C++ techniques from AFQMC

Alfredo A. Correa

Lawrence Livermore National Laboratory

November 6, 2018

Prepared by LLNL under Contract DE-AC52-07NA27344. Supported by the DOE and NNSA's Exascale Computing Project (17-SC-20-SC).
LLNL-PRES-761145

Contents

- OOP / Variant / Values
- Zip Iterator
- MPI3
- Multi Arrays
- Backend / Dispatching / BLAS
- Allocators
- Shared Memory
- GPU

Classic Polymorphism (OO). The 90's

```
struct base{virtual void f()=0;};
struct derived1{ void f(){...abc...} override;}
struct derived2{ void f(){...xyz...} override;}
```

Classic Polymorphism (OO). The 90's

```
struct base{virtual void f()=0;};
struct derived1{ void f(){...abc...} override;}
struct derived2{ void f(){...xyz...} override;}
```

```
base* a1 = opt1?new derived1:new derived2;
base* a2 = opt2?new derived1:new derived2;
```

Classic Polymorphism (OO). The 90's

```
struct base{virtual void f()=0;};
struct derived1{ void f(){...abc...} override;}
struct derived2{ void f(){...xyz...} override;}
```

```
base* a1 = opt1?new derived1:new derived2;
base* a2 = opt2?new derived1:new derived2;
```

```
a1 -> f();
a2 -> f();
```

Classic Polymorphism (OO). The 90's

```
struct base{virtual void f()=0;};
struct derived1{ void f(){...abc...} override;}
struct derived2{ void f(){...xyz...} override;}
```

```
base* a1 = opt1?new derived1:new derived2;
base* a2 = opt2?new derived1:new derived2;
```

```
a1 -> f();
a2 -> f();
```

```
delete derived1;
delete derived2;
```

Pointer semantics, allocations, open (runtime) set, hierarchies, single dispatching

Generic Programming with Templates, the STL era

```
// no base, but common syntax (later concept)
struct behavior1{ void f(){...abc...};}
struct behavior2{ void f(){...xyz...};}
template<class SomeBe> void f(SomeBe b){b.f();}
```

Generic Programming with Templates, the STL era

```
// no base, but common syntax (later concept)
struct behavior1{ void f(){...abc...};}
struct behavior2{ void f(){...xyz...};}
template<class SomeBe> void f(SomeBe b){b.f();}
```

```
behavior1 b1; // constructor
behavior2 b2; // constructor
```

Generic Programming with Templates, the STL era

```
// no base, but common syntax (later concept)
struct behavior1{ void f(){...abc...};}
struct behavior2{ void f(){...xyz...};}
template<class SomeBe> void f(SomeBe b){b.f();}
```

```
behavior1 b1; // constructor
behavior2 b2; // constructor
```

```
f(b1);
f(b2);
```

Value semantics, no allocations (stack friendly), open (compiletime) set, no hierarchies, multiple dispatching, return covariance

Value Semantics

```
a;  b;  c;
b = a;
c = b;
assert(f(b) ≡ f(c));
mutate(a);
assert(f(b) ≡ f(c));
```

Value Semantics

```
a; b; c;
b = a;
c = b;
assert(f(b) ≡ f(c));
mutate(a);
assert(f(b) ≡ f(c));
```

Even the simplest algorithm requires/assumes the concept of a well-behaved value

```
template<class T> swap(T& t1, T& t2){
    auto T = t1; t1 = t2; t2 = tmp;
}
```

3000 years of mathematical intuition and algorithms, reason locally. Plus you can use the STL, Containers,

Other semantics in C++

- Deep copy *vs.* shallow copy, "optimization" of copy
- Clone methods. `d2 = d1 -> clone();`
- Factories, `derived*d1 = Factory("registered");`
- Copy-on-write, copy-on-demand, shared-ownership, shared-state.

```
a = b; a.copy(b); a.really_copy(b); a.NoJoke_copy(b);
```

- No true random access

Great flexibility at a heavy cost, hard to reason, hard to do math, hard to write algorithms.

Besides

We already have pointers, smart pointer, references and iterators for all that. Most classes should behave like values and not their own references and pointers.

