YEG  MEETUP

Introduction to C++ Lambdas

Antal Buss

May 12 2020

What is a lambda function?

- Usually referenced as an anonymous function

  (... but it is more than that)

- Lambda comes from the Lambda Calculus (λ-calculus)
- λ-calculus is a formal system in mathematical logic for expressing computation based on function abstraction and application using variable binding and substitution.
- λ-calculus forms the basis of all functional programming languages (e.g., Lisp, ML, Scheme)

Functional Programming

- Functional programming is a programming paradigm where programs are constructed by applying and composing functions.
- In functional programming, functions are *first-class citizens*. They can be bound to names (including local identifiers), passed as arguments, and returned from other functions.
- This allows programs to be written in a declarative and composable style, where small functions are combined in a modular manner.

https://en.wikipedia.org/wiki/Functional_programming

Advantages of functional programming

Some of the advantages of functional programming are:

- Allows you to write more compressed and predictable code
- Testing and debugging is easier

Advantages of functional programming

- Functions are deterministic, meaning that for the same input parameters you will get the same result.
- Pure functions definitions does not have side-effects (you can replace the function call with its final value, without changing the values of the program).
- Function composition means the process of combining two or more functions in order to create a new function or perform calculations.

Function in C++

```
return_type function_name( parameter_list )
{
    function body;
}
```

`return_type` is the type of value that the function will return. It can be int, float or any user-defined data type.

`function_name` means any name that we give to the function. However, it must not resemble any standard keyword of C++.

`parameter_list` contains a total number of arguments that need to be passed to the function.

`function_name``(` *`parameter_list`* `)` defines the function signature (without the return type).

Function in C++

```
int plus(int a, int b)
{
    return a+b;
}
```

```
template< typename T >
T plus(T a, T b)
{
    return a+b;
}
```

```
std::cout << plus(4,9);   → 13

std::cout << plus("hello ", "world!");  → ???
```

code

Pointer to functions

The syntax for creating a non-const function pointer

```
return_type (*function_ptr_name)( type_list );
```

```
double (*plus_i_d)(int, int);
```

plus_i_d is a pointer to a function that has two integer parameters and returns a double.

plus_i_d can point to any function that matches this type.

code1 code2

Passing function to functions

Using pointers to functions we can pass functions as parameters to another functions

```
void my_print(int(*fn)(int), int value)
{
    std::cout << fn(value) << std::endl;
}
```

code

Limitations of pointer to functions

Using p*ointer to functions* is not functional programming.

Functions can be affected by external changes (no side-effects free).

```
int c = 10;

int plus(int a, int b)
{
    return a + b + c++;
}

int(*fn)(int,int) { plus };

std::cout << fn(4,9);    → 23
std::cout << fn(4,9);    → 24
```

code

Functors

- *Functors* (or function objects) are objects that can be treated as though they are a function or function pointer.
- A functor is define as a class that overloads the operator `()`.
- Functors can be "customized" because they can contain state.

```
class add_value {
    int cnte;
public:
    add_value(int v) : cnte(v) {}

    int operator()(int value) {
        return value + cnte;
    }
};
```

code1 code2 code3

C++ lambda functions

Introduced in C++11

https://www.bfilipek.com/2019/02/lambdas-story-part1.html

https://app.getpocket.com/read/2750930137

https://en.cppreference.com/w/cpp/language/lambda

C++ lambda functions

C++ lambdas were introduced in C++11, as a convenient way of defining an anonymous function object (a closure).

Lambdas are sometimes referred to as closures or lambda expressions.

Typically lambdas are used to encapsulate a few lines of code that are passed to algorithms or asynchronous methods.

C++ lambda functions

Syntax

https://docs.microsoft.com/en-us/cpp/cpp/lambda-expressions-in-cpp?view=vs-2019

C++ lambda functions - Syntax

```
[ capture ] ( parameter_list ) optionals { body }
```

capture: (capture clause) specifies which variables are captured, and whether the capture is by value or by reference.

parameter_list: lambda's input parameters (optional).

optionals: includes *return type* of a lambda expression (optional and is automatically deduced), `mutable`, exceptions, ...

C++ lambda functions - Syntax

The simples empty lambda function

```
[]{}    or    [](){}
```

An increment lambda function

```
auto fn1 = [](int x){ return x+1; }

auto fn2 = [](int x) -> double { return x+1; }
```

Using capture clause

```
double inc = 3.1;

auto fn3 = [inc](int x) -> double { return x+inc; }
```

Capture clause (Closure)

Specifies which variables are captured, and whether the capture is by value or by reference.

