3-higher-order-functions

June 3, 2024

1 Higher-order functions

We have already seen that a "functional" feature of C++, already present in early C++, is to have functions which take other functions as arguments (i.e. the standard library algorithms). The counterpart of this feature, made possible by modern C++ lambdas, is to write functions that build other functions and return them as result. One can now manage functions as first-class objects : use, construct, compose them.

1.1 Taking a lambda as parameter

If you want to create your own functions which takes a lambda as parameter (higher-order functions), one issue is again the detailed type of the lambda function. A usual practice is to rely on a template parameter.

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

[2]:
```
template< typename Collection, typename Function >
void for_all( Collection const & col, Function f )
 { std::for_each(col.begin(), col.end(), f) ; }
```

[3]:
```
auto print = []( auto val ){ std::cout<<val<<' ' ; } ;

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

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

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

1.2 Returning a lamdda as result

The complementary face of the previous practice, made possible by lambdas, is to write **functions that build other functions** and return them as a result. The lambda type issue is solved by

setting the return type as `auto`.

[4]:
```
auto multiplier( int m )
 { return [m]( int i ){ std::cout<<(m*i)<<' ' ; } ; }
```

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

for_all(v,multiplier(2)) ;
std::cout<<std::endl ;

for_all(v,multiplier(3)) ;
std::cout<<std::endl ;
```

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

2 Questions ?

3 Exercise

Try to complete the code below, where we define a higher-order function capable of building the composite of two functions `f1` and `f2`. The resulting function must return `f1(f2(i))` for any input value `i`.

[ ]:
```
%%file tmp.functions.cpp

#include <iostream>
#include <array>
#include <algorithm>

template< typename Function1, typename Function2 >
... compose( Function1 f1, Function2 f2 )
 { return ... ; }

int square( int i ) { return i*i ; }
void display( int i ) { std::cout<<i<<std::endl ; }

int main()
 {
  std::array<int,5> table { 1, 2, 3, 4, 5 } ;
  std::for_each(table.begin(),table.end(),compose(display,square)) ;
 }
```

[ ]:
```
!rm -f tmp.functions.exe && g++ -std=c++17 tmp.functions.cpp -o tmp.functions.
 ↪exe
```

[ ]:
```
!./tmp.functions.exe
```