Customization Methods:
Connecting User And Library Code

Inbal Levi

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.

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

Part 0: What is a Library

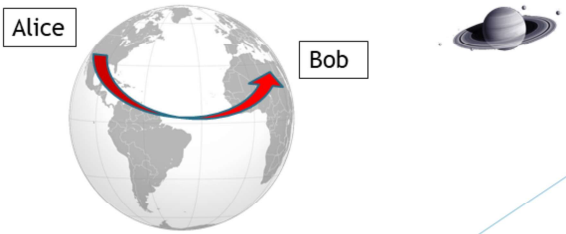
Library

```
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]
```



@@@ 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

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

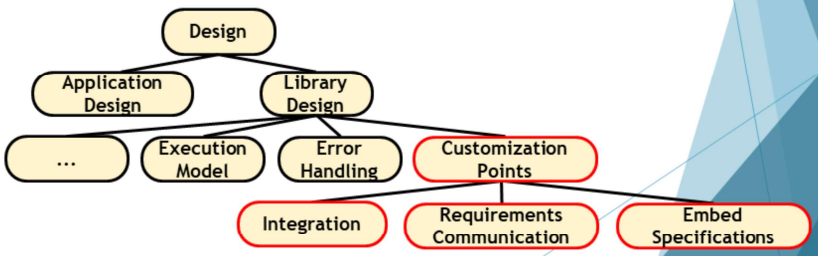
Part I: What Are Customization Points

- **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

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)

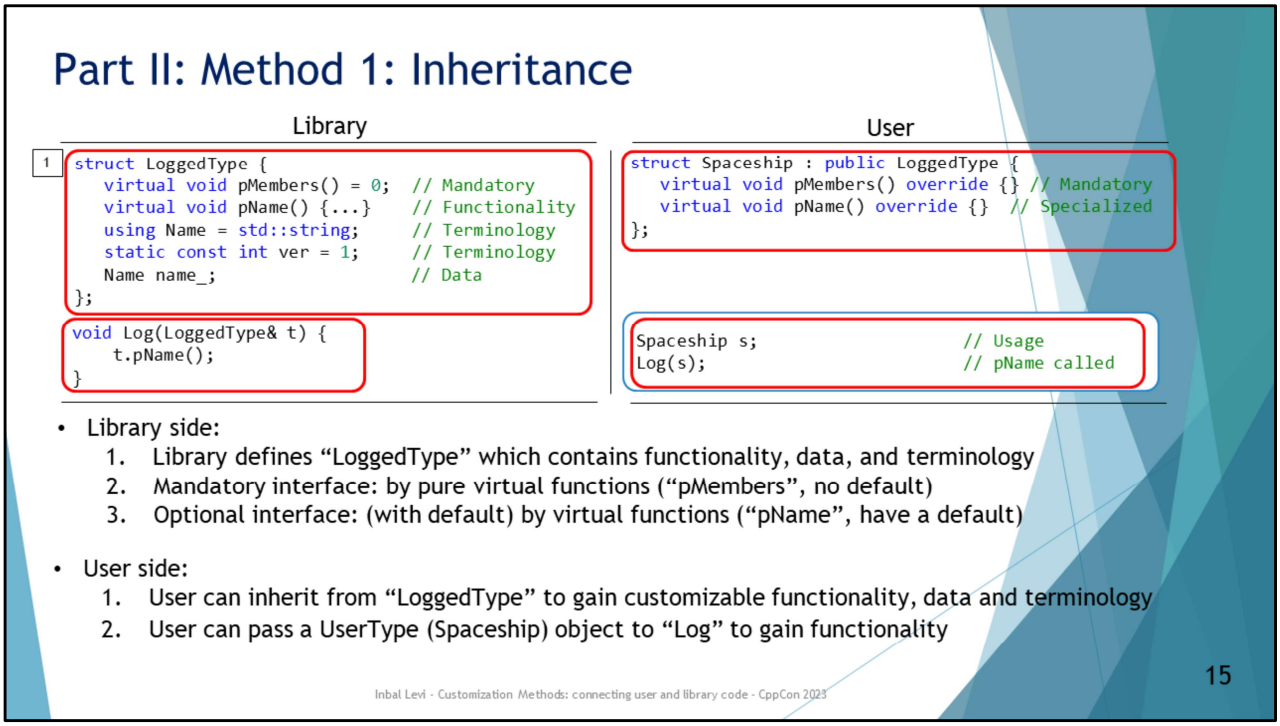
+ @@@ Deducing this removes the need for CRTP

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)

