Template-Less MetaProgramming

KRIS JUSIAK

Cppcon
The C++ Conference

2024
September 15 - 20

Template Metaprogramming (TMP)

```
template<class... Ts>
using meta_fun = this_talk<Ts...>;
```

```
❓static_assert(Template::Metaprogramming::is_hard); // ✔
```

```
❓static_assert(Template::Metaprogramming::is_hard); // ✔
```

```
❓static_assert(Template::Metaprogramming::is_powerful); // ✔
```

```
❓static_assert(Template::Metaprogramming::is_hard); // ✔
```

```
❓static_assert(Template::Metaprogramming::is_powerful); // ✔
```

```
❓static_assert((
    Template::Metaprogramming::is_easy and
    Template::Metaprogramming::is_powerful and
    Template::Metaprogramming::has_nice_error_messages and
    Template::Metaprogramming::is_fast_to_compile
  ) and ...); // ✔
```

Motivation / Examples

> *"Better Metaprogramming features make better libraries!"*
>
> *Sean Baxter*

[Examples] Standard Template Library (STL)

```
template<class... Ts>
template<class T>
constexpr variant<Ts...>::variant(T&& t)
 : index{find_index<T, Ts...>} // Powered by TMP
 , // ...
{ }
```

[Examples] Standard Template Library (STL)

```
template<class... Ts>
template<class T>
constexpr variant<Ts...>::variant(T&& t)
 : index{find_index<T, Ts...>} // Powered by TMP
 , // ...
{ }
```

```
template<size_t I, class... Ts>
constexpr auto get(tuple<Types...>&&) noexcept ->
  typename tuple_element<I, tuple<Ts...>>::type&&; // Powered by TMP
```

[Examples] Standard Template Library (STL)

```
template<class... Ts>
template<class T>
constexpr variant<Ts...>::variant(T&& t)
 : index{find_index<T, Ts...>} // Powered by TMP
 , // ...
{ }
```

```
template<size_t I, class... Ts>
constexpr auto get(tuple<Types...>&&) noexcept ->
  typename tuple_element<I, tuple<Ts...>>::type&&; // Powered by TMP
```

```
template<class TFirst, class... TRest>
array(TFirst, TRest...) -> array<
  typename Enforce_same<TFirst, TRest...>::type, // Powered by TMP
  1 + sizeof...(TRest)
>;
```

[Examples] Standard Template Library (STL)

```
template<class... Ts>
template<class T>
constexpr variant<Ts...>::variant(T&& t)
 : index{find_index<T, Ts...>} // Powered by TMP
 , // ...
{ }
```

```
template<size_t I, class... Ts>
constexpr auto get(tuple<Types...>&&) noexcept ->
  typename tuple_element<I, tuple<Ts...>>::type&&; // Powered by TMP
```

```
template<class TFirst, class... TRest>
array(TFirst, TRest...) -> array<
  typename Enforce_same<TFirst, TRest...>::type, // Powered by TMP
  1 + sizeof...(TRest)
>;
```

▪▪▪

Standard Template Library (STL) doesn't have Standard Template Metaprogramming library

Standard Template Library (STL) doesn't have Standard Template Metaprogramming library

---

https://wg21.link/p0949 - Type-based Metaprogramming Standard Library, Peter Dimov

[Examples] Performance / Memory

```
struct unpacked {
  char a; static_assert(sizeof(a) == 1u); // x86-64
  int  b; static_assert(sizeof(b) == 4u); // x86-64
  char c; static_assert(sizeof(c) == 1u); // x86-64
};
```

[Examples] Performance / Memory

```
struct unpacked {
  char a; static_assert(sizeof(a) == 1u); // x86-64
  int  b; static_assert(sizeof(b) == 4u); // x86-64
  char c; static_assert(sizeof(c) == 1u); // x86-64
};
```

```
/*
 * https://eel.is/c++draft/basic.align
 */
static_assert(12u == sizeof(unpacked));
```

[Examples] Performance / Memory

```
struct unpacked {
  char a; static_assert(sizeof(a) == 1u); // x86-64
  int  b; static_assert(sizeof(b) == 4u); // x86-64
  char c; static_assert(sizeof(c) == 1u); // x86-64
};
```

```
/*
 * https://eel.is/c++draft/basic.align
 */
static_assert(12u == sizeof(unpacked));
```

```
static_assert(8u == sizeof(pack_t<unpacked>)); // Powered by TMP
```

[Examples] Performance / Memory

```
struct unpacked {
  char a; static_assert(sizeof(a) == 1u); // x86-64
  int  b; static_assert(sizeof(b) == 4u); // x86-64
  char c; static_assert(sizeof(c) == 1u); // x86-64
};
```

```
/*
 * https://eel.is/c++draft/basic.align
 */
static_assert(12u == sizeof(unpacked));
```

```
static_assert(8u == sizeof(pack_t<unpacked>)); // Powered by TMP
```

```
static_assert(
  requires(pack_t<unpacked> p) { // Powered by TMP
    p.a;
    p.b;
    p.c;
  }
);
```

[Examples] Performance / Policies

```
constexpr std::array protocols{
  std::pair{"ftp"sv,  protocol::FTP},
  std::pair{"file"sv, protocol::FILE},
  std::pair{"http"sv, protocol::HTTP},
  std::pair{"ws"sv,   protocol::WS},
  std::pair{"wss"sv,  protocol::WSS},
};
```

[Examples] Performance / Policies

```
constexpr std::array protocols{
  std::pair{"ftp"sv,  protocol::FTP},
  std::pair{"file"sv, protocol::FILE},
  std::pair{"http"sv, protocol::HTTP},
  std::pair{"ws"sv,   protocol::WS},
  std::pair{"wss"sv,  protocol::WSS},
};
```

```
template<class TPolicy>
auto lookup(std::string_view) -> protocol; // Powered by TMP
```

[Examples] Performance / Policies

```
[1]: #uOps   [2]: Latency  [3]: RThroughput
[4]: MayLoad [5]: MayStore [6]: HasSideEffects (U)
```

```
[1]  [2]  [3]  [4]  [5]  [6]  lookup<magic_lut>: // x86-64
 1    1   0.25                 shl   edi, 3
 2    5   0.50   *             bzhi  eax, dword ptr [rsi], edi
 1    3   1.00                 imul  eax, eax, 267363435
 1    1   0.25                 shr   eax, 29
 1    1   0.25                 mov   ecx, 531
 1    1   0.25                 shrx  eax, ecx, eax
 1    1   0.25                 and   eax, 7
```

```
[1]  [2]  [3]  [4]  [5]  [6]  lookup<pext>: // x86-64
 1    1   0.50                 shl   edi, 3
 2    5   0.50   *             bzhi  eax, dword ptr [rsi], edi
 1    1   0.50                 mov   ecx, 65795
 1    3   1.00                 pext  eax, eax, ecx
 1    1   0.33                 lea   rcx, [rip + lookup]
 1    5   0.33   *             mov   eax, dword ptr [rcx + 4*rax]
                               .lookup 2 0 0 0 0 0 0 1 3 0 ... 4
```

[Examples] Domain Specific Languages (DSL)

```
struct Disconnected { };
struct Connecting   { };
struct Connected    { };
```

```
sm connection = overload{
 [](Disconnected, connect)     -> Connecting   { establish(); },
 [](Connecting,   established) -> Connected    { },
 [](auto,         ping e)                      { if (not e) reset(); },
 [](Connecting,   auto)        -> Connecting   { establish(); },
 [](auto,         disconnect)  -> Disconnected { close(); },
 [](auto,         auto)        -> Error        { },
};
```

[Examples] Domain Specific Languages (DSL)

```
struct Disconnected { };
struct Connecting   { };
struct Connected    { };
```

```
sm connection = overload{
 [](Disconnected, connect)     -> Connecting   { establish(); },
 [](Connecting,   established) -> Connected    { },
 [](auto,         ping e)                      { if (not e) reset(); },
 [](Connecting,   auto)        -> Connecting   { establish(); },
 [](auto,         disconnect)  -> Disconnected { close(); },
 [](auto,         auto)        -> Error        { },
};
```

```
static_assert(sizeof(connection) == 1u); // Powered by TMP
```

More <Template::Metaprogramming> examples - https://github.com/boostorg

[Brief history] Template Metaprogramming

- C++
    - Type-based TMP (boost.mpl -> boost.mp11, ...)
    - Heterogeneous-based TMP (boost.fusion -> boost.hana)
    - Value-based TMP (mp*, https://wg21.link/p2996*)
  - Circle-lang (Circle-lang meta model*)
- Zig-lang (comptime)
- ...

