CERN School of Computing

Template Metaprogramming for Massively Parallel Scientific Computing

Lecture 1

Expression Templates

Jiří Vyskočil

Czech Technical University
AS CR – ELI Beamlines

Inverted CERN School of Computing, 29 February – 2 March 2016

CERN School of Computing

Massively parallel scientific computing

- **processing data on large numbers of processors**
- **grid computing**
  - distributed nodes, often heterogeneous
- **clusters**
  - nodes usually in the same data center
  - fast interconnections (10G ethernet, Infiniband,...)
  - mostly homogeneous
  - ~ 10 – 10 000 of many-core nodes (K-Computer: 80 000 nodes)
  - ~ 1 000 – 1 000 000 of CPU cores (Tianhe-2: 3 120 000 cores)
- **need for efficient scalable algorithms**

CERN School of Computing

Lecture series overview

- **techniques for writing algorithms for physics computing in modern C++**
- **we will strive to produce code which is**
  - easy to read
  - efficient
  - modular
- **we will introduce some specific methods to**
  1) enable natural syntax for mathematical operations
  2) tap into the CPU's vector processing capabilities
  3) process large data sets
- **and explore their inner workings**

CERN School of Computing

In this lecture

natural syntax for efficient mathematical operations

- traditionally, efficient code was “low-level”, hard to read
- object-oriented paradigm leads to readable, but often inefficient code
- we can bridge the gap keeping the best of the both worlds by utilizing the C++ templating subsystem
- the technique is called Expression Templates (ETs)
- simply put: ETs are pieces of code closely resembling “natural” mathematical expressions which are translated into low-level code according to user-specified rules during compile time before the machine code is emitted

CERN School of Computing

In following lectures

Tomorrow: *Vectorization with Expression Templates*

- the importance and basic elements of SIMD vectorization
- applying the ET idiom to achieve vectorization while keeping the code nice and clean

Wednesday: *Templates for Iteration; Thread-level Parallelism*

- separating the concepts of “iteration” and “computation”
- utilizing the separation to easily introduce parallelization
- practical example of integrating the Maxwell's equations

CERN School of Computing

C++ Templating

- **a method of *generic programming***
- **allow one function (or class) to operate on different data types without having to be declared for all of them**

```
int i;
double d;
std::array<float, 3> A;
MyClass m;

f(i);
f(d);
f(A);
f(M);
```

without templates

```
void f(int a) {
  cout << sizeof(a);
};
void f(double a) {
  cout << sizeof(a);
};
void f(std::array<float, 3> a) {
  cout << sizeof(a);
};
void f(MyClass a) {
  cout << sizeof(a);
};
```

with templates

```
template < typename T >
void f(T a){
  cout << sizeof(a);
}
```

a contrived example

C++ Templating

- **emitted machine code is the same in both cases**
- **templated version clearly wins in maintainability**

```
int i;
double d;
std::array<float, 3> A;
MyClass m;

f(i);
f(d);
f(A);
f(M);
```

without templates

```
void f(int a) {
  cout << sizeof(a);
};
void f(double a) {
  cout << sizeof(a);
};
void f(std::array<float, 3> a) {
  cout << sizeof(a);
};
void f(MyClass a) {
  cout << sizeof(a);
};
```

with templates

```
template < typename T >
void f(T a){
  cout << sizeof(a);
}
```

a contrived example

CERN School of Computing

C++ Template syntax

- **template functions, classes, and variables (in C++11)**
- **declaration: template < *template parameters* > .....**
  - template parameters can be
    - ***typename T*** (or ***class T***) – T will be a placeholder for a name of a type
    - integral constants
- **specialization: a different version for a specific combination of template parameters**
- **instantiation: generating a specific declaration from templated declaration and the combination of parameters**
  - every combination results in a different function/class/variable
  - only those combinations which are actually used in the code are generated

