Core C++
2022
Sep. 5-7, 2022 :: Tel-Aviv, Israel

Customization Methods:

Connecting User And Library Code

Inbal Levi

THE ACADEMIC COLLEGE
of TEL AVIV-YAFFO

Who Am I?

- A C++ Developer at SolarEdge
- Active member of ISO C++ work group (WG21):
  - Israeli NB Chair
  - Ranges SG Chair
  - Assistant LEWG Chair

- I love software design
- I also have a passion for cataloguing, which is the main reason I’m giving this talk

Outline

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 I: What Are Customization Points

Part I: What Are Customization Points

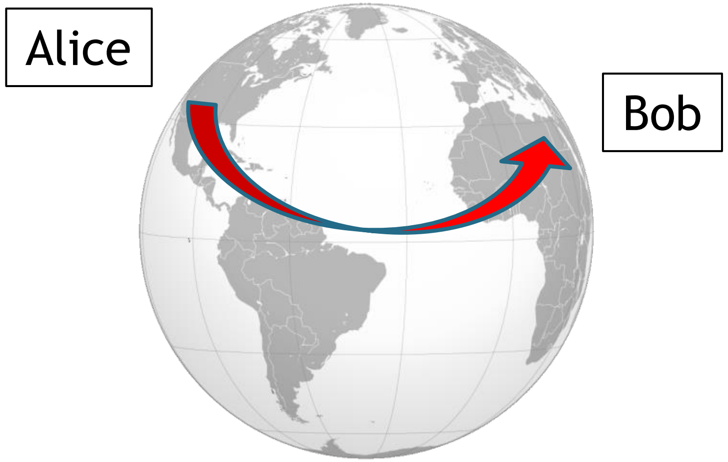
Library

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

User

```
struct UserType {
   friend ostream& operator<<(ostream& os,
         const UserType& t) {
         os << "UserType[data=" << t.data << "] ";
      return os;
   }
};
```

```
UserType t;
LibLogger(t);                // Log: UserType[data=6]
```



Part I: What Are Customization Points

- From Wikipedia:

  > “A [software] library is a collection of non-volatile resources used by computer programs, often for software development.”

- A 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

- "Customization Points" mentioned by Eric Niebler, in a blog from 2014:
  Customization Point Design in C++11 and Beyond

- In 2015, appeared in the 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."

- Can be broadened, I 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"

→ **Conclusion 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”, this talk will take a similar approach.

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

Part I: What Are Customization Points

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

→ **<u>Conclusion III</u>**: CPs are a means to embed specification (from the user into the library)

Part I: What Are Customization Points

- **To summarize:**

→ **<u>Conclusion I</u>**: CPs are a means of integration

→ **<u>Conclusion II</u>**: CPs are a means of communication

→ **<u>Conclusion III</u>**: CPs are a means to embed specification

Part I: (A Short Detour) Overload Resolution & Friends

- **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]

Part I: (A Short Detour) Overload Resolution & Friends

- **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: A 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. Deducing This (C++23)
  7. tag_invoke (not in the standard)
  8. Customizable Functions (not in the standard)
  9. Reflection (not in the standard)

- We will skip through some of the details (explore the code snippets to get more info)

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. Deducing This (C++23)
7. tag_invoke (not in the standard)
8. Customizable Functions (not in the standard)
9. Reflection (not in the standard)

Part II: Method 1: Inheritance

Library

1

```
struct LibType {
   virtual void Func1() = 0;     // Mandatory
   virtual void Func2() { }      // Functionality
   int data1;                    // Data
   using MyInt = int;            // Terminology
   static const int P = 3;       // Terminology
};
```

```
void LibAlgo(LibType& t) {
    t.Func2();
}
```

User

```
struct UserType : public LibType {
   virtual void Func1() override {}  // Mandatory
   virtual void Func2() override {}  // Specialization
};
```

```
UserType t;                                // Usage
t.Func1();
LibAlgo(t);                                // Func2 called
```

- Library side:
  1. Library defines LibType which contains functionality, data, and terminology
  2. Mandatory interface: by pure virtual functions
  3. Optional interface: (with default) by virtual functions

- User side:
  1. User can inherit from LibType to gain customizable functionality, data and terminology
  2. User can pass a UserType object to LibAlgo to gain functionality