---

* - this_talk

[By Example] Value-based TMP

Value-based TMP - find_index

```
template<class T, class... Ts>
constexpr auto find_index() -> std::size_t; // TODO
```

```
static_assert(0u == find_index<int,  int,   float, short>());
static_assert(1u == find_index<int,  float, int,   short>());
static_assert(2u == find_index<int,  float, short, int>());
static_assert(3u == find_index<void, float, short, int>());
```

Value-based TMP - find_index (pseudo code)

```
template<class T, class... Ts>
constexpr auto find_index() -> std::size_t {




}
```

Value-based TMP - find_index (pseudo code)

```
template<class T, class... Ts>
constexpr auto find_index() -> std::size_t {

  std::array ts{🪄<Ts>...};

}
```

 - magic

Value-based TMP - find_index (pseudo code)

```
template<class T, class... Ts>
constexpr auto find_index() -> std::size_t {

  std::array ts{🪄<Ts>...};

  for (auto i = 0u; i < ts.size(); ++i) { // no template for
    if (ts[i] == 🪄<T>) { // no constexpr
      return i;
    }
  }

}
```

 - magic

Value-based TMP - find_index (pseudo code)

```
template<class T, class... Ts>
constexpr auto find_index() -> std::size_t {

  std::array ts{🪄<Ts>...};

  for (auto i = 0u; i < ts.size(); ++i) { // no template for
    if (ts[i] == 🪄<T>) { // no constexpr
      return i;
    }
  }

  return ts.size();

}
```

 - magic

Value-based TMP - 

Value-based TMP - 🪄

```
std::array ts{🪄<int>, 🪄<void>}; // ?
```

Value-based TMP - 🪄

```
std::array ts{🪄<int>, 🪄<void>}; // ?
```

```
template<class T>
inline constexpr auto meta = type<T>::id;
```

```
static_assert(meta<int> == meta<int>);
static_assert(meta<int> != meta<void>);
static_assert(typeid(meta<int>) == typeid(meta<void>));
```

Value-based TMP - 🪄

```
std::array ts{🪄<int>, 🪄<void>}; // ?
```

```
template<class T> struct type {
  static void id(); // or static constexpr variable, ...
};
```

```
template<class T>
inline constexpr auto meta = type<T>::id;
```

```
static_assert(meta<int> == meta<int>);
static_assert(meta<int> != meta<void>);
static_assert(typeid(meta<int>) == typeid(meta<void>));
```

Value-based TMP - 🪄

```
std::array ts{🪄<int>, 🪄<void>}; // ?
```

```
template<class T> struct type {
  static void id(); // or static constexpr variable, ...
};
```

```
template<class T>
inline constexpr auto meta = type<T>::id;
```

```
static_assert(meta<int> == meta<int>);
static_assert(meta<int> != meta<void>);
static_assert(typeid(meta<int>) == typeid(meta<void>));
```

```
std::array ts{meta<int>, meta<void>}; // ✔
```

Value-based TMP - find_index

Value-based TMP - find_index

```
template<class T, class... Ts>
constexpr auto find_index() -> std::size_t {




}
```

Value-based TMP - find_index

```
template<class T, class... Ts>
constexpr auto find_index() -> std::size_t {
```

```
  std::array ts{meta<Ts>...};
```

```
}
```

Value-based TMP - find_index

```
template<class T, class... Ts>
constexpr auto find_index() -> std::size_t {
```

```
  std::array ts{meta<Ts>...};
```

```
  for (auto i = 0u; i < ts.size(); ++i) {
    if (ts[i] == meta<T>) {
      return i;
    }
  }
```

```
}
```

Value-based TMP - find_index

```
template<class T, class... Ts>
constexpr auto find_index() -> std::size_t {

  std::array ts{meta<Ts>...};

  for (auto i = 0u; i < ts.size(); ++i) {
    if (ts[i] == meta<T>) {
      return i;
    }
  }

  return ts.size();

}
```

Value-based TMP - find_index

```
template<class T, class... Ts>
constexpr auto find_index() -> std::size_t {
```

```
  std::array ts{meta<Ts>...};
```

```
  for (auto i = 0u; i < ts.size(); ++i) {
    if (ts[i] == meta<T>) {
      return i;
    }
  }
```

```
  return ts.size();
```

```
}
```

```
static_assert(0u == find_index<int,  int,   float, short>()); // ✔
static_assert(1u == find_index<int,  float, int,   short>()); // ✔
static_assert(2u == find_index<int,  float, short, int>());   // ✔
static_assert(3u == find_index<void, float, short, int>());   // ✔
```

Value-based TMP - find_index - template-less refactor

Value-based TMP - find_index - template-less refactor

```
constexpr auto find_index(auto t, const std::ranges::range auto& ts)
  -> std::size_t {
```

```
}
```

Value-based TMP - find_index - template-less refactor

```
constexpr auto find_index(auto t, const std::ranges::range auto& ts)
  -> std::size_t {
```

```
  for (auto i = 0u; i < ts.size(); ++i) {
    if (ts[i] == t) {
      return i;
    }
  }
  return ts.size();
```

```
}
```

Value-based TMP - find_index - template-less refactor

```
constexpr auto find_index(auto t, const std::ranges::range auto& ts)
  -> std::size_t {
```

```
  for (auto i = 0u; i < ts.size(); ++i) {
    if (ts[i] == t) {
      return i;
    }
  }
  return ts.size();
```

```
}
```

```
static_assert(0u == find_index(
  meta<int>, std::array{meta<int>, meta<float>, meta<short>}
)); // ✔
```

```
static_assert(2u == find_index(
  42, std::array{1, 2, 42, 3}
)); // ✔
```

Value-based TMP - find_index - no raw loops refactor

Value-based TMP - find_index - no raw loops refactor

```
constexpr auto find_index(auto t, const std::ranges::range auto& ts)
  -> std::size_t {
```

```
}
```

Value-based TMP - find_index - no raw loops refactor

```
constexpr auto find_index(auto t, const std::ranges::range auto& ts)
  -> std::size_t {
```

```
  if (const auto found = std::ranges::find(ts, t); found) {
    return std::distance(ts.begin(), found);
  }
  return ts.size();
```

```
}
```

Value-based TMP - find_index - no raw loops refactor

```
constexpr auto find_index(auto t, const std::ranges::range auto& ts)
  -> std::size_t {
```

```
  if (const auto found = std::ranges::find(ts, t); found) {
    return std::distance(ts.begin(), found);
  }
  return ts.size();
```

```
}
```

```
static_assert(0u == find_index(
  meta<int>, std::array{meta<int>, meta<float>, meta<short>}
)); // ✔

static_assert(3u == find_index(
  meta<void>, std::array{meta<int>, meta<float>, meta<short>}
)); // ✔
```

❓ Will 'no raw loops' refactor be faster or slower to compile?