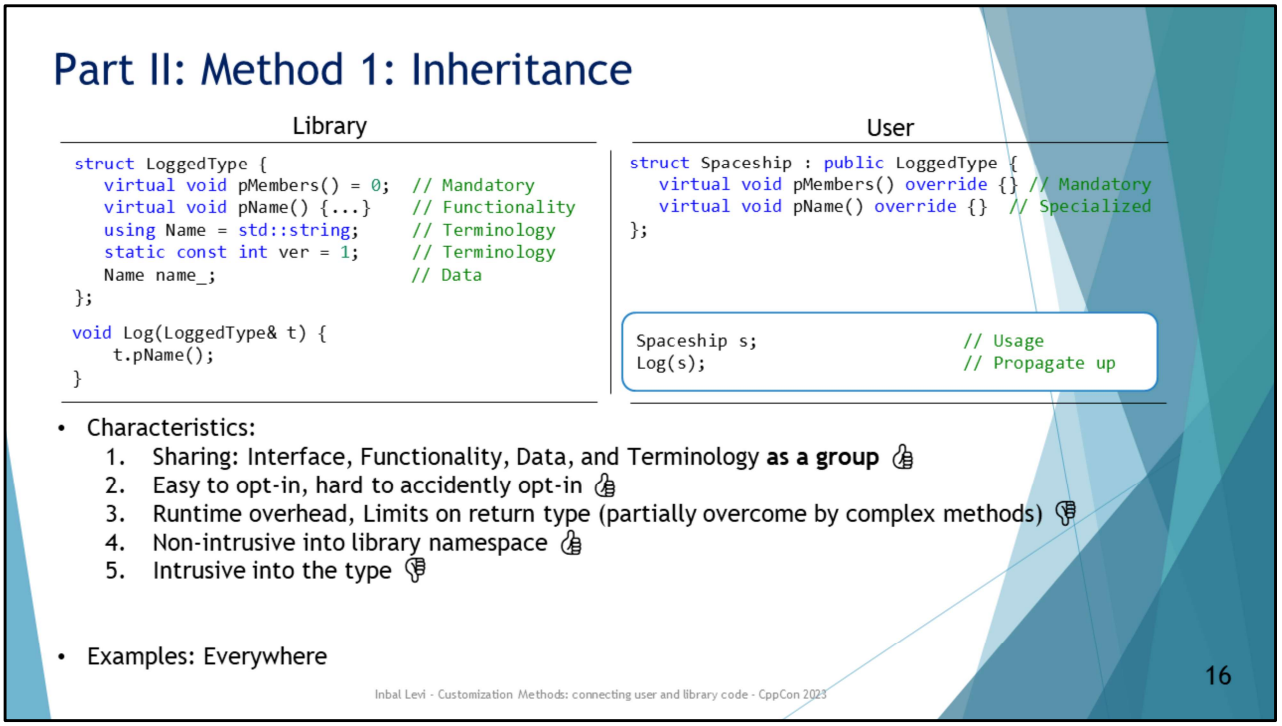
2 Slides



1. https://godbolt.org/z/eTznPrno5

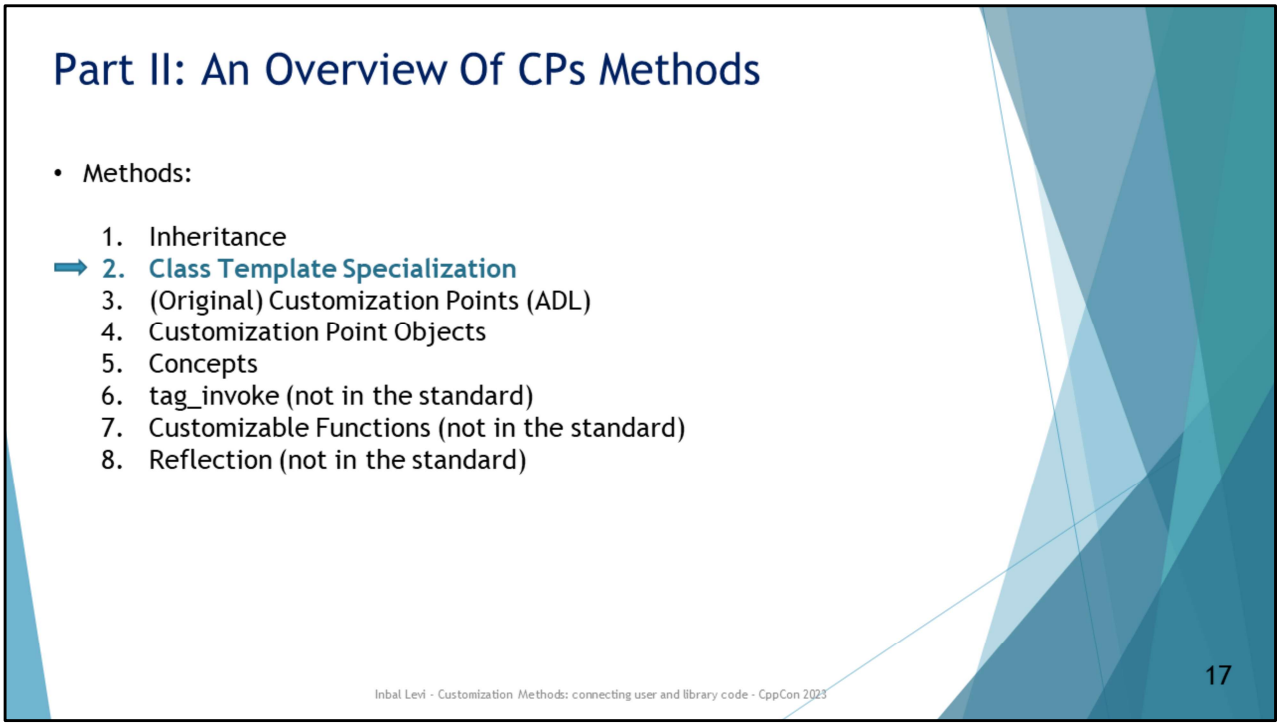
@@@ Use default if there's motivation for default implementation (something which can be done generally)

