Advanced Programming for Scientific Computing (PACS)
Lecture title: Numerical Metaprogramming and Expression Templates

Luca Formaggia

MOX
Dipartimento di Matematica
Politecnico di Milano

A.A. 2022/2023

Numerical metaprogramming: static dot product of arrays

```
template<std::size_t M> struct metaDot
{
template<std::size_t N,typename T>
  static T apply(std::array<T,N>const & a,std::array<T,N> const & b)
  {return a[M-1]*b[M-1] + metaDot<M-1>::apply(a,b);}
};
//! Specialization for the first element.
template<> struct metaDot<1>
{
  template<std::size_t N,typename T>
  static T apply(std::array<T,N>const & a,std::array<T,N> const & b)
  {return a[0]*b[0];}
};
// Operator overloading: the user uses only this!
template<std::size_t N,typename T>
inline T operator*(std::array<T,N>const& a,std::array<T,N> const& b){
return metaDot<N>::apply(a,b);}
```

The main program

```
#include "metadot.hpp"
int main(){
  using std::array;
  array<double,5> a={1.,1.,1.,1.,1.};
  array<double,5> b={1.,2.,3.,4.,5.};
  // Using overloaded operator
  std::cout<<"a*b="<<a*b<<std::endl;
}
```

If optimization is activated, the compiler resolves the operation.
The complete example in
MetaProgramming/MetaDot/metadot.hpp, where you find also a version more C++17 style.
Another interesting example is in
LinearAlgebra/Utilities/mathUtils.hpp.

Expression templates

Lets suppose to have a class to store a Vector of double (it may be std::vector<**double**>

```
class Vector
{
std::vector<double> M_data;
public:
double operator[](size_t i)const
{ return M_data[i];}
double & operator[](size_t i)
{ return M_data[i];}
...
};
```

and we overload some operators

```
Vector operator +(Vector const & a, Vector const & b)
{ Vector tmp.reserve(a.size());
for (size_t i=0u;i<a.size();++i)
   tmp[i]=a[i]+b[i]
return tmp;}
```

Drawbacks of operator overloading

Expression

```
Vector a,b,c,d;
//...
d=a+b+c;
//....
```

It produces 2 temporaries (we may save 1 with move semantic) and executes three loop cycles: one for each sum and one for the assignment. It is more efficient to have a single cycle:

```
for(std::size_t i=0;i<=a.size();++i)
d[i]=a[i]+b[i]+c[i];
```

but it is not a scalable solution: what if we want to express also $\mathbf{d} = \mathbf{a} + 5\mathbf{b} - \mathbf{c}/y$. We need a different loop. After all the semantic of operators is so nice... why abandoning it?

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 and avoiding the creation of large 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 evaluate the expression.

There are different ways of applying expression templates, and the examples are not limited to linear algebra.

We here show an implementation that uses a technique illustrated in *Advanced expression templates programming, J. H ardtlein, C. Pflaum, A. Linke and C. H. Wolters, Comput Visual Sci (2010) 13:59–68*. You find a copy of the paper on `WeBeep` (among others!).

Back to CRTP

This following class is an example of use of CPRT (curiosly recursive template pattern). Indeed any class that encapsulates an expression should derive from it using

```
class AnyExpression: public Expr<AnyExpression>
{...};
```

The prerequisite is that the derived class should contain the address operator [] and the size() method.

In the example in ExpressionTemplates/Algebra/expressionWrapper.hpp I use a conversion by cast to extract the object of derived type. In the slides I use the method asDerived(), since it is easier to understand.

A class wrapping any expression

```
template <class E>
struct Expr
{
  E const& asDerived() const {
   return static_cast<E const&>(*this);}
  E  & asDerived(){
   return static_cast<E &>(*this);}
  //Interrogates the size of the wrapped expression
  std::size_t size() const {return const this->asDerived.size();}
  // Addressing operator (delegated to derived)
  double operator [](std::size_t i) const {
    this->asDerived.operator[](i);
  }
};
```

An expression for binary operators

```
template<class LO, class RO, class OP>
class
BinaryOperator : public Expr<BinaryOperator<LO,RO,OP> >
{
public:
  BinaryOperator(LO const & l, RO const &r):M_lo(l),M_ro(r){};
  // Applies operation on operands
  double operator [](std::size_t i) const {
    return OP()(M_lo[i],M_ro[i]);}
  std::size_t size()const {return M_lo.size();}
private:
  LO const & M_lo; //a reference!
  RO const & M_ro;
};
```

