System Software Crash Couse

Samsung Research Russia
Moscow 2019

Block G: Advanced C++
9. Lambdas & Functional Programming

Eugene Zouev

Functional programming & lambda expressions in C++

Functional Programming

Two cornerstones of functional approach:

- Immutable objects
- Functions as "first-class citizens"

FP 1: Immutable objects

Data are immutable:
The operations of a program should **map** input values to output values rather than **change** data in place.

Or: Methods should not have any side effects. They should communicate with their environment only by taking arguments and returning results… For any given input the method call could be replaced by its result without affecting the program's semantics.

Advantages:

- Programs are easy-for-testing.
- Programs are verifiable.
- Programs can be parallelized.

FP 2: Functions as objects

Functions are **values** - just as integers, arrays etc.

- Function is an abstraction of an **operation**.
- Define functions anywhere (just as other variables).
- "Constant" (or unnamed, or anonymous) functions are allowed (like integer constants).
- Functions can be assigned, passed as arguments to other functions etc.

Functional is the Trend!

Almost every modern programming language has at least some “functional” features.

- **C++**: lambda expressions, function types, functions without side effects, type inference
- **C#**: function types (“delegates”), function literals, type inference
- **Java 8**: lambda expressions
- **Swift**: functions as values; closures; local functions
- **Rust**: functions as variables, as arguments, as return values; anonymous functions

Functional programming (1)

The common philosophy in functional programming:

> The program functionality is implemented as a (great) number of functions (“building blocks”), each of which performs relatively simple action on its arguments returning the result (without side effect).

There are three basic ways for programming functions (C++)... and one more:

- Define a function as a class member.
- Define function as an object member (~static)
- Define a standalone function.
- Define a function as an object!

Lambda expressions in C++

What do we really need to know about a function?

- Parameters
- Return type
- Algorithm (body)

Return type: deduced from the operand of return (the simplest case though)

```
(int x)   { return x+1; }
```

Parameters

Algorithm

Lambda expression

- Unnamed function
- Anonymous function
- Function literal

```
[](int x) { return x+1; }
```

Since C++11

Lambda expressions in C++

```
5
3.14
“abcd”
```

- Unnamed objects
- Literals

- Unnamed object: a function
- Function literal

```
[](int x) { return x+1; }
```

```
double x = 3.14;
auto f = [](int x) { return x+1; }

auto f1 = f; // Assigning lambdas
int y = f(5); // Invoking lambdas
```

Lambda expressions in C++

```
[](int x) { return x+1; }
```

Points to make clear:

- What does the construct `[]` mean exactly?
- How to specify return type explicitly?
- Technical details (a lot of ☹)
- What is the type of the lambda as a whole?
- What's new with lambdas?

Lambdas: what did we have before? – functional objects!

Expression list

Function call:

```
F ( <arguments> )
```

Function

Pointer to function

Examples:

```
int F ( int x) { return expr; }
int (*pF)(int) = F;
 . . .
int a = F(1);
int b = pF(1);
```

Functional Objects: a Side-step (2)

Function call:

$$\begin{tikzcd}
 & \text{Expression list} \arrow[d] \\
\textbf{F} & \textbf{(} \; \textit{<arguments>} \; \textbf{)} \\
\text{Function} \arrow[u] & \\
\text{Pointer to function} \arrow[uu] & \\
\textit{Object of a type with the call operator} \arrow[uuu] &
\end{tikzcd}$$

Example:

```
class C {
public:
   int operator()(int x) { return expr; }
};
  . . .
C c;
int z = c(1); // ≡ c.operator()(1);
```

Functional Objects: a Side-step (3)

If F is an object of a type with the call operator then the construct like

```
F ( <arguments> )
```

is equivalent to:

```
F.operator() ( <arguments> )
```

Just a special "name"

- If a type has the call operator **operator()** then the type is called **functional type**.
- If an object is of a functional type it is called **functional object**.

