P0949R0

Adding support for type-based metaprogramming to the standard library

Peter Dimov

History: Pre-MPL (2000-2001)

- "Generative Programming: Methods, Tools, and Applications", Krysztof Czarnecki, Ulrich Eisenecker (2000)
- Loki library's `Typelist<Head, Tail>`, Andrei Alexandrescu (2001)

Boost.MPL (2002-2004)

- http://boost.org/libs/mpl
- Dave Abrahams and Alexey Gurtovoy, 2002-2004
- `list<T1, list<T2, list<T3, nil>>>`
- … but also `vector<T1 = na, T2 = na, ⋯, T20 = na>` and others
- "Generic" algorithms a-la STL
- Metafunction `typename F<Args>::type`
- Metafunction class `typename FC::template apply<Args>::type`

MPL in C++11

- MPL is a remarkable achievement within the constraints of C++03
- But… C++03 really isn't for metaprogramming; C++11 is though
- Various attempts to rewrite MPL in C++11
- F.ex. Louis Dionne's Mpl11, https://github.com/ldionne/mpl11
- All abandoned in favor of more recent approaches

Eric Niebler's meta (2014)

- An implementation detail of Range-v3
- Described in "Tiny Metaprogramming Library", http://ericniebler.com/2014/11/13/tiny-metaprogramming-library/
- `template <class··· T> struct typelist {};`
- Algorithms only work on `typelist<T···>`
- Metafunctions have to be "quoted", f.ex. `meta_quote<typelist_size_t>`

"Simple C++ metaprogramming" (2015)

- http://pdimov.com/cpp2/simple_cxx11_metaprogramming.html
- `L<T···>` for any `L`
- `template<class··· T> struct mp_list {};`
- … but also works on `std::tuple`, `std::variant`, `packer` from N4115, `your_list<T···>`
- … and even `std::pair`, where appropriate
- Metafunctions are `F<T···>`
  - that is, `std::add_pointer_t` works as-is, without need for quoting
  - and so does f.ex. `std::pair`

"Simple C++ metaprogramming, part 2" (2015)

- http://pdimov.com/cpp2/simple_cxx11_metaprogramming_2.html
- Describes efficient algorithms for vector, set, map access to `L<T···>`
- No separate data structures needed

Cambrian explosion

- Boost.Hana (Louis Dionne), http://boost.org/libs/hana
- Brigand (Edouard Aligand & Joel Falcou), https://github.com/edouarda/brigand
- Metal (Bruno Dutra), https://github.com/brunocodutra/metal
- Kvasir.MPL, https://github.com/kvasir-io/mpl
- And quite possibly many others of which I don’t know

Boost.Mp11

- http://boost.org/libs/mp11 (since Boost 1.66)
- Refinement of "Simple C++ metaprogramming"
- Adds quoted metafunctions, `Q::template fn<T···>`
  - produced by `mp_quote`, `mp_bind`
  - all algorithms taking metafunctions have `_q` versions

Need a standard facility

- Greenspun's Tenth Rule: *"Any sufficiently complicated C or Fortran program contains an ad-hoc, informally-specified, bug-ridden, slow implementation of half of Common Lisp."*
- Could easily replace "Common Lisp" with a "metaprogramming library", remains as valid as ever
- Many existing libraries, field well-understood
- Yet people still reimplement the same functionality over and over
- The standard library implementers need it too
- Bits and pieces proposed every now and then, no coherence, no generality
  - F.ex. proposed `tuple_index`, which is `mp_find`, but limited to tuples

This Proposal

- Based on Mp11
- Rather, virtually the same as Mp11 (with a few omissions)
- No innovation over Mp11
- Everything proposed is implemented and tested
- https://github.com/boostorg/mp11
- Design principle: *keep simple uses simple*
- `tuple<int, float>` → `tuple<int&, float&>` is done with `mp_transform<add_reference_t, Tp>`
- Existing entities (`tuple`, `add_reference_t`) directly usable without adaptation

