Advanced C++ Techniques in Computational Fluid Dynamics

Richard Günther

Inst. für Astronomie und Astrophysik
Abt. Computational Physics
Contents

1. Background
2. Advanced Object Oriented programming with C++
   - C++ for Experts in 3 Minutes
   - Template Metaprogramming
   - Expression Templates
3. The POOMA Library
4. Application
5. Conclusion
Background

Work sponsored by SFB 382, “Verfahren und Algorithmen zur Simulation physikalischer Prozesse auf Höchstleistungsrechnern”; project C19, “Dreidimensionale MRHD”. Available and used codes:

- RH2D, twodimensional RHD
- TRAMP, threedimensional RHD
- NIRVANA v0, threedimensional MHD

Goals for C19:

- Modern (don’t use Fortran 77)
- Parallel (MPI/OpenMP)
- Full-Featured (physically and technically)
Object Oriented programming with C++

OO in C++ can be done using the following well-known abstraction mechanisms:

- Classes
- Class inheritance
- Virtual methods
- Operator overloading

And other, less well-known:

- Multiple (virtual) class inheritance
- Templates and (partial) specialization
C++ features suitable for CFD

- Avoid virtual methods and multiple inheritance in performance critical sections.
- Use traits classes and (partial) specialization to ease maintenance and avoid runtime conditionals.
- Use operator / function overloading for complex types to allow rapid development and ease maintainance.

Operator overloading may expose costly temporaries for fat objects. Using template metaprogramming can solve this.

Veldhuizen, 2000 (Techniques for Scientific C++)
Template specialization and function overloading as Swiss Army Knife!

template <class> class A;
class B;
template <> class A<int>
{
    void foo();
};
template <> class A<B>
{
    void bar();
};
Partial specialization of function templates not allowed. Do the trick using function overloading:

```cpp
template <int> struct WrappedInt {};  
template <class Op, class Domain>  
void doSomething(Op&, Domain&, WrappedInt<1>);  
template <class Op, class Domain>  
void doSomething(Op&, Domain&, WrappedInt<2>);  
```
or using partial template specialization of a wrapper class:

template <class Op, int Dim, template <int> class Dom>
struct WrapDoSth;

template <class Op, template <int> class Dom>
struct WrapDoSth<Op, 1, Dom> {
    static void doSomething(Op&, Dom<1>&);
};

template <class Op, template <int> class Dom>
void doSomething(Op& o, Dom& d)
{
    WrapDoSth<Op, Dom::dim, Dom>::doSomething(o, d);
}
Let the compiler do static parts of algorithms!

- Possible with templates and specialization
- Template specialization makes up a Turing-Complete language.

```cpp
template <int n> struct Factorial {
    static const int result
    = Factorial<n-1>::result * n;
};
template<> struct Factorial<1> {
    static const int result = 1;
};
const int factorial_of_7 = Factorial<7>::result;
```
Template metaprogramming is Lisp-like:

- Thinking the recursive way
- Using recursive data-structures like lists and trees
- Building static information into types
- Lots of ugly templates

You pay for it through compile time regressions.
Avoid temporaries in evaluating expressions for fat objects like Vectors, Tensors or complete Arrays

\[ x = \frac{x}{1.0 + x} \]

Allow generic functions to operate on expressions like usual objects

\[ \rho_{\text{max}} = \max \left( \frac{p}{T} \right) \]

Possible using Expression Templates (Veldhuizen, 1995)!
The complete type representing the expression would look like

\[
\text{Expr<BinaryNode<OpDiv, Reference<T>, BinaryNode<OpAdd, Scalar<double>, Reference<T>>>}
\]

Note the missing LHS from the final assignment.
The operator= of the type T triggers evaluation of the expression

template <class E> T& T::operator=(Expr<E>&);

And finally creates an instance of the overloaded operator= which looks like (for vector-like T)

for (int i=0; i<LHS.size(); ++i)
    LHS(i) = RHS(i);

This gets expanded by the compiler using inlining to

for (int i=0; i<x.size(); ++i)
    x[i] = x[i]/(1.0+x[i]);
Use computations on the expression type to select algorithms to query

- Patch intersections
- Used guards
- Necessary communication

Straightforward to allow serial, MPI and OpenMP implementation to co-exist.
The POOMA Library

The POOMA library was created at ACL/LANL (Reynders et al., 1996), now free software maintained at CodeSourcery (www.pooma.com). Extensively uses expression templates through the PETE library. Provides a framework for parallel grid-based CFD.