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



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

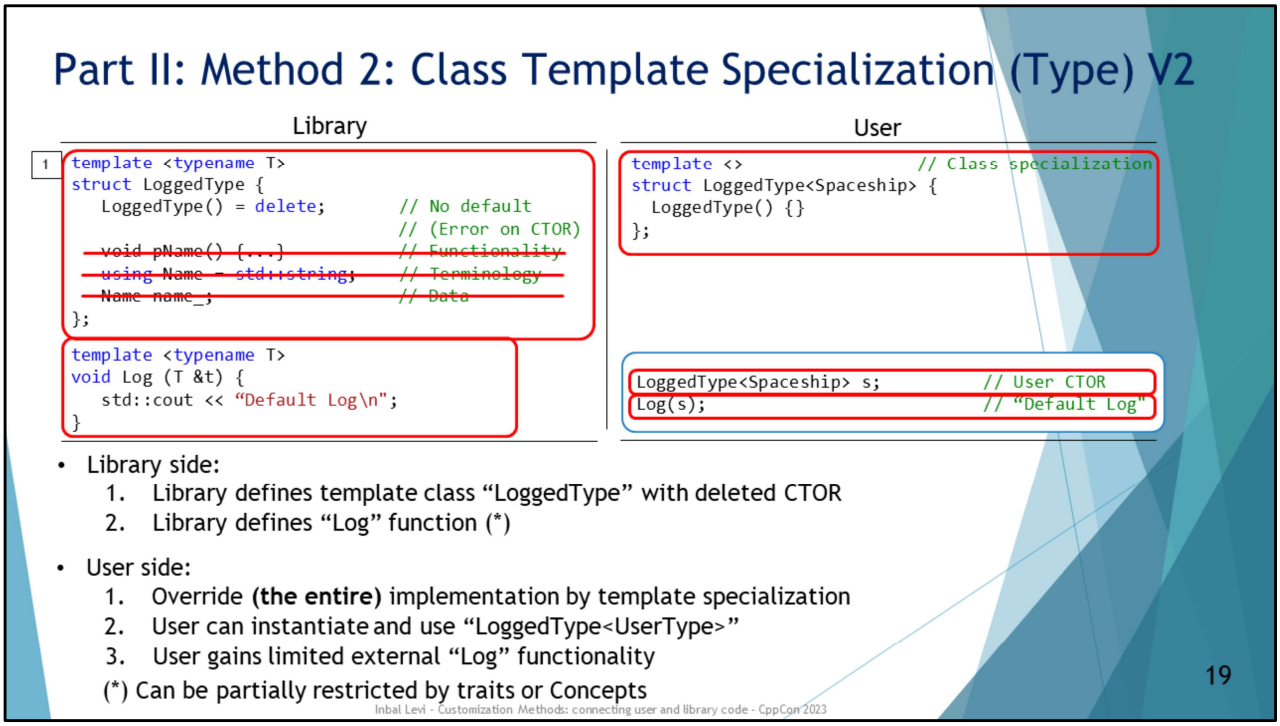
```
struct Spaceship {  // (at least) Erasable (C++17)
   ...
};
```

```
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)

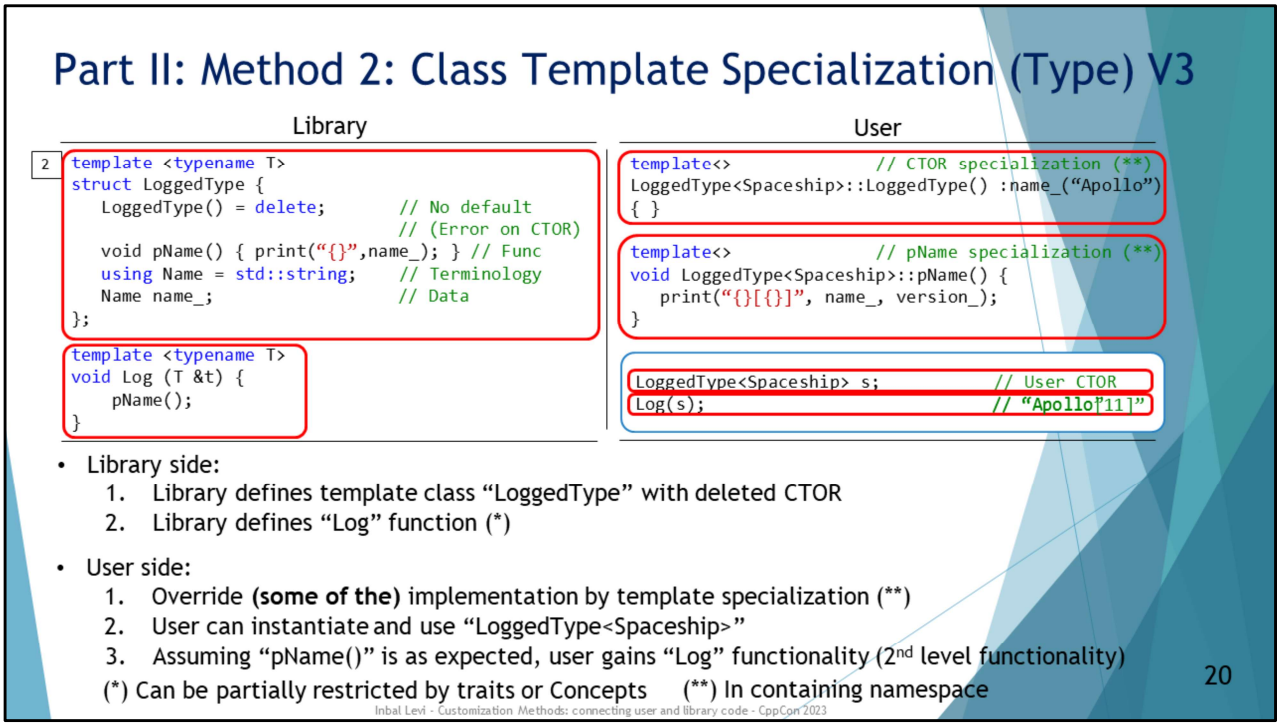
(*) Can be partially restricted by type traits or Concepts

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



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

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



@@@ 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(42);              // 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”

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
       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()
   auto operator()(const Spaceship& s)
   {}
};
```

```
Spaceship s;
Logger defaultLogger;