Regular Types represent value types (Stepanov Regularity)

```
struct R{
    R();
    R(R const& r);
    ~R();
    operator=(R const& r);
    Bool operator==(R const& r) const;
    Bool operator!=(R const& r);
    Bool operator<(R const& r); // or std::less
};
```

... and also the semantics is the expected one.

Quick test for your types, does this compile and does the expected thing?

```
std::vector<R> v={r1, r2, r3};
std::sort( v.begin(), v.end() );
assert(std::is_sorted(v.begin(), v.end());
```

When is a good idea to use OO (virtual, new, $->$, etc)?

(spoiler: almost never to do math or a simulation)

When is a good idea to use OO (virtual, new, $->$, etc)?

(spoiler: almost never to do math or a simulation) Perhaps if:

+ Immutable (but polymorphic) state
+ Shared/singleton objects/representations of logic
+ 'In-the-heap-anyway'
+ Open hierarchies (e.g. open behavior, cross compilation-boundaries)
+ Exceptions (`std::runtime_error`)
+ Resources (`std::pmr::memory_resource`)
+ Devices (`std::iostream`)

When is a good idea to use OO (virtual, new, $->$, etc)?

(spoiler: almost never to do math or a simulation) Perhaps if:

+ Immutable (but polymorphic) state
+ Shared/singleton objects/representations of logic
+ 'In-the-heap-anyway'
+ Open hierarchies (e.g. open behavior, cross compilation-boundaries)
+ Exceptions (`std::runtime_error`)
+ Resources (`std::pmr::memory_resource`)
+ Devices (`std::iostream`)

Perhaps not if:

- - - - Value Semantics
- - Mutability
- - Local reasoning

Remember, **virtual** was never used in the STL and it still went a long way

```
$ grep -R virtual /usr/include/c++/*
```

Another Runtime Polymorphism

```
struct A{void f(){...};};
struct B{void f(){...};}
template<class T> void f(T t){t.f();}
```

[1] See my article:
https://arne-mertz.de/2018/06/functions-of-variants-are-covariant/

Another Runtime Polymorphism

```
struct A{void f(){...};};
struct B{void f(){...};}
template<class T> void f(T t){t.f();}
```

```
std::variant<A, B> v1 = opt1?A{}:B{};
std::variant<A, B> v2 = opt2?A{}:B{};
```

[1] See my article:
https://arne-mertz.de/2018/06/functions-of-variants-are-covariant/

Another Runtime Polymorphism

```
struct A{void f(){...};};
struct B{void f(){...};}
template<class T> void f(T t){t.f();}
```

```
std::variant<A, B> v1 = opt1?A{}:B{};
std::variant<A, B> v2 = opt2?A{}:B{};
```

```
std::visit( f, v1 );
std::visit( f, v2 );
```

Value semantics!, no allocations, closed sets, no hierarchies, function evaluation is not idiomatic, covariance is hard. [1]
Many OO classes have been replaced by `std::variant`-based types.

[1] See my article:
https://arne-mertz.de/2018/06/functions-of-variants-are-covariant/

Zip Iterators

```
#include<alf/iterator/zipper.hpp>

std::vector<Double> v = ...;
std::vector<Double> w = ...;
assert( v.size() == w.size() );
std::sort(
    zip(begin(v), begin(w)),
    zip(end(v) , end(w) )
);

*zip(begin(v), begin(w)) = std::pair{...};
```

MPI3 wrapper: Traffics in ranges, values are communicated

In STL

```
std::copy(origin.begin(), origin.end(), target.begin());
```

In STL

```
std::copy(origin.begin(), origin.end(), target.begin());
```

```
int mpi3::main(int, char*[], mpi3::communicator world){
    mpi3::communicator w = world; // communicator& w = world;
    assert(w.size() == 2);
    if(w.rank() == 0){
        std::vector<double> v = {1.,2.,3.};
        w.send(v.begin(), v.end(), 1); // to rank 1
    }else if(world.rank() == 1){
        std::vector<double> v(3);
        w.receive(v.begin(), v.end(), 0); // from rank 0
//        w.receive(v.begin(), 0) // from rank
    }
}
```

