Type Traits Without Compiler Intrinsics

The Promise of Static Reflection

ANDREI ZISSU

Who am I

- Andrei Zissu
- Worked in multiple industries over 2 decades, mostly in C++ and on Windows
- Tech Lead at Morphisec, provider of anti-ransomware based on Automatic Moving Target Defense (AMTD)
- WG21 member for the past 3 years, mainly involved in reflection and contracts
- Disclaimer: The views expressed in this talk are mine and mine only (hopefully yours too by the time we're done here)

Agenda

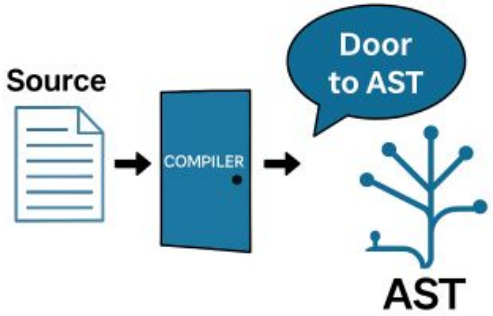
We can *probably* implement *all* type traits based on reflection

Reflection-based type traits are key to future decoupling between libraries and toolchains

A future where you are free to mix any compiler with any standard library implementation

Come with me on a journey to that future! 

Lay of the land



Intrinsic -> built-in, "magic“

Many standard concepts are built on top of type traits

Many type traits are built on top of compiler intrinsics

Libcxx

Accompanies clang

Libstd

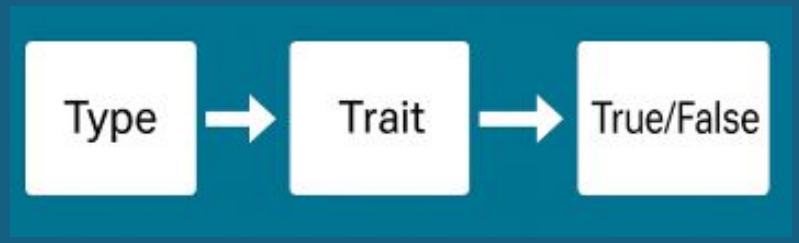
Accompanies gcc

MSVC STL

Accompanies MS Visual Studio

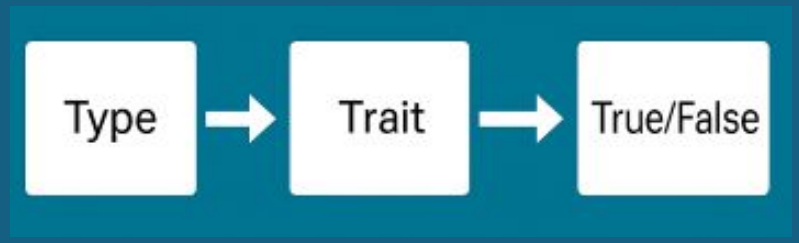
Introduction to Type Traits

```
ASM generation compiler returned: 0
Execution build compiler returned: 0
Program returned: 0
   Type is integral.
   Type is floating-point.
   Type is a pointer.
   Type is something else.
```



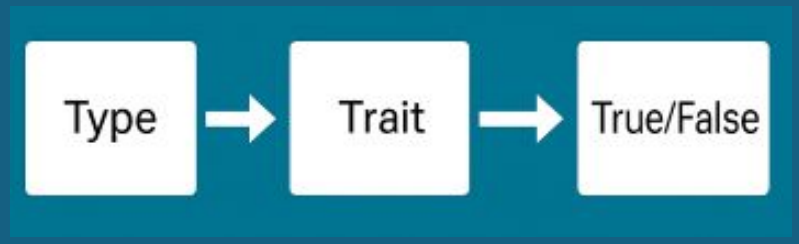

```cpp
#include <type_traits>
#include <iostream>

template <typename T>
void printTypeInfo() {
    if constexpr (std::is_integral_v<T>) {
        std::cout << "Type is integral.\n";
    } else if constexpr (std::is_floating_point_v<T>) {
        std::cout << "Type is floating-point.\n";
    } else if constexpr (std::is_pointer_v<T>) {
        std::cout << "Type is a pointer.\n";
    } else {
        std::cout << "Type is something else.\n";
    }
}

int main() {
    printTypeInfo<int>();           // Output: Type is integral.
    printTypeInfo<double>();        // Output: Type is floating-point.
    printTypeInfo<int*>();          // Output: Type is a pointer.
    printTypeInfo<std::string>();   // Output: Type is something else.
    return 0;
}
```

```
ASM generation compiler returned: 0
Execution build compiler returned: 0
Program returned: 0
   Type is integral.
   Type is floating-point.
   Type is a pointer.
   Type is something else.
```



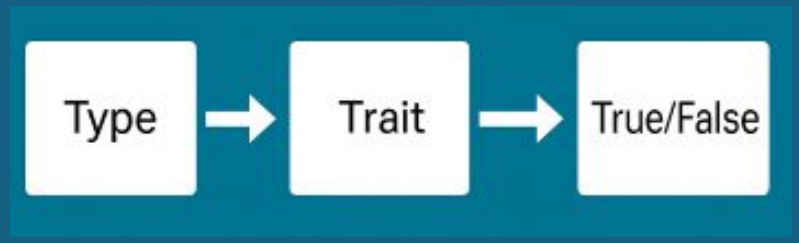

```cpp
#include <type_traits>
#include <iostream>

template <typename T>
void printTypeInfo() {
    if constexpr (std::is_integral_v<T>) {
        std::cout << "Type is integral.\n";
    } else if constexpr (std::is_floating_point_v<T>) {
        std::cout << "Type is floating-point.\n";
    } else if constexpr (std::is_pointer_v<T>) {
        std::cout << "Type is a pointer.\n";
    } else {
        std::cout << "Type is something else.\n";
    }
}

int main() {
    printTypeInfo<int>();           // Output: Type is integral.
    printTypeInfo<double>();        // Output: Type is floating-point.
    printTypeInfo<int*>();          // Output: Type is a pointer.
    printTypeInfo<std::string>();   // Output: Type is something else.
    return 0;
}
```

```
ASM generation compiler returned: 0
Execution build compiler returned: 0
Program returned: 0
   Type is integral.
   Type is floating-point.
   Type is a pointer.
   Type is something else.
```



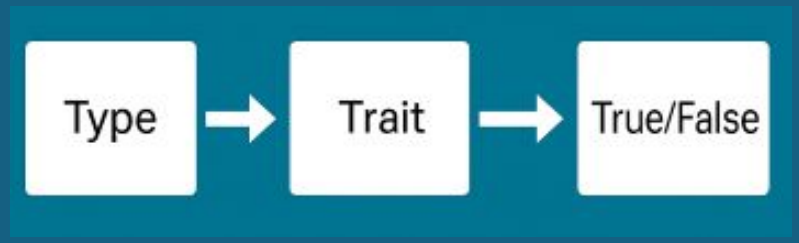

```cpp
#include <type_traits>
#include <iostream>

template <typename T>
void printTypeInfo() {
    if constexpr (std::is_integral_v<T>) {
        std::cout << "Type is integral.\n";
    } else if constexpr (std::is_floating_point_v<T>) {
        std::cout << "Type is floating-point.\n";
    } else if constexpr (std::is_pointer_v<T>) {
        std::cout << "Type is a pointer.\n";
    } else {
        std::cout << "Type is something else.\n";
    }
}

int main() {
    printTypeInfo<int>();           // Output: Type is integral.
    printTypeInfo<double>();        // Output: Type is floating-point.
    printTypeInfo<int*>();          // Output: Type is a pointer.
    printTypeInfo<std::string>();   // Output: Type is something else.
    return 0;
}
```

```
ASM generation compiler returned: 0
Execution build compiler returned: 0
Program returned: 0
   Type is integral.
   Type is floating-point.
   Type is a pointer.
   Type is something else.
```



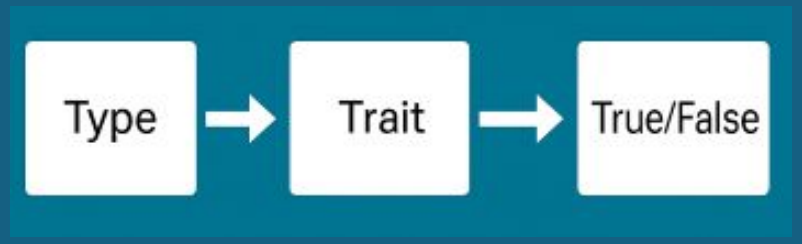

```cpp
#include <type_traits>
#include <iostream>

template <typename T>
void printTypeInfo() {
    if constexpr (std::is_integral_v<T>) {
        std::cout << "Type is integral.\n";
    } else if constexpr (std::is_floating_point_v<T>) {
        std::cout << "Type is floating-point.\n";
    } else if constexpr (std::is_pointer_v<T>) {
        std::cout << "Type is a pointer.\n";
    } else {
        std::cout << "Type is something else.\n";
    }
}

int main() {
    printTypeInfo<int>();           // Output: Type is integral.
    printTypeInfo<double>();        // Output: Type is floating-point.
    printTypeInfo<int*>();          // Output: Type is a pointer.
    printTypeInfo<std::string>();   // Output: Type is something else.
    return 0;
}
```

```
ASM generation compiler returned: 0
Execution build compiler returned: 0
Program returned: 0
   Type is integral.
   Type is floating-point.
   Type is a pointer.
   Type is something else.
```



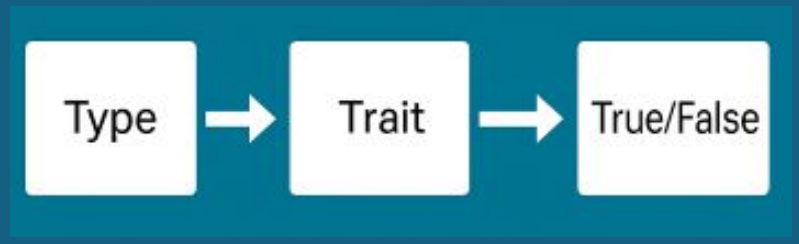

```cpp
#include <type_traits>
#include <iostream>

template <typename T>
void printTypeInfo() {
    if constexpr (std::is_integral_v<T>) {
        std::cout << "Type is integral.\n";
    } else if constexpr (std::is_floating_point_v<T>) {
        std::cout << "Type is floating-point.\n";
    } else if constexpr (std::is_pointer_v<T>) {
        std::cout << "Type is a pointer.\n";
    } else {
        std::cout << "Type is something else.\n";
    }
}

int main() {
    printTypeInfo<int>();           // Output: Type is integral.
    printTypeInfo<double>();        // Output: Type is floating-point.
    printTypeInfo<int*>();          // Output: Type is a pointer.
    printTypeInfo<std::string>();   // Output: Type is something else.
    return 0;
}
```

```
ASM generation compiler returned: 0
Execution build compiler returned: 0
Program returned: 0
   Integral value: 42
   Floating-point value: 3.14
```



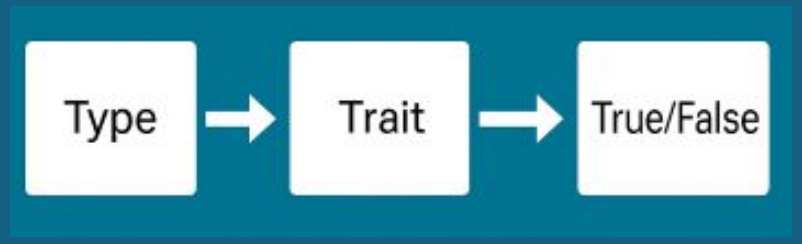

```cpp
#include <type_traits>
#include <iostream>

// Function enabled if T is an integral type
template<typename T>
typename std::enable_if<std::is_integral<T>::value>::type
print(T value) {
    std::cout << "Integral value: " << value << std::endl;
}

// Function enabled if T is a floating-point type
template<typename T>
typename std::enable_if<std::is_floating_point<T>::value>::type
print(T value) {
    std::cout << "Floating-point value: " << value << std::endl;
}

int main() {
    print(42);      // Outputs: Integral value: 42
    print(3.14);    // Outputs: Floating-point value: 3.14
    return 0;
}
```

```
ASM generation compiler returned: 0
Execution build compiler returned: 0
Program returned: 0
   Integral value: 42
   Floating-point value: 3.14
```



