70-lambdas

June 2, 2024

1 Lambda functions

Lambda (or anonymous) functions were introduced in C++11 and enhanced in C++14. They can be defined on-the-fly anywhere in the code where they are needed. This avoid a previous declaration of the function, useless when the function is to be used only once. Moreover, the capture of local variables avoids the tedious definition of an object-function (whose `operator()` is overloaded). In simple cases, the resulting code is more concise and clearer.

1.1 One-time use, anonymous, functions

Some ordinary functions are meant to be used only once. Since nested functions are not allowed in C++, one must pollute the global namespace:

[1]:
```cpp
#include <vector>
```

[2]:
```cpp
int reduce( std::vector<int> collection, int accumulator, int (*op)( int, int ) ) {
  for ( int element : collection ) {
    accumulator = op(accumulator,element) ;
  }
  return accumulator ;
}
```

[3]:
```cpp
int add( int val1, int val2 ) { return (val1+val2) ; }
```

[4]:
```cpp
int multiply( int val1, int val2 ) { return (val1*val2) ; }
```

[5]:
```cpp
#include <iostream>
```

[6]:
```cpp
std::vector<int> numbers = { 1, 2, 3, 4, 5 } ;
std::cout<<reduce(numbers,0,add)<<std::endl ;
std::cout<<reduce(numbers,1,multiply)<<std::endl ;
```

```
15
120
```

C++11 allows one-the-fly definition of an anonymous function, where it is to be used. **The function name is replaced with `[]`**. It is called a *lambda*:

[7]:

```
std::vector<int> numbers = { 1, 2, 3, 4, 5 } ;
std::cout<<reduce(numbers,0,[](int i1, int i2){ return i1+i2 ; })<<std::endl ;
std::cout<<reduce(numbers,1,[](int i1, int i2){ return i1*i2 ; })<<std::endl ;
```

```
15
120
```

1.2 Return type

As above, the compiler can guess the return type of your lambda by inspecting the instructions `return`. 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**.

[8]:

```
int addition = reduce( numbers, 0, [](int i1, int i2) -> int { return i1+i2 ; } ) ;
std::cout << addition << std::endl ;
```

```
15
```

1.3 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:

[9]:

```
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
```

1.4 Capturing local variables

Like any ordinary function, it only sees its arguments, local variables and global variables from the file. This does not compile:

[10]:

```
%%file tmp.capture.cpp

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

int main() {
  std::vector<int> numbers { 1, 2, 3, 4, 5 } ;
  int coef { 3 } ;
  std::for_each(
    numbers.begin(),numbers.end(),
    []( int a_n ){ std::cout << (coef*a_n) << " " ; }
  ) ;

}
```

```
Writing tmp.capture.cpp
```

[11]:

```
!rm -f tmp.capture.exe && g++  -std=c++17 tmp.capture.cpp -o tmp.capture.exe
```

```
tmp.capture.cpp: In lambda function:
tmp.capture.cpp:12:34: error: ‘coef’ is not captured
   12 |     []( int a_n ){ std::cout << (coef*a_n) << " " ; }
      |                                  ^~~~
tmp.capture.cpp:12:6: note: the lambda has no capture-default
   12 |     []( int a_n ){ std::cout << (coef*a_n) << " " ; }
      |      ^
tmp.capture.cpp:9:7: note: ‘int coef’ declared here
    9 |   int coef { 3 } ;
      |       ^~~~
```

A lambda function can include between its initial brackets a list of variables to be “captured” from its context, by value or by reference: * `[]`: no variable; * `[x,y,&j]`: `x` andy by value and `j` by reference; * `[&]`: all variables by reference; * `[=]`: all variables by copy; * `[=,&j]`: all variables by copy except `j` by reference; * `[&,j]`: all variables by reference, except `j` by copy.

