Advanced Programming for Scientific Computing (PACS)
Lecture title: Functions, functors and lambdas

Luca Formaggia

MOX
Dipartimento di Matematica
Politecnico di Milano

A.A. 2022/2023

Functions in C++

In C++ functions may be free functions or member functions (also called methods) of a class. And a further subdivision can be made between normal functions and template functions (including automatic return functions).

For a free, non template function the usual declaration syntax is

```
ReturnType FunName(ParamsType...);
```

or

```
auto FunName(ParamsType...)-> ReturnType;
```

or (automatic return function)

```
auto FunName(ParamsType...);
```

alternative ways to define a function

The first two forms are practically equivalent. So choose the one you like most.

In the third possibility, the return type is deduced by the compiler, we will give more details later on.

Why (free) functions?

A function represents a map from data given in input (through the arguments) and an output provided by the returned value or possibly via an argument if the corresponding parameter is a non-const reference.

Consequently, a (free) function is usually (I say usually because there are exceptions) stateless, which implies that two different calls of a function with the same input produce the same output.

Therefore, you normally implement a function whenever what you need is indeed a map input/output, like in the standard mathematical definition $f : U \to V$.

void return type and [[nodiscard]]

The return type can be **void**, which means that the function is not returning anything. Indeed **void** is a type that indicates "no value".

In C++ it is not an error not using the returned value: this is a perfectly valid piece of code

```
double fun(double x);
...
fun(4.57);// OK, returned values is discarded.
```

If you want to get a warning if the value is discarded you should use the `[[nodiscard]]` attribute

```
[[nodiscard]] double fun(double x);
...
fun(4.57);// Warning issued
```

You may also indicate a string, printed in case of discarded value:

```
[[nodiscard("Strategic value")]] double fun(double x);
```

Function declaration and definition

A function pure declaration does not contain the body of the function, and is usually contained in a header file:

```
double f(double const &);
```

It is not necessary to give a name to the parameters (but you can if you want, and it is good for documentation).

The definition is usually contained in a source file, a part from (non fully specialized) function templates, automatic return functions, inline and contexpr functions, which should be defined in a header file.

```
double f(double const & x)
{
  //... do someting
  return y;// returns a value convertible to double
}
```

Passing by value and passing by reference

I assume you already know what is meant by passing a function argument "by value" or "by reference" . In fact, "passing by reference" simply means that the function parameter is a reference, the function operates on an "alias" of the argument, without making a copy:

- Using a non-const reference, a change made in the function changes the corresponding argument, the parameter can be considered as a possible output of the function. The argument cannot be a constant object or a constant expression;
- Using a **const** reference, the parameter is a read-only alias, so the argument can also be a constant object or a constant expression. The parameter cannot be changed in the function.

Important Note: In this lecture with "reference" we usually mean l-value references. When we will introduce move semantic, we will describe r-value references (&&).

Passing a value

If you pass an argument "by value", the value is copied (or possibly moved, as we will see in a later lecture) in the parameter, which is then a local variable of the function: a change in the parameter value does not reflect on the corresponding argument.

When you pass a value, the only reason you may want the parameter to be declared **const** is to have a safer code and be sure that any attempt of changing the parameter inside the function, even indirectly by passing it by reference to another function, is forbidden.

A Note: Another possibility of obtaining the output of function call via the arguments is by using pointers. This is the only technique available in C. In C++ references are preferred for this purpose.

A basic example

```
double fun (double x, double const & y, double & z)
{
 x = x +3; // I am changing a copy of the argument
 auto w= y; // I am copying the value bound to y into w, ok
 y = 6.0; // ERROR I cannot change y
 z = 3*y; // Ok the variable bound to z will change
 ....}
 ...
 double k=3.0;
 double z=9.0;
 auto c = fun(z, 6.0,k); //ok
 // k is now 18 z is still 9.0
 auto d = fun(z,k,6.0); //Error 6.0 cannot bind to a double &!
```

General guidelines I

- Prefer passing by reference when dealing with large objects. You avoid to copy the data in the function parameter and save memory.
- If the parameter is not changed by the function (it is an "input-only parameter"), either pass it by value (better if **const**) or as const reference (es: **const** Vector &).
- A temporary object or a constant expression may be given as argument only if passed by value or by constant reference (or...let's wait the lecture on move semantic)
- You can pass polymorphic objects only by reference or with pointers. Otherwise, polymorphism is lost!

