# Programmazione Avanzata per il Calcolo Scientifico
# Advanced Programming for Scientific Computing
# Lecture title: Metaprogramming, traits and type_traits

Luca Formaggia

MOX
Dipartimento di Matematica
Politecnico di Milano

A.A. 2014/2015

# Some elements of Metaprogramming

Metaprogramming is the writing of computer programs that write or manipulate other programs (or themselves) as their data, or that do part of the work at compile time that would otherwise be done at runtime. This may allow programmers to minimize the number of lines of code to express a solution by the user (hence reducing development time), or to generate more efficient code.
In C++ we indicate with template metaprogamming techniques by which intermediate code is generated by the compiler thanks to the use of C++ templates.

# Template metaprogramming

Template metaprogramming is a metaprogramming technique in which templates are used by a compiler to generate temporary source code, which is merged by the compiler with the rest of the source code and then compiled. The output of these templates include compile-time constants, data structures, and complete functions. The use of templates can be thought of as compile-time execution.

Metaprogramming uses constructs that can be resolved at compile time (static constructs). It is a small subset of C++ constructs, but sufficient to do quite a lot of stuff.
The main aims are:

- To allow a more generic programming, by implementing a better control on types (traits and type_traits).
- Allow to increase the efficiency of some algebraic manipulation by letting the compiler to some optimization at compile time (numerical metaprogramming).

# What can we use?

1. Integral constants.
2. Literals and constant expressions.
3. The operator `sizeof()` applied to a type and (since C++11) `decltype()` and `declval<T>()`. Definition of types `typedefs`.
4. Static casts (`static_cast<>()`)
5. The resolution of the return type of a function, even if it belongs to a family of overloaded functions.
6. Instance of template classes and functions.

# Unevaluated context

An important thing to remember is that the operands of **sizeof**(), decltype() and noexcept (as well as alignof() and **typeid**()) are never evaluated, not even at compile time.

- No code is generated;
- We need only declaration of the a function or object name in this context.

```
class A {
double M_x, in M_n
...}// only declaration

sizeof(A); n bytes of an object of type A
```

# decltype and declval<T>

decltype(expr) returns the type of an expression. declval<T> (in <utility>) makes it possible to use member functions of T in decltype expressions without the need to go through constructors! It can be used only in a non-evaluated context.

```
class A{
...
public:
double foo();
}
...
decltype(std::declval<A>().foo()) n;// n is an int
```

No object of type A is constructed and foo() is not called. We just get the type of the return value!.

# A first example

```
 template <bool v>
struct Bool2Type
{
static constexpr bool value=v;
constexpr operator bool()const {return v;}
};
using true_type = Bool2Type<true>;
using false_type = Bool2Type<false>;
```

This template is a generator of types parametrized by a bool.

Beware: C++11 in <type_traits> already provides true_type and false_type as specializations of integer_constant<T, T v>

Let's suppose that the class NiftyContainer contains a pointer to an object of type T and one wants to make a copy of the object using either the copy constructor or the method clone() (if the object is polymorphic) and you want to indicate the two possibilities with a bool argument:

```
template <typename T, bool isPolymorphic>
class NiftyContainer{
...
void DoSomething(){
T* pSomeObj = ...;
if (isPolymorphic)
{
T* pNewObj = pSomeObj->clone();
... polymorphic algorithm ...
}else{
T* pNewObj = new T(*pSomeObj);
... nonpolymorphic algorithm ...
}
}
...
```

# The example does not compile!

Why? The compiler looks for the method clone() in any case!
(even if eventually, because of optimization, the part of code never executed is not produced!!)

# The solution:

```
template <typename T, bool isPolymorphic>
class NiftyContainer{
private:
void DoSomething(T* pObj, true_type){
T* pNewObj = pObj->clone();
... polymorphic algorithm ...
}
void DoSomething(T* pObj, false_type)
{
T* pNewObj = new T(*pObj);
... nonpolymorphic algorithm ...
}
public:
void DoSomething(T* pObj){
DoSomething(pObj, Bool2Type<isPolymorphic>());
}
...
```

# Traits

From Wikypedia: *a trait is a collection of methods, used as a "simple conceptual model for structuring object oriented (or generic) programs*

In C++ with traits we indicate a template class or struct that gives information about template dependent types. It is a useful toll to create generic code able to operate on a large set of types, or as an alternative to class/function specialization.