CERN School of Computing

C++ Template syntax

function template

declaration

```
template <typename T>
T mul2 (T a) {
  return 2*a;
}
```

specialization

```
template <>
int mul2(int a) {
  return (a << 1);
}
```

instantiation

```
mul2( 4 );
mul2( 6.33 );
mul2<double>( 4 );
```

class template

declaration

```
template <typename T, int N>
class Pizza {
  T   topping;
  N  diameter;
};
```

instantiation

```
Pizza<Anchovies, 35> thepizza;
```

shorthand

```
typedef Pizza<Anchovies, 42> L_Romana;
```

alias (C++11)

```
template<int N>
using Romana = Pizza<Anchovies, N>;
```

CERN School of Computing

Algorithms

- an algorithm is a sequence of operations
- in our case (computer simulations in physics)
  - manipulating numbers
  - in computer's memory
- in this lecture (C++ 11)
  - operations: functions, operators, methods, ...
  - numbers: “arithmetic types” - int, float, double[1)]…

[1)] C++11 Standard, Chapter 3.9.1

CERN School of Computing

Data

in physics

- scalars
- vectors
- matrices
- scalar fields
- vector fields
- tensor fields
- ...

in a computer

- one place in memory
- a place in memory and a length
- several places in memory
- float
- double[][]
- std::list<double>
- std::array<std::vector<double>>
- ...

Traditional “low-level” implementation

- **good performance (usually – if written correctly)**
- **handwritten, often hard to maintain**
- **implementation obfuscates the meaning**

Q: What is this code doing?

```
for (int i=0; i<3; ++i) {
  for (int j=0; j<3; ++j){
    a[i] += A[i][j] * x[j];
  }
}
c[0] = a[1]*b[2] – a[2]*b[1];
c[1] = a[2]*b[0] – a[0]*b[2];
c[2] = a[0]*b[1] – a[1]*b[0];
q = c[0]*c[0] + c[1]*c[1] + c[2]*c[2];
```

A:

$$q = \| (\boldsymbol{A}\vec{x})^T \times \vec{b} \|$$

Motivation

what we read in a paper

$$\vec{c} = A \cdot \vec{x}$$

what would we want to code

```
Vector x;
Matrix A;

Vector c = A * x;
```

and we want it to run fast!

what we traditionally had to code

```
double x[3];
double A[3][3];
double c[3];

for (int i=0; i<3; ++i) {
  for (int j=0; j<3; ++j){
    c[i] += A[i][j] * x[j];
  }
}
```

initialization statements omitted for clarity

CERN School of Computing

Object-Oriented approach to the rescue

- ***vector addition***
- **vectors (position, velocity)**
- **define as objects**
- **use a simple internal data structure**
- **overloaded operator+**
- **templatized on dimension N**

```
template <int N>
class Vector
{
  std::array<double, N> v;

  Vector(initializer_list<double>& in){
    copy(begin(in), end(in), begin(v));
  }

  Vector(Vector& b){
    copy(begin(b.v), end(b.v), begin(v));
  }

  Vector operator+ (Vector& b) {
    Vector<N> q;
    for (int i = 0; i<N; ++i)
      q.v[i] = v[i] + b.v[i];
    return q;
  }
};
```

CERN School of Computing

OO performance penalty

- **let's run a simple program using our OO vector**
- **and compare its performance to C-like low-level code**

- **OO vector: 3.9 s**
- **low-level: 1.5 s**

```
Vector<3> result;

for(int n=0; n<N; ++n){
  Vector<3> d(n, n+1, n+2);
  result = result + d;
}
```

```
double result[3];

for(int n=0; n<N; ++n){
  double d[3] = {n, n+1, n+2};
  for (int i=0; i<3; ++i) {
    result[n][i] = result[n][i] + d[n][i];
  }
}
```

CERN School of Computing

What happened? Temporary objects!

