# Scientific Programming in C++

Gábor Csányi

Cavendish Laboratory
University of Cambridge

# Main ideas

- Expressive programming: **WYSI(M)P** (what you see is (mostly) physics)
- Portability: must work with *every* major HPC compiler
- Modularity: research vs. production code
- Maintain performance: build on vendor BLAS
- Examples
  - Atomistic simulation code modules across several projects
  - DFT++: a plug–and–play basis set electronic structure code

# Expressive programming

We think about physics in terms of sophisticated formalisms

$$
\begin{aligned}
\text{Verlet alg.} \quad &: \quad \begin{cases} 1. & \mathbf{v} \leftarrow \mathbf{v} + \frac{1}{2}\Delta t \mathbf{a} \\ 2. & \mathbf{a} \leftarrow \mathbf{F}/\mathbf{m} \\ 3. & \mathbf{v} \leftarrow \mathbf{v} + \frac{1}{2}\Delta t \mathbf{a} \\ 4. & \mathbf{x} \leftarrow \mathbf{x} + \Delta t \mathbf{v} + \frac{1}{2}\Delta t^2 \mathbf{a} \end{cases} \\
E_{\mathrm{elec}} \quad &= \quad \mathrm{Tr}(\hat{\rho}\hat{\mathcal{H}}) \\
E_{\mathrm{DFT}} \quad &= \quad \sum_i \langle \psi_i | \hat{T} + \hat{V}_I | \psi_i \rangle + \frac{1}{2}\langle n | \phi \rangle + \langle n | \epsilon_{xc} \rangle
\end{aligned}
$$

There is a lot of hidden information

$$
\begin{array}{ccccc}
\mathbf{a} & & \mathbf{F} & & \mathbf{m} \\
\begin{pmatrix} \vdots \\ \vdots \\ \vdots \end{pmatrix} & \begin{matrix} \longleftrightarrow \\ \longleftrightarrow \\ \longleftrightarrow \end{matrix} & \begin{pmatrix} \vdots \\ \vdots \\ \vdots \end{pmatrix} & \begin{matrix} \longleftrightarrow \\ \longleftrightarrow \\ \longleftrightarrow \end{matrix} & \begin{pmatrix} \cdot \\ \cdot \\ \cdot \end{pmatrix}
\end{array}
$$

# What is C+ ? And why ?

Use the smallest subset of C++ that achieves the objectives, whilst remaining completely portable.

- $\surd$ Classes: collect variables that belong together (derived types in FORTRAN)
- $\surd$ Overloading: define operators on new objects

# What is C+ ? And why ?

Use the smallest subset of C++ that achieves the objectives, whilst remaining completely portable.

- ✓ Classes: collect variables that belong together (derived types in FORTRAN)
- ✓ Overloading: define operators on new objects

"When your hammer is C++, everything begins to look like a thumb."
— Steve Hoflich, comp.lang.c++

# What is C+ ? And why ?

Use the smallest subset of C++ that achieves the objectives, whilst remaining completely portable.

- √ Classes: collect variables that belong together (derived types in FORTRAN)
- √ Overloading: define operators on new objects

But avoid anything else. In particular:

- × Templates (autogeneration of classes)
- × Inheritance (hierarchies of classes)

The choice of language should not lead to a flame war

- Most computer languages can simulate a Turing Machine
- Use the language that you are most familiar with

# Two examples

Toolkit for atomistic simulation

- Collection of interoperable modules
- Research into new quantum simulation methods
- Multiscale hybrid molecular dynamics, multiple models

DFT++

- Pseudopotential density functional code
- Basis dependent parts are separated out
- Option of using plane waves or wavelets

  Development lead by T. A. Arias (MIT, later Cornell)

# A Class

Compact storage of a group of variables related to a physical concept

```
class Atoms {
      int N;                     // Number of atoms
      matrix3 latticeVectors; // 3x3 matrix
      double neighbourCutoff;
      V3Array positions;         // V3Array:  3xN array to hold 3D data
      V3Array travel;            // Due to Periodic Boundary Condition
      vector mass;               // Atomic masses
      Flag atomicNumber;         // Array of bytes
      int *n_neighbours;         // Number of neighbours
      int **connect;             // Neighbour connection table
      int (**shift)[3];          // Table of lattice shifts to get closest image
      double **distance;         // Cache of neighbour distances
      ...
};
Atoms myAtoms1, myAtoms2;                                 // Declaration
myAtoms1 = diamond_structure_100(lattice_const,Nx,Ny,Nz);    // Assigment
```

