Programmazione Avanzata per il Calcolo Scientifico
Advanced Programming for Scientific Computing
Lecture title: Functors and lambdas

Luca Formaggia

MOX
Dipartimento di Matematica
Politecnico di Milano

A.A. 2014/2015

Functors (function object)

std predefined functors

Lambda calculus

Capture specification

Function adapters and binders

Bind vs Lambda

Function type wrappers

Functors (function object)
    std predefined functors

Lambda calculus
    Capture specification

Function adapters and binders
    Bind vs Lambda

Function type wrappers

Functors (function object)

- std predefined functors

Lambda calculus

- Capture specification

Function adapters and binders

- Bind vs Lambda

Function type wrappers

Functors (function object)

A functor is a class which overloads the call operator (`operator()`). An object of this type is called function object since it may be used with a semantic very similar to that of a function.

```
struct Cube{
  double operator()(double const & x){return x*x*x;}
};
...
Cube cube; // a function object
auto y= cube(3.4)// calls operator()
auto z= Cube()(8.0)// Cube() is the default constr,
```

If the call operator returns a `bool` the function object is also called a predicate. A functor is stateless if a call to the call operator does not change the state of the object. Functors used in the standard algorithms should be stateless.

Which are the advantages?

The advantage of a functor is that it has a state, so it can store additional information to be used to calculate the result

```
class StiffMatrix{
public:
 foo(Mesh const & m, double const & vis=1.0):
     _mesh(m),_visc(vis){}
 Matrix operator()();
private:
 Mesh const & _mesh;
 double _visc;
};
...
StiffMatrix K(myMesh,4.0); //function object
...
Matrix A=K();// compute
```

Another advantage: efficiency (using templates)

```
struct Cube{
  double operator()(double const & x)const
          {return x*x*x;}
};
inline double fcube(double const & x){return x*x*x;}
// a template function
template<class T>
double dosomething(T const & f){... y=f(5.0);...}
```

Now I can do

```
double z=dosomething(fcube);
```

or

```
double z=dosomething(Cube());
```

obtaining the same result. In the first call `T` is resolved as a pointer to function, in the second the type `Cube`.

Inline does not work with pointers to functions

The difference is that in the first case since `fcube()` is passed by a pointer its inlining is suppressed in `dosomething()`.

While `Cube()` creates an object, not a pointer, and the call to `f.operator()(5.0)` will be inlined, producing a more efficient code if the call is made often in the program.

That's why functors are the preferred way to pass policies to the algorithms of the standard library.

std predefined function objects

Indeed the Standard Library provides under the header `<functional>` a large number of predefined functors that can be used, alone or combined with binders, in standard library algorithms:

```
vector<int> i={1,2,3,4,5};
vector<int> j;
transform (i.cbegin(),i.cend(),  // source
           back_inserter(j),     // destination
           negate<int>());
```

Now j={−1,−2,−3,−4,−5}. Here, `negate<T>` is a *unary functor* provided by the standard library.

Note: I need a `back_inserter` as I am inserting the transformed elements at the end (back) of vector `j`. The methods `cbegin()` and `cend()`, introduced in C++11, return constant iterators. Here `begin()` and `end()` would have worked the same, but since `i` is not changed constant iterators are better.

Some predefined functors

plus<T>	Addition (Binary)
minus<T>	Subtraction (Binary)
multiplies<T>	Multiplication (Binary)
divides<T>	Division (Binary)
modulus<T>	Modulus (Unary)
negate<T>	Negative (Unary)
equal_to<T>	equality comparison (Binary)
not_equal_to<T>	non-equality comparison (Binary)
greater, less ,greater_equal, less_equal	
logical_and<T>	Logical AND (Binary)
logical_or<T>	Logical OR (Binary)
logical_not<T>	Logical NOT (Binary)

For a full list have a look at this web page.

Lambda calculus (C++11 only)

The new standard has introduced a very powerful syntax to create short (and inlined) function quickly: the lambda calculus. With lambda function it is normally indicated an unnamed function. C++ lambda do indeed create expressions that may be passed as arguments to other functions, like pointers of function objects. But first look at a simple usage

```
auto f= [](double x){return 3*x;};// f is a lambda function
...
auto y=f(9.0); // y is equal to 27.0
```

Note that I did not need to specify the return type in this case, the compiler deduces it as decltype(3*x), which returns a **double**.

Lambda syntax

The definition of a lambda function is introduced by the `[]`, also called capture specification, the reason will be clear in a moment. We have two possible syntax

```
 [ capture spec]( arguments){ code;  return something}
```

or

```
 [ capture spec]( arguments)-> returntype
{ code
```

The second syntax is compulsory when the return type cannot be deduced automatically.

The capture specification allows you to use variables in the enclosing scope inside the lambda, either by value (a local copy is made) or by reference.