- **performance hit due to temporary object construction**
- **we have to construct a new object to hold the temporary result of every operation**

```
Vector a = b + c;

// is in fact:
temporary = b + c;
a = temporary;        // a(temporary)
```

- **4 constructor calls for a single operation** *

```
// it gets worse:
Vector r = a + b + c + d;

// is:
temp1 = a + b;
temp2 = temp1 + c;
temp3 = temp2 + d;
r     = temp3;        // r(temp3)
```

* only 3 with mandatory copy-elision optimization

CERN School of Computing

Template Metaprogramming (TMP)

- **C++ templates conceived to support generic programming**
  - a single function declaration “works” on many types
- **TMP is a technique for using the C++ template system to algorithmically generate source code at compile-time**
- **TMP was “discovered accidentally”**
  - first “metaprogram” computed primes as error messages in 1994

[1] http://aszt.inf.elte.hu/~gsd/halado_cpp/ch06s04.html#Static-metaprogramming

CERN School of Computing

A classic TMP example

- **example metaprogram: calculating the factorial**
- **evaluated at compile time**

```
template <int i> struct
Factorial
{
  static const int value = i * Factorial<i - 1>::value;
};

template <> struct Factorial<0>
{
  static const int value = 1;
};
```

recursive formula

termination clause

```
int main()
{
    return Factorial<5>::value;
}
```

generated assembly:

```
main:
        movl    $120, %eax
        ret
```

- **no runtime computation, result is hard-coded in the executable**

CERN School of Computing

Template Metaprogramming (TMP)

- **a kind of new language-in-a-language**
- **Turing-complete**
  - there's even a Tetris game run by the compiler on the command line
- **algorithmic manipulation of types (instead of variables)**
- **the result of a C++ metaprogram is a C++ program**
  - “compile-time execution” - meta-algorithm “run” by the compiler

[1] Todd L. Veldhuizen: C++ Templates are Turing Complete (2003)
[2] http://blog.mattbierner.com/stupid-template-tricks-super-template-tetris/

CERN School of Computing

Back to our example

- **object's operators are part of its interface**
- **we want to**
  - keep the OO interface
  - get rid of the OO overhead
- **Expression Templates (ET)**
  - high-level syntax for the algorithm
  - use TMP to generate its low-level implementation

CERN School of Computing

C++11 and C++14 recap

- **auto**
  - variable type automatically deduced form its initializer
  - function return type automatically deduced from the type of the variable in the return statement
- **decltype( )**
  - returns the name of the type of a variable or expression you pass to it
  - slightly different deduction rules than auto
- **decltype(auto)**
  - automatic type deduction using decltype rules
  - C++14, works on type-only expressions
  - can be used for types returned from templates

there's much more behind the scenes, but let's keep it at this today

CERN School of Computing

C++11 and C++14 recap

- **rvalue references: &&**
  - lvalue “object you can refer to”, you can take its address
  - rvalue “temporary object returned from a function”, no address
- **std::forward**
  - facilitates “perfect forwarding” of parameters between functions
  - doesn't remove additional type qualifiers while passing

```
class Pizza {
public:
  Pizza(Pizza&& rhs);
  .....
};
```

a constructor taking an rvalue reference – constructs form a temporary object

the standard says that std::forward “Returns: static_cast<T&&>(t)”

for more details, refer to:
Meyers: Effective Modern C++, O'Reilly 2015
or google your way around

Building an Expression Template

- **we want to keep the OO interface, and use TMP to generate efficient low-level code**
- **strategy:**

1) form an abstract tree representation of the expression

2) possibly simplify / modify

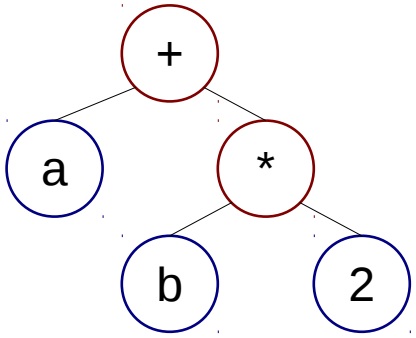
3) evaluate (lazy)

i.e. “not until it's needed”