Logger<Spaceship> spaceLogger;
defaultLogger(42);            // 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]

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)

7 Slides

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

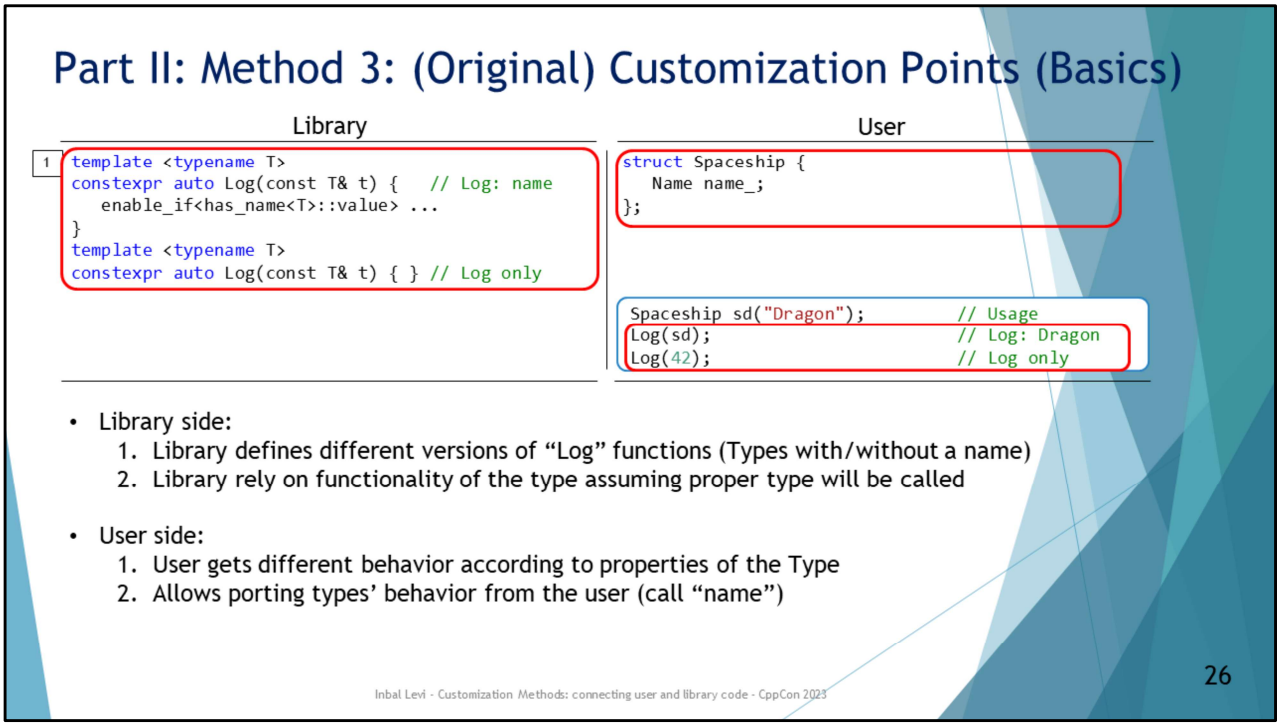
- **To compile a function call, compiler creates list of candidates, then finds correct overload**
- **Overload Resolution:**
  - Function candidates picked according to function name (Name Lookup)
  - **May** include Template Argument Deduction (TAD), Argument Dependent Lookup (ADL)
- **Template Argument 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]

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

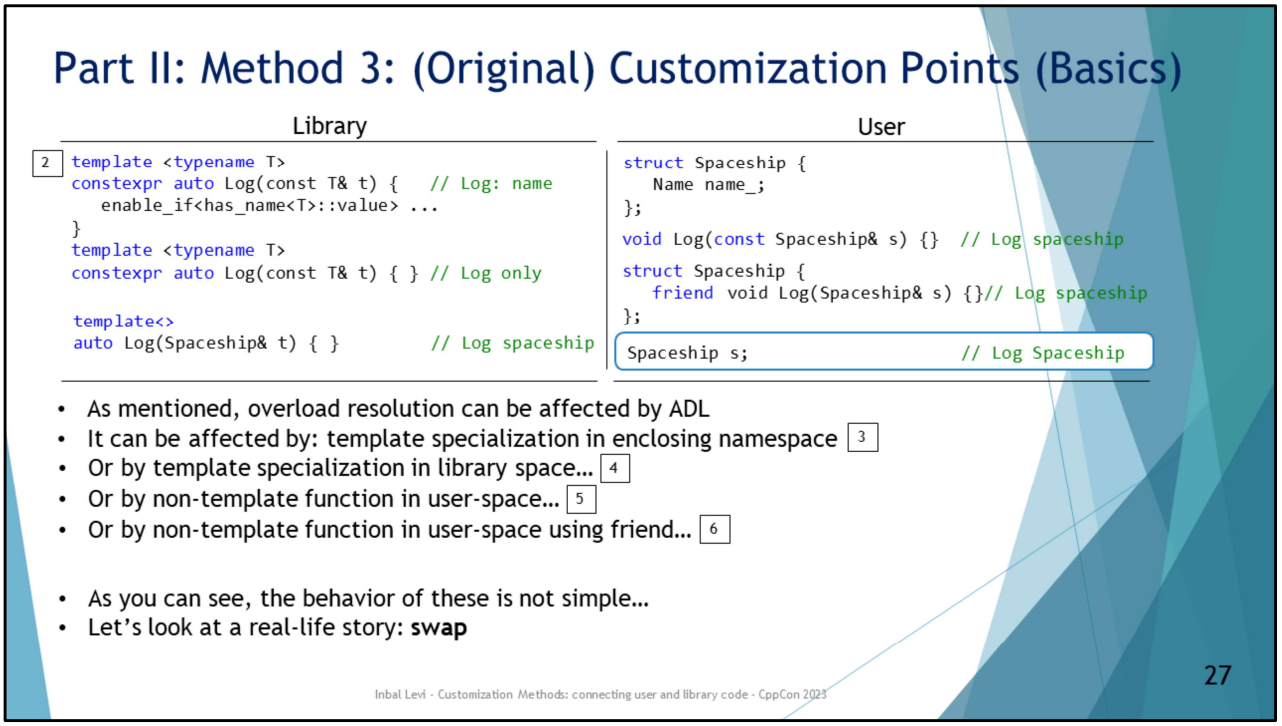
- **To summarize:**
  - To compile a function call, the compiler must first perform name lookup, which may involve argument-dependent lookup, and for function templates may be followed by template argument deduction.
