# Programmazione Avanzata per il Calcolo Scientifico
# Advanced Programming for Scientific Computing
# Lecture title: Templates

Luca Formaggia

MOX
Dipartimento di Matematica
Politecnico di Milano

A.A. 2014/2015

## Templates

There are many algorithms that work in a analogous way on different types. It would then be useful to treat types as parameter that can be changed to generate the specific algorithm.
In C++ this can be achieved using template parameters: a special kind of parameter that can be used to pass a type as argument.
There are two types of templates:function templates and class templates. We will first deal with the former.
Templates are the tool by which C++ implements the generic programming paradigm.

## Function templates

For instance, consider a function `sort(vector<T>& )` that operates on any type `T` contained in a `vector<T>`, provided that there is an ordering relation among `T` values. In particular the ordering relation can be specified by the functor `less<T>`, or by overloading `operator <`.

How does it work?

## An example of function template

```
template <typename T>
void sort(vector<T> & v){
 // uses less<T> to compare elements
 // of v
}
```

Statement **template** <**typename** T> states that `C` is a template parameter and in particular the keyword **typename** indicates that `C` is of class type. In alternative we could have written **template** <**class** C>, in this context **typename** and **class** have the same meaning.

## Another, simpler, example

We want to write a template function called `mean` that takes two arguments of numeric type and returns the mean value:

```
template<typename T>
inline T mean(T const& a, T const& b){
  return 0.5*(a+b);
}
```

It requires that the addition operator is defined for type `T` as well as the multiplication by a double.

This is an example of a definition of a function template, with a single parameter `T` of class type. It is normally contained in a header file.

## Template parameter and template arguments

How does it work? Suppose that in a source file we have

```
double a;
double b;
...
double c=mean(a,b);
```

The compiler deduce the template argument `double` from the type of the function argument and it instantiates the template function mean<**double**>, obtained by setting `T=double` in the definition of the template.

## Automatic type deduction in function template

In a function template the type of the template argument can be deduced from the type of the function argument. This makes the use of function template very handy.
We may have more than one template argument.

```
template
<typename C, typename D>
double foo(C c, D const & e){..
```

It is important to remember that a template function is generated only if it is instantiated, i.e. if it is used. This has a practical consequence: possible errors can be found only when the template is adopted!

## Resolution of ambiguous argument deduction

Sometimes the compiler cannot deduce the arguments, also because implicit conversions do not apply for template argument deduction.
For instance, the situation

```
double a; int b; mean(a,b);
```

is ambiguous. We can solve the situation in two ways:

```
mean(a,static_cast<double>(b));//uses mean<double>
mean<double>(a,b);//uses mean<double>
```

In the second case we indicate the template argument type explicitly, and then the implicit conversion `int` $\rightarrow$ `double` applies.
However, try to avoid indicating the argument type explicitly.

## Constant template parameter

A template parameter may also be an integral constant. In this case in the declaration `typename` is replaced be the type of the constant:

```
template<unsigned int N>
double Pow(double const & x){
double tmp(1);
for(unsigned int n=0;n<N;++n)tmp*=x;
return tmp;}
```

In this case we need to specify the template argument value when calling the function, since it cannot be deduced from the function arguments:

```
double y=Pow<3>(5.0);
```

Is there a more efficient way of doing it?....think about it....

## Allowed types for constant template parameters

We can use constant template parameters only for the following types: integral constants, enumerations, pointer to object or to function, lvalue reference to pointer or to functions, pointer to member object or to member function, std::nullptr_t

```
template <int i> fun();// ok
template <double* i=std::nullptr> fun2();// ok
template <double d=5> fun3();// ERROR
```

## Template parameters and arguments

To summarize

- template parameters can be of two sorts: types, declared using `typename`, or `class`, or constants, declared by indicating the type of the constant. They appear in the declaration of the function template: e.g. **template** <**typename** T, **int** N>.
- The template arguments are the actual values that are given to the parameters used to instantiate the corresponding template function: mean<**double**> is generated from the function template by setting `T=double`.

