Advanced Methods for Scientific Computing (AMSC)
Lecture title: Templates

Luca Formaggia

MOX
Dipartimento di Matematica
Politecnico di Milano

A.Y. 2024/2025

Templates

Templates are the tools with which C++ implements generic programming, which means to have classes and functions that may operate with different types. We have already seen examples of function template here we cover class tempalate.

Templates are introduced by the keyword **template** followed by <Params> where Params is a comma separated list of parameters, which may be either of class-type, introduced by the keyword **typename** or **class**, or integral values (integer, enumerators, char, pointers) indicated with the corresponding type.
Correspondingly, the arguments are types or integral constant expressions.

An example of class template

```
template<class T, int N>
class Foo{
public:
// a method can be itself a template
  template <typename C>
  double fun(C const &, T const &);
  // simplified version: double fun (auto const &, T const &);
private:
  std::array<double,N> My_data;
  T My_t;
}
```

Here, the first parameter is a type, the second is an integer value. Methods (*including operators and constructors*) can be templates as well (this is true also for ordinary classes), and follow the rules of function templates.

Template methods

A class (template or not), may have function template members.
They basically behave like function templates:

```
class FEMSpaceP1{
public:
  ...
  // this is a method template: it is equivalent to
  // template<class M>
  //auto compute_stiff(M const & mu);
  auto compute_stiff(auto const & mu);
   };
   ...
FEMSpaceP1 fem;
Matrix M;
fem.compute_stiff(M);// compute_stiff<Matrix> is instantiated
fem.compute_stiff(0.5);// compute_stiff<double> is instantiated
```

Method templates cannot be virtual!!.

Template classes are not classes...

A class template is indeed just a template of a class. The actual template class is generated when it is instantiated, that is when the template arguments are given.

Therefore, `std::vector<double>` is technically a different class than `std::vector<int>`.

Non-static methods of a class templates are instantiated only when used. This is not a bug, it's a fundamental feature! : your class will work also for template arguments that make no sense for some of its methods.

A drawback: as the actual compilation of a method of a class template is performed only if the method is actually used some compilation errors may be found only then.

Implicit class template deduction

Also the parameters of a class template can be automatically deduced, through the constructor and deduction guide. I will not give the details, only a few examples

```
template <class T1, class T2>
class Foo{
public:
// an explicit constructor with template dependend parameters
Foo (T1 a, T2 b);
...
};
...

Foo foo{3,4.0}// -> generates Foo<int,double>
```

Indeed, `std::vector v={3,4,5};` generates a `std::vector<int>`.

Explicit class template deduction: Deduction guides

You may even tell the compiler how to deduce class template parameters. This can be useful when you do not have a constructor that provides the deduction.

```
template <class A>
struct Aggr
{
  double x=0;
  A a;
};
template <class A> Aggr(double, char *) -> Aggr<std::string>;
....
Aggr a{3.0,"Hello"}; // a is an Aggr<std::string>
                     // and not a Aggr<char *>
```

Constant values as template parameters

A template parameter may also be an integral value.

```
template <std::size_t N=3> class Point // you may have defaults
{
 public:
    Point(std::array<double, N> p) : data{p} {};
    Point() = default;

 private:
    std::array<double, N> data;
};

...
Point<3> p; // a point in 3D
Point z;// another point in 3D (dafault)
Point q=std::array{3.,5.};// a Point<2>!
```

Note the use of `size_t`. If I use **int** the last statement produces an error, since the second template argument of an array is `size_t`, not **int**, and conversions are not allowed when resolving template parameters.

Class templates, a more complex example

```
template<typename T> class Vcr {
  int length;              // number of entries in vector
  std::vector<T> vr;    // entries of the vector
public:
  Vcr(int, T*);            // constructor
  int size();
  inline T& operator[](int i);
  Vcr& operator=(const Vcr&);    // overload =
  Vcr& operator+=(const Vcr&);  // v += v2
  Vcr& operator-=(const Vcr&);  // v -= v2
  auto maxnorm () const;              // maximum norm
  auto twonorm() const;               // L-2 norm
  template<class S>
  auto dot(const Vcr<S>&);// template method
};
```

Out of class method definition

Methods can be defined "out of class" also for class templates. But they are still in a header file. Odr rule does not apply to instances for template methods.

```
template<typename T>
class Vcr
{
...
double maxnorm () const;// pure declaration
...
}
// definition
template<typename T>
auto Vcr<T>::maxnorm() const
{
 Vcr a; // see note below
....
}
```

Within the *scope* of the class and that of its members one can avoid to use `<T>` to indicate `Vcr<T>`. (One may still be explicit, it is not an error).

Specialization of class templates

We may specialize methods singularly, like we have seen for the functions (with the same rules!), or a whole class. In case of the specialization of a class, all desired methods of the specialized version should be redefined.

Primary templates must be visible when defining and instantiating a specialization.

