result << endl;

	return EXIT_SUCCESS;
}
```

Наконец, мы добрались до того момента, когда лямбда-выражение начинается не с пустых скобок. Как видно в *Листинге 6*, внутри квадратных скобок могут указываться переменные. Это называется… эээм… *«список захвата»* (capture list). Для чего это нужно? На первый взгляд может показаться, что внешней областью видимости для лямбда-выражения является функция **main()** и мы можем беспрепятственно использовать переменные, объявленные в ней, внутри тела лямбда-выражения, однако это не так. Почему? Потому что фактически тело лямбды — это тело перегруженного **operator()()** (как бы это назвать… оператора функционального вызова что ли) внутри анонимного функтора, то есть для кода из *Листинга*

6 компилятор неявно сгенерирует примерно такой код:

```
// Листинг 7
#include <algorithm>
#include <cstdlib>
#include <iostream>
#include <iterator>
#include <vector>

using namespace std;

class MyLambda
{
public:
        MyLambda(int _lowerBound, int _upperBound)
                : m_lowerBound(_lowerBound)
                , m_upperBound(_upperBound)
        {}

        bool operator ()(int _n) const
        {
                return m_lowerBound <= _n && _n < m_upperBound;
        }

private:
        int m_lowerBound, m_upperBound;
};

int main()
{
        vector<int> srcVec;
        for (int val = 0; val < 10; val++)
        {
                srcVec.push_back(val);
        }

        int lowerBound = 0, upperBound = 0;
        cout << "Enter the value range: ";
        cin >> lowerBound >> upperBound;

        int result = count_if(srcVec.begin(),
                              srcVec.end(),
                              MyLambda(lowerBound, upperBound));
        cout << result << endl;
```

```
        return EXIT_SUCCESS;
}
```

Листинг 7 немного всё разъясняет. Наша лямбда превратилась в функтор, внутри тела которого мы не можем напрямую использовать переменные, объявленные в **main()**, так как это непересекающиеся области видимости. Для того чтобы доступ к **lowerBound** и **upperBound** все-таки был, эти переменные сохраняются внутри самого функтора (происходит тот самый «захват»): конструктор их инициализирует, а внутри **operator()()** они используются. Я специально дал этим переменным имена, начинающиеся с префикса «**m_**», чтобы подчеркнуть различие.

Если мы попытаемся изменить «захваченные» переменные внутри лямбды, нас ждет неудача, потому что по умолчанию генерируемый **operator()()** объявлен как **const**. Для того чтобы это обойти, мы можем указать спецификатор **mutable**, как в следующем примере:

```
// Листинг 8
#include <algorithm>
#include <cstdlib>
#include <iostream>
#include <iterator>
#include <numeric>
#include <vector>

using namespace std;

int main()
{
        vector<int> srcVec;
        int init = 0;
        generate_n(back_inserter(srcVec), 10, [init] () mutable
        {
                return init++;
        });

        ostream_iterator<int> outIt(cout, " ");
        copy(srcVec.begin(), srcVec.end(), outIt);
        cout << endl << "init: " << init << endl;

        return EXIT_SUCCESS;
}
```

Ранее я упоминал, что список параметров лямбды можно опускать, когда он пустой, однако для того чтобы компилятор правильно распарсил применение слова **mutable**, мы должны явно

указать пустой список параметров.
При выполнении программы из *Листинга 8* получаем следующее:

```
0 1 2 3 4 5 6 7 8 9
init: 0
```

Как видим, благодаря ключевому слову **mutable**, мы можем менять значение «захваченной» переменной внутри тела лямбда-выражения, но, как и следовало ожидать, эти изменения не отражаются на локальной переменной, так как захват происходит по значению. C++ позволяет нам захватывать переменные по ссылке и даже указывать *«режим захвата», используемый по умолчанию*. Что это означает? Мы можем не указывать каждую переменную в списке захвата по отдельности: вместо этого можно просто указать режим по умолчанию для захвата, и тогда все переменные из внешнего контекста, которые используются внутри лямбды, будут захвачены компилятором автоматически. Для указания режима захвата по умолчанию существует специальный синтаксис: **[=]** или **[&]** для захвата по значению и по ссылке соответственно. При этом для каждой переменной можно указать свой режим захвата, однако режим по умолчанию, естественно, указывается только единожды, причем в самом начале списка захвата. Вот варианты использования:

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

Следует отметить, что синтаксис наподобие **&out** в данном случае не означает взятие адреса. Его следует читать скорее как **SomeType & out**, то есть это просто передача параметра по ссылке. Рассмотрим пример:

```
// Листинг 9
#include <algorithm>
#include <cstdlib>
#include <iostream>
#include <iterator>
#include <vector>

using namespace std;

int main()
```

```
{
        vector<int> srcVec;
        int init = 0;
        generate_n(back_inserter(srcVec), 10, [&] () mutable
        {
                return init++;
        });

        ostream_iterator<int> outIt(cout, " ");
        copy(srcVec.begin(), srcVec.end(), outIt);
        cout << endl << "init: " << init << endl;

        return EXIT_SUCCESS;
}
```

В этот раз вместо явного захвата переменной **init**, я указал режим захвата по умолчанию: **[&]**. Теперь когда компилятор встречает внутри тела лямбды переменную из внешнего контекста, он автоматически захватывает её по ссылке. Вот эквивалентный *Листингу 9* код:

```
// Листинг 10
#include <algorithm>
#include <cstdlib>
#include <iostream>
#include <iterator>
#include <vector>

using namespace std;

class MyLambda
{
public:
        explicit MyLambda(int & _init) : init(_init) { }

        int operator ()() { return init++; }

private:
        int & init;
};

int main()
{
        vector<int> srcVec;
        int init = 0;
        generate_n(back_inserter(srcVec), 10, MyLambda(init));
```

```
        ostream_iterator<int> outIt(cout, " ");
        copy(srcVec.begin(), srcVec.end(), outIt);
        cout << endl << "init: " << init << endl;

        return EXIT_SUCCESS;
}
```

И соответственно вывод будет следующим:

> 0 1 2 3 4 5 6 7 8 9
> init: 10

Теперь вам главное не запутаться, что, где и когда передавать по ссылке. Фактически, если мы указываем **[&]** и не указываем **mutable**, то все равно сможем менять значение захваченной переменной и это отразится на локальной, потому что **operator()() const** подразумевает, что мы не можем менять, на что указывает ссылка, а это и так невозможно.

Если лямбда-выражение имеет вид **[=] (int & _val) mutable { ... }**, то переменные захватываются по значению, но меняться будет только их внутренняя копия, а вот параметр передается по ссылке, то бишь изменения отразятся и на оригинале. Если **[] (const SomeBigObject & _val) { ... }**, то ничего не захватывается, а параметр принимается по константной ссылке и т.д.

Я так понял, что выполнить захват «по константной ссылке» невозможно. Ну, может, оно нам и не надо.

А что будет, если мы напишем такое, слегка надуманное лямбда-выражение внутри метода класса:

```
// Листинг 11
#include <algorithm>
#include <cstdlib>
#include <iostream>
#include <iterator>
#include <vector>

using namespace std;

class MyMegaInitializer
{
public:
        MyMegaInitializer(int _base, int _power)
```

```
                : m_val(_base)
                , m_power(_power)
        {}

        void initializeVector(vector<int> & _vec)
        {
                for_each(_vec.begin(), _vec.end(),
                         [m_val, m_power] (int & _val) mutable
                {
                        _val = m_val;
                        m_val *= m_power;
                });
        }

private:
        int m_val, m_power;
};

int main()
{
        vector<int> myVec(11);
        MyMegaInitializer initializer(1, 2);
        initializer.initializeVector(myVec);

        return EXIT_SUCCESS;
}
```

Несмотря на все наши ожидания, код не будет скомпилирован, так как компилятор не сможет захватить **m_val** и **m_power**: эти переменные вне области видимости. Вот что говорит на это Visual Studio:

