Reflections

Compile-time Introspection of Source Code

Andrew Sutton

Reflection

"The integral ability for a program to observe or change its own code as well as all aspects of its programming language (syntax, semantics, or implementation), even at runtime."

Malenfant, Jacques & Jacques, M. & Demers, F. (1996). A Tutorial on Behavioral Reflection and its Implementation.

Reflection

"The integral ability for a program to **observe** or change its own code as well as **all aspects of its programming language (syntax, semantics, or implementation)**, even at runtime."

Reflection

"The integral ability for a program to **observe or change** its own code as well as **all aspects of its programming language (syntax, semantics, or implementation)**, even at runtime."

Reflection

"The integral ability for a program to observe or change its own code ~~as well as all aspects of its programming language (syntax, semantics, or implementation)~~, even at runtime."

Reflection

"The integral ability for a program to **observe or change its own code** ~~as well as all aspects of its programming language (syntax, semantics, or implementation)~~, even **at runtime**."

Introspection

"The integral ability for a program to **observe** ~~or change~~ **its own code** ~~as well as all aspects of its programming language (syntax, semantics, or implementation)~~, even **at runtime**."

Introspection

"The integral ability for a program to **observe** ~~or change~~ **its own code** ~~as well as all aspects of its programming language (syntax, semantics, or implementation), even at runtime~~."

Static introspection

"The integral ability for a program to **observe** ~~or change~~ **its own code** ~~as well as all aspects of its programming language (syntax, semantics, or implementation), even at runtime~~ **at compile-time**."

Static reflection

"The integral ability for a program to **observe** ~~or change~~ **its own code** ~~as well as all aspects of its programming language (syntax, semantics, or implementation), even at runtime~~ **at compile-time in order to shape its own definition**."

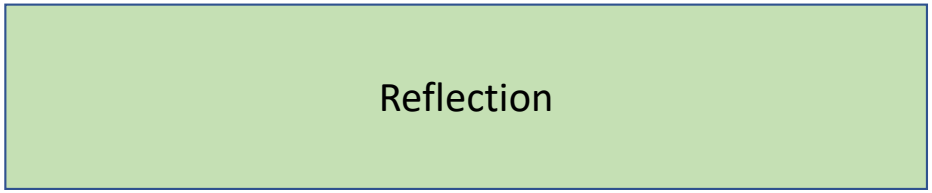
Static reflection

“The integral ability for a program to observe its own code at compile-time in order to shape its own definition.”

