

Customization Methods: Connecting User And Library Code

Inbal Levi

Inbal Levi - Customization Methods: connecting user and library code - CppCon 2023

1

Who Am I?

- A C++ Developer at Millennium.
- Active member of ISO C++ work group (WG21):
 - Library Evolution Work Group Chair
 - Israeli NB Chair
 - ISOC++ foundation & Boost foundation board member
- I love software design
- I also have a passion for cataloguing, which is the main reason I'm giving this talk
- The presentation presents my views and opinions only.

Inbal Levi - Customization Methods: connecting user and library code - CppCon 2023

Outline

Part 0: What is a Library

Part I: What are Customization Points (+ Some History...)

Part II: An overview of CPs methods (+ Some History...)

Part III: Comparing CPs methods

Part IV: What's next?

Part 0: What is a Library

Inbal Levi - Customization Methods: connecting user and library code - CppCon 2023

Part 0: What is a Library

Library

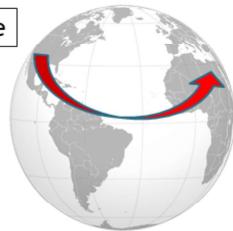
```
template <typename T>
void Logger(T t) {
    std::cout << "Log: " << t << '\n';
}
```

User

```
struct Planet {
    friend ostream& operator<<(ostream& os,
        const Planet& s) {
        os << "Planet[" << s.name << "] ";
        return os;
    }
};
```

```
Planet p;
Logger(p); // Log: Planet[Saturn]
```

Alice



Bob



Inbal Levi - Customization Methods: connecting user and library code - CppCon 2023

5

@@@ All of library code is in **LibSpace**, user code without frame is in **UserSpace**

Part 0: What is a Library

- From Wikipedia:
“A [software] library is a collection of non-volatile resources used by computer programs, often for software development.”
- Library code is often shared between **different developers**.
- Possibly from different parts of the world, who have never met...
- We need the ability to communicate and integrate library and users' code.
- **Customization points are the answer!**

Part I: What Are Customization Points

Inbal Levi - Customization Methods: connecting user and library code - CppCon 2023

Part I: What Are Customization Points

- “Customization Points” mentioned by Eric Niebler, in a blog from 2014: [Customization Point Design in C++11 and Beyond](#)
- In 2015, CP appeared in a WG21 paper: “[N4381: Suggested Design for Customization Points](#)”

“A *customization point*, as will be discussed in this document, is a function used by the Standard Library that can be overloaded on user-defined types in the user’s namespace and that is found by argument-dependent lookup.”

- In this talk, we will use:

“A *customization point* is an integration method exposed by a Library, that can be used by the user to customize (“hook”) the library according to their needs.” (*)

(*) The CPs from Eric’s paper will be referred to as “Original CPs”

→ Take Away I: CPs are a means of integration (of library and user code)

Part I: What Are Customization Points

- Users need to be able to understand which CPs exist
- To be able to opt-in: add implementations based on their own understanding
- To be able to avoid accidental opt-in: “run over” library implementation
- Barry Revzin talked about both this and the previous aspects in his paper: “P2279R0: We need a language mechanism for customization points”, we will take a similar approach.

→ Take Away II: CPs are a means of communication (between library authors and library users)

Part I: What Are Customization Points

- CPs determine which entities are shared between the Library and User spaces
- Entities can be: functionality, data and terminology
- We will focus on user types sharing functionality with library types and algorithms

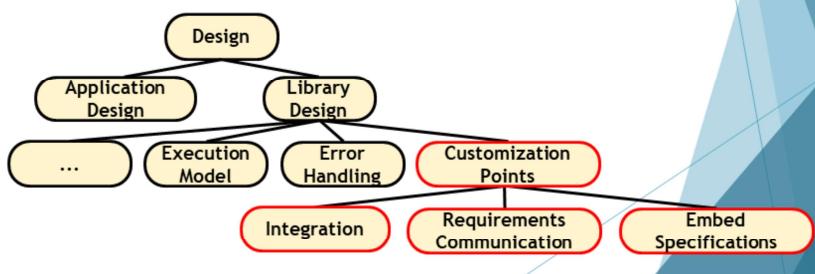
→ Take Away III: CPs are a means to embed specification (from the user into the library)

@@@ As opposed to library sharing entities with other libraries

Part I: What Are Customization Points

- To summarize:

- Take Away I: CPs are a means of integration
- Take Away II: CPs are a means to communicate requirements
- Take Away III: CPs are a means to embed specification



Part II: An Overview Of CPs Methods

Inbal Levi - Customization Methods: connecting user and library code - CppCon 2023

12

Part II: An Overview Of CPs Methods

- C++ was “born” with Customization Point methods, more were added over the years
- In this talk, we will go over the following:
 1. Inheritance
 2. Class Template Specialization
 3. (Original) Customization Points (ADL)
 4. Customization Point Objects
 5. Concepts
 6. `tag_invoke` (not in the standard)
 7. Customizable Functions (not in the standard)
 8. Reflection (not in the standard)
- We will skip through some of the details (the code snippets contains more details)

Inbal Levi - Customization Methods: connecting user and library code - CppCon 2023

13

+ @@@ Deducing this removes the need for CRTP

Part II: An Overview Of CPs Methods

- C++ was “born” with Customization Point methods, more were added over the years
- In this talk, we will go over the following:
 - **1. Inheritance**
 - 2. Class Template Specialization
 - 3. (Original) Customization Points (ADL)
 - 4. Customization Point Objects
 - 5. Concepts
 - 6. `tag_invoke` (not in the standard)
 - 7. Customizable Functions (not in the standard)
 - 8. Reflection (not in the standard)

Inbal Levi - Customization Methods: connecting user and library code - CppCon 2023

14

2 Slides

Part II: Method 1: Inheritance

Library

```
1 struct LoggedType {
    virtual void pMembers() = 0; // Mandatory
    virtual void pName() {...} // Functionality
    using Name = std::string; // Terminology
    static const int ver = 1; // Terminology
    Name name_; // Data
};

void Log(LoggedType& t) {
    t.pName();
}
```

User

```
struct Spaceship : public LoggedType {
    virtual void pMembers() override {} // Mandatory
    virtual void pName() override {} // Specialized
};

Spaceship s; // Usage
Log(s); // pName called
```

- Library side:
 1. Library defines “LoggedType” which contains functionality, data, and terminology
 2. Mandatory interface: by pure virtual functions (“pMembers”, no default)
 3. Optional interface: (with default) by virtual functions (“pName”, have a default)
- User side:
 1. User can inherit from “LoggedType” to gain customizable functionality, data and terminology
 2. User can pass a UserType (Spaceship) object to “Log” to gain functionality