Value-based TMP - tuple_element

```
template<std::size_t I, class... Ts>
struct tuple_element; // TODO
```

```
static_assert(
 std::is_same_v<int, tuple_element<0, std::tuple<int, bool>>::type> and
 std::is_same_v<bool,tuple_element<1, std::tuple<int, bool>>::type> and
);
```

Value-based TMP - tuple_element (pseudo code)

```
template<std::size_t I, class... Ts>
struct tuple_element;
```

```
template<std::size_t I, class... Ts>
struct tuple_element<I, std::tuple<Ts...>> {
  using type = 🪄<std::array{meta<Ts>...}[I]>;
};
```

Value-based TMP - 🪄

```
🪄<std::array{meta<Ts>...}[I]> // ?
```

Value-based TMP - 🪄

```
🪄<std::array{meta<Ts>...}[I]> // ?
```

```
template<class T> struct type {
  using value_type = T;
  static void id();
```

```
};
```

Value-based TMP - 🪄

```
🪄<std::array{meta<Ts>...}[I]> // ?
```

```
template<auto> struct info { constexpr auto friend get(info); };

template<class T> struct type {
  using value_type = T;
  static void id();

  constexpr auto friend get(info<id>) { return type{}; }

};
```

Value-based TMP - 🪄

```
🪄<std::array{meta<Ts>...}[I]> // ?
```

```
template<auto> struct info { constexpr auto friend get(info); };

template<class T> struct type {
  using value_type = T;
  static void id();

  constexpr auto friend get(info<id>) { return type{}; }

};

template<auto meta>
using type_of = typename decltype(get(info<meta>{}))::value_type;
```

Value-based TMP - 🪄

```
🪄<std::array{meta<Ts>...}[I]> // ?
```

```
template<auto> struct info { constexpr auto friend get(info); };

template<class T> struct type {
  using value_type = T;
  static void id();

  constexpr auto friend get(info<id>) { return type{}; }

};

template<auto meta>
using type_of = typename decltype(get(info<meta>{}))::value_type;

type_of<meta<int>> i = 42; // int i = 42;
```

Value-based TMP - 🪄

```
🪄<std::array{meta<Ts>...}[I]> // ?
```

```
template<auto> struct info { constexpr auto friend get(info); };
```

```
template<class T> struct type {
  using value_type = T;
  static void id();
```

```
  constexpr auto friend get(info<id>) { return type{}; }
```

```
};
```

```
template<auto meta>
using type_of = typename decltype(get(info<meta>{}))::value_type;
```

```
type_of<meta<int>> i = 42; // int i = 42;
```

```
type_of<std::array{meta<Ts>...}[I]> // ✔
```

Value-based TMP - tuple_element

```
template<std::size_t I, class... Ts>
struct tuple_element<I, std::tuple<Ts...>> {



};
```

Value-based TMP - tuple_element

```
template<std::size_t I, class... Ts>
struct tuple_element<I, std::tuple<Ts...>> {
```

```
  using type = type_of<std::array{meta<Ts>...}[I]>;
```

```
};
```

```
static_assert(
 std::is_same_v<int, tuple_element<0, std::tuple<int, bool>>::type> and
 std::is_same_v<bool,tuple_element<1, std::tuple<int, bool>>::type> and
); // ✔
```

Value-based TMP - tuple_element

```
template<std::size_t I, class... Ts>
struct tuple_element<I, std::tuple<Ts...>> {
```

```
  using type = type_of<std::array{meta<Ts>...}[I]>;
```

```
};
```

```
static_assert(
 std::is_same_v<int, tuple_element<0, std::tuple<int, bool>>::type> and
 std::is_same_v<bool,tuple_element<1, std::tuple<int, bool>>::type> and
); // ✔
```

C++26 - Pack indexing - https://wg21.link/p2662

```
template<std::size_t I, class... Ts>
struct tuple_element<I, std::tuple<Ts...>> {
  using type = Ts...[I]; // __type_pack_element
};
```

Value-based TMP - variant_for_t

```cpp
/**
 * std::variant with unique types without qualifiers
 * for which T type is constructible with Ts type
 *
 * @tparam T type
 * @tparam Ts... list of types
 */
template<class T, class... Ts>
using variant_for_t; // TODO
```

Value-based TMP - variant_for_t

```cpp
/**
 * std::variant with unique types without qualifiers
 * for which T type is constructible with Ts type
 *
 * @tparam T type
 * @tparam Ts... list of types
 */
template<class T, class... Ts>
using variant_for_t; // TODO
```

```cpp
struct bar { };
struct foo {
  foo(int) { }
  foo(bar) { }
};
```

```cpp
static_assert(std::is_same_v<
  std::variant<int, bar>, // because of foo(int) and foo(bar)
  variant_for_t<foo, const int&, int&&, std::string_view, bar, void>
>);
```

Value-based TMP - variant_for (pseudo code)

```
template<class T>
constexpr auto variant_for(const std::ranges::range auto& ts)
  -> std::ranges::range auto {
```

```
}
```

Value-based TMP - variant_for (pseudo code)

```
template<class T>
constexpr auto variant_for(const std::ranges::range auto& ts)
  -> std::ranges::range auto {
```

```
  std::vector<🪄> r;
```

```
}
```

Value-based TMP - variant_for (pseudo code)

```
template<class T>
constexpr auto variant_for(const std::ranges::range auto& ts)
  -> std::ranges::range auto {
```

```
  std::vector<🪄> r;
```

```
  for (auto t : ts) {
    if (🪄<std::is_constructible, T>(t)) {
      r.push_back(🪄<std::remove_cvref>(t));
    }
  }
```

```
}
```

Value-based TMP - variant_for (pseudo code)

```
template<class T>
constexpr auto variant_for(const std::ranges::range auto& ts)
  -> std::ranges::range auto {
```

```
  std::vector<🪄> r;
```

```
  for (auto t : ts) {
    if (🪄<std::is_constructible, T>(t)) {
      r.push_back(🪄<std::remove_cvref>(t));
    }
  }
```

```
  std::sort(r.begin(), r.end());
  r.erase(std::unique(r.begin(), r.end()), r.end());
```

```
}
```

Value-based TMP - variant_for (pseudo code)

```
template<class T>
constexpr auto variant_for(const std::ranges::range auto& ts)
  -> std::ranges::range auto {

  std::vector<🪄> r;

  for (auto t : ts) {
    if (🪄<std::is_constructible, T>(t)) {
      r.push_back(🪄<std::remove_cvref>(t));
    }
  }

  std::sort(r.begin(), r.end());
  r.erase(std::unique(r.begin(), r.end()), r.end());

  return r;

}
```

Value-based TMP - 🪄

```
std::vector<🪄> // ?
```

Value-based TMP - 🪄

```
std::vector<🪄> // ?
```

```
using info = decltype(meta<void>);
```

Value-based TMP - 🪄

```
std::vector<🪄> // ?
```

```
using info = decltype(meta<void>);
```

```
static_assert(typeid(meta<int>)   == typeid(info));
static_assert(typeid(meta<float>) == typeid(info));
```

Value-based TMP - 🪄

```
std::vector<🪄> // ?
```

```
using info = decltype(meta<void>);
```

```
static_assert(typeid(meta<int>)   == typeid(info));
static_assert(typeid(meta<float>) == typeid(info));
```

```
std::vector<info> // ✔
```

Value-based TMP - 

Value-based TMP - 🪄

```
🪄<std::is_constructible, T>(info) // ?
🪄<std::remove_cvref>(info)        // ?
```

Value-based TMP - 🪄

```
🪄<std::is_constructible, T>(info) // ?
🪄<std::remove_cvref>(info)        // ?
```

```
template<class Fn, class T = decltype([]{})>
constexpr auto invoke(Fn fn, info m) {
```

```
}
```

```
template<template<class...> class T, class... Ts>
constexpr auto invoke(Fn fn, info m); // type_traits
```

```
static_assert(invoke<std::is_constructible, foo>(meta<int>)); // ✔
static_assert(meta<int> == invoke<std::remove_cvref>(meta<int&>)); // ✔
```

Value-based TMP - 🪄

```
🪄<std::is_constructible, T>(info) // ?
🪄<std::remove_cvref>(info)        // ?
```

```
template<class Fn, class T = decltype([]{})>
constexpr auto invoke(Fn fn, info m) {
```

```
  constexpr auto dispatch =
    [fn]<std::size_t... Ns>(std::index_sequence<Ns...>) {
      return std::array{&Fn::template operator<meta<void> + Ns>...};
    }(std::make_index_sequence<std::distance(meta<void>, meta<T>)>{});

  return dispatch[std::distance(meta<void>, m)](fn);
```

```
}
```

```
template<template<class...> class T, class... Ts>
constexpr auto invoke(Fn fn, info m); // type_traits
```

```
static_assert(invoke<std::is_constructible, foo>(meta<int>)); // ✔
static_assert(meta<int> == invoke<std::remove_cvref>(meta<int&>)); // ✔
```

Value-based TMP - variant_for

```
template<class T>
constexpr auto variant_for(const std::ranges::range auto& ts)
  -> std::ranges::range auto {
```

```
}
```

```
static_assert(std::vector{meta<int>} == variant_for<foo>(meta<int&>));
```

