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

Luca Formaggia

MOX
Dipartimento di Matematica
Politecnico di Milano

A.A. 2023/2024

Functions in C++

In C++ functions may be free functions or member functions (also called methods) of a class. A further subdivision can be made between functions and template functions. Methods are function in the class scope, and we will treat them in detalie the lecture about classes.

For a free function the usual declaration syntax is

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

or

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

or (automatic return function)

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

In the third possibility, the return type is deduced by the compiler, we will give more details later on.
The first two forms are practically equivalent. Choose the one you like most.

Some examples of function definition

Those functions are equivalent

```
#include <cmath>

double sinlog1(double x) { return std::sin(x)*std::log(x);};

auto sinlog2(double x)->double {return std::sin(x)*std::log(x);};

auto sinlog3(double x){ return std::sin(x)*std::log(x);};
```

In the third case the return type is a double since it is the result of the multiplication of two doubles.

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 $f : U \rightarrow V$.

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 free function pure declaration does not contain the body of the function, and is usually placed 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, apart from inline and contexpr functions, which should be defined in a header file.

```
double f(double const & x)
{
  double y;
  //... do someting
  return y;
}
```

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 name 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!

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 a function call via the arguments is using pointers. This is the only technique available in C. In C++ references are preferred.

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 or as const reference (es: **const** Vector &). Use pass-by-value only for small data (**int**,**double**,...).
- 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!

Reference binding

References are aliases to existing objects and we say that they bind to the object. When used as function parameters

- (non const) lvalue references bind only (and preferably in the case of overload) to lvalues (i.e non const variables)
- const lvalue references may bind to both lvalues (const and non-const) and rvalues (temporary objects and literals). In other world, they may bind to everything.

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

What type can 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

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

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

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

What type can a function return?

In some rare cases a function returns a lvalue reference to an object passed by lvalue reference. Normally, because 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 can 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 a 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.

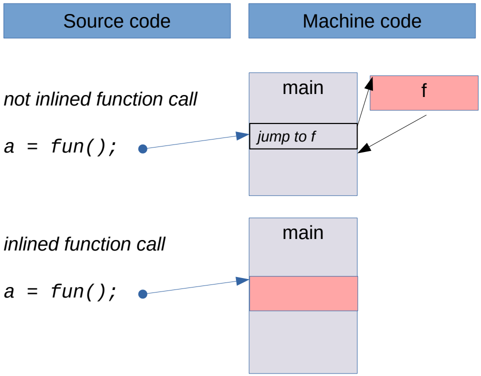
Still, compilers, if optimization is activated, are free to "inline" an inline function if they deem it appropriate (you can avoid it with a special attribute, but we omit giving more detail).

An example of **inline** function

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

It should be placed in a header file.

What happens when a function is actually inlined



Implicit **inline** functions

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

- Methods defined "in-class" (i.e. directly into the class declaration);
- `constexpr` functions

They should be all 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

The **constexpr** specifier tells the compiler to try computing the return value at compile time whenever possible.

```
constexpr double cube(double x) { return x*x*x;}
```

The compiler may evaluate at compile time the expression

```
        a=cube(3.0); // replaced 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.

.

Some guideline to 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 for: (a) small functions that you will call many times in your program in a loop; (c) 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 c++ array: if it had not been **constexpr** you could not use it as template value argument. Otherwise, write a normal function.

Recursive functions

A function can call itself

```
auto 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 note: recursive functions are elegant but non necessarily efficient. Often, non recursive implementations are more effective.

A more complex example of recursion

```
using Fun = std::function<double(double)>; // a function wrapper!
inline auto
derivative(unsigned N, double const &x, double const &h, Fun const &f)
{
  if (N == 0)
    return f(x);
  else
    {
      auto centerpoint = N % 2 == 0 ? x : x + h;
      return (derivative(N - 1, centerpoint, h, f) -
              derivative(N - 1, centerpoint - h, h, f)) /h;
    }
}
... //possible usage
auto y = derivative(3,5.0,1.e-6,std::sin);
```

Note the switch between forward and backward differences.

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\newline
...
```

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, as we will see, a rvalue 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. Static variables in a function are visible only inside the function, but their lifespan is global. I find them useful when there are actions which should be carried out only the first time a 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;.

In C++ we have a safer and more general alternative to function pointers: the function wrapper.

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:

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

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 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, the basic form

The template parameter may express a type

```
template <typename T>// or <class T>
auto add (T const & a, T const & b){return a+b;};
...
auto y= add(3,4)// T=int y =7
auto s= add(30.,45.)// T=double s=75.
auto k= add(3,5.)// Error implicit conversion does not apply
```

or an integral constant expression

```
template<int I>
 int mul (int x){return I*x;};
 ...
 auto j=mul<3>(7);// j=21.
```

In the first case the template argument is automatically deduced.

Different form of function templates

The classic form, template parameters are explicitly indicates

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

The automatic form, you use the `auto` specifier (simpler)