# Operators, Methods

Bookkeeping tasks on classes, e.g.

```
myAtoms1.updateConnectionTable();
myAtoms1.addAtom(vector3 position, int Z);
myAtoms1.print_xyz();
myAtoms1.read(FILE *stream);       // Read from an already open stream
myAtoms1.read(char *fileName);     // Open a file and read from it
myAtoms1.distance(int i, int j);   // Return distance (from cache if available)
myAtoms1.map_into_cell();
```

Methods operate on the `myAtoms1` object, distinct from any other *instance* of the class.
Rigorous bounds checking and error trapping is implemented.

Arithmetic:

```
positions += dt * velocity + (0.5*dt*dt) * acceleration;  // all V3Array objects
```

## The dreaded temporaries

The expression `a = b + c + d` is evaluated as `a = (b + (c + d))`, so the intermediate values of `t1 = c+d` and `t2 = b + t1` have to be allocated, stored and copied.

When doing quantum mechanics, this takes negligible time with *vectors*, sometimes even *matrices*.

# The dreaded temporaries

The expression `a = b + c + d` is evaluated as `a = (b + (c + d))`, so the intermediate values of `t1 = c+d` and `t2 = b + t1` have to be allocated, stored and copied.

When doing quantum mechanics, this takes negligible time with *vectors*, sometimes even *matrices*.

Entire models can be encapsulated in classes, e.g. a Tight Binding model, including all parameters and matrix operations.

```
E = TBmodel1.energy(myAtoms1);
F = TBmodel1.forces(myAtoms1);          // Reuse density matrix from previous call

TBmodel::cohesiveEnergy(Atoms &atoms, Electrons &elec){
    fillHamiltonian(atoms, elec);       // Uses precomputed distance info in atoms
    elec.diagonalizeH();                // Wrapper to a LAPACK call
    return (elec.fillings * elec.eigenvalues);         // vector dot product
}
```