Part II: Method 1: Inheritance

Library

```
struct LibType {
   virtual void Func1() = 0;     // Mandatory
   virtual void Func2() { }      // Functionality
   int data1;                    // Data
   using MyInt = int;            // Terminology
   static const int P = 3;       // Terminology
};

void LibAlgo(LibType& t) {
    t.Func2();
}
```

User

```
struct UserType : public LibType {
   virtual void Func1() override {}  // Mandatory
   virtual void Func2() override {}  // Specialization
};
```

```
UserType t;                            // Usage
t.Func1();
LibAlgo(t);                            // Backport
```

- Characteristics:
  1. Sharing: Interface, Functionality, Data, and Terminology **as a group** 👍
  2. Easy to opt-in, can’t accidently opt-in 👍
  3. There are limits on return type, runtime overhead (May be avoided by complex methods) 👎
  4. Non-intrusive into library namespace 👍
  5. Intrusive (*) into the type 👎

  (*) Has nothing to do with intrusive containers

- Examples: Everywhere

Part II: An Overview Of CPs Methods

- Methods:

  1. Virtual Functions
  2. ➡ **Class Template Specialization**
  3. (Original) Customization Points (ADL)
  4. Customization Point Objects
  5. Concepts
  6. Deducing This (C++23)
  7. tag_invoke (not in the standard)
  8. Customizable Functions (not in the standard)
  9. Reflection (not in the standard)

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

Library

1

```
template <typename T>
struct LibType {
  LibType() = delete;       // No default
                            // (Error on creation)
~~  void Foo() {}             // Functionality~~
~~  using MyInt = int;        // Terminology~~
~~  static const int P = 3;   // Terminology~~
};
```

```
template <typename T>    // Can limit by concepts (*)
void LibAlgo (T &t) {
    std::cout << "LibAlgo\n";
}
```

User

```
template <>                // Class specialization
struct LibType<UserType> {
  LibType() {}
};
```

```
LibType<UserType> t;                // User CTOR
LibAlgo(t);                         // “LibAlgo”
```

- Library side:
  1. Library defines template class LibType with deleted CTOR
  2. Library defines LibAlgo. Making it work **specifically** for LibType is limited (*)

- User side:
  1. Override **(the entire)** implementation by template specialization
  2. User can instantiate and use LibType<UserType>
  3. User gain LibAlgo functionality

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

Library

2

```
template <typename T>
struct LibType {
  LibType() = delete;       // No default
                            // (Error on creation)
  void Foo() {}             // Functionality
  using MyInt = int;        // Terminology
  static const int P = 3;  // Terminology
};

template <typename T>    // Can limit by concepts (*)
void LibAlgo (T &t) {
    Foo();
}
```

User

```
template<>      // CTOR explicit specialization (*)
LibType<int>::LibType() { }

template<>      // Explicit specialization (Foo) (*)
void LibType<int>::Foo() { }

LibType<UserType> t;              // User CTOR
LibAlgo(t);                       // “LibAlgo”
```

- Library side:
  1. Library defines template class LibType with deleted CTOR
  2. Library defines LibAlgo. Making it work **specifically** for LibType is limited (*)
- User side:
  1. Override **(some of the)** implementation by template specialization (**in containing namespace (*)**)
  2. User can instantiate and use LibType<UserType>
  3. User gains LibAlgo functionality

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

Library

3

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

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

User

```
template <>
struct LibFunc<UserType> {   // User op() (LibSpace)
   auto operator()(const UserType& t)
   {}
};
```

```
UserType userType;                      // Usage

LibFunc<UserType> LibUserFunc;   // Lib+User
LibUserFunc(userType);
```

- Library side:
  1. Library can 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 LibFunc can now get userType, and the User defined operator() will be called

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

Library

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

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

User

```
// No Implementation required for default

template <>
struct LibFunc<UserType> {    // User op() (LibSpace)
   auto operator()(const UserType& t)
   {}
};
```

```
UserType userType;
LibFunc LibDefaultFunc;
LibFunc<UserType> UserFunc;
LibDefaultFunc(42);             // (default)
UserFunc(userType);             // User Implementation
```

- Library side:
  1. Library defines a template class LibFunc with default
  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 LibDefaultFunc, or the UserFunc

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

Library

3

4

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

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

User

```cpp
// No Implementation required for default

template <>
struct LibFunc<UserType> {   // User op()
   auto operator()(const UserType& t)
   {}
};
```

```cpp
UserType t;
LibFunc libFunc;                   // Usage (default)
LibFunc<UserType> libUserFunc; // Usage ImpUser
libFunc(42);
libUserFunc(t);
```