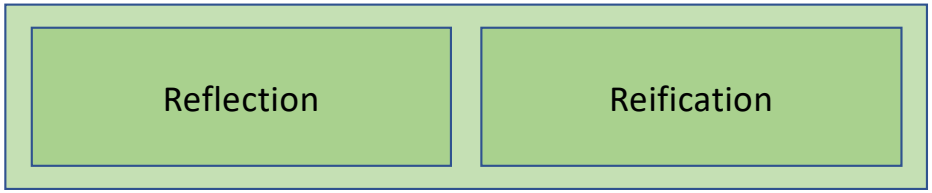
Type traits

Static reflection

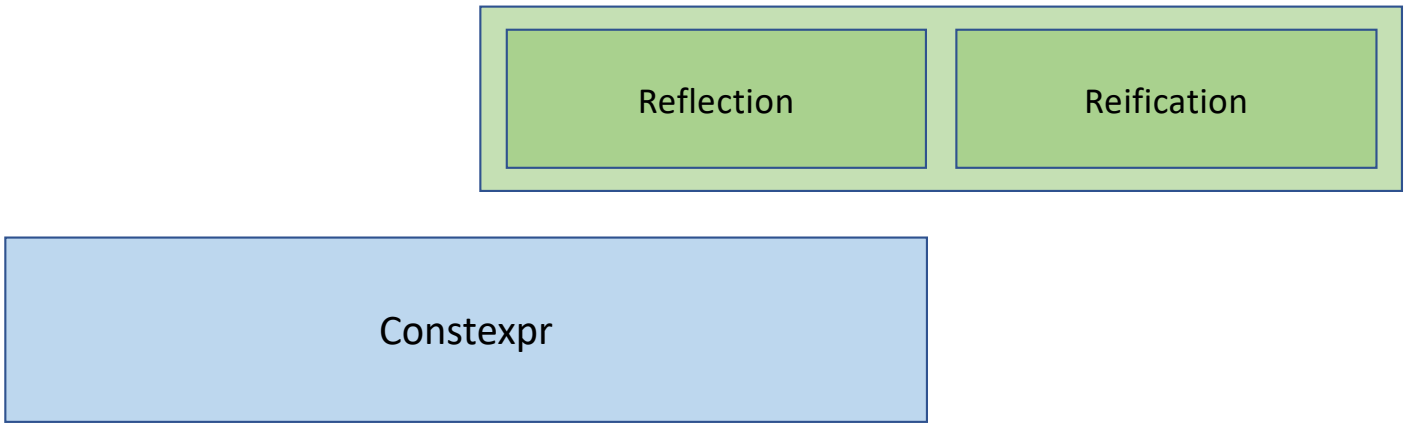
Features of static reflection



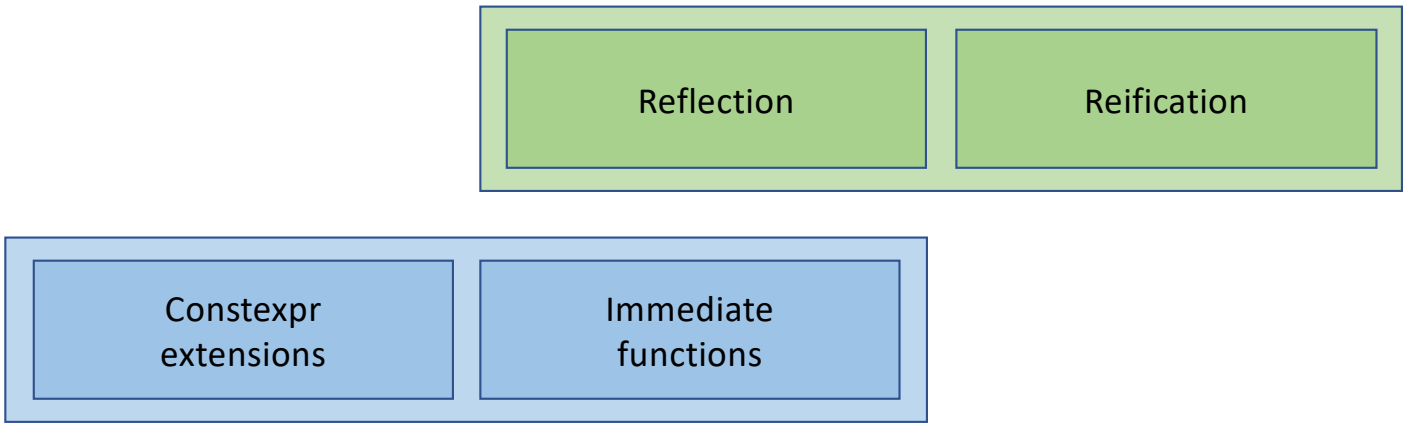
Features of static reflection



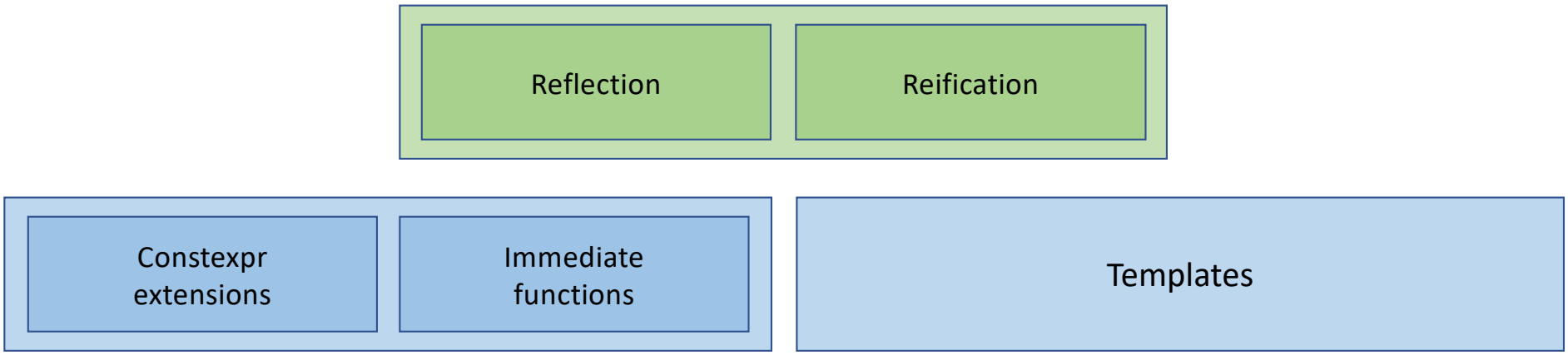
Features of static reflection



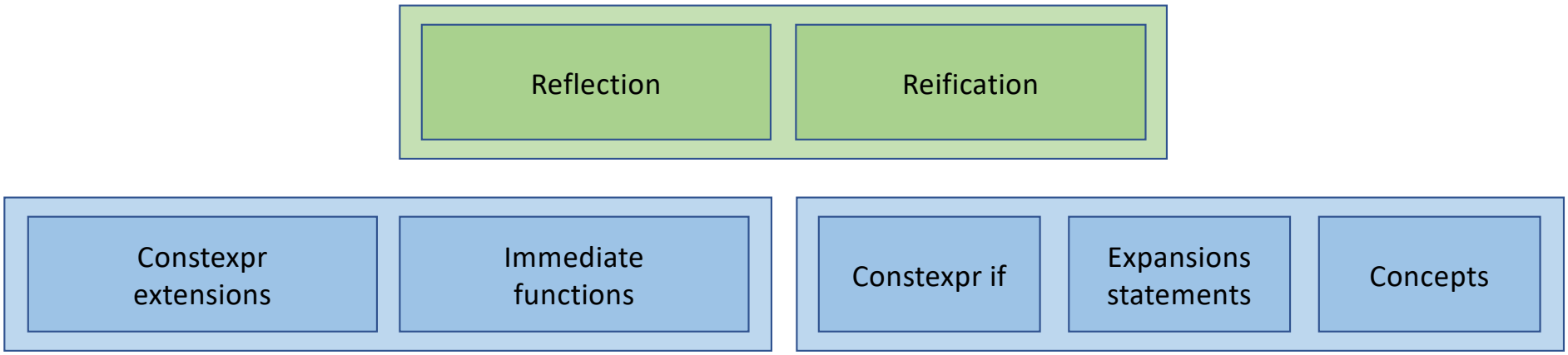
Features of static reflection



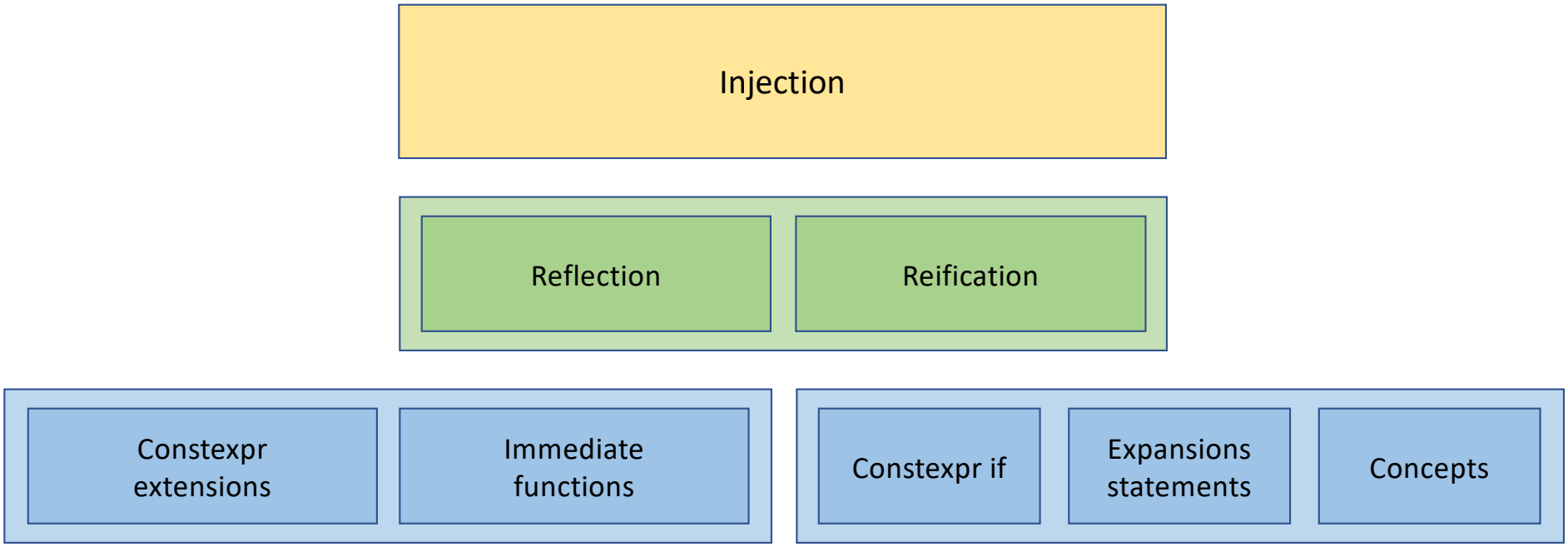
Features of static reflection



Features of static reflection



Features of static reflection



Papers supporting static reflection

N3996 – Static reflection
N4111 – Static reflection (rev. 2)
N4452 – A case for strong static reflection
N4746, N4766, N4818 – Working Draft, C++Extensions for Reflection
P0194 – Static reflection
P0385 – Static reflection: Rationale, design and evolution
P0425 – Metaprogramming by design, not by accident
P0578 – Static Reflection in a Nutshell
P0589 – Tuple-based for loops
P0590 – A design for static reflection

Papers supporting static reflection

P0595 – std::is_constant_evaluated()
P0597 – std::constexpr_vector<T>
P0598 – Reflect through values instead of types
P0633 – Exploring the design space of metaprogramming and reflection
P0670 – Static reflection of functions
P0712 – Implementing language support for compile-time meta…
P0784 – More constexpr containers
P0953 – constexpr reflexpr
P0992 – Translation and evaluation

Papers supporting static reflection

P0993 – Value-based Reflection
P1073 – constexpr! functions
P1240 – Scalable Reflection in C++
P1306 – Expansion statements
P1447 – constexpr C++ is not constexpr C
P1733 – User-friendly and Evolution-friendly Reflection: A Compromise

People involved

Andrew Sutton
Axel Naumann
Daveed Vandevoorde
David Sankel
Herb Sutter
Louis Dionne
Matúš Chochlík
Nina Ranns
Richard Smith
Sam Goodrick
Wyatt Childers

Why care about static reflection?

