CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich

Advanced C++ Course 2023

Generic programming tools: CPOs

CSCS

Customization mechanisms for generic interfaces

When writing generic algorithms and libraries

- information needs to be transferred from the user to library (author)
- compile time:
  - (syntactical) correctness
  - check type requirements (function signature, type introspection, concept)
  - compute associated types (traits, `decltype`)
- run time:
  - object properties (domain specific)
- customization
  - user must be able to *easily* customize the *default*

Customization mechanism: interface wish list

What we would like to have: customization of interfaces (functions) which

- are non-intrusive
- are explicitly opt-in
- prevent incorrect opt-in (error when wrong signature)
- are able to provide default implementations
- are simple to correctly invoke *customized* version when *default* exist
- are hard to incorrectly invoke the *default* when *customized* exist
- are clear in their intent when looking at the code (what and how to customize if at all)
- are easy to verify for a given type

1. https://brevzin.github.io/c++/2020/12/01/tag-invoke/

Dynamic Polymorphism

- traditional way of defining interface in OOP
- works by using `virtual` functions in c++

Example:

```
struct base {
    virtual void foo(int) = 0;
};

struct derived : base {
    virtual void foo(int) override;
};
```

can have default implementation or not (*pure virtual* function)

`override` keyword makes sure that we actually customize the interface
(use `-Wsuggest-override`)

How are we doing?

virtual function		
non-intrusive	no	inheritance from base
explicitly opt-in	yes	have to inherit
prevents incorrect opt-in	yes	with `override`
provides default implementations	yes	non-pure base class
simple to invoke	yes	just call from pointer-to-base
hard to incorrectly invoke	yes	just call from pointer-to-base
code shows intent	yes	`virtual` functions show customization
easy to verify for a given type	yes	when `derived*` is convertible to `base*`

Issues

- intrusive!
- otherwise fine?

Other issues

- virtual function overhead
- usually requires heap allocation
- what if return type of interface is dependent on type?

```
struct array_like {
    virtual unsigned long size() const = 0;
    virtual bool operator==(array_like const&) const = 0;
    virtual auto operator[](unsigned long) -> ????;
};
```

what about this?

- `array_like` is parametrized interface

```
template<typename T>
struct array_like {
    virtual unsigned long size() const = 0;
    virtual bool operator==(array_like const&) const = 0;
    virtual T& operator[](unsigned long) = 0;
};
```

- inherit from `array_like<T>`?

```
struct my_complex_array : array_like<std::complex> {
    virtual unsigned long size() const override;
    virtual bool operator==(array_like const&) const override;
    virtual std::complex& operator[](unsigned long) override;
}
```

- virtual call overheads persist
- introduced many interfaces:
  `array_like<std::complex>`, `array_like<double>` etc

Static Polymorphism: Class Template Specialization

- no additional runtime overhead, usually doesn't require allocation

```
// A formatter for objects of type T.
template <typename T,
          typename Char = char,
          typename Enable = void>
struct formatter {
  // A deleted default constructor indicates
  // a disabled formatter.
  formatter() = delete;
};
```

- formatter class definition for `fmt::format`: pretty empty
- from documentation: needs `parse` and `format` function

class template specialization		
non-intrusive	yes	can specialize any class
explicitly opt-in	yes	works only through specialization
prevents incorrect opt-in	partially	may still compile, but may not work correctly (concept could help)
provides default implementations	no	not possible (need to specialize whole class)
simple to invoke	yes	just call from `formatter<T>::format`
hard to incorrectly invoke	no	`formatter<U>::format` may just work
code shows intent	no	class template and `Enable` is a hint that it needs to be specialized (concept could help)
easy to verify for a given type	no	can check whether specialization exists (concept could help)

Static Polymorphism: Customization Points

- familiar from standard library functions such as `std::swap`
- no additional runtime overhead, usually doesn't require allocation
- work through ADL: argument-dependent (name) lookup
- need specific incantation:

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

make standard namespace available
unqualified call to swap (note: not `std::swap(a, b)`)

make interface available through (hidden) friend member funcion (technical reason: reduce set of functions that can be found by ADL)

```
struct x {
    int data;

    friend void swap(x& a, x& b) {
        b.data = std::exchange(a.data, b.data);
    }
};
```

- `std::swap` can also handle types without explicit `swap` interface
  - move constructible, move assignable
  - is a default implementation

customization points		
non-intrusive	yes	can overload `swap` for any type
explicitly opt-in	no	opt-in is implicit
prevents incorrect opt-in	partially	may still compile, but may not work correctly (concept could help)
provides default implementations	yes	`std::swap(...)` works for many types out of the box
simple to invoke	kinda	need to remember to make namespace available
hard to incorrectly invoke	no	use `std::swap(...)` by mistake, ADL woes
code shows intent	no	`std::swap` is just a function template in the standard library
easy to verify for a given type	no	only with separate concept

Intermezzo: What is ADL again?

