Expression Templates - Past, Present, Future

numscale
Unlocked software performance

Joel Falcou

NumScale – LRI

CppCon 2015

What is that ? C++ ??

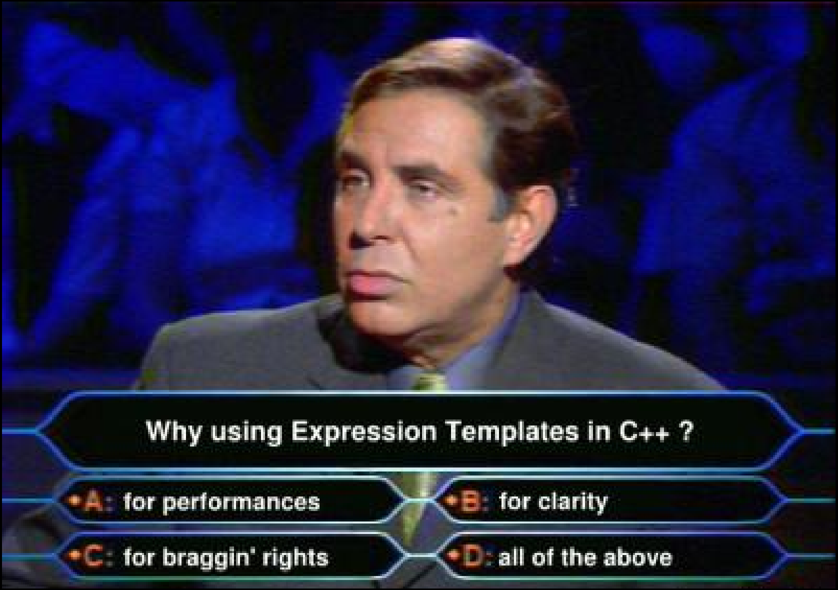
github.com/KKostya/SpiritKaleidoscope

```
template <typename Iterator>
kaleidoscope<Iterator>::kaleidoscope() : kaleidoscope::base_type(program,"program")
{
    identifier = qi::lexeme[char_("a-zA-Z") >> *char_("0-9a-zA-Z")];
    proto      = identifier > '(' >> -(identifier % ',') > ')';
    func       = proto > arithm;
    program    =   lit("extern") > proto
               | lit("def")    > func
               | arithm;

    // Error handling
    proto.name("prototype");
    func.name("function definition");
    program.name("toplevel");

    on_error<fail>
        (
         program
         , std::cout
         << ph::val("Error! Expecting ")
         << _4                                   // what failed?
         << ph::val(" here: \"")
         << ph::construct<std::string>(_3, _2)   // iterators to error-pos, end
         << ph::val("\"")
         << std::endl
        );
}
```

Expression Templates ? Really ??

Oh the infamy

- Expression Templates are 20 years old
- Yet their acceptance as a valid idiom is flacky
- Too many false assumptions
- Too many technicalities


Why using Expression Templates in C++ ?
A: for performances
B: for clarity
C: for braggin' rights
D: all of the above

Expression Templates ? Really !

Objectives

- Explain Expression Templates from the ground up
- Show they are only a clever stack of usual techniques
- See how they can be generalized: Boost.Proto
- Explore post C++11/14 addenda
- Make ETs a first class citizen of the C++ language

Basic Principles
or how does it work anyway ?

A joint discovery

Todd Veldhuizen, 1994

- Intern at RogueWave Oregon
- “Discover” Expression Templates while optimizing image processing
- Technical report & C++ Report article in 94-95

David Vandevoorde, 1995

- Independant discovery
- Work or a `valarray` implementation
- Idea spread through the “Computers in Physics” journal in 96

The original question

David Hoadley 3/17/95

Consider the following code. I have created a matrix class for complex numbers, called DCMatrix, and an operator* function:

```
int main()
{
   DCMatrix A(2,3, complex(0.0,0.0));
   DCMatrix B(3,4,complex(1.0,0.0));
   DCMatrix C(2,4);
//  fill A and B with stuff
   C = A * B;
   return (0);
};
```

If I trace its execution, I see:

```
constructor --A(2,3)
constructor --B(3,4)
constructor --C(2,4)

operator*
constructor --Result(2,4)    (from declaration within operator*)
copy constructor --anon(2,4) (values copied from Result to new anon)
destructor  --Result
operator=                    (copies values from anon to C)
destructor  --anon

destructor  --C
destructor  --B
destructor  --A
```

Result is a temporary matrix which I would rather not have, but I could not see how to write operator* without it. anon is another temporary matrix, which I didn't declare

The original answer

David Vandevoorde 3/17/95

In article <3kaqnn$g...@aggedor.rmit.EDU.AU>,
- show quoted text -

Yes, but it's not so easy to make a general robust code that achieves
this.