Traits (usually) do not have methods, but only type definitions (`typedefs`) and possibly static methods and constants.

# Traits and Policies

Traits and policies share similar objectives and indeed sometimes the term trait is used also to indicate policies: *think of a trait as a small object whose main purpose is to carry information used by another object or algorithm to determine "policy" or "implementation details" (Bjarne Stroustrup)*.

However trait normally indicates a technique to handle types while policy refers to a class that provides implementation details.

# An example

Consider this bit of code, which calculates the arithmetic average of a vector

```
template<class T>
T average(const T* data, int numElements)
{
  T sum = 0;
  for (int i=0; i < numElements; ++i)
     sum += data[i];
  return sum / numElements;
}
```

Is it really generic? It will work fine if `T` is a floating-point type, but what about integers? We can make a specialization (in C++11, since in C++98 template function partial specialization is not possible), but this would require to rewrite the code!

# A solution with traits

```
template<class T>
struct float_trait {
typedef T T_float;
};
template<>
struct float_trait<int> {
typedef double T_float;
}
// etc. etc.
template<class T>
typename float_trait<T>::T_float average(const T* data,
int numElements){
  typename float_trait<T>::T_float sum = 0;
  for (int i=0; i < numElements; ++i)
    sum += data[i];
  return sum / numElements;}
```

# A level of indirection...

We have delegated to float_trait<> the role of carrying the information about the types we want to use according to the value of T. We need now to specialize only float_trait<> and not the whole function.

Traits are heavily used in generic programming.

# Type Traits

With type traits we indicate a metaprogramming technique whose aim is to operate on types. C++11 introduces an extensive set of type traits into the language, with c++98 type traits are available through the Boost library.

But before introducing those new features of C++11, let's give two examples.

# A simple example

We want to assess if a type is an **int** (C++11 already gives us a similar tool!)

```
template<typename T>
struct IsInt{
  constexpr bool value=false;};
template<>
struct IsInt<int>{
  constexpr bool value=true;};
....
template<typename T> int f(T){
  if(IsInt<T>::value){
    // done only if T is int
...
}
```

# A more complex example

```
template<typename B, typename D>
class IsDerived{
private:
  //! A function returning true
  static true_type test(B* );
  //! A function returning false
  static false_type test(...);
public:
  //! type is true_type or false_type according to the result of the test
  using type = decltype( test(static_cast<D*>(0) ) );
  //! value is true or false according to the result of the test
  static constexpr bool value = type::value;
};
```

Note: the two overloaded test() methods are only declared!.

# Usage

```
// Compilation fais if B does not derive from A
static_assert(IsDerived<A,B>::value,"ERROR");
...
// But I can also test it
cout<< "B Derives from A?" <<IsDerived<A,B>::value <<endl;
..
```

# The SFINAE concept

The previous example relies on function overloading and the fact that variable argument (ellipsis) function are the last choice in the selection of an overloaded function: thus `test(...)` here it acts as a "catcher" of all calls with an argument not derived from `B`.
Other techniques relies to the SFINAE C++ paradigm:
"Substitution failure is not an error". It is related to template function instance mechanism by which calling a function with the wrong type of argument is not an error is there is an overloaded version or a template specialization for which the argument is valid.

# Other things to note

- The value `0` can be cast to any pointer type, it becomes the null pointer. I could have used `nullptr` instead.
- Since the function is not actually called (we are in un-evaluated context) we do not need to provide the definition! Indeed the two version of `test` are here only declared and used just to derive the corresponding return type!

The code is in MetaProgramming/IsDerived/isDerived.hpp,
Another version MetaProgramming/TypeTraitsC11/isDerived.hpp.
It uses SFINAE.

# A last example: If-then-else

To choose the type according to a condition (statically defined!).
Note that <type_traits> provides std::conditional<**bool**,A,B> that does the same thing!

MetaProgramming/ifthenelse/isfthenelse.hpp.

# Usage

```
//!This function returns B* if D derives from B
//! otherwise it returns D*
template <typename B,typename D>
IfThenElse<isDerived<B,D>::value, B*, D*>::ResultT
aStrangeFunction(B& a, D& d){....
```

# C++11 Type traits

C++11 has introduced an extensive set of type traits, many taken from those defined in the boost. You need the header <type_traits>

Before giving some indication on the main ones we give some background information.

