THE PLAN FOR TOMORROW

Compile-Time Extension Points for C++ Libraries and Applications

JeanHeyd Meneide
April 25th, 2019
New York C++ Meetup, MongoDB, New York, NY

EXTENSION GOALS: ADDING FUNCTIONALITY…

- beyond what was initially considered by application / library authors
  - Callback functions with void* userdata in C libraries

- to perform some semantically-expected task for types outside author's purview
  - std::swap

- to endow a class with a specific, compatible interface
  - virtual protected functions in iostreams

WELL-KNOWN EXTENSION METHODS

A brief overview of compile-time and runtime hybrid extension technology

WELL-KNOWN: VIRTUAL METHODS

- No surprises here: create base class and stuff it with virtual methods

```cpp
struct animal {
    virtual std::string sound () const = 0;
};

struct dog : public animal {
    virtual std::string sound () const override {
        return "woof";
    }
};
```

WELL-KNOWN: VIRTUAL METHODS USAGE

- Used extensively up to ~2008, less so now in place to static polymorphism
  - Many game engines: Ogre, Irrlicht, Doom, etc…
  - Qt: QObject and the entire class tree
  - Clang: ASTMatchers and extension points
  - C++ standard library: iostream customization points
  - One too many C++ university classes

STANDARD LIBRARY CASE STUDY: IOSTREAMS

- C++ streambufs/streams – 3 kinds different hooks
  - fixed interface, calls into…
  - protected virtual inherited members, which utilize…
  - traits type (char_traits), facets, and more inside function calls

- How it works:
  - Define your own stream type
  - derive from standard base(s), using virtual bases where needed
  - overload protected virtual functions (overflow, xsputn, underflow, unflow, …)

STREAMS: GOOD IDEA, POOR OVERALL DESIGN

- Saves on some degree of template boilerplate
  - Means you can compile and fix an `std::ostream&` API boundary
  - But you cannot accept "wide" (ugh) streams

- Issue of quality of specification versus implementation
  - Standard library implementers always complain about the specification
  - Too many things a user can customize
  - Extreme mixing of layers: 3 kinds!

BENEFITS: VIRTUAL METHODS

- Can work with super class (base class) at compile-time
  - calls the right method at runtime
  - no need to bookkeep function pointers and similar

- Heavily optimized by compiler writers to de-virtualize simple cases
  - E.g.: current-gen non-user-specialized iostreams, C++ XAML, and more

DRAWBACKS: VIRTUAL METHODS

- ABI-brittle
  - adding a function to class might append to virtual table, but may insert in middle of derived class's virtual table
  - difficult to detect mismatches

- Runtime efficiency
  - Does "X" *need* to be virtual? Must decision be delayed to runtime?

- Implementation-controlled Virtual Tables / Slicing Problem
  - Base classes must be handled as pointers / references or risk slicing

CALLBACKS WITH USERDATA

- Function which takes a strongly-typed function pointer and a void* userdata
  - Staple of C APIs everywhere, including some C standard library functions
  - Highly flexible

- Used to let (application) developer do things beyond what was envisioned
  - e.g., serialize data into to a std::vector instead of a FILE*

```
typedef int (*lua_Writer)(lua_State*, const void*, size_t, void*);

int lua_dump(lua_State* L, lua_Writer writer, void* userdata, int strip_symbols);
```

EASY TO WRAP IN C++

- Typical C call, wrapped in C++

```cpp
template <typename Callback>
int dump_handler(lua_State* L, const void* data, size_t data_size, void* userdata) {
    Callback& callback = *static_cast<Callback*>(userdata);
    return callback(L, data, data_size);
}

template <typename Callback>
void dump_with(lua_State* L, Callback&& callback, bool strip_symbols = true) {
    lua_Writer writer = &dump_handler<std::remove_cvref_t<Callback>>;
    void* userdata = static_cast<void*>(std::addressof(callback));
    lua_dump(L, writer, userdata, static_cast<int>(strip_symbols));
}
```

