C++ Best Practices 101: A miniQMC Case Study

Nevin ":-)" Liber, nliber@anl.gov

U.S. DEPARTMENT OF ENERGY

Argonne NATIONAL LABORATORY

This presentation was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of two U.S. Department of Energy organizations (Office of Science and the National Nuclear Security Administration) responsible for the planning and preparation of a capable exascale ecosystem, including software, applications, hardware, advanced system engineering, and early testbed platforms, in support of the nation's exascale computing imperative. Additionally, this presentation used resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357.

U.S. DEPARTMENT OF ENERGY

Argonne NATIONAL LABORATORY

Best Practices

- Generally accepted as superior alternative
  - Easier to reason about
  - Fewer bugs
- Reasonable people can disagree
- Need justification when doing something different
  - If done often, update Best Practices

Resource Acquisition Is Initialization

RAII

Resource Acquisition Is Initialization (RAII)

```cpp
int main(int argc, char** argv)
{
    int error_code=0;
    Kokkos::initialize(argc, argv);
    { //Begin kokkos block

        //...
    } //end kokkos block
    Kokkos::finalize();
    return error_code;
}
```

```cpp
int main(int argc, char** argv)
{
    int error_code=0;

    { //Begin kokkos block
        Kokkos::ScopeGuard _(argc, argv);
        //...
    } //end kokkos block

    return error_code;
}
```

Resource Acquisition Is Initialization (RAII)

- Constructor does initialization
- Destructor does finalization
  - Language guarantees destructor is called on scope exit
- Useful as guards or members

Resource Acquisition Is Initialization (RAII)

```cpp
int main(int argc, char** argv)
{
    int error_code=0;
    Kokkos::initialize(argc, argv);
    { //Begin kokkos block

        //...
    } //end kokkos block
    Kokkos::finalize();
    return error_code;
}
```

```cpp
int main(int argc, char** argv)
{
    int error_code=0;

    { //Begin kokkos block
        Kokkos::ScopeGuard _(argc, argv);
        //...
    } //end kokkos block

    return error_code;
}
```

- Guard
  - Not concerned how the block is exited (early return, exception, etc.)
    - termination functions `abort`, `exit`, `_Exit`, `quick_exit`, `terminate` notwithstanding

U.S. DEPARTMENT OF ENERGY

Argonne NATIONAL LABORATORY

Resource Acquisition Is Initialization (RAII)

- **Best Practice**
  - Use whenever a set of operations has to be paired up
  - Write your own if not provided
    - One class - one responsibility

Smart Pointers

`unique_ptr`, `shared_ptr`

unique_ptr

```cpp
struct Mover
{
    // ...

    /// single particle orbitals
    SPOSet* spo;
    std::unique_ptr<SPOSet> spo;

    // ...

    /// destructor
    ~Mover()
    {
        if (spo != nullptr)
            delete spo;
    }
};
```

Smart Pointers

- Manage heap allocation
- Use RAII
  - Constructor takes ownership of a heap allocated object/array
    - Does not perform the allocation itself
      - Use `make_`*`smart`* to replace raw **`new`**

Heap Allocations

- Why allocate on the heap in C++?
  - The size cannot be determined at compile time
    - Exact type not known until run time (e.g. polymorphism)
    - Variable length arrays, containers, etc.
    - Degenerate cases: object too large for stack, static space, etc.
  - The lifetime does not fit within a scope {}
  - Move semantics for immovable objects
    - ~~`atomic<int> Alpha() { return atomic<int>(0) ; }`~~
      `unique_ptr<atomic<int>> Alpha() { return unique_ptr(new atomic<int>(0)); }`

Heap Allocations

- Why allocate on the heap in C++17?
  - The size cannot be determined at compile time
    - Exact type not known until run time (e.g. polymorphism)
      - Consider `std::variant` for a bounded set of types
    - Variable length arrays
  - The lifetime may be longer than a scope {}
    - Use `std::optional` for lifetimes shorter than a scope
  - Move semantics for immovable objects
    - Mandatory RVO (Return Value Optimization) eliminates some of that need
    -

```
atomic<int> Alpha() { return atomic<int>(0) ; }
```

~~`unique_ptr<atomic<int>> Alpha() { return unique_ptr(new atomic<int>(0)); }`~~

unique_ptr

unique_ptr

- Unique (single) ownership
- Moveable but not copyable
    - Aggregates of `unique_ptr` by default moveable but not copyable
- By default `unique_ptr` destructor destroys held object
    - Calls `~T()`
    - Returns space to the heap

unique_ptr

```
struct Mover
{
    // ...

    /// single particle orbitals
    SPOSet* spo;
    std::unique_ptr<SPOSet> spo;

    // ...

    /// destructor
    ~Mover()
    {
        if (spo != nullptr)
            delete spo;
    }
};
```

- Mover aggregates (has as a member) unique_ptr<SPOSet>
    - Hand written destructor eliminated
        - Decremental development (Kevlin Henney)

U.S. DEPARTMENT OF ENERGY

Argonne NATIONAL LABORATORY

RAII

- Best Practice
  - Most classes should not have an explicit (non-defaulted) destructor
    - `~T() = default;`
      - `virtual`
      - Visibility (from `public` to `protected` or `private`)
      - Non-inlined (in .cpp file)
      - Rule of 5
  - Those that do should be managing a single resource via RAII

Rule of 5 / 3 / 0

Rule of 3

- C++98
  - If you declare any of these you should declare all of these

```
struct Alpha
{
    Alpha(Alpha const&);             // Copy constructor
    Alpha& operator=(Alpha const&); // Copy assignment operator


    ~Alpha();                        //  Destructor
};
```

  - Compiler will implicitly declare versions of these special member functions if you do not

Rule of 5

- C++11
    - If you declare any of these you should declare all of these

```
struct Alpha
{
    Alpha(Alpha const&);               // Copy constructor
    Alpha& operator=(Alpha const&); // Copy assignment operator
    Alpha& operator=(Alpha&&);        // Move assignment operator
    Alpha(Alpha&&);                   // Move constructor
    ~Alpha();                          // Destructor
};
```

