UltimateSoA

a trivial SOA binding to your beloved OO Data Hierarchy

Vincenzo Innocente

SFT/CERN

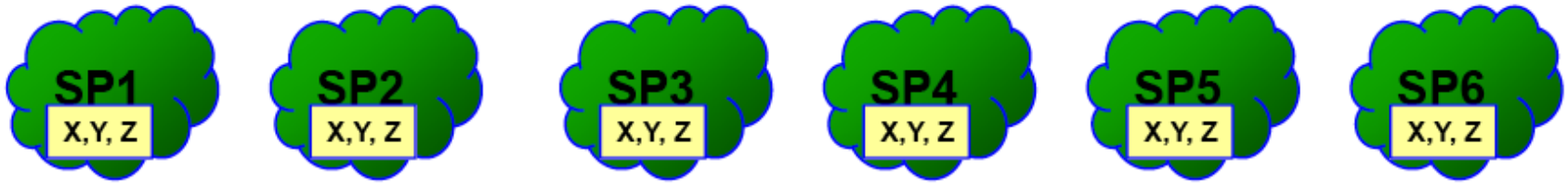
Summary

- Syntactic sugar atop esoteric language features

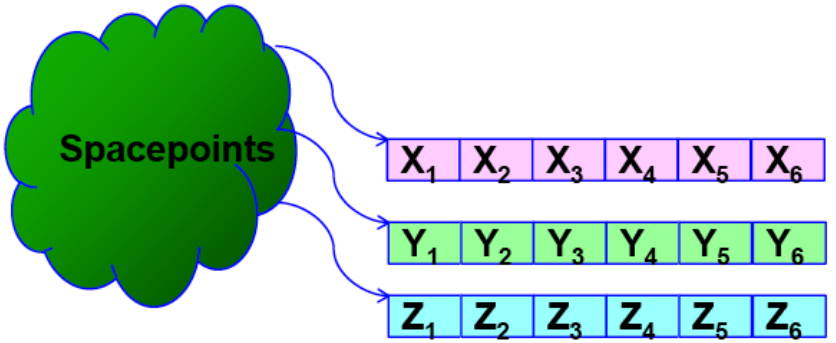
- Deliverable:
  - One header file: ~100 lines
    - Iterator implementation is longer
  - Value (CoCoMo): 2.5K CHF?
- Usable. Not (yet) a product for GPR
  - More a proof of concept

Data Organization: AoS vs SoA

- Traditional Object organization is an Array of Structure
  - Abstraction often used to hide implementation details at object level



- Difficult to fit stream computing
- Better to use a Structure of Arrays



- OO can wrap SoA as the AoS
  - Move abstraction higher
  - Expose data layout to the compiler
- Explicit copy in many cases more efficient
  - (notebooks vs whiteboard)

Object Model (Vector3D)

```
template<typename T> class Vector3D {
public:
….
template<typename V> Vector3D<T> & operator+=(V b) {
    xi+=b.x(); yi+=b.y(); zi+=b.z();  return *this;
  }
…
private:
  T xi; T yi; T zi;
};

template<typename T1, typename T2>
inline auto operator+(Vector3D<T1> a, Vector3D<T2> b)
  -> Vector3D<typename std::remove_const<typename std::remove_reference<decltype(a.x()+b.x())>::type>::type> {
  using V = Vector3D<typename std::remove_const<typename std::remove_reference<decltype(a.x()
+b.x())>::type>::type> ;
  V r=a; return r+=b;
}

template<typename V1,typename V2 >
inline
auto dot(V1 const & a, V2 const & b)  ->decltype(a.x()*b.x()) {
  return a.x()*b.x() + a.y()*b.y() + a.z()*b.z();
}
```

Object Model (classical Particle)

```
template <typename T, template<typename> class Vec=Vector3D > class Particle {
public:
 using T3D = Vec<T>;
…
Particle(T m, T3D pos, T3D vel, T3D acc) : m_pos(pos), m_vel(vel), m_acc(acc), m_mass(m) {}

void update(value dt=value(1)) {
   auto a =  value(0.5)*dt*dt*m_acc;
   m_pos += dt*m_vel + a;
   m_vel += dt*m_acc;
  }

  void scatter(unsigned int i, value wallPos) {
   m_pos[i] = wallPos - (m_pos[i]-wallPos);
   m_vel[i] = -m_vel[i];
  }

private:
  T3D m_pos;
  T3D m_vel;
  T3D m_acc;
  T m_mass;
  T m_charge;
};
```