EASY TO WRAP IN C++: NO TEMPLATES

- Fix the interface to save on template duplication for every callable…

```
using dump_function = std::function<lua_State*, const void*, data_size_t>;

int dump_handler(lua_State* L, const void* data, size_t data_size, void* userdata) {
    dump_function& callback = *static_cast<dump_function*>(userdata);
    return callback(L, data, data_size);
}


void dump_with(lua_State* L, dump_function callback, bool strip_symbols = true) {
    lua_Writer writer = &dump_handler<std::remove_cvref_t<Callback>>;
    void* userdata = static_cast<void*>(std::addressof(callback));
    lua_dump(L, writer, userdata, strip_symbols);
}
```

EASY TO WRAP IN C++: NO TEMPLATES

- std::function is expensive
  - Higher efficiency, low cost for (maybe) C++20: std::function_ref

```
using dump_function = std::function_ref<lua_State*, const void*, data_size_t>;

int dump_handler(lua_State* L, const void* data, size_t data_size, void* userdata) {
    dump_function& callback = *static_cast<dump_function*>(userdata);
    return callback(L, data, data_size);
}


void dump_with(lua_State* L, dump_function callback, bool strip_symbols = true) {
    lua_Writer writer = &dump_handler<std::remove_cvref_t<Callback>>;
    void* userdata = static_cast<void*>(std::addressof(callback));
    lua_dump(L, writer, userdata, strip_symbols);
}
```

EASY…?

- Inline and synchronous execution:
  - No need for storage
  - No need to manage lifetime

- Non-inline execution:
  - Callling it later (events in Qt, libev, etc.)? Need storage.
  - Multithreading? Need storage.
  - Storage means lifetime…

WELL-KNOWN: CALLBACKS WITH USERDATA USAGE

- Literally every C library, ever...
  - Lua, libclang,
  - libpng, libjpeg
  - jansson, libev, freetype
  - Win32: everywhere
  - C Standard Library: qsort

BENEFITS: CALLBACK WITH USERDATA

- Space and time efficient
  - especially if callback never needs to be stored
  - function pointers are cheap

- ABI-hardy
  - difficult to break ABI unless the actual callback interface changes
  - user can place extra data into void* for their needs at no cost to library

DRAWBACKS: CALLBACK WITH USERDATA

- Exception/early exit issues
  - if stored, is the callback called when an exception is tossed?

- In-lining optimizations for compiled code becomes restricted
  - `qsort(…)` vs. `std::sort(…)`
  - link time optimizations helps here

- Lifetime issues, when
  - storing the callback to call "at a later date"
  - multithreading concerns

COMPILE-TIME EXTENSION

Picking and choosing work to execute using compile time choices

EXTENSION METHODOLOGIES

- Compiler-Assisted
  - (Partial) Class Template Specializations
  - "Koenig"/Argument-Dependent Lookup (ADL)
    - Static friend functions
  - Template functions + Overloading

- Author Mandated
  - Traits/Policy/Agent templates

CLASS TEMPLATES + SPECIALIZATION

- Uses a class template

- User then (partially) specializes a class for this template

- Case studies: sol2, std::hash
  - Base template, user specialized templates
  - Using this class:

```
struct two_things {
    int a;
    bool b;
};
```

SOL2 AND STD::HASH

case studies in class template specializations

SOL2: SOL::STACK::GETTER<T, C>

```cpp
// sol.hpp

namespace sol { namespaces stack {

        template <typename T, typename C = void>
        struct getter {
                static T get(lua_State*, int, record&) {
                        /* default implementation here */
                }
        };

}} // namespace sol::stack
```

