NGsolve::What's under the hood?

C++ tricks and tips

Jay Gopalakrishnan

Portland State University

Winter 2015

Download accompanying codes for these notes from from here.

Contents

A simple factorial function

To begin, consider this simple implementation of factorial that computes factorial at *run time*.

```
int factorial (int n) {
  return n == 0 ? 1 : n * factorial(n - 1);
}

int main() {

  factorial(4); // Ok with just this line, but not with next.

  static_assert(factorial(4)==24,"Dont␣know␣4!" ); // Error:
  // compiles only if factorial(4) is known at compile time.
}
```

Factorial in Template Meta Programming

An alternate implementation via templates offers a standard example of *Template Meta Programming (TMP):*

```
template <int N>  struct factorial {
  enum {value = N*factorial<N-1>::value};
};
template <> struct factorial<0> {enum {value = 1}; };

int main() { // Compiles because 4! computed at compile time

  static_assert(factorial<4>::value==24,"Can't␣have␣error");
}
```

This computes factorial at *compile time*!

Note: In a statement like `enum {name = constant_expression}`, the compiler needs to evaluate `constant_expression` at compile time. So `enum` *was* a way to compute compile-time constants. That *was* before C++11 came along . . .

Factorial in Template Meta Programming

An alternate implementation via templates offers a standard example of *Template Meta Programming (TMP):*

```
template <int N>  struct factorial {
  enum {value = N*factorial<N-1>::value};
};
template <> struct factorial<0> {enum {value = 1}; };

int main() { // Compiles because 4! computed at compile time

  static_assert(factorial<4>::value==24,"Can't␣have␣error");
}
```

This computes factorial at *compile time*!

There are many more such metaprogramming techniques, which allow one to move some run-time tasks to the compiler. Such techniques can compute types, constants, and even complete functions.

Change is in the air

Since C++11, some TMP codes can be replaced by “regular” codes.

```
constexpr int factorial(int N) {        // the new way to compute
  return N==0 ? 1 : N*factorial(N-1); // at compile time
}

int main() {
    static_assert(factorial(4)==24,"Use␣a␣C++11␣compiler");
}
```

This also computes factorial at *compile time* because the new “`constexpr`” declares a value to be computable at compile time.

Static polymorphism

The previous example also serves to review compile-time (or static) polymorphism, e.g.:

```
template <int N>  struct factorial {
  enum {value = N*factorial<N-1>::value};
};
template <> struct factorial<0> {enum {value = 1}; };
```

This function `factorial` changes its meaning depending on its template parameter (e.g., `N=0` or `N=2`). No run-time checking is needed.

The compiler compiles every needed instantiation of templated functions at compile time and inlines them to optimize.

Dynamic polymorphism

Run-time (or dynamic) polymorphism allows programmers to declare functions in a base class that can be redefined in each derived class.

```
class NumProc //..NGSolve's base class for numerical procedures

  // member "Do" has meaning only in derived classes
  virtual void Do(LocalHeap & lh) = 0;
};
```

```
class NumProcCalcError : public NumProc { // Recall from
  virtual void Do (LocalHeap & lh) {       // previous exercise
     // First, solve the problem
     // Next, compute errors, etc
  }
};
```

If `np` is a pointer to a NumProc object, then `np->Do(lh)` selects *at run-time*, `NumProcCalcError::Do`, or `NumProcBVP::Do`, or other derived class "`Do`" functions, depending on the type of `np`.

Next

1. Very quick intro to TMP
2. Review of static & dynamic polymorphism
3. CRTP idiom or Barton-Nackman trick
4. Traits and type promotion
5. Naive vector class
6. Expression templates or ET
7. Use NGsolve's facilities
8. Did C++14 kill the beloved ET?

When are virtual functions evil?

```
class Matrix {
 public: virtual double operator()(int i, int j) =0;
};

class SymmetricMatrix : public Matrix {
 public: virtual double operator()(int i, int j);
};

class UpperTriangularMatrix : public Matrix {
 public: virtual double operator()(int i, int j);
};
```

The overhead (checking which `A(i,j)` to call) will ruin the performance of any algorithm using the above class! [Veldhuizen]

Virtual functions are highly recommended for big functions not called very often. But in situations like the above, dynamic polymorphism is not recommended. We will now study a static polymorphism alternative.