## Function template overloading

And what about pointers? If `a` and `b` are **double**∗ a call to mean(a,b) will return the mean value of the pointers! Probably I want instead the mean of the pointed values! I can solve the problem with a overloading (sometimes erroneously called partial specialization. There is no partial specialization for function templates).

```
template<typename T>
T mean(T const * a, T const *b){
 return 0.5*(*a+*b);}
```

This version will be preferred if the arguments are pointers.

## Full function template specialization

Maybe we want our mean function to operate also on user defined types. For instance on `Triangles` it may return a triangle by averaging the coordinates of the given ones. Here we have a full specialization:

```
template<>
Triangle mean (Triangle const & a,
               Triangle const & b){
     Triangle tmp;
     // compute average triangle
     return tmp;}
```

The **template**<> indicates that we are specializing the generic template, and we do not have any template parameter (full specialization).

## Overloading with free functions

Our mean function does not operate on integers correctly:

```
mean(5,6);
```

would produce an integer equal to 5! Maybe we would like to have a double equal to 5.5. However, I cannot specialize our template since the return type will not be equal to that declared in the primary (un-specialized) definition! Yet I can still use a free function:

```
double mean(int const &a, int const & b){
  return 0.5*(a+b);}
```

This is an example of the use of a free function to overload some instances of a template (if you want to avoid free function either you need to give the return value as template parameter or to use type traits...we will see them later.)

## A more complex example

```
template<typename RT, typename T> inline RT Max(T const
& a ,T const & b){
return static_cast<RT> a < b ? b : a;}
```

In this case the first template argument must be specified.

`max<float>(4,5)`

calls `max<float,int>`.

## Overloading of template functions

Overloading rules applies also to templates:

```
int const& Max (int const& a, int const& b) {...}
template <typename T>
T const& Max (T const& a, T const& b){...}
template <typename T> T const& Max (T const& a, T const&
b, T const& c){...}
int main(){
Max(7, 42, 68); // calls the function with 3 args
Max(7.0, 42.0); // calls Max<double>
Max('a', 'b'); // calls Max<char>
Max(7, 42); // calls Max()
Max<>(7, 42); // calls Max<int>
```

## Overloading of template functions

When we have overloaded template function the compiler follows a (rather complicated) rule to resolve a function call.

- Non-template functions (free functions) have the precedence: if an exact matching is found with a free function the latter is chosen;
- More specialized template functions have the precedence over less specialized templates. In other word, the compiler chooses the template where the template parameter matches with a "simpler type": `double` is a simpler type than `double *`.
- No implicit conversion is applied to template functions arguments. Yet it is applied to a free function with the same name, however if a matching template function is found the latter is preferred.
- If in doubt, the compiler issues an error and the compilation is stopped.

## Rules for template deduction

The rules for template deduction are normally what you want, but with something that is not always obvious. In particular references are stripped. So lets consider the statement

```
template <class T>

f(PartType T);
```

where, of course, `ParType` depends on `T`. An consider the following possibilities.

## ParamType is a pointer, a reference, but not an “universal reference”

When we call f(expr) if expr is a reference the reference part is ignored.

```
template<typename T>
f(T& x);
template<typename T>
g(T* x);
..
int x=27;
const int cx=3; // cx is a const int
const int & cr=cx; // cr is a int&
f(x)// T is int, ParType is int&
f(cx)// T is const int, ParType is const int&
f(rx)//T is const int, ParType is const int&
g(x)// T is int, ParType is int*
g(cx)// T is const int, ParType is const int*
g(rx)//T is const int, ParType is const int*
```

## ParamType is neither a pointer, nor a reference

When we call f(expr) if expr is a reference the reference part is ignored. Also constness is ignored, because we are passing by value.

```
template<typename T>
f(T x);
..
int x=27;
const int cx=3; // cx is a const int
const int & cr=cx; // cr is a int&
f(x)// T and ParType are both int
f(cx)// T and ParType are both int
f(rx)//T and ParType are both int
```

## ParamType is a universal reference

The rvalue reference conversion rule applies.

```
template<typename T>
f(T&& x);
int g();// a function returning an int
..
int x=27;
const int cx=3; // cx is a const int
const int & cr=cx; // cr is a int&
f(x)// T is int&, ParType is int&
f(cx)/ T is const int& ParType is const int&
f(rx)// T is const int& ParType is const int&
f(g())// T is int ParType is const int&&
f(27)// T is int ParType is const int&&
```

In the last two example we are passing to f an rvalue reference. So ParType is an rvalue reference (but x is an lvalue!! we will see this later on...).