SOL2: FULL CLASS TEMPLATE SPECIALIZATION
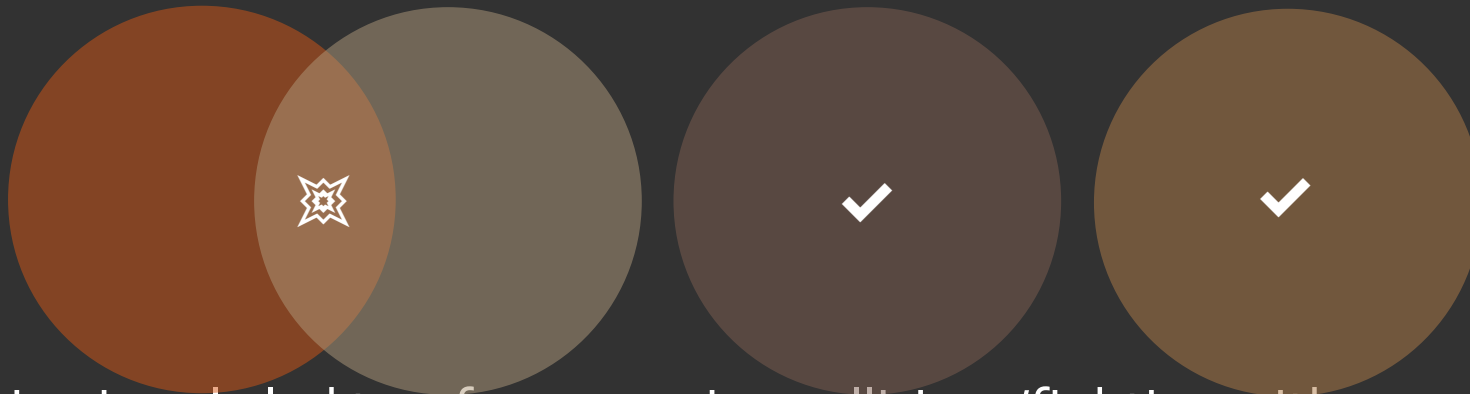
```cpp
// user.cpp

namespace sol { namespace stack {

    template <>
    struct getter<two_things> {
        static two_things get(lua_State*, int, record&) {
            /* user code here */
        }
    };

}} // namespace sol::stack
```

DRAWBACK I: SPECIALIZATION COLLISIONS

- Fail to separate base implementation versus user implementation
  - All user customization points must occupy the same finite code space
  - All explicit and specializations must not collide (mutual exclusion principle)

- Bad design in sol2 led to a few annoying collisions/fighting with user specializations

DRAWBACK I: SFINAE POWER

- Extra template argument is needed on every extension point for SFINAE traits to be applied

```
template <typename T>
struct custom_point<T, /* SFINAE here */> {
    /* … */
};
```