Value-based TMP - variant_for

```
template<class T>
constexpr auto variant_for(const std::ranges::range auto& ts)
  -> std::ranges::range auto {
```

```
  std::vector<info> r;
```

```
}
```

```
static_assert(std::vector{meta<int>} == variant_for<foo>(meta<int&>));
```

Value-based TMP - variant_for

```
template<class T>
constexpr auto variant_for(const std::ranges::range auto& ts)
  -> std::ranges::range auto {
```

```
  std::vector<info> r;
```

```
  for (auto t : ts) {
    if (invoke<std::is_constructible, T>(t)) {
      r.push_back(invoke<std::remove_cvref>(t));
    }
  }
```

```
}
```

```
static_assert(std::vector{meta<int>} == variant_for<foo>(meta<int&>));
```

Value-based TMP - variant_for

```
template<class T>
constexpr auto variant_for(const std::ranges::range auto& ts)
  -> std::ranges::range auto {
```

```
  std::vector<info> r;
```

```
  for (auto t : ts) {
    if (invoke<std::is_constructible, T>(t)) {
      r.push_back(invoke<std::remove_cvref>(t));
    }
  }
```

```
  std::ranges::sort(r);
  r.erase(std::ranges::unique(r), r.end());
```

```
}
```

```
static_assert(std::vector{meta<int>} == variant_for<foo>(meta<int&>));
```

Value-based TMP - variant_for

```
template<class T>
constexpr auto variant_for(const std::ranges::range auto& ts)
  -> std::ranges::range auto {
```

```
  std::vector<info> r;
```

```
  for (auto t : ts) {
    if (invoke<std::is_constructible, T>(t)) {
      r.push_back(invoke<std::remove_cvref>(t));
    }
  }
```

```
  std::ranges::sort(r);
  r.erase(std::ranges::unique(r), r.end());
```

```
  return r;
```

```
}
```

```
static_assert(std::vector{meta<int>} == variant_for<foo>(meta<int&>));
```

Value-based TMP - variant_for - std::ranges rafactor (C++20)

```
template<class T>
constexpr auto variant_for(const std::ranges::range auto& ts)
  -> std::ranges::range auto {
```

```
}
```

Value-based TMP - variant_for - std::ranges rafactor (C++20)

```
template<class T>
constexpr auto variant_for(const std::ranges::range auto& ts)
  -> std::ranges::range auto {
```

```
  auto&& r = ts
   | std::views::filter(is_constructible<T>)
   | std::views::transform(remove_cvref)
   | std::ranges::to<std::vector>()
   ;
```

```
}
```

```
template<class T> constexpr auto is_constructible = [](auto t) {
  return invoke<std::is_constructible, T>(t);
};
// ...
```

Value-based TMP - variant_for - std::ranges rafactor (C++20)

```
template<class T>
constexpr auto variant_for(const std::ranges::range auto& ts)
  -> std::ranges::range auto {
```

```
  auto&& r = ts
   | std::views::filter(is_constructible<T>)
   | std::views::transform(remove_cvref)
   | std::ranges::to<std::vector>()
   ;
```

```
  std::ranges::sort(r);
  r.erase(std::ranges::unique(r), r.end());
```

```
}
```

```
template<class T> constexpr auto is_constructible = [](auto t) {
  return invoke<std::is_constructible, T>(t);
};
// ...
```

Value-based TMP - variant_for - std::ranges rafactor (C++20)

```
template<class T>
constexpr auto variant_for(const std::ranges::range auto& ts)
  -> std::ranges::range auto {
```

```
  auto&& r = ts
   | std::views::filter(is_constructible<T>)
   | std::views::transform(remove_cvref)
   | std::ranges::to<std::vector>()
   ;
```

```
  std::ranges::sort(r);
  r.erase(std::ranges::unique(r), r.end());
```

```
  return r;
```

```
}
```

```
template<class T> constexpr auto is_constructible = [](auto t) {
  return invoke<std::is_constructible, T>(t);
};
// ...
```

❓ Will 'std::ranges' refactor be faster or slower to compile?

Value-based TMP - variant_for_t (pseudo code)

Value-based TMP - variant_for_t (pseudo code)

```
template<class T, class... Ts>
using variant_for_t =
  🪄<std::variant, variant_for<T>(std::array{meta<Ts>...})>
```

```
static_assert(std::is_same_v<
  std::variant<int, bar>,
  variant_for_t<foo, const int&, int&&, std::string_view, bar, void>
>);
```

Value-based TMP - 🪄

```
🪄<std::variant, variant_for<T>(std::array{meta<Ts>...})> // ?
```

Value-based TMP - 🪄

```
🪄<std::variant, variant_for<T>(std::array{meta<Ts>...})> // ?
```

```
template<template<class...> class T, std::ranges::range auto range>
using apply_t =
```

```
static_assert(std::is_same_v<
  variant<int, bar>,
  apply_t<variant, std::array{meta<int>, meta<bar>}>
>);
```

Value-based TMP - 🪄

```
🪄<std::variant, variant_for<T>(std::array{meta<Ts>...})> // ?
```

```
template<template<class...> class T, std::ranges::range auto range>
using apply_t =
```

```
  decltype(
    []<std::size_t... Ns>(std::index_sequence<Ns...>) {
      return std::declval<T<type_of<range[Ns]>...>>();
    }(std::make_index_sequence<range.size()>{})
  );
```

```
static_assert(std::is_same_v<
  variant<int, bar>,
  apply_t<variant, std::array{meta<int>, meta<bar>}>
>);
```

Value-based TMP - 🪄

```
🪄<std::variant, variant_for<T>(std::array{meta<Ts>...})> // ?
```

```
template<template<class...> class T, std::ranges::range auto range>
using apply_t =
```

```
  decltype(
    []<std::size_t... Ns>(std::index_sequence<Ns...>) {
      return std::declval<T<type_of<range[Ns]>...>>();
    }(std::make_index_sequence<range.size()>{})
  );
```

```
static_assert(std::is_same_v<
  variant<int, bar>,
  apply_t<variant, std::array{meta<int>, meta<bar>}>
>);
```

```
apply_t<std::variant, variant_for<T>(std::array{meta<Ts>...})> // ✔
```

Value-based TMP - variant_for_t

```
template<class T, class... Ts>
using variant_for_t =
  apply_t<std::variant, variant_for<T>(std::array{meta<Ts>...)>;
```

Value-based TMP - variant_for_t

```
template<class T, class... Ts>
using variant_for_t =
  apply_t<std::variant, variant_for<T>(std::array{meta<Ts>...)>;
```

```
struct bar { };
struct foo {
  foo(int) { }
  foo(bar) { }
};

static_assert(std::is_same_v<
  std::variant<int, bar>,
  variant_for_t<foo, const int&, int&&, std::string_view, bar, void>
>); // ✔
```

Value-based TMP - testing/debugging/coverage/errors

```
constexpr auto test(auto fn) {
  static_assert((fn(), true)); // compile-time (no UB, no leaks, ...)
  fn();                        // run-time (debugging, coverage)
};
```

Value-based TMP - testing/debugging/coverage/errors

```
constexpr auto test(auto fn) {
  static_assert((fn(), true)); // compile-time (no UB, no leaks, ...)
  fn();                        // run-time (debugging, coverage)
};
```

```
int main() {
  test([] {
    // given
    std::vector<info> v;
    v.push_back(meta<const int&>);
    v.push_back(meta<int&>);

    // when
    const auto r = variant_for<foo>(v);

    // then
    assert(1u == r.size());
    assert(meta<float> == r[0]); // error: meta<float> == meta<int>
  });
}
```

Value-based TMP - Summary

```
Easy ✅
Testable ✅
Debuggable ✅
Code coverage ✅
Nicer error messages 📈
```

 - https://godbolt.org/z/Kf9rovaqE

100 LOC / C++17 / gcc, clang, msvc / no dependencies

---

https://github.com/qlibs/mp

Reflection for C++26* - https://wg21.link/p2996

Value-based Metaprogramming

Meta functions for Reflection (introspection, generation)

Reflection for C++26* - https://wg21.link/p2996

```
^^T // reflection operator (reflexpr)
```

```
static_assert(^^int == ^^int);
static_assert(^^int != ^^void);
static_assert(typeid(^^int) == typeid(^^void));
```