>If not, which I fear, can I at least avoid the creation of one of them
>(presumably anon)?
>
>Thanks for any help,
>David.

I would suggest you take a look at two packages.

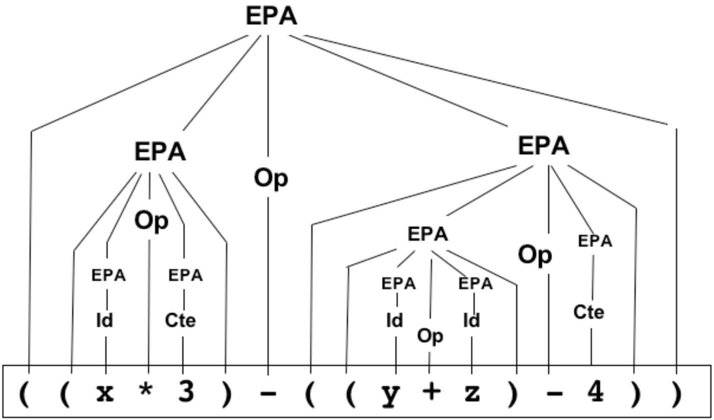
The first one is "NewMat" ("newmat08" is the latest release, I believe)
by Robert Davies (rob...@kauri.vuw.ac.nz). You may get it by ftp from
plaza.aarnet.edu.au in /usenet/comp.sources.misc/volume47/newmat08.
It uses a dynamic expression analysis technique to handle this problem.
For large objects where operations are not only "elementwise" (e.g.,
matrix addition is an elementwise operation, whereas matrix multiplication
is not) this might well be the best way to go.

The second package is my own valarray<Troy> (get it by anonymous ftp from
ftp.cs.rpi.edu in pub/vandevod/Valarray). It is _not_ a matrix library,
but an implementation of a numerical array modeled after the specs for a
similar array included in the current working paper of the ISO committee.
It makes heavy use of class and function templates to avoid the generation
of temporary arrays altogether (but the supported operations are "element-
wise" which makes things a whole lot simpler). It comes with some technical
discussion that reviews other alternatives.

A Compiler's Perspective

```
( ( x * 3 ) - ( ( y + z ) - 4 ) )
```

A Compiler's Perspective

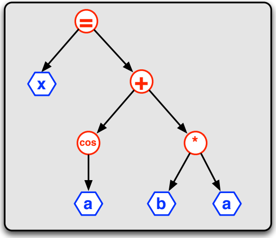


A Compiler's Perspective

```
matrix x(h,w),a(h,w),b(h,w);

x = cos(a) + (b*a);
```

```
expr<assign
    ,expr<matrix&>
    ,expr<plus
         , expr<cos
                ,expr<matrix&>
                >
         , expr<multiplies
                ,expr<matrix&>
                ,expr<matrix&>
                >
         >(x,a,b);
```

```
Arbitrary Transforms applied
     on the meta-AST
```




```
#pragma omp parallel for
for(int j=0;j<h;++j)
{
  for(int i=0;i<w;++i)
  {
    x(j,i) = cos(a(j,i))
           + ( b(j,i)
             * a(j,i)
           );
  }
}
```

From AST to C++

Storing expressions

- Need to store the operator
- Need to store the sub-expressions
- Need to traverse them to perform work

Capturing expressions

- Operators and functions are overloaded
- They don’t compute anything
- They return a representation of the local AST

From AST to C++ - expr

```cpp
template<class Tag, class Child>
struct unary_expr
{
  explicit unary_expr(const Child& p) : c(p) {}

  auto operator()() const { return op(c()); }

  private:
  Child c;
  Tag op;
};
```

From AST to C++ - expr

```
template<class Tag, class Child0, class Child1>
struct binary_expr
{
  binary_expr(const Child0& p0,const Child1& p1)
           :  c0(p0), c1(p1) {}

  auto operator()() const
  {
    return op(c0(),c1());
  }

  private:
  Child0 c0;  Child1 c1;
  Tag op;
};
```

From AST to C++ - terminal

```
template<class T> struct terminal
{
  explicit terminal(T const& v) : value{v}{}
  auto operator()() const { return value; }

  private:
  T value;
};
```

Duct-taping everything together

Issues

- How to write operators so they work with expressions ?
- Where to put them so they are found correctly ?
- How to extent this whole mess sanely ?