- SFINAE is messy
  - decltype() and is_detected SFINAE slightly less ugly than std::enable_if_t
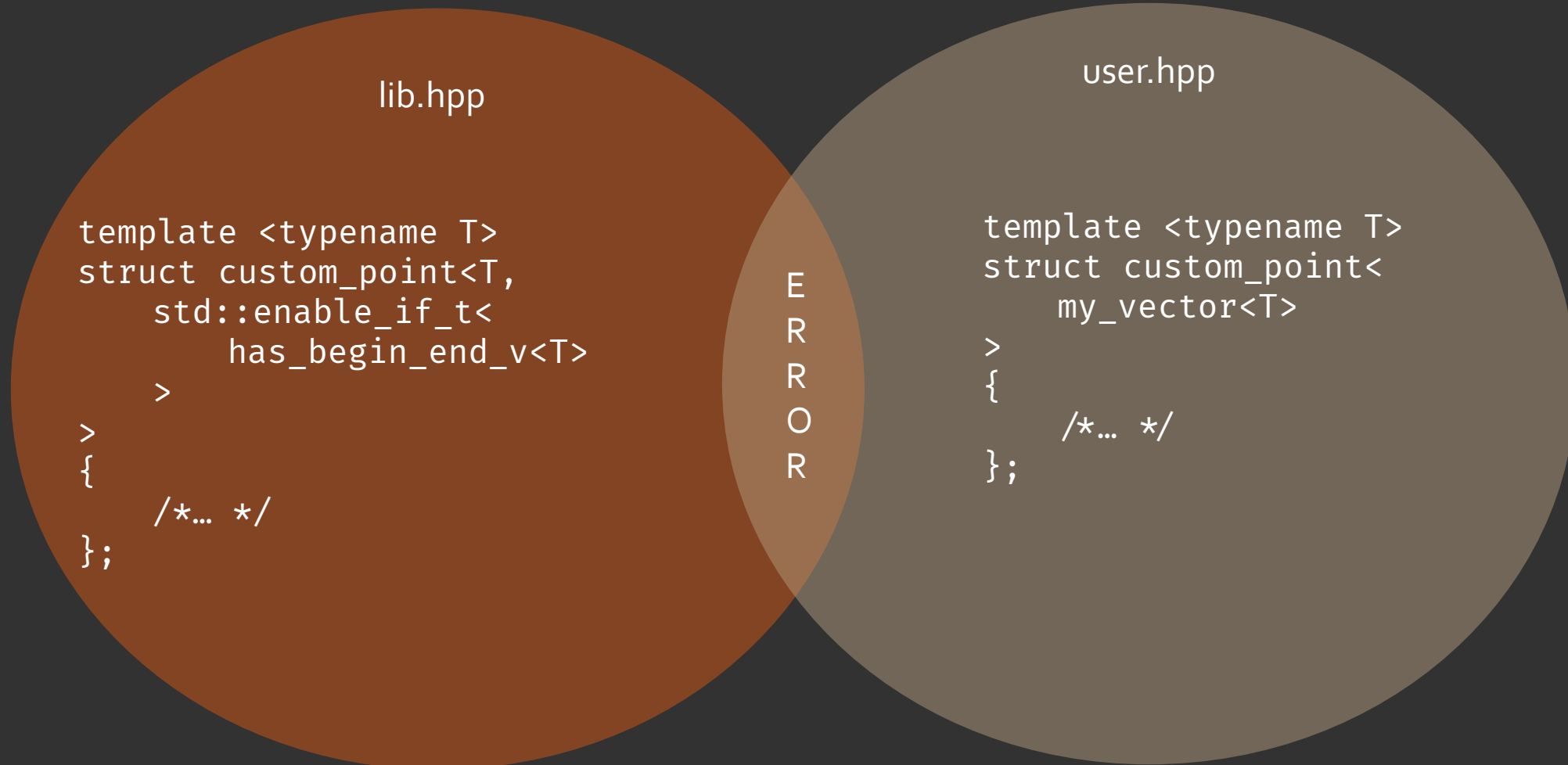 - introduces mutual exclusion principle problems

DRAWBACK II: SPECIALIZATION COLLISIONS

```cpp
namespace sol { namespace stack {

    template <typename T>
    struct getter<T, std::enable_if_t<std::is_integral_v<T>>> {
        static T get(lua_State*, int, record&) {
            /* implementation for integers */
        }
    };

} // namespace sol::stack
```

DRAWBACK II: SPECIALIZATION COLLISIONS

```cpp
namespace sol { namespace stack {

    template <>
    struct getter<std::uint64_t> {
        // ^ CLANG ERROR: ambiguous specialization
        static std::uint64_t get(lua_State*, int, record&) {
            /* implementation for integers */
        }
    };

} // namespace sol::stack
```

DRAWBACK II: SPECIALIZATION COLLISIONS

lib.hpp

```
template <typename T>
struct custom_point<T,
    std::enable_if_t<
        has_begin_end_v<T>
    >
>
{
    /*… */
};
```

E
R
R
O
R

user.hpp

```
template <typename T>
struct custom_point<
    my_vector<T>
>
{
    /*… */
};
```

DRAWBACK III: "ARCANE" KNOWLEDGE

- "The code isn't working"
  - navigating the syntax and rules of template instantiations means glazed over looks and general confusion
  - when providing support, usually teach user about partial template specialization (or just give them an example)

- Typical C++ users just want to write simple code
  - Classes (with or without static functions): ✔
  - Functions (exported, inline, etc.): ✔
  - Templates (rules of ODR, visibility of specialization at time of use, etc.): ✖

DRAWBACK IV: VISIBILITY AND DEFAULTS 👀

- Is the class template specialization…
  - visible in all possible translation units where it may be used?
  - body dependent on macros that are not defined for the entire build (and its dependencies?)