Capturing `coef` allows to resolve the previous problem:

[12]:

```
std::vector<int> numbers { 1, 2, 3, 4, 5 } ;
int coef = 3 ;
std::for_each(
  numbers.begin(),numbers.end(),
  [coef]( int a_n ){ std::cout << (coef*a_n) << " " ; }
) ;
```

```
3 6 9 12 15
```

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

[13]:

```
class Multiplier
 {
  public :
    Multiplier( int a_coef ) : m_coef(a_coef) {}
    void operator() ( int a_n ) { std::cout << (m_coef*a_n) << " " ; }
  private :
    int const m_coef ;
```

```
 } ;
```

[14]:
```
std::vector<int> numbers { 1, 2, 3, 4, 5 } ;
Multiplier m { 3 } ;
std::for_each(numbers.begin(),numbers.end(),m) ;
```

```
3 6 9 12 15
```

1.5 Capturing by reference

[15]:
```
std::vector<int> v {1,2,3,4,5} ;

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

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

```
15
```

BEWARE: when capturing by reference, as with any reference, the behavior is undefined if the original variable disappears before the lambda function is used.

1.6 Storing and reusing lambdas

[22]:
```
#include <vector>
#include <algorithm>
#include <iostream>
```

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`.

[25]:
```
auto constexpr mult2 = []( int i ){ std::cout<<(2*i)<<" " ; } ;

std::vector<int> v1 {1,2,3,4,5} ;
std::for_each(v1.begin(), v1.end(), mult) ;
std::cout<<std::endl ;

std::vector<int> v2 {6,7,8} ;
std::for_each(v2.begin(), v2.end(), mult) ;
std::cout<<std::endl ;
```

```
3 6 9 12 15
18 21 24
```

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

[28]:

```
void process( std::vector<int> const & v ) {
  auto constexpr mult2 = []( int i ){ std::cout<<(2*i)<<" " ; } ;
  std::for_each(v.begin(), v.end(), mult) ;
  std::cout<<std::endl ;
}

std::vector<int> v1 {1,2,3,4,5} ;
process(v1) ;

std::vector<int> v2 {6,7,8} ;
process(v2) ;
```

```
3 6 9 12 15
18 21 24
```

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

[29]:

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

[31]:

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

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

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

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

1.7 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 Take away

- a lambda function is anonymous, and usually meant to be used once ;
- yet one can store it in a variable, which makes it some kind of nested function ;
- thanks to the capture, lambda functions are a concise version of object-functions ;
- a generic lambda is equivalent to an object-function with a template operator().

3 Questions ?

4 Exercise

Replace below `random_unit` and `Pow` with lambdas functions. Make sure that you always get the same end result throughout your trials.

[10]:

```
%%file tmp.lambdas.cpp

#include <vector>
#include <algorithm>
#include <numeric>
#include <iostream>
#include <cmath>

// random double value in [-1,1]
void random_unit( double & a_value )
 { a_value = ((2.*std::rand())/RAND_MAX-1.) ; }

// compute value^degree
struct Pow
 {
  int m_degree ;
  Pow( int a_degree ) : m_degree {a_degree} {}
  double operator()( double a_value ) const
   { return std::pow(a_value,m_degree) ; }
 } ;

// main program
int main()
 {
  int const DIM {10} ;
  int const DEGREE {5} ;

  // generate random input
  std::vector<double> input(DIM) ;
  std::for_each(input.begin(),input.end(),random_unit) ;

  // compute output
  std::vector<double> output(DIM) ;
  std::transform(input.begin(),input.end(),output.begin(),Pow(DEGREE)) ;

  // print sum
  std::cout<<std::accumulate(output.begin(),output.end(),0.)<<std::endl ;
}
```

```
Writing tmp.lambdas.cpp
```

[8]:

```
!rm -f tmp.lambdas.exe && g++  -std=c++17 tmp.lambdas.cpp -o tmp.lambdas.exe
```

[9]:

```
!./tmp.lambdas.exe
```

```
0.599801
```