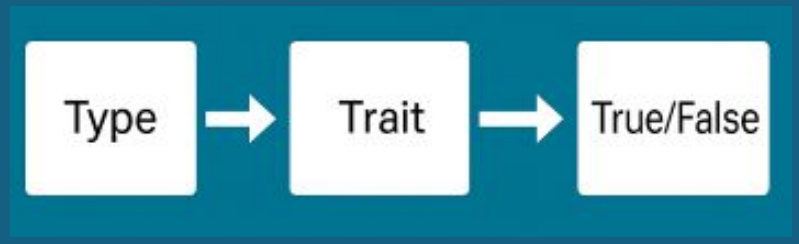

```cpp
#include <type_traits>
#include <iostream>

// Function enabled if T is an integral type
template<typename T>
typename std::enable_if<std::is_integral<T>::value>::type
print(T value) {
    std::cout << "Integral value: " << value << std::endl;
}

// Function enabled if T is a floating-point type
template<typename T>
typename std::enable_if<std::is_floating_point<T>::value>::type
print(T value) {
    std::cout << "Floating-point value: " << value << std::endl;
}

int main() {
    print(42);     // Outputs: Integral value: 42
    print(3.14);   // Outputs: Floating-point value: 3.14
    return 0;
}
```

```
ASM generation compiler returned: 0
Execution build compiler returned: 0
Program returned: 0
   Integral value: 42
   Floating-point value: 3.14
```



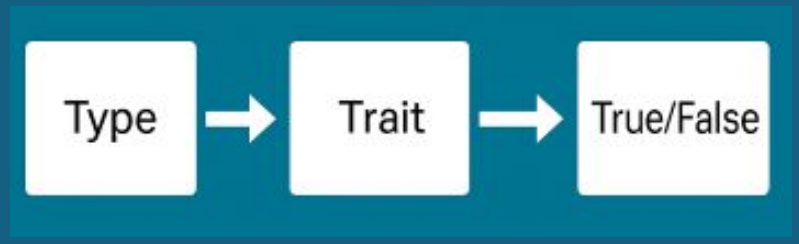

```cpp
#include <type_traits>
#include <iostream>

// Function enabled if T is an integral type
template<typename T>
typename std::enable_if<std::is_integral<T>::value>::type
print(T value) {
    std::cout << "Integral value: " << value << std::endl;
}

// Function enabled if T is a floating-point type
template<typename T>
typename std::enable_if<std::is_floating_point<T>::value>::type
print(T value) {
    std::cout << "Floating-point value: " << value << std::endl;
}

int main() {
    print(42);     // Outputs: Integral value: 42
    print(3.14);   // Outputs: Floating-point value: 3.14
    return 0;
}
```

```
ASM generation compiler returned: 0
Execution build compiler returned: 0
Program returned: 0
   Integral value: 42
   Floating-point value: 3.14
```



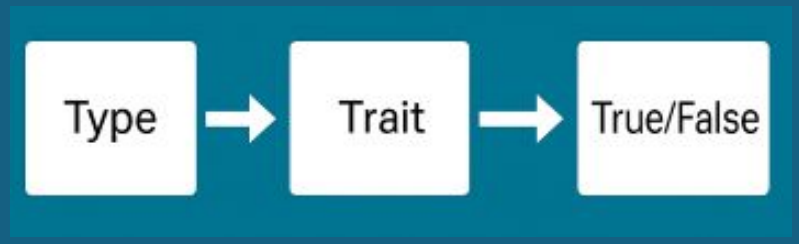

```cpp
#include <type_traits>
#include <iostream>

// Function enabled if T is an integral type
template<typename T>
typename std::enable_if<std::is_integral<T>::value>::type
print(T value) {
    std::cout << "Integral value: " << value << std::endl;
}

// Function enabled if T is a floating-point type
template<typename T>
typename std::enable_if<std::is_floating_point<T>::value>::type
print(T value) {
    std::cout << "Floating-point value: " << value << std::endl;
}

int main() {
    print(42);     // Outputs: Integral value: 42
    print(3.14);   // Outputs: Floating-point value: 3.14
    return 0;
}
```

```
ASM generation compiler returned: 0
Execution build compiler returned: 0
Program returned: 0
   Integral value: 42
   Floating-point value: 3.14
```



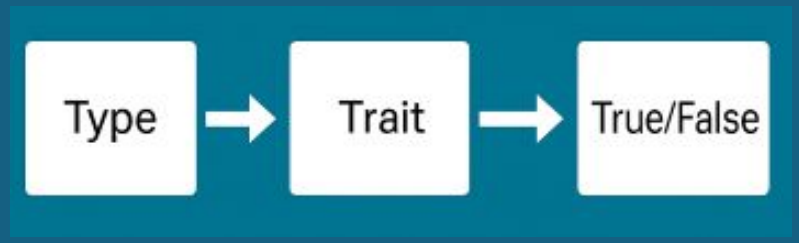

```cpp
#include <type_traits>
#include <iostream>

// Function enabled if T is an integral type
template<typename T>
typename std::enable_if<std::is_integral<T>::value>::type
print(T value) {
    std::cout << "Integral value: " << value << std::endl;
}

// Function enabled if T is a floating-point type
template<typename T>
typename std::enable_if<std::is_floating_point<T>::value>::type
print(T value) {
    std::cout << "Floating-point value: " << value << std::endl;
}

int main() {
    print(42);     // Outputs: Integral value: 42
    print(3.14);   // Outputs: Floating-point value: 3.14
    return 0;
}
```

```
ASM generation compiler returned: 0
Execution build compiler returned: 0
Program returned: 0
   Integral value: 42
   Floating-point value: 3.14
```




```cpp
#include <type_traits>
#include <iostream>

// Function enabled if T is an integral type
template<typename T>
typename std::enable_if<std::is_integral<T>::value>::type
print(T value) {
    std::cout << "Integral value: " << value << std::endl;
}

// Function enabled if T is a floating-point type
template<typename T>
typename std::enable_if<std::is_floating_point<T>::value>::type
print(T value) {
    std::cout << "Floating-point value: " << value << std::endl;
}

int main() {
    print(42);      // Outputs: Integral value: 42
    print(3.14);    // Outputs: Floating-point value: 3.14
    return 0;
}
```

```
ASM generation compiler returned: 0
Execution build compiler returned: 0
Program returned: 0
   Integral value: 42
   Floating-point value: 3.14
```




```cpp
#include <type_traits>
#include <iostream>

// Function enabled if T is an integral type
template<typename T>
typename std::enable_if_t<std::is_integral_v<T>>
print(T value) {
    std::cout << "Integral value: " << value << std::endl;
}

// Function enabled if T is a floating-point type
template<typename T>
typename std::enable_if_t<std::is_floating_point_v<T>>
print(T value) {
    std::cout << "Floating-point value: " << value << std::endl;
}

int main() {
    print(42);      // Outputs: Integral value: 42
    print(3.14);    // Outputs: Floating-point value: 3.14
    return 0;
}
```

Boolean Query Traits

- is_const
- is_lvalue_reference
- is_integral
- is_constructible
- is_same
- is_convertible

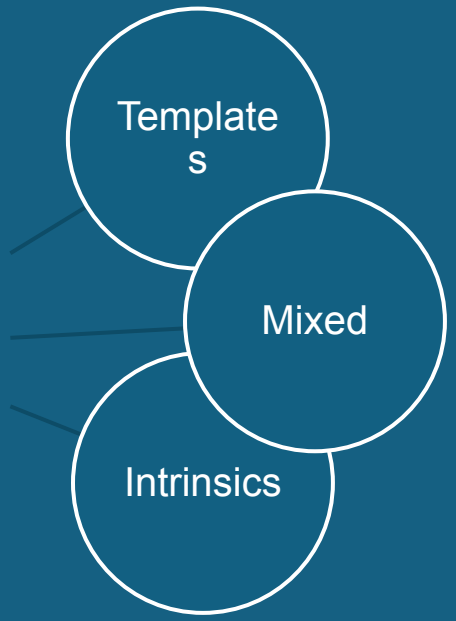
Type Transformations

remove_cv

add_lvalue_reference

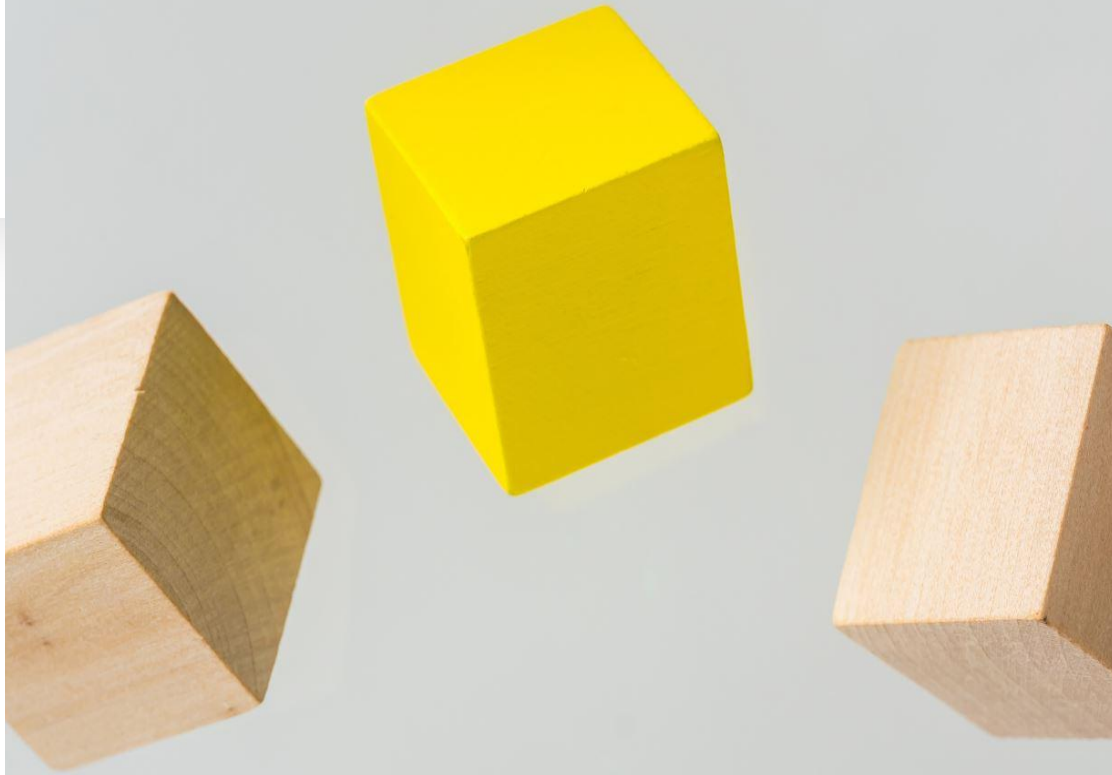
make_signed

Overall Implementation Strategies



Template-based Implementations

Basic Building Blocks: integral_constant, bool_constant enable_if



std::integral_constant

```
template<class T, T v>
struct integral_constant
{
    static constexpr T value = v;
```

std::true_type
std::false_type

```cpp
template<class T, T v>
struct integral_constant
{
    static constexpr T value = v;
```

```cpp
typedef integral_constant<bool, true> true_type;
typedef integral_constant<bool, false> false_type;
```

std::bool_constant

```cpp
template<class T, T v>
struct integral_constant
{
    static constexpr T value = v;
```

```cpp
typedef integral_constant<bool, true> true_type;
typedef integral_constant<bool, false> false_type;
```

```cpp
template< bool B >
using bool_constant = integral_constant<bool, B>;
```

std::enable_if

```
template<bool B, class T = void>
struct enable_if {};

template<class T>
struct enable_if<true, T> { typedef T type; };
```

std::enable_if

```cpp
template<bool B, class T = void>
struct enable_if {};

template<class T>
struct enable_if<true, T> { typedef T type; };
```

```cpp
template< bool B, class T = void >
using enable_if_t = typename enable_if<B,T>::type;
```

std::enable_if

```cpp
template<bool B, class T = void>
struct enable_if {};

template<class T>
struct enable_if<true, T> { typedef T type; };
```

```cpp
template< bool B, class T = void >
using enable_if_t = typename enable_if<B,T>::type;
```