  - If these produce more than one *candidate function*, then overload resolution is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



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



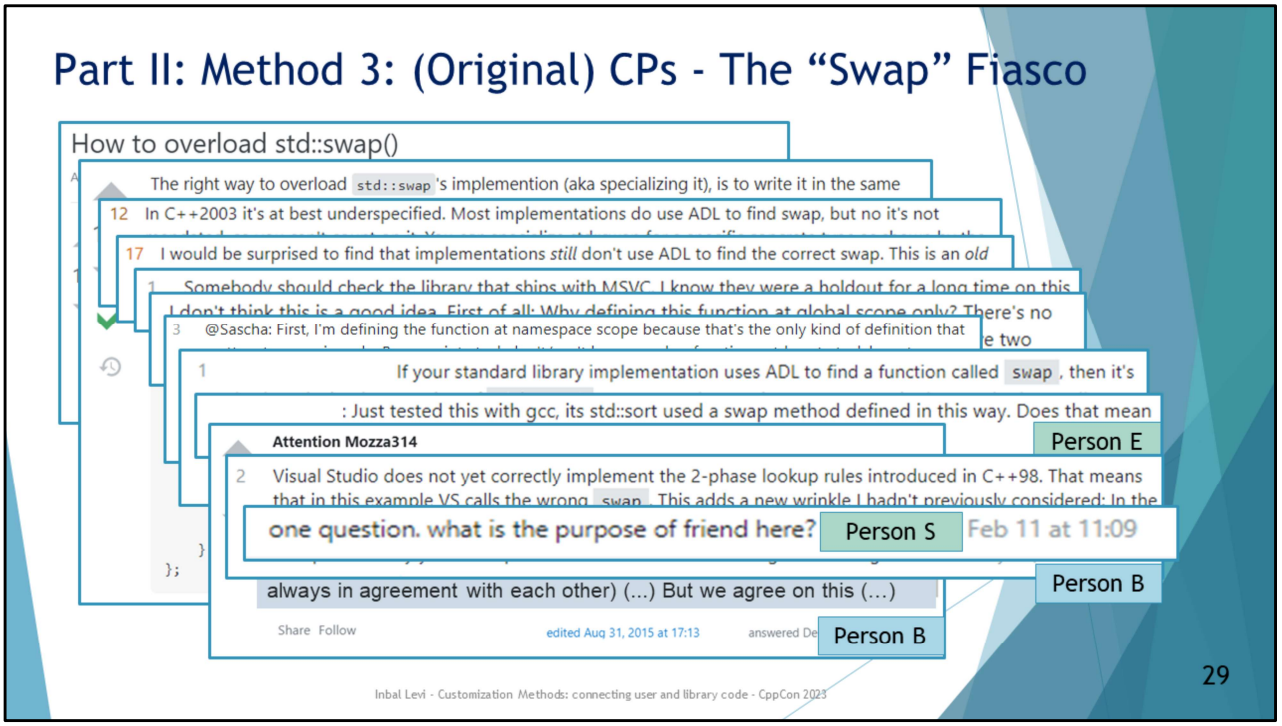
3. https://godbolt.org/z/vTeTEcYd5
4. https://godbolt.org/z/hcPEqEx6f
5. https://godbolt.org/z/rK1zffh56
6. https://godbolt.org/z/xsc5r679W

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

- (*) Namespces: 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)

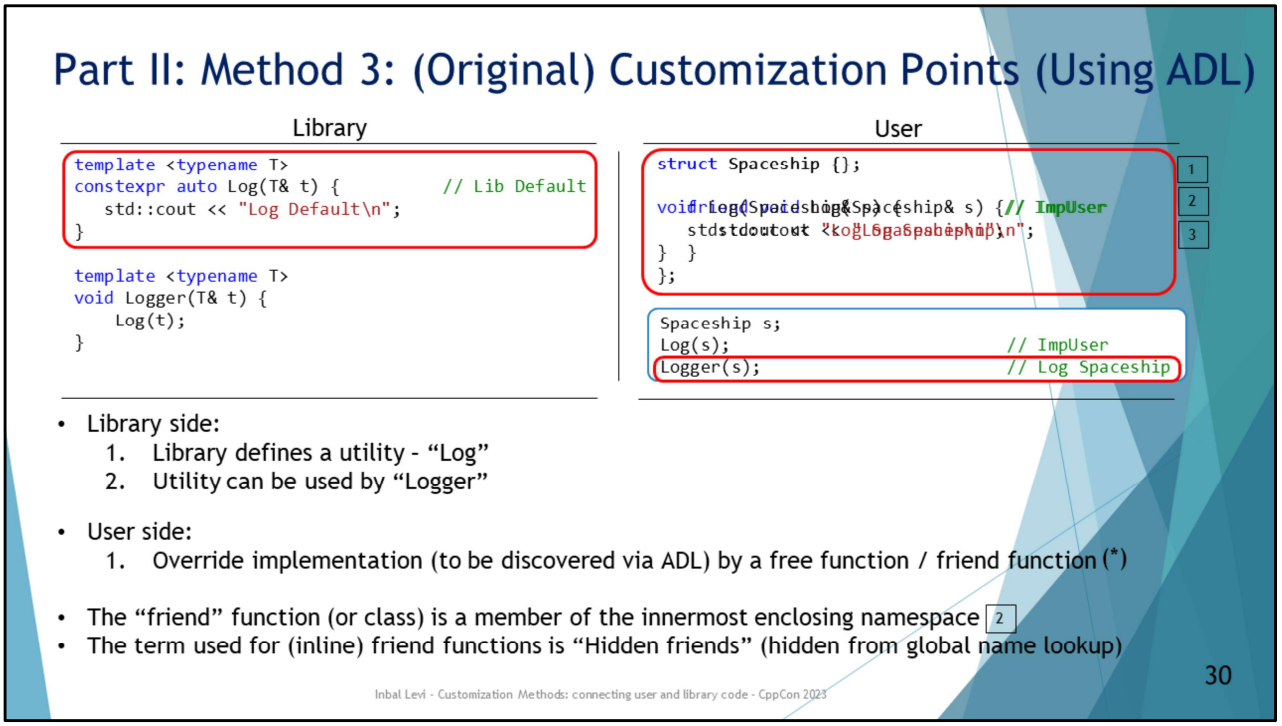
Part II.5: The “Swap” Fiasco



- 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 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)

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:

1. 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
}
```