*No temporaries of large objects (wavefunctions, Hamiltonians)*

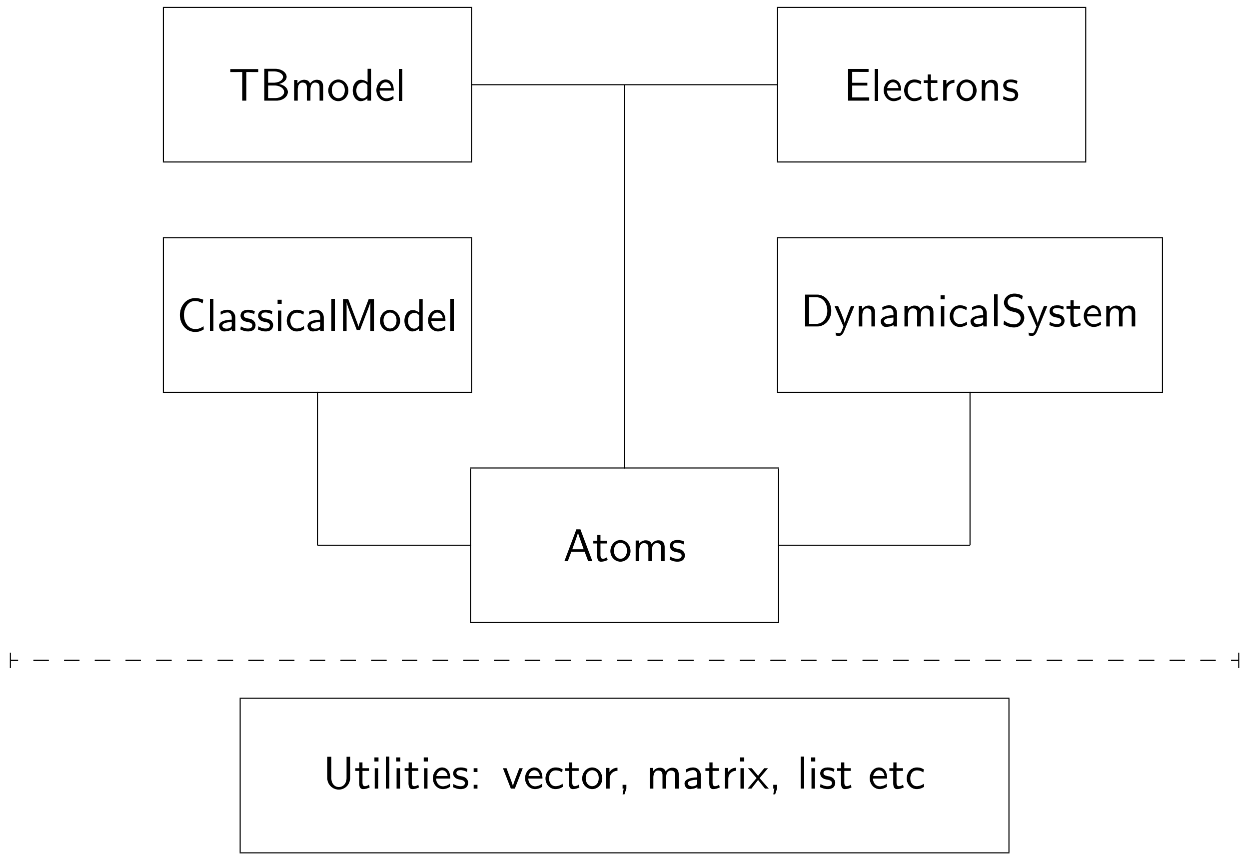
# Sssh... a little privacy

Classes can hide certain variables from other objects.

```
class Atoms{
      private:
            matrix3 latticeVectors;
            matrix3 reciprocalLattice;// Updated whenever latticeVectors changes
      public:
            ...
}
```

- Objects protect themselves from incompetence and malicious use.
- Drastic reduction of coding errors.

# Main modules




Modules are reused over and over again from 2 month 4th year projects to production code, visualization tools, etc. Dozens of independent programs.

# Research vs. Production

Any programming language that has function calls can be used to make modular code

1. Draw up the specification of the code
2. Design data structures
3. Design information flow
4. Implement modules

# Research vs. Production

Any programming language that has function calls can be used to make modular code

1. Draw up the specification of the code
2. Design data structures
3. Design information flow
4. Implement modules

However, in research

- At the outset we don't know what we want to achieve
- Testing of ideas starts with simple programs
- Code can grow from 100 lines to 30,000 lines without a *re*write
- Easily extendable objects can maintain compatibility

# How to maintain performance?

- Objects have carefully written constructors and destructors
- Optional wrappers to `malloc()` that track memory usage and detect leaks
- Avoid temporaries of large objects
- Pass arguments of large objects by reference (address)
- Run `prof` frequently, learn where bottlenecks are likely

Disclaimer: this only works if LAPACK/BLAS really does take the lion's share

# DFT++: why write a new DFT code?

Motivations (cca. 1995 at MIT):

- Old one is obsolete (20,000 lines of F77 in a single source file, don't ask..)
- Commercial code was expensive, lacked newest features
- Separate physics from basis set issues: most of the effort in writing a DFT code is actually to do with the basis set
- Educational value

# Traditional formalisms for quantum mechanics

- Heisenberg: operators $\hat{H}$, $\hat{x}$
- Schrödinger: wave functions $\psi(x)$
- Dirac: bras and kets $|\psi\rangle$, $\langle\alpha|$
- Great *conceptual* tools.

But for actual computation...

$$\begin{aligned} E &= \sum_i \langle\psi_i|\hat{T} + \hat{V}_I|\psi_i\rangle + \frac{1}{2}\langle n|\phi\rangle + \langle n|\epsilon_{xc}\rangle \\ \langle r|n\rangle &= \sum_i \|\langle r|\psi_i\rangle\|^2 \end{aligned}$$

And the gradient...

$$\delta E = \sum_i \frac{\partial E}{\partial|\psi_i\rangle} \cdot \delta|\psi_i\rangle + \text{c.c.} \qquad ??$$

× Not explicit enough for computation

Alternative: Explicit computational representation

```
dE = 0
for i=1 to nbands
   calc_grad(i,g)
   for j=1 to nbasis
      dE += g(j,i)*dpsi(j,i)
   end
end
```

Difficult to manipulate formally:

- obscures physics
- hard to communicate between humans
- implementation of new ideas is not straightforward