Built-in or user-defined

Example: "pointer-to-function" type

Functional Objects & Lambdas

Expression list

Function call: `F ( <arguments> )`

- Function
- Pointer to function
- *Object of a type with the call operator*
- ***Lambda object!!!***

```
auto f = [](int x) { return x+1; }

f(5)
[](int x){ return x-1; }(7)
```

← Weird but legal call

Lambdas: return type

```
[](int x) { return x+1; }
```

If the lambda body contains **the single return statement** then (and only then) the return type can be deduced by the compiler

```
[](int& x) { ++x; }
```

Otherwise, the return value is assumed to be **void**

```
[](int x)->int { cout<<"Hi!"; return x+1; }
```

If the lambda body contains more than one statement then the return type should be specified explicitly

```
[] { return sizeof(int); }
```

If no parameters then () can be omitted

Lambdas & closures

```
[](int x) { return x+1; }
```

The lambda **doesn't** depend on the context: **closed term**

```
[](int x) { return x + more; }
```

The lambda **does** depend on the context: **open term** or **closure**

An important issue: what happens if `more` variable **changes its value** after the closure has created?

```
 int more = 1;
...
auto addMore = [](int x){ return x+more; }
addMore(10)    // returns 11
...
more = 9999;
addMore(10)    // returns...WHAT?
```

Lambdas & closures

```
 int more = 1;
...
auto addMore = [](int x){ return x+more; }
addMore(10)    // returns 11
...
more = 9999;
addMore(10)    // returns...WHAT?
```

More concrete question:
- What exactly does the lambda capture: a **value** of an entity captured or the **reference** to the entity?

Scala's answer: the reference only

C++'s answer: either value or reference ☺

Lambdas & closures

Alternative notation: [=more]

```
int more = 1;
...
auto addMore1 = [more](int x){ return x+more; }
addMore(10)    // returns 11
...
more = 9999;
addMore(10)    // returns 11
```

This is **not** address-of operator;
just to specify the access by reference

```
int more = 1;
...
auto addMore1 = [&more](int x){ return x+more; }
addMore(10)    // returns 11
...
more = 9999;
addMore(10)    // returns 10009
```

Lambdas & closures

Notation for **lambda capture**:

`[a, &b]`	`a` gets captured by value, `b` gets captured by reference
`[&]`	the whole context gets captured by reference
`[=]`	the whole context gets captured by value
`[=, &a]`	the whole context gets captured by value except `a` that is captured by reference
`[this]`	**`this`** pointer gets captured by value
`[]`	nothing is captured from the context

Lambda's type

```
auto f = [](int x) { return x+1; }
```

The type of `f` is deduced as

```
Function<int(int)>
```

Where `Function` is the template from the Standard library.

The task for your homework:
Consider the definition of the `Function` template for better understanding.

Lambda's internal representation

```
  [](int x) { return x+1; }
```

```
class InternalName {
  public:
    int operator()(int x)
    { return x+1; }
};
```

```
[](int x) { return x+1; }(7)
```

```
InternalName(7)
```

The task for your homework:
Consider the lambda internal representation for the case of **non-empty** lambda capture.

Lambda's example

The task for your homework: Write a few reasonable examples of using lambdas. Use **capture lists** for specifying various ways of capturing contexts.

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

vector<int> myVec;

for (int val = 0; val < 10; val++)
{ myVec.push_back(val); }

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

Lambda expressions: references

- C++ ISO Standard, Sect. 8.5.1.
- https://stackoverflow.com/questions/7627098/what-is-a-lambda-expression-in-c11
- https://www.cprogramming.com/c++11/c++11-lambda-closures.html
- https://habrahabr.ru/post/66021/ (Russian)
- http://ru.cppreference.com/w/cpp/language/lambda (Russian)

[](){}

Do nothing
Accept nothing
Incognito ☺
No name
No meaning
No sense

[](){}()

Can be called only once