You don't have to qualify the namespace for functions if one or more argument types are defined in the namespace of the function.
-- Nicolai Josuttis, The C++ Standard Library: A Tutorial and Reference

- was introduced to solve a specific problem with operators

```
namespace n {
    struct A {
        A operator++();
    };
    std::ostream& operator<<(std::ostream&, A const&);
}

n::A a;          // create object from namespace n
a++;             // this is ok -> call member function on a
std::cout << a; // how to find the right function (operator)?
```

- solution: whenever we see an **unqualified** call to a possibly overloaded operator look all the namespaces associated with the types of the arguments to the operator
  - here: `global namespace`, `namespace std` and `namespace n`

- ADL was extended to non-operator functions such as `swap` and `get_next`
  - reasoning `x.foo()` should be expressable as `foo(x)` instead of `X::foo(x)`
- when does ADL apply?
  - name lookup (building a candidate set) for an **unqualified function call** (no `::`-qualification)
- when does ADL not apply?
  - not a function call: `(foo)(x)`
  - callee **is not a function**

lookup rules

- looks only at the **types** of the arguments (after resolving type aliases)
- template arguments are ignored (do not add namespace of template argument types)
- all arguments are considered in no particular order
  - produces zero or more associated types and associated namespaces, via a complicated ad-hoc process
  - for associated types: consider only the (namespace-scope) friends

```
namespace N {
    struct A {
        enum E { E0 };
        friend void f(E);
        static void g(E);
    };
}

namespace M {
    void f(int);
    void g(int);
    void test() {
        N::A::E e;
        f(e);  // ADL considers N::f (friend of N::A)
        g(e);  // ADL does not consider N::A::g
    }
}
```

- algorithm for lookup:
  - create sets of associated namespaces and associated types for each argument
  - merge them all together (and add our current namespace and all its parents)
  - find declarations of the name `foo` in any of these namespaces
  - do overload resolution for this call

- When does ADL go wrong?

```cpp
// print library
namespace lib1 {
    template <typename T>
    void print(T x) {
        std::cout << x << std::endl;
    }

    template <typename T>
    void print_n(T x, unsigned n) {
        for (unsigned i = 0; i < n; ++i)
            print(x);
    }
}
```

```cpp
// other library
namespace lib2 {
    struct unicorn { /* unicorn stuff goes here */ };

    std::ostream& operator<<(std::ostream& os, unicorn x) { return os; }

    // Don't ever call this!  It just crashes!  I don't know why I wrote it!
    void print(unicorn) { *(int*)0 = 42; }
}

int main() {
    lib2::unicorn x;
    lib1::print_n(x, 10); // boom
}
```

- form point of view of `lib1`: we have **no way of knowing** what function will be called by `print_n` *a priori*

1. https://stackoverflow.com/questions/2958648/what-are-the-pitfalls-of-adl

Static Polymorphism: Customization Point Objects (CPOs)

- idea: separate
  - the piece that the user needs to specialize (found by ADL) and
  - the piece that the user needs to invoke (*must not* specialize, ADL turned off)
- consider this helper function:

```
namespace std2 {
    template<class A, class B>
        requires /* swappable constraints */
    void swap2(A& a, B& b) {
        using std::swap;
        swap(a, b);
    }
}
```

- what would we gain?

```
using namespace std2;
swap2(a, b);
```

ADL may break this code!