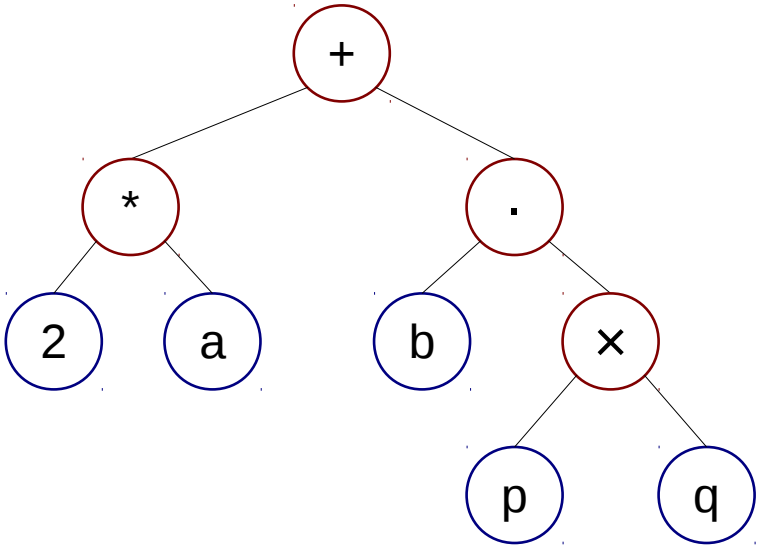
```
// expressions

A*x – b;
q * (E + cross(v, B));


// statements – evaluation needed

c = A*x – b;
Vector r = q * (E + cross(v, B));
```

Abstract Syntax Trees

- expression (arithmetic in our case) represented as a tree
- operations are nodes
- data (numbers, vectors,...) are leaves



a + b * 2



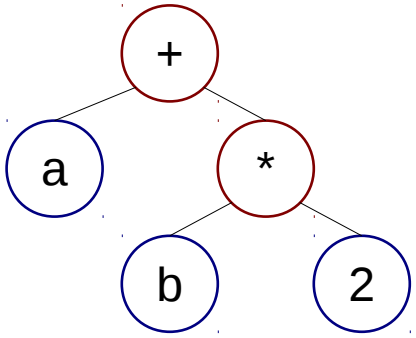
2 * a + b · (p × q)

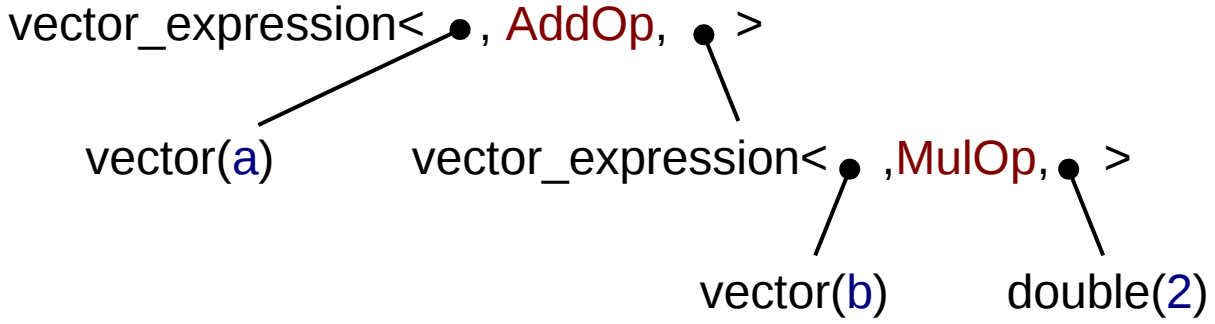
Abstract Syntax Trees and ETs

- **arithmetic expression** a + b * 2
- **is built of template type nodes**

vector_expression<LeftExp, BinaryOp, RightExp>

- **forms the following syntax tree**





- **which is actually a single expression of the type**

vector_expression< vector, AddOp, vector_expression<vector, MulOp double> >

CERN School of Computing

Tree node in C++

- node's type is decided on instantiation, and is composed of the LeftExp, BinaryOp, and RightExp
- variables l and r link to subexpressions – other nodes, or leaves depending on their type
- we pass the subexpressions to the node's constructor
- next we need to define what happens, when we add two expression together...

```
template <
  typename LeftExp,
  typename BinaryOp,
  typename RightExp
>
class vector_expression {
    LeftExp l;
    RightExp r;

public:
  vector_expression(
    LeftExp _l,
    RightExp _r ) :
      l(_l), r(_r)
    { }
};
```

