# C++ Templates for Scientific Programming

January 26, 2010

# About C++

- C++ is a *general purpose* language (unlike Fortran, Matlab).
- Runs anywhere, *for all sorts of programs* (unlike ..)
- Compiled and strongly typed (like Fortran; unlike Matlab).
- Can be called from other languages.
- Needs a bit more effort than special purpose languages.

# What is strong typing?

The stupid program

```
double average(double x, double y)
{
  return (x+y)/2;
}
```

is compiled to efficient machine instructions

```
        pushl   %ebp
        movl    %esp, %ebp
        fldl    16(%ebp)
        faddl   8(%ebp)
        popl    %ebp
        fmuls   .LC0
        ret
.LC0:
        .long   1056964608
```

But programs may also have a mathematical meaning

```
double average(double x, double y)
{
  return (x+y)/2;
}
```

Machine

↓

```
        pushl   %ebp
        movl    %esp, %ebp
        fldl    16(%ebp)
        faddl   8(%ebp)
        popl    %ebp
        fmuls   .LC0
        ret
.LC0:
        .long   1056964608
```

Mathematician

↓

$$\frac{x+y}{2}$$

# Can we generalize like a mathematician?

```
int average(int x, int y)
{
  return (x+y)/2;
}

double average(double x, double y)
{
  return (x+y)/2;
}

complex average(complex x, complex y)
{
  return (x+y)/2;
}
```

Same idea, different machine instructions.

# Templates

In C++ we have *templates* for generalizing:

```
template<typename T>
T average(T x, T y)
{
  return (x+y)/2;
}
```

The compiler will try to figure out what machine code to write

```
a = average(1, 2);
a = average(1.2, 3.4);
a = average(3, 5.5); // Error, ambigous!
a = average<double>(3, 5.5); // OK
a = average<int>(3, 5.5); // Also OK
```

We can use old code for new tricks:

```
matrix m1, m2, m3;
m3 = average(m1, m2); // Average two matrices
```

We can average anything with a + and /2 defined!

```
electron_density d1, d2, da;
da = average(d1, d2); // Average density
```

With templates we can make C++ more mathematical, but still get good performance.

With templates you can run in multiple precision[1] *without changing your code*. Specialize!

```
my_algorithm<float>(); //Single precision
my_algorithm<double>(); //Double precision
my_algorithm<qd_real>(); // Quad Double precision
```

Automatic Differentiation (if we only had a polynomial type..):

```
my_algorithm< polynomial<double> >();
```

The compiler will "cut and paste" the type (**double** etc.) into the code.

[1] http://crd.lbl.gov/~dhbailey/mpdist/

# What are the downsides?

- “Long” compilation times (compared to?)
- Easy to generate a lot (MB’s) of code
- Have to beware type conflicts and conversions
- Horrible error messages from the compiler
- Need some thought to design (but easy to use)

# Template classes

We can also create general data structures:

```
template<typename T>
class matrix
{
  int rows, columns;
  T *data;
  ...
};
```

We use it like this:

```
matrix<double> M;
matrix<complex> Mc;
M.invert();
det = Mc.determinant();
```

# Tensoring

And why not a block matrix?

```
matrix< matrix<double> > M;
```

$$M = \begin{pmatrix} M_{11} & M_{12} \\ M_{21} & M_{22} \end{pmatrix}$$

Mathematical concept of Tensor product!

$$\mathbb{R}^{n \times m} \otimes \mathbb{R}^{k \times l}$$

Will

```
M.invert();
```

work?

# Recursive templates

Binomial coefficients:

$$
\begin{array}{ccccccccccc}
 & & & & \rightarrow k & & & & & & \\
 & & & & 1 & & & & & & \\
 & & & & 1 & 1 & & & & & \\
 & & & & 1 & 2 & 1 & & & & \\
 & & & & 1 & 3 & 3 & 1 & & & \\
 & & & & 1 & 4 & 6 & 4 & 1 & & \\
 & & & 1 & 5 & 10 & 10 & 5 & 1 & & \\
 & & & & \downarrow n & & & & & &
\end{array}
$$

Recursive definition:

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

# Recursive functions

Let's program

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

Recursive solution:

```
int binomial(int n, int k)
{
  if (k == 0 or k == n)
    return 1;
  else
    return binomial(n-1,k-1) + binomial(n-1,k);
}
```

Works, but super slow ($2^n$ function calls).

# Recursive template

Template solution: calculate coefficients at *compile time*.

```
template <int n, int k>
struct binomial
{
    enum { value = binomial<n-1,k-1>::value
                 + binomial<n-1,k>::value };
};

template <int n>
struct binomial<n,n> //Terminate recursion
{
    enum { value = 1 };
};
```

# Recursive template

Use the binomial coefficient template like this:

```
cout << "binomial(10,9)=" << binomial<10,9>::value;
```

Super fast – just printing a constant instead of 1000 function calls.