Reducing boilerplate

Type-based optimization

New forms of composition

Runtime introspection

A first example

Examples

Examples are based on our Clang implementation

https://gitlab.com/lock3/clang

Documentation here:

https://gitlab.com/lock3/clang/wikis/home

Also: https://cppx.godbolt.org/

Example caveats

Examples do not compile as written

- No support for concepts
- Remove concepts from code and these should compile

Some differences in operator spelling, library from P1240

Stringifying enumerators

```
enum E {
  A, B, C
};

int main() {
  std::cout << to_string(A) << '\n';
  std::cout << to_string(B) << '\n';
  std::cout << to_string(C) << '\n';
}
```

Stringifying enumerators

```
template<enumeral T>
char const* to_string (T value) {
  template for (constexpr auto member : meta::members_of(reflexpr(T)))
    if (valueof(member) == value)
      return meta::name_of(member);
  return "<unknown>";
}
```

Stringifying enumerators

```
template<enumeral T>
char const* to_string (T value) {
  constexpr meta::info type = reflexpr(T);
  constexpr auto members = meta::members_of(type);
  template for (constexpr auto member : members)
    if (valueof(member) == value)
      return meta::name_of(member);
  return "<unknown>";
}
```

Stringifying enumerators

```
template<enumeral T>
char const* to_string (T value) {
  constexpr meta::info type = reflexpr(T);
  constexpr auto members = meta::members_of(type);
  template for (constexpr auto member : members)
    if (valueof(member) == value)
      return meta::name_of(member);
  return "<unknown>";
}
```

Stringifying enumerators

```
template<enumeral T>
char const* to_string (T value) {
  constexpr meta::info type = reflexpr(T);
  constexpr auto members = meta::members_of(type);
  template for (constexpr auto member : members)
    if (valueof(member) == value)
      return meta::name_of(member);
  return "<unknown>";
}
```

Concepts

This function is only defined for enumeration types

```
template<typename T>
concept enumeral = std::is_enum_v<T>;
```

Stringifying enumerators

```
template<enumeral T>
char const* to_string (T value) {
  constexpr meta::info type = reflexpr(T);
  constexpr auto members = meta::members_of(type);
  template for (constexpr auto member : members)
    if (valueof(member) == value)
      return meta::name_of(member);
  return "<unknown>";
}
```

Reflection operator

Maps an *expression*, *type-id*, *template-name*, or *namespace-name* to a **reflection value**

```
reflexpr(int) // type reflection
reflexpr(0) // expression reflection
reflexpr(main) // function reflection
reflexpr(std) // namespace reflection
```

The `reflexpr` operator is a constant expression

Stringifying enumerators

```
template<enumeral T>
char const* to_string (T value) {
  constexpr meta::info type = reflexpr(T);
  constexpr auto members = meta::members_of(type);
  template for (constexpr auto member : members)
    if (valueof(member) == value)
      return meta::name_of(member);
  return "<unknown>";
}
```

Reflection values

A reflection value is a value of an implementation-defined scalar type

```
namespace std::experimental::meta {
  using info = decltype(reflexpr(void));
}
```

The “value” of `meta::info` is a handle to an internal compiler data structure representing the thing reflected

Reflection values

Reflections are only meaningful during constant expression evaluation

```
constexpr meta::info r1 = reflexpr(x); // OK

meta::info r2 = reflexpr(x); // error: reflection has
                             // no runtime interpretation
```

A brief history of reflection

In the beginning there were types

```
using T = reflexpr(x); // Reflection values are types
```

Pros: clear compile-time/runtime boundary, smallest possible extension

Cons: template metaprogramming, persistent values

A brief history of reflection

WG21 (mostly): Down with template metaprogramming

- Prefer constexpr
- Prefer ephemeral values

What is the type of `reflexpr`?

A brief history of C++ reflection values

Typeful approach (P0953): reflection values are described hierarchically

The base class metaprogramming

Monotype approach (P1240): type of all reflections is `meta::info`

The UNIX file descriptor of metaprogramming

The hidden cost of abstraction

```
constexpr int f(int n) {
  int i = 0;
  for (int k = 0; k < 10000; ++k)
    i += k;
  return k;
}
```

Compiler	Time
GCC 8.2	2.09s
Clang 5.0.1	2.23s
EDG 5.0	0.54s

Initially reported by Daveed Vandevoord (EDG) at C++Now’19

The hidden cost of abstraction

```
struct Int {
  constexpr Int(int n) : v(n) { }
  operator int&() { return v; }
  int v;
};
```