- Compiler will *not necessarily* implicitly declare versions of these special member functions if you do not

Implicitly Declared C++98

- Copy constructor `T(T const&)`
  - Explicitly or implicitly declared
- Copy assignment operator `T& operator=(T const&)`
  - Explicitly or implicitly declared
- Destructor `~T()`
  - Explicitly or implicitly declared

Implicitly Declared C++11

- Copy constructor, copy assignment operator and destructor
  - Explicitly or implicitly declared (C++98 rules still apply)
- Move constructor / assignment operator implicitly declared if
  - Not explicitly declared
  - No user-declared copy constructor
  - No user-declared copy assignment operator
  - No user-declared move assignment operator / constructor
  - No user-declared destructor
- Implicitly declared copy constructor and copy assignment operator are deleted
  - Declared (explicitly or implicitly) move constructor or move assignment operator

Implicitly Declared C++11

- Backwards compatibility with C++98 classes
- C++-*utopia* rules
    - Rule of 5
    - If you explicitly declare one of the 5, you always have to declare all copy/move constructor/assignment operator
    - C++11 rules have been deprecated since C++11
        - Might be un-deprecated in C++23

Implicitly Declared Deleted

- …
- Implicitly declared copy / move constructor / assignment operator is deleted
    - If any aggregated members have inaccessible or deleted copy / move constructor / assignment operator

U.S. DEPARTMENT OF ENERGY

Argonne NATIONAL LABORATORY

Implicitly Declared Deleted

```
class Bravo
{
    std::unique_ptr<Charlie> c;
    //...
};
```

- Because `unique_ptr` has a deleted copy constructor / copy assignment operator
    - `Bravo` *implicitly* has a deleted copy constructor / copy assignment operator

U.S. DEPARTMENT OF ENERGY

Argonne NATIONAL LABORATORY

Implicitly Declared

- Easier to get correct
- Easier to reason about
- Easier for compilers to optimize
- Less verbose

Rule of 0

- Most classes should not have *any* user-declared copy / move constructor / assignment operator or destructor
  - Take advantage of implicitly declared special member functions
  - Take advantage of implicitly declared deleted special member functions

U.S. DEPARTMENT OF ENERGY

Argonne NATIONAL LABORATORY

Rule of 0

- **Best Practice**
  - Strive for Rule of 0
    - No user-declared copy / move constructor / assignment operator or destructor
  - Otherwise, Rule of 5 (hopefully rarely)
    - Explicitly declare copy / move constructor / assignment operator and destructor

unique_ptr

(Again)

unique_ptr

```
struct einspline_spo     : SPOSet { /* ... */ };
struct einspline_spo_ref : SPOSet { /* ... */ };
```

```
SpoSet* build_spo(...)
{
    auto my_spo = new einspline_spo(...);
    // ...
    return dynamic_cast<SPOSet*>(my_spo);
}
```

```
std::unique_ptr<SpoSet> build_spo(...)
{
    std::unique_ptr<einspline_spo> my_spo(new einspline_spo);                // C++11
    std::unique_ptr<einspline_spo> my_spo(std::make_unique<einspline_spo>()); // C++14

    //...

    return my_spo;  // Implicit upcasting
}
```

unique_ptr

```cpp
struct einspline_spo     : SPOSet { /* ... */ };
struct einspline_spo_ref : SPOSet { /* ... */ };

std::unique_ptr<SpoSet> build_spo(...)
{
    std::unique_ptr<einspline_spo> my_spo(new einspline_spo);                     // C++11
    std::unique_ptr<einspline_spo> my_spo(std::make_unique<einspline_spo>());  // C++14

    //...

    return my_spo;  // Implicit upcasting
}
```

- **Best Practice**
  - Put object into smart pointer as soon as possible
  - No raw `new` (or `delete`) whenever possible
  - Throw away information (derived type) as late as possible
  - No explicit casting whenever possible

Casts

dynamic_cast

```cpp
struct einspline_spo     : SPOSet { /* ... */ };
struct einspline_spo_ref : SPOSet { /* ... */ };

SpoSet* build_spo(...)
{
    auto spo_main = new einspline_spo(...);
    // ...
    return dynamic_cast<SPOSet*>(spo_main);
}
```

Casts

- Casting is a many-to-one relationship
- Casting says that you know better than the type system
- Casting can easily lead to bugs
    - Undefined behavior

Undefined Behavior

- Behavior for which C++ imposes no requirements
  - Here be dragons
  - Anything is possible
    - A `bool` variable can be both `true` and `false`
    - Time travel

```
int table[4];
bool exists_in_table(int v)
{
    for (int i = 0; i <= 4; i++) {
        if (table[i] == v) return true;
    }
    return false;
}
```

- Compiler may assume `i == 5` never occurs and optimize this to always return `true`

U.S. DEPARTMENT OF ENERGY

Argonne NATIONAL LABORATORY

Undefined Behavior

- Two edged sword
  - Invoking undefined behavior means anything can happen!
  - Compilers may assume undefined behavior cannot happen
    - + Can make correct code better (faster and/or smaller)
  - + Sanitizers can detect incorrect code at runtime
  - + Sometimes defining behavior is more error prone
    - + Developers know they can write code which depends on it

Undefined Behavior

- `unsigned`
  - Addition / subtraction fully defined to wrap
    - Very reasonable and obvious definition
    - Highly error prone
      - Cannot differentiate between accidental wrapping and deliberate wrapping
        - No-false-positive sanitizers cannot differentiate either

Undefined Behavior

- Cannot always detect it
  - `strlen(char const* s)`
    - `s` must not be `nullptr`
    - `s` must be a valid pointer
    - `s` must point to something `‘\0’`-terminated

U.S. DEPARTMENT OF ENERGY

Argonne NATIONAL LABORATORY

Undefined Behavior

- **Best Practice**
  - Avoid invoking undefined behavior in your code
  - Don't define behavior just to avoid undefined behavior
    - Defined behavior should be "easy to use correctly and hard to use incorrectly"
  - `assert()` is your friend
    - Remember `assert()` is a macro and should be side-effect free
    - C++23-*ish* Contracts will also help