CRTP idiom or the “Barton-Nackman trick”

The *Curiously Recurring Template Pattern (CRTP)* is a C++ idiom that **specializes a base class using the derived class** as a template.

```
// declare base class with derived class as template
template <class Derived> class Base  // ...

// seemingly recursive definition of a derived class
class SomeDerivedClass : public Base<SomeDerivedClass> //..
```

The CRTP idiom applied to the matrix problem

```
template<class Derived> class Matrix {  // base class
 public:
  Derived& FromDerived() {return static_cast<Derived&>(*this);}

  // At compile time, send base(i,j) to derived(i,j)
  double operator()(int i,int j) {return FromDerived()(i,j);}
};

// Inherit from base using derived:     (mindbending!)

class SymmetricMatrix : public Matrix<SymmetricMatrix> {
 public: double operator()(int i, int j);
};
class UpperTriangularMat : public Matrix<UpperTriangularMat> {
 public: double operator()(int i, int j);
};
```

There is no virtual function overhead in getting `A(i,j)` this way!

Next

1. Very quick intro to TMP
2. Review of static & dynamic polymorphism
3. CRTP idiom or Barton-Nackman trick
4. Traits and type promotion
5. Naive vector class
6. Expression templates or ET
7. Use NGsolve's facilities
8. Did C++14 kill the beloved ET?

Traits

Traits map a type into a list of its properties. We often use it as maps from template arguments into things inside classes.

To illustrate use of traits, we use this contrived example:

```
#include<iostream>                    // File: traits0.cpp

template <class T>  T Mean(const T & a, const T & b)
{ return (a+b)/2.0;}

int main() {

  double a = 1.0, b = 2.0;
  int j = 1, k = 2;
  std::cout << Mean(a,b) << std::endl;
  std::cout << Mean(j,k) << std::endl;
}
```

Compile (`make traits`) and run (`./traits0`).

Traits

Traits map a type into a list of its properties. We often use it as maps from template arguments into things inside classes.

To illustrate use of traits, we use this contrived example:

```
#include<iostream>                    // File: traits0.cpp

template <class T>  T Mean(const T & a, const T & b)
{ return (a+b)/2.0;}

int main() {

  double a = 1.0, b = 2.0;
  int j = 1, k = 2;
  std::cout << Mean(a,b) << std::endl;
  std::cout << Mean(j,k) << std::endl;
}
```

The problem is that `Mean` *of two integers may be truncated incorrectly.*

Traits for type promotion

We solve this problem using the following traits map: $\begin{cases} int \mapsto double \\ double \mapsto double \end{cases}$

```
                          // File traits1.cpp
template<class T> struct MeanTrait  {using resultT = T; };
                          // int    ->   double:
template<> struct MeanTrait<int>    {using resultT = double;};
                          // double ->   double:
template<> struct MeanTrait<double> {using resultT = double;};

// change Mean's output to the result of the traits map
template <class T>  typename MeanTrait<T>::resultT
Mean(const T & a, const T & b) {   return (a+b)/2.0; }
```

This is sometimes called **type promotion**.

Quiz: What if `T=complex<double>`?

Check out related new C++11 facilities in `std::type_traits`!

Next

1. Very quick intro to TMP
2. Review of static & dynamic polymorphism
3. CRTP idiom or Barton-Nackman trick
4. Traits and type promotion
5. Naive vector class
6. Expression templates or ET
7. Use NGsolve's facilities
8. Did C++14 kill the beloved ET?