Multi Dimensional Array Container

```
#include<alf/multi/array.hpp>
Double C[2][2] = {
    {150, 16},
    { 30, 1},
};
multi::array<Double, 2, Allocator> A = {{...}, {...}};
std::sort(A.begin(0), A.end(0)); // sort by rows
std::sort(B.begin(1), B.end(1)); // sort by cols

multi::array_ref<Double, 2> A = C; // reference sematics is flagged
```

Thin layer of generic code in QMCPack's Linear Algebra

```
dgemm(int l, int n, int m, double alpha, double* a, int lda,
    ↪ double* b, int ldb, double beta, double* c, int ldc)
```

Thin layer of generic code in QMCPack's Linear Algebra

```
dgemm(int l, int n, int m, double alpha, double* a, int lda,
    ↪ double* b, int ldb, double beta, double* c, int ldc)
```

better

```
template<typename T> // T is a number
Ohmms::gemm(int l, int n, int m, T alpha, T* a, int lda, T* b,
    ↪ int ldb, T beta, T* c, int ldc)
```

Thin layer of generic code in QMCPack's Linear Algebra

```
dgemm(int l, int n, int m, double alpha, double* a, int lda,
    ↪ double* b, int ldb, double beta, double* c, int ldc)
```

better

```
template<typename T> // T is a number
Ohmms::gemm(int l, int n, int m, T alpha, T* a, int lda, T* b,
    ↪ int ldb, T beta, T* c, int ldc)
```

best

```
template<class T, class Matrix1, class Matrix2, class Matrix3>
ma::gemm(Matrix1&& A, Matrix2&& B, Matrix3&& C, T ...){
    assert( ...check size compatibility ...);
    using Ohmms::gemm;
    return gemm(A.shape()[0], B.shape()[0], C.shape()[1],
    ↪ alpha, A.data(), A.strides()[0], B.data(), B.strides
    ↪ ()[0], beta, C.data(), C.strides()[0]
}
```

Generic code in QMCPack Linear Algebra Backend

```
multi::array<double, 2> const A=...
multi::array<double, 2> const B=...
multi::array<double, 2> C=...
ma::gemm(A, B, C);
```

```
multi::array<double, 3> A=...;
multi::array_ref<double, 2> const B=...
multi::array<double, 2> C=...
ma::dgemm(A[2], B, C({3, 8}, {5, 10}) ); // subrange of C
```

Shared Memory and GPU Allocators

```
multi::array<double, 2, mpi3::shm::allocator<double> > A
    ↪ ({1000,1000}, &world);

// A.data() is a mpi3::shm::pointer<double>

multi::array<double, 2, gpu::allocator<double> > B
    ↪ ({1000,1000});

// B.data() is a gpu::pointer<double>
```

Compound interest of Generic Programming

```
multi::array<double, 2, gpu::allocator<double> > A
    ↪ ({1000,1000});
multi::array<double, 2, gpu::allocator<double> > B
    ↪ ({1000,1000});
multi::array<double, 2, gpu::allocator<double> > C
    ↪ ({1000,1000});

ma::gemm(A, B, C);  // ??? .data() is a gpu::pointer<double>
```

Compound interest of Generic Programming

```
multi::array<double, 2, gpu::allocator<double> > A
    ↪ ({1000,1000});
multi::array<double, 2, gpu::allocator<double> > B
    ↪ ({1000,1000});
multi::array<double, 2, gpu::allocator<double> > C
    ↪ ({1000,1000});

ma::gemm(A, B, C); // ??? .data() is a gpu::pointer<double>
```

A new usage needs a new back-end, but nothing else!

```
template<typename T> // T is a number type
gpu::gemm(int l, int n, int m, T alpha, pointer<T> a, int lda,
    ↪ pointer<T> b, int ldb, T beta, pointer<T> c, int ldc){
    CudaBlasGemm(l, n, m, a.get(), lda, b.get(), ...);
}
```

Conclusions

- Try not to add new virtual functions
- Give your arithmetic types value semantics
- Use constructors and assignment instead of (pointer-) factories
- (make_* functions are ok)
- OO can coexists with Generic Programming and slowly fade away
- Use the stack for objects, or be stack-friendly
- Use dynamic memory as an internal implementation detail
- Use algorithms (100+ STL algorithms)
- (chances are your loops are in essence existing algorithms)
- Use containers, even custom containers/iterators/pointers
- Use dynamic memory generically through allocators