Inbal Levi - Customization Methods: connecting user and library code - CppCon 2023

15

1. <https://godbolt.org/z/eTznPrn05>

@@@ Use default if there's motivation for default implementation (something which can be done generally)

@@@ All of library code is in LibSpace, user code without frame is in UserSpace

Part II: Method 1: Inheritance

Library	User
<pre>struct LoggedType { virtual void pMembers() = 0; // Mandatory virtual void pName() {...} // Functionality using Name = std::string; // Terminology static const int ver = 1; // Terminology Name name_; // Data }; void Log(LoggedType& t) { t.pName(); }</pre>	<pre>struct Spaceship : public LoggedType { virtual void pMembers() override {} // Mandatory virtual void pName() override {} // Specialized }; Spaceship s; Log(s);</pre>

- Characteristics:
 1. Sharing: Interface, Functionality, Data, and Terminology **as a group** ↗
 2. Easy to opt-in, hard to accidentally opt-in ↘
 3. Runtime overhead, Limits on return type (partially overcome by complex methods) ↘
 4. Non-intrusive into library namespace ↗
 5. Intrusive into the type ↘
- Examples: Everywhere

Inbal Levi - Customization Methods: connecting user and library code - CppCon 2023

16

1. @@@ Terms:

1. Overload: multiple functions with the same name, for different num of param / different params types (resolved by overload resolution)
2. Hide – when function declaration in the inner space hides the one from the outer space
3. Covariant return type:
https://en.wikipedia.org/wiki/Covariant_return_type
4. 5 (*) “Intrusive” here Has nothing to do with intrusive containers

(*) P2279: Intrusive’s definition: intrusive into the type which is being opt-into. “You simply cannot opt types that you do not own into an abstract interface, with the fundamental types not being able to opt into any abstract interface at all.”

Part II: An Overview Of CPs Methods

- Methods:
 1. Inheritance
 - 2. **Class Template Specialization**
 3. (Original) Customization Points (ADL)
 4. Customization Point Objects
 5. Concepts
 6. `tag_invoke` (not in the standard)
 7. Customizable Functions (not in the standard)
 8. Reflection (not in the standard)

Inbal Levi - Customization Methods: connecting user and library code - CppCon 2023

17

5 Slides

Part II: Method 2: Class Template Specialization (Type) V1

Library

```
template <typename T>
struct Container {
    T at(size_type idx) {...}      // Functionality
    using iterator = ...;          // Terminology
    constexpr T* data();           // Terminology
};
```

```
template <typename RandomIt>      // 2nd level Func
void Sort (RandomIt &first, ...) {
    ...
    // Assuming Terminology
}
```

User

```
struct Spaceship { // (at least) Erasable (C++17)
    ...
};
```

```
Container<Spaceship> fleet = { ... };
fleet.at(0);                                // Functionality
Sort(fleet.begin(), fleet.end()); // 2nd level
```

- Library side:
 1. Library defines a template class “Container”
 2. Library defines “Sort” function (partial ability restrict it **specifically** for “Container” (*)
- User side:
 1. Create template specialization for the specific type
 2. User can use functionality from “Container” in specialized “Container<Spaceship>”
 3. User gains “Sort” functionality (2nd level functionality)

(*) Can be partially restricted by type traits or Concepts

18

@@@ (*) Concepts are only limiting by behavior / qualities

Part II: Method 2: Class Template Specialization (Type) V2

Library

```
1 template <typename T>
2 struct LoggedType {
3     LoggedType() = delete;           // No default
4     // (Error on CTOR)
5     void pName() {...}             // Functionality
6     using Name = std::string;       // Terminology
7     Name name_;                   // Data
8 };
9
10 template <typename T>
11 void Log (T &t) {
12     std::cout << "Default Log\n";
13 }
```

User

```
template <> // Class specialization
struct LoggedType<Spaceship> {
    LoggedType() {}
};

LoggedType<Spaceship> s; // User CTOR
Log(s); // "Default Log"
```

- Library side:
 1. Library defines template class “LoggedType” with deleted CTOR
 2. Library defines “Log” function (*)
 - User side:
 1. Override (**the entire**) implementation by template specialization
 2. User can instantiate and use “LoggedType<UserType>”
 3. User gains limited external “Log” functionality
- (*) Can be partially restricted by traits or Concepts

19

@@@ All of library code is in **LibSpace**, user code without frame is in **UserSpace**

1. Basic: <https://godbolt.org/z/nej3977z8>

Part II: Method 2: Class Template Specialization (Type) V3

Library

```
2 template <typename T>
struct LoggedType {
    LoggedType() = delete;           // No default
                                    // (Error on CTOR)
    void pName() { print("{}", name_); } // Func
    using Name = std::string;        // Terminology
    Name name_;                      // Data
};

template <typename T>
void Log (T &t) {
    pName();
}
```

User

```
template<> // CTOR specialization (**)
LoggedType<Spaceship>::LoggedType() :name_("Apollo")
{ }

template<> // pName specialization (**)
void LoggedType<Spaceship>::pName() {
    print("{}[{}]", name_, version_);
}

LoggedType<Spaceship> s;           // User CTOR
Log(s);                          // "Apollo[11]"
```

- Library side:
 1. Library defines template class “LoggedType” with deleted CTOR
 2. Library defines “Log” function (*)
 - User side:
 1. Override (**some of the**) implementation by template specialization (**)
 2. User can instantiate and use “LoggedType<Spaceship>”
 3. Assuming “pName()” is as expected, user gains “Log” functionality (2nd level functionality)
- (*) Can be partially restricted by traits or Concepts (**) In containing namespace

20

@@@ This is done in the **lib containing namespace** (and not on user space)
2. Partial template specialization: <https://godbolt.org/z/veGqnca94>

(**) In containing namespace

Part II: Method 2: Class Template Specialization (Functor) V4

Library

```
3 template <typename T = void>
4 struct Logger {           // Default do something
    void operator()() {
        std::cout << "Default Log\n";
    }
};

template <>
struct Logger<void> {      // Delegation
    template <typename T>
    auto operator()(const T&) -> decltype(auto) {
        std::cout << "Do A, ";
        return Logger<T>{}();
    }
};
```

User

```
// No Implementation required for default

template <>
struct Logger<Spaceship> {   // User op() (LibSpace)
    auto operator()(const Spaceship& s)
};

Spaceship s;
Logger defaultLogger;
Logger<Spaceship> spaceLogger;
defaultLogger(s);           // Do A, Default Log
spaceLogger(s);             // User Implementation
```