This class encapsulate a binary operation. It stores references to the operand and uses the operator OP in the addressing operator. This way [] $\rightarrow$ M_lo[i] OP M_ro[i]

A note

I have chosen to implement the operators as function objects since they must have the semantic of types as they appear as template parameters. However, since the operators are clearly stateless, I could have instead defined a static function member apply(). In this case, the statement

```
return OP{}(M_lo[i],M_ro[i]);}
```

becomes

```
return OP::apply(M_lo[i],M_ro[i]);}
```

and do not need to create an object of type OP with the default constructor. However, there is no practical advantage. We will see that OP has no data members, and the creation of a stateless object is a no-operation (no code is generated for the creation).

Basic arithmetic operators

```
  struct Add{
    double operator ()(double i, double j) const{return i+j;}
  };
  //! The basic Multiplication
  struct Multiply{
    double operator()(double i, double j)const{return i*j;}
  };
...
```

They express the operators.

User level operators

```
// ! Useful alias
template <class LO, class RO>
using AddExpr<LO,RO>=BinaryOperator<LO,RO,Add>
template <class LO, class RO>
using MultExpr<LO,RO>=BinaryOperator<LO,RO,Multiply>

//! Addition of expression
template <class LO, class RO>
inline AddExpr<LO,RO> operator +(LO const & l, RO const & r){
return AddExpr<LO,RO>(l,r);}

//! Multiplication of expressions
template <class LO, class RO>
inline MultExpr<LO,RO> operator *(LO const & l, RO const & r){
return MultExpr<LO,RO>(l,r);}
..
```

They return the object that encapsulates the operation and the operands.

The final touch

We want now to apply the expression template to a class that holds a vector. For simplicity we consider just vectors of doubles (but you may generalize it by adding a template parameter). The key are the copy-constructor and copy-assignment that take an expression as parameter:

```
class Vector : public Expr<Vector>
{
  std::vector<double> M_data;
public:
...
  template <class T>
  Vector(const Expr<T> & e);
  template <class T>
  Vector & operator = (const Expr<T> & e);
  ...
```

The constructor

```
template <class T>
Vector(const Expr<T> & e):
{
  const T & et=e.asDerived();
  M_data.reserve(et.size());
  for (auto i=0u; i<et.size();++i)
       M_data.emplace_back(et[i]);
}
```

It assigns the result of the expression to the newly created Vector

The assignment operator

```
template <class T>
Vector & operator = (const Expr<T> & e)
{
  const T & et=e.asDerived();
  M_data.resize(et.size());
  for (auto i=0u; i<et.size();++i) M_data[i]=et[i];
  return *this;
}
```

It assigns the result of the expression to the existing Vector

Expression templates at work

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

```
d=AddExpr<Vector,Vector>(a,b)+c
d=AddExpr<AddExpr<Vector,Vector>,Vector> (tmp,c)
```

The loop in the assignment to M_data becomes:

```
M_data[i]=AddExpr(AddExpr(a[i],b[i]),c[i])
```

Becomes:

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

Expression templates at work

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

```
d=AddExpr<Vector,Vector>(a,b)+c
d=AddExpr<AddExpr<Vector,Vector>,Vector> (tmp,c)
```

The loop in the assignment to M_data becomes:

```
M_data[i]=AddExpr(AddExpr(a[i],b[i]),c[i])
```

Becomes:

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

Expression templates at work

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

```
d=AddExpr<Vector,Vector>(a,b)+c
d=AddExpr<AddExpr<Vector,Vector>,Vector> (tmp,c)
```

The loop in the assignment to M_data becomes:

```
M_data[i]=AddExpr(AddExpr(a[i],b[i]),c[i])
```

Becomes:

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

Expression templates at work

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

```
d=AddExpr<Vector,Vector>(a,b)+c
d=AddExpr<AddExpr<Vector,Vector>,Vector> (tmp,c)
```

The loop in the assignment to `M_data` becomes:

```
M_data[i]=AddExpr(AddExpr(a[i],b[i]),c[i])
```

Becomes:

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

Expression templates at work

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

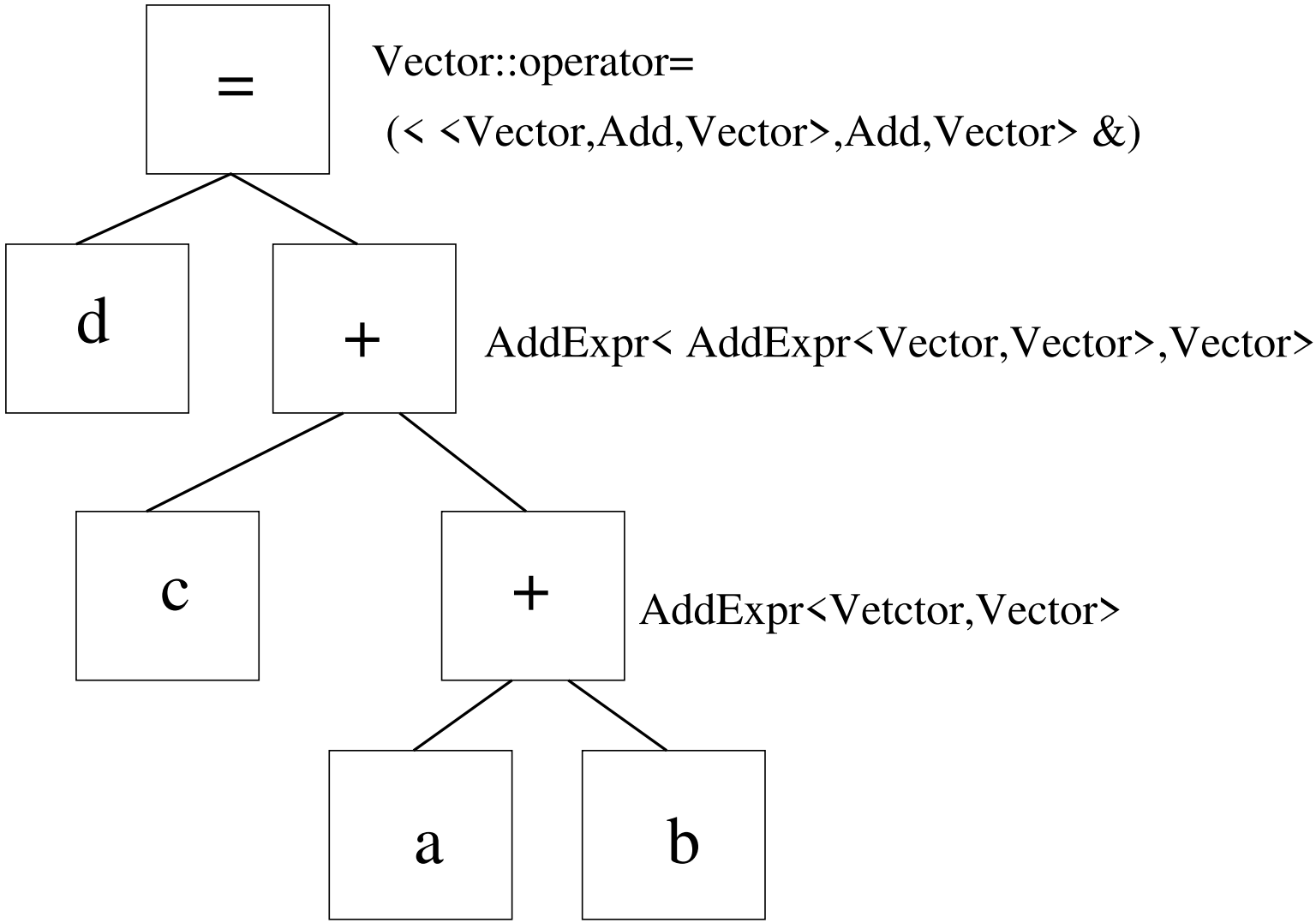
```
d=AddExpr<Vector,Vector>(a,b)+c
d=AddExpr<AddExpr<Vector,Vector>,Vector> (tmp,c)
```

The loop in the assignment to M_data becomes:

```
M_data[i]=AddExpr(AddExpr(a[i],b[i]),c[i])
```

Becomes:

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


=
Vector::operator=
(< <Vector,Add,Vector>,Add,Vector> &)
d
+
AddExpr< AddExpr<Vector,Vector>,Vector>
c
+
AddExpr<Vetctor,Vector>
a
b

Expression templates in practice

You find the complete example in ExpressionTemplates/Algebra/operators.hpp and the other file sin that directory. You may extend the implementation to include also matrix operation... yet a full implementation of expression templates is rather delicate.

The technique is very powerful and is implemented, for instance, in the Eigen or the Armadillo++ libraries for linear algebra.

Another example (you need cmake to compile it) is in ExpressionTemplates/Integral/. It uses expression templates for numerical quadrature.