Achieving ultimate performance in financial data processing through compile time introspection: CME MDP3 Example

Ultimate performance

- Unattainable
- Assumes algorithms and data structures are already the best.
- In this presentation:
  - How to help the compiler know more so it micro-optimizes more
  - The same techniques give reliability
  - Are extensible and
  - May save effort

Financial data processing

- This is a practical application of techniques
- Full real life complexity of CME MDP3
- Performance is important
- The general ideas must be able to be understood by common programmers, not just hardcore metaprogrammers

Compile time introspection (reflection)

- More frequently referred to as “reflection”, however, this presentation is not about program-modifying reflection, only self-inspection, hence introspection.

- The code will “reason” about itself
- Metaprogramming techniques

Presenter

Crabel Capital Management:

- Large trader of futures,
- “eager” consumer of CME MDP3 for automated trading
- Performance is important
- C++ environment over Linux for automated trading:
  - Conservative about toolchain versions
  - We care about not requiring C++ 14
  - But open to modernization
  - Made this presentation possible

Eduardo Madrid:

- Interested in participating in the community
- Senior developer
- Educator
- Blogger at

http://thecppzoo.blogspot.com/

- Fluent in precursors such as Prolog, Functional Programming languages

Why intermediate to advanced?
Required familiarity:

- Recursion
- Templates,
- variadic templates,
- specializations and
- overload resolution rules.
- It helps to also understand
  - SFINAE, metaprogramming, type injection, unevaluated contexts, void_t, …

Why intermediate to advanced? What if I don't know any of

- SFINAE
- Metaprogramming
- Type injection
- Unevaluated contexts
- void_t

?

Fear not.

What these concepts do will be explained when used, but not the details of how they do it. You may miss some nuance about the solutions to practical challenges I arrived at.

- Metaprogramming has complications.

How can introspection help?

Not a theoretical discussion, but concrete examples illustrated:

- Serialization to string
- Example case specific to application domain:
  - Null values
  - Traversing “repeating groups”
  - Subscription/dispatching

Serialization

Frequently needed feature:

Converting your data into a representation suitable to be persisted.

Frequently, it is exactly the same pseudo code:

- For all members **m** in structure instance **A**:
  - Serialize **m**

Serialization in C++

Iterating over the members of a class is using the class definition as program data, **introspection**

But C++ does not have any primitives to iterate over the members of structures or classes!

Two classes of solutions:

1. Code Generators
2. Preprocessor Macros

This presentation is about providing introspection building blocks that add as little complexity as possible to code generators

Very simple example

```
struct Aggregate {
    int member1;
    std::string member2;
};
```

```
std::string to_string(const Aggregate &a) {
    return
        "(Aggregate member1:" +
        std::to_string(a.member1) +
        " member2:" +
        a.member2 + ')';
}
```

Very simple example

The intended output would be something like this:

(Aggregate member1:1 member2:Bellevue)

Properties:

- The implementation of “to_string” is hand-coded, can be made maximally efficient
- It is brittle, the code may even build, but not do what is intended of it:
  - If another member is added
  - If the names of the members change
  - Changes in the order of members

Very simple example

Assume there would be run-time introspection as in languages such as Java:

- The code will have a run time penalty over hand coded: traversal of the members
- But the code is now resilient
- Obviously, the compiler does not gain extra information

Very simple example

Missed opportunity: If class definitions don't change during program execution, inspection of members should be hoisted to compilation time.

- If this is achieved:
  - Potentially code as performing as hand-coded
  - Resilient

Implementing specifications, protocols

Protocols and specifications tend to describe their data in language-agnostic ways. It makes sense to **automatically** convert specifications to C++ models, to use *code generators*.

- Minimize the effort to make the code generator (it is not a money maker)
- Still we want to take full advantage of its existence
- Not the preferred place to implement optimizations:
  - One form of premature optimization: done automatically
  - Potentially a pessimization
  - Separation of concerns: The code generator *models* an specification, the compiler optimizes.