CERN School of Computing

Creating the Expression

```
template <typename RE>
auto operator+ (RE&& other) const ->
  decltype(auto)
{
  return
    vector_expression<
      vector_expression<LeftExp,BinaryOp,RightExp> const&,
      BinaryOp,
      decltype(std::forward<RE>(other))
    >( *this,
       std::forward<RE>(other) )
  ;
}
```

- **define the addition operator for the vector_expression class**
- **we are adding the other expression to the current expression**
- **C++14 syntax saves a lot of typing here**

CERN School of Computing

Creating the Expression

```
template <typename RE>
auto operator+ (RE&& other) const ->
  decltype(auto)
{
  return
   vector_expression<
     vector_expression<LeftExp,BinaryOp,RightExp> const&,
     BinaryOp,
     decltype(std::forward<RE>(other))
   >( *this,
      std::forward<RE>(other) )
  ;
}
```

- **the result of the operation is a new expression**

CERN School of Computing

Creating the Expression

```
template <typename RE>
auto operator+ (RE&& other) const ->
  decltype(auto)
{
  return
    vector_expression<
      vector_expression<LeftExp,BinaryOp,RightExp> const&,
      BinaryOp,
      decltype(std::forward<RE>(other))
    >( *this,
       std::forward<RE>(other) )
  ;
}
```

- **the result of the operation is a new expression**
- **this new expression has two branches, each of them linking to some subexpression**

CERN School of Computing

Creating the Expression

```
template <typename RE>
auto operator+ (RE&& other) const ->
  decltype(auto)
{
  return
    vector_expression<
      vector_expression<LeftExp,BinaryOp,RightExp> const&,
      BinaryOp,
      decltype(std::forward<RE>(other))
    >( *this,
       std::forward<RE>(other) )
  ;
}
```

- **the two subexpressions will be connected via specified binary operation (addition in this case)**
- **the operation will become the part of the resulting expressions type signature**

CERN School of Computing

Creating the Expression

```
template <typename RE>
auto operator+ (RE&& other) const ->
  decltype(auto)
{
  return
    vector_expression<
      vector_expression<LeftExp,BinaryOp,RightExp> const&,
      BinaryOp,
      decltype(std::forward<RE>(other))
    >( *this,
       std::forward<RE>(other) )
  ;
}
```

- **left-hand side of the new expression will be the current expression**

CERN School of Computing

Creating the Expression

```
template <typename RE>
auto operator+ (RE&& other) const ->
  decltype(auto)
{
  return
    vector_expression<
      vector_expression<LeftExp,BinaryOp,RightExp> const&,
      BinaryOp,
      decltype(std::forward<RE>(other))
    >( *this,
       std::forward<RE>(other) )
  ;
}
```

- **RE is whatever we got on the right side of the + operator**
  - it might be a vector, or another expression
  - we need to preserve its fully qualified type

CERN School of Computing

Creating the Expression

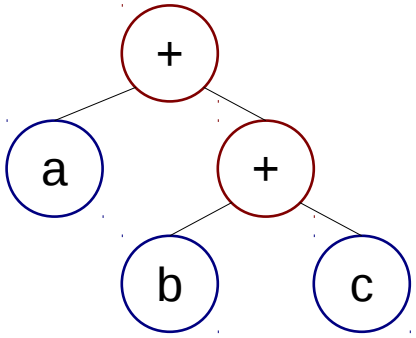
```
template <typename RE>
auto operator+ (RE&& other) const ->
  decltype(auto)
{
  return
    vector_expression<
      vector_expression<LeftExp,BinaryOp,RightExp> const&,
      BinaryOp,
      decltype(std::forward<RE>(other))
    >( *this,
       std::forward<RE>(other) )
  ;
}
```