Overloading (specialization) of a template method

We may decide to overload just a method

```
template <class T>
double
Vcr<complex<T> >::maxnorm(){
  double tmp(0);
  for(auto i=vr.begin();i<vr.end();++i)
                tmp=std::max(tmp,std::norm(*i));
}
```

Now on a `Vcr<complex<T> >` the method `maxnorm()` calls the "specialized" version.

Note: the template function `std::norm<T>(T x)` is provided by the standard library and computes the modulo of a complex number if T is a `std::complex`.

Partial specialization of a (whole) class

```
template<class T>
class Vcr< complex<T> > {
  int length;
  std::vector<complex<T> >vr;
public:
  ...
  int size() { return length; }
  complex<T>& operator[](int i) const{
          return vr[i]; }
       ...
  auto maxnorm () const;
  auto twonorm() const;
...
};
```

We need to redefine all methods. Even those of the primary template that would be fine: there is no "inheritance" from the primary template class.

Full class template specialization

```
template<>
class Vcr<bool> {
  int length;
  byte* vr;
public:
   ...
....
};
```

Now `Vcr<bool> a;` will instantiate the fully specialized class. Again I have to re-define all the methods of the specialized version I plan to use, even those identical to the primary: specialization is not inheritance.

Definition of methods of a specialized class template

Partial specialization:

```
template<class T>
double Vcr<complex<T>>::maxnorm() const {
  double nm(0);
  for (int i = 1; i < length; i++)
  nm = std::max(nm,std::norm(vr[i]));
  return nm;
}
```

Full specialization (for a class of 2D point):

```
template<>
inline double Vcr<Point2D>::maxnorm() const{
  double nm(0);
  for (int i = 1; i < length; i++)
  nm = std::max(nm,std::abs(vr[i].x()), std::abs(vr[i].y()));
  return nm;}
```

Note the **inline**. You need it if you keep the full specialization in the header file.

Some rules

- The declaration of the primary (i.e. un-specialized) class or function template must be visible when you define a specialization. A pure/forward declaration is enough:

```
template <class T> MyClass<T>; \\ forward decl.
template<> MyClass<double>{
...}; // full specialization.
```

- When specializing a class template we need to provide the definition of all specialized methods (at least the ones we are going to use);
- Specialization of a template must correspond to a possible instance of the primary template;
- Also at the moment of the instance of a specialized template, the declaration of the primary template must be visible.

Template template parameter

A template parameter may be ... a template!.

```
template<template <typename> class C>
class MyClass{
private:
C<double> a;
}
```

The parameter `C` is a template that takes a single template argument. I can give defaults

```
template<template T,
template <typename> class C=complex<T> >
class AnotherClass{
private:
C  a;
}
```

Now AnotherClass<**float**> a; will store a complex<**float**>.

Template alias

Templates can be long and tedious to write. We have a nice shortcut that can be useful in many situations

```
template <typename T>
using Dictionary = std::map< std::string , T >;
...
Dictionary<int> bcIdentifier;
bcIdentifier[ "Dirichlet" ] = 1;
bcIdentifier[ "Neumann" ]   = 2;
...
```

Here `Dictionary<T>` is an alias of `std::map< std::string, T >`.

Other examples of type alias

```
using Real=double;

template<typename T, int N> class Point; // a template for points
// A template alias:
template<typename T>
using Point2D = Point<T,2>;
template<class T> using ptr = T*;  // ptr<T> is  an alias for poi

 //usage
Point2D<double> myPoint; // Defining a Point<double,2>
Real a; // Defining a double
ptr<int> xPtr; // defining a pointer to int
std::vector< Point2D<float>> vp;// a vector of points
```

Templates for specifying policies

Templates may be a way to specify a policy: a part of the implementation of a class. An example is the comparison operator of a std::set.

Let's make a simple example. We want to compare two strings for equality and have the possibility of managing case sensitive and case insensitive comparisons.

Two policy classes used in a function template

```
class Ncomp {// normal compare
public:  // I rely on the existing operator
 bool eq(char const & a, char const & b)const
  { return a==b;}
};
class Nocase { // compare by ignoring case
public:
  bool eq(char const  & a, char const & b) const
  { return std::toupper(a)==std::toupper(b);}
};

//A functor expressing comparison
template <class P>
struct stringEquality{
bool  operator(const string & a, const string & b) const{
  if(a.size()==b.size()){
    for(unsigned int  i=0;i<a.size();++i)
      if  (!comp.eq(a[i],b[i])) return false;
    return true;}
  else return false;
}
private:
 P comp;
};
```

Two policy classes used in a function template

In this case you may also use static methods

```
class Ncomp {// normal compare
public:  // I rely on the existing operator
 static  bool eq(char const & a, char const & b)const
  { return a==b;}
};
class Nocase { // compare by ignoring case
public:
  static bool eq(char const  & a, char const & b) const
  { return std::toupper(a)==std::toupper(b);}
};

//A functor expressing comparison
template <class P>
struct stringEquality{
bool  operator(const string & a, const string & b) const{
  if(a.size()==b.size()){
    for(unsigned int i=0;i<a.size();++i)
      if (!P::eq(a[i],b[i])) return false;
    return true;}
  else return false;
}
}};
```