C++ introspection building blocks

```
template<typename> struct Members {};
///< Specializations have "type" member

/// Relates a member pointer to an index
template<typename T, T mPtr, unsigned Index>
struct MemberBinding {
    constexpr static T value = mPtr;
    constexpr static unsigned index = Index;
};

template<typename...> struct Pack {};
```

Introspection Building Blocks

- “Members” is an empty template (!?)
- “Pack” is an empty variadic template (!!??)
- “MemberBinding” only declares internally its own template parameters (!?)
  - strange pattern template<typename T, T Val>

Metaprogramming parenthesis

- C++ Metaprogramming is weird. Really.
  - *Declarative* in nature and Functional Programming Paradigm in style:
    - Learn concepts such as *unification* where they are natural as in Prolog, called “substitution” in C++
    - Learn something like Haskell for the Functional Programming Paradigm (immutability, pattern matching are prominent), pattern matching in C++ through overload resolution rules is beyond confusing
      - Dealing with variadic templates is like programming in LISP
  - Weird does not mean hard, but perhaps fun?
  - Stephen Dewhurst: It was *discovered* C++ supports metaprogramming, or it was invented for C++, C++ was not *designed* to support it.
- Public acknowledgement to Stephen Dewhurst and Walter Brown.
- How do you know you “nailed it”? the code has Walter Brown’s *“void_t quality”*: succinct, simple, yet profoundly subtle. The existence of such things attests to C++'s richness.

Metaprogramming parenthesis

- This subject matter is at the rough edges of the language.
  - Perhaps you can be an accidental pioneer
  - It gives you a different appreciation
  - The techniques are independently useful
  - You learn much more than just C++
  - It may be a worthy fight:
    - Plug: “The inherent complexity of Generic Programming is to capture the innermost nature of what is being programmed”, the accidental complexity forced me to apply Functional and Logic programming, Artificial Intelligence, and lots of mathematics.

Introspection Building Blocks

- These are *structural* templates, their only purpose is to convey structure:
  - To be specialized, a template must be declared (“Members”)
  - “Pack” just puts together a bunch of types
  - “MemberBinding” does what its name indicates
- These are used declaring things, momentarily we will see how these declarations are used in practice

Simple C++ introspection

- Example of code generator output:

```
#include <string>

struct Aggregate {
    int member1;
    std::string member2;
};

template<> struct Members<Aggregate> {
    using type = Pack<
        MemberBinding<decltype(&Aggregate::member1), &Aggregate::member1, 0>,
        MemberBinding<decltype(&Aggregate::member2), &Aggregate::member2, 1>
    >;
    constexpr static const char *name = "Aggregate";
    static const char *const *names() {
        static const char *rv[] = { "member1", "member2" };
        return rv;
    }
};
```

Simple C++ introspection

What has just happened?

The structure “Aggregate" has been *decorated* by creating an *specialization* of the template “Members" on it.

- The decoration is *non-intrusive*, occurs *outside*.

The specialization defines:

- a member type “type" which is a collection, “Pack”, of *bindings* between member pointers and “Aggregate”

- Also binds the names, *identifiers* to the members

Simple C++ introspection

- Why is “names" a class function instead of class data member? Otherwise it would require definition in a .cpp. Inlined, no performance penalty, even after C++ 11 (not thread sensitive)

```
#include <string>

struct Aggregate {
    int member1;
    std::string member2;
};

template<> struct Members<Aggregate> {
    using type = Pack<
        MemberBinding<decltype(&Aggregate::member1), &Aggregate::member1, 0>,
        MemberBinding<decltype(&Aggregate::member2), &Aggregate::member2, 1>
    >;
    constexpr static const char *name = "Aggregate";
    static const char *const *names() {
        static const char *rv[] = { "member1", "member2" };
        return rv;
    }
};
```

Simple C++ introspection

- Notice “names” is a run-time tool:
  - What would be the benefit of hoisting to compilation time introspection of the names of the members?
  - At compilation time the structural properties and small integers are the drivers

```
#include <string>

struct Aggregate {
    int member1;
    std::string member2;
};

template<> struct Members<Aggregate> {
    using type = Pack<
        MemberBinding<decltype(&Aggregate::member1), &Aggregate::member1, 0>,
        MemberBinding<decltype(&Aggregate::member2), &Aggregate::member2, 1>
    >;
    constexpr static const char *name = "Aggregate";
    static const char *const *names() {
        static const char *rv[] = { "member1", "member2" };
        return rv;
    }
};
```