Casts

- Casting is a many-to-one relationship
- Casting says that you know better than the type system
- Casting can easily lead to bugs
    - Undefined behavior

U.S. DEPARTMENT OF ENERGY

Argonne NATIONAL LABORATORY

static_cast

- Conversion between types
  - May have a runtime cost
- Can be used to down cast (Base to Derived) if you know you have an object of the derived type
- “Safest”

```
enum class E { Five = 5, };

E e = E::Five;
std::cout << +static_cast<std::underlying_type_t<E>>(e) << '\n'; // 5
```

const_cast

- Remove `const`/`volatile`
- No run time cost
- Only legal if the original object is non-`const`
- Usually to interface with legacy code or C code
    - C: Array of pointers to non-`const` cannot be promoted to an array of pointers to `const`

```
// Warning in C++; invalid in C (see execv for details)
int A(int argc, char const* const argv[]) { return !!argv[argc]; }
int main(int argc, char* argv[]) { return A(argc, argv); }
```

const_cast

```cpp
int i = 2;
int const& ci = i;
++const_cast<int&>(ci);
std::cout << i << '\n'; // 3
```

reinterpret_cast

- “Pretend” the bits are really for the new type
  - No run time cost
  - Tends to be at the lowest abstraction levels
  - Dereferencing only legal for “similar” types according to type aliasing
    - See CppReference.com for details

U.S. DEPARTMENT OF ENERGY

Argonne NATIONAL LABORATORY

dynamic_cast

- Checked down / cross casting
  - Uses RTTI (Run Time Type Information)
    - Source type has to have at least one virtual function
    - Expensive
      - May be O(n) (n is the hierarchy depth)
      - `strcmp` of mangled names (gcc)

U.S. DEPARTMENT OF ENERGY

Argonne NATIONAL LABORATORY

dynamic_cast

- `dynamic_cast<T*>(p)` returns `nullptr` if `p` is not convertible to a pointer to `T`
- `dynamic_cast<T&>(u)` throws `std::bad_cast` if `u` is not convertible to a reference to **T**
- Down cast asks an object “Are you really type **T**?”
  - Not good OO design
- Cross cast (multiple inheritance) asks an object “Do you have capability C?”

U.S. DEPARTMENT OF ENERGY

Argonne NATIONAL LABORATORY

C style cast

- In the beginning…
- `(type)variable`
  - Combination of `static_cast`, `reinterpret_cast` & `const_cast`
  - Actually worse…

```
struct Base { virtual ~Base() = default; };
struct Derived : private Base {};

Derived d;
Base* b = (Base*)(&d);
```

- Functional cast (C++)
  - `type(variable)`
- And many, many more…

Casts

- Rarely a need to up cast
- `unique_ptr` does not define the equivalent of explicit casting operations
- **Best Practice**
  - Rarely cast

U.S. DEPARTMENT OF ENERGY

Argonne NATIONAL LABORATORY

dynamic_cast

```cpp
struct einspline_spo     : SPOSet { /* ... */ };
struct einspline_spo_ref : SPOSet { /* ... */ };

SpoSet* build_spo(...)
{
    auto my_spo = new einspline_spo(...);
    // ...
    return dynamic_cast<SPOSet*>(my_spo);
}
```

unique_ptr<T[]>

unique_ptr<T[]>

- `std::unique_ptr<char[]> up(new char[size]);`
  - Moveable
  - Size not queryable
  - Default initializes (doesn't zero) array
- `std::vector<char> v(size);`
  - Copyable, resizable, etc.
  - Size queryable
  - Value initializes (zeros) the array

unique_ptr<T[]>

- C++11

```
std::unique_ptr<char[]> up(new char[size]);
```

- C++14

```
std::unique_ptr<char[]> up(std::make_unique<char[]>(size));
std::unique_ptr<char[]> up(new char[size]);
```

  - `make_unique`
    - No raw **new**
    - Value initializes (zeros) the array

- C++20

```
std::unique_ptr<char[]> up(std::make_unique_default_init<char[]>(size));
```

shared_ptr

shared_ptr

- Reference counted
  - increment/decrement happen atomically
- Copyable
  - Copying increments the reference count
- Trackable / breaking cycles
  - `weak_ptr`
- Separate control block (deleter, strong/weak refCounts, etc.)
  - `make_shared` combine object and control block into one allocation

shared_ptr

- Useful when you do not know the last user of the data structure
  - Across threads
- Hard to reason about
  - Essentially a hidden global variable

InfoStream

(miniQMC)

InfoStream

- Wrapper around output streams
  - Turning on/off, redirecting to `cout`, `cerr`, file
- Hard to reason about
  - May or may not own a stream
  - Copy/move semantics incorrect
- Streams are non-trivial to wrap

InfoStream

```cpp
class InfoStream
{
public:
  InfoStream(std::ostream* output_stream)
      : prevStream(NULL), nullStream(new std::ostream(NULL)), ownStream(false)
  {
    currStream = output_stream;
  }

  InfoStream(InfoStream& in)
      : prevStream(NULL), nullStream(new std::ostream(NULL)), ownStream(false)
  {
    redirectToSameStream(in);
  }

  ~InfoStream();

  std::ostream& getStream(const std::string& tag = "") { return *currStream; }

  void flush() { currStream->flush(); }

private:
  // Keep track of whether we should delete the stream or not
  bool ownStream;

  std::ostream* currStream;

  // save stream during pause
  std::ostream* prevStream;

  // Created at construction. Used during pause
  std::ostream* nullStream;
};

template<class T>
inline InfoStream& operator<<(InfoStream& o, const T& val)
{
  o.getStream() << val;
  return o;
}
```

```cpp
class InfoStream
{
public:
  explicit InfoStream(std::ostream* output_stream)
      : currStream(output_stream)
  {}

  explicit InfoStream(InfoStream& in)
  {
    redirectToSameStream(in);
  }

  std::ostream& getStream() const { return *currStream; }

  void flush() const { currStream->flush(); }

private:
  // Keep track of whether we should delete the stream or not
  std::unique_ptr<std::ofstream> ownStream;

  std::ostream* currStream = nullptr;

  // save stream during pause
  std::ostream* prevStream = nullptr;

  // Used during pause
  static std::ostream nullStream;

  template<typename T>
  friend InfoStream& operator<<(InfoStream& o, const T& val)
  { return o.getStream() << val; }
};
```