- switch off ADL by using objects (remember ADL does not apply for non-functions)

```cpp
namespace std2 {
    namespace hidden {

        // "poison pill" to hide overloads of swap() that might be found in parent namespace
        // we want to limit to only finding overloads by ADL.
        void swap() = delete;

        // define function object with operator() that forwards to call to unqualified 'swap()'
        struct swap_helper {
            template<class A, class B>
                requires /* swappable constraints */
            void operator()(A& a, B& b) const {
                using std::swap;
                swap(a, b);
            }
        };
    }

    // use inline namespace to avoid potential conflicts with hidden friend functions
    // which add functions with name 'swap' into enclosing namespace
    inline namespace swap_cpo {
        inline constexpr hidden::swap_helper swap;
    }
}
```

```cpp
struct x {
    int data;
    friend void swap(x& a, x& b) {
        b.data = std::exchange(a.data, b.data);
    }
};
```

- now both ways work

```cpp
using namespace std2;
swap(a, b);
```

```cpp
std2::swap(a, b);
```

- important gain: we can now check constraints on types (concept checking)

customization points objects		
non-intrusive	yes	can overload `swap` for any type
explicitly opt-in	no	opt-in is implicit
prevents incorrect opt-in	partially	library author can write checks (failure is earlier)
provides default implementations	yes	`swap_helper` *is* a default implementation
simple to invoke	yes	both qualified and unqualified calls work
hard to incorrectly invoke	yes	qualified call works equally well
code shows intent	no	hard to see from objects
easy to verify for a given type	partially	with separate concept

other issues

- ususally requires more code
- customization point objects reserve their identifier globally (two libraries with same CPO name may clash)

Static Polymorphism: tag_invoke

- Solve shortcommings of CPOs
  - do not claim identifier name globally (which is needed for ADL)
  - handle (type-erased) wrapper types transparently
- idea: use one CPO called `tag_invoke` which takes an arbitrary CPO as argument

implementation

```
namespace hidden {
    struct tag_invoke_fn {
        template<typename CPO, typename... Args>
        constexpr auto operator()(CPO cpo, Args&&... args) const /* noexcept clause */
            -> decltype(tag_invoke((CPO &&) cpo, (Args &&) args...)) {
            return tag_invoke((CPO &&) cpo, (Args &&) args...);
        }
    };
}
inline constexpr hidden::tag_invoke_fn tag_invoke{};

// some more helper traits and values:
template <auto& CPO>
using tag_t = ...;

template <typename CPO, typename... Args>
using tag_invoke_result_t = ...;

template <typename CPO, typename... Args>
inline constexpr bool is_tag_invocable_v = ...;
```

use

```
namespace std2 {
    // simplified way to write CPO
    inline constexpr struct swap_fn {
        template<typename A, typename B>
            requires /* swappable constraints */
        auto operator()(A& a, B& b) const /* noexcept clause */
        -> decltype(tag_invoke(*this, a, b)) { // trailing return type SFINAE
            return tag_invoke(*this, a, b);
        }
    } swap{};
}

struct x {
    int data;
    friend void tag_invoke(tag_t<std2::swap>, x& a, x& b) {
        b.data = std::exchange(a.data, b.data);
    }
};
```

- both ways work

```
using namespace std2;
swap(a, b);
```

```
std2::swap(a, b);
```

- globally reserve a single name: `tag_invoke`

	`tag_invoke`	
non-intrusive	yes	can overload `tag_invoke` for any type
explicitly opt-in	yes	opt-in is explicit
prevents incorrect opt-in	partially	library author can write checks (failure is earlier)
provides default implementations	yes	by adding overload for `operator()` in our CPO
simple to invoke	yes	qualified call
hard to incorrectly invoke	yes	qualified call
code shows intent	partially	recognize `tag_invoke` friend function
easy to verify for a given type	partially	with separate concept

other issues

- ususally requires similar amount of code than CPOs
- not yet standard (`tag_invoke` requires some machinery behind the scenes)

Conclusions

- many nuances to consider
- ADL can lead to subtle errors
- lack of language feature requires elaborate library (workarounds)
- `tag_invoke` is best option for now
  - if we care about a few free functions
- if we need to create a whole type for an interface
  - other options (inherit mixins with CRTP, class template specializations etc)

References

- https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1895r0.pdf
- https://brevzin.github.io/c++/2020/12/19/cpo-niebloid/
- https://brevzin.github.io/c++/2020/12/01/tag-invoke/
- https://quuxplusone.github.io/blog/2019/08/02/the-tough-guide-to-cpp-acronyms/#cpo
- https://quuxplusone.github.io/blog/2019/04/26/what-is-adl/
- https://www.fi.muni.cz/pv264/files/pv264_s06b_niebloids.pdf