Reflection for C++26* - https://wg21.link/p2996

```
^^T // reflection operator (reflexpr)
```

```
static_assert(^^int == ^^int);
static_assert(^^int != ^^void);
static_assert(typeid(^^int) == typeid(^^void));
```

```
[: ... :] // splicer operator (unreflexpr)
```

```
typename [: ^^int :] i = 42; // int i = 42;
```

```
static_assert(typeid([: ^^int :]) == typeid(int));
```

Reflection for C++26* - https://wg21.link/p2996

this_talk (C++17)	p2996 (C++26*)
`meta<T>`	`^^T`
`using info = decltype(meta<void>)`	`using info = decltype(^^::)`
`type_of<T>`	`typename [: T :]`
`apply_t`	`substitute`
`invoke`	`reflect_invoke, extract, expand`

Reflection for C++26* - https://wg21.link/p2996

```
constexpr auto find_index(auto t, const std::ranges::range auto& ts)
  -> std::size_t {
```

```
}
```

```
static_assert(
  0u == find_index(^^int,   std::array{^^int, ^^float, ^^short}) and
  1u == find_index(^^float, std::array{^^int, ^^float, ^^short}) and
  2u == find_index(^^short, std::array{^^int, ^^float, ^^short}) and
  3u == find_index(^^void,  std::array{^^int, ^^float, ^^short})
); // ✔
```

Reflection for C++26* - https://wg21.link/p2996

```
constexpr auto find_index(auto t, const std::ranges::range auto& ts)
  -> std::size_t {
```

```
  if (const auto found = std::ranges::find(ts, t); found) {
    return std::distance(v.begin(), found);
  }
  return ts.size();
```

```
}
```

```
static_assert(
  0u == find_index(^^int,   std::array{^^int, ^^float, ^^short}) and
  1u == find_index(^^float, std::array{^^int, ^^float, ^^short}) and
  2u == find_index(^^short, std::array{^^int, ^^float, ^^short}) and
  3u == find_index(^^void,  std::array{^^int, ^^float, ^^short})
); // ✔
```

Reflection for C++26* - https://wg21.link/p2996

```
template<std::size_t I, class... Ts>
struct tuple_element<I, std::tuple<Ts...>> {
```

```
};
```

```
static_assert(
 std::is_same_v<int, tuple_element<0, std::tuple<int, bool>>::type> and
 std::is_same_v<bool,tuple_element<1, std::tuple<int, bool>>::type> and
);
```

Reflection for C++26* - https://wg21.link/p2996

```
template<std::size_t I, class... Ts>
struct tuple_element<I, std::tuple<Ts...>> {
```

```
  using type = typename [: array{^^Ts...}[I] :];
```

```
};
```

```
static_assert(
 std::is_same_v<int, tuple_element<0, std::tuple<int, bool>>::type> and
 std::is_same_v<bool,tuple_element<1, std::tuple<int, bool>>::type> and
);
```

Reflection for C++26* - https://wg21.link/p2996

```
constexpr auto variant_for(auto t, const std::ranges::range auto& ts)
  -> std::ranges::range auto {
```

```
}
```

Reflection for C++26* - https://wg21.link/p2996

```
constexpr auto variant_for(auto t, const std::ranges::range auto& ts)
  -> std::ranges::range auto {

  auto&& r = ts
   | std::views::filter([](auto m) {
       return std::meta::test_trait(^^std::constructible_from, {t, m});
     })
   | std::views::transform(std::meta::type_remove_cvref)
   | std::ranges::to<std::vector>()
   ;

  std::ranges::sort(r, [](auto lhs, auto rhs) {
    return std::meta::identifier_of(lhs) <
           std::meta::identifier_of(rhs);
  });
  r.erase(std::ranges::unique(r), r.end());

  return r;

}
```

Reflection for C++26* - https://wg21.link/p2996

```
template<class T, class... Ts>
using variant_for_t =
```

```
static_assert(std::is_same_v<
  std::variant<int, bar>,
  variant_for_t<foo, const int&, int&&, std::string_view, bar, void>
>);
```

Reflection for C++26* - https://wg21.link/p2996

```
template<class T, class... Ts>
using variant_for_t =
```

```
  typename [:
    std::meta::substitute(
      ^^std::variant, fn(^^T, std::vector{^^Ts...})
    )
  :];
```

```
static_assert(std::is_same_v<
  std::variant<int, bar>,
  variant_for_t<foo, const int&, int&&, std::string_view, bar, void>
>);
```

Circle-lang - https://www.circle-lang.org

Member packs - https://wg21.link/p1858

Circle-lang meta model (Metaprogramming)

Reflection

...

Circle-lang - https://www.circle-lang.org

```
template<class T, class... Ts>
constexpr auto find_index() {



}
```

```
static_assert(0u == find_index<int,  int, float, short>());
static_assert(1u == find_index<int,  float, int, short>());
static_assert(2u == find_index<int,  float, short, int>());
static_assert(3u == find_index<void, float, short, int>());
```

Circle-lang - https://www.circle-lang.org

```
template<class T, class... Ts>
constexpr auto find_index() {

  return T == Ts... ?? int... : sizeof...(Ts);

}
```

```
static_assert(0u == find_index<int,  int, float, short>());
static_assert(1u == find_index<int,  float, int, short>());
static_assert(2u == find_index<int,  float, short, int>());
static_assert(3u == find_index<void, float, short, int>());
```

??: constexpr conditional - https://github.com/seanbaxter/circle/blob/master/conditional/README.md

Circle-lang - https://www.circle-lang.org

```
template<std::size_t I, class... Ts>
struct tuple_element<I, std::tuple<Ts...>> {
```

```
};
```

```
static_assert(
 std::is_same_v<int, tuple_element<0, std::tuple<int, bool>>::type> and
 std::is_same_v<bool,tuple_element<1, std::tuple<int, bool>>::type> and
);
```

Circle-lang - https://www.circle-lang.org

```
template<std::size_t I, class... Ts>
struct tuple_element<I, std::tuple<Ts...>> {

  using type = Ts...[I];

};
```

```
static_assert(
 std::is_same_v<int, tuple_element<0, std::tuple<int, bool>>::type> and
 std::is_same_v<bool,tuple_element<1, std::tuple<int, bool>>::type> and
);
```

Comprehension - https://github.com/seanbaxter/circle/blob/master/comprehension/README.md

Circle-lang - https://www.circle-lang.org

```
template<class T, class... Ts>
using variant_for_t = std::variant<
```

```
>;
```

```
static_assert(std::is_same_v<
  std::variant<int, bar>,
  variant_for_t<foo, const int&, int&&, std::string_view, bar, void>
>);
```

