Compile-Time Polymorphism in C++ :

Performance, Generics, and Extensibility

Timothy J. Williams

Advanced Computing Laboratory

Seminar at IBM T. J. Watson Research Center

February 9, 2000

Outline

- C++
  - Polymorphism
  - Generic programming
- POOMA
- Performance
- Generic programming
- Extensibility

`Array`

- Parallel evaluation

C++ Classes

- User-defined type
  - Member data
  - Member functions
- Declared variable of this type is *object*

- Like Java class
- Like C **`struct`** w/ functions

C++ Classes

- User-defined type
  - Member data
  - Member functions
- Declared variable of this type is *object*
- Like Java class
- Like C **`struct`** w/ functions

```
Class Date {
   int day, month, year;
   Date(int d, int m, int y)  {
      { day = d;
        month = m;
        year = y; }
   void addYears(int n)
      { year += n; }
};

// February 9, 2000:
Date today(9,2,2000);

// February 9, 2525:
today.addYears(525);
```

C++ Class Templates

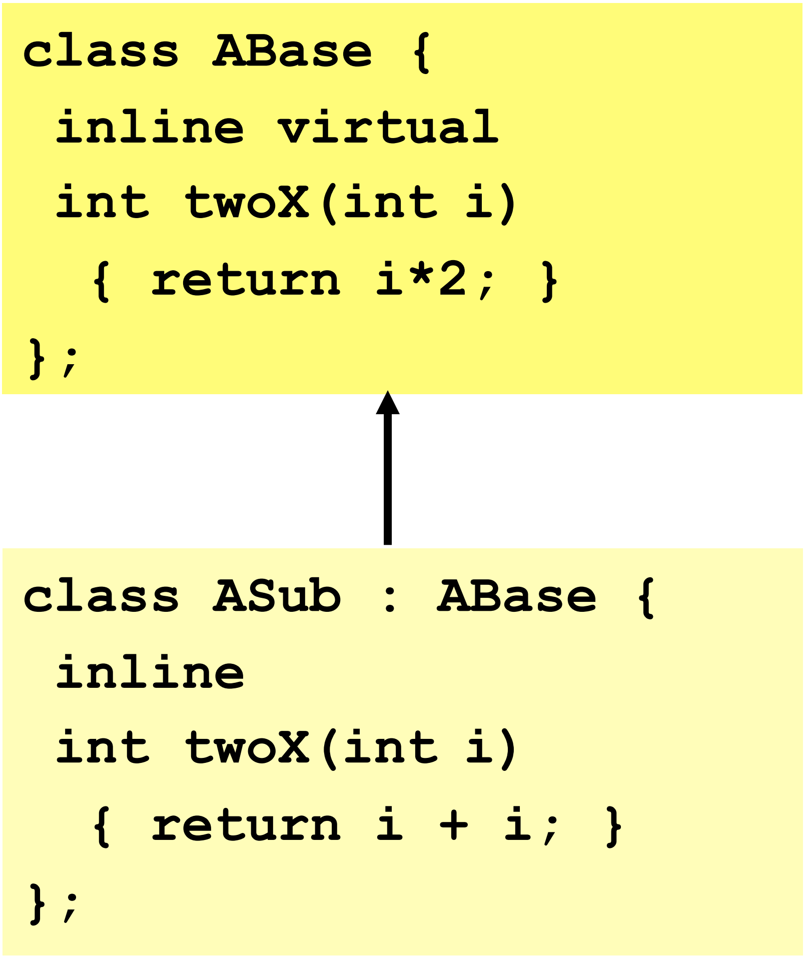
- Parameterized type

```
template<int Dim, class T>
class NDArray {
  T *data;
  NDArray(int *sizes)
    { for (int d=0; d < Dim; d++)
        { nElements *= sizes[d]; }
      data = new T[nElements]
    }
};
```

- Declared object w/specific parameters is *template instance*

```
int sizes[2] = {10,10};
NDArray<1,double> a1(sizes);
NDArray<2,int> a2(sizes);
```

