Compile-time type introspection using SFINAE

Jean Guegant - C++ Stockholm 0x02 - February 2017

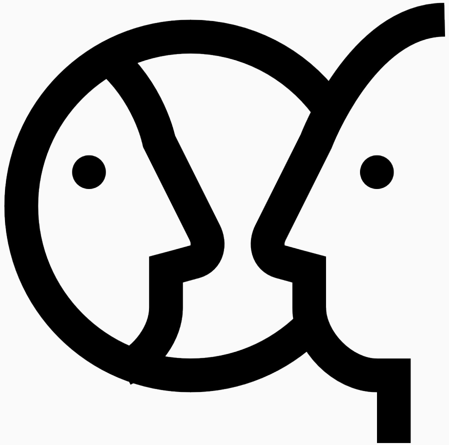
Acknowledgement

- Boost.Hana (Louis Dionne)
- cppreference.com
- JetBrain (Clion license)

Introspection in C++ you said?

Introspection

- *"**Type introspection** is the ability of a program to examine the type or properties of an **object** at **runtime**."*
- *"**Reflection** is the ability examine, introspect, and modify its own structure and behavior at **runtime**."*
- Generic programming, flexibility, interfacing...
- Available in most of the modern languages



Runtime introspection in C++

- RTTI (runtime type information)
  - **typeid**
  - **std::type_info**
  - **...**
- Highly limited
- Not always available
- Compiler specific
- Runtime cost
- No **reflection**

```cpp
#include <iostream>

struct Base { virtual ~Base() = default; };
struct Derived : Base {};

Derived d;
Base &b = d;
std::cout << typeid(b).name() << std::endl;
```

```cpp
struct A {};

std::string to_string(const A&)
{
    return "I am a A!";
}


struct B
{
    std::string serialize() const
    {
        return "I am a B!";
    }
};


struct C { std::string serialize; };

std::string to_string(const C&)
{
    return "I am a C!";
}
```

```cpp
struct D : A
{
    std::string serialize() const
    {
        return "I am a D!";
    }
};


struct E
{
    struct Functor
    {
        std::string operator()()
        {
            return "I am a E!";
        }
    };

    Functor serialize;
};
```

Compile-time introspection in C++

- Template metaprogramming
- Constexpr
- Leverage on compiler
  - Type-safety
  - No runtime cost
- Compile-time polymorphism

```cpp
auto has_serialize = hana::is_valid([](auto&& x) -> decltype(x.serialize()) { });

template <class T> auto serialize(T& obj)
{
   if constexpr (has_serialize(obj)) {
       return obj.serialize();
   } else {
       return to_string(obj);
   }
}

A a;
B b;
C c;
D d;
E e;

std::cout << serialize(a) << std::endl;
std::cout << serialize(b) << std::endl;
std::cout << serialize(c) << std::endl;
std::cout << serialize(d) << std::endl;
std::cout << serialize(e) << std::endl;
```

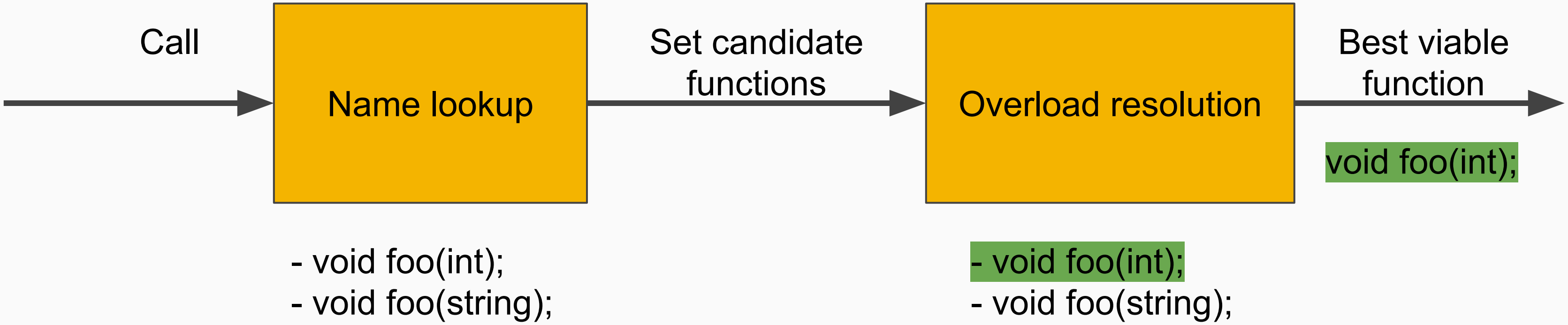
```cpp
auto has_serialize = hana::is_valid([](auto&& x) -> decltype(x.serialize()) { });

template <class T> auto serialize(T& obj)
{
   if constexpr (has_serialize(obj)) {
       return obj.serialize();
   } else {
       return to_string(obj);
   }
}

A a;
B b;
C c;
D d;
E e;

std::cout << serialize(a) << std::endl;                    // I am a A!
std::cout << serialize(b) << std::endl;                    // I am a B!
std::cout << serialize(c) << std::endl;                    // I am a C!
std::cout << serialize(d) << std::endl;                    // I am a D!
std::cout << serialize(e) << std::endl;                    // I am a E!
```