```
«error C3480: 'MyMegaInitializer::m_power': a lambda capture variable must be from an enclo
```

Как же быть? Чтобы получить доступ к членам класса, в capture-list нужно поместить **this**:

```
// Листинг 12
#include <algorithm>
#include <cstdlib>
#include <iostream>
#include <iterator>
#include <vector>

using namespace std;

class MyMegaInitializer
```

```
{
public:
        MyMegaInitializer(int _base, int _power)
                : m_val(_base)
                , m_power(_power)
        {}

        void initializeVector(vector<int> & _vec)
        {
                for_each(_vec.begin(), _vec.end(), [this] (int & _val) mutable
                {
                        _val = m_val;
                        m_val *= m_power;
                });
        }

private:
        int m_val, m_power;
};

int main()
{
        vector<int> myVec(11);
        MyMegaInitializer initializer(1, 2);
        initializer.initializeVector(myVec);

        for_each(myVec.begin(), myVec.end(), [] (int _val)
        {
                cout << _val << " ";
        });
        cout << endl;

        return EXIT_SUCCESS;
}
```

Данная программа делает именно то, чего мы ожидали:

> 1 2 4 8 16 32 64 128 256 512 1024

Следует заметить, что **this** можно захватить только по значению, и если вы попытаетесь произвести захват по ссылке, компилятор выдаст ошибку. Даже если вы в коде из *Листинга 12* напишете **[&]** вместо **[this]**, то **this** будет все равно захвачен по значению.

Прочее

Помимо всего вышеперечисленного, в заголовке лямбда-выражения можно указать throw-list — список исключений, которые лямбда может сгенерировать. Например, такая лямбда не может генерировать исключения:

```
[] (int _n) throw() { … }
```

А такая генерирует только **bad_alloc**:

```
[=] (const std::string & _str) mutable throw(std::bad_alloc) -> bool { … }
```

И т.п.

Естественно, если его не указывать, то лямбда может генерировать любое исключение.

К счастью, в финальном варианте стандарта throw-спецификации объявлены устаревшими. Вместо этого оставили ключевое слово **noexcept**, которое говорит, что функция не должна генерировать исключение вообще.

Таким образом, общий вид лямбда-выражения следующий (сорри за такой «вольный вид» грамматики):

```
lambda-expression ::=
              ‘[’  [<список_захвата>]  ‘]’
            [ ‘(’  <список_параметров> ‘)’  [‘mutable’ ] ]
            [ ‘noexcept’ ]
            [ ‘->’  <тип_возвращаемого_значения> ]
              ‘{’  [<тело_лямбды>]  ‘}’
```

Повторное использование лямбда-выражений. Генерация лямбда-выражений.

Все вышеперечисленное довольно удобно, но основная мощь лямбда-выражений приходится на то, что мы можем сохранить лямбду в переменной или передавать как параметр в функцию. В Boost для этого есть класс Function, который, если я не ошибаюсь, войдет в новый стандарт STL (возможно, в немного измененном виде). На данный момент уже можно поюзать фичи из обновленного STL, однако, пока что эти фичи находятся в подпространстве имен **std::tr1**.

Возможность сохранения лямбда-выражений позволяет нам не только повторно использовать лямбды, но и писать функции, которые генерируют лямбда-выражения, и даже лямбды, которые генерируют лямбды.

Рассмотрим следующий пример:

```
// Листинг 13
#include <algorithm>
#include <cstdlib>
#include <functional>
#include <iostream>
#include <iterator>
#include <vector>

using namespace std;
using std::tr1::function;

int main()
{
	vector<int> myVec;
	int init = 0;
	generate_n(back_inserter(myVec), 10, [&]
	{
		return init++;
	});

	function<void (int)> traceLambda = [] (int _val) -> void
					{
						cout << _val << " ";
					};

	for_each(myVec.begin(), myVec.end(), traceLambda);
	cout << endl;

	function<function<int (int)> (int)> lambdaGen =
		[] (int _val) -> function<int (int)>
		{
			return [_val] (int _n) -> int { return _n + _val; };
		};

	transform(myVec.begin(), myVec.end(), myVec.begin(), lambdaGen(2));
	for_each(myVec.begin(), myVec.end(), traceLambda);
	cout << endl;

	return EXIT_SUCCESS;
}
```

Данная программа выводит:

```
0 1 2 3 4 5 6 7 8 9
2 3 4 5 6 7 8 9 10 11
```

Рассмотрим подробнее. Вначале у нас инициализируется вектор с помощью **generate_n()**. Тут всё просто. Далее мы создаем переменную **traceLambda** типа **function<void (int)>** (то есть функция, принимающая **int** и возвращающая **void**) и присваиваем ей лямбда-выражение, которое выводит на консоль значение и пробел. Далее мы используем только что сохраненную лямбду для вывода всех элементов вектора.

После этого мы видим немаленькое объявление **lambdaGen**, которая является лямбда-выражением, принимающим один параметр **int** и возвращающим другую лямбду, принимающую **int** и возвращающую **int**.

Следом за этим мы ко всем элементам вектора применяем **transform()**, в качестве мутационной функции для которого указываем **lambdaGen(2)**.
Фактически **lambdaGen(2)** возвращает другую лямбду, которая прибавляет к переданному параметру число 2 и возвращает результат. Этот код, естественно, немного надуманный, ибо то же самое можно было записать как

```
transform(myVec.begin(), myVec.end(), myVec.begin(), bind2nd(plus<int>(), 2));
```

однако в качестве примера довольно показательно.

Затем мы снова выводим значения всех элементов вектора, используя для этого сохраненную ранее лямбду **traceLambda**.

На самом деле, данный код можно было записать еще короче. В новом стандарте C++ значение ключевого слова **auto** будет заменено. Если раньше **auto** означало, что переменная создается в стеке, и подразумевалось неявно в случае, если вы не указали что-либо другое (**register**, к примеру), то сейчас это такой себе аналог **var** в C# (то есть тип переменной, объявленной как **auto**, определяется компилятором самостоятельно на основе того, чем эта переменная инициализируется).
Следует заметить, что **auto**-переменная не сможет хранить значения разных типов в течение одного запуска программы. C++ как был, так и остается статически типизированным языком, и указание **auto** лишь говорит компилятору самостоятельно позаботиться об определении типа: после инициализации сменить тип переменной будет уже нельзя.

Кроме того что ключевое слово **auto** весьма полезно при работе с циклами вида

```
for (auto it = vec.begin(); it != vec.end(); ++it)
{
        // ...
}
```

его очень удобно использовать с лямбда-выражениями. Теперь код из *Листинга 13* можно переписать так:

```
// Листинг 14
#include <algorithm>
#include <cstdlib>
#include <functional>
#include <iostream>
#include <iterator>
#include <vector>

using namespace std;
using std::tr1::function;

int main()
{
	vector<int> myVec;
	int init = 0;
	generate_n(back_inserter(myVec), 10, [&]
	{
		return init++;
	});

	auto traceLambda = [] (int _val) -> void { cout << _val << " "; };

	for_each(myVec.begin(), myVec.end(), traceLambda);
	cout << endl;

	auto lambdaGen = [] (int _val) -> function<int (int)>
	{
		return [_val] (int _n) -> int { return _n + _val; };
	};

	transform(myVec.begin(), myVec.end(), myVec.begin(), lambdaGen(2));
	for_each(myVec.begin(), myVec.end(), traceLambda);
	cout << endl;

	return EXIT_SUCCESS;
}
```

Пожалуй, на этом я закончу описание лямбда-выражений. Если будут вопросы, поправки или замечания, с удовольствием выслушаю.

PROFIT!

ETA (20.02.2012): Оказалось, что для кого-то эта статья до сих пор актуальна, поэтому поправил подсветку синтаксиса и подкорректировал информацию про throw-списки в объявлении лямбд. Помимо непосредственно лямбда-выражений другие фичи из нового стандарта C++11 (например, списки инициализации контейнеров) решил не добавлять, так что статья осталась практически в первозданном виде.

lambda, lambda functions, lambda expressions, c++0x, c++, c++11