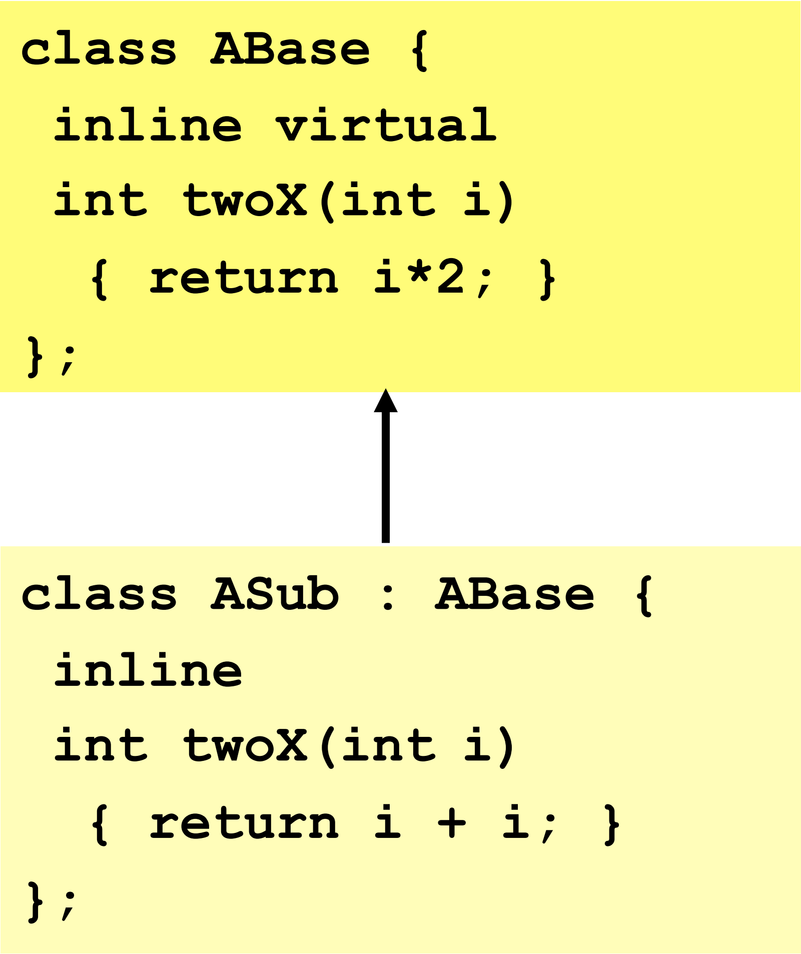
Runtime Polymorphism



```
class ABase {
 inline virtual
 int twoX(int i)
   { return i*2; }
};
```

```
class ASub : ABase {
 inline
 int twoX(int i)
   { return i + i; }
};
```

Runtime Polymorphism



```
class ABase {
 inline virtual
 int twoX(int i)
  { return i*2; }
};
```

```
class ASub : ABase {
 inline
 int twoX(int i)
  { return i + i; }
};
```

```
int foo(ABase &a) {
 int sum = 0;
 for (int i = 0;
          i < 1000000000;
          i++)
    {
      sum += a.twoX(i);
    }
 return sum;
}
```

Runtime Polymorphism

```
class ABase {
 inline virtual
 int twoX(int i)
   { return i*2; }
};
```

```
class ASub : ABase {
 inline
 int twoX(int i)
   { return i + i; }
};
```

```
int foo(ABase &a) {
 int sum = 0;
 for (int i = 0;
          i < 1000000000;
          i++)
 {
   sum += a.twoX(i);
 }
 return sum;
}
```

Can’t inline …
One billion function calls!