- Library side:
 1. Library defines a template class “Logger” with default “op()”
 2. Default implementation also delegates call to user
- User side:
 1. User override implementation by template specialization (or inheritance)
 2. We can call either the “defaultLogger”, or the “spaceLogger”

Inbal Levi - Customization Methods: connecting user and library code - CppCon 2023

21

3. Delegation (default do something): <https://godbolt.org/z/zsPcTnP9r>
4. User gain functionality using **Inheritance**: <https://godbolt.org/z/brKPsb83d>

Part II: Method 2: Class Template Specialization

Library

```
3 template <typename T = void>
4 struct Logger {           // Default do something
5     void operator()() {
6         std::cout << "Default Log\n";
7     }
8 };
9 template <>
10 struct Logger<void> {    // Delegation
11     template <typename T>
12     auto operator()(const T&) -> decltype(auto) {
13         std::cout << "Do A, ";
14         return Logger<T>{}();
15     }
16 };
17 }
```

User

```
// No Implementation required for default
template <>
struct Logger<Spaceship> { // User op()
    auto operator()(const Spaceship& s)
};

Spaceship s;
Logger defaultLogger;
Logger<Spaceship> spaceLogger;
defaultLogger(s);           // Do A, Default Log
spaceLogger(s);            // User Implementation
```

- Characteristics:
 1. Namespace or type intrusive (User adds code into LibSpace or inherits from `Logger<void>`) ↗
 2. Default implementation can be provided ↘
 3. If default exists, the indication of what to opt-in on is weaker ↙
 4. No mechanism for verifying the interface behavior ↙
- Examples: C++ STL (V1), fmtlib (V2)

5

Inbal Levi - Customization Methods: connecting user and library code - CppCon 2023

22

Additional info:

3. Template specialization: Of class in UserSpace fails (<https://godbolt.org/z/Y896ejWxo>),
4. Template specialization: Of class in global space works (<https://godbolt.org/z/6PdebYn1K>) (*)
5. <https://fmt.dev/latest/api.html#formatting-user-defined-types>

Part II: An Overview Of CPs Methods

- Methods:
 1. Inheritance
 2. Class Template Specialization
 - 3. **(Original) Customization Points (ADL)**
 4. Customization Point Objects
 5. Concepts
 6. `tag_invoke` (not in the standard)
 7. Customizable Functions (not in the standard)
 8. Reflection (not in the standard)

Inbal Levi - Customization Methods: connecting user and library code - CppCon 2023

23

7 Slides

Part II: (A Short Detour) Discovery By Overload Resolution

- To compile a function call, compiler creates list of candidates, then finds correct overload
- Overload Resolution:
 - Function candidates picked according to function name (Name Lookup)
 - May include Template Argument Deduction (TAD), Argument Dependent Lookup (ADL)
- Template Argument Deduction (TAD):
 - Function templates with same name create additional candidates
- Argument-Dependent Lookup (ADL):
 - Searching not only in function call namespace, but also in function arguments' namespaces
 - Namespaces are ordered (inner first)
 - Arguments are not ordered (ambiguous if found in more than one argument's namespace)¹

Inbal Levi - Customization Methods: connecting user and library code - CppCon 2023

24

1. Namespaces ordering: <https://godbolt.org/z/qvMcYn1Kd> (if in both the namespace and the argument's namespace)

@@@ SFINAE is introduced by David Vandevoorde, to describe entity's behavior
If unqualified look finds an object, **no ADL happens**

Part II: (A Short Detour) Discovery By Overload Resolution

- **To summarize:**

- To compile a function call, the compiler must first perform [name lookup](#), which may involve [argument-dependent lookup](#), and for function templates may be followed by [template argument deduction](#).
- If these produce more than one *candidate function*, then [overload resolution](#) is performed to select the function that will be called.
- If the result is ambiguous (= multiple candidates with same level of fit), compilation fails.

Part II: Method 3: (Original) Customization Points (Basics)

Library

```
1 template <typename T>
constexpr auto Log(const T& t) { // Log: name
    enable_if<has_name<T>::value> ...
}
template <typename T>
constexpr auto Log(const T& t) { } // Log only
```

User

```
struct Spaceship {
    Name name_;
};
```

```
Spaceship sd("Dragon");           // Usage
Log(sd);                          // Log: Dragon
Log(42);                          // Log only
```

- Library side:
 1. Library defines different versions of “Log” functions (Types with/without a name)
 2. Library rely on functionality of the type assuming proper type will be called
- User side:
 1. User gets different behavior according to properties of the Type
 2. Allows porting types’ behavior from the user (call “name”)

Inbal Levi - Customization Methods: connecting user and library code - CppCon 2023

26

1. <https://godbolt.org/z/xfGcWM76q>

Specialization issue: Function template void vs. auto case:

1. Fails for specializing “auto” template with “void”:

<https://godbolt.org/z/ejv9nKrEq>

Due to breaking the rule: <https://eel.is/c++draft/dcl.spec.auto#general-13> (TAD: <http://eel.is/c++draft/temp.deduct#decl-1>)

2. Works for “void” for both the template and specialization:

<https://godbolt.org/z/7ea68K7rM>

Part II: Method 3: (Original) Customization Points (Basics)

Library

```
2 template <typename T>
constexpr auto Log(const T& t) { // Log: name
    enable_if<has_name<T>::value> ...
}
template <typename T>
constexpr auto Log(const T& t) { } // Log only

template<>
auto Log(Spaceship& t) { } // Log spaceship
```

User

```
struct Spaceship {
    Name name_;
};

void Log(const Spaceship& s) {} // Log spaceship
struct Spaceship {
    friend void Log(Spaceship& s) {} // Log spaceship
};
Spaceship s; // Log Spaceship
```

- As mentioned, overload resolution can be affected by ADL
- It can be affected by: template specialization in enclosing namespace [3]
- Or by template specialization in library space... [4]
- Or by non-template function in user-space... [5]
- Or by non-template function in user-space using friend... [6]
- As you can see, the behavior of these is not simple...
- Let's look at a real-life story: **swap**

Inbal Levi - Customization Methods: connecting user and library code - CppCon 2023

27

3. <https://godbolt.org/z/vTeTEcYd5>
4. <https://godbolt.org/z/hcPEqEx6f>
5. <https://godbolt.org/z/rK1zffh56>
6. <https://godbolt.org/z/xsc5r679W>

(*) “Hidden friends”: inline (header defined) friend function, visible for ADL as if it was on the external scope.

- (*) Namespaces: <https://en.cppreference.com/w/cpp/language/namespace>
Out-of-namespace definitions and redeclarations are only allowed
 - after the point of declaration,
 - at namespace scope, and
 - in namespaces that enclose the original namespace (including the global namespace).