Object Model (Line, Plane)

```
template<typename T> struct Plane : public Surface {
  Plane(): p(),n(){}
  Plane(Point<T> ip, Vect<T> in) : p(ip),n(in){}
  Point<T> p;
  Vect<T> n;
};

template<typename T> struct Line final : public Trajectory {
…..
  Point<T> p;
  Vect<T> c;

  Point<value> go(value t) const { return p + c*t; }
};
// return t for the point of plane-line crossing
template<typename T1, typename T2>
inline
float cross(Plane<T1> const & plane, Line<T2> const & line) {
  return dot(plane.n,plane.p-line.p)/dot(plane.n,line.c);
}

// return distance of a point from a plane
template<typename T1, typename T2>
inline
float distance(Plane<T1> const & plane, Point<T2> const & point) {
  return dot(plane.n,point-plane.p);
}
```

Object Model (Generic class)

```
template<typename Float=float, typename Int=int, typename String=std::string, typename Bool=unsigned char,
typename VecInt = std::vector<int>>
struct DataT {
  DataT(){}
  DataT(Float ia, Float ib, Float ic, Int is, Int it, String inam, VecInt iw, Bool io) :
    a(ia),b(ia), c(ia), status(is), type(it), name(inam), what(iw), ok(io){}

  void reset() { ok= false;}

  Float a,b,c;
  Int status;
  Int type;
  String name;
  VecInt what;
  Bool ok;

};

template<typename Cont> void comp(Cont & cont) {
for (auto i=0U; i<cont.size(); ++i) cont[i].a = cont[i].b*cont[i].c;
}

template<typename Cont> void reset(Cont & cont) { for (auto && d : cont) d.reset();}
```

Basic Concept

```
Vector3D<float> va{-3, 0, 2.};              // standard local vectors
Vector3D<float> vb{1, 2, 3.14};

float x[5]={1.f}, y[5]={1.f}, z[5]={1.f};          // sort of SoA

Vector3D<float const&>  v1(x[0],y[0],z[0]);   //   right hand binding
Vector3D<float&>        v5(x[4],y[4],z[4]);   //   left hand binding

v5 = v1+vb-va;    // just works!
```

- SoA implemented as a hierarchy of std::tuple of std::vector
  - matching the *natural* data-model hierarchy
- binding through the constructors

Syntactic Sugar

```
// make sure to support references as template argument
template<typename T> class Vector3D {
public:
  using value = typename std::remove_const<typename std::remove_reference<T>::type>::type;
  using  ref = typename std::add_lvalue_reference<T>::type;
  using cref = typename std::add_lvalue_reference<typename std::add_const< T>::type>::type;
  constexpr Vector3D(){}
// binding through the constructors
  constexpr Vector3D(T ix, T iy, T iz) : xi(ix), yi(iy), zi(iz) {}
  template<typename V>
  constexpr Vector3D(V v) : xi(v.x()), yi(v.y()), zi(v.z()) {}
  ....
 cref z() const { return zi;}
 ref x() { return xi;}
  ...
};
```

```
// declare SOA binding in a Trait class
template<typename T>
struct UltimateSoaTraits<Vector3D<T>> {
  using CREF = Vector3D<T const &>;
  using REF = Vector3D<T&>;
  using VAL = Vector3D<T>;
  // same order as constructor
  using SOATUPLE = std::tuple<UltimateSoa<T>, UltimateSoa<T>, UltimateSoa<T>>;
};
```

```
// declare SOA binding in a Trait cla
template<typename T>
struct UltimateSoaTraits<Line<T>> {
  using CREF = Line<T const &>;
  using REF = Line<T&>;
  using VAL = Line<T>;
  using V = Vect<T>;
  using SOATUPLE = std::tuple<UltimateSoa<V>, UltimateSoa<V>>;
};
```

```
// instantiate containers (instead of std::vector<T>)
template<typename T>
using Container = UltimateSoa<T>;
Container<float>  f(10);
Container<Vect<float>> vv(10);
Container<Line<float>> vl(10);
```