Compile-Time Polymorphism

```
template<class HowTwoX>
class A;
```

```
    class TwoMult {};
    class TwoAdd {};
```

```
class A<TwoMult> {
  inline
  int twoX(int i)
    { return 2*i; }
};
```

```
            class A<TwoAdd> {
              inline
                int twoX(int i)
                 { return i + i; }
            };
```

Compile-Time Polymorphism

```
template<class HowTwoX>
class A;
```

```
    class TwoMult{};
    class TwoAdd{};
```

```
class A<TwoMult>{
 inline
 int twoX(int i)
  { return 2*i; }
};
```

```
      class A<TwoAdd>{
       inline
        int twoX(int i)
         { return i + i; }
      };
```

```
template<class HowTwoX>
int foo(A<HowTwoX> &a)
{
 int sum = 0;
 for (int i = 0;
      i < 1000000000;
      i++)
   { sum += a.twoX(i); }
 return sum;
}
```

Compile-Time Polymorphism

```
template<class HowTwoX>
class A;
```

```
class TwoMult {};
class TwoAdd {};
```

```
class A<TwoMult> {
 inline
 int twoX(int i)
  { return 2*i; }
};
```

```
class A<TwoAdd> {
 inline
  int twoX(int i)
   { return 2*i; }
};
```

```
template<class HowTwoX>
int foo(A<HowTwoX> &a)
{
 int sum = 0;
 for (int i = 0;
      i < 1000000000;
      i++)
   { sum += a.twoX(i); }
 return sum;
}
```

Inlines!

Generic Programming

- Standard Template Library
  - Containers

    ```
    template<class T> queue;
    template<class T> list {
      list::iterator begin();
      list::iterator end();
    };
    ```

  - Algorithms

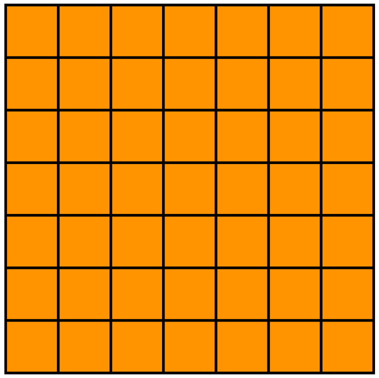
    ```
    template<class Iterator, class T>
    T sum(Iterator first, Iterator last, T &iv);
    ```

- Generic algorithms act on any type which is a *model* of a *concept*

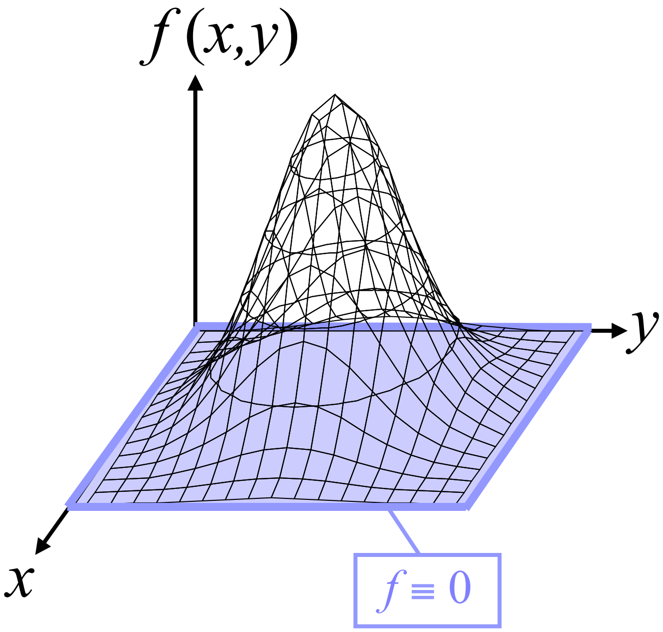
POOMA

- Parallel Object-Oriented Methods and Applications
  - C++ class library for computational science applications
  - Fields, particles, meshes, operators, I/O
    - Distributed objects
  - High-level data-parallel API encapsulates parallelism
    - SMARTS dataflow-driven, thread-based parallelism
    - Message-passing between contexts (in progress)
  - Example uses
    - Compressible, multi-material hydrodynamics
    - Accelerator physics — particle-in-cell
- http://www.acl.lanl.gov/pooma