We *cannot* do

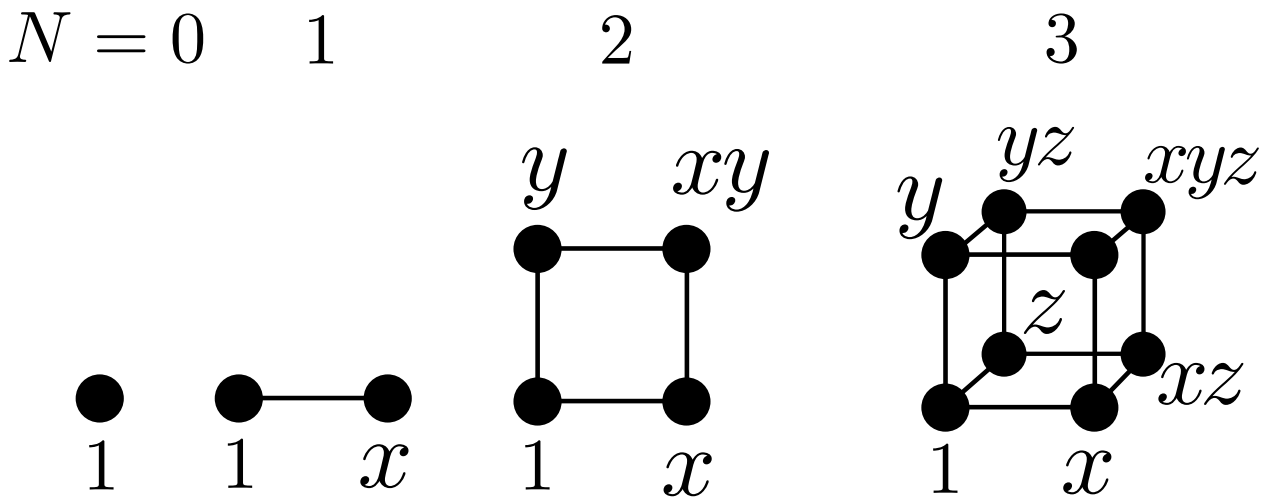
```
for (int i=0;i<3;i++)
  cout << "binomial(3," << i << ")="
       << binomial<3,i>::value;
```

Why not?

No specific loop construct for templates, have to use recursive definitions. ☹

# A useful template class

Let's end with a real example: "Cubic" polynomials




We have $2^N$ terms, so $N$ probably smaller than $\sim 20$.

# Cube template class

An $N$-cube consists of two cubes of lower dimension,

```
template< typename T, int N >
struct cube
{
  cube< T, N-1 > lower[2];
};
```

except for the 0-cube, which has only one coefficient. A *template specialization*!

```
template<typename T>
struct cube<T,0>
{
  T data;
};
```

# Arithmetics

We can add cube polynomials of the same type

```
template< typename T, int N >
cube<T,N> operator+(cube<T,N> c1, cube<T,N> c2)
{
  cube<T,N> res;
  res.lower[0] = c1.lower[0] + c2.lower[0];
  res.lower[1] = c1.lower[1] + c2.lower[1];
  return res;
}
```

# Arithmetics

And multiply with truncation (spot the recursion?):
$(a+bx)(c+dx) = ac + (ad+bc)x + \mathcal{O}(x^2)$

```
cube<T,N> operator*(cube<T,N> c1, cube<T,N> c2)
{
  cube<T,N> res;
  res.lower[0] = c1.lower[0]*c2.lower[0];
  res.lower[1] = c1.lower[0]*c2.lower[1]
               + c1.lower[1]*c2.lower[0];
  return res;
}
```

# Intrinsics

And a handful of intrinsics: $f(a+bx) = f(a) + f'(a)bx + \mathcal{O}(x^2)$

```
cube<T,N> exp(cube<T,N> c)
{
  cube<T,N> res;
  res.lower[0] = exp(c.lower[0]);
  res.lower[1] = exp(c.lower[0])*c.lower[1];
  return res;
}
```

Other functions only slightly more complicated. Also have to take care of $N = 0$ case. Add `const &`, assignment operators etc, and..