Write a simple vector class in C++98

```
template <typename T = double>  class MyFirstVec {
private:
  int sz;
  T * el;

public:
  MyFirstVec (int s) { sz = s; el = new T[s]; } // constructor
  ~MyFirstVec() { delete [] el; }               // destructor
  int Size () const { return sz; }
  T & operator() (int i) { return  el[i]; }     // i'th element
  const T & operator() (int i) const { return el[i]; }

  MyFirstVec<T> & operator=(const MyFirstVec<T> & v)
    { /* copy contents of v into this vector */ }
  MyFirstVec<T> & operator=(const T & t )
    { /* fill all elements with value t */ }
};
```

Quiz: Overload +, *, etc

```
template<typename T>
MyFirstVec<T> operator+ (const MyFirstVec<T> & x,
                         const MyFirstVec<T> & y) {
  MyFirstVec<T> output(x.Size());
  for (int i = 0; i < x.Size(); i++) output(i) = x(i) + y(i);
  return output;
}
template<typename T>
MyFirstVec<T> operator* (double a, const MyFirstVec<T> & x)
{ /* fill in */}
template<typename T>
ostream & operator<<(ostream & os, const MyFirstVec<T>& w)
{ /* fill in */}
```

- This would allow us to say “`z = x+ 1000 * y;`” for objects of type `MyFirstVec<double> x,y,z;`
- Open `la0_exercise.hpp`, complete it, and save it to `la0.hpp`

Using the vector class: `eg0.cpp`

```cpp
#include "la0.hpp"
#include <cstdlib>               // for atoi (alpha2integer)
#include <chrono>                // for timing
using namespace std;
                                 //argc=num cmd line args
int main(int argc, char *argv[]){//argv[1]=cmd line string arg

  chrono::time_point<std::chrono::system_clock> start, end;
  start = chrono::system_clock::now();
  int N = atoi( argv[1] );       // convert string arg to int

  MyFirstVec<double> x(N), y(N), z(N);
  x = 2.0;
  y = 3.0;
  z = 1.1* ( 1.1* (3.0 * x + y )  + x ) + y;

  end = chrono::system_clock::now();
  chrono::duration<double> elapsed = end - start;
  cout << "elapsed␣time:␣" << elapsed.count() << "s\n";
}
```

Compare with NGsolve's vector class

1. Write another driver `eg1.cpp`, by replacing `MyFirstVec<double>` with `ngbla::Vector<double>` in the previous file.

2. Compare the performance of MyFirstVec and Vector using:

   ```
   g++ eg0.cpp -o eg0
   ```

   ```
   ngscxx eg1.cpp -o eg1 -lngstd -lngbla
   ```

   (or compile using the given Makefile) and then execute

   `./eg0 10000000` and `./eg1 10000000`

Is there a difference in the reported timing?
Why?

The overhead

The difference is speed arises because of the memory allocations in the naive implementation, e.g.,

- `x+y` requires a temporary vector of the size of `x` to be created (see definition of `operator+`).
- Similarly, `operator*` requires another temporary vector.
- Thus, in `eg0.cpp`, to implement `z = 1.1*(1.1* (3.0 * x + y ) + x ) + y`, the compiler must allocate 6 temporary vectors of the size of `x`.

Is this an "unavoidable cost of the object oriented nature of C++"?

The overhead

The difference is speed arises because of the memory allocations in the naive implementation, e.g.,

- `x+y` requires a temporary vector of the size of `x` to be created (see definition of `operator+`).
- Similarly, `operator*` requires another temporary vector.
- Thus, in `eg0.cpp`, to implement `z = 1.1*(1.1* (3.0 * x + y ) + x ) + y`, the compiler must allocate 6 temporary vectors of the size of `x`.

Is this an "unavoidable cost of the object oriented nature of C++"?

Bah! No! We can overcome this problem with expert techniques.

"C++ is expert friendly." –Stroustrup

Next

1. Very quick intro to TMP
2. Review of static & dynamic polymorphism
3. CRTP idiom or Barton-Nackman trick
4. Traits and type promotion
5. Naive vector class
6. Expression templates or ET
7. Use NGsolve's facilities
8. Did C++14 kill the beloved ET?