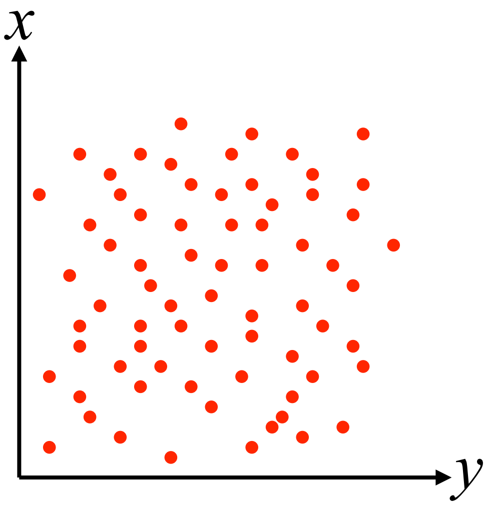
POOMA Key Abstractions



`Array`

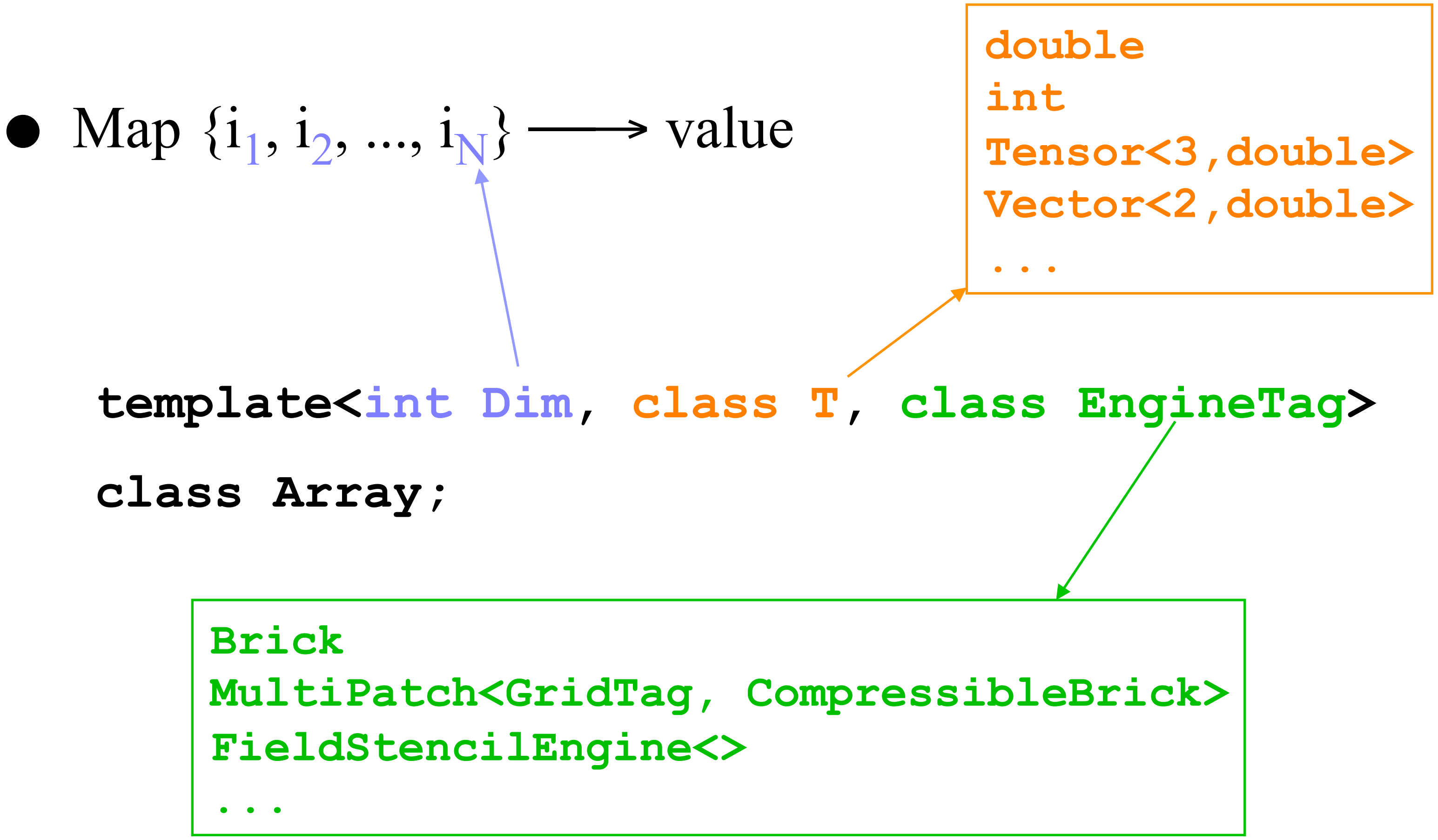


`Field`



`Particles`

Array Class



- Map $\{i_1, i_2, ..., i_N\} \longrightarrow$ value

```
double
int
Tensor<3,double>
Vector<2,double>
...
```

```
template<int Dim, class T, class EngineTag>
class Array;
```

```
Brick
MultiPatch<GridTag, CompressibleBrick>
FieldStencilEngine<>
...
```

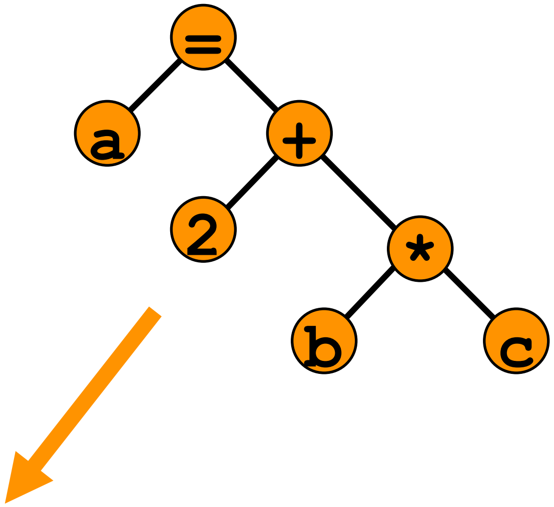
Array Syntax

```
Array<2,double,Brick> a(…), b(…), c(…);

// Whole-array operations:
a = 2 + b*c;

// Subset operations:
Interval<1> I(0, 13), J(2, 20);
Interval<2> I2(I, J);

a(I,J) += b(I,J);
c(I2) = b(I2) + pow(c(I2), 3);

// Stencils:
a(I,J) = (a(I+1, J+1) - a(I-1, J-1))/2;
```

Performance

Expression Templates

- Operators return objects of expression types
  - Tag classes for operator type
  - Combined into parse tree
    - Compile-time traversal



```
a = 2 + b*c;
```

```
Expression<
 TBTree<OpAssign, Array1
  TBTree<OpPlus, Scalar<int>,
   TBTree<OpMultiply, ConstArray2, ConstArray3>>>>
```

Expression Templates (cont’ d)

- Ultimate return type is **`Array`**

```
Array<2, double, ExpressionTag<…> >
```

— Expression engine—

- Evaluation code compiled is efficient

```
for (int i = 0; i < a.size(0); i++) {
  for (int j = 0; j < a.size(1); j++) {
    a(i,j) = 2.0 + b(i,j)*c(i,j);
  }
}
```