2. Users can do so as well (where std have reasonable fallback), but this will bypass any constrains

```
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"

1. @@@ 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)

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”

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…

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)

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

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++0X Concepts could maybe offer a wider solution, but are out of scope for this talk
- Examples: Ranges library

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>\
      {}\
      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…

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 be 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

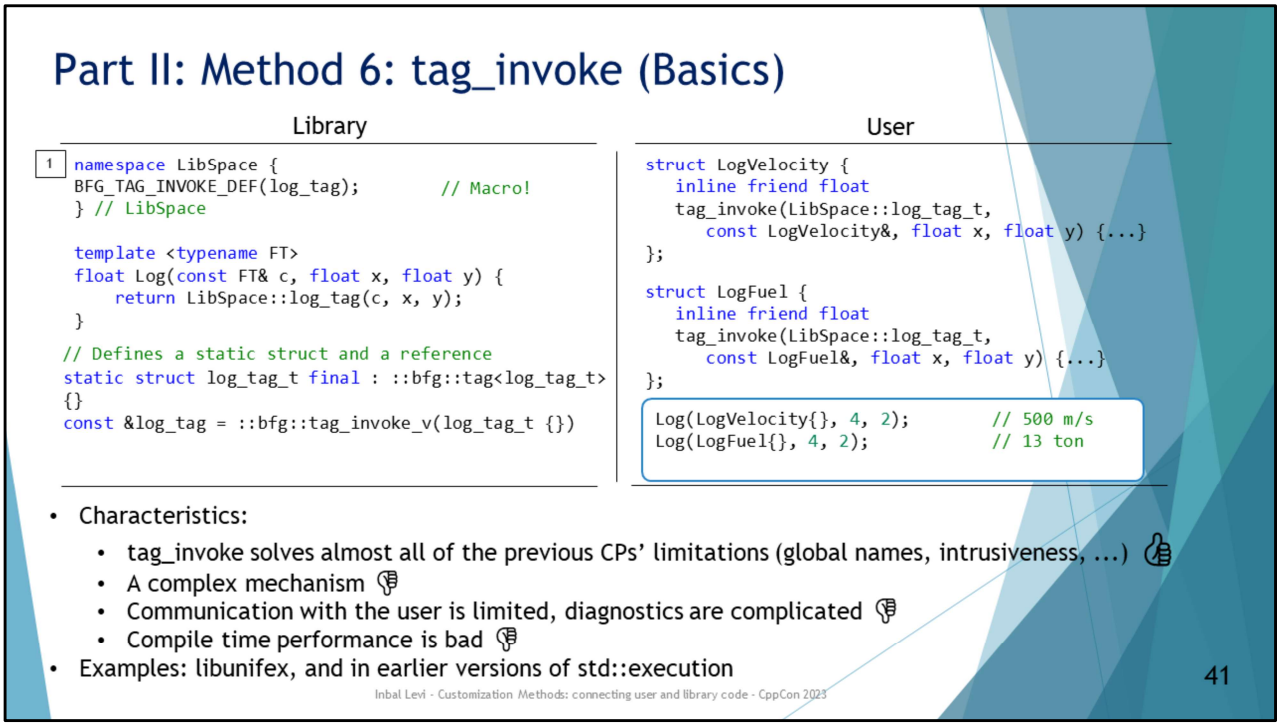
```
namespace LibSpace {
BFG_TAG_INVOKE_DEF(log_tag);          // Macro!
} // LibSpace