Simple C++ introspection

- To specify at compilation time a member, the practical way is using it as member address, but its type needs specification, perhaps just decltyped: One member becomes two template parameters

```
#include <string>

struct Aggregate {
    int member1;
    std::string member2;
};

template<> struct Members<Aggregate> {
    using type = Pack<
        MemberBinding<decltype(&Aggregate::member1), &Aggregate::member1, 0>,
        MemberBinding<decltype(&Aggregate::member2), &Aggregate::member2, 1>
    >;
    constexpr static const char *name = "Aggregate";
    static const char *const *names() {
        static const char *rv[] = { "member1", "member2" };
        return rv;
    }
};
```

```cpp
#include <sstream>

/// General case
template<
    typename A,
    typename MT,
    MT MPTR,
    unsigned Ndx,
    typename... Rest
>
void printPack(
    std::ostream &out,
    const A &v,
    Pack<MemberBinding<MT, MPTR, Ndx>, Rest...> *
) {
    using M = Members<A>;
    out << ' ' << Ndx << ':' << M::names()[Ndx] << ':' << v.*MPTR;
    printPack(out, v, (Pack<Pack...> *)nullptr);
}
/// Recursion end case
template<typename A>
void printPack(std::ostream &out, const A &v, Pack<> *) {}

template<typename T> std::string to_string(const T &v) {
    std::ostringstream ostr;
    using M = Members<T>;
    ostr << '(' << M::name;
    printPack(ostr, v, (typename M::type *)nullptr);
    ostr << ')';
    return ostr.str();
}
```

Introspection-using code

“printPack” is a template function:

- Takes three arguments: Output stream, the aggregate value and a pointer without even a name, not ever used in the definition block
- Enough template parameters to indicate:
    - The aggregate type
    - A “Pack” of member bindings
        - The first member binding in the pack is fully specified
- Callers to “printPack” do not specify template parameters, they pass in a third argument, a pointer, whose type allows the *template substitution* mechanism to fill in the template arguments. This technique is called “type injection”
    - Type injection seems to be the most practical way to deal with template parameters to functions because they enable automatic type deduction
- The implementation is recursive
- The recursion ends calling a more specific (not to confuse with specialized) overload of “printPack”, the case of an empty “Pack”
    - This recursion is a compilation-time device only if the compiler optimizes
    - Neither the extra type injection pointer is “there” in the object code

Introspection-using code

Complication: What about members that are introspection-decorated types themselves?

This requires a way to recognize those types decorated with a “Members” specialization when calling a function:

- Must be an overload
- There still must be a fall-through case for values not introspectible
- Requires intricate knowledge of overload resolution rules or lots of trial and error to get right
  - Compiler versions will bite you.
- Recommendation: to clear the forest of complications of metaprogramming, use Walter Brown’s chain saw of void_t.

void_t

```
#if not old GCC
template<typename...> using void_t = void;
#else
template<typename... Pack>
struct void_t_impl { using type = void; };
template<typename... Pack>
using void_t = typename void_t_impl<Pack...>::type;
#endif
```

- Will be used in function arguments to detect whether its template parameters are valid types
    - If valid, void_t becomes an alias for void and the overload where it appears remains viable.
    - Otherwise, the overload gets deleted from the viable set.
        - This is called *“substitution failure”*, which is not a compilation error, *SFINAE*.
- Use a void_t<…> pointer in the function declaration, not defaulted
- Inject a null void pointer at the call site, default arguments are fine but bifurcate the arity, a chance for more complications

Realistic C++ introspection

- Remember that overloads with ellipsis are the overloads of last resort

```
/// Overload for T where there is a
/// Members<T> specialization
template<typename T>
void to_str_impl(
    std::ostream &out, const T &v,
    void_t<typename Members<T>::type> *
);
/// Less specific overload
template<typename T>
void to_str_impl(
    std::ostream &out, const T &v,
    ...
) { out << v; }
```

Realistic C++ introspection

- In the to_str_impl function
  - the type T is automatically deduced in each overload
    - The difference is in the type of the last function argument:
      - If void_t<typename Members<T>::type> is valid, a void *
      - Else, an ellipsis
      - Call sites inject a void *, then both overloads may match, but if the void * is valid, it wins overload resolution.