A possible usage

```
stringEquality<Ncomp> normal;
stringEquality<Nocase> nocase;
auto x= normal("Luca","luca"); // x=false
auto y= nocase("Luca","luca"); // y=true
```

A more complete and working example

See the example in Templates/Compare/main.cpp

The use of this in templates

```
void myfun(){...}

...
template <typename T> class Base{
public:
void myfun(); // very bad idea!
};
template <typename T>
class Derived : Base<T> {
public:
void foo() { myfun();...} // WHICH myfun()???
```

In a class template derived names are resolved only when a template class is instantiated (only then the compiler know the actual template argument).
Other names are resolved immediately. So in this case the free function `myfun()` is used!

Solution

Use **this**:

```
void myfun();
...
template <typename T> class Base{
public:
void myfun();};
template <typename T>
class Derived : Base<T> {
public:
void foo() { this->myfun();...}
```

or use the qualified name:

```
template <typename T>
class Derived : Base<T> {
public:
void foo() { Base<T>::myfun();...}
```

In this case the compiler understand that `myfun()` is a dependent name and will resolve it only at the instance of the template class.

Variadic templates

We can specify that a function or a class template takes an arbitrary number of template parameters (parameter pack). Very handy when you want specialization with different number of template parameters.

```
template<typename... Tlist>
class manyParameters{ // primary template
void SampleFunction(Tlist&... params){...};
// now the specializations
```

It is a complex feature, the full explanation goes beyond the scope of these lectures, yet it may increase the capability of the language greatly. If you are a nerd, you may want to go native and watch the lecture by Andrei Alexandrescu variadic templates are funadic. You have several examples in the Examples of the course.

Explicit template instantiation

We can tell the compiler to produce the code corresponding to a template function or class using the explicit istantiation. If a source file contains, for instance

```
template class Myclass<double>;
template class Myclass<int >;
template double func(double const &);
```

the object file will contain the code corresponding to the template function **double** func<T>(T **const** &) with `T=double` and that of all methods of the template class Myclass<T> with `T=double` and `T=int`.
This can be useful to save compile or when debugging template classes (since the code for all class methods is generated).
Note: you can explicitly instantiate a class only for values of the parameter arguments for which all class methods make sense!

External template explicit instantiation

To speed up compilation time, what written in the previous slide is not sufficient, we need to tell the compiler that an explicit template instantiation is provided by another compilation unit. This is done using the keyword `extern` in the forward declaration contained in the header file that will be included in all translation units using the instance.

This way, we can speedup the compilation of template classes and functions see the example in Templates/ExplicitInstantiation

The file `mytemp.hpp`

```
template <typename T>
class Myclass{
public:
  Myclass(T const &i):my_data(i){}
  double fun();
...};

template
<typename T> double func(T const &a){...}

// Extern explicit template instantiations
extern template class Myclass<double>;
extern template class Myclass<int>;
extern template double func<double>(double const &);
```

Now source files which include `mytemp.hpp` will not create the machine code for the templates declared **extern**, leaving the corresponding symbols undefined.

The file myfile.cpp

```
#include "mytemp.hpp"
template class Myclass<double>; // explicit instantiantion
template class Myclass<int>; // explicit instantiation
template double func(double const &); // explicit instantiation
```

When compiling this file the template functions and classes indicated are explicitly instantiated: the machine code is generated.

Usually the corresponding object file is stored into a library that will be linked if needed.

The main file

```
int main(){
  Myclass<double> a(5.0);
  Myclass<int> b(5);
  Myclass<char> c('A');
  double d=func(5.0);// call on a double
  double e=func(5);// call on a int
}
```

Using `nm --demangling main.o` we can see that only the code for the constructor and destructor of Myclass<**char**> and that for func<**int**>() has been generated! The other template instances are left undefined and will be resolved by the linker (of course we need to link object file or the library that contains them!).

Interfacing with python and explicit template instances

If you want to create a python module from your C++ template, for instance using pybind11, since python does not have tempaltes you need to instantiate what you want explicitly. You have some examples in the course Examples.

Constrained templates

You can use concepts to constrain class templates

```
#include <concepts> // get some standard concepts
template <std::floating_point T>
class Foo{
Foo(double); //-> conversion from double
...}
template <typename T>
requires std::convertible_to<T,double>
class Toto{
...
}
...
Foo<float> a(10.);// ok
Foo<std::string> b;// Failure: string not floating point type
Goo<Foo> c;//ok double -> Foo conversion is viable
Goo<std::vector<double>>; // Failure: no conversion to double
```

In the first case we could use the simplified form since std::floating_point<T> takes only one template argument. In the second case, we resorted to a requires clause.