## Default arguments in function template parameter (C++11 only!)

For obscure reasons in C++98 it was forbidden to give a default values to function template parameters, while it was allowed for class templates (as we will see...). C++11 introduced finally this possibility:

```
template<typename RT=double, typename T>
inline RT Max(T const & a,T const & b){
return static_cast<RT> (a < b ? b : a);}
```

Now Max(3.0,4.0) works. Clearly it makes little sense to give a default value to template parameters that are deduced from the function arguments. Even if it is not an error....

## Declaration and definition

Also for function templates one can separate a pure declaration

```
template <typename T>
T mean (T const& a, T const& b);
```

from the definition, which contains the body of the function

```
template <typename T>
T mean (T const& a, T const& b){
return 0.5d*(a+b); }
```

(remember however that a definition is also a declaration).

Because of the particular mechanism of template instantiation both should be put in a header file unless it is a complete specialization: in that case the definition goes in a cpp file. We will see later the possible organization.

## A note

Template definitions go in a header file because the istantiation, i.e. the determination of the actual types to be used for the template arguments, must be resolved at compilation stage. This is not true for full specializations because the type of the arguments is in this case already known! So the compiler do not have to instantiate the template, it uses the full specialized definition directly, like if it were an ordinary function.

## Template methods

An (ordinary) class may have template methods:

```
class FEMSpaceP1{
public:
...
// Project the FemSpace FEMS on
// the L2 space.
template <typename FEMS>
 void L2Project(FEMS const & fspace);
}
```

Template methods can be specialized like function templates.

Template methods cannot be virtual.

## Class templates

A class template is a class where some types are left free and set through template parameters, in a way similar to function templates:

```
template <typename T>
class A{
member declarations, here T is a generic type
};
```

An instance of `A<T>`, for instance `A<double>` is a template class. It is obtained from the definition above by replacing `T` with `double`. The object defined by `A<double> a;` is called *template object*.

```
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
  Vcr(const Vcr&);       // copy constructor
  ~Vcr();                // destructor
  int size();
  inline T& operator[](int i);
  Vcr& operator=(const Vcr&);    // overload =
  Vcr& operator+=(const Vcr&);   // v += v2
  Vcr& operator-=(const Vcr&);   // v -= v2
  double maxnorm () const;             // maximum norm
  double twonorm() const;              // L-2 norm
  template<class S>
  friend S dot(const Vcr<S>&, const Vcr<S>&);// dot product
};
```

## Methods of a class template

*Methods* in a class template behave similarly to function templates. An essential property is that methods of a class templates are instantiated only if effectively used.

```
Vcr<double> a; ...
a.maxnorm();
```

The compiler generates the code for `Vcr<double>::maxnorm()` but not, for instance, that for `Vcr<double>::twonorm()`.

This aspect is essential for generic programming to work effectively: I can use my `Vcr` also for storing types for which some `Vcr` methods do not make sense, as long as I do not use them!! It is the base of the so-called SFINAE paradigm: Substitution Failure Is Not an Error: when substituting the deduced type for the template parameter fails, the specialization is discarded from the overload set instead of causing a compile error

## Usage of a class template

```
Vcr<double> a,p;
Vcr<int> b;
Vcr<float *> z;
Vcr<Vcr<double> > c;//note the space before >
double h=dot(a,p);// usa dot<double>
```

In C++11 I can avoid the space:

`Vcr<Vcr<double>> c`

In C++98 I need them (the compiler otherwise would interpret >> as the byte-shift operator and issue an error.)

## Member definition

```
template<typename T>
Vcr<T>::Vcr(const Vcr & vec) {
vr.resize(vec.length);
for (int i = 0; i < length; i++) vr[i] = vec[i];
}
template<typename T> Vcr<T>& Vcr<T>::operator=(const Vcr&
vec) {
if (this != &vec) {
if (length != vec.length ) error("bad vector sizes");
for (int i = 0; i < length; i++) vr[i] = vec[i];
}return *this;}
```

Within the *scope* of the class and that of its members I can avoid to use `<T>` to indicate `Vcr<T>`.