# Wish list

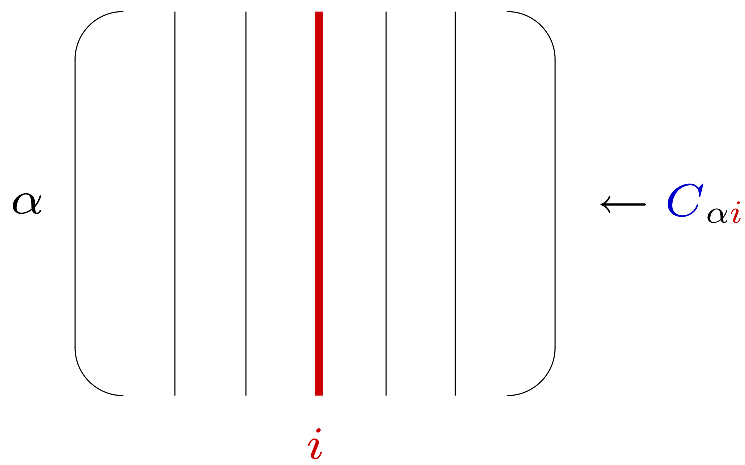
What would we like for a Quantum Computational Formalism?

- Ease of formal manipulation (like Dirac notation): good for communication
- Explicit: no ambiguities in implementation
- Performance: linear algebraic (aim: within factor of 2 of "equivalent" F77)
- Modular and flexible: easy development and parallelization

# DFT++ formalism

Given basis functions $|\alpha\rangle$ to represent the wavefunctions $|\psi\rangle$,

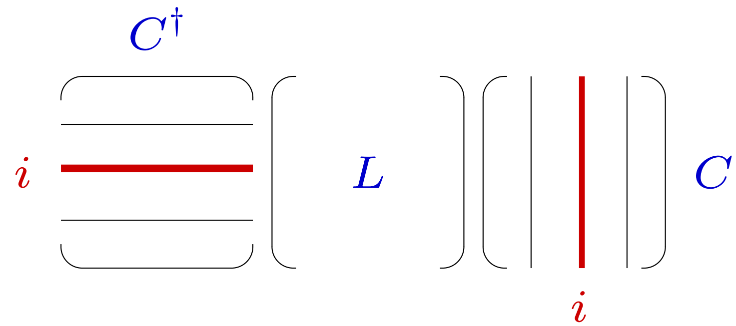
$$|\psi_i\rangle = \sum_\alpha |\alpha\rangle\, C_{\alpha i}$$




- Work with computational representation $C$
- Rewrite all formulas in terms of such matrices
- Basic Linear Algebra Subroutines $\Rightarrow$ high performance

# Kinetic energy

$$L_{\alpha\beta} \equiv \langle \alpha | \nabla^2 | \beta \rangle$$




$$T = -\frac{1}{2} \sum_i \langle \psi_i | \nabla^2 | \psi_i \rangle = -\frac{1}{2} \sum_i C_i{}^\dagger L C_i$$

$$\Rightarrow T = -\frac{1}{2} \mathrm{Tr} \left( C^\dagger L C \right) = -\frac{1}{2} \mathrm{Tr} \left( L \cdot C C^\dagger \right)$$

# Electron density

Grid of points $p$ in real space

$$\mathcal{I}_{p\alpha} \equiv b_\alpha(p)$$

$$\begin{array}{cc} p & \begin{pmatrix} \quad & \quad & \quad & \quad \end{pmatrix} \\ & \alpha \end{array} \leftarrow \mathcal{I}_{p\alpha} \qquad \begin{array}{cc} p & \begin{pmatrix} \quad & \quad & \quad & \quad \end{pmatrix} \\ & i \end{array} \leftarrow \Psi_{pi}$$

$$\begin{aligned} \psi_i(p) &= \sum_\alpha b_\alpha(p)\, C_{\alpha i} = (\mathcal{I}C)_{pi} \\ n(p) &= \sum_i \psi_i(p)\psi_i(p)^* = \mathrm{diag}\left(\mathcal{I}CC^\dagger\mathcal{I}^\dagger\right) \end{aligned}$$

$$\begin{array}{cc} p & \begin{pmatrix} \quad & & \\ & \quad & \\ & & \quad \end{pmatrix} \\ & p' \end{array} \leftarrow (\Psi\Psi^\dagger)_{pp'}$$