Concepts

- *List* (`L<T···>` for any class template `L` where `T···` are types)
- *Metafunction* (`F<T···>`, an alias or a class template)
- *Quoted metafunction* (`Q::fn<T···>`, a class)
- *Set*, a list whose elements are unique
- *Map*, a list of pairs
  - more generally, a list of lists, with the inner lists having at least one element (the key)
- Proposal includes algorithms such as `mp_sort` and utility components such as `mp_if`

List-agnostic

- Provides a canonical `mp_list`, but does not require it
- All operations work on any template class whose parameters are types
- Such as `std::tuple<int>`, `std::pair<void, float>`…
- … `template<class T1, class T2> struct pumpkin;`
- `mp_reverse<pumpkin<int, float>>` → `pumpkin<float, int>`
- Returns whatever is passed
- When more than one, returns the first
- `mp_append<std::tuple<void>, pumpkin<int, float>>` → `std::tuple<void, int, float>`
- Within limits; `mp_append<pumpkin<int, float>, std::tuple<void>>` → ill-formed, `pumpkin<int, float, void>` not possible
- `mp_append<>` → `mp_list<>` (had to pick something)

Metafunctions

- Anything that matches `template<class⋯> class F`
- Alias templates: `std::add_pointer_t`
- But also class templates: `std::is_same`, `std::pair`
- *Keep simple uses simple*
- `tuple<int, float>` → `tuple<int&, float&>` is done with `mp_transform<add_reference_t, Tp>`
- Not with something like `unpack_sequence_t<transform<quote<add_reference_t>, as_typelist<Tp>>, tuple>`
- `mp_transform<F, L>` takes `F` first, because…
- .. it's actually `mp_transform<F, L⋯>` and works on many lists, not just one (piecewise)
- With `mp_list` for `F`, it performs a "zip" (transpose) operation
  - `mp_transform<mp_list, mp_list<X1, X2>, mp_list<Y1, Y2>>` → `mp_list<mp_list<X1, Y1>, mp_list<X2, Y2>>`

Quoted metafunctions

- Metafunctions keep simple uses simple, but
  - you can’t store them in `mp_list`
  - you can’t return them from a metafunction
  - the language has an annoying limitation, expanding a template parameter pack into a fixed parameter list is disallowed in certain contexts
- Hence, quoted metafunctions, types with a nested `fn` metafunction member
- Created by `mp_quote<F>`, evaluated by `mp_invoke<Q, T···> = typename Q::template fn<T···>`
- Also returned by higher-order operations such as `mp_bind`, `mp_bind_front`, `mp_bind_back`
- All operations taking a metafunction, such as `mp_transform`, have a variant with a `_q` suffix (`mp_transform_q`) taking a quoted metafunction
- When you get the "can’t expand into a fixed parameter list" error, try quoting the metafunction and using the `_q` algorithm instead

List operations and algorithms

- `mp_push_front`, `mp_reverse`, `mp_append`, `mp_sort`, `mp_find`…
  - and many more
- Efficient random access with `mp_at` for any list, no separate "vector" needed
  - not so impressive today when there's an intrinsic for it (`__type_pack_element`)
- Generally named after their equivalent STL algorithms…
  - except `mp_fold` is not `mp_accumulate` because really
  - and when there's no STL equivalent a Common Lisp name is used for nostalgia points (`mp_append`, `mp_cond`)
  - and sometimes the established name (`mp_take`, `mp_drop`) in the metaprogramming field is used, which often comes from Haskell

Naming

- Consistent use of `mp_` prefix allows coexistence with similarly-named parts of the standard library (`sort`) and keywords (`if`, `bool`)
- Algorithms and operations taking an integral nontype template parameter have a `_c` suffix
  - `template<class L, size_t I> using mp_at_c`
  - `template<class L, class I> using mp_at = mp_at_c<L, size_t{I::value}>`
- `size_t{I::value}` causes a substitution failure when `I::value` is not convertible to `size_t` without narrowing
  - such as for instance when it's `-1`

Naming (cont.)