Also, they must use qualified-id syntax. (since C++14)

Part II.5: The “Swap” Fiasco

Inbal Levi - Customization Methods: connecting user and library code - CppCon 2023

28

Part II: Method 3: (Original) CPs - The “Swap” Fiasco

How to overload std::swap()

The right way to overload `std::swap`'s implementation (aka specializing it), is to write it in the same place where `std::swap` is defined.

In C++2003 it's at best underspecified. Most implementations do use ADL to find `swap`, but no it's not guaranteed to work.

I would be surprised to find that implementations *still* don't use ADL to find the correct `swap`. This is an old issue.

Somebody should check the library that shuns with MSVC. I know they were a holdout for a long time on this. I don't think this is a good idea. First of all: Why defining this function at global scope only? There's no reason for that.

@Sascha: First, I'm defining the function at namespace scope because that's the only kind of definition that makes sense. If your standard library implementation uses ADL to find a function called `swap`, then it's guaranteed to work.

If your standard library implementation uses ADL to find a function called `swap`, then it's guaranteed to work.

: Just tested this with gcc, its `std::sort` used a `swap` method defined in this way. Does that mean that in this example VS calls the wrong `swap`? This adds a new wrinkle I hadn't previously considered.

Attention Mozza314

one question. what is the purpose of friend here?

Person S Feb 11 at 11:09

Person B

Person B

Person E

Person B

29

- <https://stackoverflow.com/questions/11562/how-to-overload-stdswap>
- @@@ The right way to overload `swap` is by defining inline friend `swap` function.
- @@@ Using the two-step `swap` call in function templates

Part II: Method 3: (Original) Customization Points (Using ADL)

Library

```
template <typename T>
constexpr auto Log(T& t) {           // Lib Default
    std::cout << "Log Default\n";
}

template <typename T>
void Logger(T& t) {
    Log(t);
}
```

User

```
struct Spaceship {};

void friend Log(Spaceship& s) { // ImpUser
    std::cout << "Log Spaceship\n";
}
};

1
2
3
```

```
Spaceship s;
Log(s);                                // ImpUser
Logger(s);                             // Log Spaceship
```

- Library side:
 1. Library defines a utility - “Log”
 2. Utility can be used by “Logger”
- User side:
 1. Override implementation (to be discovered via ADL) by a free function / friend function (*)
- The “friend” function (or class) is a member of the innermost enclosing namespace [2]
- The term used for (inline) friend functions is “Hidden friends” (hidden from global name lookup)

Inbal Levi - Customization Methods: connecting user and library code - CppCon 2023

30

1. ADL:

1. Free fun: <https://godbolt.org/z/3s8dbe8jE>
2. Friend: <https://godbolt.org/z/374jo55xs>

2. Hidden Friend:

1. <https://www.justsoftwaresolutions.co.uk/cplusplus/hidden-friends.html>
2. <https://stackoverflow.com/questions/56795676/hidden-friends-declarations-and-definitions#:~:text=Hidden%20friends%20need%20to%20be,of%20the%20definition%20of%20class.>

3. Example: Operator <<: <https://godbolt.org/z/1vdPYn71K>

Part II: Method 3: (Original) Customization Points (Using ADL)

Library

```
1 template <typename T>
constexpr auto Log(T& t) {           // Log Default
    std::cout << "Log Default\n";
}

template <typename T>
void Logger(T& t) {
    Log(t);
}
```

User

```
struct Spaceship {

    friend void Log(Spaceship& s) { // ImpUser
        std::cout << "Log Spaceship\n";
    }
};
```

```
Spaceship s;
Log(s);                                // ImpUser
Logger(s);                             // Log Spaceship
```

- Characteristics:

1. Porting functionality from “Spaceship” into the LibSpace ↗
2. Non-intrusive into LibSpace, intrusive into UserType ☺
3. “Trusting” overload resolution to do the right thing counting on ADL adds complexity ↘
4. Since the mechanism is based on names, names have to be reserved globally ↙

- Examples: operator<<, std::swap, std::begin, std::end

Part II: Method 3: (Original) Customization Points (Using ADL)

- “Suggested Design for Customization Points” (by Eric Niebler) suggested the following:
 - Calls within a library should be Unqualified (to add the std::swap option to the candidates)

```
template <typename T>
void Sort(T& a, T& b) {
    std::swap(a, b);      // Wrong
}
```

```
template <typename T>
void Sort(T& a, T& b) {
    using std::swap;
    swap(a, b);          // Right
}
```
 - Users can do so as well (where std have reasonable fallback), but this will bypass any constraints

```
using std::swap; // Brings std::swap into the scope
swap(c,d);      // Unqualified call - uses ADL and ignores std
```
- To solve potential issues mentioned, Eric Niebler presented: “Customization Point Objects”

Inbal Levi - Customization Methods: connecting user and library code - CppCon 2023

32

- @@@ Pitfall – usually we can't recognize it by behavior

Part II: An Overview Of CPs Methods

- Methods:
 1. Inheritance
 2. Class Template Specialization
 3. (Original) Customization Points (ADL)
 - 4. **Customization Point Objects**
 5. Concepts
 6. `tag_invoke` (not in the standard)
 7. Customizable Functions (not in the standard)
 8. Reflection (not in the standard)

Inbal Levi - Customization Methods: connecting user and library code - CppCon 2023

33

2 Slides

Part II: Method 4: Customization Point Objects

Library

```
1 namespace __detail {
    template <typename T>
    constexpr auto Log(T& t) {...} // Log Default

    struct Log_fn { // Log Obj Imp
        template<class T>
        auto operator()(T& t1) const {
            Log(std::forward<T>(t1));
        }
    };
    // Create a global Log_fn ...
}
```

User

```
struct Spaceship {};

void Log(Spaceship& s1) {
    std::cout << "Log Spaceship\n";
}

using LibSpace::Log;
using UserSpace::Log;
Spaceship s;
Log(s); // Log Spaceship
```

- Library side:
 1. Library defines a template function “Log”
 2. Library creates a “Log_fn” functor with template “operator()”
 3. Library creates a global “Log_fn” (with some magic to avoid ODR violation)
- User side:
 1. User creates a “Log” accepting “Spaceship”

Inbal Levi - Customization Methods: connecting user and library code - CppCon 2023

34

1. Swap CPOs simplified implementation: <https://godbolt.org/z/sEj8TsYaj> (based on example from Eric Niebler’s blog: <http://ericniebler.com/2014/10/21/customization-point-design-in-c11-and-beyond/>)
2. CPOs directly in LibSpace (without __detail): <https://godbolt.org/z/4KocG93cP>
 1. Will not work unqualified because the call would be ambiguous
 2. Algorithm LibFunc will (correctly) trigger ADL lookup