# Required operations in DFT++

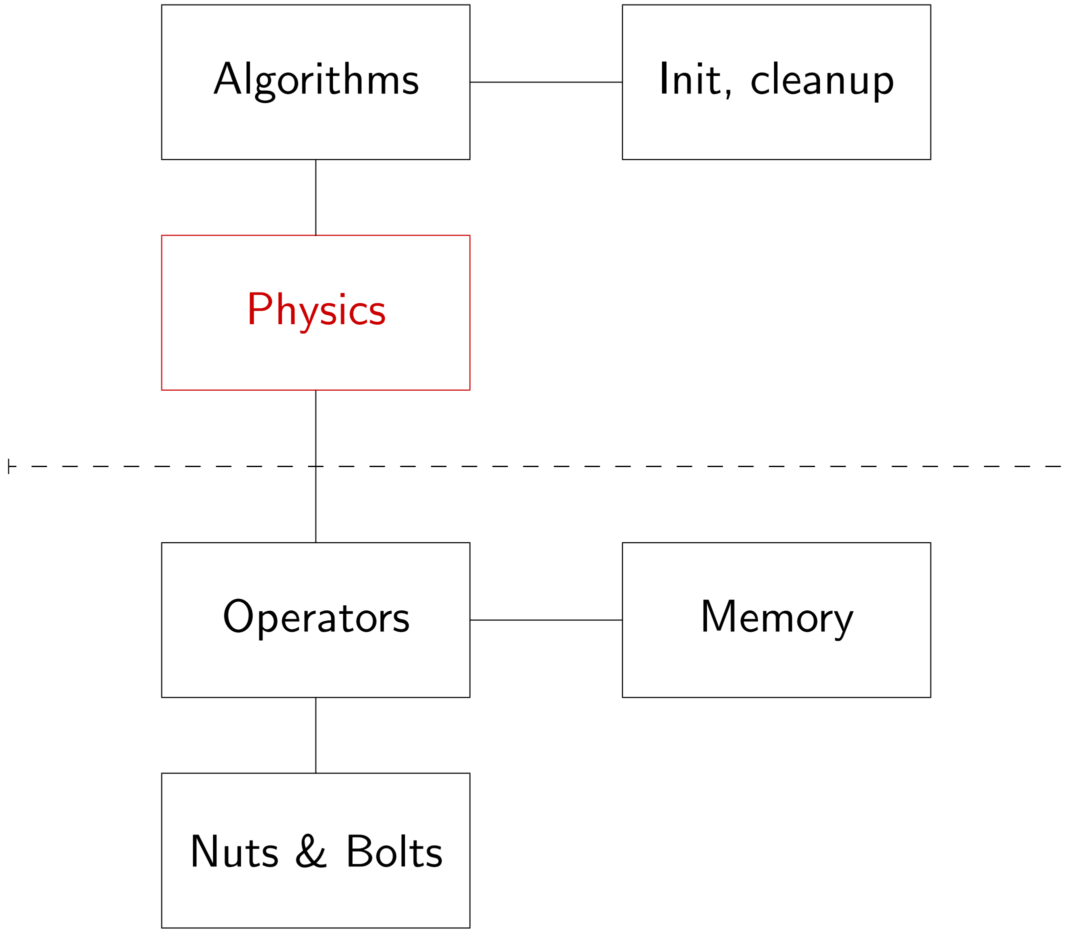
Basis independent:

- Tr, diag, Diag
- Matrix $+$, -, *

Basis dependent:

- $L$ and $\mathcal{O}$: $\mathcal{O}_{\alpha\beta} = \langle\alpha|\beta\rangle$
- $\mathcal{I}$, $\mathcal{I}^\dagger$, $\mathcal{I}^{-1}$,$\mathcal{I}^{-\dagger}$
- Ionic potential $V_I$:

$$(V_I)_\alpha = \int dr\, b_\alpha(r)^*\, V_I(r)$$



# LDA in DFT++

$$
\begin{aligned}
E_{LDA} &= -\frac{1}{2}\text{Tr}\left(C^\dagger L C\right) + \\
&\quad (\mathcal{I}^{-1}n)^\dagger \left[V_I + \frac{1}{2}\mathcal{O}\phi + \mathcal{O}\mathcal{I}^{-1}\epsilon_{xc}(n)\right]
\end{aligned}
$$

$$
\begin{aligned}
n &= \text{diag}(\mathcal{I}CC^\dagger\mathcal{I}^\dagger) \\
\phi &= -4\pi L^{-1}\mathcal{O}\mathcal{I}^{-1}n
\end{aligned}
$$