SFINAE

Substitution Failure Is Not An Error

Behind the scene of a function call

- Example: foo(42);



For function templates

- Example: foo<int>(42, 43);

Name lookup → Template argument substitution → Template argument deduction → Template argument substitution → Overload resolution →

Name lookup

- template
<class T, class U>
foo(T, U);

Template argument substitution

- template
<class int, class U>
foo(**int**, U);

Template argument deduction

U = int

Template argument substitution

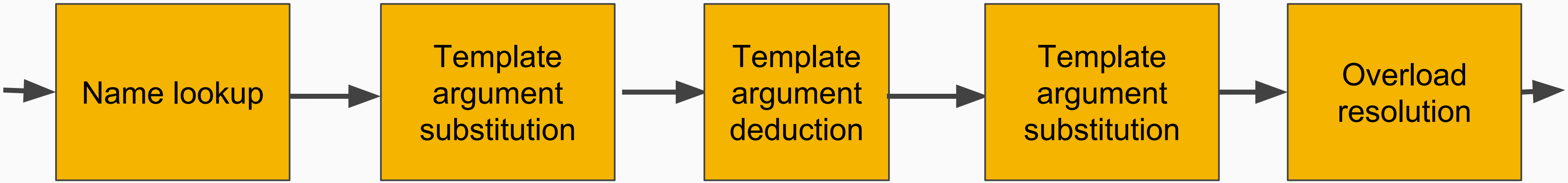
- template
<class int, class int>
foo(int, **int**);

Overload resolution

- template
<class int, class int>
foo(int, int);

For function templates

- Example: foo(42); SFINAE (Substitution Failure Is **Not** An **Error**)



- template
<class T>
void foo(T, T::x* = 0);

- void foo(int);

- template
<class T>
void foo(T, T::x* = 0);

- void foo(int);

T = int

- template
<class int>
void foo(int, int::x* = 0);

- void foo(int);

void foo(int);

No compiler error !

Overload resolution

- Function candidate set ⇒ viable functions ⇒ best viable function
- Best viable function rules ~=
  1. Best conversion sequence (int ⇒ int > int ⇒ double)
  2. **Function** over **function template**
  3. **Function template** over **variadic function**
  4. Best specialised function template

A bit of C++ archeology (C++98)

From SFINAE to a bool in the ancient times

```
// Best viable function if has type called x
template <class T>
bool has_type_x(T t, typename T::x* = 0)
{
    return true;
}


// Sink if no member type called x
bool has_type_x(...)
{
    return false;
}
```

```
struct A
{
    typedef int x;
};

A a;

std::cout << has_type_x(42) << std::endl; // 0
std::cout << has_type_x(a) << std::endl; // 1

// ERROR or WARNING (VLA)!
// has_type is not compile-time
char test[has_type_x(42) * 4 + 40];
```

From SFINAE to a compile-time bool ...

- **sizeof**: size of the object of the type that would be returned by expression, if **evaluated**.

```
template <class T> struct has_type_x
{
   // Discriminative return types.
   typedef char yes[1];
   typedef char no[2];

   template<class C>
   static yes& test(typename C::x* = 0);

   template<class> static no& test(...);

   // Compile-time evaluation using sizeof.
   enum { value = sizeof(test<T>(0)) == sizeof(yes) };
};
```

```
std::cout << has_type_x<A>::value; // 1
std::cout << has_type_x<int>::value; // 0


// Compile-time, hurray!
char test[has_type_x<A>::value * 4 + 40];
```

```cpp
template <class T> struct has_serialize
{
   typedef char yes[1];
   typedef yes no[2];

   // Check that &C::serialize has the same signature as the first argument!
   // really_has<std::string (C::*)(), &C::serialize> should be substituted by
   // really_has<std::string (C::*)(), std::string (C::*)() &C::serialize> and work!
   template <class U, U u> struct really_has;

   // Two overloads for non-const and const!
   template <typename C> static yes& test(really_has<std::string (C::*)(), &C::serialize>*);
   template <typename C> static yes& test(really_has<std::string (C::*)() const, &C::serialize>*);

   template <typename> static no& test(...);

   enum { value = sizeof(test<T>(0)) == sizeof(yes) };
};

std::cout << has_serialize<B>::value << std::endl; // 1
std::cout << has_serialize<int>::value << std::endl; // 0
```