General guidelines II

- In the "passing by value" case, i.e. when the parameters are not references, the use of **const** is less critical, since we are using in the function a copy of the argument. However, it does not hurt declaring **const** what is not changed inside the function (and helps the compiler to produce better machine code).
- In case of POD (**int**, **double**) there is little gain in passing by const reference. Particularly with **int**, since the compiler may decide to store the parameter in a CPU register. But it is just a matter of taste.

In Bindings/ you find an example about reference binding (look only at the part concerning l-value references).

What type does a function return?

A free function normally returns a value or **void** (no value returned).

Never return a non-const l-value reference, or pointer, to a local function object

without this would have worked. With it we get an error as we can't return a ref to m, since m is a local variable which will vanish when we return

```
Matrix & pippo()
{
    // An object in the scope of the function
    Matrix m=identity(10,10);
    return m;
}
```

This is wrong, even if you change it in Matrix **const** & pippo().

Always return by value objects created in the scope of a function

What type does a function return?

In some rare cases a function returns a l-value reference to an object passed by l-value reference. Normally when you want concatenation. A typical case is the streaming operatpr

```
std::ostream & operator<<(std::ostream & out, Myclass const & m)
 {
    ....// some stuff
    return out;// return the stream
 }
```

This allows concatenation:

```
std::cout<<myclass<<" concatenated with  "<<x;
```

What type does a function return?

In the case of a method of a class, you may return a reference to a member variable, to allow its modification

```
double & setX(){return this->x_;} // x_ variable member
...
x.setX()=10;// I change the member
```

or, occasionally, const references when you want to have the possibility of using potentially big variable members without copying them.

```
Matrix const & getM()const {return this->bigMatrix_;}
...
y=x.getM()(8,9);// get an element of the matrix
```

A typical case is the subscript operator, **operator**[](**int**);
We will see more examples in the lecture about classes.

inline

The **inline** attribute applied to a function declaration used to suggest the compiler to "inline" the function in the executable code, instead of inserting a jump to the function object.

In modern C++, **inline** simply means: "do not apply the one-definition rule" to this function (or variable). Definitions of **inline** functions are in header files. They will be recompiled in all translation units that use the functions, but the linker will not complain about multiple definitions, it will just use the first one.

Still, compilers, if optimization is activated, are free to "inline" an inline function if they deem it appropriate (you can avoid it with a special attribute, but we omit giving more detail).

An example of **inline**

```
inline double cube( double const & x)
{
   return x*x*x;
}
```

It should be placed in a header file.

Implicit **inline** functions

You do not need to write **inline** (it is implicit) in the case of

- Function templates (unless fully specialized or explicitly instantiated)
- Methods defined "in-class"
- `constexpr` functions
- Automatic parameter functions

They should be all defined in a header file.

You do not need to, but if you write **inline** in those cases it is not an error. Just redundant.

constexpr functions

With the **constexpr** specifier we are telling the compiler to try computing the function at compile time whenever possible.

```
// the argument is given by value
constexpr double cube(double x)
{ return x*x*x;}
```

By doing so the compiler may evaluate at compile time the expression

```
a=cube(3.0); // replaces it with a=9.0
```

A `constexpr` function has to comply with some restrictions, the main ones:

- It can call only other constexpr functions;
- Cannot be a virtual method;

.

constexpr functions

- Constexpr functions should be defined in a header file
- Constexpr implies inline: if an argument of the function is not a constant expression, a constexpr function behaves like a normal inline function;
- Constexpr functions make sense only for (A) small functions that you will call many times in your program (this is true also for inline functions); (B) when the value returned by the function is used in a context where you need a constant expression. Take for example the method size() of an array: if it had not been **constexpr** you could not use it as template value argument.

Otherwise, write a normal function.

Examples of implicit **inline** functions

```
// function template
template< class T>
 T fun(const T & x){ ..... }

 class MyClass{
 ...
 // in class definition
 double fun(double x){ return 5*x;}
 }
 // constexpr function
 constexpr double f(double x){return 1./x;}

 // Automatic parameter function (C++20)
 double fun(auto const& x, auto y){...}
```

Recursive functions

A function can call itself

```
double myPow( double const & x,unsigned n)
{
if(n==0) return 1.0;
else
return x*myPow(x,n-1);
}
```

We will see other examples with template functions.