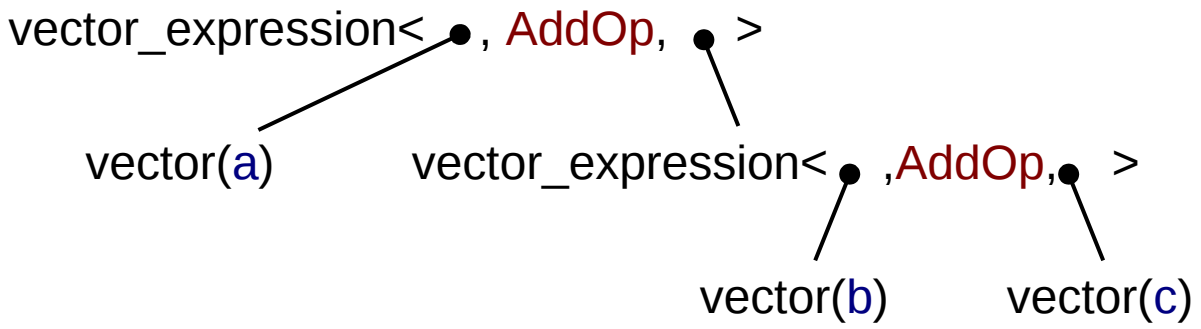
- **chaining operators creates the AST at compile time**
- **the *vector* class, which holds the data (we'll implement it later) has a similar operator+**
  - with ***vector*** as LeftExp type, and ***AddOp<double>*** as BinaryOp

Creating the Expression

- **by repeatedly calling the operator+ on the subexpressions, we build the abstract syntax tree out of template nodes of generalized type** vector_expression<LeftExp, BinaryOp, RightExp>

- **for an example expression a+b+c, we have the following tree**



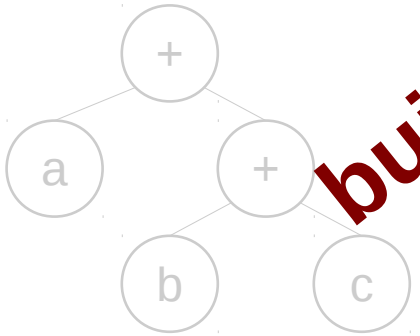


- **which is actually a single expression of the type**

vector_expression< vector, AddOp, vector_expression<vector, AddOp, vector> >

Creating the Expression

- by repeatedly calling the operator+ on the subexpressions, we build the abstract syntax tree out of template nodes of generalized type vector_expression<LeftExp, BinaryOp, RightExp>
- for an example expression a+b+c, we have the following tree



vector_expression< , AddOp, >

vector(a) vector_expression< ,AddOp, >

vector(b) vector(c)

built at compile time

no calculation done yet

- which is actually a single expression of the type

vector_expression< vector, AddOp, vector_expression<vector, AddOp, vector> >

CERN School of Computing

Lazy evaluation

- **defer evaluation until the result is needed**

  x = a + b + c;
- **expression can be further manipulated (simplifications, ...)**
- **need an evaluation trigger**
- **only when we really have to evaluate**
  - assignment using operator =
  - variable constructor
  - argument to an operator/function not included in our ET class
- **the types of the whole statement are:**

vector = vector_expression< vector, AddOp, vector_expression<vector, AddOp, vector> >

CERN School of Computing

Evaluating the Expression I

vector = vector_expression< vector, AddOp, vector_expression<vector, AddOp, vector> >

- **the assignment operator must trigger the evaluation**
  - because it will store the result
- **we tell the compiler to replace the operator= by the loop**
  - which might also get unrolled
- **we're basically casting vector_expression to vector**
  - while calling operator[] on the vector_expression object
- **still compile-time**

```
template <int N>
class vector {
    template <typename RightExpr>
    vector& operator = (RightExpr&& re)
    {
      for (int   i = 0;   i < N;   ++i)
        v[i]  =  re[i];
      return *this;
    }
};
```

We know how to formally convert **vector_expression** into a **vector**, but how do we calculate the actual numbers which form the result?

