Variadic Templates

"Look Ma, no recursion!"

Daniel Frey

2015-11-12

Disclaimer

- Opinions expressed are solely my own and do not express the views or opinions of my employer.
- All mistakes are mine, please point them out.
- When taking a note for a question, please write down the slide number!

Rationale

- Variadic templates and template meta programming are used everywhere.
- `std::tuple` is the most obvious example.
- Using `std::tuple` can increase compile-time significantly.
- Used behind the scenes of other APIs, e.g., a database framework.

C++11

- This talk is C++11 or newer.
- Tested with GCC/Clang.
- I omitted `std::` qualifications.
- No error detection/handling, real-world code is slightly longer due to `static_assert()`, etc.

C++14

- Some C++14 helpers are used, implementations for C++11 are available.
- `something_t<…>` is short for `typename something<…>::type`.
- http://cppreference.com

Type Traits

```
#include <type_traits>
```

Most important for this talk:

```
integral_constant<T, N>

true_type
false_type

is_same<T, U>
```

Type Traits

```
template<typename T, T V>
struct integral_constant
{
  static constexpr T value = V;
};
```

The above skips some standardized members.

Type Traits

```
template<typename T, T V>
struct integral_constant
{
  static constexpr T value = V;
};

// Standardized in C++17
template<bool B>
using bool_constant =
  integral_constant<bool, B>;
```

Type Traits

```
using true_type =
  bool_constant<true>;

using false_type =
  bool_constant<false>;
```

```
true_type::value → true

false_type::value → false
```

Type Traits

```
template<typename, typename>
struct is_same : false_type {};

template<typename T>
struct is_same<T, T> : true_type {};
```

```
is_same<int, int>::value → true

is_same<int, double>::value → false
```

Variadic Templates

```
template<typename...>
struct typelist {};



typelist<int, double, long>
```

Variadic Templates

```
template<typename...>
struct typelist {};



typelist<int, double, long>

typelist<>

typelist<int>
```

Variadic Templates

```
template<int...>
struct intlist {};
```

```
intlist<1, -3, 42, 1701>
```

Variadic Templates

```
template<bool...>
struct boollist {};
```

```
boollist<true, false, false>
```

Logical And

```
template<bool...>
struct is_all;

template<bool B, bool... Bs>
struct is_all<B, Bs...>
  : bool_constant<
      B && is_all<Bs...>::value>
{};

template<>
struct is_all<> : true_type {};
```

Logical And

```
template<int... Is>
using is_all_even =
  is_all<(Is % 2 == 0)...>;
```

```
is_all_even<0, 2, 42>::value ➞ true

is_all_even<4, 5, 6>::value ➞ false
```

Logical And

```
template<int... Is>
using is_all_even =
  is_all<(Is % 2 == 0)...>;
```

```
is_all_even<0, 2, 42>::value → true

is_all_even<4, 5, 6>::value → false
```

Logical And

```
template<int... Is>
using is_all_even =
  is_all<(Is % 2 == 0)...>;
```

```
is_all_even<0, 2, 42>::value → true

is_all_even<4, 5, 6>::value → false
```

Logical And

```
template<bool...>
struct is_all : true_type {};

template<bool B, bool... Bs>
struct is_all<B, Bs...>
  : bool_constant<
        B && is_all<Bs...>::value>
{};
```

Logical And

```
// Recursive solution
template<bool...>
struct is_all : true_type {};

template<bool B, bool... Bs>
struct is_all<B, Bs...>
  : bool_constant<
      B && is_all<Bs...>::value>
{};
```

Logical And

```
// The idea
template<bool... Bs>
using is_all =
  is_same<boollist<Bs...>,
          boollist<true...>>;
```

Logical And

```
// The idea
template<bool... Bs>
using is_all =
  is_same<boollist<Bs...>,
          boollist<true...>>;
```

Nope, does not work! :(

Logical And

```
// Solution 1
template<bool... Bs>
using is_all =
  is_same<boollist<Bs...>,
          boollist<(Bs || true)...>>;
```

Logical And

```
// Solution 2
template<bool... Bs>
using is_all =
  is_same<boollist<true, Bs...>,
          boollist<Bs..., true>>;
```

C++17

```
// C++17 fold expression
template<bool... Bs>
using is_all =
  bool_constant<(Bs && ...)>;
```