One may still be explicit, it is not an error.

## Member definition

```
template<typename T>
Vcr<T>::Vcr<T>(const Vcr<T> & vec) {
vr.resize(vec.length);
for (int i = 0; i < length; i++) vr[i] = vec[i];
}
template<typename T> Vcr<T>& Vcr<T>::operator=(const Vcr<T
vec) {
if (this != &vec) {
if (length != vec.length ) error("bad vector sizes");
for (int i = 0; i < length; i++) vr[i] = vec[i];
}return *this;}
```

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

## The dot product

```
template<typename S>
S dot(const Vcr<S> & v1, const Vcr<S> & v2) {
if (v1.length != v2.length ) error("bad vector sizes");
S tm = v1[0]*v2[0];
for (int i = 1; i < v1.length; i++) tm += v1[i]*v2[i];
return tm; }
```

Here `<S>` in `Vcr<S>` cannot be omitted.

*Note:* `S` is here just a placeholder, I could have used `template<typename T>`.

## Integer constants as parameters

Also for classes *integral constant expressions, pointers, enumeration...* con be template parameters:

```
template <typename T, int NDIM>
class Point{
private:
T data[NDIM];
... }; ...
Point<double,3> p;// a point in R^3.
```

Beware: `int n;...   Point<double,n> p;` is wrong if `n` is not a constant expression. In other words: the compiler must know the actual value of the template argument when the object is created.

## Default parameter value

We may give a default value to the parameters, with the usual rule of starting from the rightmost parameter.

```
template <typename T, int NDIM=2>
class Point{
...
}; ...
template <typename T=double> Vcr{
...}
```

Now `Point<int> a;` is equivalent to `Point<int,2>` and `Vcr<> p;` is equivalent to `Vcr<double>`.

## Prerequisite on template parameters

The language gives very little support in prescribing the prerequisite of the type that may be passed as template arguments. Some checks may be done using type traits (introduced in C++11), but still they do not cover all the possible specifications (and their use may be rather complex).

So it is very important to document template code to let the user know what it is required from the template parameter: should it be copy-constructible? Does the code carry out particular operations on template variables (addition, comparisons...)?

## 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 methods of the specialized version must be defined (specialization is NOT inheritance).
For instance `Vcr<T>` implements methods that are sound for a wide variety of types. Yet for some types it may be necessary, or simply computationally convenient, to use specialized methods.
For instance, for a `Vcr<complex<double> >`, *(you must include the `<complex>` header)*, the method `maxnorm()` has to be rewritten (as well as the function `dot`).

## Partial specialization (overloading) of a template method

We may decide to partially specialize the 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>(complex<T> const &)` is provided by the standard library.

## Specializing the 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]; }
        ...
  double maxnorm () const;
  double twonorm() const;

  template<class S>
  friend complex<S> dot(const Vcr<complex<S> >&,
                           const Vcr<complex<S> >&);
};
```

Now we need to redefine all methods! Class specialization is needed if you have to change the return type of methods.

## Full class template specialization

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

  friend complex<double> dot(const Vcr&, const Vcr&);
};
```

Now `Vcr< complex<double> > a;` will use the fully specialized class.

## Definition of methods of a specialized class template

```
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; }
```

For the full specialization:

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

Note that here we do not need **template**$<>$.

## Other examples

```
template <typename T1, int N>
class MyClass{
...}
```

Partial specialization for pointers

```
template <typename T1, int N>
class MyClass<T1*, N>{
...}
```

Specialization for the integer constant

```
template <typename T>
class MyClass<T,3>{
...}
```

Full

```
template <>
class MyClass<char,0>{
...}
```

## Specialization on integer template parameters

The specialization on integer template parameters can be used for recursively defined function:

```
template<unsigned int N>
struct Pow{
  static double apply(double const & x){
    return x*Pow<N-1>::apply(x);}
};
// Specialization for 0.
template<>
struct Pow<0>{
  static double apply(double const & x){return 1;}
};
```

See Pow/Pow.hpp.