- Silent ODR violation that compiles, links and runs on all known compilers.
  - Problematic with all compile-time extension points where default is not a noisy error
  - 📦 Specialization must be tightly packaged with class when used!

DRAWBACK V: HEADER BLOAT

- Header contamination becomes a real problem to avoid ODR issues
  - entirety of sol2 comes along for the ride

- produces longer build times
  - avoided with careful forward declaration of every required template and class
  - unfortunately, the standard itself does not provide forward-declaring headers
  - "modules will solve it" – unfortunately, little tangible evidence I can personally provide

STD::HASH<T>

- Employs same struct specialization technique, but
  - is substantially simpler
  - has only one template argument

- Well-used, so this simple case has become idiomatic
  - lack SFINAE makes it easier to teach

STD::HASH<T>

```cpp
namespace std {

    template<>
    struct hash<two_things> {
        size_t operator()(const two_things& tt) const noexcept {
            auto h1(std::hash<int>{}(tt.a));
            auto h2(std::hash<bool>{}(tt.b));
            return my_hash_mix(h1, h2);
        }
    };

} // namespace std
```

STD::HASH<T>: BENEFITS AND DRAWBACKS

- Avoids arcane knowledge requirements by
  - being extraordinarily simple (write a function call operator)
  - not having a SFINAE parameter (avoids mutual exclusion)

- Takes core specializations away from user
  - Pre-defined for enum types, integral types, etc.

- Same visibility / header contamination issues
  - <functional> comes along for the ride 🐪, no forward declarations!

FREQUENT LIBRARY VENDOR COMPLAINT

- "They are opening up namespace std / my namespace ⚙!"
  - paper to solve this presented in Rapperswil, Switzerland, 2018; [p0665](#)
  - allows a user to specialize outside classes in the namespace where the class is defined

- Library vendors are hyper-sensitive to users opening up namespace std
  - people have done all sorts of interesting things in their code bases
  - required all large stdlib implementations to employ `__ugly _Identifiers`

(TEMPLATE) FUNCTION OVERLOADING

- Step into the namespace of the function and add a similar name
  - does not depend on Name Lookup to "find" the function in associated namespaces

- Usually explicitly blessed by library author as "possible"
  - old usage: viable way to customize `std::swap`

- Case study
  - Boost.Serialization

ADDING OVERLOADING TO BOOST.SERIALIZATION:

```cpp
namespace boost { namespace serialization {

    template <class Archive>
    void serialize(Archive& ar, two_things& g, unsigned int version) {
        ar & tt.a;
        ar & tt.b;
    }

}} // namespace boost::serialization
```

OVERLOADING: BENEFITS AND DRAWBACKS

- Direct additions to namespace separate extension point from target
  - benefit: if optional and not required, user can move customization function to independent header / implementation files
  - drawback: if required and not optional, then separation may not be desired and causes boilerplate

- Same complaint from library vendors
  - opening up other namespaces 💢!
  - potential for name collisions and similar

ARGUMENT-DEPENDENT LOOKUP

- Complicated set of rules

- Rely on namespaces of arguments to add additional symbols to unqualified calls
  - primary intentional use: "generic" (templated) code to work with arbitrary types
  - primary unintentional use: operators to "just find the right call" for
    a == b

- Case studies:
  - std::swap (the wrong way)
  - std::ranges / range-v3 (the right way)