Substitution Failure Is Not An Error - SFINAE

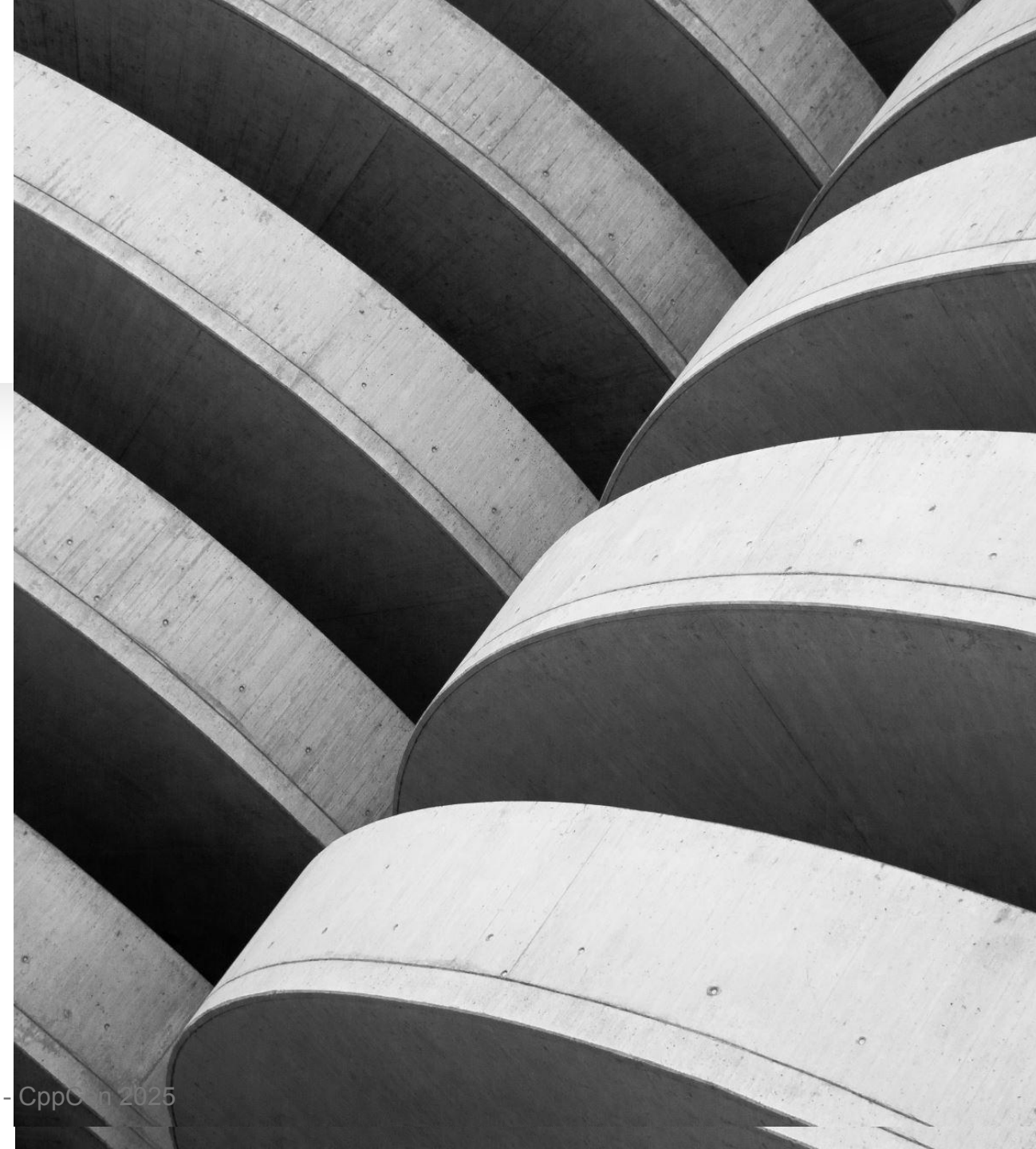
```
ASM generation compiler returned: 0
Execution build compiler returned: 0
Program returned: 0
   Integral value: 42
   Floating-point value: 3.14
```

```cpp
#include <type_traits>
#include <iostream>

// Function enabled if T is an integral type
template<typename T>
std::enable_if_t<std::is_integral_v<T>>
print(T value) {
    std::cout << "Integral value: " << value << std::endl;
}

// Function enabled if T is a floating-point type
template<typename T>
std::enable_if_t<std::is_floating_point_v<T>>
print(T value) {
    std::cout << "Floating-point value: " << value << std::endl;
}

int main() {
    print(42);       // Outputs: Integral value: 42
    print(3.14);     // Outputs: Floating-point value: 3.14
    return 0;
}
```

Implementing helper variable templates

```
template< class T >
constexpr bool is_integral_v = is_integral<T>::value;
```

Struct Templates



std::is_null_pointer (gcc)

```cpp
template<typename _Tp>
struct is_null_pointer
  : public std::false_type { };

template<>
struct is_null_pointer<std::nullptr_t>
  : public std::true_type { };

template<>
struct is_null_pointer<const std::nullptr_t>
  : public std::true_type { };

template<>
struct is_null_pointer<volatile std::nullptr_t>
  : public std::true_type { };

template<>
struct is_null_pointer<const volatile std::nullptr_t>
  : public std::true_type { };
```

std::is_null_pointer (gcc)

```
template<typename _Tp>
struct is_null_pointer
  : public std::false_type { };

template<>
struct is_null_pointer<std::nullptr_t>
  : public std::true_type { };

template<>
struct is_null_pointer<const std::nullptr_t>
  : public std::true_type { };

template<>
struct is_null_pointer<volatile std::nullptr_t>
  : public std::true_type { };

template<>
struct is_null_pointer<const volatile std::nullptr_t>
  : public std::true_type { };
```

std::is_signed (gcc)

```
template<typename _Tp, bool = is_arithmetic<_Tp>::value>
struct __is_signed_helper
  : public false_type { };

template<typename _Tp>
struct __is_signed_helper<_Tp, true>
  : public __bool_constant<_Tp(-1) < _Tp(0)> { };

template<typename _Tp>
struct is_signed
  : public __is_signed_helper<_Tp>::type { };
```

std::is_signed (gcc)

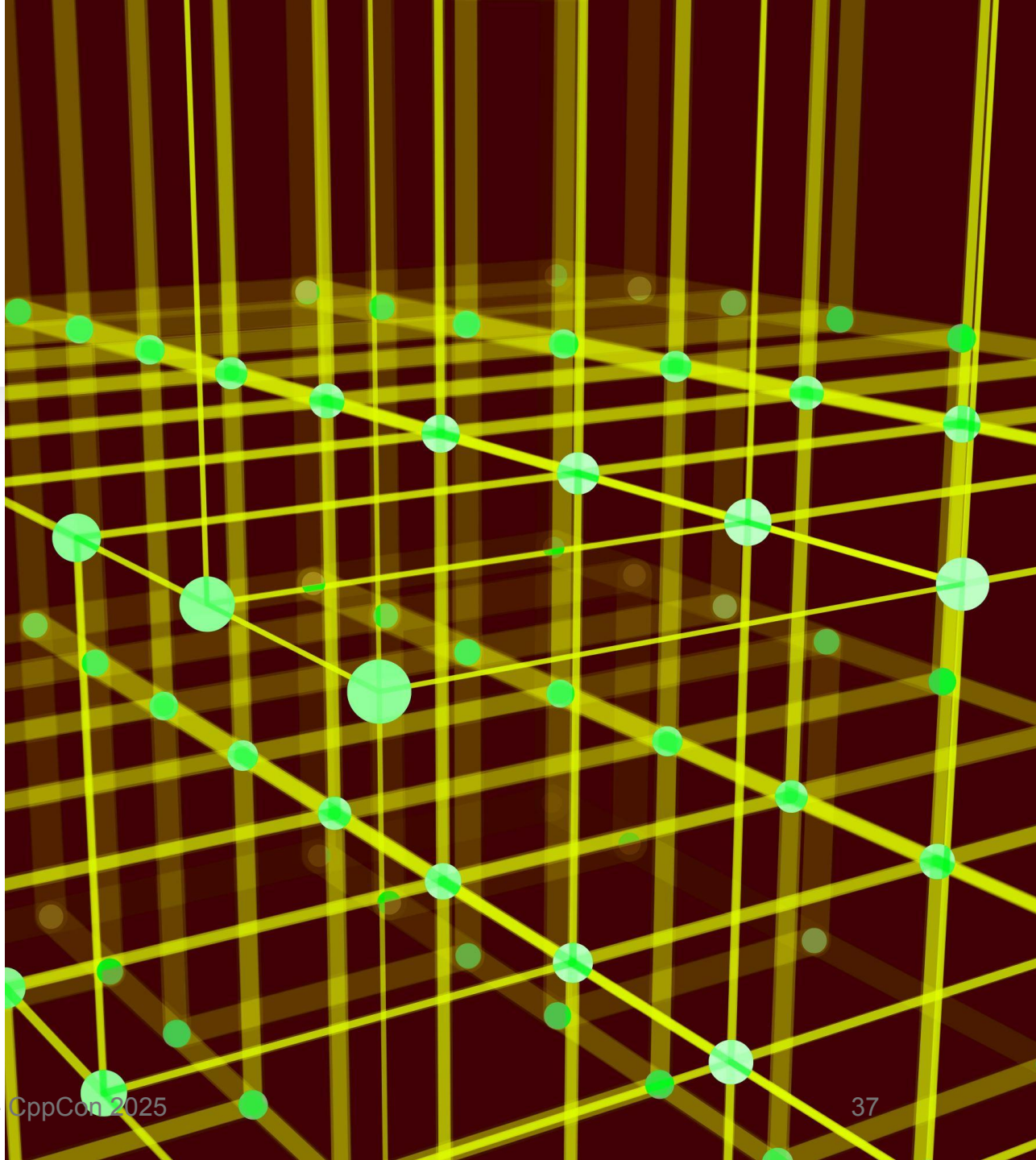
```cpp
template<typename _Tp, bool = is_arithmetic<_Tp>::value>
struct __is_signed_helper
  : public false_type { };

template<typename _Tp>
struct __is_signed_helper<_Tp, true>
  : public __bool_constant<_Tp(-1) < _Tp(0)> { };

template<typename _Tp>
struct is_signed
  : public __is_signed_helper<_Tp>::type { };
```

std::is_signed (gcc)

```
template<typename _Tp, bool = is_arithmetic<_Tp>::value>
struct __is_signed_helper
  : public false_type { };

template<typename _Tp>
struct __is_signed_helper<_Tp, true>
  : public __bool_constant<_Tp(-1) < _Tp(0)> { };

template<typename _Tp>
struct is_signed
  : public __is_signed_helper<_Tp>::type { };
```

Variable Templates

```
template <typename _Tp>
  inline constexpr bool is_signed_v = is_signed<_Tp>::value;
template <typename _Tp>
  inline constexpr bool is_unsigned_v = is_unsigned<_Tp>::value;
```



std::is_const (msvc)

```
_EXPORT_STD template <class>
constexpr bool is_const_v = false;

template <class _Ty>
constexpr bool is_const_v<const _Ty> = true;

_EXPORT_STD template <class _Ty>
struct is_const : bool_constant<is_const_v<_Ty>> {};
```

std::is_const (msvc)

```
_EXPORT_STD template <class>
constexpr bool is_const_v = false;

template <class _Ty>
constexpr bool is_const_v<const _Ty> = true;

_EXPORT_STD template <class _Ty>
struct is_const : bool_constant<is_const_v<_Ty>> {};
```

Type Traits Implemented via Compiler Internals


Some code from the 3 big compilers/librarie
s

clang (std::is_const)

```
#if __has_builtin(__is_const)

template <class _Tp>
struct _LIBCPP_TEMPLATE_VIS _LIBCPP_NO_SPECIALIZATIONS is_const : _BoolConstant<__is_const(_Tp)> {};

#  if _LIBCPP_STD_VER >= 17
template <class _Tp>
_LIBCPP_NO_SPECIALIZATIONS inline constexpr bool is_const_v = __is_const(_Tp);
#  endif

#else

template <class _Tp>
struct _LIBCPP_TEMPLATE_VIS is_const : public false_type {};
template <class _Tp>
struct _LIBCPP_TEMPLATE_VIS is_const<_Tp const> : public true_type {};

#  if _LIBCPP_STD_VER >= 17
template <class _Tp>
inline constexpr bool is_const_v = is_const<_Tp>::value;
#  endif

#endif // __has_builtin(__is_const)
```

clang (std::is_enum)

```
template <class _Tp>
struct _LIBCPP_TEMPLATE_VIS _LIBCPP_NO_SPECIALIZATIONS is_enum : public integral_constant<bool, __is_enum(_Tp)> {};

#if _LIBCPP_STD_VER >= 17
template <class _Tp>
_LIBCPP_NO_SPECIALIZATIONS inline constexpr bool is_enum_v = __is_enum(_Tp);
#endif
```

gcc

```
/// is_enum
template<typename _Tp>
  struct is_enum
  : public __bool_constant<__is_enum(_Tp)>
  { };

/// is_union
template<typename _Tp>
  struct is_union
  : public __bool_constant<__is_union(_Tp)>
  { };

/// is_class
template<typename _Tp>
  struct is_class
  : public __bool_constant<__is_class(_Tp)>
  { };
```

gcc (std::is_reference)

```
  /// is_reference
#if _GLIBCXX_USE_BUILTIN_TRAIT(__is_reference)
  template<typename _Tp>
    struct is_reference
    : public __bool_constant<__is_reference(_Tp)>
    { };
#else
  template<typename _Tp>
    struct is_reference
    : public false_type
    { };

  template<typename _Tp>
    struct is_reference<_Tp&>
    : public true_type
    { };

  template<typename _Tp>
    struct is_reference<_Tp&&>
    : public true_type
    { };
#endif
```

