Advanced Methods for Scientific Computing (AMSC)
Lecture title: Functions, functors and lambdas

Luca Formaggia

MOX
Dipartimento di Matematica
Politecnico di Milano

A.Y. 2024/2025

Functions and function objects in C++

In modern C++ the concept of function has been extended to function object (or functors). A function object can be

- standard (also called "free") or template functions;
- lambda expressions;
- class-type objects where you define a special operator called the "call operator";

Then you have "function members" of a class, also called "methods", which are function in the scope of the class that can access the class data members.

Free functions

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 case, 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 (the second form is getting more and more popular nowadays!)

A function may return nothing, in which case ReturnType is **void**.

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);};

void printThat(std::string that){std::cout<<that;}
```

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

When "free" functions?

A "free" function should 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 l-value reference.

Consequently, a (free) function is usually (there are exceptions) is stateless, which implies that two different calls of a function with the same arguments produce the same result.

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

The [[nodiscard]] attribute

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 when 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 (forward) declaration and definition

A free function forward declaration does not contain the body of the function, and is 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). It's role is to declare the function in all translation units that include the header file.

The definition of a free function is usually contained in a source file, apart from inline and constexpr functions, which should be defined in a header file.

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

It's role is to provide the actual code!

Function identifiers

A function is uniquely identified by

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

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 due to implicit conversions!

Function overloading is one of the key features of C++!

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" . These slides are just for 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 l-value 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.
- Using a (non-const) r-value reference we allow as argument only an object that can be moved.

Passing a value

If you pass an argument "by value", the argument value is copied in the corresponding parameter (unless you move it, but let's forget about it as the moment) The main thing is that the parameteris a local variable of the function whose change of value does not reflect on the corresponding argument.

When you pass a value, you can still declare the parameter **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 argument 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

- Prefer passing by reference when dealing with large objects. You save memory.Use pass-by-value only for small data (**int**,**double**,...);
- If the parameter is not changed by the function (it is an "input-only parameter"), either pass it by (const) value 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 l-value reference (or by an r-value reference)
- You can pass polymorphic objects only by reference or with pointers. Otherwise, polymorphism is lost!

What type can a function return?

A non-void, function normally returns a value. A method may also return lvalue references (const or non-const) to data member of the class. 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().

What type can a function return?

In some 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 operator

```
std::ostream & operator<<(std::ostream & out, Myclass const & m)
 {
    out<<"value␣of␣x␣"<<m.x;// some output
    return out;// return the stream
 }
```

This allows concatenation:

```
std::cout<<myclass<<"␣concatenated␣with␣this␣string";
```

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 and constexpr functions

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 put in a header file. They will be recompiled in all translation units that include the file, but the linker will not complain about multiple definitions, it will just use the first one.

It is still true that compilers are free to "inline" the function if they deem it appropriate (you can avoid it with a special attribute, see here).

constexpr functions

A **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 is implicitly inline and should be defined in a header file. This is normally used only for function that you expect to be used often with constant expression argument. Or when you need to have a constant expression (see lecture on templates)

Recursive functions

A function can call itself

```
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 differencing. 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.

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:

```
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,{},{});// Same as u=f(v,0.,0)
```

The default constructor is called. For obvious reasons, the corresponding parameter cannot be an non-const l-value reference.

A note Using braces on default initialization of a double initializes it to zero!

```
double a; // value undefined
double b{};// value is zero
```

Statefull functions: 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!. In C++ for situations like this one, prefer creating a class-type object and avoid statefull free functions!

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.

Pointers to a set of overloaded functions

In C++ you may have function overloading.

In this case, handling pointers to function may become trickier. We do not go into details, if you are interested you may have a look here, or, for more technical details, here.

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 the corresponding template function and compile the corresponding code.
Function templates are useful when you want a function that may work with argument of different types and 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 deduced automatically, in the second case we must indicate it.

Different forms of function templates

In the "traditional" form parameters are explicitly indicated:

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

In the automatic form, you use the `auto` specifier:

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

You can also mix:

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

Different forms 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.

Constrained function templates (C++20)

With the use of concepts you can constrain the possible argument type

```
#include <concepts>
auto fun (std::integral x); // version for integral types
auto fun(std::floating_point x);// version for floating point
```

Possible usage

```
int i=10;
auto res =fun (i); // integral vesion used
auto res2 = fun(3.0);// floating point version used
auto z = fun("Hi");//COMPILATION ERROR!
```

Concepts are a new feature that allow safer and sometimes simpler template programming.

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.

decltype(**auto**) return type

auto strips qualifiers and references. So is you want a function return an automatically deduced reference (if can be used in a method that operates on a data member) 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 of which you have not full control.

```
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.

Functors (function objects)

A function object or functor is a class-type 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 call operator does not change the struct data members you should declare it **const** (as with any other method).

Why functors?

A characteristic 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
```

Moreover, a functor is an object of class type. A free function is instead represented just by a pointer.
Functors are use extensively by the c++ standard library.

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

Lambda expressions (also called simply lambdas) creates a callable object on the fly, like 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**.

The basic 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>
double f(vector<double> const &v,double x, double y){
auto pos = std::find_if (v.begin(), v.end(),   // range
      [&x,&y](double 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.

Generalized 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 if the returned type is deduced, they must return the same type.

```
 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!

Function wrappers

And now, the catch all function wrapper. The class template `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).

An example

In the directory Horner you find an example that uses a function wrapper to specify different policies for evaluating a polynomial at a point x:

- 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!

Another example of use of 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.

Note: Function wrappers introduce some overhead, since the callable object is called indirectly through a pointer: flexibility comes at a price.

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.

Side-effects

An important concept is that of side-effect. A callable object has a side-effect if it modifies an object outside the scope of the function in an indirect way. i.e. not through the returned values.
In this case, you need to be careful in a parallel implementation!

Let's consider the following Cont7 functor that counts how many times it has been called with a 7:

```
struct Count7{
    void operator ()(int i){if (i==7) ++count7;}
    int count7=0;
};
...
Count7 c7;
for (auto i:x) c7(i);
std::cout<<" The number of 7 in x are "<<c7.count7;
```

Each call may modify the internal state of the Count7 object. So it has a side-effect.

Side-effects and parallel programming

In a distributed memory paradigm like MPI, each process has its own copy of a Count7 object, which counts the 7s 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:

```
std::vector<int> x;
// fill x with values
Count7 c7; // I want to count the 7
#pragma omp parallel for //start parallel loop
for (auto i:x) c7(i);
std::cout<<" The number of 7 is "<<c7.count7;
```

In a multithread environment, the answer is almost surely wrong and different at each run of the code. You have to make the counter update atomic!

Pure functions

A function or functor is said to be a pure function if

- It has no side-effects;
- It returns values only "by value" (i.e. through the return statement) and not "by reference" or through pointers;
- It returns the same value when called with the same arguments.

Function/functors passed to standard algorithms are normally requested to be pure functions.

Examples

```
double byTwo(double x){return 2.x;}// pure function
// another pure function
double middleElement(cons std::vector<double>& v)
{
return v[v.size()/2];
}
int f(int & x) //IMPURE function
{
    return (++x)/2;
}
void out(){ // IMPURE function
  std::cout << "Hello, world!" << std::endl;
}
```