Functional

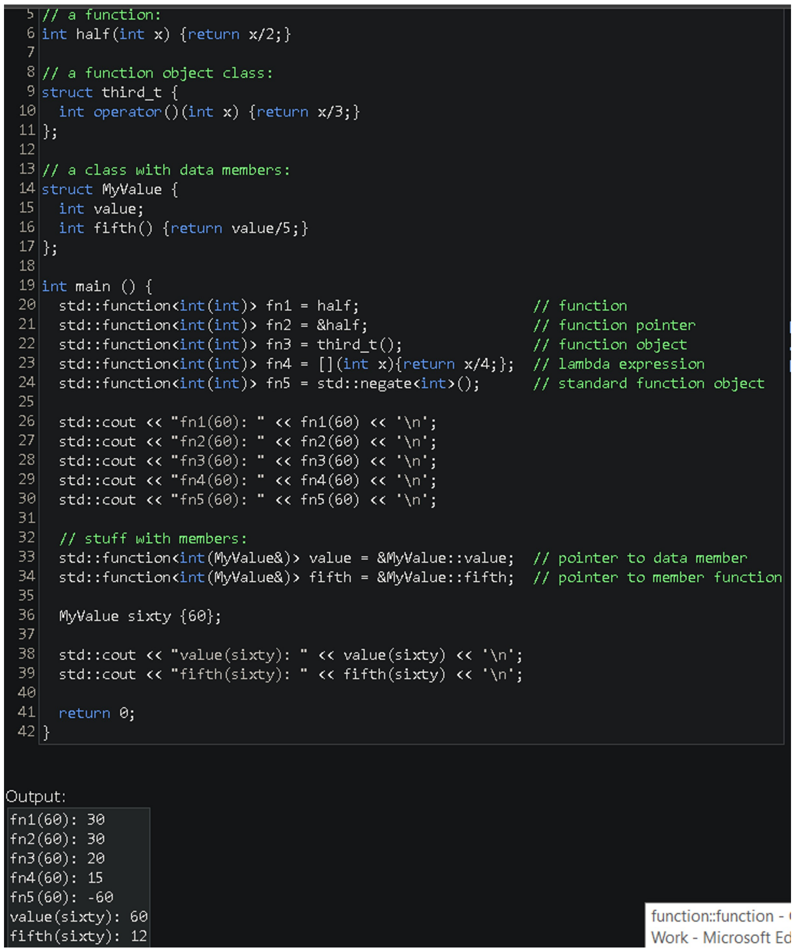
09 June 2021 07:07

1. <functional> provides callable objects which can be used in algo funcs

1. Wrapper classes - classes that hold objects and have an interface similar to that obj also adding or changing some of features
   a. std::function - class that wraps callable element into copyable object (function, function-pointer, functors).. Type depends solely on its call signature
   b. std::reference_wrapper - class that wraps reference
2. Functions - create objects of wrapper classes -> ref, cref (create reference_wrapper to a const T reference)
3. Operator classes - which performs relational, logical, arithmetic, bitwise operations on fundamental and user-defined types.

2. std::function

Line 23 is std::function<int(int)> fn4 = std::function<int(int)>(lambda).. So are other lines... typical Class initialization by implicit conversion

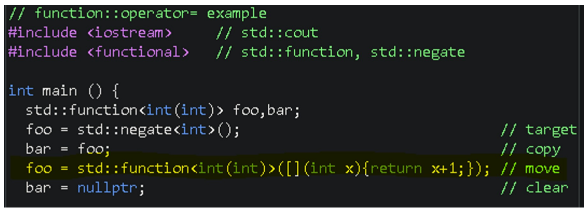


```
 5 // a function:
 6 int half(int x) {return x/2;}
 7
 8 // a function object class:
 9 struct third_t {
10   int operator()(int x) {return x/3;}
11 };
12
13 // a class with data members:
14 struct MyValue {
15   int value;
16   int fifth() {return value/5;}
17 };
18
19 int main () {
20   std::function<int(int)> fn1 = half;                    // function
21   std::function<int(int)> fn2 = &half;                   // function pointer
22   std::function<int(int)> fn3 = third_t();               // function object
23   std::function<int(int)> fn4 = [](int x){return x/4;};  // lambda expression
24   std::function<int(int)> fn5 = std::negate<int>();      // standard function object
25
26   std::cout << "fn1(60): " << fn1(60) << '\n';
27   std::cout << "fn2(60): " << fn2(60) << '\n';
28   std::cout << "fn3(60): " << fn3(60) << '\n';
29   std::cout << "fn4(60): " << fn4(60) << '\n';
30   std::cout << "fn5(60): " << fn5(60) << '\n';
31
32   // stuff with members:
33   std::function<int(MyValue&)> value = &MyValue::value;  // pointer to data member
34   std::function<int(MyValue&)> fifth = &MyValue::fifth;  // pointer to member function
35
36   MyValue sixty {60};
37
38   std::cout << "value(sixty): " << value(sixty) << '\n';
39   std::cout << "fifth(sixty): " << fifth(sixty) << '\n';
40
41   return 0;
42 }
```