- Characteristics:
  1. Namespace or type intrusive (User adds code into LibSpace or inherits from LibFunc<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: fmtlib (uses the Type version) 5

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. Deducing This (C++23)
   7. tag_invoke (not in the standard)
   8. Customizable Functions (not in the standard)
   9. Reflection (not in the standard)

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

Library

1

```
template <typename T>
constexpr auto LibFunc(T& t) { }           // Imp1
template <typename T>
constexpr auto LibFunc(const T& t) { }     // Imp2
```

User

```
struct UserType {};
```

```
UserType t;                          // Usage
LibFunc(t);                          // Imp1
LibFunc(42);                         // Imp2
```

- Library defines difference ‘versions’ of LibFunc
- This method relies on the fact that the proper function will be called
- This method also allows porting types’ behavior from the user

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

Library

2

```
template <typename T>
constexpr auto LibFunc(T& t) { }          // Imp1
template <typename T>
constexpr auto LibFunc(const T& t) { }    // Imp2


template<>                                // ImpUser
auto LibFunc(UserType& t) { }
```

User

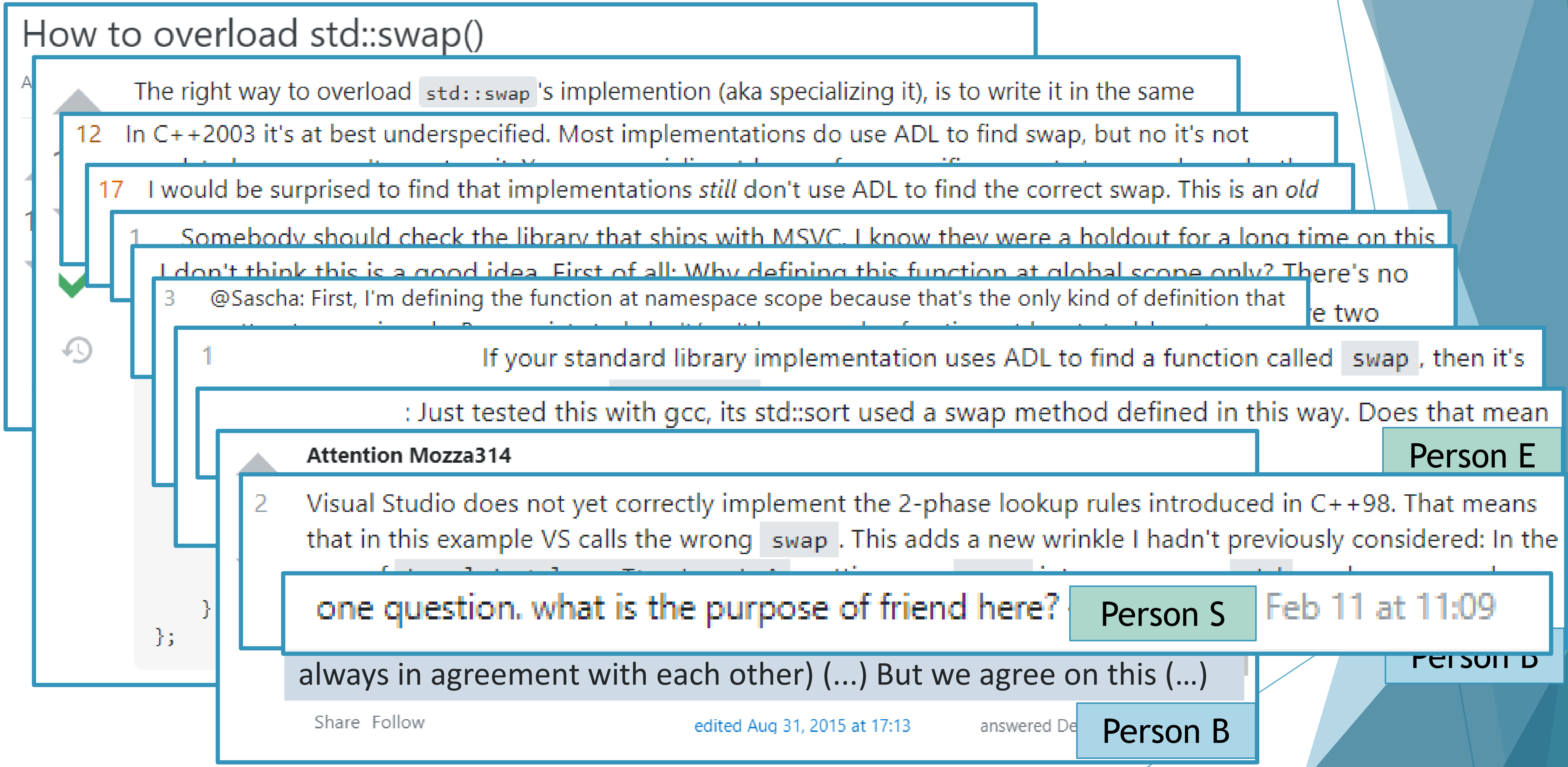
```
struct UserType {};

void LibFunc(UserType& t) {}           // ImpUser

struct UserType {
   friend void LibFunc(UserType& t)    // ImpUser
   {}
};

UserType t;                            // ImpUser
```