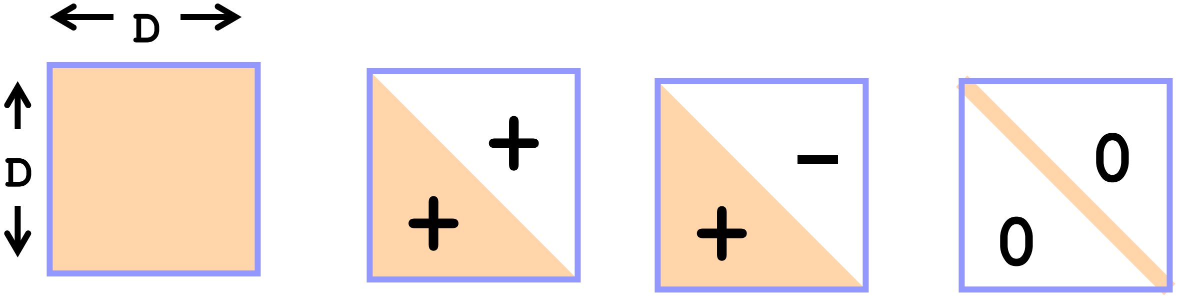
Scalar Array Indexing

- Compile-time polymorphic indexing

```
template<class Dim, class T, class EngineTag>
class Array {
  typedef Engine<Dim, T, EngineTag> Engine_t;
  typedef typename Engine_t::Index_t Index_t;
  T operator()(Index_t i, Index_t j) const
    { return engine(i, j); }
  Engine_t engine;
};
```

- Function **engine(i,j)** is a non-virtual → *inlined*

Tensor Class

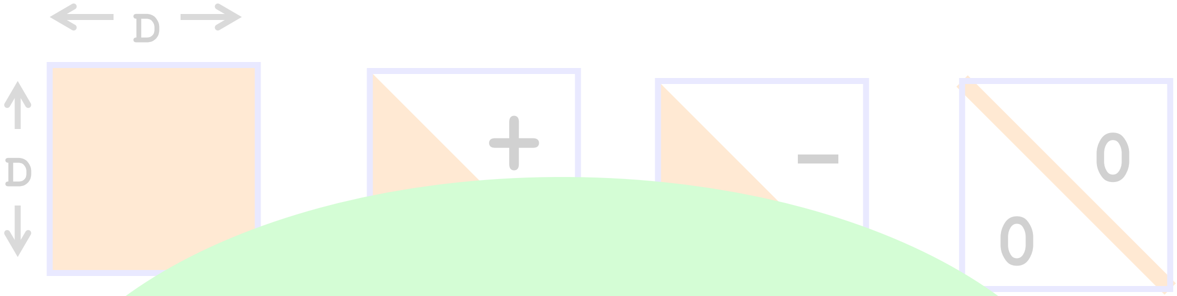


```
template<int D, class T, class EngineTag> class Tensor;

template<int D, class T, class EngineTag> class TensorEngine
{ … T data[TensorStorageSize<D, EngineTag>::Size]; …};
```

Computed at compile-time

```
template<int D> TensorStorageSize<Symmetric>
{… static const int Size = (D*D - D)/2 + D; …};
```

`Tensor` Class



```
template<                                        Tensor;

template                                          nsorEngine
{ … T da                                          ; …};
```

Trivial example

of

template metaprogram

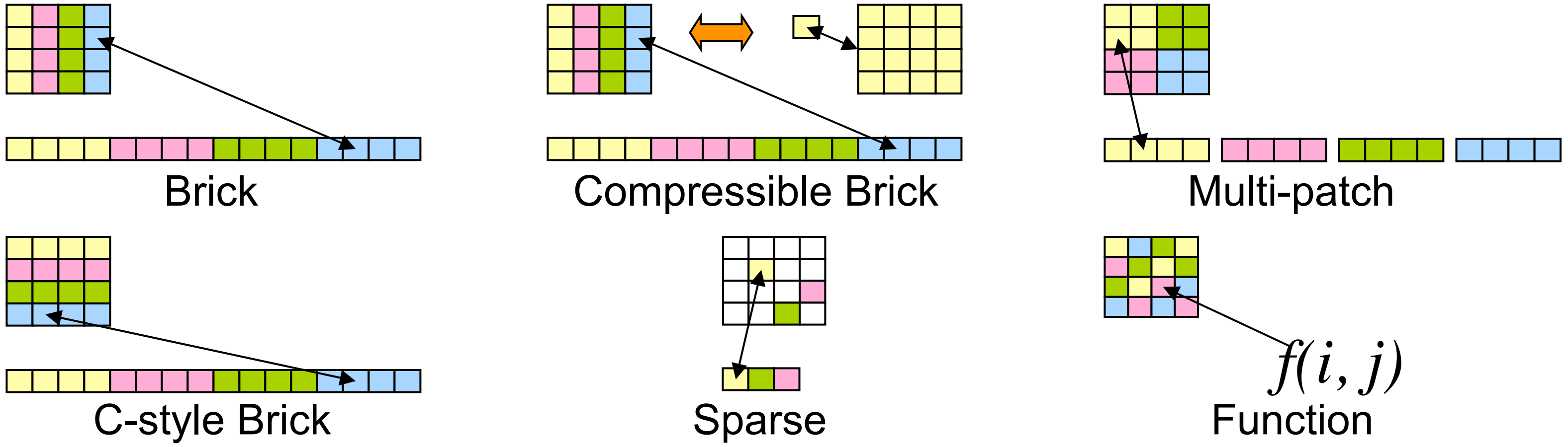
Computed at compile-time

```
template<int D> TensorStorageSize<Symmetric>
{… static const int Size = (D*D - D)/2 + D; …};
```