- variables with a **&** prefix are accessed by reference
- variables without prefix are accessed by value.

Default capture mode

- [**&**] all variables that you refer to are captured by reference.
- [=] all variables that you refer to are captured by value.

You can use a default capture mode, and then specify the opposite mode explicitly for specific variables.

Capture clause (Closure)

```
int a;
double b=1;

[=]{ a = b; };       // ERROR: a capture by value, non-mutable.
[&]{ a = b; };       // OK
[a, b]{ a = b; };    // ERROR: a capture by value, non-mutable.
[&, b]{ a = b; };    // OK
[=, b]{ a = b; };    // ERROR: a capture by value and redundant capture.
[=, &a]{ a = b;};    // Ok
[&, &a]{ a = b;};    // Ok, but redundant capture.
[=]() mutable { a = b; };  // OK
```

code

Lambda functions and Functors

Lambda functions are "*syntax sugar*" to define functors.

The compiler converts a lambda definition into a function object (functor)

```
int inc = 8;
auto inc_fn = [inc](int x) {
  return x+inc;
};
```



```
class __lambda_6_19
{
public:
  inline int operator()(int x) const
  {
    return x + inc;
  }

private:
  int inc;

public:
  __lambda_6_19(int _inc)
  : inc{_inc}
  {}
};
```

code

Evolution of lambdas in C++

C++11

- Initial support for lambdas with some restrictions

C++14

Added two significant enhancements to lambda expressions:

- Captures with an initialiser
- Generic lambdas

Evolution of lambdas in C++

Captures with an initialiser

```
int a = 4;
int b = 7;

auto inc_fn = [inc=a+b](int x) {
  return x+inc;
};
```

code

Evolution of lambdas in C++

Generic lambdas

```
auto println = [](auto x) {
  std::cout << x << std::endl;
};

println(45);
println(23.6);
println("Hello world!");
```

code

Evolution of lambdas in C++

C++17

Added two significant enhancements:

- `constexpr` lambdas
- Capture of `*this`

Evolution of lambdas in C++

`constexpr` lambdas

```
constexpr int value = 9;

constexpr auto inc_fn = [value] (int x) {
  return x + value;
};

if constexpr (inc_fn(2) == 11)
  std::cout << "good";
else
  std::cout << "wrong";
```

code

Evolution of lambdas in C++

Capture of `*this`

```
struct Counter {
  int value;

  auto inc() {
    return [this] { std::cout << value++; };
  }
};

int main() {
  auto c1 = Counter{10}.inc();
  c1();
  c1();
}
```

code

Evolution of lambdas in C++

C++20

Few of the enhancements:

- Template lambdas
- Pack expansion in lambda init-capture

Evolution of lambdas in C++

Template lambdas

```
template<typename T, typename U=T>
struct point { T x; U y; };

...

auto flip = []<typename T, typename U>(const point<T,U>& p)
    {
       point<U,T> new_p{p.y,p.x};
       return new_p;
    };
```

code

Variadic lambda functions

```
auto sum = [] (auto... value) {
  return (0 + ... + value);
};

std::cout << sum(1,3,5,6,9) << std::endl;
```

code

Evolution of lambdas in C++

Pack expansion in lambda init-capture

```
auto delay_invoke = [](auto fn, auto... args) {
    return [fn=fn,...args=std::move(args)]() {
        return fn(args...);
    };
};

...

int main()
{
  auto d_fn = delay_invoke(sum,5,4,3,2,1);
  std::cout << d_fn();
}
```

code

std::function type

Class template std::function is a general-purpose polymorphic function wrapper.

Instances of std::function can store, copy, and invoke any Callable target (functions, lambda expressions, bind expressions, or other function objects, as well as pointers to member functions and pointers to data members).

```
using fn_t = std::function<int(int,int)>;

fn_t f1 = fn_ptr;
fn_t f2 = [] (auto a, auto b) { return a+b; };
fn_t f3 = std::bind(minus, _2, _1);
```

code

Partial function application (Currying)

https://thispointer.com/stdbind-tutorial-and-usage-details/

https://en.cppreference.com/w/cpp/utility/functional/bind

References

- CppCon 2019: Arthur O'Dwyer “Back to Basics: Lambdas from Scratch”
- Lambdas: From C++11 to C++20
- https://en.cppreference.com/

YEG  MEETUP

Introduction to C++ Lambdas

Antal Buss

May 12 2020

Thank you