C++17

```
// C++17 fold expression
template<bool... Bs>
using is_all =
  bool_constant<(Bs && ...)>;
```

C++17

```
// C++14 variable template
// C++17 fold expression
template<int... Is>
constexpr bool is_all_even =
   ((Is % 2 == 0) && ...);
```

```
is_all_even<0, 2, 42> → true

is_all_even<4, 5, 6> → false
```

is_all_same

```
template<typename T, typename... Ts>
using is_all_same =
  is_all<is_same<T, Ts>::value...>;
```

```
is_all_same<int, int, int>::value
→ true
```

```
is_all_same<int, int, void>::value
→ false
```

is_all_same

```
template<typename T, typename... Ts>
using is_all_same =
  is_all<is_same<T, Ts>::value...>;
```

- Requires at least one type.
- No recursion, all the way down.
- Often you build new traits with `using` and smaller building blocks.

Logical Or

```
template<bool... Bs>
using is_any =
  bool_constant<
    !is_all<!Bs...>::value>;
```

enable_if_any/all

```
// C++11 / C++14
template<typename R, bool... Bs>
using enable_if_any =
  enable_if<is_any<Bs>::value..., R>;

template<typename R, bool... Bs>
using enable_if_all =
  enable_if<is_all<Bs>::value..., R>;
```

enable_if_any/all

```
// C++17 fold expression
template<typename R, bool... Bs>
using enable_if_any =
  enable_if<(Bs || ...), R>;

template<typename R, bool... Bs>
using enable_if_all =
  enable_if<(Bs && ...), R>;
```

Tuple

```
template<typename...>
struct tuple { … };
```

- Very useful as a real object.
- Can also be used as a `typelist`.
- When used as such, do not instantiate it!

Replace Type

```
template<
   typename T, typename U,
   typename... Ts>
using replace_t =
   tuple<
     conditional_t<
       is_same<Ts, T>::value, U, Ts
     >...
   >;
```

Replace Type

```
template<
  typename T, typename U,
  typename... Ts>
using replace_t =
  tuple<
    conditional_t<
      is_same<Ts, T>::value, U, Ts
    >...
  >;
```

Replace Type

```
template<
  typename T, typename U,
  typename... Ts>
using replace_t = …;
```

```
replace_t<int, double,
  void, int, long, int>
➙ tuple<void, double, long, double>
```

Tuple Size

```
template<typename>
struct tuple_size;

template<typename... Ts>
struct tuple_size<tuple<Ts...>>
  : integral_constant<size_t,
      sizeof...(Ts)>
{};
```

Tuple 🐈

```
template<typename... Ts>
constexpr tuple<CTypes...>
  tuple_cat(Ts&&... args) { … }
```

Tuple 🐈

```
template<typename... Ts>
constexpr tuple<CTypes...>
  tuple_cat(Ts&&... args) { … }
```

- Each element of `Ts` is itself a `tuple<Us...>`.
- `CTypes` is the concatenation of all `Us...` from all `Ts`.

Tuple 🐈

```
auto t = tuple_cat(
  make_tuple(1, true),
  make_tuple(1.0),
  make_tuple(1UL, nullptr));
```

```
decltype(t)
➙ tuple<int, bool,
    double,
    unsigned long, nullptr_t>
```

Tuple 🐈

```
template<typename... Ts>
using tuple_cat_t =
  decltype(
    tuple_cat(declval<Ts>()...)
  );
```

Tuple 🐈

```
template<typename... Ts>
using tuple_cat_t =
  decltype(
    tuple_cat(declval<Ts>()...)
  );
```

- The above does not always work…
- …but it is easy to fix. (see appendix A)

Remove Type

```
template<typename T, typename... Ts>
using remove_t =
  tuple_cat_t<
    conditional_t<
      is_same<Ts, T>::value,
      tuple<>,
      tuple<Ts>
    >...
  >;
```

Remove Type

```
template<typename T, typename... Ts>
using remove_t = …;
```

```
remove_t<int, void, int, long, int>
➙ tuple<void, long>

remove_t<void, void, int, void>
➙ tuple<int>
```

Indices

```
template<typename T, T...>
struct integer_sequence { … };
```

- Standardized in C++14.
- Extremely useful!