Circle-lang - https://www.circle-lang.org

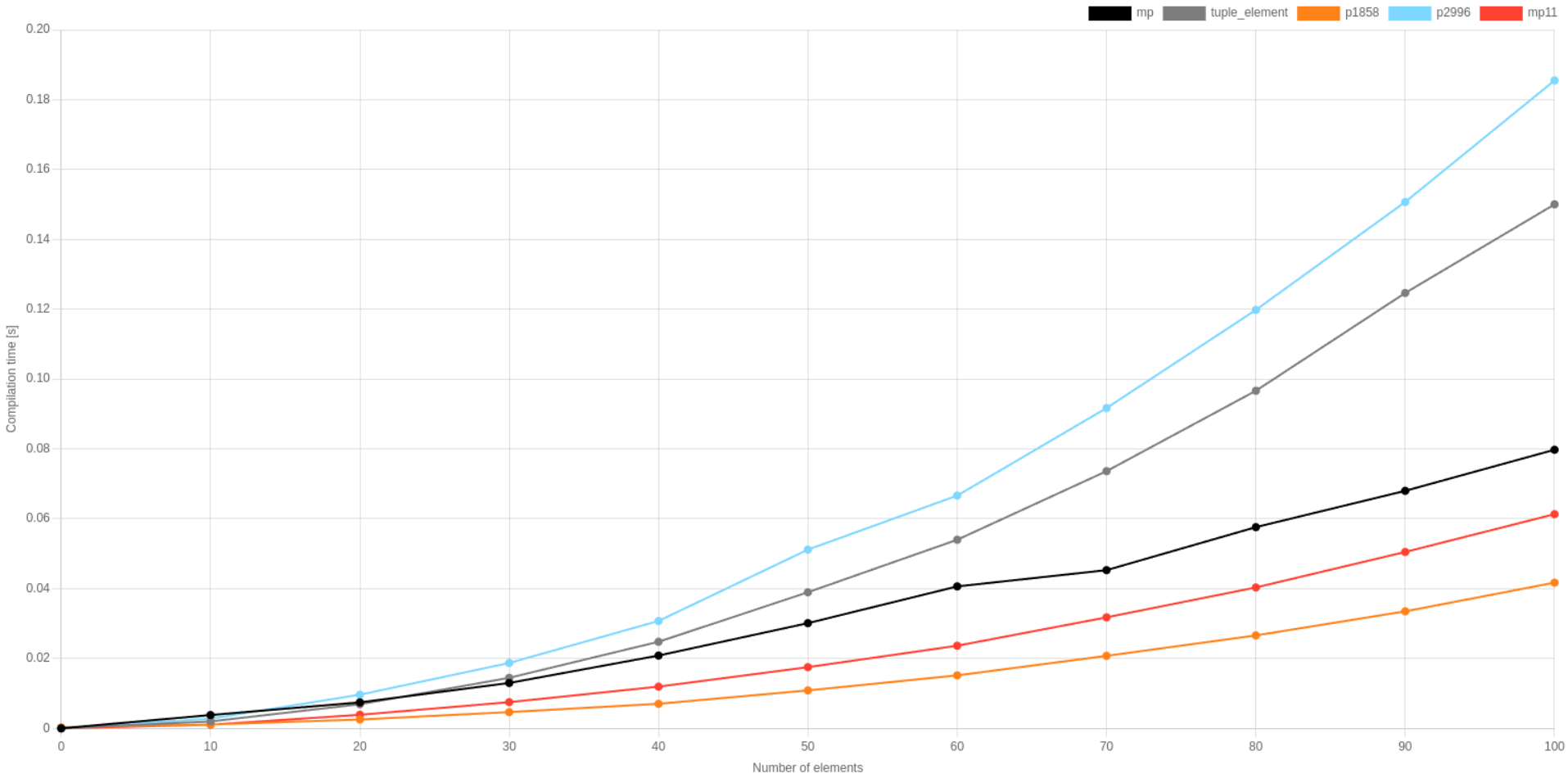
```
template<class T, class... Ts>
using variant_for_t = std::variant<

  (if std::is_constructible_v<T, Ts> => Ts.remove_cvref...).unique...

>;
```

```
static_assert(std::is_same_v<
  std::variant<int, bar>,
  variant_for_t<foo, const int&, int&&, std::string_view, bar, void>
>);
```

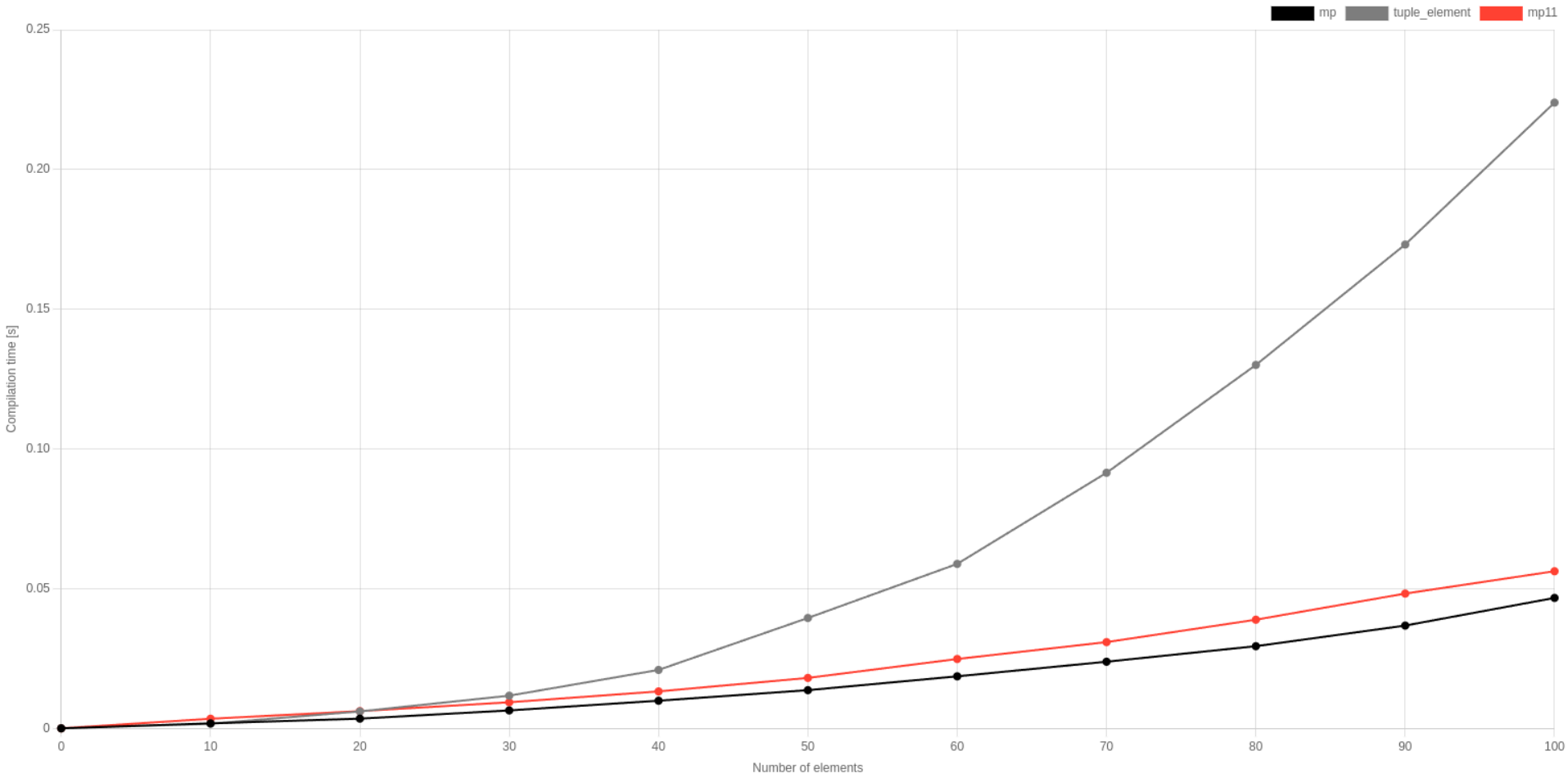
Imperative arguments - https://github.com/seanbaxter/circle/blob/master/imperative/README.md