A suggestion: recursive (standard) functions are elegant but non necessarily efficient. Often, non recursive implementations are better.

Default parameters

In the declaration of a function, the rightmost parameters may be given a default value.

```
vector<double> crossProd(vector<double>const &,
vector<double>const &, const int ndim=2);
...
a=crossProd(c,d);//it sets ndim=2
...
```

Default constructed arguments (another use of braces)

There are cases where one wants to pass as argument a default constructed argument, or, in general an object constructed on the fly. You can enjoy another goody of brace initialization.

```
double f(std::vector<double> const & v, double x, int k);
....

// passing an empty vector
 y = f({}, 4.,6);
// passing a vector of three doubles
 z = f({1.,2.,3.}, 5. ,6);
...
 u = f(v,3.,{});// Same as f(v,3,0)
```

The parameter must be either a value or a const- lvalue reference (or a r-value reference)

Static function variables

In the body of a function we can use the keyword **static** to declare a variable whose lifetime spans beyond that of the function call.
A static variable in a function is visible only inside the function, but its lifespan is global. They are useful when there are actions which should be carried out only the first time the function is called.

```
int funct(){
  static bool first=true;
  if(first){
      // Executed only the first time
        first=false;}
  else{
      //Executed from the second call onwards
      ...
      }
```

In this case the function is not stateless anymore!

Pointers to functions

```
double integrand(double x);
...
using Pf = double (*)(double)
//typedef double (*Pf)(double);
double simpson(double, double, Pf const f, unsigned n);
...
// passing function as a pointer
auto integral= simpson(0,3.1415, integrand,150);
// Using a pointer to function
Pf p_int=std::sin;
integral= simpson(0,3.1415, P_int,150);
...
```

The name of the function is interpreted as pointer to that function, you may however precede it by &: Pf p_int=&integrand;.

We will see in a while a safer and more general alternative to function pointers, the function wrapper of the STL.

An example

In the directory Horner you find an example that uses functions and function pointers to compare the efficiency of evaluating a polynomial at point x with two different rules:

- Classic rule $y = \sum_{i=0}^{n} a_i x^i$;
- The more efficient Horner's rule:

$$y = (\ldots(a_n x + a_{n-1})x + a_{n-2})x \ldots + a_0$$

Try with an high order polynomial (e.g. $n = 30$) and try to switch on/off compiler optimization acting on the local `Makefile.inc` file, and even activate parallelism using a standard algorithm!

Function names and function identifiers

A function is identified by

- Its name, fun in the previous examples. More precisely, its full qualified name, which includes the namespace, for instance std::transform.
- The number and type of its parameters;
- The presence of the **const** qualifier (for methods);
- The type of the enclosing class (for methods).

Two functions with different identifiers are eventually treated as different functions. It is the key for function overloading.

Note: the return type is NOT part of the function identifier!

A recall of function overloading

```
int fun(int i);
double fun(double const & z);
//double fun(double y);//ERROR! Ambiguous
 ....
 auto x = fun(1);// calls fun(int), x is a int
 auto y = fun(1.0);// calls fun(double const &), y is a double
```

The function that gives the best match of the arguments type is chosen. Beware of possible ambiguities, and implicit conversions!

(like a double and a (const) reference to a double, one implicit conversion

function templates

We will discuss templates in details a special lecture. Yet, we can anticipate function templates, since their use is quite intuitive.
A function template is not a function: it is the template of a function, where some types are parametrised and unknown when writing the template. Only at the moment of the instance, i.e. when the function is used, the parameter type can be resolved and the compiler can produce and compile the corresponding compiled code.
Function templates are useful when you want a function that may work for several argument types and you want to spare avoid repetition.

What is nice in function templates is that the parameter type may be deduced from the type of the arguments given to the function.

function templates

T will become like an alias for the deduced type of the parameter

we write a function f in this way

```
template<class T>
double f(T const & x){...}
```

The template function parameters are automatically deduced from the type of the corresponding argument when the template is instantiated. For instance,

then the compiler tries to deduce the type for T from how we call that f

```
auto y=f(5.0);
```

will instantiate a **double** f(**double const** &), that is the function obtained by setting T=**double**.