Indices

```
template<typename T, T...>
struct integer_sequence { … };



integer_sequence<long>

integer_sequence<unsigned, 0, 42, 1>

integer_sequence<int, -2, 5, -1, 9>
```

Indices

```
template<size_t... Ns>
using index_sequence =
  integer_sequence<size_t, Ns...>;
```

```
index_sequence<>

index_sequence<2, 5, 11, 3, 7, 13>

index_sequence<0, 1, 2, 3, 4, 5, 6>
```

Indices

```
template<typename T, T N>
using make_integer_sequence =
  integer_sequence<T,
    /* a sequence 0, …, N-1 */>;

template<size_t N>
using make_index_sequence =
  make_integer_sequence<size_t, N>;
```

Indices

```
make_index_sequence<3>
➞ index_sequence<0, 1, 2>

make_index_sequence<0>
➞ index_sequence<>

make_index_sequence<7>
➞ index_sequence<0, 1, 2, 3, 4, 5, 6>
```

Indices

```
template<typename... Ts>
using index_sequence_for =
  make_index_sequence<sizeof...(Ts)>;
```

```
index_sequence_for<int, void, int>
➞ index_sequence<0, 1, 2>
```

Replace Index

```
template<
   typename T,
   size_t N, typename U,
   typename = make_index_sequence<
     tuple_size<T>::value
   >
>
struct replace_n;
```

Replace Index

```
template<typename... Ts,
  size_t N, typename U,
  size_t... Ns>
struct replace_n<tuple<Ts...>,
  N, U,
  index_sequence<Ns...>>
{
  using type = tuple<
    conditional_t<Ns==N, U, Ts>...
  >;
};
```

Replace Index

```
template<typename... Ts,
  size_t N, typename U,
  size_t... Ns>
struct replace_n<tuple<Ts...>,
  N, U,
  index_sequence<Ns...>>
{
  using type = tuple<
    conditional_t<Ns==N, U, Ts>...
  >;
};
```

Replace Index

```
template<typename T,
   size_t N, typename U>
using replace_n_t =
   typename replace_n<T, N, U>::type;



using T1 = tuple<int, void, long>;

replace_n_t<T1, 1, double>
➙ tuple<int, double, long>
```

Replace Index

```
using T1 = tuple<int, double, long>;

using T2 = replace_n_t<T1, 1, void>;
using T3 = replace_n_t<T2, 0, T1>;
using T4 = replace_n_t<T3, 2, int>;
```

Replace Index

```
using T1 = tuple<int, double, long>;

using T2 = replace_n_t<T1, 1, void>;
using T3 = replace_n_t<T2, 0, T1>;
using T4 = replace_n_t<T3, 2, int>;
```

```
T4 ➝ tuple<
  tuple<int, double, long>, void, int
>
```