msvc

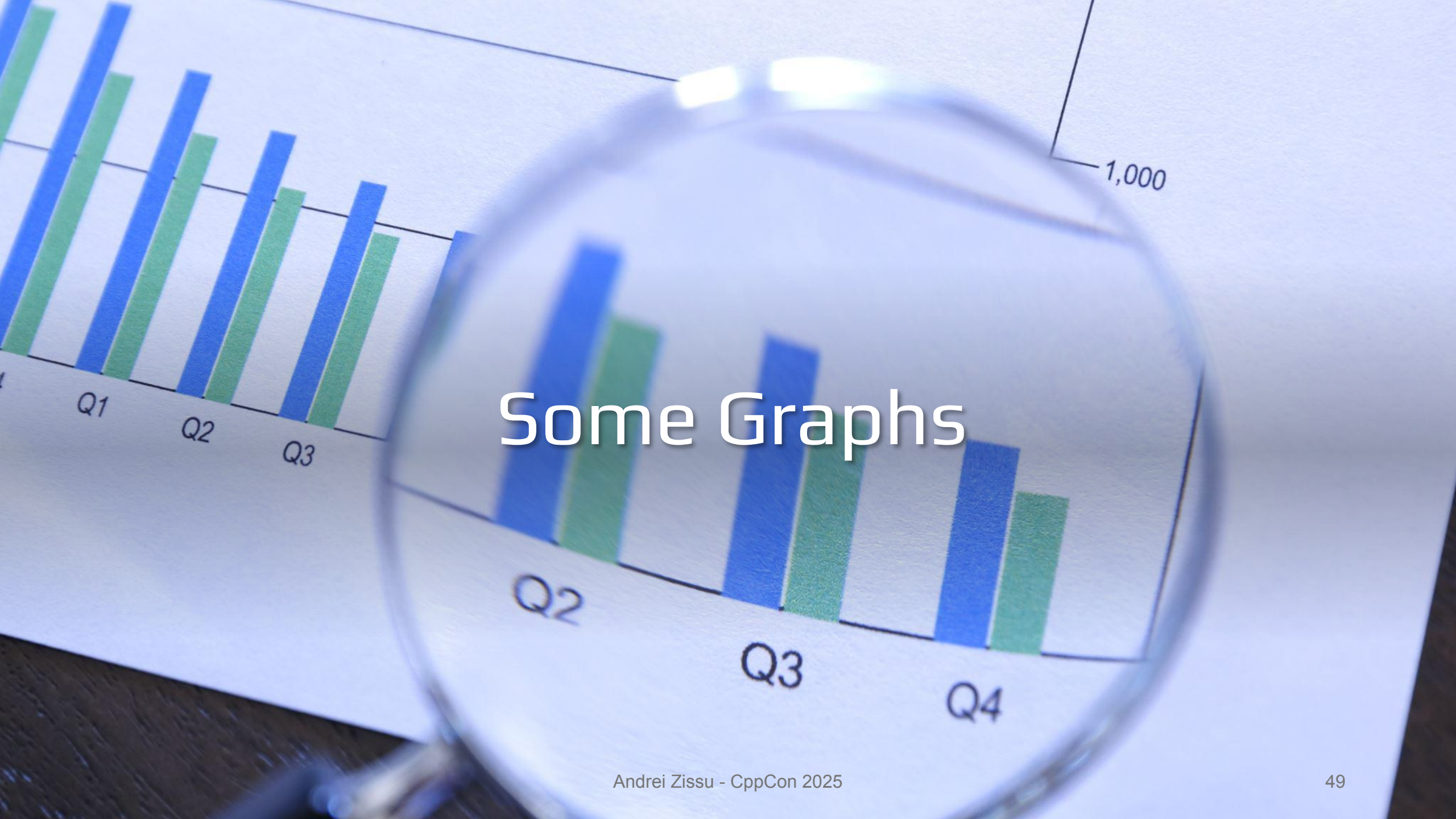
```
_EXPORT_STD template <class _Ty>
struct is_polymorphic : bool_constant<__is_polymorphic(_Ty)> {};

_EXPORT_STD template <class _Ty>
constexpr bool is_polymorphic_v = __is_polymorphic(_Ty);

_EXPORT_STD template <class _Ty>
struct is_abstract : bool_constant<__is_abstract(_Ty)> {}; // det

_EXPORT_STD template <class _Ty>
constexpr bool is_abstract_v = __is_abstract(_Ty);

_EXPORT_STD template <class _Ty>
struct is_final : bool_constant<__is_final(_Ty)> {}; // determine

_EXPORT_STD template <class _Ty>
constexpr bool is_final_v = __is_final(_Ty);
```

msvc (is_member_function_pointer)

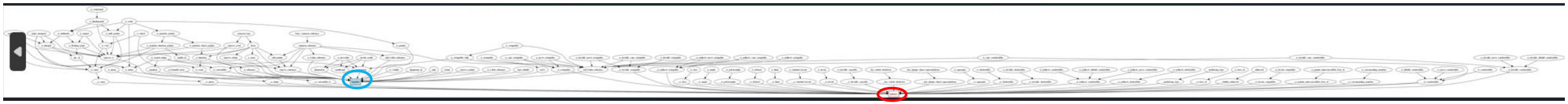
```
#ifdef __clang__
_EXPORT_STD template <class _Ty>
constexpr bool is_member_function_pointer_v = __is_member_function_pointer(_Ty);
#else // ^^^ Clang / Other vvv
_EXPORT_STD template <class _Ty>
constexpr bool is_member_function_pointer_v = _Is_memfunptr<remove_cv_t<_Ty>>::_Bool_type::value;
#endif // ^^^ Other ^^^

_EXPORT_STD template <class _Ty>
struct is_member_function_pointer : bool_constant<is_member_function_pointer_v<_Ty>> {};
```

And this is what _Is_memfunptr looks like...

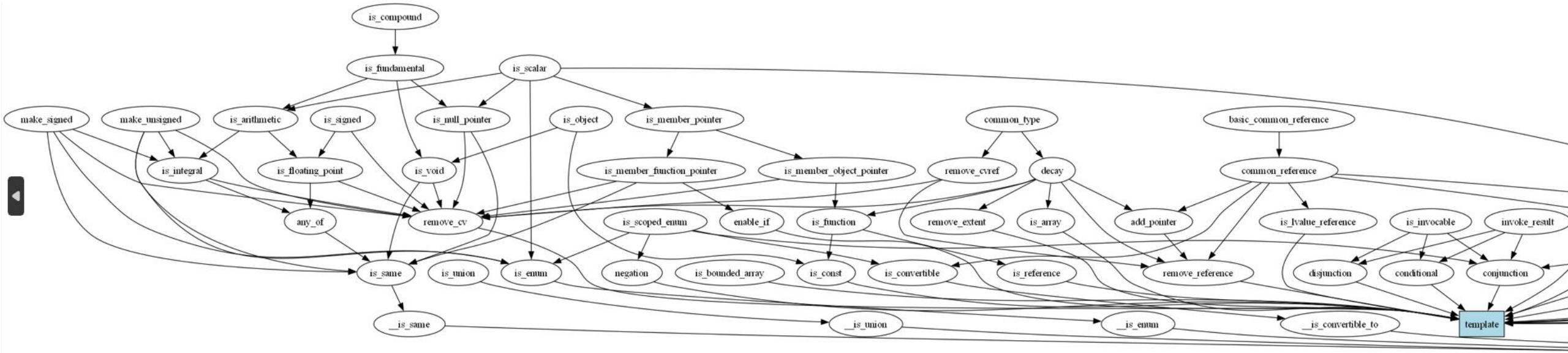
```
template <class _Ty>
struct _Is_memfunptr { // base class for member function pointer predicates
    using _Bool_type = false_type; // NB: members are user-visible via _Weak_types
};

#define _IS_MEMFUNPTR(CALL_OPT, CV_OPT, REF_OPT, NOEXCEPT_OPT)                          \
    template <class _Ret, class _Arg0, class... _Types>                                 \
    struct _Is_memfunptr<_Ret (CALL_OPT _Arg0::*)(_Types...) CV_OPT REF_OPT NOEXCEPT_OPT> \
        : _Arg_types<CV_OPT _Arg0*, _Types...> {                                        \
        using _Bool_type                                        = true_type;            \
        using _RESULT_TYPE_NAME _CXX17_DEPRECATE_ADAPTOR_TYPEDEFS = _Ret;               \
        using _Class_type                                       = _Arg0;                \
        using _Guide_type = enable_if<!is_same_v<int REF_OPT, int&&>, _Ret(_Types...)>; \
    };

_MEMBER_CALL_CV_REF_NOEXCEPT(_IS_MEMFUNPTR)
#undef _IS_MEMFUNPTR

#define _IS_MEMFUNPTR_ELLIPSIS(CV_REF_NOEXCEPT_OPT)                                 \
    template <class _Ret, class _Arg0, class... _Types>                             \
    struct _Is_memfunptr<_Ret (_Arg0::*)(_Types..., ...)                            \
            CV_REF_NOEXCEPT_OPT> { /* no calling conventions for ellipsis */        \
        using _Bool_type                                        = true_type;        \
        using _RESULT_TYPE_NAME _CXX17_DEPRECATE_ADAPTOR_TYPEDEFS = _Ret;           \
        using _Class_type                                       = _Arg0;            \
        using _Guide_type                                       = enable_if<false>; \
    };

_CLASS_DEFINE_CV_REF_NOEXCEPT(_IS_MEMFUNPTR_ELLIPSIS)
#undef _IS_MEMFUNPTR_ELLIPSIS

#if _HAS_CXX23 && !defined(__clang__) // TRANSITION, DevCom-10107077, Clang has not implemented Deducing this
#define _IS_MEMFUNPTR_EXPLICIT_THIS_GUIDES(CALL_OPT, CV_OPT, REF_OPT, NOEXCEPT_OPT) \
    template <class _Ret, class _Self, class... _Args>                              \
    struct _Is_memfunptr<_Ret(CALL_OPT*)(_Self, _Args...) NOEXCEPT_OPT> {           \
        using _Bool_type  = false_type;                                             \
        using _Guide_type = _Identity<_Ret(_Args...)>;                              \
    };
```



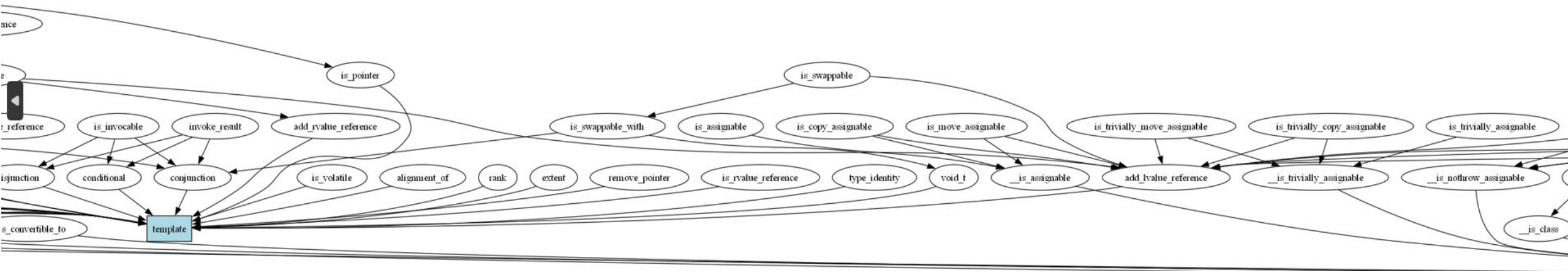
Some Graphs



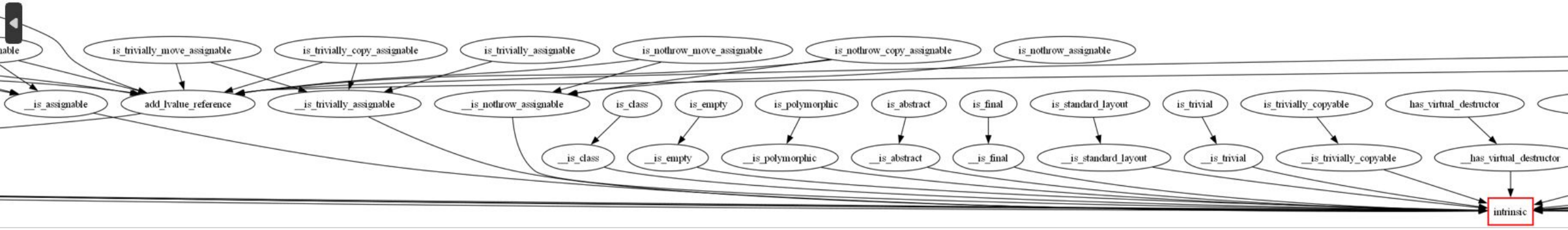
MSVC


is_compound
is_fundamental
is_scalar
make_signed
make_unsigned
is_arithmetic
is_signed
is_null_pointer
is_object
is_member_pointer
common_type
basic_common_reference
is_integral
is_floating_point
is_void
is_member_function_pointer
is_member_object_pointer
remove_cvref
decay
common_reference
any_of
remove_cv
is_scoped_enum
enable_if
is_function
remove_extent
is_array
add_pointer
is_lvalue_reference
is_invocable
invoke_result
is_same
is_union
is_enum
negation
is_bounded_array
is_const
is_convertible
is_reference
remove_reference
disjunction
conditional
conjunction
__is_same
__is_union
__is_enum
__is_convertible_to
template