Duct-taping everything together

Issues

- How to write operators so they work with expressions ?
- Where to put them so they are found correctly ?
- How to extent this whole mess sanely ?

Use the Curiously Recursive Template Pattern Luke !

Flashback on the CRTP

```
template <class T>  struct Base
{
  void interface()
  {
    static_cast<T &>(*this).implementation();
  }
};

struct Derived : Base<Derived>
{
  void implementation();
};
```

From AST to C++ - base_expr

```cpp
template <class Derived>
struct base_expr
{
  Derived const& self() const { return static_cast<const Derived&>(*this); }
  auto operator()() const { return self()(); }
};

template<class Tag, class Child0>
struct unary_expr : base_expr<unary_expr<Tag,Child0>>
{ /* ... */ };

template<class Tag, class Child0, class Child1>
struct binary_expr : base_expr<binary_expr<Tag,Child0,Child1>>
{ /* ... */ };

template<class T>
struct terminal : base_expr<terminal<T>>
{ /* ... */ };
```

From AST to C++ - Operators definition

```
struct plus_
{
  template<class T>
  T operator()(T a, T b) const { return a+b; }
};

template<class D1, class D2>
binary_expr<plus_,D1, D2>
operator+( base_expr<D1> const& d1
         , base_expr<D2> const& d2
         )
{
  return {d1.self(), d2.self()};
}
```

From CRTP to Boost.Proto
or modern C++ design’s lovely head

Why Boost.Proto ?

Manual ET are limiting

- Too much boilerplate
- Data and traversal are tied together
- Not easy to wrok with other similar code

Solutions

- Find a generic interface for E.Ts
- Come closer to usual compiler techniques
- Solve the extensions problem

The Rise of DSELs

What are DSLs ?

- DSL = declarative, non-turing complete language
- Solve one problem but solve it right
- Usually interpreted or compiled externally

What are DSELs ?

- Library designed as a DSLs
- Compiled by the regular compiler
- Expressions Templates are a way to define DSELs

Boost.Proto

A Expression Template Toolkit

- Generalize E.T implementation strategy
- Decorrelate tree construction from traversal
- Support arbitrary extensions

A DSEL Compiler Toolkit

- Notion of Grammar
- Notion of Semantic Actions
- Interoperability between DSELs

Tree Generation

```cpp
#include <boost/proto/proto.hpp>

boost::proto::terminal<int>::type x;

int main()
{
  auto u = x*x+x-3/~x;
  boost::proto::display_expr(u);
}
```

Tree Generation

```
minus(
    plus(
        multiplies(
            terminal(0)
          , terminal(0)
        )
      , terminal(0)
    )
  , divides(
        terminal(3)
      , complement(
            terminal(0)
        )
    )
)
```

Tree Validation

```
#include <boost/proto/proto.hpp>

using boost::proto::_;
using boost::proto::or_;
using boost::proto::plus;
using boost::proto::multiplies;
using boost::proto::terminal;

struct epa  : boost::proto::or_< terminal<_>
                               , plus<epa,epa>
                               , multiplies<epa,epa>
                               >
{};

int main()
{
  terminal<int>::type x{2};

  std::cout << boost::proto::matches<decltype(x+x), epa>::value << "\n";
  std::cout << boost::proto::matches<decltype(x/!x), epa>::value << "\n";
}
```

Tree Transform

```cpp
#include <boost/proto/proto.hpp>

using boost::proto::_;
using boost::proto::or_;
using boost::proto::when;
using boost::proto::_value;
using boost::proto::otherwise;
using boost::proto::_default;
using boost::proto::terminal;

struct eval  : boost::proto::or_< when<terminal<_>,_value>
                                , otherwise< _default<eval> >
                                >
{};

int main()
{
  terminal<int>::type x{2};

  eval e;
  std::cout << e(x+x*x+3) << "\n";
}
```

Tree Custom Transform

```
struct custom : boost::proto::callable
{
  using result_type = double;

  template<typename T, typename U>
  result_type operator()(T const& t, U const& u) const
  {
     return (t+u)*100;
  }
};

struct eval  : boost::proto::or_< when<terminal<_>,_value>
                                , when< plus<eval,eval>, custom(eval(_left),eval(_right))
     >
                                , otherwise< _default<eval> >
                                >
{};

int main()
{
  terminal<int>::type x{2};

  eval e;
  std::cout << e(x+x*x+3) << "\n";
}
```

Semantic Wrapping

From tree to expression

- Proto tree are semantic-less
- Semantic comes from the DSEL specs
- Proto Domain ties Semantic to Tree

