2-lambda-functions

June 3, 2024

1 Lambda functions

> C++11, C++14, and C++17 introduced quite a few features that make writing programs in the functional style much easier. The additional features are mostly syntactic sugar, but important syntactic sugar, in the form of the `auto` keyword and lambdas.
> *Ivan Cukic*

1.1 Lambda functions: anonymous and defined on the fly

Many standard library algorithms, such as `std::for_each`, take a function as final argument. Starting with C++11, rather than defining such function separately, one can define it directly where it is used, by replacing its name with `[]`. These unnamed functions are called **anonymous functions** or **lambda functions**.

[1]:
```
#include <algorithm>
#include <vector>
#include <iostream>
```

[2]:
```
std::vector<int> v {1,2,3,4,5} ;
std::for_each(v.begin(),v.end(), []( int i ){ std::cout<<i*i<<' ' ; } ) ;
std::cout<<std::endl ;
```

```
1 4 9 16 25
```

1.2 To modify elements, pass the argument by reference

If your lambda must modify the received element, you have to (naturally) declare this element as a reference:

[4]:
```
std::vector<int> v {1,2,3,4,5} ;

std::for_each(v.begin(),v.end(),[]( int & i )
 { i = 2*i ; }) ;

std::for_each(v.begin(),v.end(),[]( int i )
 { std::cout<<i<<' ' ; }) ;
std::cout<<std::endl ;
```

```
2 4 6 8 10
```

1.3 Return type

If your lambda function needs to return a result, it's about as easy. The compiler can guess the return type of your lambda by inspecting the instruction `return`. In the following example, we combine `std::sort` with a lambda so to reverse the ordering.

[5]:
```
std::vector<int> v {1,2,3,4,5} ;

std::sort(v.begin(), v.end(), []( int lhs, int rhs )
 { return (lhs>rhs) ; }) ;

std::for_each(v.begin(), v.end(), []( int i )
 { std::cout<<i<<' ' ; }) ;
```

```
5 4 3 2 1
```

To improve the readability of the code, or to help the compiler in certain ambiguous cases, we can explain the return type of a lambda. This is called **trailing return type declaration**.

[6]:
```
std::vector<int> v {1,2,3,4,5} ;

std::sort(v.begin(), v.end(), []( int lhs, int rhs ) -> bool
 { return (lhs>rhs) ; }) ;

std::for_each(v.begin(), v.end(), []( int i ) -> void
 { std::cout<<i<<' ' ; }) ;
```

```
5 4 3 2 1
```

These early examples already show practical uses of lambdas. But their killing feature is their ability to capture local variables.

1.4 Local variables capture

A lambda function, when introduced with empty brackets "[]", can only access to its own arguments and global variables, like any other function. Yet, one can insert between the brackets a list of context variables to be captured, by value or by reference: * `[x,y,&j]` : `x` and `y` by value, `j` by reference ; * `[]` : nothing captured ; * `[&]` : all variables by reference ; * `[=]` : all variables by value ; * `[=,&j]` : all variables by value, except `j` by reference ; * `[&,j]` : all variables by reference, except `j` by value.

[7]:
```
std::vector<int> v {1,2,3,4,5} ;

int multiplier = 2 ;
std::for_each(v.begin(), v.end(), [multiplier]( int & i )
 { i = multiplier*i ; }) ;

int accumulator = 0 ;
std::for_each(v.begin(), v.end(), [&accumulator]( int i )
 { accumulator += i ; }) ;
```

```
std::cout<<accumulator<<std::endl ;
```

30

A lambda is equivalent to some function-object, which capture the variables as members, and reuse them in the implementation of `operator()`:

[8]:
```
class Multiplier
 {
  public :
    Multiplier( int m ) : m_m{m} {}
    void operator()( int & i ) { i = m_m*i ; }
  private :
    int m_m ;
 } ;
```

BEWARE: when capturing by reference, as with any reference, the behavior is undefined if the original variable disappears before the lambda function is used. And from a *functional* point of view, such a kind of side-effect context modification **is rather impure**!

In the case where we want to capture a large object without duplicating it and without making it modifiable, we can regret that C ++ does not allow to capture a variable as a **constant reference**.

1.5 Storing and reusing lambdas

A lambda function is a "first class object", and can be stored in a variable, to be reused later as any normal function. The type of the lambda is implementation-dependent. The usual practice is to declare above variable `auto`.

A noteworthy difference with an ordinary function : you can **nest it** in any block.

Again, beware not to capture by reference something which may be destrcuted before the lambda is used.

[10]:
```
#include <vector>
#include <algorithm>
#include <iostream>
```

[11]:
```
int m ;
auto mult = [&m]( int i ){ std::cout<<(m*i)<<" " ; } ;
std::vector<int> v {1,2,3,4,5} ;

m = 2 ;
std::for_each(v.begin(), v.end(), mult) ;
std::cout<<std::endl ;

m = 3 ;
std::for_each(v.begin(), v.end(), mult) ;
std::cout<<std::endl ;
```

```
2 4 6 8 10
3 6 9 12 15
```

1.6 Generic lambdas

If you want to reuse your lambda with different input types, you can also use `auto` in the functions parameters:

[12]:
```
auto print = []( auto val ){ std::cout<<val<<' ' ; } ;

std::vector<int> vi{1,2,3,4,5} ;
std::for_each(vi.begin(),vi.end(),print) ;
std::cout<<std::endl ;

std::vector<double> vd{1.1,2.2,3.3,4.4} ;
std::for_each(vd.begin(),vd.end(),print) ;
std::cout<<std::endl ;
```

```
1 2 3 4 5
1.1 2.2 3.3 4.4
```

The first `auto` triggers type inference. The second is rather some simplified form of `template`. If we look for the equivalent function-object, it might look like this:

[13]:
```
class Print
 {
  public :
    template< typename Value >
    void operator()( Value val )
     { std::cout<<val<<' ' ; }
 } ;
```

Note that it is the execution operator (`operator()`) that is parameterized, and not the class itself.

2 Questions ?