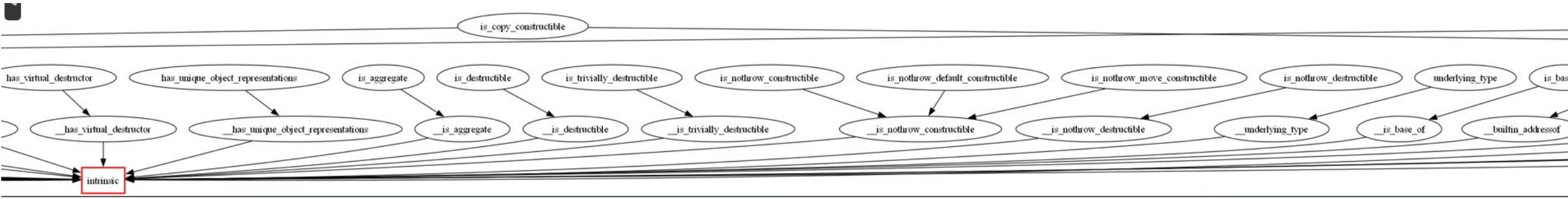
MSVC


is_pointer
is_swappable
is_invocable
invoke_result
add_rvalue_reference
is_swappable_with
is_assignable
is_copy_assignable
is_move_assignable
is_trivially_move_assignable
is_trivially_copy_assignable
is_trivially_assignable
conditional
conjunction
is_volatile
alignment_of
rank
extent
remove_pointer
is_rvalue_reference
type_identity
void_t
__is_assignable
add_lvalue_reference
__is_trivially_assignable
__is_nothrow_assignable
template

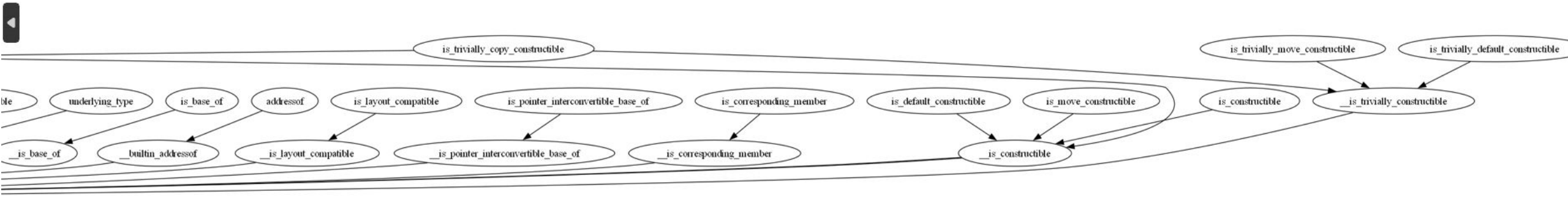
MSVC


is_trivially_move_assignable
is_trivially_copy_assignable
is_trivially_assignable
is_nothrow_move_assignable
is_nothrow_copy_assignable
is_nothrow_assignable
__is_assignable
add_lvalue_reference
__is_trivially_assignable
__is_nothrow_assignable
is_class
is_empty
is_polymorphic
is_abstract
is_final
is_standard_layout
is_trivial
is_trivially_copyable
has_virtual_destructor
__is_class
__is_empty
__is_polymorphic
__is_abstract
__is_final
__is_standard_layout
__is_trivial
__is_trivially_copyable
__has_virtual_destructor
intrinsic

MSVC

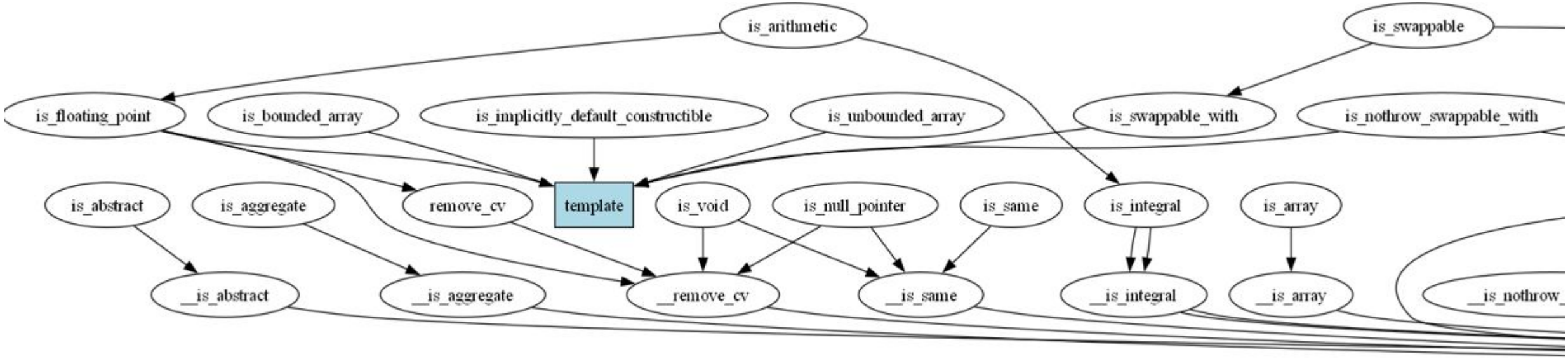

is_copy_constructible
has_virtual_destructor
has_unique_object_representations
is_aggregate
is_destructible
is_trivially_destructible
is_nothrow_constructible
is_nothrow_default_constructible
is_nothrow_move_constructible
is_nothrow_destructible
underlying_type
__has_virtual_destructor
__has_unique_object_representations
__is_aggregate
__is_destructible
__is_trivially_destructible
__is_nothrow_constructible
__is_nothrow_destructible
__underlying_type
__is_base_of
__builtin_addressof
intrinsic

MSVC

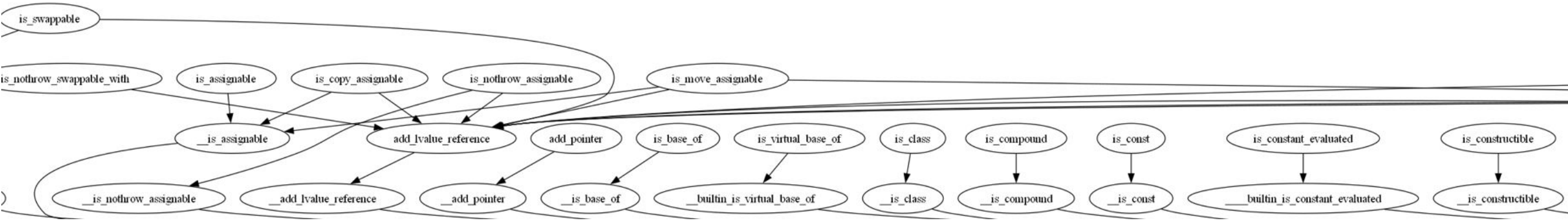

is_trivially_copy_constructible
is_trivially_move_constructible
is_trivially_default_constructible
underlying_type
is_base_of
addressof
is_layout_compatible
is_pointer_interconvertible_base_of
is_corresponding_member
is_default_constructible
is_move_constructible
is_constructible
__is_trivially_constructible
__is_base_of
__builtin_addressof
__is_layout_compatible
__is_pointer_interconvertible_base_of
__is_corresponding_member
__is_constructible

MSVC

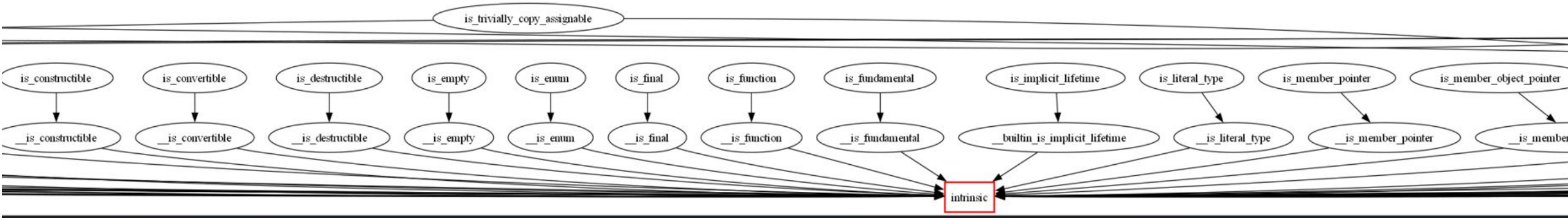
CLANG (libcxx)



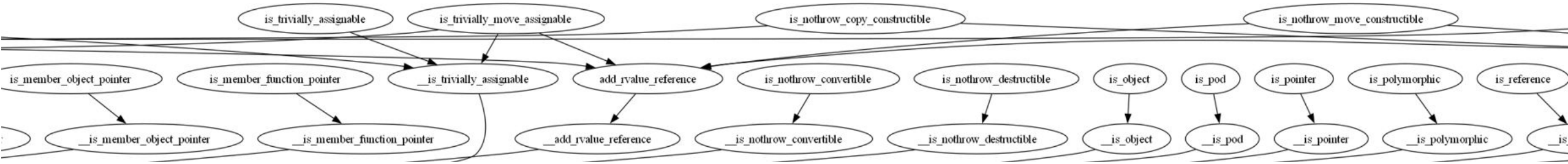
CLANG (libcxx)



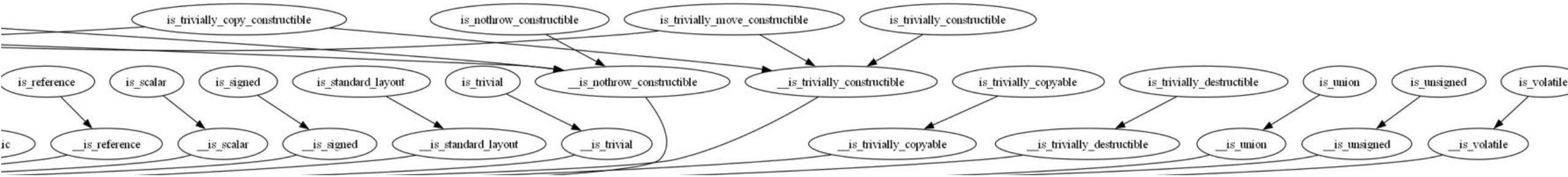
CLANG (libcxx)



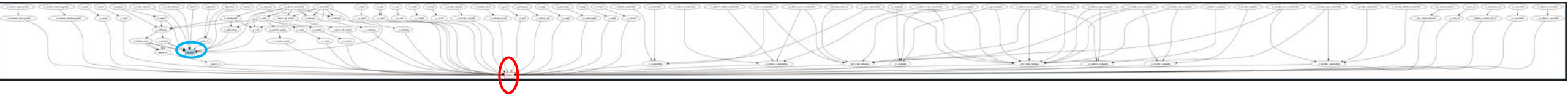
CLANG (libcxx)



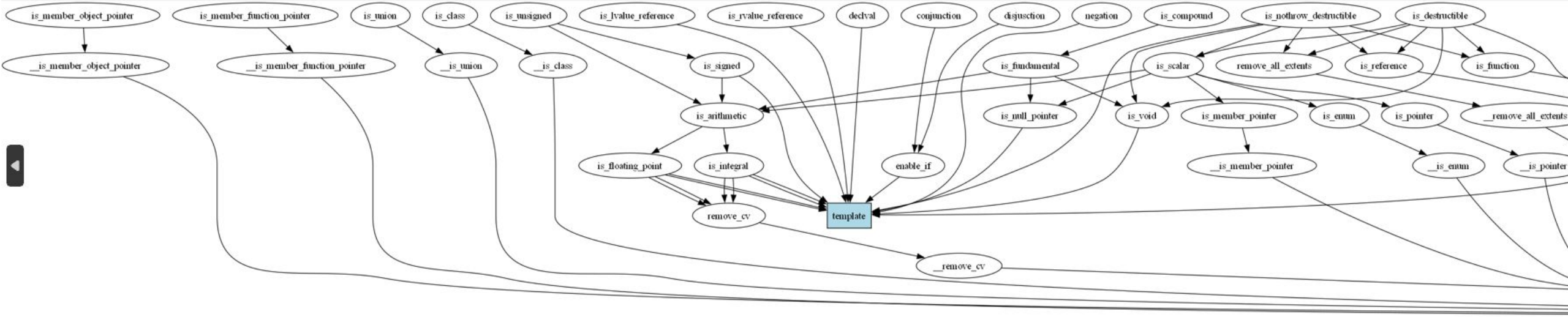
CLANG (libcxx)