ARGUMENT-DEPENDENT LOOKUP: SWAP

- swap(a, b) // invokes ADL because call name is unqualified
  - looks in the namespace of a and b, as well as the current scope's namespace
  - likely a bug in generic algorithm if written outside std/a or b are not std

- std::swap(a, b) // does not invoke ADL because call name is qualified
  - looks only in namespace std
  - likely a bug if used in a generic algorithm

- Proper way:

```
using std::swap;
swap(a, b);
```

"STD SWAP TWO-STEP": VERBOSITY IS FAILURE

"The problem with the Two-Step is that it forces users to type *more* to do the right thing. FAIL. Most damning, it requires users to either blindly memorize and regurgitate the Two-Step pattern, or worse: understand two-phase name lookup in templates."

– Eric Niebler, October 2014

ARGUMENT DEPENDENT LOOKUP: RANGE-V3

- Create a callable function object which does the two-step with an internal detail namespace's swap
  - Invokes ADL but prevents qualified call to ns::swap(a, b) being a bug
  - ADL is done "for you": function object takes care of it

ADL DONE RIGHT™

```cpp
namespace std { namespace detail {
    template <Swappable A, Swappable B> // important!
    void swap (A& a, B& b) { /* default implementation */ }

    struct swap_func {
        template <typename A, typename B>
        void operator()(A& a, B& b) const noexcept {
            using swap;
            swap(a, b);
        };
    };
}} // namespace std::detail
```

ADL DONE RIGHT™

- Create a constexpr object of the proper name sitting in the namespace

```
namespace std {
    inline constexpr const auto swap = detail::swap_func{};
} // namespace std
```

ADL DONE RIGHT™: VERY SIMPLE

```cpp
struct two_things {
    int a;
    bool b;

    // just this
    friend void swap (two_things& left, two_things& right);
};
```

WAIT, IS THAT A FRIEND FUNCTION?

- Friend functions contain a few encapsulation benefits and help avoid name collisions

- friend functions are the same as free functions, but:
  - hidden from qualified (`my_namespace::func_name`) calls due to being inside the class
  - findable by calls which invoke ADL

FRIEND FUNCTIONS CASE STUDY: ABSEIL

- Abseil uses this extensively for its customization points
  - in particular, AbslHashValue

```cpp
struct Circle {

    template <typename H>
    friend H AbslHashValue(H h, const Circle& c) {
        return H::combine(std::move(h), c.center_, c.radius_);
    }


private:
    std::pair<int, int> center_;
    int radius_;
};
```

BENEFITS: ADL DONE RIGHT™

- No Two-Step;
  - no subtle missed bugs in generic code
  - no inconsistency in "always qualify your calls"
  - Customization point writer gets there "first"
    - impose initial base-level concepts on the type

- Allows user to define swap in namespace next to class / as friend function
  - just a function: easy to write and read

DRAWBACK I: OVERLOADING CATCH-ALLS

- Base implementation provided by author must SFINAE away or it will catch all calls and hard-error everything
  - must use decltype SFINAE, concept, or std::enable_if_t

- Users may not properly constrain their overloads and write catch-alls
  - If users write a "generic" catch-all and do not properly constrain, the extension point is ruined for everyone

DRAWBACK II: HIGH COLLISIONS

- Does your function take perfect forwarding references?
  - prepare to cry: overloads in the same space may consume more calls than intended
  - worse: they might even unintentionally work but do the non-performant / wrong thing!

```cpp
namespace std {

    template <class Pointer, class Smart, class... Args>
    auto out_ptr(Smart& s, Args&&... args) noexcept;

} // namespace std
```

BUT EVERYONE WILL PROPERLY CONSTRAIN!

And other lies I told myself after I read Eric's blog post…

NO. 😫

- They will not constrain it.

- They will not use
  "only concrete types".

- The world is not full
  of only experts.