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>\
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



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)

1 Slides

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"

@@@ 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)**

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
}
```

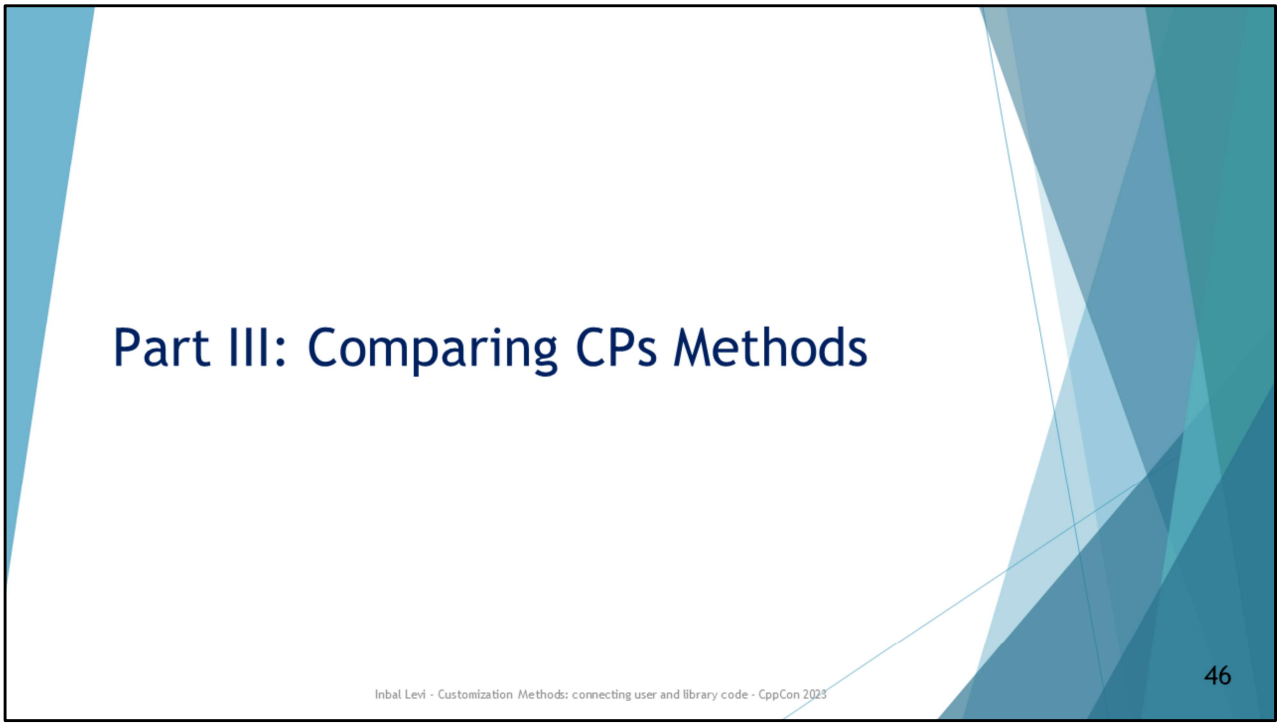
```
Spaceship s;
Log(s);                    // Reflection Logger
```

- 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"?

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
Inbal Levi - Customization Methods: connecting user and library code - CppCon 2023
46

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

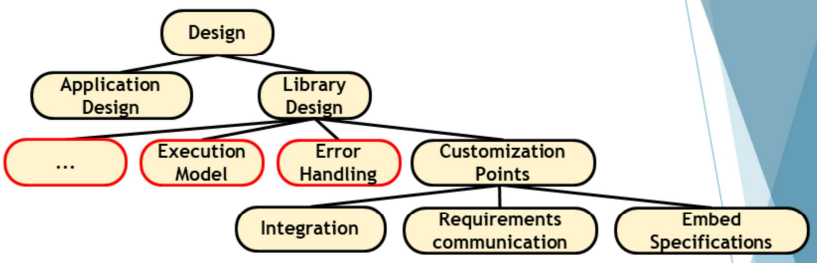
Properties:

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)

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
- Twitter.com/Inbal_l
- sinbal2lextra@gmail.com

Would love to get your input!

References

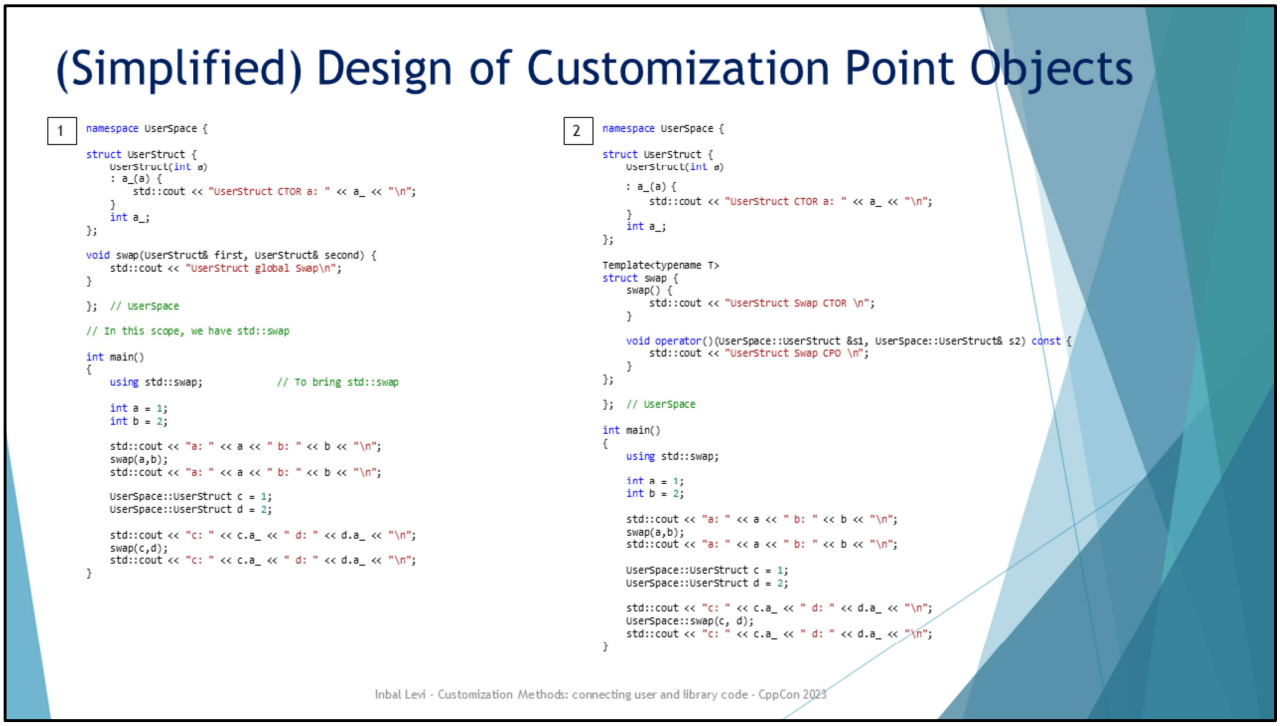
- **The Talk Is Based On:**
  - **P2279: We need a language mechanism for customization points (Barry Revzin)**
  - **Talk: 'tag_invoke' - An actually good way to do customization points (Gašper Ažman)**
  - **N4381: Suggested Design for Customization Points (Eric Niebler)**

- 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 (Dominiak, 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)

- 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) ☺



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