CERN School of Computing

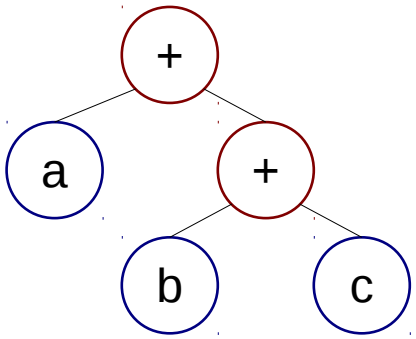
Evaluating the Expression II

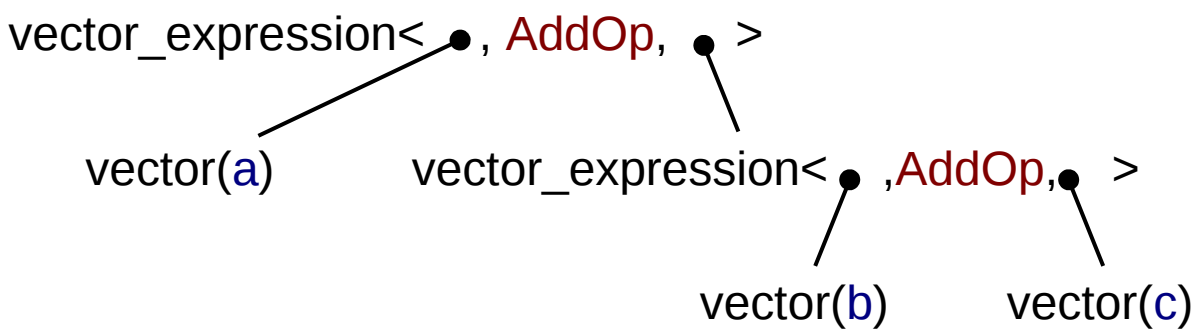
- **operator[] applies the specified BinaryOp on the subexpressions' i-th element**
- **operator[] recursively propagates through the tree**
- **when the function reaches a leaf (a vector), operator[] returns the i-th component**
- **evaluation itself can be done using a static function which implements the binary operation**

```
template<RightExpr,BinaryOp,LeftExpr>
class vector_expression
{
  auto operator [](int i)
    const -> decltype(auto)
    {
      return BinaryOp::apply(l[i], r[i]);
    }
};

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

Evaluating the Expression III

- assignment operator trigger the cast from vector_expression to vector
- which is performed by recursively applying operator[] in a loop to subexpressions connected by operator+
- thus the compiler translates the statement **x = a+b+c**, where the expression **a+b+c** is represented by the syntax tree





- into the following code

```
for(int i=0; i<3; ++i) {
  x[i] = a[i] + b[i] + c[i];
}
```

CERN School of Computing

Generated assembly

- **the ET code reads as naturally as the OO code**
- **a good compiler will emit the same instructions for well-written ET code, as in a hand-coded version**

version	number of instructions in the loop	time spent in the loop
simple OO	15	3.9 s
expr. templ.	8	1.5 s
hand-coded	8	1.5 s

simple object

```
Vector<3> result;
for(int n=0; n<N; ++n){
  Vector<3> d(n, n+1, n+2);
  result = result + d;
}
```

expression template

```
vector<3> result;
for(int n=0; n<N; ++n){
  vector<3> d(n, n+1, n+2);
  result = result + d;
}
```

hand-coded