CLANG (libcxx)

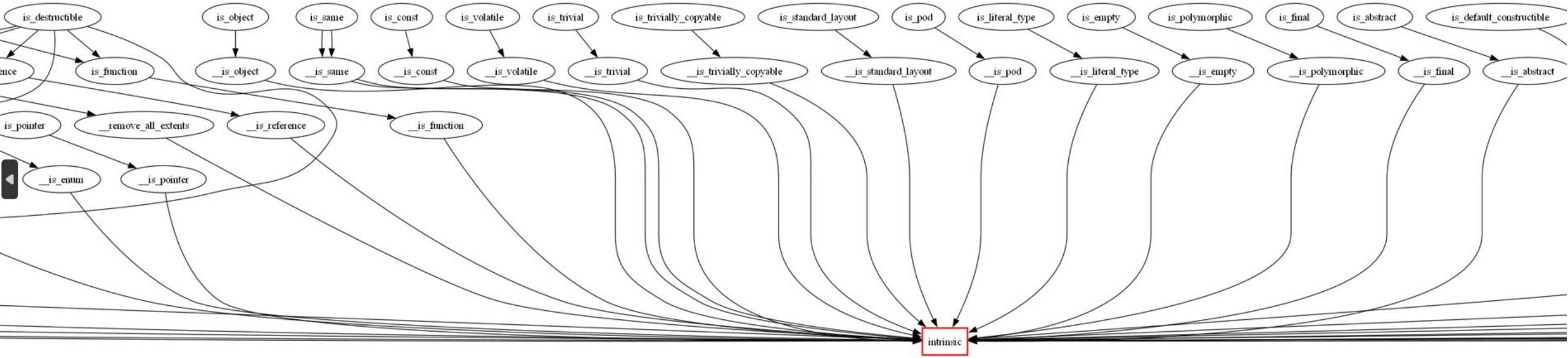


GCC (libstd)


is_member_object_pointer
is_member_function_pointer
is_union
is_class
is_unsigned
is_lvalue_reference
is_rvalue_reference
declval
conjunction
disjunction
negation
is_compound
is_nothrow_destructible
is_destructible
__is_member_object_pointer
__is_member_function_pointer
__is_union
__is_class
is_signed
is_fundamental
is_scalar
remove_all_extents
is_reference
is_function
is_arithmetic
is_null_pointer
is_void
is_member_pointer
is_enum
is_pointer
__remove_all_extents
is_floating_point
is_integral
enable_if
__is_member_pointer
__is_enum
__is_pointer
remove_cv
template
__remove_cv

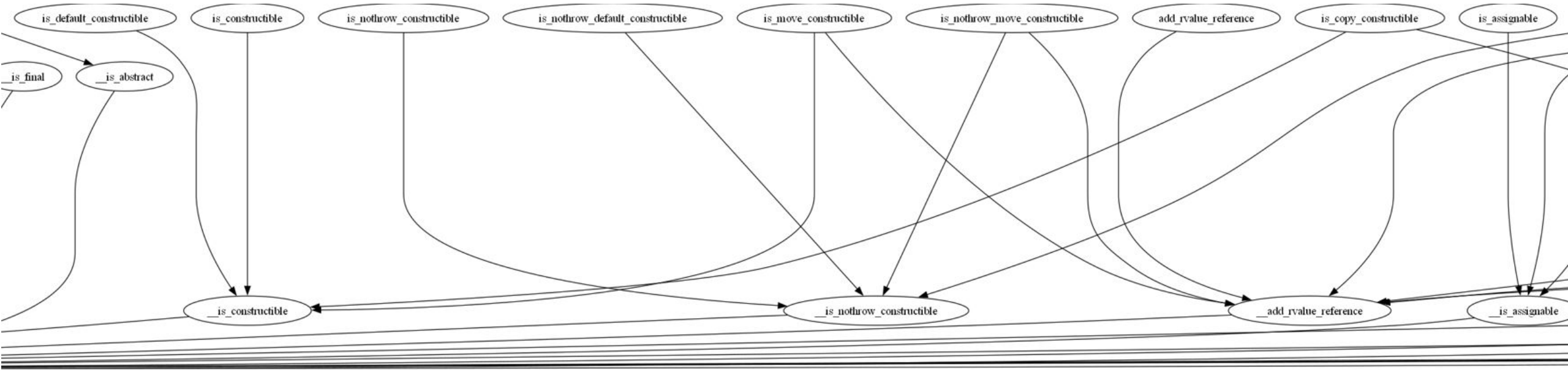
GCC (libstd)


GCC


is_destructible
is_object
is_same
is_const
is_volatile
is_trivial
is_trivially_copyable
is_standard_layout
is_pod
is_literal_type
is_empty
is_polymorphic
is_final
is_abstract
is_default_constructible
is_function
__is_object
__is_same
__is_const
__is_volatile
__is_trivial
__is_trivially_copyable
__is_standard_layout
__is_pod
__is_literal_type
__is_empty
__is_polymorphic
__is_final
__is_abstract
is_pointer
__remove_all_extents
__is_reference
__is_function
__is_enum
__is_pointer
intrinsic

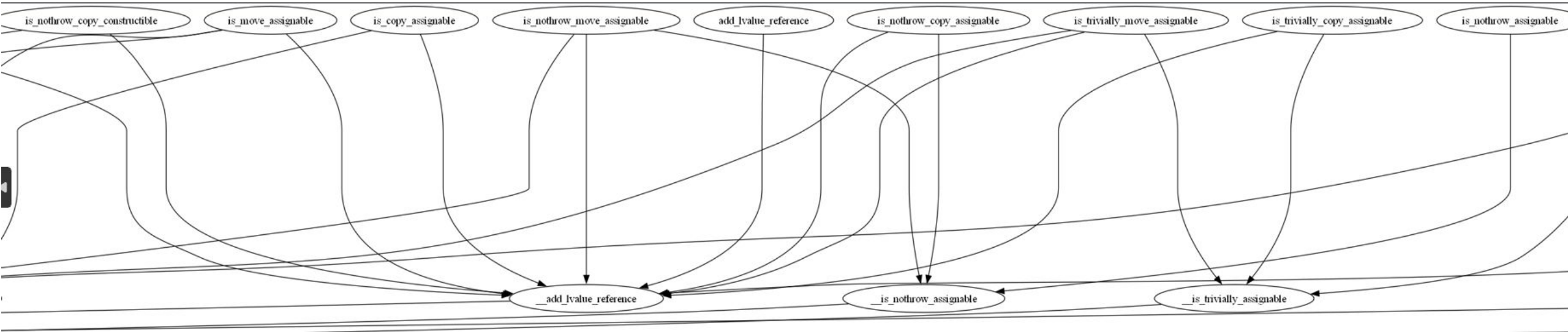
GCC (libstd)


GCC


is_default_constructible
is_constructible
is_nothrow_constructible
is_nothrow_default_constructible
is_move_constructible
is_nothrow_move_constructible
add_rvalue_reference
is_copy_constructible
is_assignable
__is_final
__is_abstract
__is_constructible
__is_nothrow_constructible
__add_rvalue_reference
__is_assignable

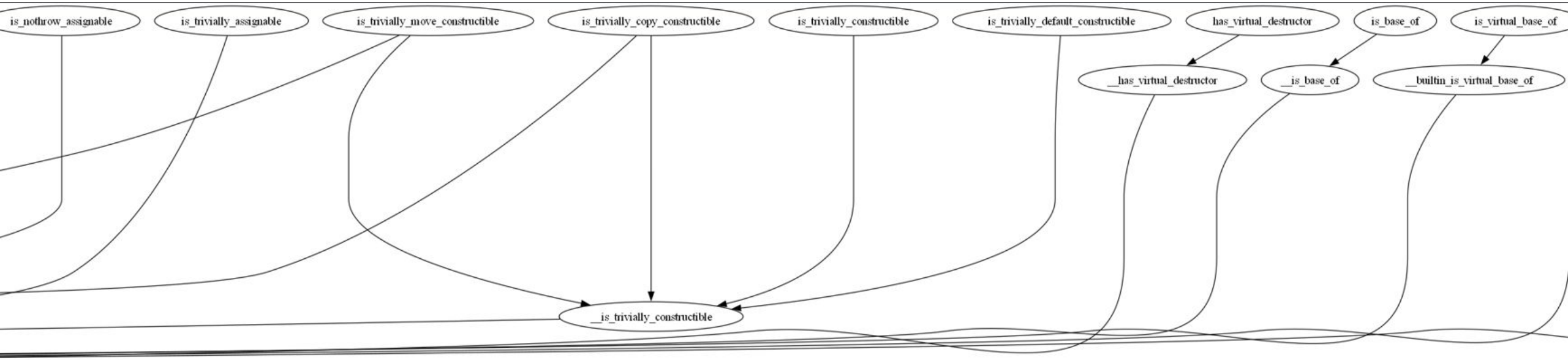
GCC (libstd)


GCC

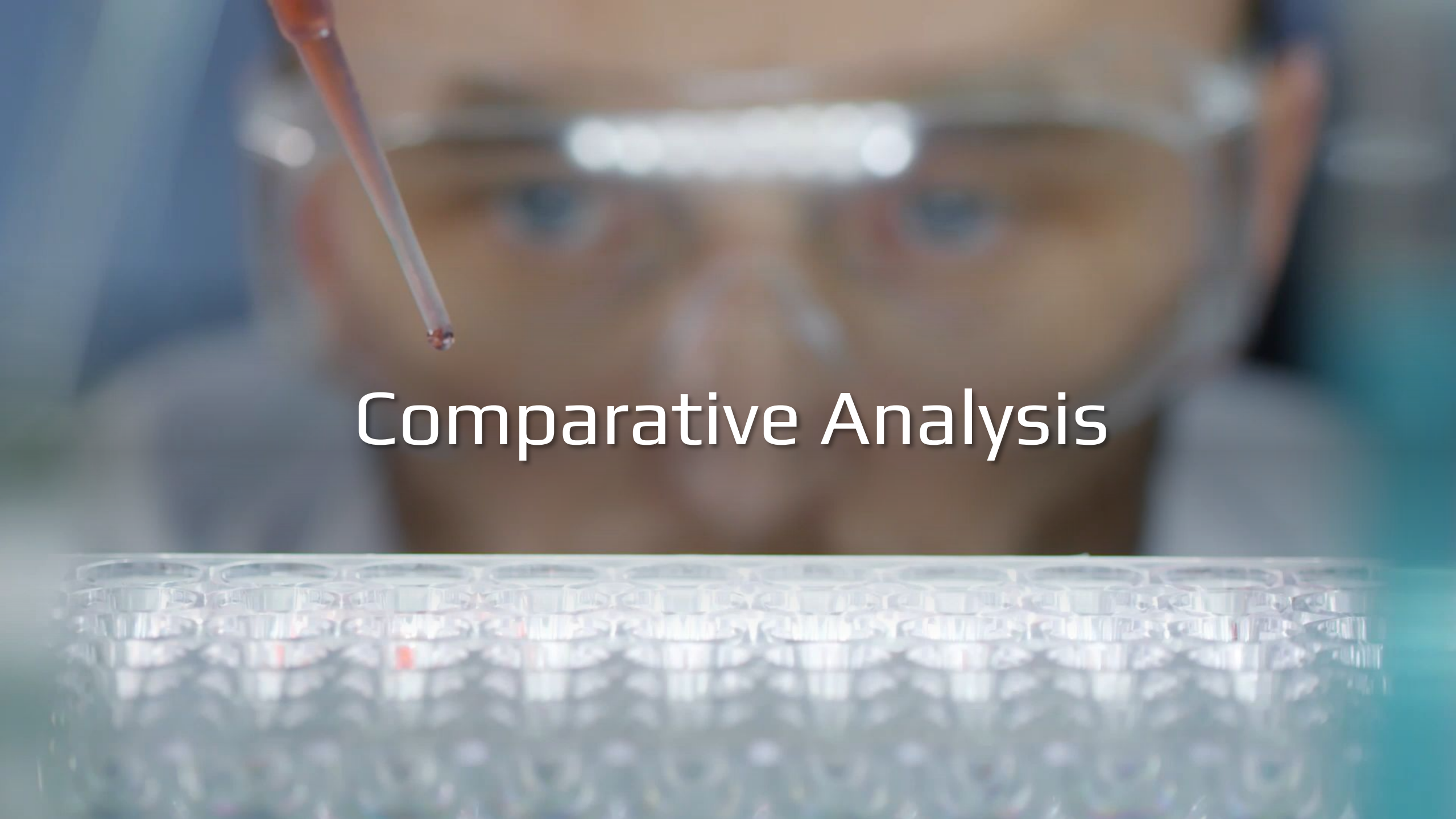

is_nothrow_copy_constructible
is_move_assignable
is_copy_assignable
is_nothrow_move_assignable
add_lvalue_reference
is_nothrow_copy_assignable
is_trivially_move_assignable
is_trivially_copy_assignable
is_nothrow_assignable
__add_lvalue_reference
__is_nothrow_assignable
__is_trivially_assignable