Example 4: C++03 - First has_serialize implementation

A first draft of serialize

```cpp
template <class T>
std::string serialize(const T& obj)
{
    if (has_serialize<T>::value) {
        // Dead branch for int?
        return obj.serialize();
    } else {
        return std::to_string(obj);
    }
}

// Should print 42?
serialize(42);
```

```cpp
std::string serialize(const int& obj)
{
    if (false) {
        // Error: int(42).serialize();
        return obj.serialize();
    } else {
        return std::to_string(obj);
    }
}

// int has no function member serialize
serialize(42);
```

Compile-time branching using enable_if

- Utility to **SFINAE** according to a compile-time **boolean**, for instance on return types

```
template<bool B, class T = void> // Default template version.
struct enable_if {}; // Doesn't define "type", substitution will fail if you try to access it.

template<class T> // A specialisation used if the expression is true.
struct enable_if<true, T>
{
    typedef T type; // Do have a "type" and won't fail on access.
};

enable_if<true, std::string>::type x = "hej"; // std::string x = “hej”;
enable_if<false, std::string>::type x = "fail"; // Failure no type declare!
```

Finally a working serialize!

```
// Use has_serialize + enable_if to SFINAE on the return type.
template <class T>
enable_if<has_serialize<T>::value, std::string>::type
serialize(const T& obj)
{
    return obj.serialize();
}

// Contra-SFINAE to avoid ambiguity
template <class T>
enable_if<!has_serialize<T>::value, std::string>::type // Watch out for the “!”
serialize(const T& obj)
{
    return to_string(obj);
}

serialize(42); // It works !
```

SFINAE pre-C++11

- Highly verbose
- Tricks
- Badly standardised
- Compiler specific
  - **typeof** in GCC
  - **Expression SFINAE** not required in the standard (e.g `&C::serialize`)

⇒ `Use Boost when necessary`

C++1x to our rescue

C++11’s expression SFINAE tools

- **Expression SFINAE:**
  - An **Ill-formed expression** used in a template parameter type or function type will **SFINAE.**
- **decltype** keyword: get type and value category of an expression. Similar to **typeof, sizeof**.
- **declval** utility: fake instantiation of a type (even without an accessible constructor).

```
struct NonDefault {
   NonDefault() = delete;
   int foo() const {return 1;}
};

decltype(std::declval<NonDefault>().foo()) x = 42; // int x = 42;

decltype(std::declval<NonDefault>().foo(), bool()) x = 1;  // bool x = 1;
```

C++11's constexpr

- **constexpr** keyword: hint that an expression is constant and could be evaluate directly at compile time.

```
constexpr int factorial(int n)
{
   return n <= 1? 1 : (n * factorial(n - 1));
}

// No needs for VLA, factorial evaluation is at compile-time!
char test[factorial(2) + 1];
```

- **std::false_type** and **std::true_type:** types that encapsulate a constexpr boolean "true" and a constrexpr boolean "false".

```
struct A : std::false_type { };
```

```cpp
template <class T> struct has_serialize
{
    // We test if the type has serialize using decltype and declval.
    template <typename C>
    static constexpr decltype(std::declval<C>().serialize(), bool())
    test(int /* unused */)
    {
        // We can return values, thanks to constexpr instead of playing with sizeof.
        return true;
    }

    template <typename C>
    static constexpr bool test(...)
    {
        return false;
    }

    // int is used to give the precedence!
    static constexpr bool value = test<T>(int());
};


std::cout << has_serialize<B>::value << std::endl; // 1
std::cout << has_serialize<int>::value << std::endl; // 0
```

Example 5: C++11 - A better has_serialize implementation - Using member functions **SFINAE**

```cpp
// Primary template, inherit from std::false_type.
// ::value will return false.
// Note: the second unused template parameter is set to default as std::string!!!
template <typename T, typename = std::string>
struct has_serialize
       : std::false_type
{

};

// Partial template specialisation, inherit from std::true_type.
// ::value will return true.
template <typename T>
struct has_serialize<T, decltype(std::declval<T>().serialize())>
       : std::true_type
{

};

std::cout << has_serialize<B>::value << std::endl; // 1
std::cout << has_serialize<int>::value << std::endl; // 0
```

- **Note:** see C++17 **std::void_t** for a similar **SFINAE** tool using a default argument.