```cpp
#include <utility>
#include <iostream>

namespace new_std
{
  namespace __detail
  {
    // @@@@ Define "begin"s free functions @@@@
    // For arrays
    template<class T, size_t N>
    constexpr T* begin(T (&a)[N]) noexcept
    {
      return a;
    }

    // For containers
    // (trailing return type needed for SFINAE)
    template<class _RangeLike>
    constexpr auto begin(_RangeLike && rng) ->
      decltype(std::forward<_RangeLike>(rng).begin())
    {
      return std::forward<_RangeLike>(rng).begin();
    }

    // @@@@ begin functor @@@@
    // Class __begin_fn, with templated operator()(R&& r)
    struct __begin_fn
    {
      template<class R>
      constexpr auto operator()(R && rng) const ->
        decltype(begin(std::forward<R>(rng)))
      {
        return begin(std::forward<R>(rng)); // Unqualified Function call
      }
    };
  } // __detail

  // @@@@@ To avoid ODR violations: @@@@@
  // 1. (Boilerplate) Define struct (wrapper), which contains static constepxr 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 constepxr T value
  template<class T>
  constexpr T __static_const<T>::value;

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

int main()
{
    using namespace new_std;
    int arr[3] = {1,2,3};
    std::cout << begin(arr) << "\n";
}
```


NOTICE: This is slides code, not production level code

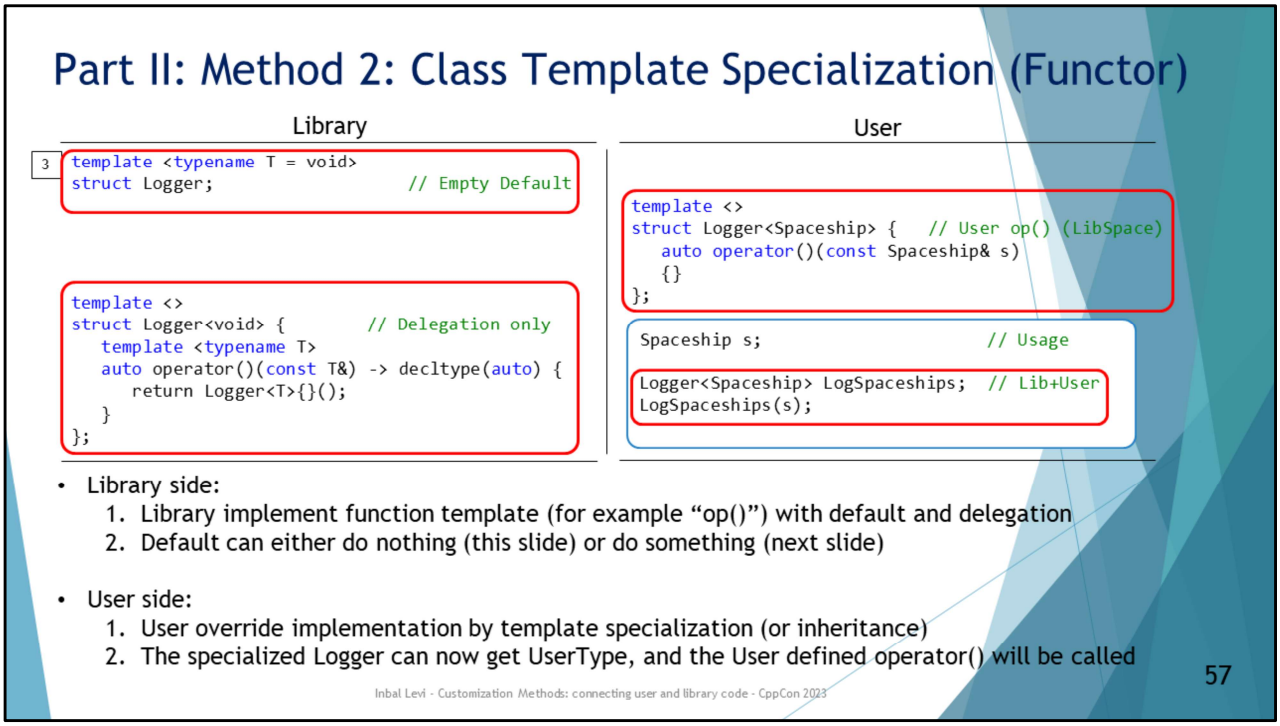
1. Full “begin” CPO example (including return types): https://godbolt.org/z/1d53zsbdG

CPOs Full Example - swap

```cpp
#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<R>(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 constepxr 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 constepxr 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);
}
```


NOTICE: This is slides code, not production level code

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



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