Benchmarks - https://qlibs.github.io/mp

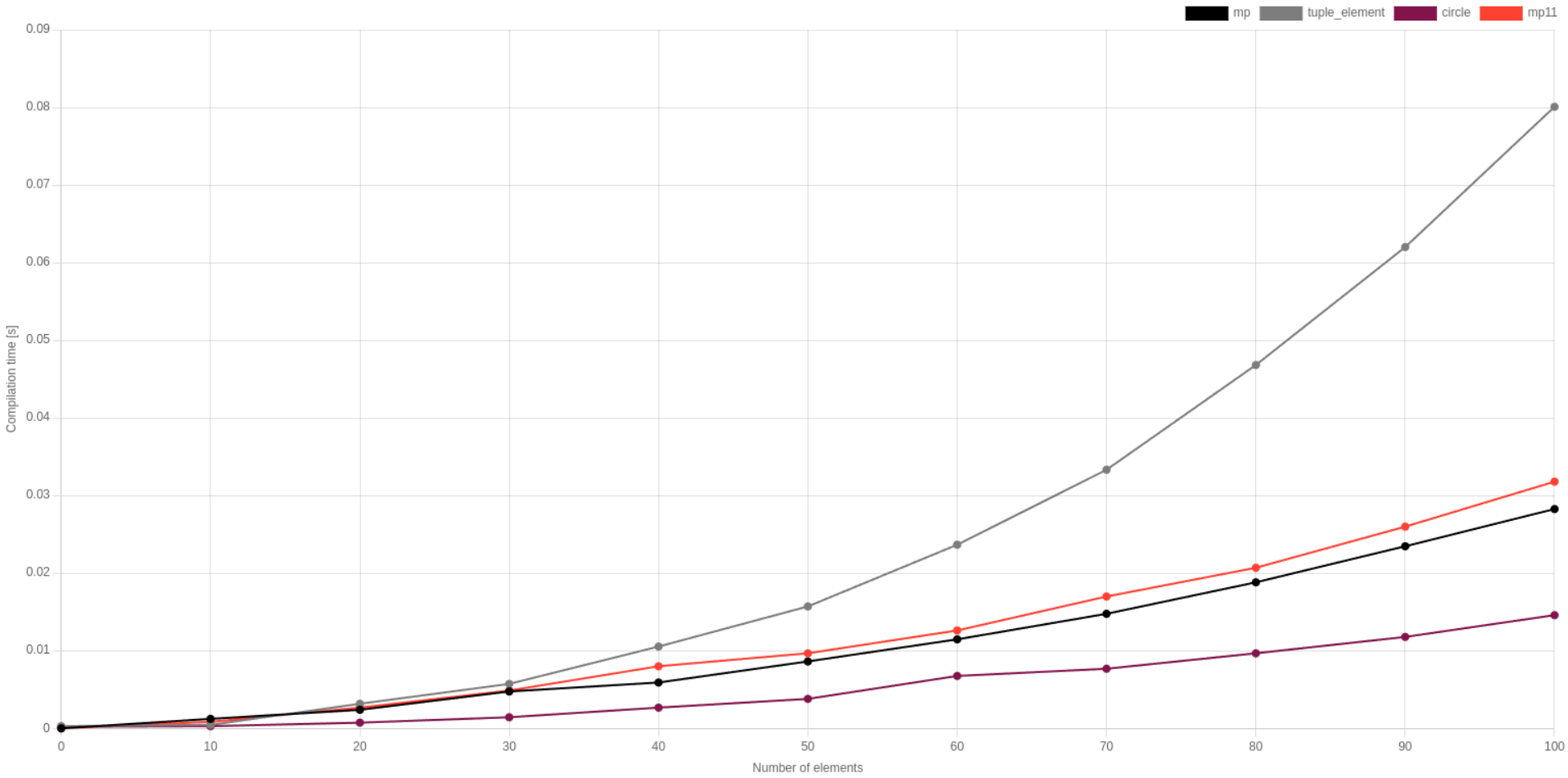
clang-p2996 -stdlib=libc++ -std=c++2c -freflection | at | includes | 100 | Compare | benchmarks/ | results/ |



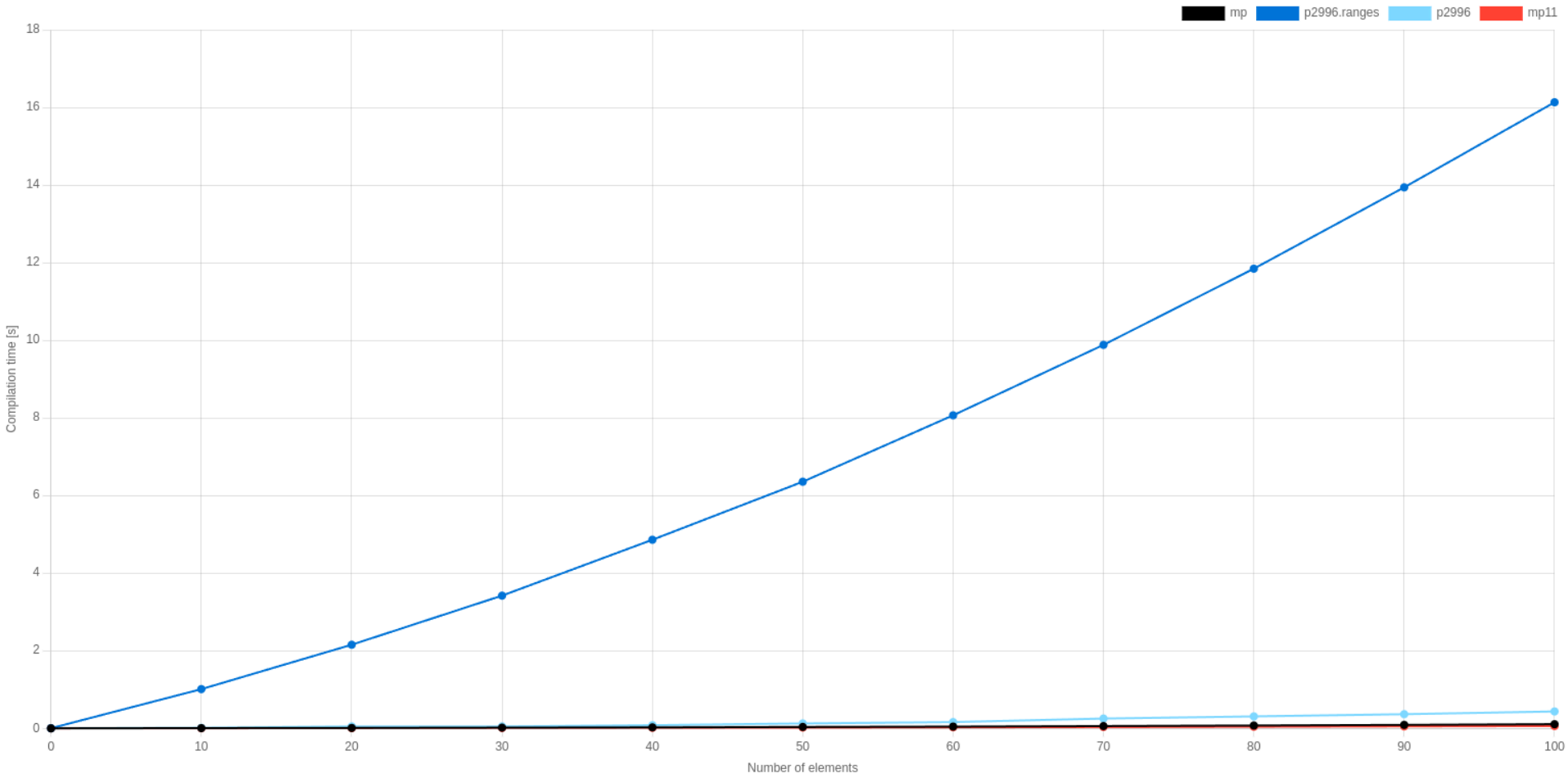
g++-14 -std=c++20 | at | includes | 100 | Compare | benchmarks/ | results/ |