```
double f(auto const & x,auto & y){...}
```

You can also mix:

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

Different form of function templates

You normally indicate the parameter explicitly when you want to impose some constraints or you need the type inside the body function. Let's see an example

```
template<class T >
T sum1 (T const & x, T const & y);
auto sum2 (auto const & x, auto const & y);
template<class T >
T foo (T const & x)
{
 std::vector<T> v;// I need a vector of T's
}
```

In sum1 I am forcing the function arguments and return value be of the same type. In sum2 instead the argument type may differ and the returned value type is deduced by the compiler.

Instance and specialization

The actual (template) function is generated at the moment of its instance:

```
template<typename T>
double f (T const & x);
...
double y=f(5.0);// f<double> is generated
std::vector<int>v;
double z=f(v); f<std::vector<int>> is generated
```

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

```
    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.

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
```

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);}
```

Note the use of **if constexpr** to resolve the branch at compile time.

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);
```

or use an automatic function.

Automatic return type with automatic function

The combination provides an extremely generic function

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

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

```
using namespace std::literals;//for complex and string literals
double x=9.56;
int j =9;
 ...
auto x = add(x,j);// x is a double
auto y = add(1,j);// y is an int

auto z = add(x,5.0+3.0i);// using complex literals
// here z is a std::complex<double>
auto s = add("Hello␣"s,"World"s);
// s is "Hello World"
```

decltype(**auto**)

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.

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

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.

Constrained templates

A template type can in principle be anything. But we can limit the set of usable types employing concepts. We will discuss concepts in details at a later stage. Some use of concepts is already present in the examples of the course.

Let's look here to some example using concepts already provided by the standard library

```
#include<concepts>
template <std::integral T>
T fun (T x); // T can be only an integral type
auto foo (std::floating_point auto const & x)
// foo accepts ony floating point types
```

It can be applied also to generalised lambda expressions, which we will see later.

Constrained induced overload

Overloading applies also to non overlapping contraints.

```
#include<concepts>
auto foo (std::floating_point auto const & x)
{ .. //version for floating points}
auto foo (std::integral auto const & x)
{ .. //version for integral types}
...

auto x = foo(3); // second version called
auto y = foo(3.0);// first version called
```

In the lecture about template (meta)programming we will see also how to generate our own concepts.

Functors (function objects)

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
```

When the call operator returns a **bool** the functor is called predicate. If the operator does not change the struct data members you should declare it **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 few predefined functors

```
#include <alorithm>
vector<int> i={1,2,3,4,5};
vector<int> j;
j.resize(i.size());
std::transform(
      i.begin(),i.end(),   // source
      j.begin(),           // destination
      negate<int>());      // operation
```

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

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 functions on the fly: the lambda expressions (also called lambda functions or simply lamdas). They are similar to Matlab anonymous functions like f = @(x) x^2. Let's start with 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.

Capture specification

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

We will detail the use of **this** in the lecture on classes.

An example

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

```
#include<algorithm>
template <class T>
T f(vector<T> const &v,T x, T y){
auto pos = std::find_if (v.begin(), v.end(),  // range
     [&x,&y](T i) {return i > x && i < y;} );// criterion
std::assert(pos!=v.end());//fails if not found
return *pos;
}
```

I have used the find_if() standard algorithm, which takes a predicate as third parameter and returns the iterator to the first element that satisfies it. I've created the predicate on the fly with a lambda.

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.
```

Generic Lambdas

You can allow lambda functions to derive the parameter type from the type of the arguments, as in automatic functions. An example.

```
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) It is better to capture large variable by reference.

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.

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 <unsigned N>
inline auto numDeriv = [](auto const &f, double const &h) {
  if constexpr (N == 0u)
    return [f](auto x) { return f(x); };
  else
    {
      return [f, h](auto x) {
        auto const center = (N % 2 == 0) ? x : x + h;
        auto prev= numDeriv<N - 1u>(f, h);
        return (prev(center) - prev(center - 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
```

Explanation

That example is rather complex. Let's have a look.

numDeriv<N> is a variable template that will be set to a lambda expression that takes a callable object `f` and a spacing `h`, and returns a lambda expression computing a certain derivative of the function at a given point.

The template parameter is here an unsigned integer that it will be equal to the order of numerical derivative I want. The lambda expression that numDeriv<N> contains is itself built by recursive calls to numDeriv<M>, with `M=N-1,...,0`.

It looks like a nice piece of functional programming! Try to understand how it works!

Callable objects

The term callable object (or just a callable) indicates any object `f` where the syntax `f(args..)` is allowed and may return a value. Examples of callables are

- 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.

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.

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.

A solution

Use a lambda expression

```
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 sometimes, 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!