- 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

Part II.5: The “Swap” Fiasco

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



How to overload std::swap()

The right way to overload `std::swap`'s implemention (aka specializing it), is to write it in the same

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

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

1 Somebody should check the library that ships 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

e two

3 @Sascha: First, I'm defining the function at namespace scope because that's the only kind of definition that

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

: Just tested this with gcc, its std::sort used a swap method defined in this way. Does that mean

Person E

Attention Mozza314

2 Visual Studio does not yet correctly implement the 2-phase lookup rules introduced in C++98. That means that in this example VS calls the wrong `swap`. This adds a new wrinkle I hadn't previously considered: In the

one question. what is the purpose of friend here? Person S Feb 11 at 11:09

```
    }
};
```

always in agreement with each other) (...) But we agree on this (...)

Share Follow

edited Aug 31, 2015 at 17:13

answered De

Person B

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

Library

```
template <typename T>
constexpr auto LibFunc(T& t) {          // LibImp
   std::cout << "LibFunc Generic\n";
}

template <typename T>
void LibAlgo(T& t) {
    LibFunc(t);
}
```

User

```
struct UserType {};

void LibFunc(UserType& u) {                      // ImpUser
   std::cout << “LibFunc UserType\n";
}
```

```
   friend void LibFunc(UserType& u) {          // ImpUser
      std::cout << "UserType\n";
   }
};
```

1

2

3

```
UserType t;
LibFunc(t);                        // ImpUser
LibAlgo(t);                        // Port ImpUser
```

- Library side:
  1. Library defines a utility
  2. Utility can be used by LibAlgo

- User side:
  1. Override implementation (to be discovered via ADL) by a free function / friend function (*)

(*) The friend class or function is a member of the innermost enclosing namespace. 2
Term used for (inline) friend functions is “Hidden friends” (hidden from global name lookup)

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

Library

1

```
template <typename T>
constexpr auto LibFunc(T& t) {        // LibImp
   std::cout << "LibFunc Generic\n";
}

template <typename T>
void LibAlgo(T& t) {
    LibFunc(t);
}
```

User

```
struct UserType {};

void LibFunc(UserType& u) {     // ImpUser
   std::cout << “LibFunc UserType\n";
}
```

```
UserType t;
LibFunc(t);                    // ImpUser
LibAlgo(t);                    // Port ImpUser
```

- Characteristics:
  1. Porting functionality for UserType 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) presented the following issues:

  1. Calls within a template should be Unqualified (to add the std::swap option to the candidates)

```
template <typename T>
void foo(T& a, T& b) {

    std::swap(a, b);    // Wrong
}
```

```
template <typename T>
void bar(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 the issues mentioned, Eric Niebler presented: “Customization Point Objects”

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. Deducing This (C++23)
7. tag_invoke (not in the standard)
8. Customizable Functions (not in the standard)
9. Reflection (not in the standard)

Part II: Method 4: Customization Point Objects

Library

1

```
namespace __detail {

template <typename T>
constexpr auto LibFunc(T& t) { }  // Func Imp

struct LibFunc_fn {                   // Func Obj Imp
   template<class T>
   auto operator()(T&& t1) const {
      LibFunc(std::forward<T>(t1));
   }
};