First C++11 type traits are template classes (actually structs), It is not important to know the details, just that they either provide a value or a type (or both).
If a value is provided, the trait contains a static variable called value, containing the value. In the other case, it contains a **typedef** called type, which provides the type. Moreover, in the cases the trait returns a value it has a **typedef** value_type (the type of the value), and is convertible to that type (providing value).

# An example

is_pointer<T> may be used to interrogate if a type is a pointer.

```
#include <type_traits>
class Matrix{...};
using MP=Matrix*;
bool is_p=std::is_pointer<Matrix>; // false
bool js_p=std::is_pointer<MP>; // true
typename std::is_pointer<MP>::value_type j;// j is a bool
bool z= is_pointer<MP>(); // z is true.
```

## true_type and false_type

Two special traits are used to incorporate in an object the meaning of true and false. They are made in such a way that they are convertible to a bool of the corresponding value, therefore an object of those classes can be used in conditional operators:

```
if(std::true_type())
```

is equivalent to

```
if(true)
```

# Organization of standard type_traits

- Primary type categories: is_int<T>, is_pointer<T>, is_function<T>, is_rvalue_reference<T>, is_enum<T>, etc. Used to interrogate some fundamental characteristic of types.
- Composite type categories: is_scalar<T>, is_reference<T>, is_member_pointer<T> etc.
- Type properties: is_const<T>, is_trivial <T>, is_abstract<T>, is_polymorphic<T>
- Supported operations: is_copy_constructible<T>, is_assignable<T>, has_vistual_destructor<T>.
- Type relationships: is_base_of<B,D>, is_convertible<From,To>
- Type modifications: remove_const<T>, add_const<T>, make_unsigned<T>.

## static_assert(expr,string)

C++11 adds to the assert() statement the statement static_assert, that evaluates a condition at compile time and aborts the compilation if it is not satisfied.

```
template<class B, class D>
class{
static_assert(is_base_of<B,D>::value,"B must be base class of D");
...
}
```

# Some technicalities

Some of the type traits provided by C++11 may be used to check the memory layout of a type. We need to clarify some concepts (you may skip these two slides!).
A type is trivial if can be statically initialized. It also means that it is legal to copy data around via memcpy, rather than having to use a copy constructor. A trivial class or struct is defined as one that:

- Has a trivial default constructor. This may use the default constructor syntax (SomeConstructor() = default;).
- Has trivial copy and move constructors.
- Has trivial copy and move assignment operators.
- Has a trivial destructor, which must not be virtual.
  Constructors are trivial only if there are no virtual member functions of the class and no virtual base classes.
- All the non-static data members are trivial.

For a definition of trivial constructor and trivial copy construtors look cppreference.com.

# Some technicalities, PartII

A type that is standard-layout means that it orders and packs its members in a way that is compatible with the C language. A class or struct is standard-layout, by definition, provided:

- It has no virtual functions
- It has no virtual base classes
- All its non-static data members have the same access control (public, private, protected)
- All its non-static data members, including any in its base classes, are in the same one class in the hierarchy The above rules also apply to all the base classes and to all non-static data members in the class hierarchy
- It has no base classes of the same type as the first defined non-static data member

A class/struct/union is a POD if it is trivial, standard-layout, and all of its non-static data members and base classes are PODs.

# An example on numerical metaprogramming $X^n$

```
template <long int X, unsigned int N>
struct Pow{
  using value_type = long int;
  const static value_type value = X * Pow<X,N-1>::value;
  constexpr operator value_type()const {return value;}
};
template <long int X>
struct Pow<X,0>{
  using value_type = long int;
  const static value_type value = 1l;
  constexpr operator value_type()const {return 1l;}
};
```

# Expression templates

Lets suppose to have a template for vectors

```
template<typename T>
class Vtr{
...
Vtr<T> operator +(const Vtr<T> & b) const{
Vtr<T> tmp(b);
for(int i=0;i<=size();++i)tmp[i]+=_data[i];
return tmp;}
T & operator [](int i){return _data[i];}
T & operator +(Vtr const & a)...
...};
```