The hidden cost of abstraction

```
constexpr int f(integer n) {
  Int i = 0;
  for (Int k = 0; k < 10000; ++k)
    i += k;
  return k;
}
```

Compiler	Time
GCC 8.2	18.7s
Clang 5.0.1	6.25s
EDG 5.0	2.49s

The hidden cost of abstraction

```
constexpr int f(integer n) {
  Int i = 0;
  for (Int k = 0; k < 10000; ++k)
    i += k;
  return k;
}
```

Compiler	Time
GCC 8.2	18.7s – **9 times slower!**
Clang 5.0.1	6.25s – **3 times slower!**
EDG 5.0	2.49s – **5 times slower!**

The hidden cost of abstraction

abstraction == penalty

Reflection values in the future

Will (likely) continue to be monotype

P1733: User-friendly and evolution-friendly reflection: A compromise

New feature to allow overloading on “reflection value” using concepts

Stringifying enumerators

```
template<enumeral T>
char const* to_string (T value) {
  constexpr meta::info type = reflexpr(T);
  constexpr auto members = meta::members_of(type);
  template for (constexpr auto member : members)
    if (valueof(member) == value)
      return meta::name_of(member);
  return "<unknown>";
}
```

Reflection library

The meta library provides an extensive set of facilities to query properties of types, declarations, expressions, and other language constructs

```
consteval range members_of(reflection, predicate-list)
```

Takes a list of functions to filter results. Returns something iterable

Immediate functions

An immediate function (`consteval`) is evaluated as constant expression wherever it appears in the program, except in some cases

- This feature was added specifically to support the meta library
- Helps prevent accidental misuse of meta library functions and reduce typing
- Defines the boundary between metaprogramming and runtime programming

Immediate functions

Example:

```
std::cout << meta::has_static_storage(reflexpr(x));
```

Equivalent to:

```
constexpr bool v = meta::has_static_storage(reflexpr(x));
std::cout << v;
```

Stringifying enumerators

```
template<enumeral T>
char const* to_string (T value) {
  constexpr meta::info type = reflexpr(T);
  constexpr auto members = meta::members_of(type);
  template for (constexpr auto member : members)
    if (valueof(member) == value)
      return meta::name_of(member);
  return "<unknown>";
}
```

Expansion statements

Expands for each element in an iterable or destructurable value

```
template for (constexpr int x : /* some range */)
  cout << x;
```

Feature intended for C++20, but missed the ship deadline

Expansion

```
{
  constexpr int x = /* first value */;
  cout << x;
}
{
  constexpr int x = /* second value */;
  cout << x;
}
...
```

Stringifying enumerators

```
template<enumeral T>
char const* to_string (T value) {
  constexpr meta::info type = reflexpr(T);
  constexpr auto members = meta::members_of(type);
  template for (constexpr auto member : members)
    if (valueof(member) == value)
      return meta::name_of(member);
  return "<unknown>";
}
```

Reification operators

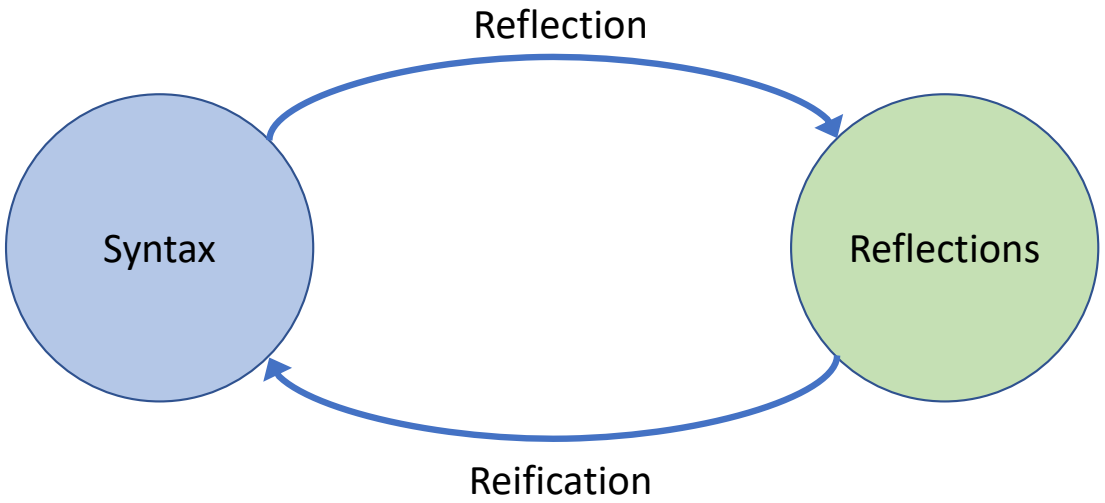
Maps a reflection value to a syntactic form (i.e., generates a term)

The `valueof` operator generates an expression that refers to an object or value or computes a constant expression

```
valueof(reflexpr(0)) == 0
```