Don’t evaluate until needed

```
template<typename T> MyFirstVec<T>     // RECALL from la0.hpp
operator+ (const MyFirstVec<T> & x, const MyFirstVec<T> & y) {

  MyFirstVec<T> output(x.Size());
  // ... add ...
  return output;
}
```

Idea for improvement:

- The `operator+` is inefficient, since the memory to store the result is not available (so must be created) when calling `x+y`.
- So, how about *returning an object representing the sum* of two vectors, a representation without memory overhead that knows how to evaluate the sum upon demand?

Expression Templates (ET) and Parse trees

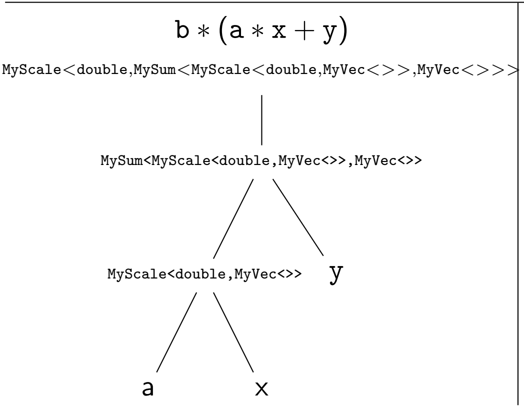
For objects `x,y,z` of type `MyVec<>`, when the compiler sees `z = x + y`,

- we want it to first build an "expression" representing the sum, `MySum< MyVec<>, MyVec<> >(x,y)`, [Veldhuizen]
- but evaluate it only when `z.operator=` is called. [Schöberl]

Expression Templates (ET) and Parse trees

For objects `x,y,z` of type `MyVec<>`, when the compiler sees `z = x + y`,

- we want it to first build an "expression" representing the sum, `MySum< MyVec<>, MyVec<> >(x,y)`, [Veldhuizen]
- but evaluate it only when `z.operator=` is called. [Schöberl]



For even moderately complex expressions, the parse tree gets painfully complex, e.g., see the case of `z=b*(a*x + y)`.

But as long as the pain is only inflicted on the compiler (and not on the programmer), its ok.

A minimalist ET

```
template <class Tx, class Ty>  class MySum {
  const Tx & x;
  const Ty & y;
public:
  MySum (const Tx & xx, const Ty & yy) : x(xx), y(yy) {}
  double operator() (int i) const { return x(i) + y(i); }
};

template <class Tx, class Ty>   MySum<Tx,Ty>
operator+ (const Tx & x, const Ty & y)
{  return MySum<Tx,Ty>(x,y); }
```

```
template <typename T = double> class MyVec {// Like MyFirstVec
public:                                     // :
  template<class A, class B> MyVec<T> &     // :
  operator=(const MySum<A,B> & s) {         // add this
    for (int i = 0; i < sz; i++) el[i] = s(i);
    return *this;
  }
};
```

A minimalist ET

```
template <class Tx, class Ty>  class MySum {
  const Tx & x;
  const Ty & y;
public:
  MySum (const Tx & xx, const Ty & yy) : x(xx), y(yy) {}
  double operator() (int i) const { return x(i) + y(i); }
};

template <class Tx, class Ty>   MySum<Tx,Ty>
operator+ (const Tx & x, const Ty & y)
{  return MySum<Tx,Ty>(x,y); }
```

Take a look at the implementation in `la1.hpp`. Type

```
make expr1  &&  ./expr1 10000000
```

Timing should now be comparable to the implementation using NGsolve's vector (`./eg1 10000000`) !

A minimalist ET

```
template <class Tx, class Ty>  class MySum {
  const Tx & x;
  const Ty & y;
public:
  MySum (const Tx & xx, const Ty & yy) : x(xx), y(yy) {}
  double operator() (int i) const { return x(i) + y(i); }
};

template <class Tx, class Ty>   MySum<Tx,Ty>
operator+ (const Tx & x, const Ty & y)
{  return MySum<Tx,Ty>(x,y); }
```

But we must fix these problems:

- The return type of `MySum::operator()(i)` must be general (otherwise this will only work for `MyVec<double>`).
- A line like this `z * "garbage" + "out";` will compile, because we have overloaded `operator+` for everything! Vector * String = ?!