Complete example

In https://github.com/thecppzoo/cppcon2016/

Possible improvements

- Instance, class functions
- Take advantage of inheritance
- Compile-time selection of subsets of member fields
- Semantic-aware serialization, for example, not serializing optional members in instances where they are not present
- Forwarding of configuration options to the serialization primitives.
- Alas, this is not a presentation on serialization

Null values in MDP3

- An application specific example,
- But it has generality.
- There are some fields that must be present in the encoding but are not really there. Three + 1 types:
    1. Normal types: null is denoted by a magic value of that type
    2. Enumerations: The value is the “null_value” enumerator
    3. Unions: The union has a member “NullValue” and its value is true.
    4. Aggregates are null if all of their members are null
- Will allow me to show what the compiler does with introspection
- The code, warts and all, is in production

Nulls #1-Optionals

- For the first case, optional data members where characterized with this template:

```
template<typename T, T NullValue> struct optional {
    T value;
    constexpr operator T() const noexcept { return value; }
    constexpr bool valid() const noexcept { return NullValue != value; }
};

using Int32NULL = optional<int32_t, 2147483647>;
```

Nulls #1-Optionals

- The optional requires
  - base type
  - The magic value of that type.

- Again, just an structural pattern

```
template<typename T, T NullValue> struct optional {
    T value;
    constexpr operator T() const noexcept { return value; }
    constexpr bool valid() const noexcept
    { return NullValue != value; }
};

using Int32NULL = optional<int32_t, 2147483647>;
```

SFINAE on bool (equivalent to std::enable_if)

```
/// Implementation detail of a meta tool to convert \c true
into a SFINAE type
template<bool> struct TrueImpl {};
/// Specialization: if argument is true, member type exists
template<> struct TrueImpl<true> { using type = void; };
/// Meta tool to convert true into a SFINAE type
template<bool b> using True = typename TrueImpl<b>::type;
```

```
/// User-facing API to drive detection of null values
///
/// Drives the implementation overloads through a
/// type-token parameter
/// \note user code is allowed to further overload this
template<typename T>
inline constexpr bool is_null(const T &v);

/// Captures the pattern of \c T being an optional
template<typename T, T NullValue> inline constexpr bool
is_null(optional<T, NullValue> v)
    { return NullValue == v; }
```

```
/// Implementation of is_null
///
/// \tparam T unrestricted type, used when all the other overloads fail
/// \param v any value
/// \param ... variable arguments to make this the overload of last resort
/// \return false
/// \note User code is allowed to overload is_null_impl
    template<typename T> constexpr inline bool
is_null_impl(T v, ...) { return false; }

/// Captures the pattern of \c T being an enumeration and have null_value
    template<typename T> constexpr inline bool
is_null_impl(T v, void_t<True<std::is_enum<T>::value>, decltype(T::null_value)> *)
    { return T::null_value == v; }

/// Captures the pattern of \c T being an union with a field NullValue
    template<typename T> constexpr inline bool
is_null_impl(T v, void_t<True<std::is_union<T>::value>, decltype(T::NullValue)> *)
    { return v.NullValue; }

/// Captures the pattern of there being an specialization of \c Members for \c T
    template<typename T> constexpr inline bool
is_null_impl(const T &v, void_t<typename Members<T>::type> *) {
    return packIsNull(v, (typename Members<T>::type *)nullptr);
}
```

Nulls

- The way to traverse the members is as in serialization.

```cpp
template<typename A> inline constexpr bool
packIsNull(const A &, Pack<> *) { return true; }

/// Captures a non-empty \c Pack pattern
template<typename A, typename T, T MemberPtr, typename... Tail>
inline constexpr bool
packIsNull(const A &agg, Pack<Member<T, MemberPtr>, Tail...> *);

/// Captures the pattern of there being an specialization of Members for T
template<typename T> constexpr inline bool
is_null_impl(const T &v, void_t<typename Members<T>::type> *) {
    return packIsNull(v, (typename Members<T>::type *)nullptr);
}
```

```cpp
template<typename A, typename T, T MemberPtr, typename... Tail>
inline constexpr bool
packIsNull(const A &agg, Pack<Member<T, MemberPtr>, Tail...> *) {
    return
        is_null(agg.*MemberPtr) && packIsNull(agg, (Pack<Tail...> *)nullptr);
}

}

template<typename T> inline constexpr bool is_null(const T &v)
{ return detail::is_null_impl(v, (void *)nullptr); }
```

Performance experiments

Example of “MaturityMonthYear”:

```
#define EXACT_LAYOUT
__attribute__((aligned(1), packed))

/// Year, Month and Date
/// \note semantic type MonthYear
struct MaturityMonthYear {
    optional<uint16_t, 65535> year; ///< YYYY
    optional<uint8_t, 255> month; ///< MM
    optional<uint8_t, 255> day; ///< DD
    optional<uint8_t, 255> week; ///< WW
} EXACT_LAYOUT;
```