It is possible to fully specialize a template function to bound it to specific argument types. Here, an example of full specialization for **double**:

if we call this f, then this specialization gets the priority over the generic-template ones

```
template<>
double f(const std::complex<double> & x){...}
```

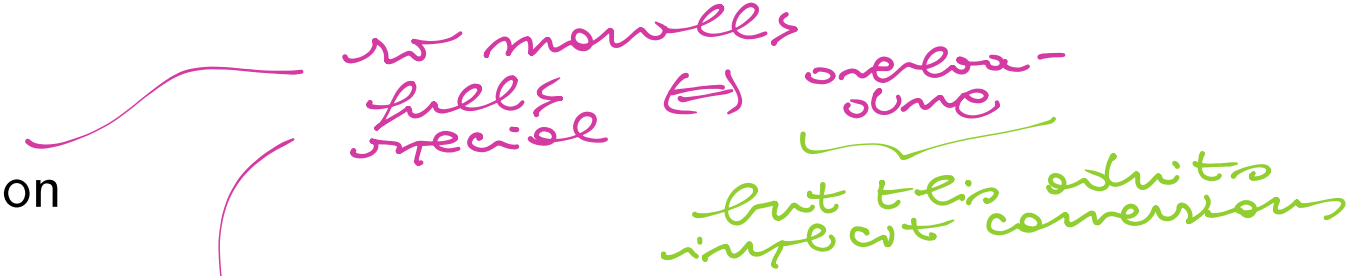
Now x=f(std::complex<**double**>{5.0,3.0}) will use the specialized version.

Overloading for special cases (it is not a specialization)

```
// Primary template
template <class T>
T dot(std::vector<T> const & a, std::vector<T> const & b)
{
        T res =0;
        for (std::size_t i= i; i<a.size();++i) res+=a[i]*b[i];
        return res;
}
        // overloading for complex
template<class T>
T dot(std::vector<std::complex<T>> const & a,
     std::vector<std::complex<T>> const & b)
{
        T res =0;
        for (std::size_t i= i; i<a.size();++i)
                res+=a[i].real()*b[i].real()
                    +a[i].imag()*b[i].imag()_;
        return res;
}
```

Now dot(x,y) calls the version for vector of complex numbers if a and b are complex.

A note



The full specialization

```
template<>
double f(const std::complex<double> & x){...}
```

can be replaced by an overloading, just omitting the **template** construct.

```
double f(const std::complex<double> & x){...}
```

The result is almost the same. The main difference is that with full specialization implicit conversion is not admitted, the type must be matched exactly.

Recursion with function templates

Template parameters can also be integral constants, this allows compile time recursion:

```
        // Primary template
template<unsigned N>
constexpr double myPow(cost double & x){ return x*myPow<N-1>(x);}
// Specialization for 0
template<>
constexpr double myPow<0>(const double& x){return 1.0;}
...
double y = myPow<5>(20.); // 5^20
```

now we have a normal function but with another "parameter" N (known at compile time) ⇒ more efficient

this call (thanks also to constexpr) will be solved at compile time

The following version is even better:

```
template<unsigned N>
constexpr double myPow(const double & x)
 if constexpr (N==0) return 1.0;
 else                return x*myPow<N-1>(x);
 }
```

needed for compile-time resolution

An important note

A thing to remember: implicit conversion does not apply to template function arguments

```
template<class T>
 double fun(T a, T b);
 ...
 int i=9;
 double x=20.;
 // ERROR compiler cannot resolve fun(int, double)
 double y=fun(i,x);
 // Ok I am explicitely converting int->double
 double y =fun(static_cast<double>(i),x)
 // Or I can explicitely tell which version I want
  double y =fun<double>(i,x)
```

Note: of course I may do

```
template<class T1, class T2>
double fun(T1 a, T2 b);
```

Automatic parameter functions

We have a simpler (yet less flexible) way to construct function taking arbitrary argument types. We can replace

```
template<typename T, typename M>
double fun(T const & x, M y);
```

with

```
double fun(auto const & x, auto y);
```

Simpler, isn't it? (why have they waited until C++20?).

Automatic parameter functions are just function templates in disguise, so their definition is in a header file.