## 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>;
template<> MyClass<double>{
...}; // full specialization.
```

- When specializing a class template we need to provide the definition of all specialized methods.
- Specialization of a function template must correspond to a possible instance of the primary function template.
- Also at the moment of the instance of a specialized template at least a pure declaration (or forward declaration of the class) of the primary template must be visible.

## A note

When specializing a function template we can omit to characterize the name of the function with the <> only if the type can be deduced from the function arguments. Example:

```
template <typename T>
bool myfun(){T var,...}
```

A specialization written an

```
template <> bool myfun(){int var,...}
```

is wrong: the type is not deducible from the arguments. We need to write

```
template <> bool myfun<int>(){int var,...}
```

## 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<double> a;
}
```

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

## Template alias (C++11 only)

Templates can be long and tedious to write. C++11 has introduced 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 >.

## Templates for specifying policies

Templates may be a way to specify a policy. Let's make a simple example. We want to compare two strings for equality and have the possibility of distinguish case sensitive and case insensitive comparisons.

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

## The working example

See the example in Templates/Compare/main.cpp

## A possible file organization with templates

stack.hpp

```
#ifndef __STACK_HH
#define __STACK_HH
template <class T>
stack{
  void push(const T   &);
 ...
};

#include "stack_imp.hpp"

#endif
```

stack_imp.hpp

```
#ifndef __STACK_IMP_H
#define __STACK_IMP_H

template<class T>
void stack<T>:: push(const T&){
                ...
}
template<class T>
T & stack<T>top(){

...
}
...
#endif
```

## Another possible organization with templates

Another possibility is to leave everything in one header file. However, if the functions/methods are long it may be worthwhile, for the sake of clarity, to separate definitions from declaration. You put at the beginning of the file classes and function with only the declaration of long functions and methods.
Then, at the end of the file, you add the definitions. In this way the file is easier to read.

## The typename keyword

Types that depend on the template parameter must be indicated with the keyword typename:

```
 class Matrix{
typedef double Real;
...
}
template <typename M> typename M::Real Fnorm(M & matrix){
typename M::Real a;
...
}
```

When the compiler process the function template does not know the type of the parameter. So how can it know that M::Real indicates a type? It will interpret it as a static class member instead!. Creating a syntax error.

## Curiously recurring template patters

This rather surprising technique is used to implement what is called “static polymorphism”, the base class delegates the implementation to the derived class, but using templates!!

```
template <class D>
class Base{
 double f(){static_cast<D*>(this)->f();}
...};
class Derived: public Base<Derived>{
// implementation of f
double f(){...}
}
```

See CRTP/crtp.hpp

## What's the usage of CRTP

The CRTP implements a static polymorphism, without the overhead of virtual function calls. At a price of more rigidity (it is static, the type is known at compile time!).
Moreover it simplifies parameter argument deduction if the `Derived` is itself a template class, as we will see in the next slide.

## Another use of `typename`

Taken from the Eigen manual

```
#include <Eigen/Dense>
using namespace Eigen;

template <typename Derived1, typename Derived2>
 void copyUpperTriangularPart(MatrixBase<Derived1>& dst, const
MatrixBase<Derived2>& src){
/* Note the 'template' keywords in the following line! */
dst.template triangularView<Upper>() =
     src.template triangularView<Upper>();}
```

`MatrixBase` is the base class template for all Eigen matrices and it adopts the CRTP technique. Now `Derived` can be a complex template class. Yet, who cares..., the compiler will automatically deduce it: the function will then work with any Eigen matrix! (and no virtual call overhead).

Yet we need to help the compiler if we call template method of template dependent arguments.

## Explanation

The reason that the template keyword is necessary has to do with the rules for how templates are supposed to be compiled in C++. The compiler has to check the code for correct syntax without knowing the actual value of the template arguments (Derived1 and Derived2 in the example).

That means that the compiler cannot know that `dst.triangularPart` is a member template and that the following < symbol is part of the delimiter for the template parameter. Since `dst.triangularPart` is a member template the programmer should specify this explicitly with the template keyword and write `dst.template triangularPart`.