Reflection and reification

Reification operators are inverses for reflection



Reification operators

Takes a constant expression reflection and generates syntax

```
typename(reflexpr(int)) // generates int
namespace(reflexpr(std)) // generates std
template(reflexpr(std::pair)) // generates std::pair
valueof(reflexpr(main)) // generates a pointer to main
idexpr(reflexpr(main)) // generates the expression main
unqualid(reflexpr(main)) // generates the id ‘main’
...
```

Specification divergence

Our implementation uses different names than P1240

- P1240's `unreflexpr` covers aspects of our `valueof` and `idexpr` reifiers
- We spell (`.` *reflection* `.`) as `unqualid`
- We spell (`<` *reflection* `>`) as `templarg`

Stringifying enumerators

```
template<enumeral T>
char const* to_string (T value) {
  constexpr meta::info type = reflexpr(T);
  constexpr auto members = meta::members_of(type);
  template for (constexpr auto member : members)
    if (valueof(member) == value)
      return meta::name_of(member);
  return "<unknown>";
}
```

Structural hashing

Structural hashing

Based on Howard Hinnant's 2014 "Types Don't Know #" proposals (N3980)

- Similar to serialization
- Hash codes are accumulated by "appending" bits into a code

Basic algorithms

```
template<hash_algorithm H, integral T>
void hash_append(H& hash, T n) {
  hash(&n, sizeof(N));
}
template<hash_algorithm H, input_range R>
void hash_append(H& hash, R const& range) {
  for (auto const& x : range)
    hash_append(hash, x);
}
```

Extensions for class types

Would have to overload for each concrete class or class template

Use static reflection to define a single operation for all classes

Structural hashing

```
template<hash_algorithm H, class_type T>
void hash_append(H& hash, T const& obj) {
  for template (constexpr auto m : meta::members_of(reflexpr(T))) {
    if constexpr (meta::is_nonstatic_data_member(m)) {
      auto ptr = valueof(m);
      hash_append(hash, obj.*ptr));
    }
  }
}
```

Structural hashing

```
template<hash_algorithm H, class_type T>
void hash_append(H& hash, T const& obj) {
  constexpr auto type = reflexpr(T);
  for template (constexpr auto m : meta::members_of(type)) {
    if constexpr (meta::is_nonstatic_data_member(m)) {
      auto ptr = valueof(m);
      hash_append(hash, obj.*ptr));
    }
  }
}
```

Structural hashing

```
template<hash_algorithm H, class_type T>
void hash_append(H& hash, T const& obj) {
  constexpr auto type = reflexpr(T);
  for template (constexpr auto m : meta::members_of(type)) {
    if constexpr (meta::is_nonstatic_data_member(m)) {
      auto ptr = valueof(m);
      hash_append(hash, obj.*ptr));
    }
  }
}
```

Structural hashing

```
template<hash_algorithm H, class_type T>
void hash_append(H& hash, T const& obj) {
  constexpr auto type = reflexpr(T);
  for template (constexpr auto m : meta::members_of(type)) {
    if constexpr (meta::is_nonstatic_data_member(m)) {
      auto ptr = valueof(m);
      hash_append(hash, obj.*ptr));
    }
  }
}
```

Structural hashing

```
template<hash_algorithm H, class_type T>
void hash_append(H& hash, T const& obj) {
  constexpr auto type = reflexpr(T);
  for template (constexpr auto m : meta::members_of(type)) {
    if constexpr (meta::is_nonstatic_data_member(m)) {
      auto ptr = valueof(m);
      hash_append(hash, obj.*ptr));
    }
  }
}
```

Structural hashing

```
template<hash_algorithm H, class_type T>
void hash_append(H& hash, T const& obj) {
  constexpr auto type = reflexpr(T);
  for template (constexpr auto m : meta::members_of(type)) {
    if constexpr (meta::is_nonstatic_data_member(m)) {
      auto ptr = valueof(m);
      hash_append(hash, obj.*ptr));
    }
  }
}
```

Runtime introspection

Nemesis: a library for runtime introspection

A library of facilities to support:

- Runtime introspection of types and declarations
- Dynamic typing
- Dynamic method invocation
- Scripting language bindings

Nemesis: a library for runtime introspection

A library of facilities to support:

- Runtime introspection of types and declarations
- ~~Dynamic typing~~
- ~~Dynamic method invocation~~
- ~~Scripting language bindings~~

Runtime introspection

Use static reflection to enable runtime introspection

```
type_info const& r1 = reflect<int>(); // no constexpr!
type_info const& r2 = reflect<my_class>();
// ...
```

Type information

```
struct type_info
{
protected:
  type_info(type_kind k);
public:
  bool is_fundamental() const;
  // ...
  bool is_class() const
  // ...
};
```

Derived type information

```
struct class_type_info : type_info
{
  class_type_info(char const* n)
    : type_info(tk_class), name(n)
  { }

  std::string name;
  std::vector<decl_info const*> members;
};
```