# Drawbacks of operator overloading

Expression

```
Vrt<double> a,b,c,d;
...
d=a+b+c;
....
```

Produces 2 temporaries (we may save 1 with move semantic) and the execution of three loop cycles: two for the sums and one for the assignment. It is mode efficient to have a single cycle:

```
for(int i=0;i<=a.size();++i){
d[i]=a[i]+b[i]+c[i];}
```

# Expression templates

Expression templates is a metaprogramming technique by which we let the compiler to do some optimization at compile time that would be not possible otherwise. In particular we may use it to maintain the synthetic and intuitive notation provided by operator overloading, while obtaining the efficiency of a single loop cycle, avoiding the creation of big temporaries.
The idea is to defer the evaluation of the operations involved in the expression by creating special class templates that incorporate the structure of the expression to be evaluated. Those classes will contain references to the operands. The assignment operator will cause the expression to be evaluated.

# A class representing a binary operation

```
template<class Lo, class Op, class Ro>
class LOR{
public:
typedef typename Ro::T T;
Lo & lo;
Ro & ro;
LOR(Lo & a, Ro &b):lo(a),ro(b)
T operator[](int i){
return Op::apply(lo[i],ro[i]);
}};
```

# The policy for the addition

```
struct Add {
template<typename T>
static T apply(T const & a, T const & b ){
return a+b;}
};

// Overloading of + operator
template <class Lo, class Ro>
LOR<Lo,Add,Ro>
operator +(Lo & a, Ro & b){
return LOR<Lo,Add,Ro>(a,b);
}
```

# The copy assignment operator

Vrt is a class of vectors

```
template<typename TT>
class Vtr{
private:
T * _data
public:
typedef TT T;
...
template<class Lo, Class Op, class Ro>
void operator =(LOR<Lo,Op,Ro> const & exp){
for(int i=0;i<=size();++i) _data[i]=exp[i] }
};
```

# Expression templates at work

```
d=a+b+c
```

```
d=LOR<Vtr<T>,Add,Vtr<T> >(a,b)+c
d=LOR<LOR<Vtr<T>,Add,Vtr<T> >,Add,Vtr<T> >(tmp,c)
```

The loop in the assignment to _data becomes:

```
_data[i]=Add::apply(Add::apply(a[i],b[i]),c[i])
```

Becomes:

```
_data[i]=(a[i]+b[i])+c[i]
```

# Expression templates at work

```
d=a+b+c
```

```
d=LOR<Vtr<T>,Add,Vtr<T> >(a,b)+c
d=LOR<LOR<Vtr<T>,Add,Vtr<T> >,Add,Vtr<T> >(tmp,c)
```

The loop in the assignment to _data becomes:

```
_data[i]=Add::apply(Add::apply(a[i],b[i]),c[i])
```

Becomes:

```
_data[i]=(a[i]+b[i])+c[i]
```

# Expression templates at work

```
d=a+b+c
```

```
d=LOR<Vtr<T>,Add,Vtr<T> >(a,b)+c
d=LOR<LOR<Vtr<T>,Add,Vtr<T> >,Add,Vtr<T> >(tmp,c)
```

The loop in the assignment to _data becomes:

```
_data[i]=Add::apply(Add::apply(a[i],b[i]),c[i])
```

Becomes:

```
_data[i]=(a[i]+b[i])+c[i]
```

# Expression templates at work

```
d=a+b+c
```

```
d=LOR<Vtr<T>,Add,Vtr<T> >(a,b)+c
d=LOR<LOR<Vtr<T>,Add,Vtr<T> >,Add,Vtr<T> >(tmp,c)
```

The loop in the assignment to `_data` becomes:

```
_data[i]=Add::apply(Add::apply(a[i],b[i]),c[i])
```