A base class for expressions

- To solve the first problem, use traits and type promotion. (Exercise!)
- To solve the second problem, we inherit `MySum`, `MyScale`, etc. from a base class `MyExpression`.
- The operators *, + etc are then overloaded only for `MyExpression` objects.
- The class `MyExpression` must know how to evaluate a vector expression represented by a derived class object. If we implement this by virtual functions, things will be slow. We want to implement this by static polymorphism.
- So we use the CRTP idiom.

Overview of the implementation

```
template <typename Tderived> class MyExpression  {
  // base class ...
};
template <class Tx, class Ty>                         //derived
class MySum : public MyExpression< MySum<Tx,Ty> > {//with CRTP
  // ... operator()(i) gives i-th component of sum
};
template <class Tx, class Ty>  MySum<Tx,Ty>
operator+(const MyExpression<Tx>& x,const MyExpression<Ty>& y){
  // ... simply returns a MySum object
}

template <typename T>
class MyVec : public MyExpression< MyVec<T> >  {
public:
  template<class A> MyVec<T> &
  operator=(const MyExpression<A> & E ) {
    // ...  instead of E(i), use derived class's operator(i)
  }
};
```

Quiz

- The file `la2_exercise.hpp` provides an implementation of `MySum`. Complete the file by providing an implementation of `MyScale` class and rename it to `la2.hpp`.

- Compile using `make expr2`. Run `./expr2 10000000` and check timings as before.

- Implement type promotion using traits and revise the type definition of `DataTrait`, the return type of `operator()(int i)`. Your code should work (at least) for vectors of `int`, `double`, and `complex<double>`.

  Assuming your new vector class is in the file `la3.hpp`, the driver routine `expr3.cpp` (which also checks if you can add vectors of `int` and `double` etc.) should compile (`make expr3`) and run (`./expr3 10000000`) smoothly.

ET: Postscript

- Expression Templates were considered a “C++ gem” and found its way in several libraries providing mathematical vector classes. However, it is time consuming and difficult to maintain. My purpose with the previous few slides was to explain just enough of ET so that you can digest the source code in `ngsolve/basiclinalg`.

- Next, **forget what you wrote and use what is in NGsolve.**

- NGsolve’s linear algebra facilities have full-featured matrices and vectors integrated into expression templates. It has classes with and without their own memory management. **Exercise: Learn about these from** `ngsolve-code/programming_demos/demo_bla.cpp`.

- ET, in the form we saw, is now obsolete, as we shall see . . .

Next

1. Very quick intro to TMP
2. Review of static & dynamic polymorphism
3. CRTP idiom or Barton-Nackman trick
4. Traits and type promotion
5. Naive vector class
6. Expression templates or ET
7. Use NGsolve's facilities
8. Did C++14 kill the beloved ET?

Move constructor in C++11

Add these members to the old “naive” vector class in `la0.hpp`:

```
MyVec(MyVec<T> && v)   {                    // Move constructor
 sz = v.sz;   swap(el,v.el);
 v.sz=0;      v.el = nullptr;
}
MyVec<T> & operator=(MyVec<T> && v)   { // Move assignment
 sz = v.sz;   swap(el,v.el);
 return *this;
}
```

Or better yet, just let the compiler add it for you:

```
MyVec<T> & operator=(MyVec<T> && v)=default;// Move assignment
MyVec(MyVec<T> &&) = default;               // Move constructor
```

Just adding these two lines changes the semantics of the old `operator+`.

Move semantics

```
template<typename T>                    // the same old naive operator+
MyVec<T> operator+ (const MyVec<T> & x, const MyVec<T> & y) {
  MyVec<T> output(x.Size());
  for (int i = 0; i < x.Size(); i++) output(i) = x(i) + y(i);
  return output;
}                                       // File: la4.cpp
```

Compile: `make la4`. Run: `./la4 10000000`.

```
MyVec<double> x(N), y(N);
x = 2.0; y = 3.0;                       // new called thrice (for x,y,z)
MyVec<double> zz = x+y;                 // so no temporaries were made!
```

This is because of the new language rules:

- `x+y` provides an `rvalue reference` (to move assignment).
- `return output` invokes move operations (instead of copy).