InfoStream

```
class InfoStream
{
public:
  InfoStream(std::ostream* output_stream)
      : prevStream(NULL), nullStream(new std::ostream(NULL)), ownStream(false)
  {
    currStream = output_stream;
  }

  InfoStream(InfoStream& in)
      : prevStream(NULL), nullStream(new std::ostream(NULL)), ownStream(false)
  {
    redirectToSameStream(in);
  }

  ~InfoStream();

  std::ostream& getStream(const std::string& tag = "") { return *currStream; }

  void flush() { currStream->flush(); }

private:
  // Keep track of whether we should delete the stream or not
  bool ownStream;

  std::ostream* currStream;

  // save stream during pause
  std::ostream* prevStream;

  // Created at construction. Used during pause
  std::ostream* nullStream;
};

template<class T>
inline InfoStream& operator<<(InfoStream& o, const T& val)
{
  o.getStream() << val;
  return o;
}
```

InfoStream

```
class InfoStream
{
public:
  InfoStream(std::ostream* output_stream)
      : prevStream(NULL), nullStream(new std::ostream(NULL)), ownStream(false)
  {
    currStream = output_stream;
  }

  InfoStream(InfoStream& in)
      : prevStream(NULL), nullStream(new std::ostream(NULL)), ownStream(false)
  {
    redirectToSameStream(in);
  }

  ~InfoStream();

  std::ostream& getStream(const std::string& tag = "") { return *currStream; }

  void flush() { currStream->flush(); }

private:
  // Keep track of whether we should delete the stream or not
  bool ownStream;

  std::ostream* currStream;

  // save stream during pause
  std::ostream* prevStream;

  // Created at construction. Used during pause
  std::ostream* nullStream;
};

template<class T>
inline InfoStream& operator<<(InfoStream& o, const T& val)
{
  o.getStream() << val;
  return o;
}
```

- Single parameter constructors should be `explicit`

InfoStream

```
class InfoStream
{
public:
  InfoStream(std::ostream* output_stream)
      : prevStream(NULL), nullStream(new std::ostream(NULL)), ownStream(false)
  {
    currStream = output_stream;
  }

  InfoStream(InfoStream& in)
      : prevStream(NULL), nullStream(new std::ostream(NULL)), ownStream(false)
  {
    redirectToSameStream(in);
  }

  ~InfoStream();

  std::ostream& getStream(const std::string& tag = "") { return *currStream; }

  void flush() { currStream->flush(); }

private:
  // Keep track of whether we should delete the stream or not
  bool ownStream;

  std::ostream* currStream;

  // save stream during pause
  std::ostream* prevStream;

  // Created at construction. Used during pause
  std::ostream* nullStream;
};

template<class T>
inline InfoStream& operator<<(InfoStream& o, const T& val)
{
  o.getStream() << val;
  return o;
}
```

- Share a common sentinel

InfoStream

```cpp
class InfoStream
{
public:
  InfoStream(std::ostream* output_stream)
      : prevStream(NULL), nullStream(new std::ostream(NULL)), ownStream(false)
  {
    currStream = output_stream;
  }

  InfoStream(InfoStream& in)
      : prevStream(NULL), nullStream(new std::ostream(NULL)), ownStream(false)
  {
    redirectToSameStream(in);
  }

  ~InfoStream();

  std::ostream& getStream(const std::string& tag = "") { return *currStream; }

  void flush() { currStream->flush(); }

private:
  // Keep track of whether we should delete the stream or not
  bool ownStream;

  std::ostream* currStream;

  // save stream during pause
  std::ostream* prevStream;

  // Created at construction. Used during pause
  std::ostream* nullStream;
};

template<class T>
inline InfoStream& operator<<(InfoStream& o, const T& val)
{
  o.getStream() << val;
  return o;
}
```

- Use member initializers
  - Otherwise, does construct then assign
  - Requires default constructibility

InfoStream

```
class InfoStream
{
public:
  InfoStream(std::ostream* output_stream)
      : prevStream(NULL), nullStream(new std::ostream(NULL)), ownStream(false)
  {
    currStream = output_stream;
  }

  InfoStream(InfoStream& in)
      : prevStream(NULL), nullStream(new std::ostream(NULL)), ownStream(false)
  {
    redirectToSameStream(in);
  }

  ~InfoStream();

  std::ostream& getStream(const std::string& tag = "") { return *currStream; }

  void flush() { currStream->flush(); }

private:
  // Keep track of whether we should delete the stream or not
  bool ownStream;

  std::ostream* currStream;

  // save stream during pause
  std::ostream* prevStream;

  // Created at construction. Used during pause
  std::ostream* nullStream;
};

template<class T>
inline InfoStream& operator<<(InfoStream& o, const T& val)
{
  o.getStream() << val;
  return o;
}
```

- Declare member functions which do not modify state as `const`

U.S. DEPARTMENT OF ENERGY

Argonne NATIONAL LABORATORY

InfoStream

```
class InfoStream
{
public:
  InfoStream(std::ostream* output_stream)
      : prevStream(NULL), nullStream(new std::ostream(NULL)), ownStream(false)
  {
    currStream = output_stream;
  }

  InfoStream(InfoStream& in)
      : prevStream(NULL), nullStream(new std::ostream(NULL)), ownStream(false)
  {
    redirectToSameStream(in);
  }

  ~InfoStream();

  std::ostream& getStream(const std::string& tag = "") { return *currStream; }

  void flush() { currStream->flush(); }

private:
  // Keep track of whether we should delete the stream or not
  bool ownStream;

  std::ostream* currStream;

  // save stream during pause
  std::ostream* prevStream;

  // Created at construction. Used during pause
  std::ostream* nullStream;
};

template<class T>
inline InfoStream& operator<<(InfoStream& o, const T& val)
{
  o.getStream() << val;
  return o;
}
```

