Lambdas in C++

Pan Xiaoyue

Smarkets

July 11, 2018

An lambda example

- A lambda is an anonymous function, e.g.,

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

- Call the lambda:

```
int x = simple_lambda(100);
std::cout << "x:" << x << "\n";
```

Exercise: lambda syntax

1. Write a lambda which prints an integer.

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

int main () {
    std::vector < int > a {1 ,2 ,3};
    //TODO: define print_lambda here
    //std::for_each will call print_lambda
    std::for_each (a.begin(), a.end(), print_lambda);
    return 0;
}
```

2. Generalise it to not just integer

C++'s lambda definition

Syntax: [captures] (params) $\rightarrow$ ret { body }

- captures: can be both by *value* and by *reference*

```
std::vector<int> a{0,1,2,3,4,5,6};
//lambda1 captures a by value
auto lambda1 = [a](int x) { return a[x]; };
//lambda2 captures a by reference
auto lambda2 = [&a](int x) { a[x]++; };
```

- What's the result of *lambda1(0)* and *lambda2(0)*?

Why do we need captures?

- Because the code that calls the lambda *may not have access* to the needed variables

```
std::vector<int> a{0,1,2,3,4,5,6};
auto lambda2 = [&a](int x) { a[x]++; };

std::vector<int> indices{0,3};
//std::for_each (lambda2's caller) can't access a
std::for_each(indices.begin(), indices.end(), lambda2);
```

Exercise: captures

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

int main() {
    std::vector<int> a{1,2,3};
    int sum = 0;
    //TODO: define a lambda and use std::for_each
    //Afterwards, sum should be 6
    return 0;
}
```

A little detour: structs and classes

- They are almost the same (diff: default accessibility of member variables and functions)

```
struct Squirrel {
  std::string name;
  int age;
  Squirrel(const std::string &name, int age):
    name(name), age(age){}
};
Squirrel new_squirrel("Charlie", 5);
```

- Constructors and member initializer lists

Callable object

- A callable object is an object that *can be called like a function*, e.g.,

```
a_callable_object();
another_callable_object(args);
```

- How do we make an object callable? e.g.,

```
Squirrel new_squirrel("Charlie", 5);
new_squirrel();
```

  - By overloading the operator *()*.

Make squirrels callable

- Overload *()* in the Squirrel struct

```
#include <string>
#include <iostream>

int main() {
    struct Squirrel {
        std::string name ;
        int age ;
        Squirrel (const std::string& name , int age):
            name (name) , age (age){}
        void operator()() {
            std::cout << "Squirrel " << name << " is called.\n";
        }
    };
    Squirrel new_squirrel ( "Charlie" , 5);
    new_squirrel();
    return 0;
}
```

Why do we talk about struct and callable object?

- A lambda is a callable object.

The lambda in

```
//Assume we have std::vector<int> a{0,1,2,3,4,5,6};
auto lambda = [&a](int x) { a[x]++; };
```

is equivalent to

```
struct ExplicitLambda {
  // captures go here
  std::vector<int> &a;
  ExplicitLambda (std::vector<int> &vec): a(vec) {}
  // body of the lambda goes into a callable operator
  void operator() (int x) {
    a[x]++;
  }
};
```

- lambda(0) is the same as calling an object of the struct with 0.

Exercise: sort the squirrels

Write a lambda which can be used by std::sort to sort Squirrel objects by age/name.

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

struct Squirrel {
    std::string name;
    int age;
    Squirrel (const std::string &name , int age): name(name) , age(age){}
};
int main() {
    std::vector<Squirrel> squirrels
    {{"Emily", 4}, {"Luis", 1},
       {"Charlie", 5}, {"Sara", 9}};
    //TODO: write a comparison_lambda for Squirrel
    //TODO: sort the squirrels
  return 0;
}
```

Exercise: one way to shoot yourself in the foot

What does this code print?

```
#include <functional>
#include <iostream>

std::function<void()> create_alarm () {
  std::string message = "Wakeup call!\n";
    return [&](){ std::cout << message; };
}

int main() {
  create_alarm()();
}
```

Exercise: if you find the exercises too simple

What's the output of this code? (Taken from https://en.cppreference.com/w/cpp/language/lambda)

```
#include <iostream>

int main()
{
    int a = 1, b = 1, c = 1;

    auto m1 = [a, &b, &c]() mutable {
        auto m2 = [a, b, &c]() mutable {
            std::cout << a << b << c << '\n';
            a = 4; b = 4; c = 4;
        };
        a = 3; b = 3; c = 3;
        m2();
    };

    a = 2; b = 2; c = 2;

    m1();
    std::cout << a << b << c << '\n';
}
```