Benchmark

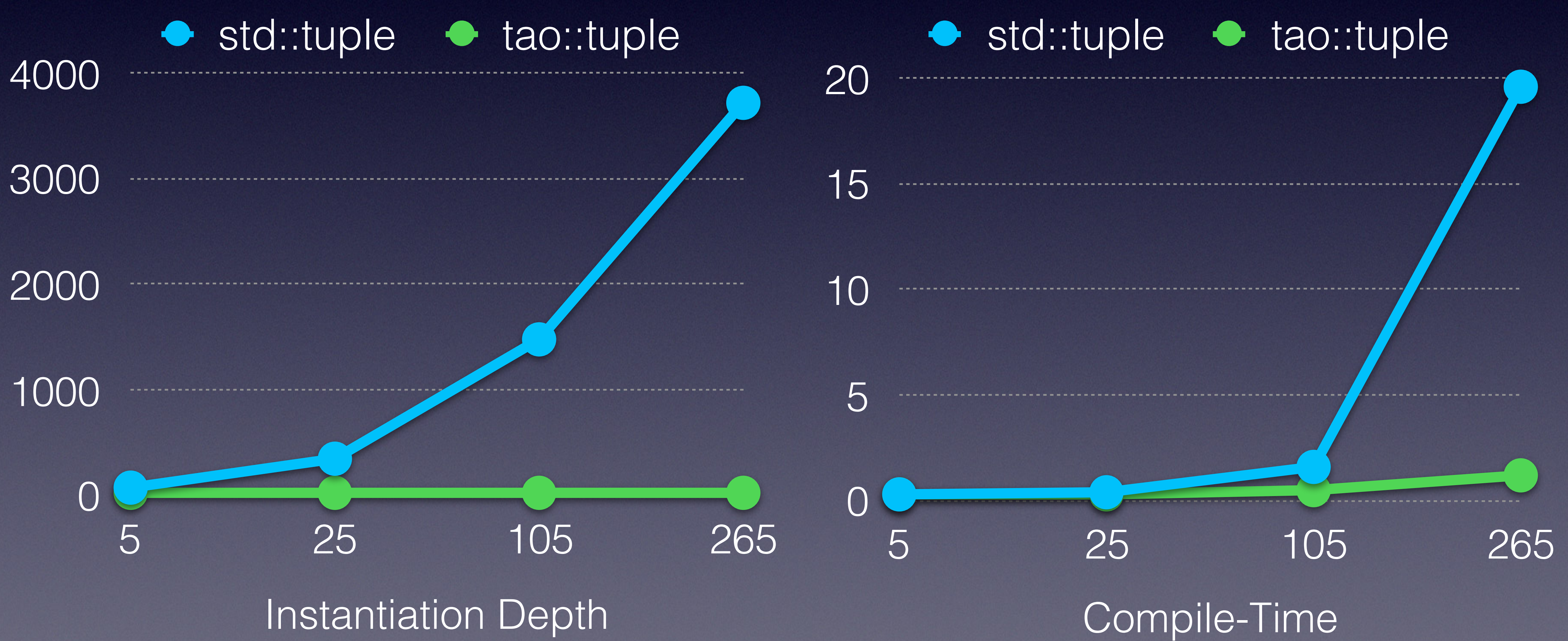
- Create a tuple, with `tuple_cat()`, 5 elements
- Join this tuple with itself and some other elements. Result tuple is 25 elements.
- Repeat, result tuple is 105 elements.
- Repeat, final result tuple is 265 elements.
- Measure instantiation depth and compile time.

Benchmark

System	Compiler	Tuple	Instantiation Depth	Compile Time
Linux	GCC 5	libstdc++ std::tuple	3719	19.6s
Linux	GCC 5	libstdc++ tao::tuple	26	1.2s
Mac OS X	Apple LLVM 7.0	libc++ std::tuple	514	>70.0s
Mac OS X	Apple LLVM 7.0	libc++ tao::tuple	15	1.7s