Becomes:

```
_data[i]=(a[i]+b[i])+c[i]
```

# Expression templates at work

```
d=a+b+c
```

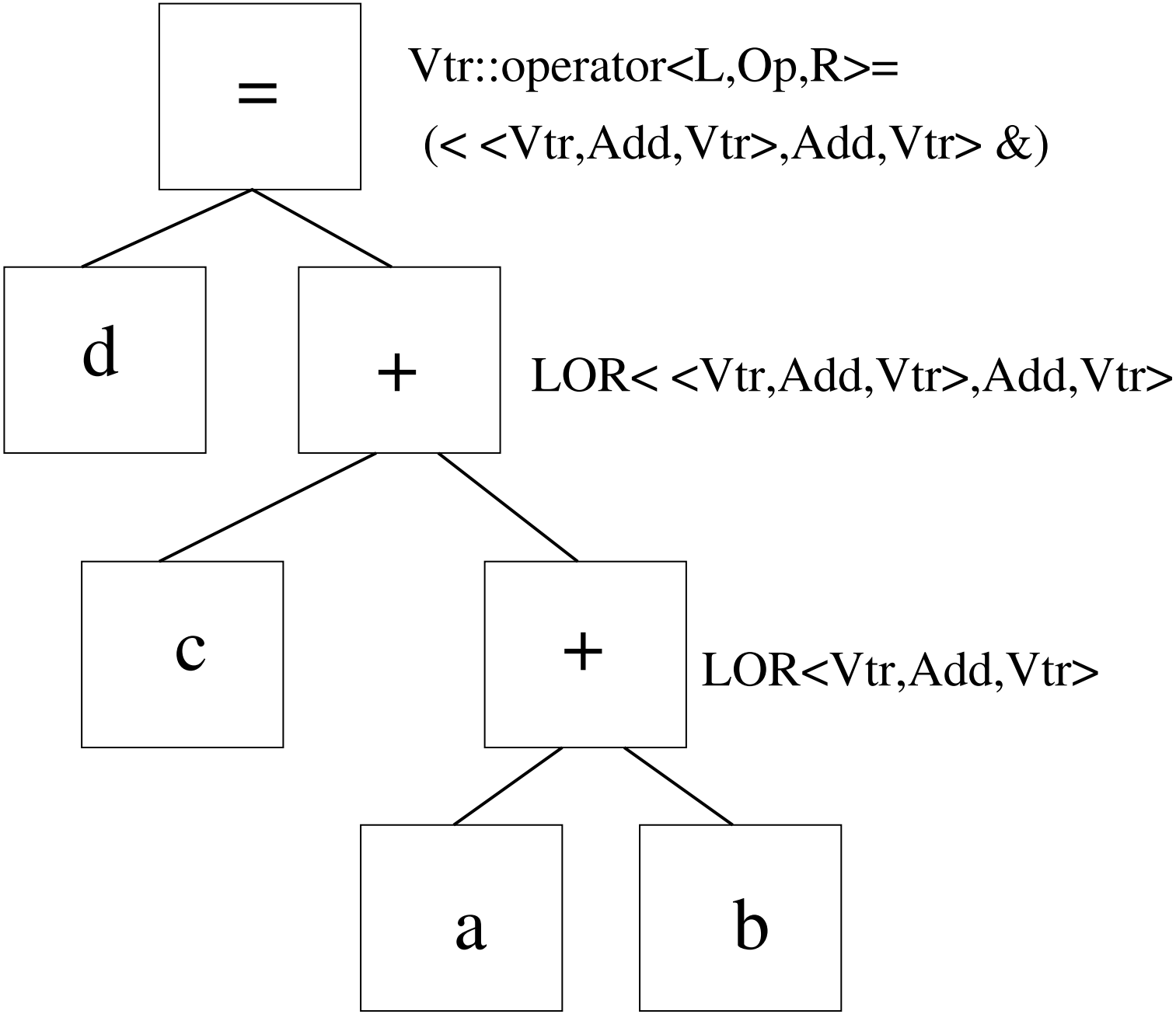
```
d=LOR<Vtr<T>,Add,Vtr<T> >(a,b)+c
d=LOR<LOR<Vtr<T>,Add,Vtr<T> >,Add,Vtr<T> >(tmp,c)
```

The loop in the assignment to _data becomes:

```
_data[i]=Add::apply(Add::apply(a[i],b[i]),c[i])
```

Becomes:

```
_data[i]=(a[i]+b[i])+c[i]
```


=
Vtr::operator<L,Op,R>=
(< <Vtr,Add,Vtr>,Add,Vtr> &)
d
+
LOR< <Vtr,Add,Vtr>,Add,Vtr>
c
+
LOR<Vtr,Add,Vtr>
a
b

# Expression templates in practice

A full implementation of expression templates is rather delicate. But the technique is very powerful. It is implemented, for instance, in the Eigen or the Blitz++ libraries for linear algebra, and also used to speed up finite element matrix assembly in the `htpp://www.lifev.org`LifeV library.