Part II: Method 4: Customization Point Objects

Library

```
1 namespace __detail {
    template <typename T>
    constexpr auto Log(T& t) {...} // Log Default

    struct Log_fn { // Log Obj Imp
        template<class T>
        auto operator()(T&& t1) const {
            Log(std::forward<T>(t1));
        }
    };
    // Create a global Log_fn ...
}
```

User

```
struct Spaceship {};

void Log(Spaceship& s1) {
    std::cout << "Log Spaceship\n";
}

using LibSpace::Log;
using UserSpace::Log;
Spaceship s;
Log(s); // Log Spaceship
```

- Characteristics:
 1. Porting functionality for UserType into the LibSpace ↗
 2. Non-intrusive into LibSpace, non intrusive into UserType ↗ ↗ ↗
 3. Since the mechanism is based on names, names have to be reserved globally ↗
 4. Complex mechanism to avoid ODR violation ↗
- Examples: ranges::swap, ranges::begin, ranges::end, ranges...

Inbal Levi - Customization Methods: connecting user and library code - CppCon 2023

35

1. Swap CPOs simplified implementation: <https://godbolt.org/z/sEj8TsYaj> (based on example from Eric Niebler's blog: <http://ericniebler.com/2014/10/21/customization-point-design-in-c11-and-beyond/>)
2. CPOs directly in LibSpace (without __detail): <https://godbolt.org/z/4KocG93cP>
 1. Will not work unqualified because the call would be ambiguous
 2. Algorithm LibFunc will (correctly) trigger ADL lookup

Part II: An Overview Of CPs Methods

- Methods:
 1. Inheritance
 2. Class Template Specialization
 3. (Original) Customization Points (ADL)
 4. Customization Point Objects
- 5. Concepts
 6. tag_invoke (not in the standard)
 7. Customizable Functions (not in the standard)
 8. Reflection (not in the standard)

Inbal Levi - Customization Methods: connecting user and library code - CppCon 2023

36

1 Slides

Part II: Method 5: Concepts

Library

```
template<typename T>
concept LoggedType = requires {
    typename T::logged_type_tag;
};

template <LoggedType T>
constexpr auto Log(T& t) { } // Define Init

template <typename T> void Logger(const T& t) { // Constraints
    Log(t); // Constraints
}
```

User

```
struct Spaceship {
    using logged_type_tag = void;
};

using LibSpace::LibFunc;
Spaceship s;
Log(s); // Usage
Logger(s);
```

- Characteristics:
 1. Clear, expressive code 🎉 🎉 🎉
 2. Namespace or type intrusive 🙅
 3. Need to reserve names globally 🙅
 4. Library needs to carefully apply the constraints to avoid collision 🙅
 5. Concepts are great for syntactic conformance 🎉
 6. Using tags to achieve semantic (meaning) conformance 😊C++OX Concepts could maybe offer a wider solution, but are out of scope for this talk
- Examples: Ranges library

Inbal Levi - Customization Methods: connecting user and library code - CppCon 2023

37

1. C++20 Concepts allow syntactic (structural) conformance but not semantic conformance

1. (Nominal concepts in C++20)

Part II: An Overview Of CPs Methods

- Methods:
 1. Inheritance
 2. Class Template Specialization
 3. (Original) Customization Points (ADL)
 4. Customization Point Objects
 5. Concepts
 - 6. **tag_invoke (not in the standard)**
 7. Customizable Functions (not in the standard)
 8. Reflection (not in the standard)

3 Slides

Part II: Method 6: tag_invoke (Basics)

Library

```
1 namespace LibSpace {
    BFG_TAG_INVOKE_DEF(log_tag);           // Macro
} // LibSpace

template <typename FT>
float Log(const FT& c, float x, float y) {
    return LibSpace::log_tag(c, x, y);
}

// Defines a static struct and a reference
#define BFG_TAG_INVOKE_DEF(Tag) \
    static struct Tag##_t final : ::bfg::tag<Tag##_t>\n    {}\\
    const &Tag = ::bfg::tag_invoke_v(Tag##_t {})
```

User

```
struct LogVelocity {
    inline friend float
    tag_invoke(LibSpace::log_tag_t,
               const LogVelocity&, float x, float y) { }
};

struct LogFuel {
    inline friend float
    tag_invoke(LibSpace::log_tag_t,
               const LogFuel&, float x, float y) { }
};

Log(LogVelocity{}, 4, 2);                // 500 m/s
Log(LogFuel{}, 4, 2);                   // 13 ton
```

- Mechanism proposed in P1895 by Lewis Baker, Eric Niebler, Kirk Shoop (targeted C++23)
- Example is based on duck_invoke implementation (by René Ferdinand Rivera Morell)
- There are a few others, this one is the simplest one I could find...

39

Inbal Levi - Customization Methods: connecting user and library code - CppCon 2023

1. Simplest implementation I could find: Implementation using duck_invoke:
<https://godbolt.org/z/j8o9bGs1z>
2. Eric Niebler's simple tag_invoke:
<https://gist.github.com/ericniebler/056f5459cf259da526d9ea2279c386bb>
1. @@@ Did not make it into the standard, work was held back by the consideration of the Function CPs

@@@ All of library is same namespace, user code without frame is user space

Part II: Method 6: tag_invoke (Basics)

Library

```
1 namespace LibSpace {
BFG_TAG_INVOKE_DEF(log_tag);           // Macro!
} // LibSpace

template <typename FT>
float Log(FT& c) float x, float y) {
    return LibSpace::log_tag(c, x, y);
}

// Defines a static struct and a reference
#define BFG_TAG_INVOKE_DEF(tag) ::bfg::tag<log_tag_t>
{} static struct Tag## t final : ::bfg::tag<Tag## t>
{const log_tag = ::bfg::tag_invoke_v(log_tag_t {})}
const &Tag = ::bfg::tag_invoke_v(Tag## t {})
```

User

```
struct LogVelocity {
    inline friend float
    tag_invoke(LibSpace::log_tag_t,
               const LogVelocity&, float x, float y) {...}
};

struct LogFuel {
    inline friend float
    tag_invoke(LibSpace::log_tag_t,
               const LogFuel&, float x, float y) {...}
};

Log(LogVelocity{}, 4, 2); // 500 m/s
Log(LogFuel{}, 4, 2); // 13 ton
```

- Library Side:

- Library defines a `log_tag` object (using the macro)
- Library defines `Log` function, delegates to `log_tag`

- User Side:

- User defines functionality using the reserved word “`tag_invoke`” with first param `log_tag_t`

(*) We can add things such as delegation of the return type

Inbal Levi - Customization Methods: connecting user and library code - CppCon 2023