Move semantics

```
template<typename T>                    // the same old naive operator+
MyVec<T> operator+ (const MyVec<T> & x, const MyVec<T> & y) {
  MyVec<T> output(x.Size());
  for (int i = 0; i < x.Size(); i++) output(i) = x(i) + y(i);
  return output;
}                                       // File: la4.cpp
```

Compile: `make la4`. Run: `./la4 10000000`.

```
MyVec<double> x(N), y(N);
x = 2.0; y = 3.0;                       // new called thrice (for x,y,z)
MyVec<double> zz = x+y;                 // so no temporaries were made!
```

This is because of the new language rules:

- `x+y` provides an `rvalue reference` (to move assignment).
- `return output` invokes move operations (instead of copy).

But this doesn't solve the problem for more complex expressions like

```
z = 1.1* ( 1.1* (3.0 * x + y ) + x ) + y;
```

Lambda expressions

Idea: Instead of defining classes like `MySum`, `MyScale` etc., capture the expression as a lambda function (a new C++11 feature), e.g.,

```
template <typename Tx, typename Ty> inline auto
operator+(const MyVec<Tx> & x, const MyVec<Ty> & y) {
  return  ( [&](int i){ return x(i) + y(i);} );
}                                       // need more ideas
```

But this would then need us to define further arithmetic for the returned lambdas. So we create an expression class like before:

```
template <typename F> class MyExpression  {
  F f;
public:
  MyExpression (F func) : f(func) {;}
  auto  operator() (int i) { return f(i); }
  const auto operator() (int i) const { return f(i); }
};
```

All it does is to hold and evaluate a lambda function `f`.

Avoiding the type of lambda

With the new `MyExpression` class we can try:

```
template <typename Tx, typename Ty, typename F> inline auto
operator+(const MyVec<Tx> & x, const MyVec<Ty> & y) {
  return MyExpression<F>( [&](int i){ return x(i) + y(i);} );
}                                                 // doesn't work !
```

But unfortunately, the compiler doesn't have enough information to deduce `F` when seeing `z=x+y`.

So we introduce a simple expression instantiator as a function template:

```
template <typename F> inline
MyExpression<F> ExprInstantiator(F f) {
  return MyExpression<F> (f);
}
```

The death of ET?

Now arithmetic operators can return through the expression instantiator:

```
template <typename Tx, typename Ty> inline auto
operator+(const MyVec<Tx> & x, const MyVec<Ty> & y) {
  return ExprInstantiator( [&](int i){ return x(i) + y(i);} );
}

template <typename Ta, typename Fx> inline auto
operator*(Ta a, const MyExpression<Fx> & x) {
  return ExprInstantiator( [&](int i){ return a*x(i);} );
}
```

There is no need to implement type promotion because the "`auto`" return types of the lambda's can already deduce the return type of arithmetic on simple data types. (We use C++14 to avoid trailing return types etc.)

A minimalist implementation

- Take a look at `la5.cpp`. (This file should look simpler than the ET we implemented previously in `la3.cpp`.)
- Compile using `make la5`. You will need `g++-4.9` or higher in order to pass the flag `-std=c++14`.
- When running “`./la5 10000000`” you should observe timings similar to your previous ET implementation or similar to NGsolve timings.

So is ET really dead?

The reincarnation of ET

- Expression templates are not really dead. The last word on ET is likely yet to be written.
- In fact, the lambda function technique we used is simply the reincarnation of ET in the brave new world of C++14.
- The idea of building parse trees, to delay execution until memory to store the result is available, lives on.
- Free advice: Unless it is your business to write mathematical matrix/vector packages, continue to use a good existing ET implementation like NGsolve.

- And finally, let's now focus on coding finite elements!