ThePhD 02/02/2019
No idea why that trait is triggering and causing the error. I'll have to try to construct a small, reproducible error at some point, but my fix should get you going at least.
Oh.
That's why.
`tgui::to_string` is templated, and lying.
This has nothing to do with any of the classes or anything you wrote.

linbob 02/02/2019
i didnt think so

ThePhD 02/02/2019
`tgui::to_string` is a templated function that invokes `operator<<(YourClass&)` in order to serialize a string.
sol2 picks that up when it goes through its complicated list of "does this thing have a `to_string` function".
`tgui` did not properly SFINAE-constrict its overload, so it says it takes `ContextMenu` when it, in fact, does not and the code inside cannot.
Templates were a mistake...
ADL was a mistake too.
Either way, nothing you or I can do about it. `tgui` would need to add some proper SFINAE to their templated function.
(Or just, you know. Not write a function like that.)

ADL DONE... RIGHT™?

- "The Best Anyone Could Have Done With The Tools At Hand."

- range-v3 niebloids (begin, end, iter_move, dereference, etc.) contain no opt-out mechanism
  - does not prevent the ADL problem for unintentionally bad actors
  - makes it even more apparent when it does happen
  - cannot call "just the standard {begin/end/swap}" because it exists in an implementation-defined detail namespace now

ADL AND OVERLOADING: BIGGER PROBLEMS

- Functions can catch unintended calls
  - even if they are not templated
  - consider void* pointer conversions, derived -> base conversions, and more

- Results in a huge problems for ADL and overloading
  - unintended "catches" of base types and other things a user would find surprising

- Case study: sol3

SOL3: ADL EXTENSION POINTS

- Associated extension points
  - sol::stack::get a value maps to sol_lua_get
  - sol::stack::check a type maps to sol_lua_check
  - sol::stack::push maps to sol_lua_push

```
int sol_lua_push(sol::types<two_things>, lua_State* L,
    const two_things& things);

two_things sol_lua_get(sol::types<two_things>, lua_State* L,
    int index, sol::stack::record& tracking)

template <typename Handler>
bool sol_lua_check(sol::types<two_things>, lua_State* L,
    int index, Handler&& handler, sol::stack::record& tracking)
```

SOL3: ADL EXTENSION POINT FUNCTION CALL

```cpp
template <typename T, typename... Args>
int push(lua_State* L, T&& t, Args&&... args) {
        if constexpr (is_adl_sol_lua_push_v<T, Args...>) {
                return sol_lua_push(...);
        }
        else {
                /* hit default if constexpr internals */
        }
}
```

SOL3: TRAIT IMPLEMENTATION

```cpp
template <typename... Args>
using adl_sol_lua_push_test_t = decltype(sol_lua_push(
    static_cast<lua_State*>(nullptr), std::declval<Args>()...
));


template <typename... Args>
inline constexpr bool is_adl_sol_lua_push_v =
    is_detected_v<adl_sol_lua_push_test_t, Args...>;
```

SOL3: AN EXAMPLE OF OLD PROBLEMS

- Why type tags?
  - To solve conversion problems; for example, old conversion points

```cpp
int sol_lua_push(lua_State* L, void* vp);

struct unrelated {};

int main (int, char*[]) {
    sol::state lua{};
    unrelated obj{};
    unrelated* some_pointer = &obj;
    // calls the above, not the default!
    sol::stack::push(lua, some_pointer);
    return 0;
}
```

DRAWBACKS: ADL AND OVERLOADS

- Must guard against conversions
  - can develop smarter and more complicated traits
  - prefer a type tag if the space of the ADL is unconstrained

- Have templated functions that take multiple perfect-forwarding arguments?
  - just do not bother here; overload resolution will drive users crazy

TRAITS TYPES

Injecting compile-time extensions into the type system

TRAITS/POLICIES/AGENTS: TEMPLATED CLASSES

- Deployed for classes which need customizability
  - std::basic_string<CharType, TraitsType>
  - std::basic_ostream<CharType, TraitsType>
  - std::vector<T, Allocator>
  - std::map<Key, Value, Predicate, Allocator>
  - glm::mat<Rows, Colums, Type, Precision>
  - nlohmann::basic_json<MapType, ArrayType, StringType, BoolType, SignedIntegerType, UnsignedIntegerType, FloatingType, Allocator, Serializer
    >;