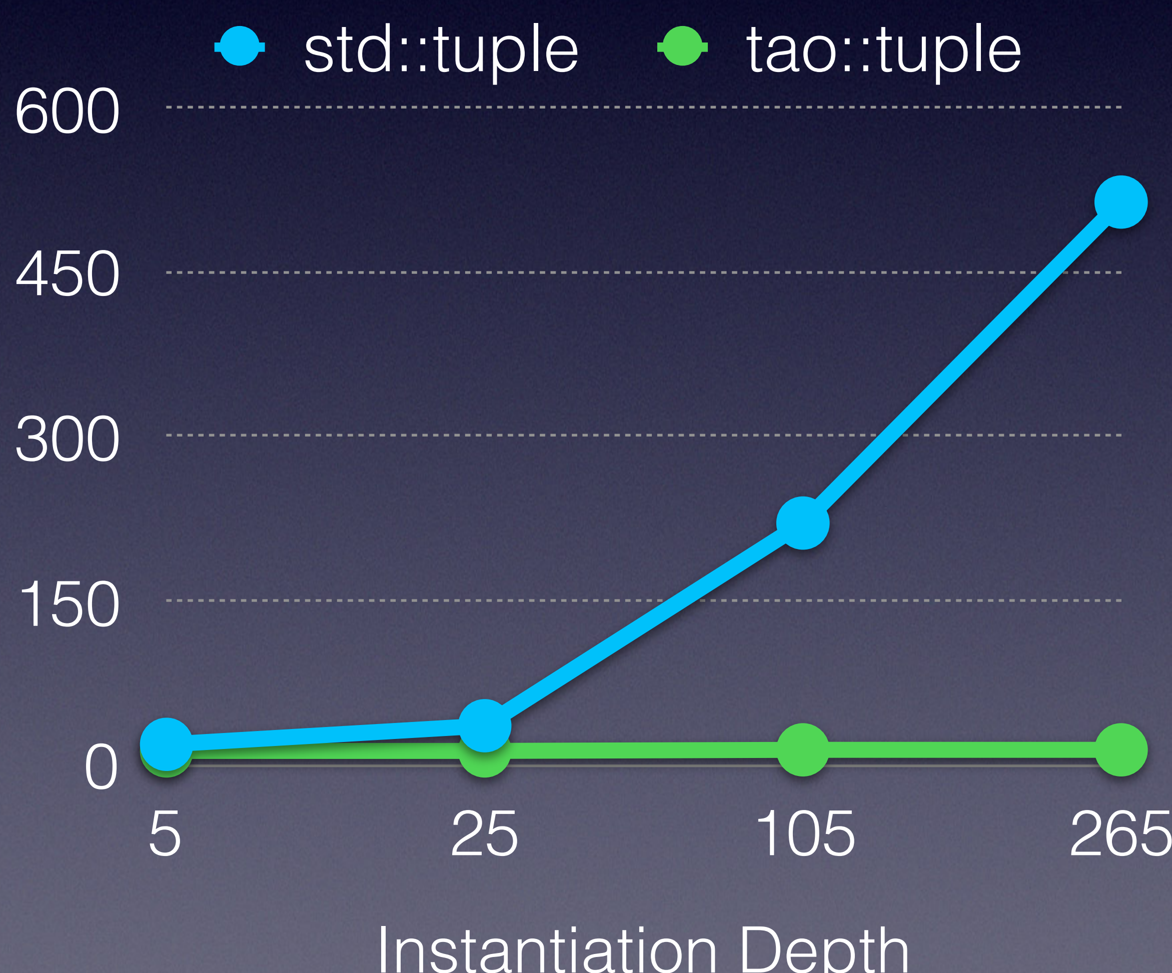
Benchmark

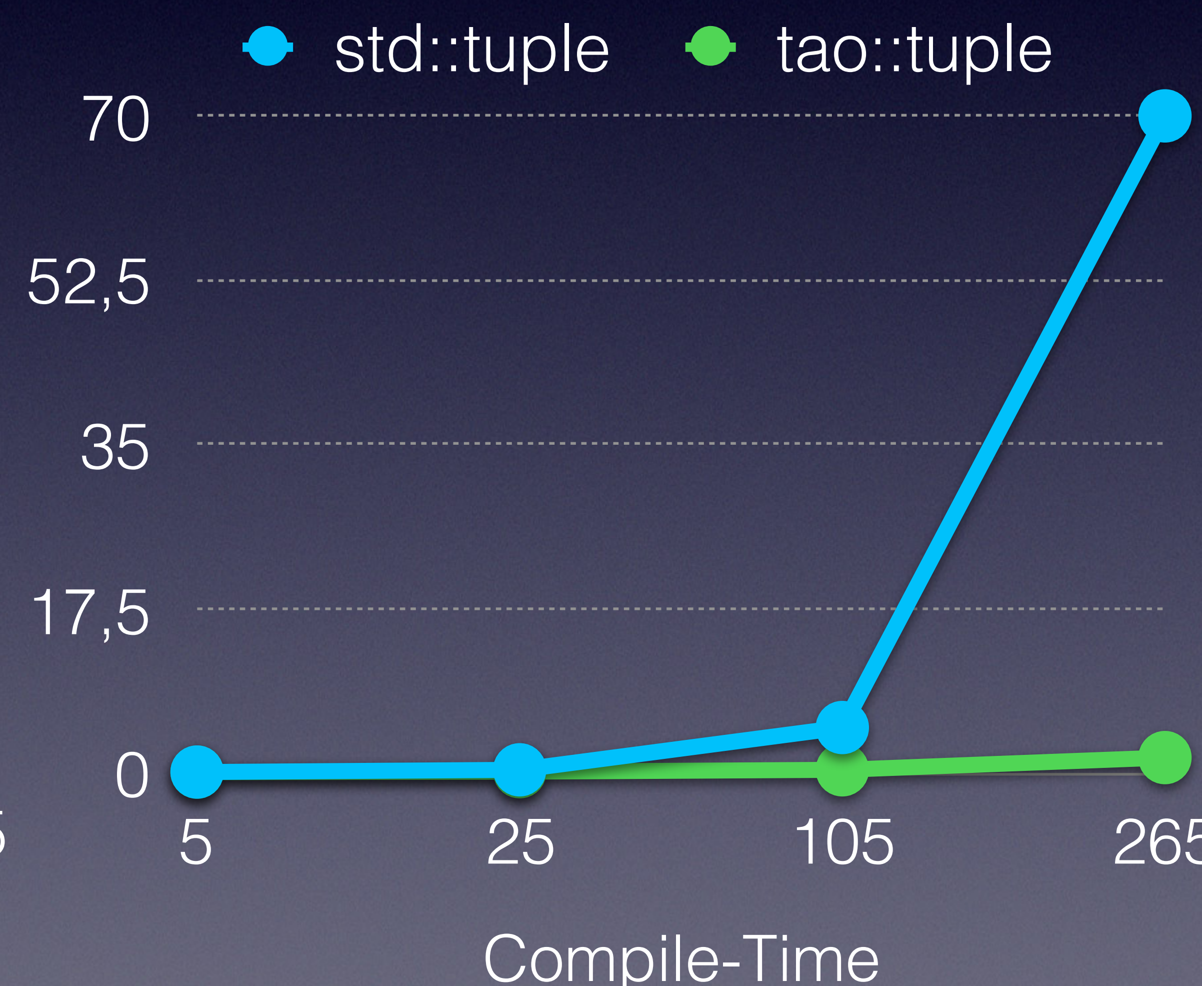
GCC 5 with libstdc++

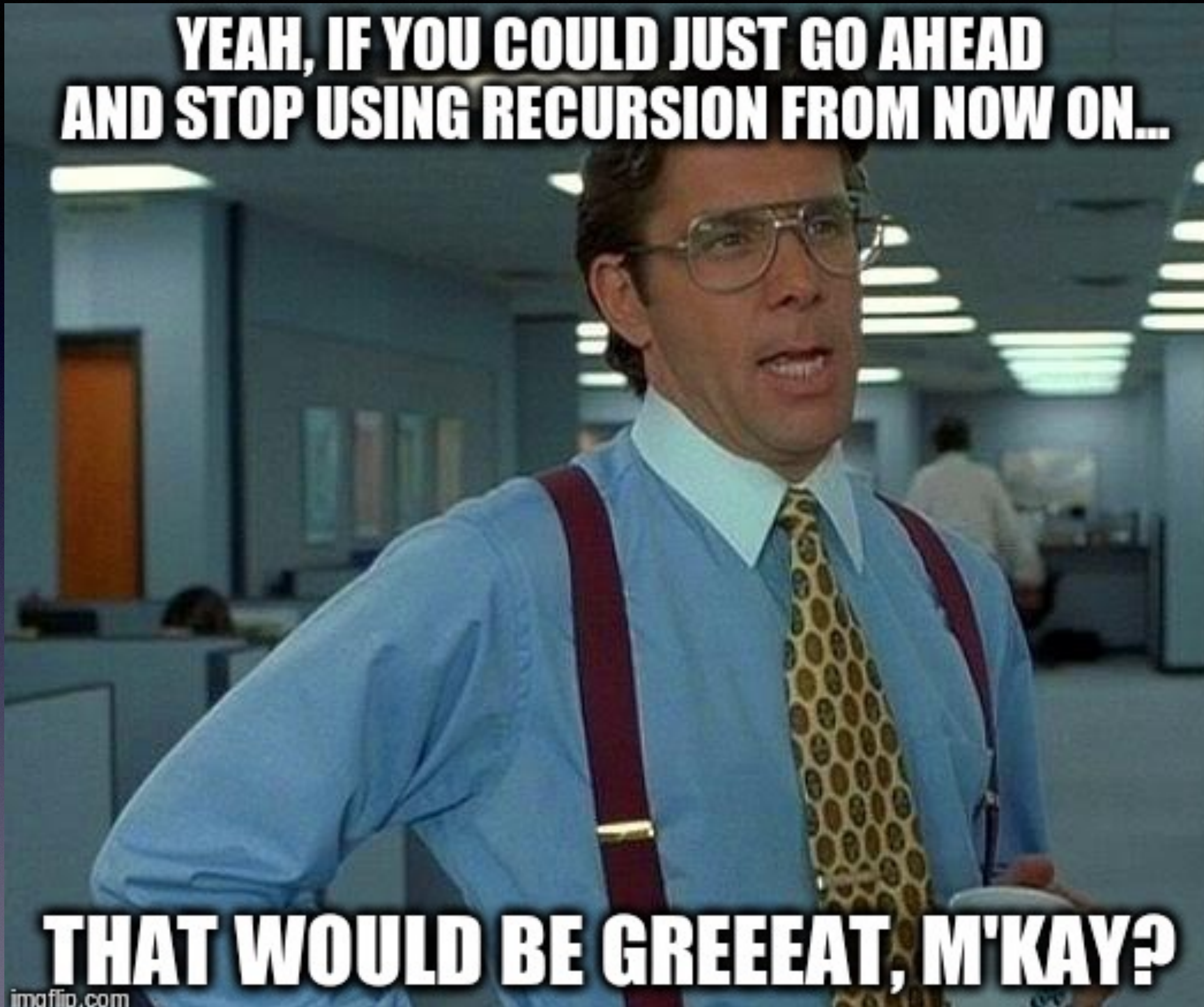


Benchmark

Apple LLVM 7.0 (~Clang 3.7) with libc++






YEAH, IF YOU COULD JUST GO AHEAD
AND STOP USING RECURSION FROM NOW ON...
THAT WOULD BE GREEEAT, M'KAY?
imgflip.com

Questions?

Thank you!

https://github.com/taocpp/

Appendix A

Fixing `tuple_cat_t`

Tuple 🐈 (fixed)

```
template<typename... Ts>
using tuple_cat_t =
  decltype(
    tuple_cat(declval<Ts>()...)
  );
```

- Does not work with **`void`**.
- Solution: Temporarily wrap the types.

Tuple 🐈 (fixed)

```
template<typename>
struct wrapper {};
```

Tuple 🐈 (fixed)

```
template<typename>
struct wrap;

template<typename... Ts>
struct wrap<tuple<Ts...>>
{
  using type = tuple<wrapper<Ts>...>;
};
```

Tuple 🐈 (fixed)

```