Automatic return type deduction

We can let the compiler deduce the return type, both for ordinary functions and templates, even if the technique is more useful for templates (or automatic functions)

```
auto mul(auto const & x , auto const & y){
return x*y;
}
```

we have to add these (if we want) so auto gets just the basic type

The return type is deduced by the compiler. For instance,

```
double x=9.56;
int j =9;
...
auto x = mul(x,j);// x is a double
auto y = mul(1,j);// y is an int
// here z is a std::complex<double>
auto z = mul(x,std::complex<double>{1.,2.});
```

decltype(**auto**)

this extracts the whole type

Remember that **auto** strips qualifiers and references. So is you want a function return an automatically deduced reference (it makes sense only for methods) you have to do

```
auto & iAmReturningARef();
```

In special situations you may need decltype(**auto**) instead of **auto**: typically when you want an adaptor to forward the result of the call to another function.

a thing to control the interface of a function, like a wrapper

```
decltype(auto) adapter(const double & x)
{
    return aFunction(x,4,0);
}
```

return "what should be returned" from the function

Maybe I do not know what type aFunction returns. But here I do not care, I just grab the exact type of the returned value.

Functors (function object)

A function object or functor is a class object (often a struct) which overloads the call operator (`operator()`). It has a semantic very similar to that of a function:

```
struct Cube{
  double m=1.0;
  double operator()(double const & x) const {return m*x*x*x;}
};
...
Cube cube{3.}; // a function object, cube.m=3
auto y= cube(3.4)// calls operator()(3.4)
cube.m=9; // change cube.m
auto l= cube(3.4)// Again with a different m
auto z= Cube{}(8.0)//I create the functor on the fly
```

If the call operator returns a `bool` the function object is a predicate. If a call to the call operator does not change the data members of the object you should declare **operator**() `const` (as with any other method).

Why functors?

A characteritic of a functor is that it may have a state, so it can store additional information to be used to calculate the result

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

STL predefined function objects

Under the header <functional> you find a lot of predefined functors

```
vector<int> i={1,2,3,4,5};
vector<int> j;
transform (i.begin(),i.end(),   // 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.

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 expressions

We have a very powerful syntax to create short (and inlined) functions quickly: the lambda expressions (also called lambda functions or simply lamdas). They are similar to Matlab anonymous functions, like f = @(x) x^2. Let's look at a simple example.

```
...
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 **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 diffferent possible syntax (simplified version)

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

or

```
[ capture spec]( parameters)-> 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.

like a way to include more arguments into the expression

[]	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 object
[∗**this**]	Captures a copy of the enclosing class object

An example

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 (v.begin(), v.end(),   // range
     [x,y](int i) {return i > x && i < y;});// criterion
return *pos;
}
```

I have used the `find_if` algorithm which takes as third parameter a predicate and returns the iterator to the first element that satisfies it.

something that returns a bool

Other examples

```
  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.

// We also have generic lambdas
auto fun2=[](auto const & x){return x+pi;};

auto x = fun2(3.0L); // x=6.1415 is a long double
```

Generic Lambdas

You can allow lambda functions to derive the parameter type from the type of the arguments. Another example...addition of std::**string** means concatenation.

```
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"
```

Some notes:

1) avoid capturing all variables. It is better to specify just those you need.

2) **auto** is a nice feature. Don't abuse it. If the type you want is well defined, specifying it may help understanding your code.

Generalised capture

You can give alternative names to captured objects. Normally it is not needed, but it can be handy sometimes

```
double x=4.0;
...
// x captured by reference in r
// i is taken equal to x+1
auto f = [&r=x, i=x+1.]{
          r=r*4;
          return r*i;
          }
...
double res = f();
// Now x=16 and res=80.!
```

Multiple returns

Lambdas can have more then one return (like an ordinary function), but they must return the same type (no type conversion allowed on the returned objects).

```
 double f(){...} // ordinary fun
 auto g = [=] ()
{
 while( something() ) {
   if( expr ) {
       return f() * 42.;
    }
 }
 return 0.0;// & multiple returns
}           //   (types must be the same)
```

Lambdas as adapters (binders)