GENERICS

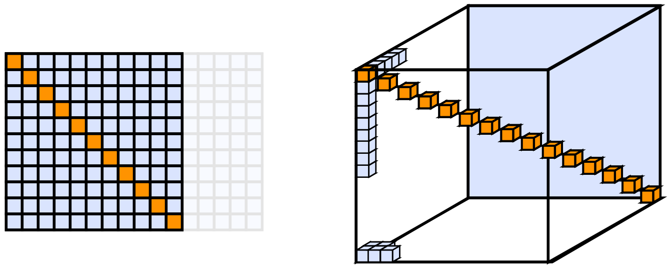
Separate Interface from Implementation

- **`Array`** is interface, **`Engine`** classes are implementation
  - POOMA defines **`Array`** class once
  - Add new **`Engine`** classes later
  - Polymorphic **`Array`** indexing "does the right thing"



Generic Function of **`Array`**

$$\mathrm{trace}(a) \equiv \sum_{i=0}^{N_0-1} a(i,i,i,....)$$



```
template<int Dim, class T, class EngineTag>
inline T trace(const Array<Dim, T, EngineTag> &a)
{
  Interval<Dim> equalIndices;
  T tr = 0;
  for (int i = 0; i < a.length(0); i++) {
    for (int d = 0; d < Dim; d++) {
      equalIndices[d] = Interval<1>(i,i);
    }
    tr += sum(a(equalIndices));
  }
  return tr;
}
```

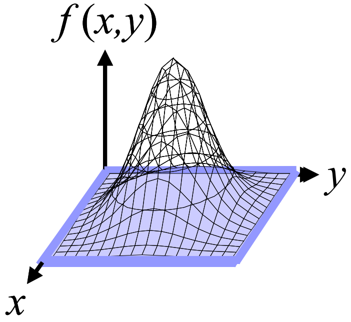
Generic Function of `Array` (cont’ d)

- If **`a`**, **`b`**, and **`c`** are **`Array`**s, these work:

`trace(a);`	whole array
`trace(Interval<Dim>(…));`	indexed subarray
`trace(a + b*c);`	array expression

Only computed on diagonal elements referenced in **`trace()`**

- *Generic*: **`trace`** source independent of
  - Dimensionality
  - Type **`T`**
  - Engine type



Field Class

```
template<class Geometry, class T, class EngineTag>
class Field;
```

```
DiscreteGeometry<Cell, RectilinearMesh<3> >
DiscreteGeometry<FaceRCTag<0>, RectilinearMesh<2> >
...
```

```
template<class Centering, class Mesh>
class DiscreteGeometry;

  template<int Dim, class CoordinateSystem, class T>
  class RectilinearMesh;
```

Extensibility

Add New Elemental Type

- Rank 3 tensor T_{ijk}

```
template<int Dim>
class R3Tensor {…
  double &operator()(int i, int j, int k) {…}
…};
```