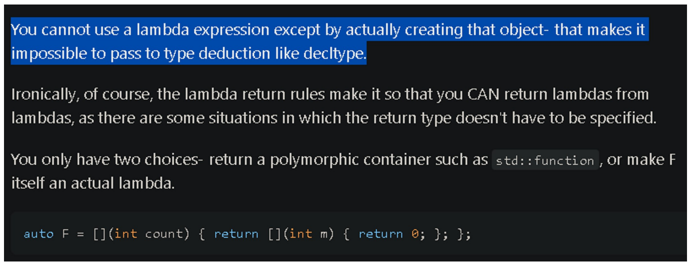
Output:

```
fn1(60): 30
fn2(60): 30
fn3(60): 20
fn4(60): 15
fn5(60): -60
value(sixty): 60
fifth(sixty): 12
```

Move initialization.. std::function<int(int)>(lambda) creates a r-value function object of lambda



```
// function::operator= example
#include <iostream>     // std::cout
#include <functional>   // std::function, std::negate

int main () {
  std::function<int(int)> foo,bar;
  foo = std::negate<int>();                              // target
  bar = foo;                                             // copy
  foo = std::function<int(int)>([](int x){return x+1;}); // move
  bar = nullptr;                                         // clear
```



You cannot use a lambda expression except by actually creating that object- that makes it impossible to pass to type deduction like decltype.

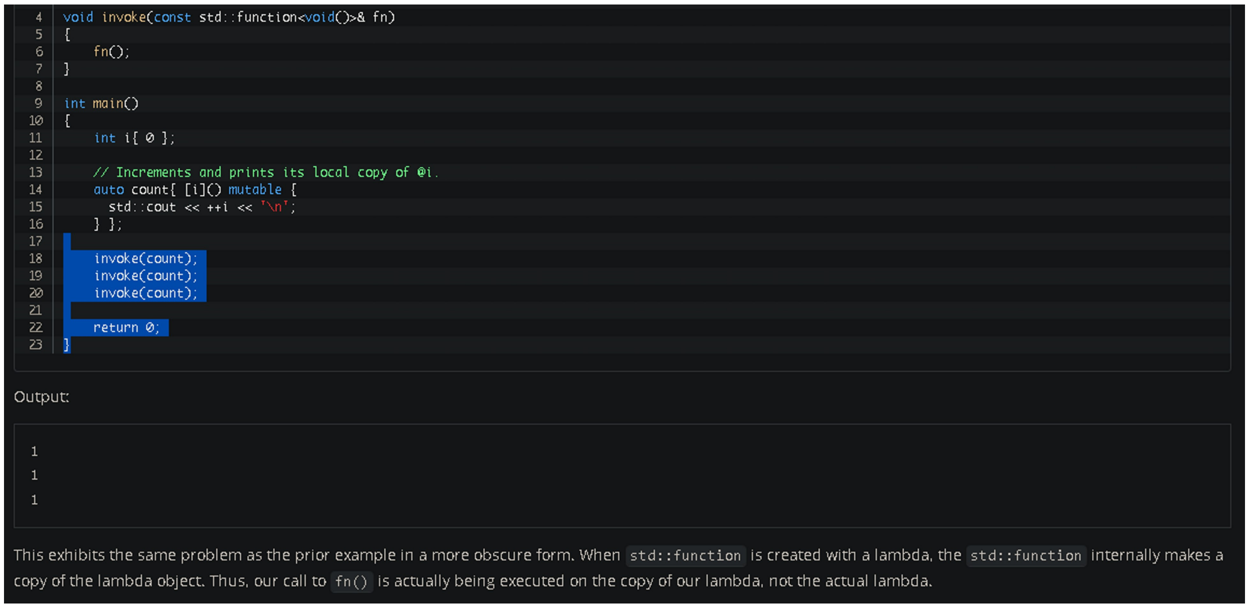
Ironically, of course, the lambda return rules make it so that you CAN return lambdas from lambdas, as there are some situations in which the return type doesn't have to be specified.