# Super fast special purpose automatic differentiation

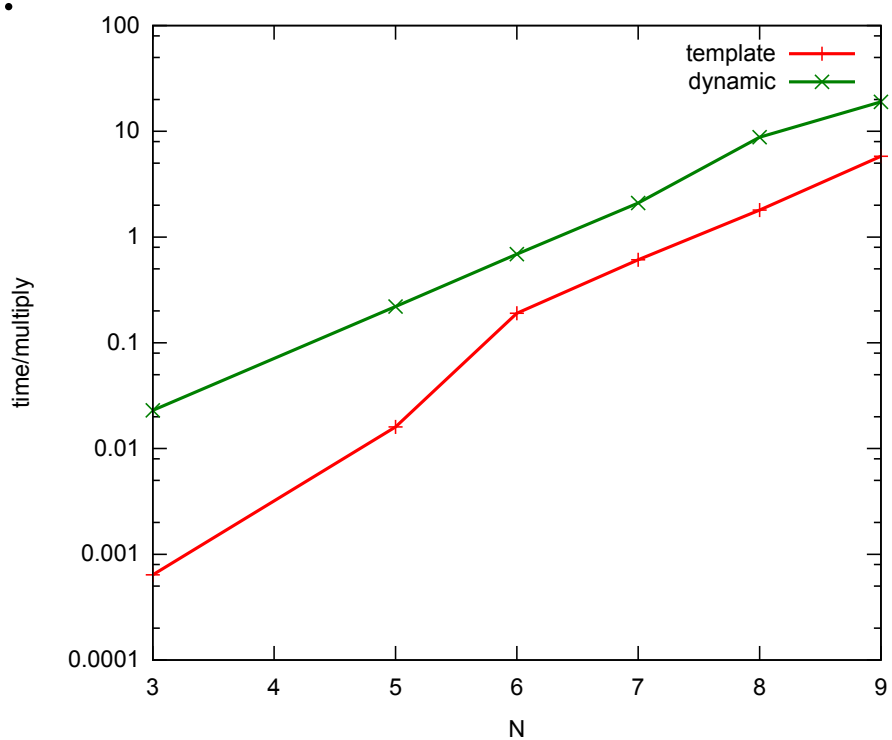
```
template<typename T>
T f(T x, T y)
{
  return x*log(x) + sqrt(x*y);
}


..
cube<double,3> x,y,z;
.. // Initialize x and y
z = f(x,y); //Taylor expand f
```

The “cube” form is encountered in some problems, but is inefficient in general.

# Performance of the cube class

Multiplication performance compared to standard recursive implementation:




Templates can be efficiently optimized for small sizes. Hybrid approaches are possible. General Taylor 50x faster with templates.

# Software

Don't lock up useful code in complicated programs!

1. Fast (small) multivariate polynomial multiplication
   `http://polymul.googlecode.com`
2. Taylor expansions and automatic differentiation
   `http://libtaylor.googlecode.com`

# A general Taylor series library

I have developed a general (order and number of variables) Taylor expansion library.

- High order automatic differentiation.
- `C++` template implementation.
- High performance (but not "fast") multiplications.
- Arbitrary coefficient type (high precision possible).
- Standard math functions implemented.
- Efficient change of variables (example: rotate multipoles)
- GPU implementation coming.

Code is freely available. *Suitable for high order problems with a small number of active variables*.

# Taylor library example

```
template<typename T>
T dist(T x, T y)
{
    return sqrt(x*x + y*y);
}

void expand_it()
{
    //Set up two infinitesimal variables
    taylor<double, 2, 3> dx(0,0), dy(0,1);
    cout << "Expansion:" << dist(2+dx,3+dy) << endl;
}
```

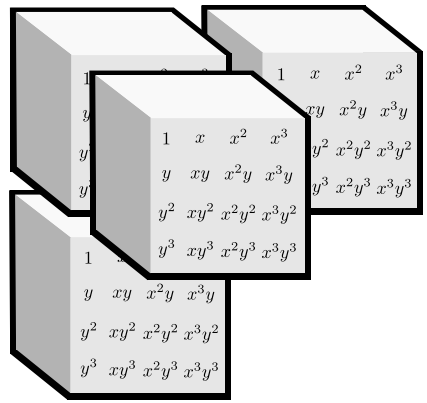
Works reasonably up to $\sim$ 1000 coefficients.

# Taylor class

Problem specific structures from tensoring:

$$\begin{array}{cccc} 1 & x & x^2 & x^3 \\ y & xy & x^2y & x^3y \\ y^2 & xy^2 & x^2y^2 & x^3y^2 \\ y^3 & xy^3 & x^2y^3 & x^3y^3 \end{array}$$

$$\begin{array}{cccc} 1 & x & x^2 & x^3 \\ y & xy & x^2y & \\ y^2 & xy^2 & & \\ y^3 & & & \end{array}$$



```
taylor< taylor<double, 5,N>, 3,1> x;
```

# Getting started with C++ templates

- Not covered in this talk: Expression templates
- Look at existing libraries, but make sure they are up to date!
- Make sure your compiler is up to date (i.e. GCC)
- Get a good *modern* C++ reference and start programming

Why templates?

- Brings back flexibility in a compiled language
- Reusable *concepts*
- High performance