TRAITS: BAD REPUTATION

- Interfaces for char_traits, allocators, and more from the standard
  - early designs using new features in the standard
  - not thoroughly vetted
  - imbued in things it had
    no business being in (IO and friends)

Member functions

assign [static]	assigns a character (public static member function)
eq [static] **lt** [static]	compares two characters (public static member function)
move [static]	moves one character sequence onto another (public static member function)
copy [static]	copies a character sequence (public static member function)
compare [static]	lexicographically compares two character sequences (public static member function)
length [static]	returns the length of a character sequence (public static member function)
find [static]	finds a character in a character sequence (public static member function)
to_char_type [static]	converts int_type to equivalent char_type (public static member function)
to_int_type [static]	converts char_type to equivalent int_type (public static member function)
eq_int_type [static]	compares two int_type values (public static member function)
eof [static]	returns an *eof* value (public static member function)
not_eof [static]	checks whether a character is *eof* value (public static member function)

TRAITS: LATER ITERATIONS SUCCESSFUL

- std::map and std::unordered_map made better use of traits
  - Predicate and Hash follow guidelines of std::hash
  - Single-responsibility principle for Predicate and Hash

- nlohmann::json is a templated type with sensible defaults
  - just change template parameter details if you do not like them!

DRAWBACK: TRAITS

- Change the template, change the type
  - cannot interoperate with sibling types by default (unless explicitly programmed in)

- Brittle ABI
  - change default template parameters -> change name mangling
  - change template name -> any using/typedefs change name mangling

- "Too much customizability"
  - Need to resist temptation to repeat mistake of std::char_traits

TRAITS: BETTER FOR MULTITHREADED ENVIRONMENTS

- For when user has more control over the system and does not have to work in existing code
  - each class can have highly customized behavior specific to needs
  - avoids needing to share a single global universe of overload resolution / ADL space with others
  - great for application space
  - great for environments that are already type-rich / generic (C++)

- Can deploy one trait class in one area, another in a separate area
  - will not crash the application space

ADL: BETTER FOR STATELESS CONSISTENCY

- "pick up and play" feeling
  - works from anywhere, obeys the same rules (however complicated)
  - better for the library developer space

- sol3 picked ADL extension points for many reasons
  - user had to be able to be consistent across translation units
  - harder to have fixed ABI with trait-based state classes
  - sol::state / sol::state_view can interop with the scripting layer
  - better for handling type-deficient Lua and C coding environment (interop)

STRUCT SPECIALIZATION: BEST FOR PRECISE MATCHING

- Class SFINAE is some of the most expensive SFINAE one can perform
  - second only to non-concept function SFINAE in template arguments
  - SFINAE done on the return type of a function is faster
  - if constexpr is fastest

- Template matching is very precise and does not do even basic conversions
  - less flexible than overload conversions
  - must define template for base class, first derived, second derived, etc. even if they are all do the same thing

- Has no overload resolution issues (!!)

FUTURE TALKS?

- Fully Runtime Extension Points for C++ Applications
  - Unassisted Runtime DLL Loading
    - LoadLibrary + GetProcAddress / dlopen + dlsym
  - Hooking
    - mhook / LD_PRELOAD
  - Hot Reloading
  - Debug Gap Placements to compile new code into
    - Visual C++ debug compilation

- "Versioning" for the purposes of loading/calling code
  - ABI restrictions and friends

THANK YOU !

- Eric Feselier, Titus Winters
  - Challenged me to research generic
    extension mechanisms for std::out_ptr (p1132)

- Isabella Muerte
  - "Tell them an ADL customization point is insane"
    (she was right; overloading concerns were insane)

- Lounge<C++>

#include <C++>
www.includecpp.org

QUESTIONS