}
// Create a global LibFunc_fn (...)
```

User

```
struct UserType {};

void LibFunc(UserType& t1) { }

using LibSpace::LibFunc;
using UserSpace::LibFunc;
UserType t;
LibFunc(t);                    // User Implementation
```

- Library side:
  1. Library defines a template function LibFunc
  2. Library creates a LibFunc_fn functor with template operator()
  3. Library creates a global LibFunc_fn (with some magic to avoid ODR violation)
- User side:
  1. User creates a LibFunc accepting UserType

Part II: Method 4: Customization Point Objects

Library

1

```
namespace __detail {

template <typename T>
constexpr auto LibFunc(T& t) { }  // Func Imp


struct LibFunc_fn {                   // Func Obj Imp
   template<class T>
   auto operator()(T&& t1) const {
      LibFunc(std::forward<T>(t1));
   }
};

}
// Create a global LibFunc_fn (...)
```

User

```
struct UserType {};

void LibFunc(UserType& t1) { }
```

```
using LibSpace::LibFunc;
using UserSpace::LibFunc;
UserType t;
LibFunc(t);                   // User Implementation
```

- 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…

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. Deducing This (C++23)
  7. tag_invoke (not in the standard)
  8. Customizable Functions (not in the standard)
  9. Reflection (not in the standard)

Part II: Method 5: Concepts

Library

```cpp
template<typename T>
concept LibConcept = requires {
   typename T::lib_concept_tag;
};
template <LibConcept T>
constexpr auto LibFunc(T& t) { }      // ImpDefInit

template <typename T>                 // constraints
void LibAlgo(const T& t) {
      // constraints
}
```

User

```cpp
struct UserType {
    using lib_concept_tag = void;
};
```

```cpp
using LibSpace::LibFunc;
UserType t;
LibFunc(t);                        // Usage
LibAlgo(t);
```

- 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. Using tags to extend beyond syntactic conformance 😐
     C++0X Concepts could maybe offer a wider solution, but are out of scope for this talk

- Examples: Ranges library

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. ➡ **Deducing This (C++23)**
7. tag_invoke
8. Customizable Functions (not in the standard)
9. Reflection (not in the standard)

Part II: Method 6: Deducing This

Library

```
struct LibType {
   auto LibFunc() const&;
   auto LibFunc() &;
   auto LibFunc() &&;
};
```

```
struct LibType {
   template<typename Self>
   auto LibFunc(this Self&& self);
};
```

```
struct LibType {
   auto LibFunc(this const LibType& t);
   auto LibFunc(this LibType& t);
   auto LibFunc(this LibType&& t);
};
```

User

- Mechanism proposed in P0847 by Gašper Ažman, Sy Brand, Ben Deane, Barry Revzin (voted into C++23)
- Allows specifying or deducing the value category of the expression from within the member function
- Syntax difference is mainly the addition of an explicit “this”

Part II: Method 6: Deducing This

Library

```
struct LibType {
   template <typename Self>
   auto LibAlgo(this Self&& self, int a) {
      self.UserFunc1(self.UserFunc2(a));  // From User
   }
};
```

User

```
struct UserType : public LibType {
   auto UserFunc1(int a) { }
   auto UserFunc2(int a) const {
      return a * 6;
   }
};
```

```
using UserSpace::UserType;
UserType t;
t.LibAlgo(42);                        // Usage
```

- Library Side:
  - Library defines LibType
  - LibType defines LibAlgo template (which can port functionality and data from UserType)
- User Side:
  - User defines UserType inheriting from LibType, to get a CRTP-like behavior

- Explicit “self” template parameter provides advantages over Virtual Functions:
  1. Porting usability at **compile time** (and without the need for declaring (pure) virtual functions)
  2. Generalized return type

Part II: Method 6: Deducing This

Library

```
1 struct LibType {
2    template <typename Self>
     auto LibAlgo(this Self&& self, int a) {
        self.UserFunc1(self.UserFunc2(a));   // From User
     }
  };
```

User

```
struct UserType : public LibType {
   auto UserFunc1(int a) { }
   auto UserFunc2(int a) const {
      return a * 6;
   }
};
```

```
using UserSpace::UserType;
UserType t;
t.LibAlgo(42);                     // Usage
```

- Characteristics:
  1. Clear, expressive code 👍
  2. Non-intrusive - user doesn’t need to add code into LibSpace 👍
  3. No need to reserve names globally 👍
  4. Explicitly opt-in 👍
  5. Possibility for accidental unintended opt-in 👎

- Examples: None yet

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. Deducing This (C++23)

  ➡ 7. tag_invoke

  8. Customizable Functions (not in the standard)
  9. Reflection (not in the standard)

Part II: Method 7: tag_invoke (Basics)

Library

1

