Returning a lambda with captures from a function

In C++11 one can write lambdas with captures (and that's awesome!)

```
auto myfunc = [&] (int i) {return i + j;}; // j being somewhere in the lambda's context
```

That is awesome! However, it would be very nice if one could return such a lambda from a function, or even from another lambda. Is this possible at all?

c++ function c++11 lambda

edited Dec 17 '14 at 15:27
Maroun Maroun
51.6k 12 84 146

asked Dec 17 '14 at 15:25
Matteo Monti
1,517 3 21 46

Yup, its possible. There are plenty of examples: stackoverflow.com/… – matsjoyce Dec 17 '14 at 15:28

Closures in C++ don't extend the lifetime of variables captured by reference. That's different from what you see in other languages. I blogged on this once – Kos Dec 17 '14 at 15:33

1 @Kos C++ speaks of "capture", not "closure". I suspect that this is the reason; there is no closure, in the classical sense. – James Kanze Dec 17 '14 at 15:39

@JamesKanze C++11 [expr.prim.lambda]/2: "The evaluation of a *lambda-expression* results in a prvalue temporary (12.2). This temporary is called the *closure object*." C++11 speaks of "closure," but as with so many terms stolen from functional programming we use it to mean something else ;) – Casey Dec 17 '14 at 15:52

@Casey It's a "closure object", not a "closure":-). Seriously, there was never any question of C++ implementing true closure; that would require garbage collection of some sort, and the possibility that local variables weren't on the stack. – James Kanze Dec 17 '14 at 17:15

2 Answers

In C++11, you'd have to wrap it in a function object of known type to return it from a function:

```
std::function<int(int)> get_lambda() {
    return [&] (int i) {return i + j;};
}
```

In C++14, you can use `auto` to return the lambda type itself:

```
auto get_lambda() {
    return [&] (int i) {return i + j;};
}
```

In either dialect, you could return it from a lambda:

```
auto get_lambda = [&] {return [&] (int i) {return i + j;};};
```

Note that you wouldn't want to return this particular lambda, since it captures a reference to a local variable `j`. The variable will be destroyed, leaving the reference invalid, when the function returns.

edited Dec 17 '14 at 15:36

answered Dec 17 '14 at 15:29
Mike Seymour
179k 11 226 418

std::function<int(int)> get_lambda() { int j=3; return [&] (int i) {return i + j;}; } int main() { cout<<get_lambda()(4)<<endl; } It just prints out some random value! Prints 32771… what's going on in here?? – Matteo Monti

Dec 17 '14 at 15:33

@MatteoMonti: As I mentioned in the last sentence, you're capturing a reference to a local variable, which has been destroyed by the time you call the lambda. Don't do that. – Mike Seymour Dec 17 '14 at 15:35

@MatteoMonti: Note that your lambda is capturing `j` by reference, which is only in scope during the call to `get_lambda()` . – Jason R Dec 17 '14 at 15:35

Oh gosh. What a fool. That was obvious. Thank you very much! – Matteo Monti Dec 17 '14 at 15:37

Ok, but what if I wanted it to work with some temporary variable? Say that I want a function to which I can give an int j and it returns a lambda that provided with an int i returns i + j. How could I do this? j will always be destroyed at the end of the function that creates the lambda, right? How can I work around this? – Matteo Monti Dec 17 '14 at 15:44

You can return a lambda with captures. Since c++14 introduced automatic return types this is particularly easy.

Here's an example of how to create a function that will apply a binary operator (multiplication here, but it's a template parameter so you can give it anything) with one argument fixed during creation and the second provided during calling

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

template<typename F, typename T>
auto opby(F func, T arg)
{
    return [=](auto val) {
        return func(val, arg);
    };
}


int main()
{
 auto mu = opby(std::multiplies<int>(), 2);
 std::cout << mu(3) << std::endl;
}
```

it prints 6. The returned lambda had **captured by value its enclosing scope** so a function is created that will multiply by two any argument you'll give it.

The only caveat is when **capturing by reference** : you have to ensure that the closure won't transcend the lifetime of captured objects.

edited Dec 17 '14 at 15:57

answered Dec 17 '14 at 15:31

Lorah Attkins
762 1 13