Lambda expressions may be conveniently used as binders: adapters created by binding to a fixed value some arguments of a function.

```
// A function with 3 arguments
double foo(double x, double y, int n);
// A function for numerical integration
template<class F>
Simpson(F f, double a, double b)...
// I want to do the integral of foo with respect
// to the first argument, the others given
auto f=[](double x){return foo(x,3.14,2);}
double r = Simpson(f,0.,1.);
```

You have in C++ also specific tools for "binding", but lambdas are of simpler usage.

lambdas and constexpr

constexpr variables are imported into the scope of a lambda expression with no need of capture:

```
constexpr double pi=3.1415926535897;
auto shiftsin=[](double const & x){return std::sin(x+pi/4.);};
...
double y = shiftsin(3.2);// y=sin(3.2+pi/4.)
```

This is nice feature of constant expressions.

An example of use of [this]

With **this** we get the **this** pointer to the calling object!

```
class Foo{
public:..
void compute() const;
private:
double x_=1.0;
vector<double> v_};
... // definition
void foo::compute(){
    auto prod=[this](double a){x_*=a;};
    std::for_each(v_.begin(),v_.end(),prod);}
...
Foo myFoo;
...
auto res=myFoo.compute();
```

Here compute() uses the lambda prod that changes the member x_.
To be more explicit you can write **this**->x_*=a;.

An example of use of [*this]

With *this we get a copy of the calling object! Here generic capture is handy:

```
class Foo{
public:..
void compute() const;
private:
double x_=1.0;
vector<double> v_};
... // definition
void foo::compute(){
    auto prod=[foo=*this](double a){return foo.x*a};
    std::for_each(v_.begin(),v_.end(),prod);}
...
Foo myFoo;
...
auto res=myFoo.compute();
```

Now, in the execution of the last statement, foo inside the lambda is a copy of myFoo.

Functions (or lambdas) returning lambdas

Here a template lambda function that returns a lambda function computing the approximation of the N-th derivative of a function by forward finite differences (see Derivatives/Derivatives.hpp for other solutions);

```
template <std::size_t N>
auto numDeriv=[](auto f, double h){
  if constexp (N==0u)
      return [f] (auto x){return f(x);}
  else{
      auto prev=numDeriv<N-1u>(f,h);
      return [=](auto x){
            return (prev(x+h)-prev(h))/h;};
  };
};
..
auto f=[](double x){x*std::sin(x);}
auto ddf=numDeriv<2>(f,0.001);
double x = ddf(3.0); //numerical II derivative at x=3.0
```

in both cases we return a function

Callable objects

The term callable object (or just callable) indicates any object f where the syntax f(args..) is allowed and may return a value. It can be

- An ordinary function;
- A pointer to function;
- A lambda expression;
- A function object (functor).

A callable object may be called in the usual way of by using the std::invoke utility present in the header <functional>. I am not giving more details on invoke, you may find them here.

Function 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 any callable object as first class objects.

```
int func(int, int); // a function
struct F2{ // a function object
int operator()(int, int) const;};
...
// a vector of functions
std::vector<std::function<int(int,int)>> tasks;
tasks.push_back(func);// wraps a function
tasks.push_back(F2{});// wraps a functor
tasks.push_back([](int x,int y){return x*y;});// a lambda
for (auto i : tasks) cout<<i(3,4)<<endl;
```

It prints the result of `func(3,4)`, `F2(3,4)` and 12 (3×4).

Function wrappers

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

See the examples in , RKFSolver, FixedPointSolver and NewtonSolver. The first implements a RK45 adaptive algorithm for integration of ODEs and systems of ODEs. The other two codes deal with the solution of non-linear systems.

Function wrappers introduce a little overhead, since the callable object is stored internally as a pointer, but they are extremely flexible, and often the overhead is negligible.

Stateless function objects

A function object is said to be stateless if its state does not depend from previous calls. Consequently, the object returns the same values when called twice with the same arguments.
For instance,

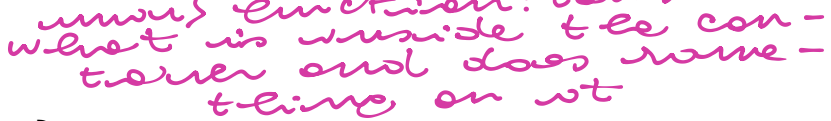
```
struct Count7{
    void operator ()(int i){if (i==7) ++count7;}
    int count7=0;
}
```

is not stateless (i.e it is stateful): it modifies the state of the Count7 object.