You only have two choices- return a polymorphic container such as `std::function`, or make F itself an actual lambda.

```
auto F = [](int count) { return [](int m) { return 0; }; };
```

3. Operators supported =, bool to check to see if callable, () operator to call

4. Standard library functions may copy function objects,. To avoid it
   Fortunately, C++ provides a convenient type (as part of the <functional> header) called std::reference_wrapper that allows us to pass a normal type as if it were a reference. For even more convenience, a std::reference_wrapper can be created by using the std::ref() function. By wrapping our lambda in a std::reference_wrapper, whenever anybody tries to make a copy of our lambda, they'll make a copy of the reference instead, which will copy the reference rather than the actual object.
   https://www.learncpp.com/cpp-tutorial/lambda-captures/

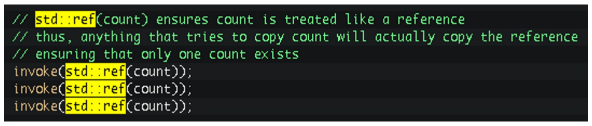


```
void invoke(const std::function<void()>& fn)
{
    fn();
}

int main()
{
    int i{ 0 };

    // Increments and prints its local copy of @i.
    auto count{ [i]() mutable {
      std::cout << ++i << '\n';
    } };

    invoke(count);
    invoke(count);
    invoke(count);

    return 0;
}
```

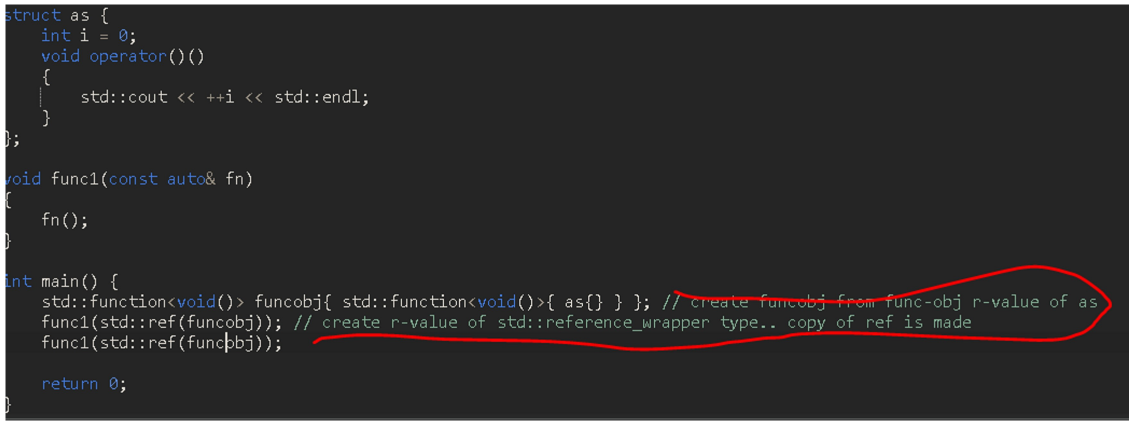
Output:

```
1
1
1
```

This exhibits the same problem as the prior example in a more obscure form. When `std::function` is created with a lambda, the `std::function` internally makes a copy of the lambda object. Thus, our call to `fn()` is actually being executed on the copy of our lambda, not the actual lambda.



```
// std::ref(count) ensures count is treated like a reference
// thus, anything that tries to copy count will actually copy the reference
// ensuring that only one count exists
invoke(std::ref(count));
invoke(std::ref(count));
invoke(std::ref(count));
```



```
struct as {
    int i = 0;
    void operator()()
    {
        std::cout << ++i << std::endl;
    }
};

void func1(const auto& fn)
{
    fn();
}

int main() {
    std::function<void()> funcobj{ std::function<void()>{ as{} } }; // create funcobj from func-obj r-value of as
    func1(std::ref(funcobj)); // create r-value of std::reference_wrapper type.. copy of ref is made
    func1(std::ref(funcobj));

    return 0;
}
```

5. std::greater<type>(a,b) returns bool if first arg is > than second.
   Ik

6. F
   Ik

7. D
   Ik

8. F
   I

9. F
   I

10. F
    I

11. F

12. F
    I

13. F
    I
14. F
    I
15. F
    I
16. F
    I
17. F
    I
18. D
    I
19. F
    I
20. F
    I
21. F
    I
22. F
23. F
    I
24. F
    I
25. F
    I
26. F
    I
27. F
    I
28. F
    I
29. D
    I
30. F
    I
31. F
    I
32. F
    I
33. F
34. F
    I
35. F
    I
36. F
    I
37. F
    I
38. F
    I
39. F
    I
40. D
    I
41. F
    I
42. F
    I
43. F
    I
44. F
45. F
    I
46. F
    I
47. F
    I

48. F
I
49. F
I