GCC (libstd)

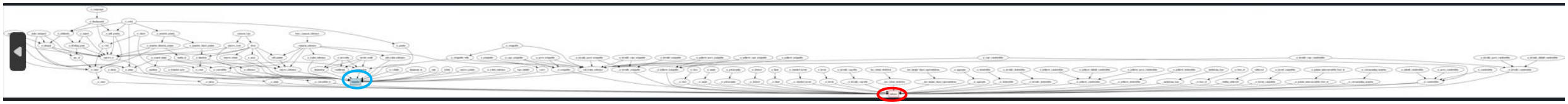

GCC


is_nothrow_assignable
is_trivially_assignable
is_trivially_move_constructible
is_trivially_copy_constructible
is_trivially_constructible
is_trivially_default_constructible
has_virtual_destructor
is_base_of
is_virtual_base_of
__has_virtual_destructor
__is_base_of
__builtin_is_virtual_base_of
__is_trivially_constructible

GCC (libstd)



Comparative Analysis

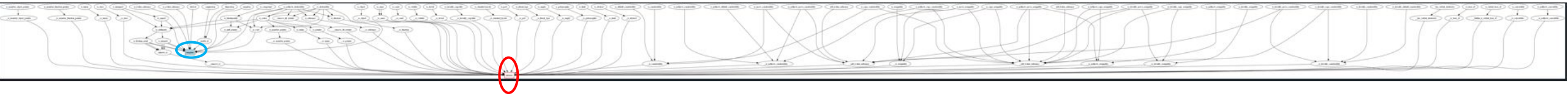


- Rich (a.k.a complicated (convoluted?) template-based type traits
- Late comer into the intrinsics world

MSVC

- Most heavily inclined towards intrinsics-based type traits
- Probably started like this from the get-go (just my conjecture)

CLANG (libcxx)



- Somewhat of a middle ground between clang and MSVC
- Increasingly dependent on intrinsics

GCC (libstd)

Assumptions and Caveats

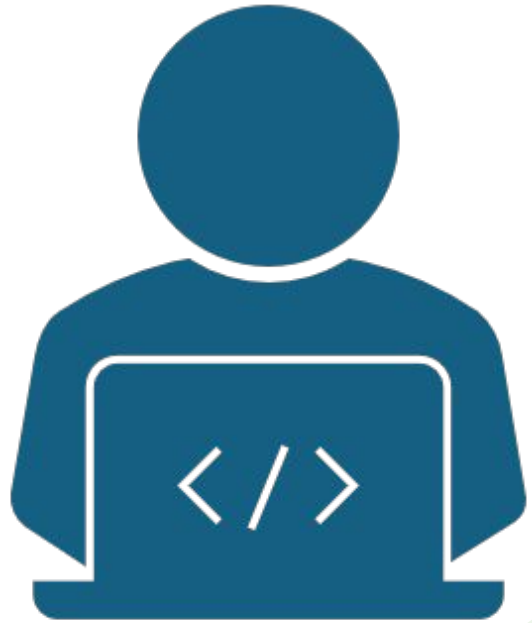
Intrinsic Implementations	• Only intrinsics shown when alternative implementations exist
Standard Type Traits Focus	• Analysis focused on standard type traits
Simplified Details	• Intermediate templates not shown
Covered only ~half the standard type traits	• Focused mainly on is_xxx traits
Potential Errors	• Because I'm only human

Cross-Toolchain Support

- MSVC -> clang
- Libcxx (clang) -> gcc

Recap - Drawbacks and Limitations of Current Type Trait Implementations

Complicated, hard to follow, logical programming style	• Presumably hard to maintain • Long cryptic compilation errors (concepts only partly to the rescue)
Template-based ones likely slow to compile	• The composite ones may often require maintenance
Some traits require "magic"	• e.g.: is_implicit_lifetime (P2674) • Magic wands often trivially and portably implementable via reflection



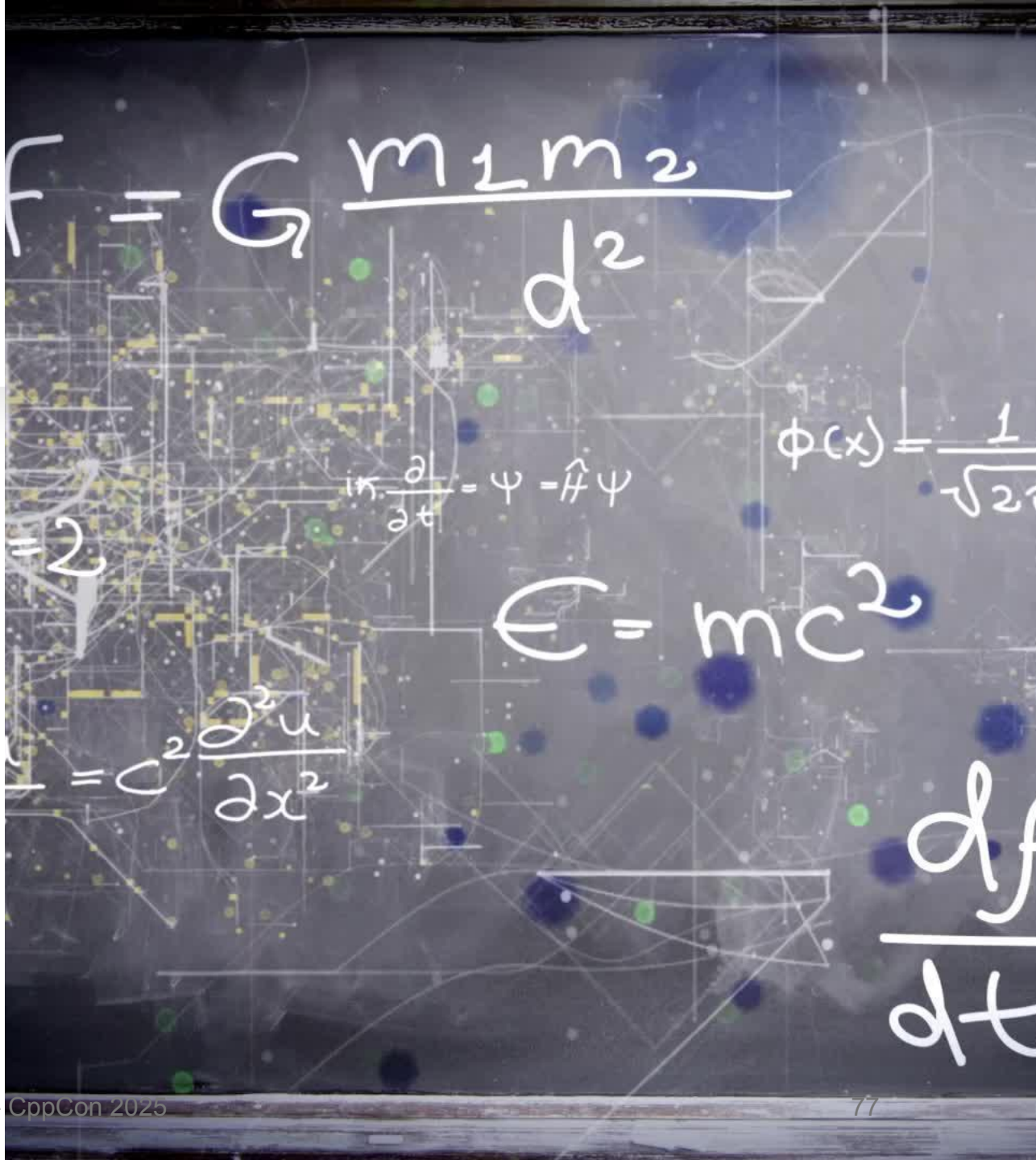
Static Reflection

- What do we mean by “static”?
  - Static = compile time
  - Dynamic reflection is more prevalent in other languages

Current State

Key Features Approved for C++26

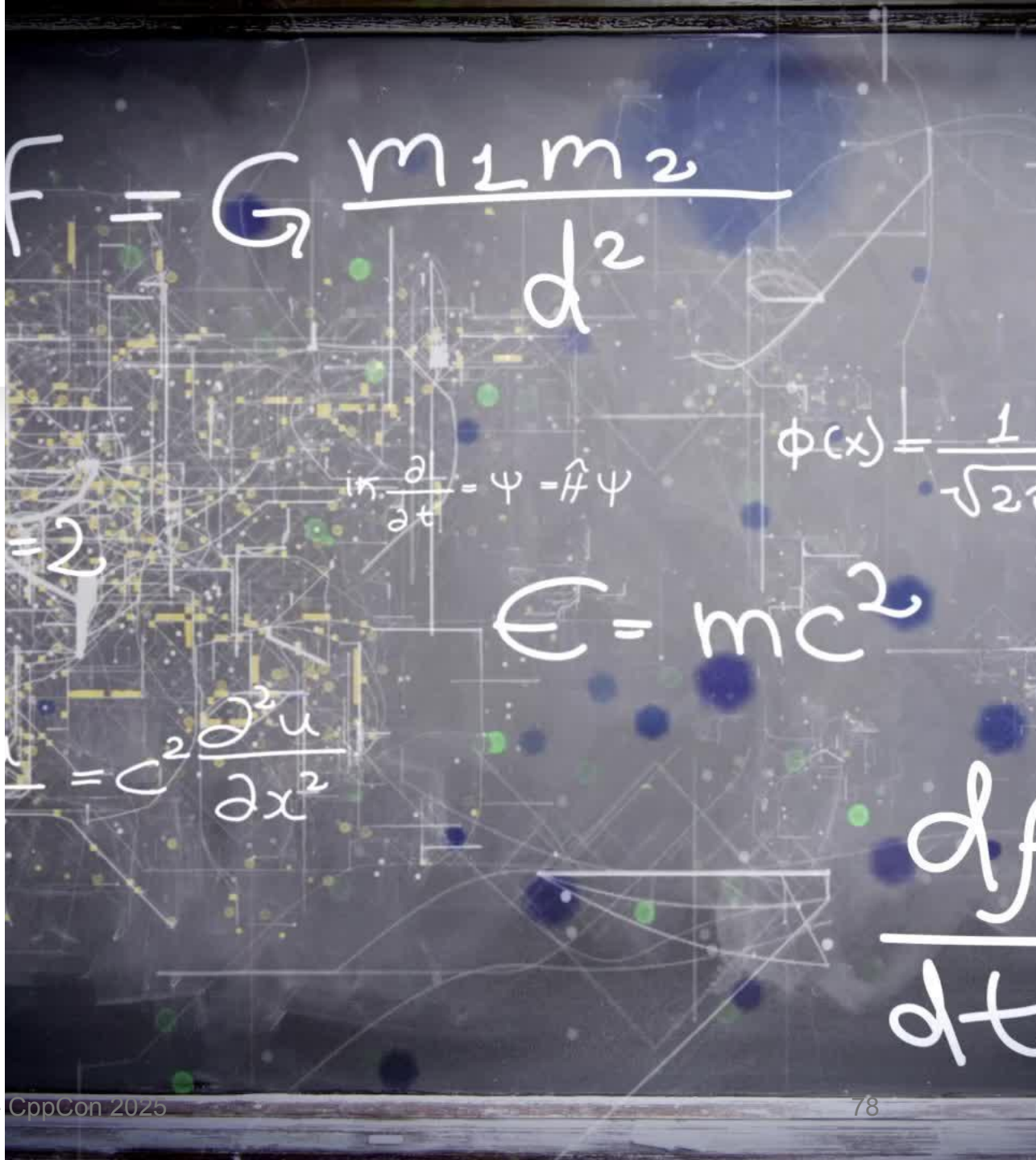
1. Approved in plenary with value-based approach (P2996)
2. Includes introspection but excludes injection (except splicing and *define_aggregate*)
3. No AST modifications; reflection uses pure value semantics
4. Expansion statements (*template for*) also approved (P1306)