- Use `unique_ptr<ofstream>` for ownStream
  - Automatically deletes the stream on destruction if we own it
  - No explicit destructor needed for `InfoStream`
  - Makes `InfoStream` non-copyable

InfoStream

```
class InfoStream
{
public:
  InfoStream(std::ostream* output_stream)
      : prevStream(NULL), nullStream(new std::ostream(NULL)), ownStream(false)
  {
    currStream = output_stream;
  }

  InfoStream(InfoStream& in)
      : prevStream(NULL), nullStream(new std::ostream(NULL)), ownStream(false)
  {
    redirectToSameStream(in);
  }

  ~InfoStream();

  std::ostream& getStream(const std::string& tag = "") { return *currStream; }

  void flush() { currStream->flush(); }

private:
  // Keep track of whether we should delete the stream or not
  bool ownStream;

  std::ostream* currStream;

  // save stream during pause
  std::ostream* prevStream;

  // Created at construction. Used during pause
  std::ostream* nullStream;
};

template<class T>
inline InfoStream& operator<<(InfoStream& o, const T& val)
{
  o.getStream() << val;
  return o;
}
```

- Are move semantics correct?
  - No
    - Moving fundamental data types (ints, pointers, etc.) is the same as copying them
    - `currStream` may have multiple `InfoStream` owners
      - All deleting them in their destructors

U.S. DEPARTMENT OF ENERGY

Argonne NATIONAL LABORATORY

InfoStream

```
class InfoStream
{
public:
  InfoStream(std::ostream* output_stream)
      : prevStream(NULL), nullStream(new std::ostream(NULL)), ownStream(false)
  {
    currStream = output_stream;
  }

  InfoStream(InfoStream& in)
      : prevStream(NULL), nullStream(new std::ostream(NULL)), ownStream(false)
  {
    redirectToSameStream(in);
  }

  ~InfoStream();

  std::ostream& getStream(const std::string& tag = "") { return *currStream; }

  void flush() { currStream->flush(); }

private:
  // Keep track of whether we should delete the stream or not
  bool ownStream;

  std::ostream* currStream;

  // save stream during pause
  std::ostream* prevStream;

  // Created at construction. Used during pause
  std::ostream* nullStream;
};

template<class T>
inline InfoStream& operator<<(InfoStream& o, const T& val)
{
  o.getStream() << val;
  return o;
}
```

- Use hidden friends
  - Easy to do
  - Expertise needed to understand why
  - Name is injected into enclosing namespace
    - *But only for argument dependent lookup*
    - Not visible to qualified or unqualified lookup
    - Smaller overload set
      - Faster compilation
    - Avoids some accidental implicit conversions

U.S. DEPARTMENT OF ENERGY

Argonne NATIONAL LABORATORY

InfoStream

```
class InfoStream
{
public:
  InfoStream(std::ostream* output_stream)
      : prevStream(NULL), nullStream(new std::ostream(NULL)), ownStream(false)
  {
    currStream = output_stream;
  }

  InfoStream(InfoStream& in)
      : prevStream(NULL), nullStream(new std::ostream(NULL)), ownStream(false)
  {
    redirectToSameStream(in);
  }

  ~InfoStream();

  std::ostream& getStream(const std::string& tag = "") { return *currStream; }

  void flush() { currStream->flush(); }

private:
  // Keep track of whether we should delete the stream or not
  bool ownStream;

  std::ostream* currStream;

  // save stream during pause
  std::ostream* prevStream;

  // Created at construction. Used during pause
  std::ostream* nullStream;
};

template<class T>
inline InfoStream& operator<<(InfoStream& o, const T& val)
{
  o.getStream() << val;
  return o;
}
```

- Cannot insert manipulators into `InfoStream`

```
InfoStream out(&std::cout);
out << "Hello, world";
out << std::endl;    // fails to compile
```

  - `endl` is really a function *template*

```
template<class charT, class traits>
basic_ostream<charT, traits>&
endl(basic_ostream<charT, traits>& os);
```

    - Cannot infer T in

```
operator<<(InfoStream& o, const T&)
```

    - This is *subtle*

U.S. DEPARTMENT OF ENERGY

Argonne NATIONAL LABORATORY

InfoStream

```
class InfoStream
{
public:
  InfoStream(std::ostream* output_stream)
      : prevStream(NULL), nullStream(new std::ostream(NULL)), ownStream(false)
  {
    currStream = output_stream;
  }

  InfoStream(InfoStream& in)
      : prevStream(NULL), nullStream(new std::ostream(NULL)), ownStream(false)
  {
    redirectToSameStream(in);
  }

  ~InfoStream();

  std::ostream& getStream(const std::string& tag = "") { return *currStream; }

  void flush() { currStream->flush(); }

private:
  // Keep track of whether we should delete the stream or not
  bool ownStream;

  std::ostream* currStream;

  // save stream during pause
  std::ostream* prevStream;

  // Created at construction. Used during pause
  std::ostream* nullStream;
};

template<class T>
inline InfoStream& operator<<(InfoStream& o, const T& val)
{
  o.getStream() << val;
  return o;
}
```