```
namespace LibSpace {
BFG_TAG_INVOKE_DEF(lib_tag);          // Macro
} // LibSpace

template <typename FT>
float LibFunc(const FT& c, float a, float b) {
    return LibSpace::lib_tag(c, a, b);
}

// 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 UserFunc1 {
   inline friend float
   tag_invoke(LibSpace::lib_tag_t,
      const UserFunc1&, float a, float b) { }
};

struct UserFunc2 {
   inline friend float
   tag_invoke(LibSpace::lib_tag_t,
      const UserFunc1&, float a, float b) { }
};
```

```
LibFunc(UserFunc1{}, 4.2, 3);    // Usage
LibFunc(UserFunc2{}, 4.2, 3);    // Usage
```

- 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…

Part II: Method 7: tag_invoke (Basics)

Library

1

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

```
template <typename FT>
float LibFunc(const FT& c, float a, float b) {
    return LibSpace::lib_tag(c, a, b);
}
```

```
// 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 {})
```

```
static struct lib_tag_t final : ::bfg::tag<lib_tag_t>
{}
const &lib_tag = ::bfg::tag_invoke_v(lib_tag_t {})
```

User

```
struct UserFunc1 {
   inline friend float
   tag_invoke(LibSpace::lib_tag_t,
      const UserFunc1&, float a, float b) { }
};
```

```
struct UserFunc2 {
   inline friend float
   tag_invoke(LibSpace::lib_tag_t,
      const UserFunc2&, float a, float b) { }
};
```

```
LibFunc(UserFunc1{}, 4.2, 3);    // Usage
LibFunc(UserFunc2{}, 4.2, 3);    // Usage
```

- Library Side:
  - Library defines a lib_tag object (using the macro)
  - Library defines LibFunc, delegates to tag
- User Side:
  - User defines functionality using the reserved word “tag_invoke” with first param lib_tag_t

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

Part II: Method 7: tag_invoke (Basics)

Library

1

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

template <typename FT>
float LibFunc(const FT& c, float a, float b) {
    return LibSpace::lib_tag(c, a, b);
}

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

User

```
struct UserFunc1 {
   inline friend float
   tag_invoke(LibSpace::lib_tag_t,
      const UserFunc1&, float a, float b) { }
};

struct UserFunc2 {
   inline friend float
   tag_invoke(LibSpace::lib_tag_t,
      const UserFunc2&, float a, float b) { }
};
```

```
LibFunc(UserFunc1{}, 4.2, 3);    // Usage
LibFunc(UserFunc2{}, 4.2, 3);    // Usage
```

- 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

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. Deducing This (C++23)
    7. tag_invoke
    8. ➡ **Customizable Functions (not in the standard)**
    9. Reflection (not in the standard)

Part II: Method 8: Customizable Functions

Library

```
template <typename T>
virtual auto LibFunc1(T& t) = 0;          // Imp1

template <typename T>
virtual auto LibFunc2(T& t) default {     // Imp2
   // …
}
```

User

```
struct UserType {  // UserImp
   template <typename T>
   friend void LibSpace::LibFunc1(T& t) override
   { }
```

```
UserType t;                       // Usage
LibFun2(t);
```

- 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”

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. Deducing This (C++23)
  7. tag_invoke
  8. Customizable Functions (not in the standard)
  9. **Reflection (not in the standard)**

Part II: Method 9: Reflection

Library

```
template<class_type T, structural_subtype_of<T> U>
void LibFunc(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 UserType {
   // Implementation conforming to the
      requirements
}
```

```
UserType t;                          // Usage
LibFun(t);
```

- 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”?

Part III: Comparing CPs Methods

Comparing Customization Points Methods

		Inheritance	CTS	CPs (ADL)	CPOs	Concepts (+nominal)	Deducing This	tag_invoke	Custom functions	Reflection
Integrate	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

Part IV: What's Next?

Part IV: What’s Next?

→ P2300: std::execution did not make it into C++23
→ (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 the solutions, unfolded from last, into C++26 ☺

- Meanwhile, we’ll have to settle with non language solutions…

Would love to get your input!

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)

References

- Related Sources:
  - 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

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";
    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";
}
```

CPOs Full Example - begin

1

```
#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";
}
```

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<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);
}
```

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