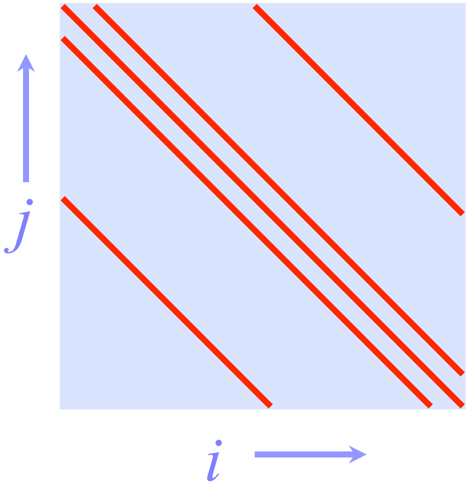
- Plugs into **`Array`**:

```
Array<2, R3Tensor<2>, Brick> t(10,10);
Array<2, double, Brick> s(10,10);
…
s = t.comp(0,2,1);
```

Add New Engine Type

- 2D square tridiagonal with fringes
  - Don’t store zeroes
  - Store 5 vectors of values
- Scalar indexing function:

```
template <class T>
class Engine<2, T, TridFringeTag> {…
  const T &operator()(int i, int j) {
      If (i,j) intersects red line, return data[band][i].
      If not, return zero_m;
  …}
  T *data[5];
  T zero_m; …
```

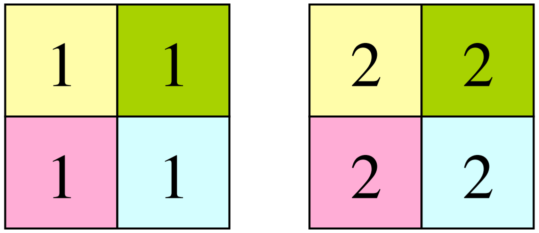


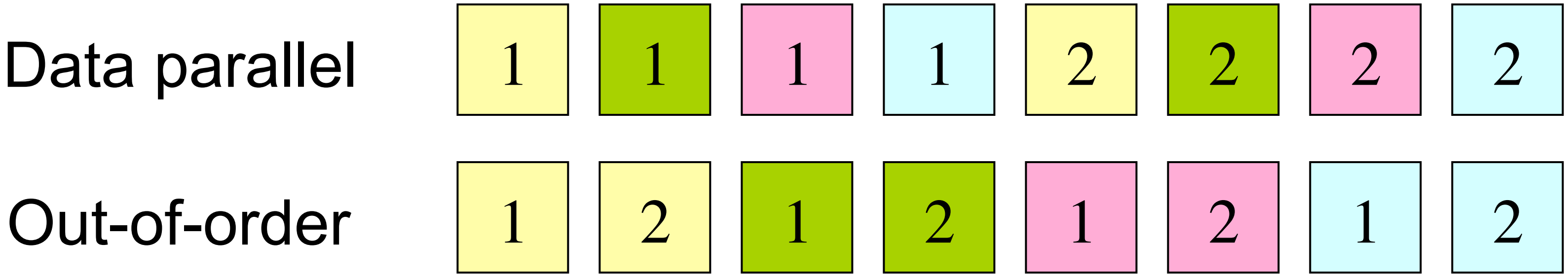
- Plugs into **Array** expression system

Parallel Evaluation

- Data parallel syntax is great for expressiveness, but not great for cache:

```
a = b + 2 * c;      ①
c = 0.5 * (a - b);  ②
```





Out of order execution can yield a 2-2.5x speedup

Compilation Consequences

- Must compile all template instances used
  - Classes
  - Functions
- Nearly nothing in **`libpooma.a`**
- Open source by definition
- More compile errors (but fewer runtime bugs)
- Each new expression generates new code to compile

No-Cost Software

- POOMA
  - http://www.acl.lanl.gov/pooma
- Portable Expression Template Engine (PETE)
  - Standalone package
  - http://www.acl.lanl.gov/pete

BSD-style license: free for any use,
commercial or non-commercial