Reflection function

```
template<typename T>
type_info const& reflect() {
  constexpr meta::info type = reflexpr(T);
  if constexpr (meta::is_fundamental_type(type))
    return *make_fundamental_type<T>();
 ...
  if constexpr (meta::is_class_type(T));
    return *make_class_type<T>();
  ...
}
```

Creating information

```
template<typename T>
class_type_info const* make_class_type() {
  constexpr meta::info type = reflexpr(T);
  auto result = class_types().emplace(meta::name_of(type));
  if (result.second) {
    auto& ci = const_cast<class_type_info&>(*result.first);
    make_class_definition<T>(ci);
  }
  return &*result.first;
}
```

Creating information

```
template<typename T>
class_type_info const* make_class_type() {
  constexpr meta::info type = reflexpr(T);
  auto result = class_types().emplace(meta::name_of(type));
  if (result.second) {
    auto& ci = const_cast<class_type_info&>(*result.first);
    make_class_definition<T>(ci);
  }
  return &*result.first;
}
```

Creating information

```
template<typename T>
class_type_info const* make_class_type() {
  constexpr meta::info type = reflexpr(T);
  auto result = class_types().emplace(meta::name_of(type));
  if (result.second) {
    auto& ci = const_cast<class_type_info&>(*result.first);
    make_class_definition<T>(ci);
  }
  return &*result.first;
}
```

Making class definitions

```
template<typename T>
void make_class_definition(class_type_info& info) {
  constexpr meta::info type = reflexpr(T);
  template for (constexpr auto m : meta::members_of(type)) {
    if constexpr (meta::is_data_member(m))
      make_member_variable<m>(info);
    else if constexpr (meta::is_member_function(m))
      make_member_function<m>(info);
  }
}
```

Making class definitions

```
template<typename T>
void make_class_definition(class_type_info& info) {
  constexpr meta::info type = reflexpr(T);
  template for (constexpr auto m : meta::members_of(type)) {
    if constexpr (meta::is_data_member(m))
      make_member_variable<m>(info);
    else if constexpr (meta::is_member_function(m))
      make_member_function<m>(info);
  }
}
```

Making class definitions

```
template<typename T>
void make_class_definition(class_type_info& info) {
  constexpr meta::info type = reflexpr(T);
  template for (constexpr auto m : meta::members_of(type)) {
    if constexpr (meta::is_data_member(m))
      make_member_variable<m>(info);
    else if constexpr (meta::is_member_function(m))
      make_member_function<m>(info);
  }
}
```

Making class definitions

```
template<typename T>
void make_class_definition(class_type_info& info) {
  constexpr meta::info type = reflexpr(T);
  template for (constexpr auto m : meta::members_of(type)) {
    if constexpr (meta::is_data_member(m))
      make_member_variable<m>(info);
    else if constexpr (meta::is_member_function(m))
      make_member_function<m>(info);
  }
}
```

Making member data

```
template<meta::info member>
void make_member_variable(class_type_info& info) {
  const char* name = meta::name_of(member);
  using M = typename(meta::type_of(member));
  type_info const& type = reflect<M>();
  auto& decls = declarations();
  auto* var = decls.make_mem_var(name, type);
  info.members.push_back(var);
}
```

Making member data

```
template<meta::info member>
void make_member_variable(class_type_info& info) {
  const char* name = meta::name_of(member);
  using M = typename(meta::type_of(member));
  type_info const& type = reflect<M>();
  auto& decls = declarations();
  auto* var = decls.make_mem_var(name, type);
  info.members.push_back(var);
}
```

Making member data

```
template<meta::info member>
void make_member_variable(class_type_info& info) {
  const char* name = meta::name_of(member);
  using M = typename(meta::type_of(member));
  type_info const& type = reflect<M>();
  auto& decls = declarations();
  auto* var = decls.make_mem_var(name, type);
  info.members.push_back(var);
}
```

Making member data

```
template<meta::info member>
void make_member_variable(class_type_info& info) {
  const char* name = meta::name_of(member);
  using M = typename(meta::type_of(member));
  type_info const& type = reflect<M>();
  auto& decls = declarations();
  auto* var = decls.make_mem_var(name, type);
  info.members.push_back(var);
}
```

Dynamic typing

Use runtime-reflection to provide typing guarantees

```
struct object {
  object operator[](std::string const& str);

  template<typename... Ts)
  object operator()(Ts&&.. args);

  type_info const* type;
  std::any value;
};
```

Dynamic typing

Use runtime-reflection to provide typing guarantees

```
struct object {
  object operator[](std::string const& str);

  template<typename... Ts)
  object operator()(Ts&&.. args);

  type_info const* type;
  std::any value;
};
```

Dynamic typing

Use runtime-reflection to provide typing guarantees

```
struct object {
  object operator[](std::string const& str);

  template<typename... Ts)
  object operator()(Ts&&.. args);

  type_info const* type;
  std::any value;
};
```