- Arrays, Fields
- Tiny objects like Vectors, Tensors, Matrices

Fields contain
- Multiple components and centering points
- Mesh abstraction
- Relations between Fields and for BCs
To suite our needs POOMA was extended by:

- Non-uniform meshes and spherical and cylindrical coordinates
- HDF5 serial and parallel I/O
- Native OpenMP parallelization
- Native MPI parallelization
- Glue to PETSc library
- Autogeneratable reference documentation
Using POOMA

Program finite differences using and combining any of
- Expression templates
- Simple stencils
- Generic stencil-like sub-programs
- Patch-local sub-programs
Using POOMA, Expression templates

Using expression templates is as simple as writing expressions with Arrays or Fields as arguments:

\[
\rho = \rho_0 \times \text{pow}(\text{positions}(\text{rh}).\text{comp}(0), -1.5) \\
\times \exp(-\text{pow}(\cos(\text{positions}(\text{rh}).\text{comp}(1))/0.05, 2))
\]

\[
(\rho = \rho_0 \cdot r^{-1.5} \cdot e^{-\left(\frac{\cos \theta}{0.05}\right)^2})
\]

Expression template support in POOMA includes reductions and selections:

\[
\rho = \text{where}(\rho < 1e-6, 1e-6);
\]

\[
dt = \text{min}(\text{norm}(\text{spacings}(v)/(v+cs)));
\]
Using POOMA, simple stencils

Finite differencing of one quantity

```cpp
struct Gradient {
    template <class F>
    double operator()(const F& f, int i, int j) {
        return 0.5*(f.read(i+1,j) - f.read(i-1,j))
              + 0.5*(f.read(i, j+1) - f.read(i, j-1));
    }
};
```

```cpp
flux = Stencil<div>()(rho*v);
rho = dt * flux;
```
Using POOMA, stencil-like sub-programs (a)

Complex local operations

```cpp
struct MinMod {
    template <class LHS, class RHS1, class RHS2>
    void operator()(LHS& l, RHS1& r1, RHS2& r2,
                    Loc<Dim>& I)
    {
        if (r1(I) > 0 && r2(I) > 0)
            l(I) = std::min(r1(I), r2(I));
        else if (r1(I) < 0 && r2(I) < 0)
            l(I) = std::max(r1(I), r2(I));
        else
            l(I) = 0;
    }
};
```
Using POOMA, stencil-like sub-programs (b)

```cpp
ScalarCode<MinMod>()(s,
    Stencil<BackDiff>(rho*v),
    Stencil<CentralDiff>(rho*v),
    Stencil<ForwDiff>(rho*v));
```

Or use manual written loops per patch, like for glueing to Fortran code.
It’s a trade-off between re-usable components and classic style of writing code.
Using all this for 3d MRHD. Current state:

- threedimensional parallel ideal hydro in spherical, cylindrical and Cartesian coordinates
- Poisson solver for self-gravity in Cartesian coordinates

Physical applications

- Stellar oscillations as a test-problem
- 3d non-viscous protoplanetary accretion disks
Conclusion

Using C++ for Computational Fluid Dynamics applications is feasible.

- Available libraries for advanced C++ like Boost
- Available libraries for (parallel) arrays like POOMA, Blitz++

→ As easy as using Fortran9x
- Performance comparable to C, Fortran
- Free compilers available!
- More people know C++ than Fortran9x