Be careful when passing stateful callable objects to algorithms of the standard library. Some care must be taken to avoid having bad surprises!

Stateless functors and std containers and algorithms

The concept of stateless functor is important because the standard library implicitly assumes the functors are stateless. Let's consider this example




```
int main(){
    std::vector<int> v={1,7,7,8,7,9};
    Count7 c7; // a counter of 7
    std::for_each(v.begin(),v.end(),c7); // apply c7 to v
    std::cout<<" The number of sevens is "<<c7.count7;
}
```

The answer is: `The number of sevens is 0`!!!! Why!!!

Analysis of the problem

Looking at cppreference.com I see that the declaration of std::for_each is

```
template< class InputIt, class UnaryFunction >
UnaryFunction for_each(InputIt first,InputIt last,UnaryFunction f);
```

It is a template function that takes as first two arguments two iterators and as third argument a UnaryFunction, that is a callable object that takes just one argument. So it looks all right but... the callable object is taken by value! It means that for_each makes an internal copy of c7, an uses it to operate on the vector. My c7 object is unchanged!! So the counter c7.count7 is still zero.

Is there a solution? Yes, actually two solutions, the first specific, the second more general.

The solution

A first solution, valid for for_each, is to note that it returns the callable object passed as argument!. So, you can do

```
c7 = std::for_each(v.begin(),v(end),c7);
```

to overwrite c7 with the one used inside for_each.

A second solution is using the magic of std::ref(), and do

```
std::for_each(v.begin(),v(end),std::ref(c7));
```

Now, c7 is passed by reference, so any operation inside for_each is reflected in my c7 object.

ref() is a small utility that returns a reference wrapper, a first-class object holding a reference to the argument. A little trick that may be used in all similar situations. The trick is applicable only on template function parameters.

For the nerds

How does std::ref work? Well, here my "poor man" version,

```
template <class T>
T& ref(T& x)
{
    return x;
}
```

Have you understood how it works? Think about it.

Another solution (actually my preferred one)

Use a lambda expression (they are explained later!)

```
Count7 c7;
std::for_each(v.begin(),v(end),[&c7](int i){c7(i);} );
```

Here c7 is captured as a reference by the lambda. The lambda will be copied, but the copy will just carry on the reference. So inside for_each I am using an alias of the c7 object in the calling code.

So, stateful functors can be perfectly fine, but you have to take some care. In particular, watch out for possible unwanted copies!

Side-effect

Another important concept is that of side-effect. A function has a side-effect if it modifies an object outside the scope of the function. So stateful callable objects have side-effect.

In this case, you need to be careful in a parallel implementation!

Let's consider our Cont7 example. In a distributed memory paradigm like MPI, each process has its own copy of a c7, so it counts the 7 in the portion of the vector assigned to it. To have the total number you have to sum them up!

If you use a shared memory paradigm instead, like threads or openMP, you must ensure that different threads do not access the counter at the same time! You have to make the counter update atomic!

Function Expression Parsers

Functions in C++ must be defined at compile time. They can be pre-compiled, put into a library file and even *loaded dynamically* (as we will see in a next lecture), Yet, you have still to compile them!.

Sometimes however it can be useful to be able to specify simple functions run-time, maybe reading them from a file. In other words, to interpret a mathematical expression, instead of compiling it.

This, of course introduces some overhead (after all we are using a compiled language for efficiency!). Yet, in several cases we can afford the price for the benefit of a greater flexibility!.

Possible Parsers

We need to use some external tools that parses a mathematical expression, that can contain variables, and evaluate it for a given value of the variables.
Possible alternatives (not exhaustive)

- Interface the code with an interpreter, for instance interfacing with Octave;
- Use a specialized parser. Possible alternatives: boost::spirit, μParser and μParserX, a more advanced (but slower) version of μParser.

μparser

In this course we will see μParser and μParserX.

A copy of the software is available in the directories Extras/muparser and Extras/muparserX. To compile and install it (under the Examples/lib and Examples/include directories) just launch the script indicated in the `README.md` file.

Those directories are in fact submodules that link to a fork of the original code, adapted for the course. That's one of the reasons of the need of `--recursive` when you clone the repository of Examples and Exercises.

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