U.S. DEPARTMENT OF ENERGY

Argonne NATIONAL LABORATORY

InfoStream

```cpp
class InfoStream
{
public:
  InfoStream(std::ostream* output_stream)
      : prevStream(NULL), nullStream(new std::ostream(NULL)), ownStream(false)
  {
    currStream = output_stream;
  }

  InfoStream(InfoStream& in)
      : prevStream(NULL), nullStream(new std::ostream(NULL)), ownStream(false)
  {
    redirectToSameStream(in);
  }

  ~InfoStream();

  std::ostream& getStream(const std::string& tag = "") { return *currStream; }

  void flush() { currStream->flush(); }

private:
  // Keep track of whether we should delete the stream or not
  bool ownStream;

  std::ostream* currStream;

  // save stream during pause
  std::ostream* prevStream;

  // Created at construction. Used during pause
  std::ostream* nullStream;
};

template<class T>
inline InfoStream& operator<<(InfoStream& o, const T& val)
{
  o.getStream() << val;
  return o;
}
```

```cpp
class InfoStream
{
public:
  explicit InfoStream(std::ostream* output_stream)
      : currStream(output_stream)
  {}

  explicit InfoStream(InfoStream& in)
  {
    redirectToSameStream(in);
  }

  std::ostream& getStream() const { return *currStream; }

  void flush() const { currStream->flush(); }

private:
  // Keep track of whether we should delete the stream or not
  std::unique_ptr<std::ofstream> ownStream;

  std::ostream* currStream = nullptr;

  // save stream during pause
  std::ostream* prevStream = nullptr;

  // Used during pause
  static std::ostream nullStream;

  template<typename T>
  friend InfoStream& operator<<(InfoStream& o, const T& val)
  { return o.getStream() << val; }
};
```

U.S. DEPARTMENT OF ENERGY

Argonne NATIONAL LABORATORY

InfoStream

```
class InfoStream
{
public:
  explicit InfoStream(std::ostream* output_stream)
      : currStream(output_stream)
  {}

  explicit InfoStream(InfoStream& in)
  {
    redirectToSameStream(in);
  }

  std::ostream& getStream() const { return *currStream; }

  void flush() const { currStream->flush(); }

private:
  // Keep track of whether we should delete the stream or not
  std::unique_ptr<std::ofstream> ownStream;

  std::ostream* currStream = nullptr;

  // save stream during pause
  std::ostream* prevStream = nullptr;

  // Used during pause
  static std::ostream nullStream;

  template<typename T>
  friend InfoStream& operator<<(InfoStream& o, const T& val)
  { return o.getStream() << val; }
};
```

U.S. DEPARTMENT OF ENERGY

Argonne NATIONAL LABORATORY

InfoStream

- Constructors are `explicit`
- Share a common `nullStream`
- Uses member initializers
- `const` member functions
- All variables initialized
- `ownStream` is stored in a `unique_ptr`
- No explicit destructor
- `InfoStream` inserter is hidden friend

```
class InfoStream
{
public:
  explicit InfoStream(std::ostream* output_stream)
      : currStream(output_stream)
  {}

  explicit InfoStream(InfoStream& in)
  {
    redirectToSameStream(in);
  }

  std::ostream& getStream() const { return *currStream; }

  void flush() const { currStream->flush(); }

private:
  // Keep track of whether we should delete the stream or not
  std::unique_ptr<std::ofstream> ownStream;

  std::ostream* currStream = nullptr;

  // save stream during pause
  std::ostream* prevStream = nullptr;

  // Used during pause
  static std::ostream nullStream;

  template<typename T>
  friend InfoStream& operator<<(InfoStream& o, const T& val)
  { return o.getStream() << val; }
};
```

InfoStream

- Are move semantics correct
  - Still no
    - `currStream` can point to ownStream
  - Fix
    - Explicitly delete copy / move operations and declare destructor (Rule of 5)

```cpp
class InfoStream
{
public:
  explicit InfoStream(std::ostream* output_stream)
      : currStream(output_stream)
  {}

  explicit InfoStream(InfoStream& in)
  {
    redirectToSameStream(in);
  }

  std::ostream& getStream() const { return *currStream; }

  void flush() const { currStream->flush(); }

private:
  // Keep track of whether we should delete the stream or not
  std::unique_ptr<std::ofstream> ownStream;

  std::ostream* currStream = nullptr;

  // save stream during pause
  std::ostream* prevStream = nullptr;

  // Used during pause
  static std::ostream nullStream;

  template<typename T>
  friend InfoStream& operator<<(InfoStream& o, const T& val)
  { return o.getStream() << val; }
};
```

InfoStream

- **Best Practice**
  - No maybe or maybe not ownership semantics
    - `InfoStream` should either never own a stream or always own a stream
  - Get copy/move semantics correct
    - Rule of 0
    - Much easier to do when first writing class
    - Explicitly `= delete` otherwise *(rarely)*

InfoStream

```
class InfoStream
{
public:
  InfoStream(std::ostream* output_stream)
      : prevStream(NULL), nullStream(new std::ostream(NULL)), ownStream(false)
  {
    currStream = output_stream;
  }

  InfoStream(InfoStream& in)
      : prevStream(NULL), nullStream(new std::ostream(NULL)), ownStream(false)
  {
    redirectToSameStream(in);
  }

  ~InfoStream();

  std::ostream& getStream(const std::string& tag = "") { return *currStream; }

  void flush() { currStream->flush(); }

private:
  // Keep track of whether we should delete the stream or not
  bool ownStream;

  std::ostream* currStream;

  // save stream during pause
  std::ostream* prevStream;

  // Created at construction. Used during pause
  std::ostream* nullStream;
};

template<class T>
inline InfoStream& operator<<(InfoStream& o, const T& val)
{
  o.getStream() << val;
  return o;
}
```

```
class InfoStream
{
public:
  explicit InfoStream(std::ostream* output_stream)
      : currStream(output_stream)
  {}

  explicit InfoStream(InfoStream& in)
  {
    redirectToSameStream(in);
  }

  std::ostream& getStream() const { return *currStream; }

  void flush() const { currStream->flush(); }

private:
  // Keep track of whether we should delete the stream or not
  std::unique_ptr<std::ofstream> ownStream;

  std::ostream* currStream = nullptr;

  // save stream during pause
  std::ostream* prevStream = nullptr;

  // Used during pause
  static std::ostream nullStream;

  template<typename T>
  friend InfoStream& operator<<(InfoStream& o, const T& val)
  { return o.getStream() << val; }
};
```

Refactoring

Refactoring

```
const std::vector<WaveFunction*> extract_wf_list(const std::vector<Mover*>& mover_list)
{
    std::vector<WaveFunction*> wf_list;
    wf_list.reserve(mover_list.size());
   for (auto it = mover_list.begin(); it != mover_list.end(); it++)
        wf_list.push_back(&(*it)->wavefunction);

    return wf_list;
}
```