## The general rule

- A dependent name is a name that depends (directly or indirectly) on a template parameter. In the example, dst is a dependent name because it is of type `MatrixBase<Derived1>` which depends on the template parameter Derived1.
- If the code contains either one of the constructs `xxx.yyy` or `xxx->yyy` and `xxx` is a dependent name and `yyy` refers to a member template, then the template keyword must be used before `yyy`, leading to `xxx.template yyy` or `xxx->template yyy`.
- If the code contains `xxx::yyy` and `xxx` is a dependent name and `yyy` refers to a member `typedef`, then the `typename` keyword must be used before the whole construction: `typename xxx::yyy`.

## The use of this in templates

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

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. This to avoid ambiguous situations. So in this case the free function `myfun()` is used!

## Solution

```
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 (C++11 only)

C++11 allows to specify that a function or a class may take an arbitrary number of template parameters. This is very handy when you want specialization with different number of template parameters.

```
template<typename... Arguments>
class manyParameters{
void SampleFunction(Arguments... params){
...}
```

It is a new feature, the explanation goes beyond the scope of these lectures, yet it may increase the capability of the language greatly. Think for instance to

```
template<typename... BaseClasses>
class VariadicTemplate : public BaseClasses...{..
```

A class that derives from its template parameters, which may be present in any number! A new way to implement composition!.

## Another use of variadic templates

Sometimes you want to have a template template parameter with a variable number of parameters. For instance, many standard containers, like li!vector¡T¿! and set<T> require a single compulsory template argument, but they have other defaulted ones. So,

```
template<class T, template<class> class C>
class Foo{
...// other members
 C<T> M_container};
```

will not work if C=vector or C=set.

But

```
template<class T, template<class...> class C>
class Foo{
...// other members
 C<T> M_container};
```

works fine for all containers that take only one compulsory template argument. The other arguments will be defaulted!!

## An interesting example

Another example of usage of variadic templates is in CompositionWithVariadicTemplates/main.cpp where variadic templates are used to implement the composition design pattern.
If you are a nerd, you may want to go native and watch the lecture by Andrei Alexandrescu variadic templates are funadic.

## Template instantiation and linkage

The compiler produces (only) the code corresponding to function templates and class templates members that are actually used in the code, for the selected value of the template parameters.
It means that all compilation units that contains, for instance, an instruction of the type `vector<double> a;` produce the machine code corresponding to the default constructor of a `vector<double>`. If we then have `a.push_back(5.0)`, the code for the `push_back` method is produced, and so on.

It then happens frequently, if you have many compilation units, that the same machine code is produced several times. It is the linker that eventually selects only one instance to produce the final executable. However, there is a (almost inevitable) wasteful production of code (and compilation time).

## Explicit template istantiation

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 is sometimes useful, particularly when debugging template classes (we make sure that 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 method!s make sense!

## External template explicit instantiation (C++11 only)

In C++11 we can tell the compiler that an explicit template istantiation is provided by another compilation unit, using the keyword `extern`.
In this way we can speedup the compilation of template classes and functions if we know beforehand that they will be mostly used for certain value of the template argument. Let's 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 then put in a library.

## 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 the library that contains them!).

## Reference wrappers

C++11 has solved a problem that in C++98 was difficult or impossible to solve. Assume we have the following template function

```
template <class Mat>
double L2Norm (Mat x){
// computes the L2 norm of Mat
// Mat being a matrix (for instance a Eigen matrix)
}
```

Here the matrix is passed by value, then the function makes a local copy. However, we may want for efficiency to pass a constant reference instead!
To do that, we should change the definition of the function into **double** L2Norm (Mat **const** & x). However, suppose the function is provided by an external library, certainly we would prefer not to mess with it. Even writing a specialization in this case does not work since the reference is not a first class type: the compiler will not be able to disambiguate the call.

## Reference wrappers

The solution is provided in the <functional> header, by the two reference wrappers std::ref() and std::cref(), which provide an implicit conversion to a reference (and to a constant reference).

```
MatrixXd A;
...
double norm=L2Norm (std::cref(A));
```

Now A is passed by reference to a constant value. If we need a non-const reference we may do

```
double norm=L2Norm (std::ref(A));
```

Another example in ReferenceWrapper/main_wrap.cpp