Minimization: simply differentiate w.r.t. $C$

$$
\begin{aligned}
\delta E_{LDA} &= \text{Tr}\left(\delta C^\dagger \cdot [H\,C] + [C^\dagger H] \cdot \delta C\right) \\
H &= -\frac{1}{2}L + \mathcal{I}^\dagger[\text{Diag } V]\mathcal{I} \\
V &= \mathcal{I}^{-\dagger}\left[V_I + \mathcal{O}\phi + \mathcal{O}\mathcal{I}^{-1}\epsilon_{xc}(n)\right] \\
&\quad +[\text{Diag } \epsilon'_{xc}(n)]\mathcal{I}^{-\dagger}\mathcal{O}\mathcal{I}^{-1}n
\end{aligned}
$$

# Fully functional DFT code in C++

```
// Initializations ...
for (i=0; i < niter; i++)
  {
    U = Y^O(Y);
    C = Y * ( U^(-0.5) );
    n = diagouter(I(C));
    phi = -4*PI*invL(O(J(n)));
    // $E = -\frac{1}{2}\mathrm{Tr}\left(C^\dagger L C\right) +$
    //      $(\mathcal{J}n)^\dagger \left[V_I + \frac{1}{2}\mathcal{O}\phi + \mathcal{O}\mathcal{J}\epsilon_{xc}(n)\right]$
    E = -0.5 * Tr(C^L(C)) +
        J(n)^(V_I + 0.5*O(phi) + O(J(exc(n))));
    V = Jdag(V_I + O(J(exc(n))) + O(phi)) +
        Diag(excprime(n)) * Jdag(O(J(n)));
    HC = -0.5 * L(C) + Idag(Diag(V) * I(C));
    G = (HC - O(C * (C^HC))) * (U^(-0.5));
    Y -= stepsize * G;
  }
// Cleanup ...
```

# Extensions

- Localized basis O(N) methods
- Gradient corrections (GGA)
- Self–interaction corrections (SIC)
- Nonlinear core corrections
- Fully parallelized using MPI (serial part 2%)
- Spin (LSDA)

All of these were added *a posteriori* with relative ease.

Originally, the operators $\mathcal{I}$, $\mathcal{I}^{-1}\ldots$ were implemented as Fourier Transforms (using FFTW), yielding a plane wave code.

Later, wavelet transforms were added as an option.

# Spin in DFT++

Instead of the usual sum over k–points, create a new class:

```
class QuantumNumber
{
    vector3 kvec;
    int spin;
    ...
}
```

Each quantum state has an object of type `QuantumNumber` associated with it.

$$\epsilon_{xc}(n) \quad \longrightarrow \quad \epsilon_{xc}(n_\uparrow, n_\downarrow)$$

$$V(n) \quad \longrightarrow \quad V_\uparrow(n_\uparrow, n_\downarrow) \quad , V_\downarrow(n_\uparrow, n_\downarrow)$$

Top level of code unchanged!

# The future?

Is it possible to achieve high–level syntax (à la MATLAB and MATHEMATICA) and maintain high performance?

The BLITZ++ library

Nice constructions, complexity hidden by amazing trickery, e.g.

- `a=b+c+d` evaluated in a single loop
- *Range* constructions:

  ```
  Range i(1,N-1), j(2,N);
  A(i) = B(i)+B(j);
  ```
- Claims matching (or better) speed than straight F77

# The future?

Is it possible to achieve high–level syntax (à la MATLAB and MATHEMATICA) and maintain high performance?

The BLITZ++ library

Nice constructions, complexity hidden by amazing trickery, e.g.

- `a=b+c+d` evaluated in a single loop
- *Range* constructions:

  ```
  Range i(1,N-1), j(2,N);
  A(i) = B(i)+B(j);
  ```
- Claims matching (or better) speed than straight F77

Unfortunately the library is not very portable
NO Intel, NO Portland Group, NO Sun compiler, NO Cray

*No thanks!*

# Summary

- Minimal use of object oriented features on top of C
- Expressive software: looks like physics
- Modular, reusable code
- Low barrier, incremental software implementation
- Maintain LAPACK/BLAS performance