What AMD64 assembler do you expect for this?:

```
bool fun(const MaturityMonthYear &m)
{
    return mdp3::meta::is_null(m);
}
```

GCC 6.2 -O2:

```
fun(mdp3::MaturityMonthYear const&):
        xorl    %eax, %eax
        cmpw    $-1, (%rdi)
        je      .L7
.L1:
        rep ret
.L7:
        cmpb    $-1, 2(%rdi)
        jne     .L1
        cmpb    $-1, 3(%rdi)
        jne     .L1
        cmpb    $-1, 4(%rdi)
        sete    %al
        ret
```

Performance experiments

All of the techniques of:

- recursion on the list of members
- the extra argument in type injection
- the trampolines to recognize patterns

Disappeared from the object code!

Note the compiler does not merge the comparisons

Performance experiments

What about

```
bool fun(mdp3::MaturityMonthYear &m)
{
    m.week.value = 5;
    return mdp3::meta::is_null(m);
}
```

We get this:

```
fun(mdp3::MaturityMonthYear&):
        movb    $5, 4(%rdi)
        xorl    %eax, %eax
        ret
```

Performance experiments

- The compiler knows the member “week” must be different to 255, because 255 is a compilation time constant provided by the code generator.
- Introspection moved the effort to traverse members from the code generator to a no-performance penalty library in C++
- A requirement such as “is_null” can be programmed much later than the code generator.
- Neither the code generator nor the generated code need modification for new features
- The user can override the library with more specific overloads, this is not possible in general with the output of a code generator
- And the compiler has more visibility into what the code is doing

Nested Data

- In MDP3 most message types have arrays of nested data, called “Repeating Groups”.
- My characterizations of messages was to make them specializations of this template:

```
template<unsigned id> struct message_t;
```

Nested Data

- Messages may have more than one repeating group (sub arrays)
- The beginning of a repeating group is the end of the preceding group
- Modeled as ranges, called “flyweights” (they are also flyweights) due to cultural reasons

```cpp
template<typename...> using void_t = void;

template<unsigned Id> struct Message;

template<unsigned Id, typename = void>
struct Pattern{ constexpr static int value = 1; };

template<unsigned Id>
struct Pattern<Id, void_t<typename Message<Id>::type>>
{ constexpr static int value = 2; };

template<> struct Message<88> { using type = int; };
template<> struct Message<109> {};

int values[] = {
    Pattern<0>::value,
    Pattern<88>::value,
    Pattern<109>::value,
    Pattern<200>::value
};
```

void_t in class templates

- It works because the partial specialization where void_t appears has one parameter less
- The default parameter must be void, whatever void_t aliases
- Whenever testing a pattern of “having a decoration”, use a member of the decoration in the test.
- Default template parameter not necessary, but not harmful either, unlike as function argument

Nested Data

See code in github.

The entry point is “to_stream_weak_message”

which ends calling printing

the repeating groups of the message

Dispatcher

```
template<unsigned Id> struct Message;

struct Dispatcher {
    using function_t = void (*)(const void *, void *);
    function_t table[200];
    template<unsigned Id> void subscribe(
        void (*function)(const Message<Id> *, void *)
    ) {
        table[Id] = reinterpret_cast<function_t>(function);
    }
};

struct Something {};
void consume88(const Message<88> *mptr, void *context) {
    auto &m = *mptr;
    auto &thy = *reinterpret_cast<Something *>(context);
}
bool fun(Dispatcher &d) {
    d.subscribe(consume88);
    return d.table[88] == (Dispatcher::function_t)consume88;
}
```

Dispatcher

- Preserves type safety:
- No performance penalty, latent opportunities for further optimization
- No complications
- User code

Acknowledgements

- To my friends, coworkers and victims of my classes at “Crabel”, who suffered through my attempts at refining the presentation of these topics providing valuable feedback

- To friends and coworkers with whom we devised the techniques that made this presentation possible, including Brett Polivka at Magnetar Capital, Julio García, Rohan Berdarkar and Joseph Elble at “Crabel”

Plugs

- In this presentation we covered deficiencies in C++ through code generators, in my blog I have covered, albeit incompletely, the option of preprocessor macros, especially boost seq.

- Metaprogramming has both inherent and accidental complexity, I cover the inherent complexity part in my favorite article thus far, “innermost nature”

- I think my original idiom of the “Power Set Covariant Return Type” may be of interest to the community