Basic outline

- Defines a wrapper with desired semantic
- Proto domain wrap tree into this type
- Proto operators keep domain informations around


Which traps should I look for ?
A: auto
B: dandling sub-expressions
C: compilation times
D: all of the above

Automatic type deduction

Problem statement

- auto captures the tree no the value type
- Generic code issues: T is not what you think it is
- Performance issues: redundant evaluation

```
array x(10),y(10);
auto z = x+y*3.f;

// fail
assert(is_same<decltype(z),array>::value);
```

Automatic type deduction

Problem statement

- auto captures the tree no the value type
- Generic code issues: T is not what you think it is
- Performance issues: redundant evaluation

```
template<class T> T bad_double( T const& x )
{
  return x+x;
}

template<class T> auto good_double( T const& x )
{
  return x+x;
}
```

Automatic type deduction

Problem statement

- auto captures the tree no the value type
- Generic code issues: T is not what you think it is
- Performance issues: redundant evaluation

```
template<class T> auto foo( T const& x )
{
  auto d = x+x;
  return d*d;
}
```

Automatic type deduction

Problem statement

- `auto` captures the tree no the value type
- Generic code issues: T is not what you think it is
- Performance issues: redundant evaluation

Solutions

- Manual evaluation requests
- Viral application of `auto`
- Library-side fix, e.g `terminal_of`
- Proposal N4035 (see later)

The Sub-expression issue

Problem Statement

- Optimization of expression storage by reference
- Function returning expression referencing local data
- Potential dandling reference to temporary data

```
template<class T> auto foo( T const& x )
{
  array t = sum(2*x + 3); // allocate memory
  return 2*t - 1/t;        // copy ? ref ?
}
```

Compilation time

Problem Statement

- Small scale DSELs compiles somehow fast
- Longer expression = longer compiles time
- 1st source : symbol names
- 2nd source : transform complexity

Solutions

- K.I.S.S
- Language support for expression capture ?
- C++14 Lambdas (see Shiki later)

Next Level Expressions Templates

`operator auto()`, `lambdas`, oh my !

Expression Templates in C++11/14

Is it still worth it ?

- E.T are primarily seen as performance tools
- But: move semantic, copy elision why bother ?
- Are E.T an outdated technology ?

Answers

- E.T in modern C++ make libraries more intentionnal
- E.T based library extract informations compilers can't
- Optimizations opportunities arise at algorithmic level
- E.T are a way to perform domain specific optimizations

auto and Expression templates

The N4035 proposal

- Provides `operator auto()`
- Changes the semantic of auto for UD types
- Old auto still applicable
- Applicable to E.T and all proxy types

```
class product_expr
{
  public:
  product_expr(const matrix& arg1, const matrix& arg2)
  : arg1(arg1), arg2(arg2) {}

  using auto= matrix;

  private:
  const matrix &arg1, &arg2;
};
```

auto and Expression templates

The N4035 proposal

- Provides `operator auto()`
- Changes the semantic of auto for UD types
- Old auto still applicable
- Applicable to E.T and all proxy types

```
matrix a,b;

// store a matrix
auto z = a*b;

// store a product_expr
explicit auto w = a*b;
```

The shiki library

What is Shiki ?

- Boost.Hana inspired C++14 E.T engine
- Idiomatic rewrite of something close to Boost.Proto
- Rely on most C++14 features

Main assets

- Designed to be easy to use, easy to develop
- Faster compile times by using symbol name compression
- Multiple entry level API

Let’s go radical

Which other languages do E.T ?

- Template Haskell
- MetaOCaml
- Common points : code fragment as first class citizen

```
let a = .<(1 + 2)>.;;
  -> val a : int code = .<(1 + 2)>.

let b = .< 3. * .~a>.;;
  -> val b : double code = .<3. * (1 + 2)>.

let c = .!b;;
  -> val c : double
```

Conclusions

or was this headache worth it ?

Expression Templates ? Really !

What did we learn ?

- E.Ts helps create concise and intentionnal code
- Implementation can be made simple
- Tools exists to simplify using such an idiom
- Some gotcha with respect to new standards

What to do ?

- See if your design can't be more language than library
- Play with expression templates as a way to increase clarity
- Rely on tools, experiments, complains to your favoriting compiler team

Credits

- Joel de Guzman for inventing the lambda DSELs probably before me
- Louis Dionne for fixing my implementation
- Eric Niebler for Boost.Proto
- Herb Sutter and Peter Gottschling for N3748/4035

Thanks for your attention