C++11's auto

- **auto** specifier to let the compiler infer a type:

```
std::string foo();
auto test = foo(); // Inferred: std::string test = foo();
```

- **auto** specifier for function declarations using the trailing return type syntax:
    - auto function(params...) -> return type

```
template <typename T>
auto foo(const T& t) -> decltype(t.serialize()) // Possibility to SFINAE on t.serialize()
{
    return t.serialize();
}
```

C++14’s generic lambdas

- **auto parameters:** act like a template parameter would.

```
auto foo = [](auto& t) -> decltype(t.serialize()) { return t.serialize(); };
```

```
struct UnamedType
{
     // /!\ This signature is nice for SFINAE!
    template <typename T>
    auto operator()(T& t) const -> decltype(t.serialize())
    {
        return t.serialize();
    }
};

auto foo = UnamedType();
```

Blending time

is_valid requirements:

```
auto has_serialize = is_valid([](auto&& x) -> decltype(x.serialize()) { });
std::cout << has_serialize(42); // 0
```

- A function:
  - Compile-time.
  - Take a generic lambda that **SFINAE** on its return type.
  - Returns an object of a callable type, a “**container**”:
    - Keep the lambda type (**CheckingType**).
    - Take as an argument another object of any type (**TypeToCheck**).
    - Check whether that **TypeToCheck** would **SFINAE** on the lambda (**CheckingType**).
    - Return a compile-time boolean.

Building our is_valid function

```
template <class CheckingType> // Take any kinf od lambda!.
constexpr container<CheckingType> is_valid(const CheckingType& t)
{
    return container<CheckingType>();
}

template <class CheckingType>
struct container
{
     // ????
}
```

```
template <class CheckingType> // Store the type used to SFINAE.
struct container
{
private:
    template <class TypeToCheck>
    constexpr auto test_validity(int /* unused */)
          // SFINAE on trailing return:
         // Fakely instantiate the type used for checking and the type to check, and SFINAE on them.
          -> decltype(std::declval<CheckingType>()(std::declval<TypeToCheck>()), bool())
    {
        return true;
    }

    template <class Param>
    constexpr bool test_validity(...) { return false; } // Classic sink!

public:
    template <class TypeToCheck>
    constexpr bool operator()(const TypeToCheck& p) // Callable (Functor).
    {
        return test_validity<TypeToCheck>(int()); // Forward a type to check using SFINAE.
    }
};
```

Container for is_valid

C++17's if constexpr

- **if constexpr** keywords: permit to use conditional statements at compile-time.

```
template <class T>
std::string serialize(const T& obj)
{
    if constexpr (has_serialize(obj)) {
        // Discarded statement for an int!
        return obj.serialize();
    } else {
        return std::to_string(obj);
    }
}

serialize(42); // No compilation error in the discarded statement!
std::cout << serialize(b) << std::endl;                     // I am a B!
```

Conclusion

- Introspection definitely possible in C++:
  - Compile-time “only”...
  - … but zero runtime cost
  - Doesn’t work with inheritance.
- Improved with each new standard revisions
  - From a hack (C++98) to a well integrated toolkit (C++17)
  - Used in the STL
- Other directions:
  - C++ Static Reflection via template pack expansion (P0255R0)
  - Concepts TS

Thank you for your attention!

Any questions?

Who am I?

- 25 years old
- Male
- French
- C++ Developer during the day
- C++ Hobbyist during the night
- jguegant@gmail.com
- http://jguegant.github.io/blogs/tech/sfinae-introduction.html

```cpp
template <class T> void foo(typename T::x t);

template <class T> void foo(T t, typename T::x* t = nullptr);

template <class T> void foo(T t);

template <class T> void foo(const T& t);

void foo(int t);

void foo(double t);

void foo(...);


foo(42); // Which overload?
```

Example 3: Test - SFINAE - Best viable function

```
template <class T> void foo(typename T::x t);

template <class T> void foo(T t, typename T::x* t = nullptr);

template <class T> void foo(T t);

template <class T> void foo(const T& t);

void foo(int t);

void foo(double t);

void foo(...);


foo(42); // Which overload?
```

~~`template <class T> void foo(typename T::x t);`~~ `// Non-deduced`

~~`template <class T> void foo(T t, typename T::x* t = nullptr);`~~ `// SFINAE`

```
template <class T> void foo(T t);

template <class T> void foo(const T& t);

void foo(int t);

void foo(double t);

void foo(...);


foo(42); // Which overload?
```

Example 3: Test - SFINAE - Best viable function

```
template <class T> void foo(typename T::x t); // Non-deduced
template <class T> void foo(T t, typename T::x* t = nullptr); // SFINAE
template <class T> void foo(T t); // Rank 2 (rvalue int ranks Exact Match)
template <class T> void foo(const T& t); // Rank 2 (const int& ranks Exact Match)
void foo(int t); // Rank 1
void foo(double t); // Rank 3
void foo(...); // Rank 4

foo(42); // Which overload?
```

Example 3: Test - SFINAE - Best viable function

Today’s menu

1. Introspection in C++ you said?
2. SFINAE
3. A bit of C++ archeology
4. C++1x to our rescue
5. Blending time