Dynamic typing

Use runtime-reflection to provide typing guarantees

```
struct object {
  object operator[](std::string const& str);

  template<typename... Ts)
  object operator()(Ts&&.. args);

  type_info const* type;
  std::any value;
};
```

Dynamic typing

Use runtime-reflection to provide typing guarantees

```
struct object {
  object operator[](std::string const& str);

  template<typename... Ts)
  object operator()(Ts&&.. args);

  type_info const* type;
  std::any value;
};
```

Observations

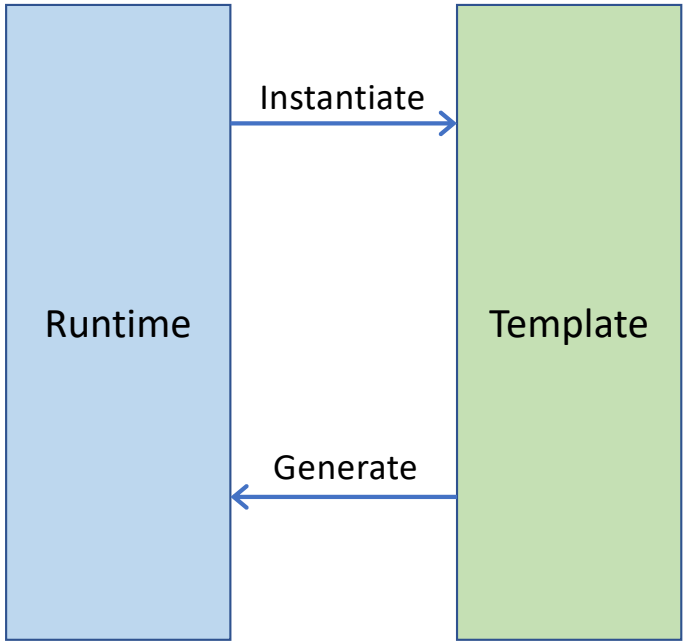
An architecture for reflective metaprograms

All examples have a common pattern:

- Runtime code uses templates
- Templates use reflection
- Reflection uses consteval

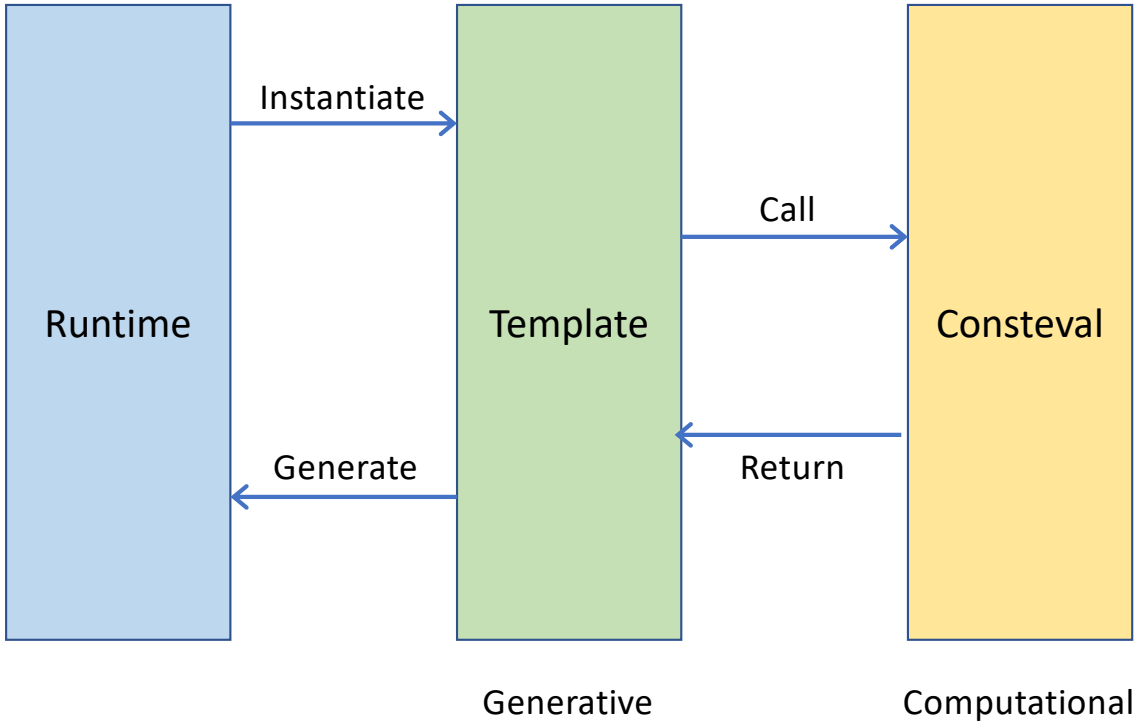
The compile-time divide

The compile-time divide



Generative

The compile-time divide



Final thoughts

With great power…

A very powerful tool

Potentially very easy to go overboard

- ~~Reflect all things!~~
- Compile-times matter

Benefits outweigh costs if used responsibly

Thank you!

Questions?