Part II: Method 6: tag_invoke (Basics)

Library

```
1 namespace LibSpace {
    BFG_TAG_INVOKE_DEF(log_tag);           // Macro!
} // LibSpace

template <typename FT>
float Log(const FT& c, float x, float y) {
    return LibSpace::log_tag(c, x, y);
}

// Defines a static struct and a reference
static struct log_tag_t final : ::bfg::tag<log_tag_t>
{};
const &log_tag = ::bfg::tag_invoke_v(log_tag_t {})
```

User

```
struct LogVelocity {
    inline friend float
    tag_invoke(LibSpace::log_tag_t,
               const LogVelocity&, float x, float y) {...}
};

struct LogFuel {
    inline friend float
    tag_invoke(LibSpace::log_tag_t,
               const LogFuel&, float x, float y) {...}
};

Log(LogVelocity{}, 4, 2);                // 500 m/s
Log(LogFuel{}, 4, 2);                   // 13 ton
```

- Characteristics:
 - tag_invoke solves almost all of the previous CPs' limitations (global names, intrusiveness, ...)
 - A complex mechanism
 - Communication with the user is limited, diagnostics are complicated
 - Compile time performance is bad
- Examples: libunifex, and in earlier versions of std::execution

Inbal Levi - Customization Methods: connecting user and library code - CppCon 2023

41

2. Improvement of usage: <https://godbolt.org/z/qnnEbf3qo>

@@@ Now we can move implementations across namespaces, in a non-intrusive way

Part II: An Overview Of CPs Methods

- Methods:
 1. Inheritance
 2. Class Template Specialization
 3. (Original) Customization Points (ADL)
 4. Customization Point Objects
 5. Concepts
 6. tag_invoke (not in the standard)
 - 7. **Customizable Functions (not in the standard)**
 8. Reflection (not in the standard)

Part II: Method 7: Customizable Functions

Library

```
template <typename T>
virtual auto LogVelocity(T& t) = 0;           // No Imp

template <typename T>
virtual auto LogFuel(T& t) default { // Def Imp
    // ...
}
```

User

```
struct Spaceship {                         // User Imp
    template <typename T>
    friend void LibSpace::LogVelocity(T& t)
override
    {}
Spaceship s;                                // Usage
LogFuel(s);
```

- Mechanism proposed in 2018 in the paper: “P1292: Customization Point Functions (Matt Calabrese)”
- Reproposed in 2022 in P2547 by Lewis Baker, Corentin Jabot, Gašper Ažman (targeting C++26)
- Characteristics:
 1. User can add implementation in UserSpace (as hidden friend) ↗
 2. No need to reserve names globally ↗
 3. Opt-in explicitly, inability to incorrectly opt-in ↗
 4. Not in the language yet 😞
- Examples: The future imagined “P2300: std::execution”

Inbal Levi - Customization Methods: connecting user and library code - CppCon 2023

43

@@@ All templates can be constrained with concepts

Part II: An Overview Of CPs Methods

- Methods:
 1. Inheritance
 2. Class Template Specialization
 3. (Original) Customization Points (ADL)
 4. Customization Point Objects
 5. Concepts
 6. `tag_invoke` (not in the standard)
 7. Customizable Functions (not in the standard)
 - 8. **Reflection (not in the standard)**

Inbal Levi - Customization Methods: connecting user and library code - CppCon 2023

44

1 Slides

Part II: Method 8: Reflection

Library

```
template<class_type T, structural_subtype_of<T> U>
void Log(const T& src, U& dst){
    constexpr auto members =
        meta::data_members_of(reflexpr(src));
    template for (constexpr meta::info a : members){
        constexpr meta::info b = meta::lookup(dst,
            meta::name_of(a));
        dst.[b] = src.[a];
    }
} // `structural_copy` From P2237R0
```

User

```
struct Spaceship {
    // Implementation conforming to the
    // requirements
}
```

```
Spaceship s;
Log(s); // Reflection Logger
```

- Characteristics:

1. Opt-in explicitly, limitations on incorrectly opt-in 😕
2. Introspective is limited to compile time capabilities 🚧
3. Algorithm is replaced with meta code. Implementation burden increases on library side 🤦
4. And decreases on user side(?) 😊
5. Reflection is a powerful tool 😎 😎 😎

Can reflection provide “The Ultimate Customization Point”?

Inbal Levi - Customization Methods: connecting user and library code - CppCon 2023

45

1. @@@ There's experimental implementations such as by Matúš Chochlík ([CompilerExplorer](#))
1. Reflection experimental: <https://compiler-explorer.com/z/shMz3of5P>
2. <https://compiler-explorer.com/z/TrYEYhqMK>
3. <https://github.com/matus-chochlik/llvm-project>

Part III: Comparing CPs Methods

46

Inbal Levi - Customization Methods: connecting user and library code - CppCon 2023

2 Slides

Part III: Comparing Customization Points Methods

		Inheritance	CTS	CPs (ADL)	CPOs	Concepts (+nominal)	Deducing This	tag_invoke	Custom functions	Reflection
Integrat	Share functionality	As group	As group	Yes	Yes	Yes	Yes	Yes	Yes	As group
	Share data, terminology	As group	As group	No	Some	No	Yes	No	No	As group
	Opt-in explicitly	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes	Some
Comm	Communicate interface	Yes	No	Some	Some	Yes	Some	Yes	Yes	Some
	Default implementation	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
	Inability to opt-in wrongly	Some	No	No	No	Some	Some	No	Some	Some
Special	Verify type conformance	Yes	Some	No	Yes	Yes	Some	Yes	Yes	Some
	Not type intrusive	No	Some	Yes	Yes	No	No	Yes	Yes	Yes
	Associated types	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Impl	Don't reserve names globally	Yes	No	No	Yes	No	Yes	Yes	Yes	Yes
	Done at compile time	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes

Inbal Levi - Customization Methods: connecting user and library code - CppCon 2023

47

Properties:

1. What interface needs to be customized
2. Provide default implementations
3. Opt-in explicitly
4. Inability to incorrectly opt-in
5. Invoke the customized implementation
6. Verify a type implements an interface
7. Atomic group of functionality
8. Flexibility in extending the interface
9. Clear indication of narrowing the interface
10. Ability to share data with the user
11. Ability to share functionality with the user

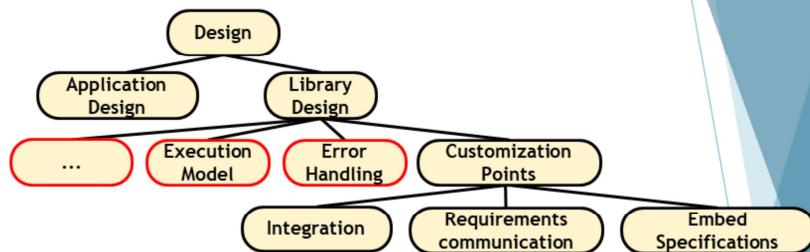
Part IV: What's Next?