- Algorithms and operations taking a metafunction have a form with a `_q` suffix taking a quoted metafunction
  - `template<template<class⋯> class F, class⋯ L> using mp_transform`
  - `template<class Q, class⋯ L> using mp_transform_q = mp_transform<Q::template fn, L⋯>;`
- `_c` and `_q` never appear together because `_q` operations and algorithms only have type parameters
- … which makes the `_q` forms valid metafunctions

Set operations

- A set is any list whose elements are distinct, f.ex. `tuple<int, float, double>`
- No separate data structure
- `mp_set_contains`, `mp_set_push_back`, `mp_set_push_front`
- `mp_set_contains` is more efficient than `mp_contains` but ill-formed when the argument is not a set
  - `mp_set_contains<L<T···>, U> = is_base_of<mp_identity<U>, mp_inherit<mp_identity<T>···>>`
- An arbitrary list can be turned into a set by removing duplicates via `mp_unique`
  - `mp_unique<L<T···>>` is incidentally `mp_set_push_back<L<>, T···>`

Map operations

- A map is usually a list of pairs, `mp_list<pair<K1, V1>, pair<K2, V2>>`
- `Ki` must be distinct; the list of the map keys (`mp_map_keys<M>`) is a set
- Signature operation is lookup by key (`mp_map_find<M, K>`)
- In general, the list elements may also be lists having at least one element
  - such as `mp_list<mp_list<K1, V1>, mp_list<K2>, mp_list<K3, V3, W3>>`
- `mp_map_contains`, `_insert`, `_replace`, `_update`, `_erase`

Integral constants

- `mp_int<I> = integral_constant<int, I>`
- `mp_size_t<N> = integral_constant<size_t, N>`
- `mp_bool<B> = integral_constant<bool, B>`
- `mp_true = mp_bool<true>`, `mp_false = mp_bool<false>`
  - Same as `bool_constant`, `true_type`, `false_type`, but provided for consistency
- `mp_to_bool<T> = mp_bool<static_cast<bool>(T::value)>`
- `mp_not<T> = mp_bool< !T::value>`

Utilities

- `mp_identity`, `mp_identity_t`, `mp_inherit`, ...
- `mp_if<C, T, E>`, `mp_if_c<C, T, E>`, ...
  - `mp_if_c` same as `conditional_t`, provided for consistency
- `mp_valid<F, T···>` - `mp_true` if `F<T···>` is valid
  - Same as `is_detected<F, T···>`, provided for consistency
- `mp_defer<F, T···>` - has nested `type = F<T···>` when valid, no `type` otherwise
  - Very useful for making SFINAE-friendly traits, among other things
- `mp_quote`, `mp_quote_trait`, `mp_invoke`

Helper metafunctions

- `mp_all<T···> = mp_bool<(T::value && ···)>`
- `mp_and<T···>` - as above, but with short-circuiting
- `mp_any<T···> = mp_bool<(T::value || ···)>`
- `mp_or<T···>` - as above, with short-circuiting
- `mp_same<T···>` - `mp_true` when all types same
- `mp_plus<T···>` - integral constant with `value = (T::value + ··· + 0)`
- `mp_less<T1, T2>` - as `mp_bool<(T1::value < T2::value)>` but compares signed/unsigned properly
- `mp_min<T···>`, `mp_max<T···>`

Bind

- `mp_bind`: same as `std::bind` but for types
  - `mp_bind<mp_less, mp_bind<alignment_of, _1>, mp_bind<alignment_of, _2>>`
  - `mp_bind<mp_identity_t, X>` → a quoted metafunction that returns `X`, sometimes spelled `always<X>` in other libraries
- `mp_bind_front`, `mp_bind_back`: like `bind_front` and `bind_back` from P0356
  - `mp_bind_front<F, T···>::fn<U···>` → `F<T···, U···>`
  - `mp_bind_back<F, T···>::fn<U···>` → `F<U···, T···>`
  - P0356 cites Eric Niebler's 2014 post, which has `meta_bind_front` and `meta_bind_back`

Conclusion

- Need standard facilities for type manipulation
- Proposal based on Boost.Mp11
  - implemented, performant, well tested
  - works out of the box on existing types and entities
- Standard primitives open door to compiler support
  - with associated performance gains