template<typename>
struct unwrap;

template<typename... Ts>
struct unwrap<tuple<wrapper<Ts>...>>
{
  using type = tuple<Ts...>;
};
```

Tuple 🐈 (fixed)

```
template<typename T>
using wrap_t =
  typename wrap<T>::type;

template<typename T>
using unwrap_t =
  typename unwrap<T>::type;
```

Tuple 🐈 (fixed)

```
template<typename... Ts>
using tuple_cat_t =
  unwrap_t<
    decltype(
      tuple_cat(
        declval<wrap_t<Ts>>()...
      )
    )
  >;
```

Appendix B

A better `make_integer_sequence`

Make Indices

```
template<typename T, T N>
using make_integer_sequence =
  integer_sequence<T,
    /* a sequence 0, …, N-1 */>;
```

- Naive implementation recurses for each step.
- `libstdc++` currently implements it that way.

Make Indices

```
template<typename T, T N>
using make_integer_sequence =
  integer_sequence<T,
    /* a sequence 0, …, N-1 */>;
```

- A better implementation has O(log N).
- Based on http://stackoverflow.com/a/13073076

Make Indices

```
template<typename, size_t, bool>
struct _double;
```

Make Indices

```
template<typename T, T... Ns,
  size_t N>
struct _double<
  integer_sequence<T, Ns...>,
  N, false>
{
  using type = integer_sequence<T,
    Ns..., (N + Ns)...>;
};
```

Make Indices

```
template<typename T, T... Ns,
  size_t N>
struct _double<
  integer_sequence<T, Ns...>,
  N, true>
{
  using type = integer_sequence<T,
    Ns..., (N + Ns)..., 2 * N>;
};
```

Make Indices

```
template<typename T, T N,
  typename = void>
struct _generate;

template<typename T, T N>
using _generate_t =
  typename _generate<T, N>::type;
```

Make Indices

```
template<typename T, T N>
struct _generate
  : _double<
      _generate_t<T, N / 2>,
      N / 2,
      N % 2 == 1>
    >
{};
```