Current State

Key Points About std::meta::info

1. Relies on std::meta::info - opaque type
2. Motivated by future compatibility with C++ type model
3. Future type-full wrapper libraries?



```
namespace std {
  namespace meta {
    using info = decltype(^^::);
  }
}
```

std::meta::info

— any (C++) type and type alias

— any function or member function

— any variable, static data member, or structured binding

— any non-static data member

— any enumerator

— any template

— any namespace (including the global namespace) or namespace alias

— any object that is a *permitted result of a constant expression*

— any value with *structural type* that is a permitted result of a constant expression

— the null reflection (when default-constructed)

std::meta::info

```
constexpr auto r = ^^int;
typename [r:] x = 42;          // Same as: int x = 42;
typename [^^char:] c = '*';    // Same as: char c = '*';
```

- **Back-And-Forth**

Some Examples from P2996

```
constexpr auto r = ^^int;
typename [r:] x = 42;          // Same as: int x = 42;
typename [^^char:] c = '*';    // Same as: char c = '*';
```

- **Back-And-Forth**

Some Examples from P2996

Oops - should actually be [:r:]

```
conste                                    ;
typename [r:] x = 42;          // Same as: int x = 42;
typename [^^char:] c = '*';    // Same as: char c = '*';
```

- **Back-And-Forth**

Some Examples from P2996

```
constexpr auto r = ^^int;
typename [r:] x = 42;          // Same as: int x = 42;
typename [^^char:] c = '*';    // Same as: char c = '*';
```

- **Back-And-Forth**

Some Examples from P2996

```cpp
struct S { unsigned i:2, j:6; };

consteval auto member_number(int n) {
    if (n == 0) return ^^S::i;
    else if (n == 1) return ^^S::j;
}

int main() {
    S s{0, 0};
    s.[member_number(1):] = 42;  // Same as: s.j = 42;
    s.[member_number(5):] = 0;   // Error (member_number(5) is not a constant)
}
```

- **Selecting Members**

Some Examples from P2996

```
struct S { unsigned i:2, j:6; };

consteval auto member_number(int n) {
    if (n == 0) return ^^S::i;
    else if (n == 1) return ^^S::j;
}

int main() {
    S s{0, 0};
    s.[member_number(1):] = 42;  // Same as: s.j = 42;
    s.[member_number(5):] = 0;   // Error (member_number(5) is not a constant).
}
```

- **Selecting Members**

Some Examples from P2996

```
struct S { unsigned i:2, j:6; };

consteval auto member_number(int n) {
    if (n == 0) return ^^S::i;
    else if (n == 1) return ^^S::j;
}

int main() {
    S s{0, 0};
    s.[member_number(1):] = 42;  // Same as: s.j = 42;
    s.[member_number(5):] = 0;   // Error (member_number(5) is not a constant).
}
```

- **Selecting Members**

Some Examples from P2996

```
constexpr std::array types = {^^int, ^^float, ^^double};
constexpr std::array sizes = []{
  std::array<std::size_t, types.size()> r;
  std::views::transform(types, r.begin(), std::meta::size_of);
  return r;
}();
```

- **List of Types to List of Sizes**

Some Examples from P2996

```cpp
template <typename E>
  requires std::is_enum_v<E>
constexpr std::string enum_to_string(E value) {
  template for (constexpr auto e : std::meta::enumerators_of(^^E)) {
    if (value == [:e:]) {
      return std::string(std::meta::identifier_of(e));
    }
  }

  return "<unnamed>";
}

enum Color { red, green, blue };
static_assert(enum_to_string(Color::red) == "red");
static_assert(enum_to_string(Color(42)) == "<unnamed>");
```

- **Enum to String**

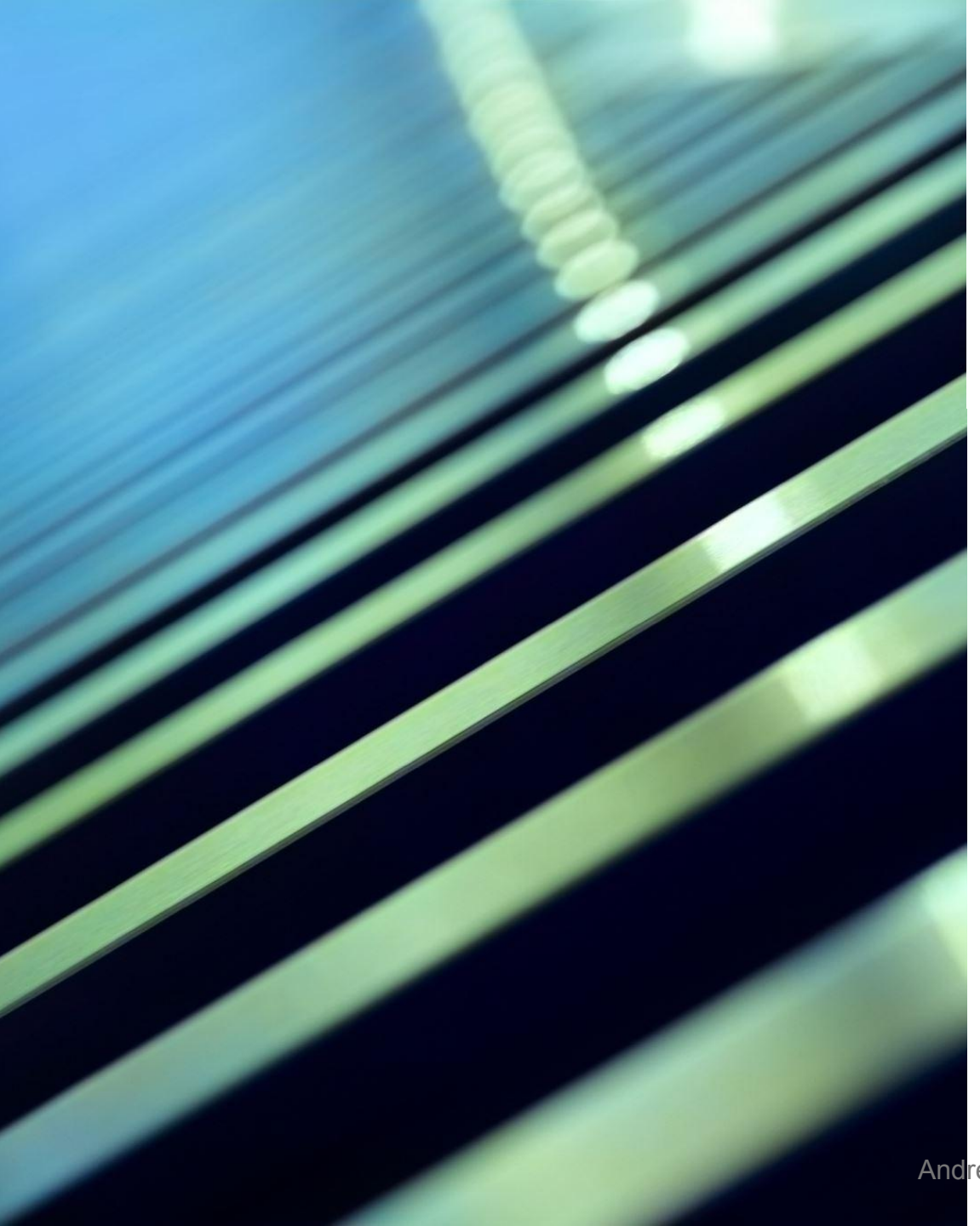
Some Examples from P2996

The Discarded Candidate – Type-Based Reflection

Challenges of Value-Based Reflection

1. Type hierarchy (rather than meta::info *values*)
2. Offers a complex OO API
3. Likely hard on build times and memory usage
4. May hinder future language scheme changes

Reflection and Type Traits

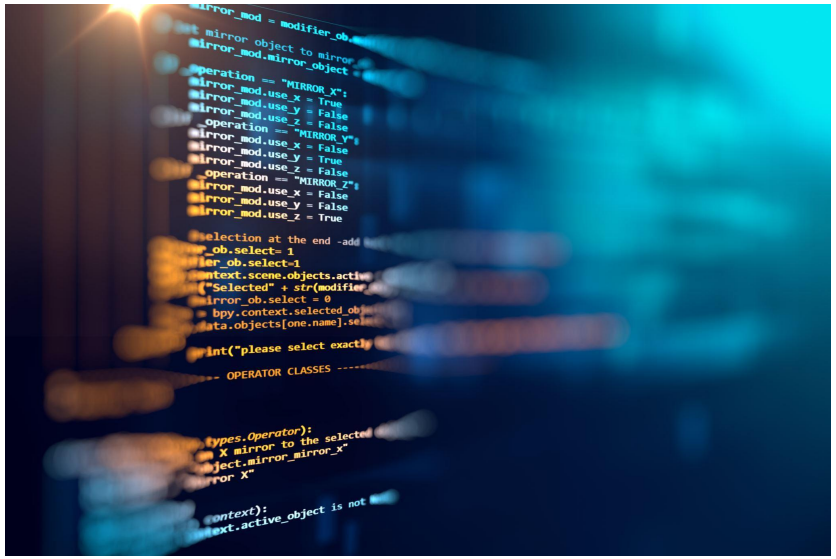


- Can we use reflection to yank intrinsics out of standard traits implementations?

Perhaps…
Probably…
Mostly…

- Conceptual POC (Bloomberg experimental P2996 GCC fork on Godbolt)
- Aided by MS Copilot + Think Deeper
- ~half the current standard type traits implemented, with no stdlib dependencies

https://compiler-explorer.com/z/4cn5j45Gv



```cpp
// ------------------------------------------------
// PRIMARY TYPE TRAITS
// ------------------------------------------------

template <typename T>
struct is_void : std::bool_constant<
    std::meta::is_same_type( ^^T, ^^void )
> { };

template <typename T>
struct is_null_pointer : std::bool_constant<
    std::meta::is_same_type( ^^T, ^^std::nullptr_t )
> { };

template <typename T>
struct is_integral : std::bool_constant<
    std::meta::is_integral_type( ^^T )
> { };

template <typename T>
struct is_floating_point : std::bool_constant<
    std::meta::is_floating_point_type( ^^T )
> { };

template <typename T>
struct is_array : std::bool_constant<
    std::meta::is_array_type( ^^T )
> { };

template <typename T>
struct is_pointer : std::bool_constant<
    std::meta::is_pointer_type( ^^T )
> { };
```

```
// ------------------------------------------------------------------------
// TYPE MODIFICATIONS
// ------------------------------------------------------------------------
template <typename T>
using remove_const_t = typename [: std::meta::remove_const( ^^T ) :];

template <typename T>
using remove_volatile_t = typename [: std::meta::remove_volatile( ^^T ) :];

template <typename T>
using remove_cv_t = typename [: std::meta::remove_cv( ^^T ) :];

template <typename T>
using remove_reference_t = typename [: std::meta::remove_reference( ^^T ) :];

template <typename T>
using remove_pointer_t = typename [: std::meta::remove_pointer( ^^T ) :];

template <typename T>
using add_pointer_t = typename [: std::meta::add_pointer( ^^T ) :];

template <typename T>
using add_lvalue_reference_t = [: std::meta::add_lvalue_reference( ^^T ) :];

template <typename T>
using add_rvalue_reference_t = [: std::meta::add_rvalue_reference( ^^T ) :];
```

Should we implement *all* type traits based on reflection?

- Definitely worth looking into
- Should be faster than complex template-based implementations
  - But possibly a little slower than direct intrinsics invocations (due to the added thin template layer)
- Code would be trivial where matching reflection traits exist
  - That's the low hanging fruit we should start with
- Not doing this would create 2 sources of truth
  - Which might diverge if either one is buggy
  - "In terms of" rocks!

Summary and Future Directions

TLDR

- Type traits are mostly implemented via compiler intrinsics, but often still via highly complex and long-building template shenanigans
- Value-based (std::meta::info) static reflection is coming in C++26
  - Initially without code injection, only with splicing
- Many type traits are directly implementable via reflection metafunctions
  - Rather than templates and compiler intrinsics
- Which begs the question…

Can we have standard library implementations completely independent from toolchains?

<meta> will have to be compiler-specific

- An always non-portable part of the standard library
- May use bespoke utilities with no cyclic dependencies.
- Example of a <meta> implementation - from the Bloomberg clang P2996 fork:
  - https://github.com/bloomberg/clang-p2996/blob/p2996/libcxx/include/meta

Key Takeaway

Reflection-based type traits are key to future decoupling between libraries and toolchains

- If that's what we want
  - Possible motivation: combining different vendors' strengths and velocities in implementing new C++ standards
  - Additional motivation: single source of truth
- We can get there gradually by ever-increasing the portable parts, ultimately being left only with <meta> and a few other performance-heavy vendor-specific parts
  - Which would also be standardized, but wouldn't have portable implementations
- Disclaimer – These are my own opinions. Cross-compiler portability of the standard library is currently not a stated goal of WG21.

Thank You for joining me in my reflections on the future!

Stay in touch!

andrziss@gmail.com
https://www.linkedin.com/in/andreizissu/

Questions?