SoA with AoS interface

```
template<typename T, bool=!std::is_arithmetic<T>::value> class UltimateSoa {};

template<typename T>
class UltimateSoa<T, true> {
public:
  using VAL =  typename UltimateSoaTraits<T>::VAL;
  using CREF = typename UltimateSoaTraits<T>::CREF;
  using REF =  typename UltimateSoaTraits<T>::REF;
  using Data = typename UltimateSoaTraits<T>::SOATUPLE;

  template<typename V, std::size_t... I>
  V t2r_impl(unsigned int j, std::index_sequence<I...>) {
    return V(std::get<I>(m_data)[j] ...);
  }
  REF operator[](unsigned int j) {
    return t2r_impl<REF>(j,std::make_integer_sequence<std::size_t,std::tuple_size<Data>::value>{});
  }
  CREF operator[](unsigned int j) const {
   return t2r_impl<CREF>(j,std::make_integer_sequence<std::size_t,std::tuple_size<Data>::value>{});
  }
  .....

  Data m_data;
  unsigned int m_n=0;
};

template<typename T>
class UltimateSoa<T, false> : public details_UltimateSoa::AVector<T> { ...};
```

Example/test

```
template<typename T> using Container = UltimateSoa<T>;
// template<typename T> using Container = std::vector<T>;

#include <iostream>
int main() {
  Container<float>  f(10);
  Container<Vect<float>> vv(10);
  Container<Line<float>> vl(10);

  using V = Vect<float>;
  using L = Line<float>;

  std::cout << "size = " << vl.size()<< std::endl;
  for (auto i=0U; i<vl.size(); ++i) {
    auto val = float(i);
    vl[i] = L(V{val,val,val},V{val,val,val});
  }

for (auto i=0U; i<vl.size(); ++i) vv[i] = vl[i].go(3.4f);
```

```
 using std::swap;
 swap(vl[4],vl[7]);
 std::cout << vl[4].p.y() << ' ' << vl[7].p.y() << std::endl;

for (auto && l : vl) l.p.y() = -l.p.x();
 vl[5].p.y() = 23;

 auto && m = std::max_element(vl.begin(),vl.end(),
                              [](auto a, auto b){ return a.p.y()<b.p.y();});
 std::cout << "max " << (*m).p.x() << ' ' << (*m).p.y() << std::endl;


 for (auto && l : vl) std::cout << l.p.y ()<< ' ';
 std::cout << std::endl;
 std::sort(vl.begin(),vl.end(),[](auto a, auto b){ return a.p.y()<b.p.y();});

  for (auto && l : vl) std::cout << l.p.y ()<< ' ';
  std::cout << std::endl;

 for (auto && l : vl) std::cout << l.p.z() << ' ';
 std::cout << std::endl;
```

Binding for generic DoD

```
using Data = DataT<>;
template<> struct UltimateSoaTraits<Data> {

 template<typename ...Args>
 static auto makeRefT(DataT<Args...>) ->  DataT<typename std::add_lvalue_reference<Args>::type...> {
    return  DataT<typename std::add_lvalue_reference<Args>::type...>();
  }

 template<typename ...Args>
 static auto makeCRefT(DataT<Args...>) ->  DataT<typename std::add_lvalue_reference<typename std::add_const<Args>>::type...> {
  return  DataT<typename std::add_lvalue_reference<typename std::add_const<Args>>::type...>();
  }

 using REF = decltype(makeRefT(Data()));
 using CREF = decltype(makeCRefT(Data()));

 using SOATUPLE = std::tuple<UltimateSoa<float,false>,UltimateSoa<float,false>,UltimateSoa<float,false>,
                             UltimateSoa<int,false>,UltimateSoa<int,false>,UltimateSoa<std::string,false>,
                             UltimateSoa<std::vector<int>,false>, UltimateSoa<unsigned char,false>
                            >;

};

using DoDContainer = UltimateSoa<Data,true>;
```

Alternative/prior works

- Do it yourself
- boost::zip_iterator (+above)
- INTEL's arrow-street
- VC, Cilk++, OpenCL
- Root trees

Conclusion

- Providing a SoA with an interface that mimics an AoS is trivial
- With some effort a product for GPR can be developed
  - Some wide-spread coding habits may be difficult, if not impossible, to support.
- Is it worth?
  - C++ is a rich language: a different implementation may better fit some specific use-cases
  - In any case the object-model (and eventually the user code) will require some adaptation