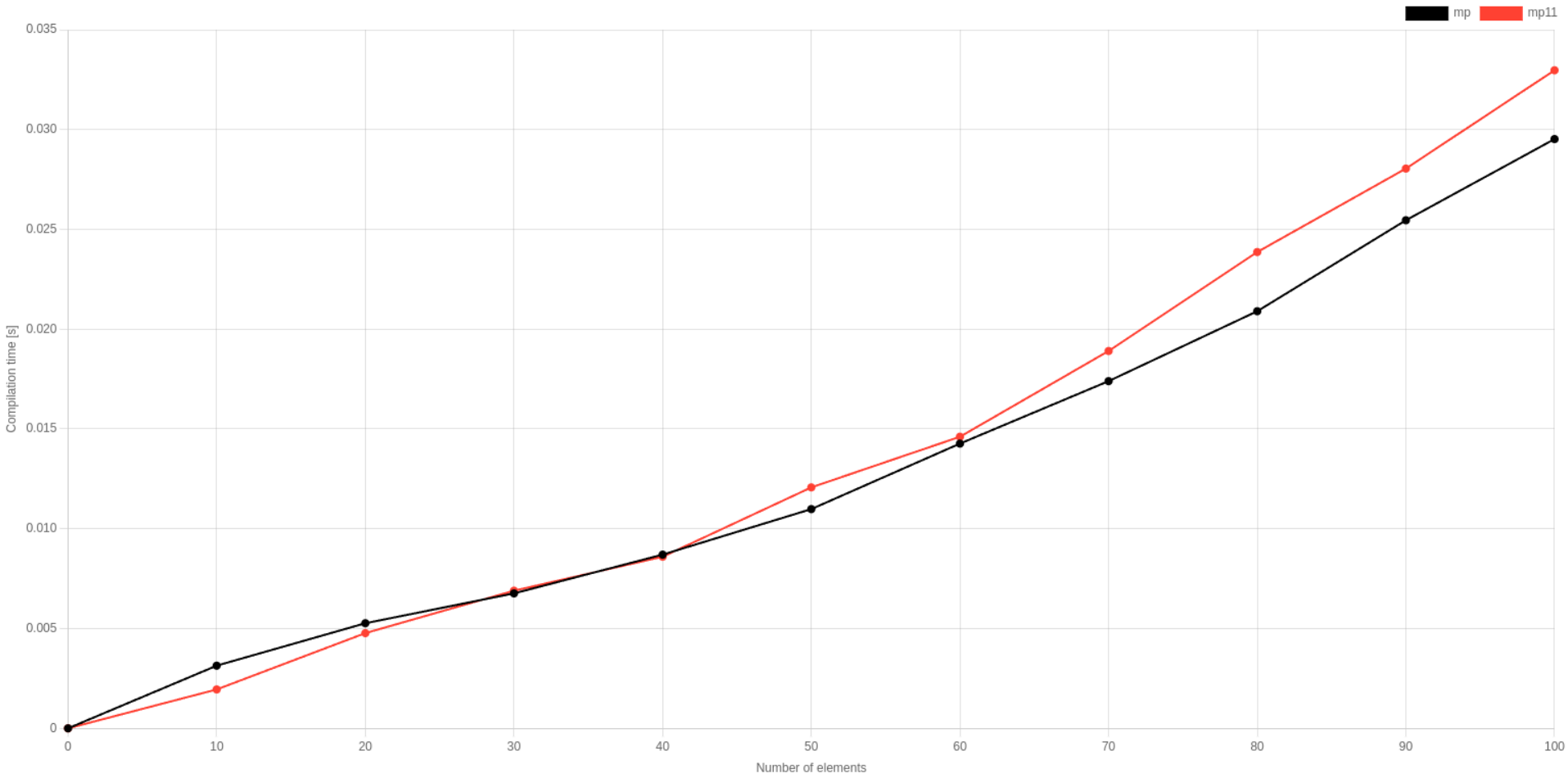
circle-214 -std=c++20 | at | includes | 100 | Compare | benchmarks/ | results/ |

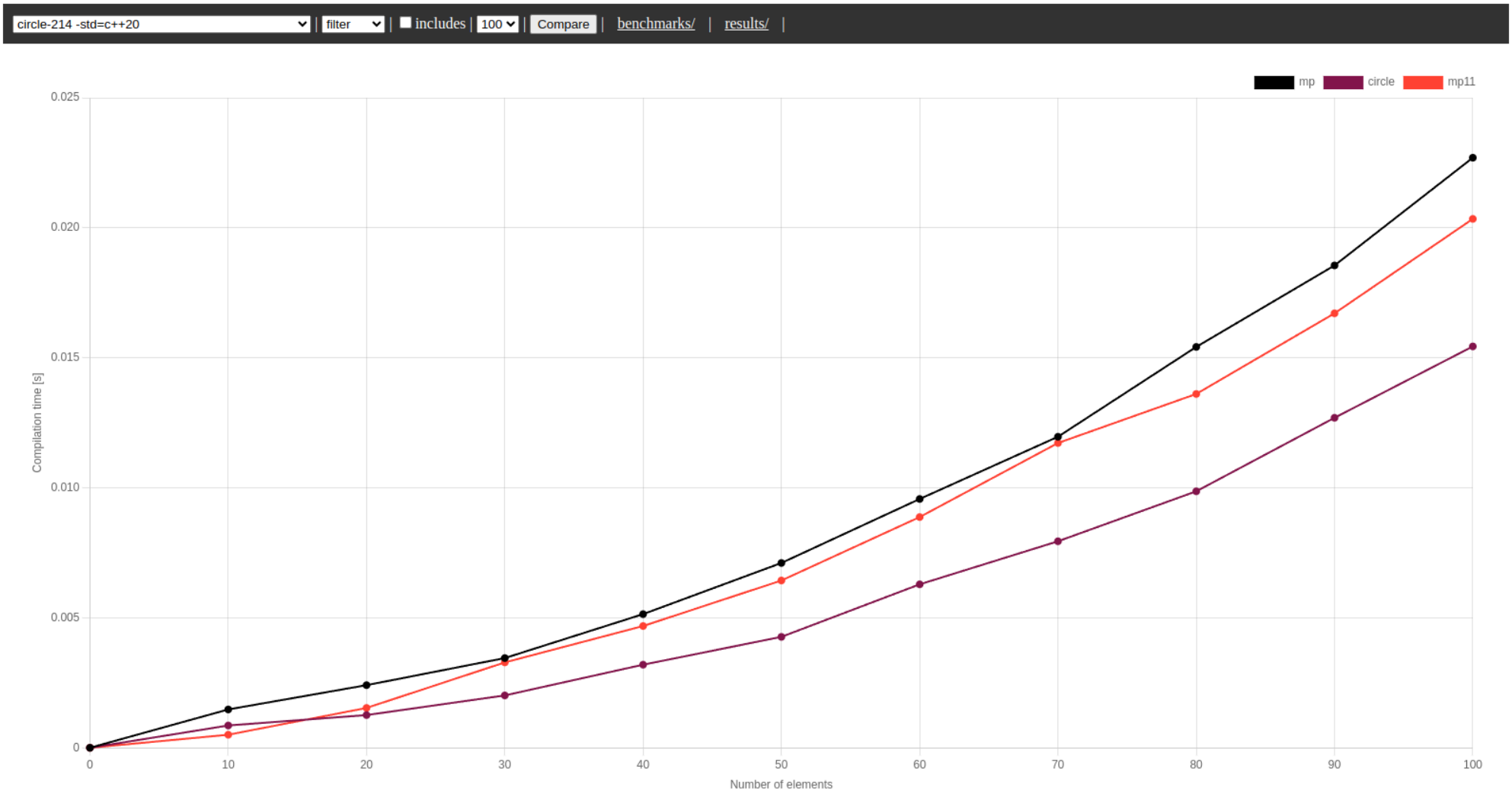


clang-p2996 -stdlib=libc++ -std=c++2c -freflection | filter | includes | 100 | Compare | benchmarks/ | results/ |



g++-14 -std=c++20 | filter | includes | 100 | Compare | benchmarks/ | results/ |




circle-214 -std=c++20
filter
includes
100
Compare
benchmarks/
results/
mp
circle
mp11
0.025
0.020
0.015
0.010
0.005
0
Compilation time [s]
0
10
20
30
40
50
60
70
80
90
100
Number of elements

Benchmarks

Benchmarks

Circle-lang meta model is the fastest to compile all around

Benchmarks

Circle-lang meta model is the fastest to compile all around

Type-based Metaprogramming with template aliases/builtins (boost.mp11) is much faster to compile than recursive template instantiations (std::tuple)

Benchmarks

Circle-lang meta model is the fastest to compile all around

Type-based Metaprogramming with template aliases/builtins (boost.mp11) is much faster to compile than recursive template instantiations (std::tuple)

Value-based Metaprogramming with STL is significantly slower to compile than with raw primitives!

Benchmarks

Circle-lang meta model is the fastest to compile all around

Type-based Metaprogramming with template aliases/builtins (boost.mp11) is much faster to compile than recursive template instantiations (std::tuple)

Value-based Metaprogramming with STL is significantly slower to compile than with raw primitives!

Value-based Metaprogramming has a lot of potential (std::simd, std::execution, ...) but constexpr evaluation has to be JITTED instead of INTERPRETED

Summary

"Better Metaprogramming features make better libraries!"

Sean Baxter

Further readings

- C++20 Metaprogramming library - https://github.com/qlibs/mp
- Reflection for C++26* - https://wg21.link/P2996
- Implementing P2996 Metaprogramming model with P2996 - https://godbolt.org/z/694e1hrbM
- Circle-lang Metaprogramming - Sean Baxter - CppNow 2022 - https://www.youtube.com/watch?v=15j4bkipuAg
- Zig-lang comptime - https://ziglang.org/documentation/master/#comptime

---

kris@jusiak.net | @krisjusiak | linkedin.com/in/kris-jusiak