- Non-controversial stuff
  - `reserve()` space if amount is known
  - `const` on return type has no effect

Refactoring

```
std::vector<WaveFunction*> extract_wf_list(const std::vector<Mover*>& mover_list)
{
    std::vector<WaveFunction*> wf_list;
    wf_list.reserve(mover_list.size());
    for (auto it = mover_list.begin(); it != mover_list.end(); it++)
    for (auto it = std::begin(mover_list); it != std::end(mover_list); it++)
        wf_list.push_back(&(*it)->wavefunction);
    return wf_list;
}
```

- `.begin()`
  - Less general
  - \+ Less characters

- `std::begin(c)`
  - \+ More general
    - Generic (template) code
  - More characters

Refactoring

```
std::vector<WaveFunction*> extract_wf_list(const std::vector<Mover*>& mover_list)
{
    std::vector<WaveFunction*> wf_list;
    wf_list.reserve(mover_list.size());
    for (auto it = mover_list.begin(); it != mover_list.end(); it++)
    for (std::vector<Mover*>::const_iterator it = mover_list.begin(); it != mover_list.end(); it++)
    wf_list.push_back(&(*it)->wavefunction);
    return wf_list;
}
```

- **auto**
  + Terse
  + Exact match
  - Writers over Readers
- `std::vector<Mover*>::const_iterator`
  - Verbose
  - Possible implicit conversion
  + Readers over Writers

Loops & Algorithms

(more refactoring)

Evolution of loops

```
const std::vector<WaveFunction*> extract_wf_list(const std::vector<Mover*>& mover_list)
{
    std::vector<WaveFunction*> wf_list;
    wf_list.reserve(mover_list.size());
   for (auto it = mover_list.begin(); it != mover_list.end(); it++)
        wf_list.push_back(&(*it)->wavefunction);

    return wf_list;
}
```

U.S. DEPARTMENT OF ENERGY

Argonne NATIONAL LABORATORY

Evolution of loops

```
for (auto it = mover_list.begin(); it != mover_list.end(); it++)
    wf_list.push_back(&(*it)->wavefunction);

for (auto& m : mover_list)
    wf_list.push_back(&m->wavefunction);

std::for_each(std::begin(mover_list), std::end(mover_list),
              [&](Mover& mover){ wf_list.push_back(&m->waveFunction); });

std::transform(std::begin(mover_list), std::end(mover_list), std::back_inserter(wf_list),
               [](Mover* m){ return &m->wavefunction; });
```

Evolution of loops

```
for (auto it = mover_list.begin(); it != mover_list.end(); it++)
    wf_list.push_back(&(*it)->wavefunction);

for (auto& m : mover_list)
    wf_list.push_back(&m->wavefunction);

std::for_each(std::begin(mover_list), std::end(mover_list),
              [&](Mover& mover){ wf_list.push_back(&m->waveFunction); });

std::transform(std::begin(mover_list), std::end(mover_list), std::back_inserter(wf_list),
               [](Mover* m){ return &m->wavefunction; });
```

- Most flexible
- Hardest to reason about
  - Have to reason about init, condition, expression & body
    - Modify `it`
    - Control flow (break / continue)

U.S. DEPARTMENT OF ENERGY

Argonne NATIONAL LABORATORY

Evolution of loops

```
for (auto it = mover_list.begin(); it != mover_list.end(); it++)
    wf_list.push_back(&(*it)->wavefunction);

for (auto& m : mover_list)
    wf_list.push_back(&m->wavefunction);

std::for_each(std::begin(mover_list), std::end(mover_list),
              [&](Mover& mover){ wf_list.push_back(&m->waveFunction); });

std::transform(std::begin(mover_list), std::end(mover_list), std::back_inserter(wf_list),
               [](Mover* m){ return &m->wavefunction; });
```

- Somewhat easier to reason about
  - No access to iterators
    - No `it` to modify
  - Only have to reason about the body
    - Control flow (break / ~~continue~~)

Evolution of loops

```
for (auto it = mover_list.begin(); it != mover_list.end(); it++)
    wf_list.push_back(&(*it)->wavefunction);

for (auto& m : mover_list)
    wf_list.push_back(&m->wavefunction);

std::for_each(std::begin(mover_list), std::end(mover_list),
              [&](Mover& mover){ wf_list.push_back(&m->waveFunction); });

std::transform(std::begin(mover_list), std::end(mover_list), std::back_inserter(wf_list),
               [](Mover* m){ return &m->wavefunction; });
```

- Algorithm
  - Even easier to reason about
  - Benefits of range-based for
  - No need to reason about control flow
    - well, except for exceptions
  - Every element is accessed
- More verbose

Evolution of loops

```
for (auto it = mover_list.begin(); it != mover_list.end(); it++)
    wf_list.push_back(&(*it)->wavefunction);

for (auto& m : mover_list)
    wf_list.push_back(&m->wavefunction);

std::for_each(std::begin(mover_list), std::end(mover_list),
              [&](Mover& mover){ wf_list.push_back(&m->waveFunction); });

std::transform(std::begin(mover_list), std::end(mover_list), std::back_inserter(wf_list),
               [](Mover* m){ return &m->wavefunction; });
```

- Algorithm
    - Self-documenting
    - Most declarative
- More verbose
    - Requires knowledge of conversion (`back_inserter`) and algorithms (`transform`) out there

Evolution of loops

```
for (auto it = mover_list.begin(); it != mover_list.end(); it++)
    wf_list.push_back(&(*it)->wavefunction);

for (auto& m : mover_list)
    wf_list.push_back(&m->wavefunction);

std::for_each(std::begin(mover_list), std::end(mover_list),
              [&](Mover& mover){ wf_list.push_back(&m->waveFunction); });

std::transform(std::begin(mover_list), std::end(mover_list), std::back_inserter(wf_list),
               [](Mover* m){ return &m->wavefunction; });
```

- Still on the fence about Best Practice
    - Prefer algorithms to hand-coded loops
        - If there isn't one, write one
    - Prefer range-based for to hand-coded loops
        - Less verbosity matters
            - Especially for small bodies