[]	Captures nothing
[&]	Captures all variables by reference
[=]	Captures all variables by making a copy
[=, &foo]	Captures any referenced variable by making a copy, but capture variable `foo` by reference
[bar]	Captures only `bar` by making a copy
[**this**]	Captures the `this` pointer of the enclosing class

The capture specification gives a great flexibility to the lambdas
We make some examples: return the first element i such that $i > x$ and $i < y$

```
#include<algorithm>
int f(vector<int> const &v,int x, int y){
auto pos = find_if (coll.cbegin(), coll.cend(), // range
     [=](int i) {return i > x && i < y;});// criterion
return *pos;
}
```

I have used the `find_if` algorithm which takes as third argument a predicate and returns the first element that satisfies it. Note the use of `cbegin` and `cend` (in fact not strictly necessary).

Another example

```
std::vector<double>a={3.4,5.6,6.7};
std::vector<double>b;
auto f=[&b](double c){b.emplace_back(c/2.0);};
auto d=[](double c){std::cout<<c<<" ";};
for (auto i: a)f(i); // fills b
for (auto i: b)d(i); //prints b
// b contains a/2.
```

A little note: avoid capturing all variables. It is better to specify the one you need.

Generic Lambdas (C++14)

The newest standard has increased the flexibility of the lambdas. Now you can allow lambda functions to derive the parameter type from the type of the arguments!

```
auto add=[](auto x, auto y){return x + y;};
double a(5), b(6);
string s1("Hello␣");
string s2("World");
auto c=add(a,b); //c is a double equal to 11
auto s3=add(s1,s2);// s is "Hello World"
```

An example of use of [this]

```
class foo{
public:..
  void prova();
private:
double _x;
vector<double> _v};
... // definition
void foo::prova(){
  auto prod=[this](double a){_x*=a;};
  std::for_each(_v.begin(),_v.end(),prod);
}
```

Now the method `prova()` compute a cumulative product of the contents of `v` and stores it in the member variable `_x`.

Function adapters and binders

A function adapter is a special function object that enables the composition of function objects with each other, with certain value, or with special functions. Here are the new adapters provided by C++11 (the old C++98 adapters are now deprecated).

`bind(op,args)`	Binds args to op
`mem_fn(op)`	Calls `op()` as member function for an object or pointer to object
`not1(op)`	Unary negation: `!op(param)`
`not2(op)`	Binary negation: `!op(param1,param2)`

We will describe here only the most powerful of them: `bind`. You find the description of the others in any good book (like the new edition of the Lippman).

`bind()` binds parameter for a callable object: if a function, member function, function object, lambda require some parameters you can bind them so specific or passed arguments. For passed arguments you can used predefined placeholders _1, _2,.. defined in the namespace `std::placeholders`. You need to include `<functional>`

A simple (useless) example

```
#include<iostream>
#include<functional>
double fun(double a, double b){return a*b;}

int main(){
  using namespace std;
  using namespace std::placeholders;
  auto f=bind(fun,3.0,_1);
  cout<<f(4);// calls fun(3,4)
}
```

The placeholder `_1` indicates the first (and only) passed argument to the bound function `f`. So `f(a)` is equivalent to `fun(3.0,a)`. Note the use of `using`, otherwise we should have written `std::placeholders::_1`.

`bind()` versus lambda

Everything that can be done by using `bind()` can be done with `lambda` functions.

Personally I find lambdas great!. Actually there is little reason to use `bind` now that we have lambdas!

Function type wrappers

An now the catch all function wrapper. The class `std::function<>` declared in `<functional>` provides polymorphic wrappers that generalize the notion of function pointer. It allows you to use callable objects (functions, member functions, function objects and lambdas) as first class objects.

```
int func(int, int);
...
// a vector of functions
vector<function<int(int,int)>> tasks;
tasks.push_back(fun);
tasks.push_back([](int x,int y){return x*y;});
for (auto i : tasks) cout<<i(3,4)<<endl;
```

It prints the result of `func(3,4)` and 12.

Function wrappers are very useful when you want to have a common interface to callable objects.

See the examples in RK45 and NonLinSys. The first implements a RK45 adaptive algorithm for integration of a scalar ODE and the other two algorithms for the solution of non-linear systems.

Function Expression Parsers

Functions in C++ must be defined at compile time. Sometimes however can be useful to be able to specify simple functions run-time, maybe reading them from a file.
To that purpose you can use a `parser`. A nice parser is muParser.
A copy of `muParser` is available in the directory `Extra`. To compile and install it (under the Examples/lib and Examples/include directories) just launch the script `install.sh` that you find in the `muParser` directory. Or follow the instruction you find on the web site.

A simple example of the use of muparser

You find a simple example on the use of muParser in
test_Muparser.cpp
and the other files in that directory.