```
double result[3];
for(int n=0; n<N; ++n){
  double d[3] = {n, n+1, n+2};
  for (int i=0; i<3; ++i) {
    result[n][i] = result[n][i] + d[n][i];
  }
}
```

CERN School of Computing

Expression transformation / simplification

- **expressions can be simplified or otherwise transformed before evaluation**
- **templates are Turing-complete**
  - any transformation algorithm might be employed
- **be aware of potential dangers of floating-point arithmetic**

1) David Goldberg: What Every Computer Scientist Should Know About Floating-Point Arithmetic

CERN School of Computing

Transformation example I

- **our evaluation trigger might first perform a transformation**
- **the result of the transformation will be a vector_expression**
- **the transformation is performed at compile-time**
- **remember the trigger is vector::operator= not vector_expression::operator=**

```
// same as before
template <int N>
class vector {
    template <typename RightExpr>
    vector& operator = (RightExpr&& re)
    {

      // the Transform functions returns
      // a vector_expression, no cast is
      // done here, no runtime computation
      auto transformed = Transform(re);
      for (int   i = 0;   i < N;   ++i)
        v[i]  =  transformed[i];
      return *this;
    }
};
```

CERN School of Computing

Transformation example II

- **this example recursively changes all operations in the AST to subtraction (not very useful)**
- **the single-parameter specialization will stop the recursion in case the parameter is not a vector expression**
- **more precisely – we stop the recursion in the general case, and only recurse if we do have a vector_expression**

```
// this is here to stop the recursion
template<typename Expr>
auto Transform(const Expr& expr) -> Expr
{
    return expr;
}

// only instantiated for vector_expression type
template<LE, BinaryOp, RE>
auto Transform (
  vector_expression<LE, BinaryOp, RE>
  const& expr
) -> decltype(auto)                    input
{
    return Transform(                  output
        vector_expression <
            LE, SubtractOp, RE >
              (expr.l, expr.r)
    );
}
```

some details omitted

CERN School of Computing

Algorithm specialization

- some operations might benefit form a specialized algorithm
- create a partial specialization of the vector_expression template
- you can even offload the computation to a library routine

- suppose we also have a class Matrix, and some great optimized algorithm perform multiplication by a matrix **A * x**
- note that in this case, the evaluation is triggered by the specialized operator
  - whether this is what we wanted depends on the algorithm

```
template < typename RightExpr >
class vector_expression<
  Matrix, MulOp, RightExpr
> {
  .....
  template <typename RE>
  auto operator* (RE&& other) const ->
      decltype(auto)
   {
     result = .....; // run our algorithm
     return result;
   }
};
```

CERN School of Computing

RVO, copy elision, C++11 moves

- **modern compilers can be very smart about optimizations**
- **Return-Value-Optimization and copy-elision together with additional optimizations turned on by -O3 might remove calls to most superfluous constructors**
- **used together with C++11 move semantics, the performance of simple OO expressions *might be* on par with ETs**
  - this is by no means guaranteed
  - different compilers have vastly different results
  - it's difficult to argue about more complex cases

version	gcc 5.3.1	icc 16.0.1
simple OO	2.1 s	5.4 s
expr. templ.	2.0 s	1.3 s
hand-coded	2.0 s	1.3 s

different machine than previous slides
icc -O3 -fp-model fast -xHost
gcc -O3 -ffast-math -march=native

CERN School of Computing

Potential pitfalls

- **simple ETs without appropriate specialized algorithms might be under-performing**
- **performance depends on the compiler**
  - be careful when using complex libraries with lots of TMP
  - test, measure, profile
- **proper inlining is of particular concern**
  - more about inlining in lecture 3

CERN School of Computing

Take Away Messages

- **Expression Templates allow you to use object-like syntax without the inherent performance penalty**
- **ETs construct an Abstract Syntax Tree representing your "mathematical expression"**
- **the AST can be arbitrarily manipulated – the templating system is Turing-complete**
- **"lazy evaluation" is triggered by assignment**
- **evaluated expressions translate into "low-level" code according to the rules implemented by the ET engine**
  - a good ET engine produces efficient low-level code

CERN School of Computing

Thank you for your attention!

you can find full examples and links to additional sources at:
vysko.cz/icsc2016