U.S. DEPARTMENT OF ENERGY

Argonne NATIONAL LABORATORY

```
template<class T, typename ~~T~~B~~OOL~~>
~~const~~ std::vector<T*>
filtered_list(const std::vector<T*>& input_list, const std::vector<~~T~~B~~OOL~~>& chosen)
{
    using filtered_type = std::vector<T*>;
    using size_type = typename filtered_type::size_type;

    std::vector<T*> final_list;
    final_list.reserve(input_list.size());
    for (~~int~~size_type iw = 0; iw ~~<~~!= input_list.size(); ++iw~~++~~)
        if (chosen[iw])
            final_list.push_back(input_list[iw]);
    final_list.shrink_to_fit();
    return final_list;
}
```

- Only use all-uppercase for macros
    - “Stop the CONSTANT SHOUTING” - *Jonathan Wakely*
- Don’t implicitly mix types
- Consider making this an algorithm
- != vs. <, ++iw vs. iw++
    - Tension between C++ way and OpenMP way

Templates for common code

(Refactoring)

```cpp
SPOSet* build_SPOSet(bool useRef,...)
{
  if (useRef)
  {
    auto* spo_main = new miniqmcreference::einspline_spo_ref<…>;
    spo_main->set(nx, ny, nz, num_splines, nblocks);
    spo_main->Lattice.set(lattice_b);
    return dynamic_cast<SPOSet*>(spo_main);
  }
  else
  {
    auto* spo_main = new einspline_spo<OHMMS_PRECISION>;
    spo_main->set(nx, ny, nz, num_splines, nblocks);
    spo_main->Lattice.set(lattice_b);
    return dynamic_cast<SPOSet*>(spo_main);
  }
}
```

```cpp
namespace
{
    // Helper for public build_SPOSet which builds it independent of the
    // derived SPOSetType
    template<typename SPOSetType>
    std::unique_ptr<SPOSet>
    build_SPOSet(…)
    {
        std::unique_ptr<SPOSetType> spo_main(new SPOSetType);
        spo_main->set(nx, ny, nz, num_splines, nblocks);
        spo_main->Lattice.set(lattice_b);
        return spo_main;
    }
} // namespace

std::unique_ptr<SPOSet> build_SPOSet(bool useRef,…)
{
  return useRef ?
      build_SPOSet<miniqmcreference::einspline_spo_ref<…>>(nx, ny, nz, num_splines, nblocks, lattice_b, init_random) :
      build_SPOSet<miniqmcreference::einspline_spo    <…>>(nx, ny, nz, num_splines, nblocks, lattice_b, init_random);
}
```

- Refactor common code into function template
  - Anonymous **`namespace`**
- Consider **`variant<einspline_spo<…>, einspline_spo_ref<…>>`**
  - C++17

```
namespace
{
    // Helper for public build_SPOSet which builds it independent of the
    // derived SPOSetType
    template<typename SPOSetType>
    std::unique_ptr<SPOSet>
    build_SPOSet(…)
    {
        std::unique_ptr<SPOSetType> spo_main(new SPOSetType);
        spo_main->set(nx, ny, nz, num_splines, nblocks);
        spo_main->Lattice.set(lattice_b);
        return spo_main;
    }
} // namespace

std::unique_ptr<SPOSet> build_SPOSet(bool useRef,…)
{
  return useRef ?
      build_SPOSet<miniqmcreference::einspline_spo_ref<…>>(nx, ny, nz, num_splines, nblocks, lattice_b, init_random) :
      build_SPOSet<miniqmcreference::einspline_spo    <…>>(nx, ny, nz, num_splines, nblocks, lattice_b, init_random);
}
```

Placement `new`

new

- `new T(…)`
  - Needed for lifetimes different than a scope
  - Performs two operations
    - Acquire space from heap
    - Construct an object into that space

U.S. DEPARTMENT OF ENERGY

Argonne NATIONAL LABORATORY

Placement new

- `new (address) T(…)`
  - Needed for lifetimes different than a scope
  - Performs one operation
    - ~~Acquire space from heap~~
    - Construct an object into that space

U.S. DEPARTMENT OF ENERGY

Argonne NATIONAL LABORATORY

Placement new

- ***Fraught with peril!***
- User responsibility
  - Space is uninitialized
  - `~T()` is eventually called exactly once
- Otherwise, undefined behavior
  - Rules about uninitialized space, lifetime of objects, pointers & references, type punning, etc. are expert-level complicated

U.S. DEPARTMENT OF ENERGY

Argonne NATIONAL LABORATORY

Kokkos View of Views

```
void resize()
{
  //    psi.resize(nBlocks);
  //    grad.resize(nBlocks);
  //    hess.resize(nBlocks);

  psi  = Kokkos::View<vContainer_type*>("Psi", nBlocks);
  grad = Kokkos::View<gContainer_type*>("Grad", nBlocks);
  hess = Kokkos::View<hContainer_type*>("Hess", nBlocks);

  for (int i = 0; i < nBlocks; ++i)
  {
    //psi[i].resize(nSplinesPerBlock);
    //grad[i].resize(nSplinesPerBlock);
    //hess[i].resize(nSplinesPerBlock);

    //Using the "view-of-views" placement-new construct.
    new (&psi(i)) vContainer_type("psi_i", nSplinesPerBlock);
    new (&grad(i)) gContainer_type("grad_i", nSplinesPerBlock);
    new (&hess(i)) hContainer_type("hess_i", nSplinesPerBlock);
  }
}
```

- Placement new should not be over *existing* objects
  - Existing object destructor never called
    - In a loop
- Recommendation
  - Avoid placement new

Best Practices

Best Practices

- C++ Core Guidelines
  - http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines
- Collaborative Collection of C++ Best Practices - Jason Turner
  - https://github.com/lefticus/cppbestpractices
- C++ Coding Standards
  - Andrei Alexandrescu & Herb Sutter
  - Preface: *Think.*
  - 0. Don't sweat the small stuff

Q & A