Inbal Levi - Customization Methods: connecting user and library code - CppCon 2023

48

1 Slides

Part IV: Additional Aspects In Library Design



Additional aspects to consider:

1. Execution error handling strategy (in oppose to integration errors)
2. Defaults
3. "Global" effects - capturing names, overload resolution, ADL, etc.
4. Library may co-exist with other Libraries
5. Runtime flow - dynamically decided types and functionality
6. Execution model (multi-thread, Async, etc.)
7. Documentation, development process, tools, bug reports, ...

Core C++ 2023: Development Strategies - The stuff around the code (Marshall Clow)

Inbal Levi - Customization Methods: connecting user and library code - CppCon 2023

Part IV: What's Next?

- P2300: std::execution did not make it into C++23 (was forwarded to C++26)
 - (Possible) change in customization mechanism was one of the major setbacks
 - tag_invoke was considered into C++23, but was eventually dropped, due to...
 - “P2547: Language support for customisable functions” proposed alternative. BUT...
 - The need for language-based CPs may (?) be solved by reflection
 - Reflection proposals are still in motion, it’s unclear which usability we can expect
- I would hope that we’ll be able to get solutions, unfolded from last, into C++29 😊
 - Meanwhile, we’ll have to settle with non language solutions...

Thanks!

Thank you for listening 😊

Special thanks to:

- **CoreC++ user group**
- Gašper Ažman
- Yehezkel Brant
- Kilian Henneberger

- [Linkedin.com/inballevi](https://www.linkedin.com/inballevi)
- [Twitter.com/lnbal_l](https://twitter.com/lnbal_l)
- sinbal2lextra@gmail.com

Would love to get your input!

References

- The Talk Is Based On:
 - [P2279](#): We need a language mechanism for customization points (Barry Revzin)
 - [Talk](#): 'tag_invoke' - An actually good way to do customization points (Gašper Ažman)
 - [N4381](#): Suggested Design for Customization Points (Eric Niebler)
- Related Papers:
 - [P1292](#): Customization Point Functions (Matt Calabrese)
 - [P2547](#): Language support for customisable functions (Lewis Baker, Corentin Jabot, Gašper Ažman)
 - [P1895](#): tag_invoke: A general pattern for supporting customizable functions (Lewis Baker, Eric Niebler, Kirk Shoop)
 - [P1240](#): Scalable Reflection in C++ (Wyatt Childers, Andrew Sutton, Faisal Vali, Daveed Vandevoorde)
 - [P2237](#): Metaprogramming (Andrew Sutton)
 - [P0707](#): Metaclasses: Generative C++ (Herb Sutter)
 - [P2560](#): Comparing value- and type-based reflection (Matúš Chochlík)
 - [P2300](#): std::execution (Dominik, Baker, Howes, Shoop, Garland, Niebler, Adelstein Lelbach)
 - [P2098](#): Proposing std::is_specialization_of (Walter E. Brown, Bob Steagall)
 - [N1758](#): Concepts for C++0x (Siek, Gregor, Garcia, Willcock, Jarvi, Lumsdaine)

Inbal Levi - Customization Methods: connecting user and library code - CppCon 2023

52

- More Related Info:
 - Jonathan Muller: C++20 concepts are structural: What, why, and how to change it? (<https://www.foonathan.net/2021/07/concepts-structural-nominal/>)
 - Howard Hinnant: Engineering Distinguished Speaker Series (<https://www.youtube.com/watch?v=vLinb2fgkHk>)
 - Arthur O'Dwyer: Customization point design for library functions (<https://quuxplusone.github.io/blog/2018/03/19/customization-points-for-functions/>)
 - Stack overflow: what does it mean when something is SFINAE-friendly? (<https://stackoverflow.com/questions/35033306/what-does-it-mean-when-one-says-something-is-sfinae-friendly>)
 - Specializations and namespaces: <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3730.html>
 - Inline namespaces: <https://www.ibm.com/docs/en/zos/2.4.0?topic=only-inline-namespace-definitions-c11>
 - Placeholder type in template specialization: <https://eel.is/c++draft/dcl.spec.auto#general-13>

References

- More Sources:
 - [Talk: Development Strategies - The stuff around the code \(Marshall Clow\)](#)
 - [Talk: Deducing this Patterns \(Ben Deane\)](#)
 - [Blog: Customization Point Design in C++11 and Beyond \(Eric Niebler\)](#)
 - [Blog: Niebloids and Customization Point Objects \(Barry Revzin\)](#)
 - [Blog: Generic programming blog post \(Boost\)](#)
 - [Blog: C++20 concepts are structural: What, why, and how to change it? \(Jonathan Muller\)](#)
 - [Libunifex: Sender/receiver async library](#)
 - [value_from_tag: Boost lib tag_invoke implementation](#)
 - [Duck_invoke: Single header simple tag_invoke implementation](#)
 - [Simple tag_invoke: By Eric Niebler](#)
 - [CompilerExplorer: Reflection experimental \(Matúš Chochlík\)](#)
 - [Github: Reflection experimental \(Matúš Chochlík\)](#)
 - [Blog: C++ Libraries Part I & II \(Inbal Levi\) 😊](#)

(Simplified) Design of Customization Point Objects

```
1 namespace UserSpace {
    struct UserStruct {
        UserStruct(int a)
        : a_(a) {
            std::cout << "UserStruct CTOR a: " << a_ << "\n";
        }
        int a_;
    };
    void swap(UserStruct& first, UserStruct& second) {
        std::cout << "UserStruct global Swap\n";
    }
} // UserSpace
// In this scope, we have std::swap
int main()
{
    using std::swap;           // To bring std::swap
    int a = 1;
    int b = 2;
    std::cout << "a: " << a << " b: " << b << "\n";
    swap(a,b);
    std::cout << "a: " << a << " b: " << b << "\n";
    UserSpace::UserStruct c = 1;
    UserSpace::UserStruct d = 2;
    std::cout << "c: " << c.a_ << " d: " << d.a_ << "\n";
    swap(c,d);
    std::cout << "c: " << c.a_ << " d: " << d.a_ << "\n";
}
```



```
2 namespace UserSpace {
    struct UserStruct {
        UserStruct(int a)
        : a_(a) {
            std::cout << "UserStruct CTOR a: " << a_ << "\n";
        }
        int a_;
    };
    template<typename T>
    struct swap {
        swap() {
            std::cout << "UserStruct swap CTOR\n";
        }
        void operator()(UserSpace::UserStruct &s1, UserSpace::UserStruct& s2) const {
            std::cout << "UserStruct swap CPO\n";
        }
    };
} // UserSpace
int main()
{
    using std::swap;
    int a = 1;
    int b = 2;
    std::cout << "a: " << a << " b: " << b << "\n";
    swap(a,b);
    std::cout << "a: " << a << " b: " << b << "\n";
    UserSpace::UserStruct c = 1;
    UserSpace::UserStruct d = 2;
    std::cout << "c: " << c.a_ << " d: " << d.a_ << "\n";
    UserSpace::swap(c, d);
    std::cout << "c: " << c.a_ << " d: " << d.a_ << "\n";
}
```

Inbal Levi - Customization Methods: connecting user and library code - CppCon 2023

NOTICE: This is slides code, not production level code

1. Swap free function: <https://godbolt.org/z/KchvT6x4j>
2. Swap object without names reservation bypass: <https://godbolt.org/z/exTqo3xKh>

CPOs Full Example - begin

```
1 #include <utility>
2 #include <iostream>
3
4 namespace new_std
5 {
6     namespace __detail
7     {
8         // @@@@ Define "begin"s free functions @@@@
9         // For arrays
10        template<class T, size_t N>
11        constexpr T* begin(T (&a)[N]) noexcept
12        {
13            return a;
14        }
15
16        // For containers
17        // (trailing return type needed for SFINAE)
18        template<class _Rangelike>
19        constexpr auto begin(_Rangelike&& rng) ->
20            decltype(std::forward<_Rangelike>(rng).begin())
21        {
22            return std::forward<_Rangelike>(rng).begin();
23        }
24
25        // @@@@ begin functor @@@@
26        // Class __begin_fn, with templated operator()(R&& r)
27        struct __begin_fn
28        {
29            template<class R>
30            constexpr auto operator()(R && rng) const ->
31                decltype(begin(std::forward<R>(rng)))
32            {
33                return begin(std::forward<R>(rng)); // Unqualified Function call
34            }
35        };
36    } // __detail
37
38    // @@@@ To avoid ODR violations: @@@@
39    // 1. (Boilerplate) Define struct (wrapper), which contains static constexpr T value
40    template<class T>
41    struct __static_const
42    {
43        static constexpr T value{};
44    };
45
46    // 2. (Boilerplate) Declare & Define global instance of the above struct, which
47    // contains static constexpr T value
48    template<class T>
49    constexpr T __static_const<T>::value;
50
51    // std::begin is a global function object!
52    // (defining in anonymous namespace is equivalent to defining in global namespace)
53    // 3. Define "begin" global object
54    namespace
55    {
56        constexpr auto const & begin =
57            __static_const::__detail::__begin_fn::value;
58    }
59
60    int main()
61    {
62        using namespace new_std;
63        int arr[3] = {1,2,3};
64        std::cout << begin(arr) << "\n";
65    }
66}
```

Inbal Levi - Customization Methods: connecting user and library code - CppCon 2023

NOTICE: This is slides code, not production level code

1. Full “begin” CPO example (including return types):
<https://godbolt.org/z/1d53zsbdG>

CPOs Full Example - swap

```
1 #include <utility>
#include <iostream>

namespace new_std
{
    namespace __detail
    {
        // @@@@ Define "swap"s free functions @@@@
        // For T&'s
        template<class T>
        constexpr void swap(T &t1, T&t2) noexcept
        {
            std::cout << "Swap generic\n";
        }

        // For ints (specialization of the swap)
        void swap(int a, int b)
        {
            std::cout << "Swap int\n";
        }

        // @@@@ swap functor @@@@
        // Class __swap_fn, with templated operator()(R&& r) -->
        swap(forward<T>(r))
        struct __swap_fn
        {
            template<class T>
            constexpr auto operator()(T& t1, T&& t2) const
            {
                std::cout << "Swap functor\n";      // Unqualified Function call
                swap(std::forward<T>(t1), std::forward<T>(t2));
            }
        };
    } // __detail

        // @@@@ To avoid ODR violations: @@@@
        // 1. (Boilerplate) Define struct (wrapper), which contains static constexpr T value
        template<class T>
        struct __static_const
        {
            static constexpr T value{};
        };
        // 2. (Boilerplate) Declare & Define global instance of the above struct, which
        // contains static constexpr T value
        template<class T>
        constexpr T __static_const<T>::value;

        // std::swap is a global function object!
        // (defining in anonymous namespace is equivalent to defining in global namespace)
        // 3. Define "swap" global object
        namespace
        {
            constexpr auto const & swap =
                __static_const<__detail::__swap_fn>::value;
        }

int main()
{
    using namespace new_std;
    double c = 1;
    double d = 2;
    int a = 1;
    int b = 2;
    swap(a, b);
    swap(c, d);
}
```

Inbal Levi - Customization Methods: connecting user and library code - CppCon 2023

NOTICE: This is slides code, not production level code

1. Full “swap” CPO example: <https://godbolt.org/z/6j6W61P19>

Part II: Method 2: Class Template Specialization (Functor)

Library

```
3 template <typename T = void>
struct Logger; // Empty Default
```



```
template <>
struct Logger<void> { // Delegation only
    template <typename T>
    auto operator()(const T&) -> decltype(auto) {
        return Logger<T>{}();
    }
};
```

User

```
template <>
struct Logger<Spaceship> { // User op() (LibSpace)
    auto operator()(const Spaceship& s)
}
Spaceship s; // Usage
Logger<Spaceship> LogSpaceships; // Lib+User
LogSpaceships(s);
```

- Library side:
 1. Library implement function template (for example “op()”) with default and delegation
 2. Default can either do nothing (this slide) or do something (next slide)
- User side:
 1. User override implementation by template specialization (or inheritance)
 2. The specialized Logger can now get UserType, and the User defined operator() will be called

Inbal Levi - Customization Methods: connecting user and library code - CppCon 2023

57

Additional example for CTS (with a functor)

3. Delegation (**more examples are in Gasper's talk**) (default do nothing):

<https://godbolt.org/z/TM67coxvn>

Paper talking about namespaces specialization (was not voted): <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3730.html>

Part II: Additional data

- Few principles from Eric Niebler's "[N4381: Suggested Design for Customization Points](#)":
 1. Qualified and unqualified calls should behave identically
 2. Unqualified calls should not bypass constraints checking
 3. Calls to the customization point should be optimally efficient
 4. No violations of the one-definition rule
- Issues with Niebloids:
 - <https://thephd.dev/a-weakness-in-the-niebloids>