Make Indices

```
template<typename T, T N>
struct _generate<T, N,
  enable_if_t<(N == 0)>
>
{
  using type =
    integer_sequence<T>;
};
```

Make Indices

```
template<typename T, T N>
struct _generate<T, N,
  enable_if_t<(N == 1)>
>
{
  using type =
    integer_sequence<T, 0>;
};
```

Make Indices

```
template<typename T, T N>
using make_integer_sequence =
  _generate_t<T, N>;
```

- Timing for `make_index_sequence<10000>`
- GCC 5: ~2.5s (~0.15s with a small change)
- Clang 3.6: ~0.15s

Make Indices

- libc++ has a very efficient library-based solution, better than the one I've shown.
- Best solution is a compiler intrinsic.
- Visual C++ will get one soon:

```
template<typename T, T N>
using make_integer_sequence =
  __make_integer_seq<T, N>;
```

Appendix C

More non-recursive goodies

Plus

```
template<typename, typename>
struct plus;

template<typename A, typename B>
using plus_t =
  typename plus<A, B>::type;
```

Plus

```
template<typename A, A... As,
         typename B, B... Bs>
struct plus<
  integer_sequence<A, As...>,
  integer_sequence<B, Bs...>>
{
  using type =
    integer_sequence<
      common_type_t<A, B>,
      (As + Bs)...>;
};
```

Minus

```
template<typename, typename>
struct minus;

template<typename A, typename B>
using minus_t =
  typename minus<A, B>::type;
```

Minus

```
template<typename A, A... As,
         typename B, B... Bs>
struct minus<
  integer_sequence<A, As...>,
  integer_sequence<B, Bs...>>
{
  using type =
    integer_sequence<
      common_type_t<A, B>,
      (As - Bs)...>;
};
```

Sum

```
template<typename T, T... Ns>
struct sum
  : integral_constant<T, (Ns + ...)>
{};
```

- Simple with C++17's fold expressions.
- But what about C++11/C++14?

Sum

```
// Helper
template<size_t, size_t N>
struct _chars
{
  char _dummy[N + 1];
};
```

Sum

```
// Helper
template<typename, size_t...>
struct _collector;

template<size_t... Is, size_t... Ns>
struct _collector<
  index_sequence<Is...>, Ns...
>
  : _chars<Is, Ns>...
{};
```

Sum

```
// Helper
template<size_t N, size_t... Ns>
using _sum =
  integral_constant<size_t,
    sizeof(
      _collector<
        make_index_sequence<N>,
        Ns...
      >
    ) - N
  >;
```

Sum

```
template<size_t... Ns>
struct sum
  : _sum<sizeof...(Ns) + 1, Ns..., 0>
{};
```

- Works with C++14 (C++11).
- This version is `size_t` only, can be extended for integer types.

Sum

```
template<typename T, T... Ns>
struct sum<
  integer_sequence<T, Ns...>
>
  : sum<T, Ns...>
{};
```

Partial Sum

```
template<size_t,
  typename S,
  typename =
    make_index_sequence<S::size()>>
struct _partial_sum;
```

Partial Sum

```
template<size_t I,
  typename T, T... Ns,
  size_t... Is>
struct _partial_sum<I,
  integer_sequence<T, Ns...>,
  index_sequence<Is...>
>
  : sum<T, ((Is < I) ? Ns : 0)...>
{};
```

Partial Sum

```
template<size_t I,
  typename T, T... Ns>
struct partial_sum
  : _partial_sum<I,
      integer_sequence<T, Ns...>>
{};
```

Partial Sum

```
template<size_t I,
  typename T, T... Ns>
struct partial_sum<I,
  integer_sequence<T, Ns...>
>
  : _partial_sum<I,
      integer_sequence<T, Ns...>>
{};
```

Exclusive Scan

```
template<typename S,
  typename =
    make_index_sequence<S::size()>>
struct _exclusive_scan;
```

Exclusive Scan

```
template<typename S, size_t... Is>
struct _exclusive_scan<S,
  index_sequence<Is...>>
{
  using type =
    integer_sequence<
      typename S::value_type,
      partial_sum<Is, S>::value...>;
};
```

Exclusive Scan

```
template<typename T, T... Ns>
struct exclusive_scan
  : _exclusive_scan<
      integer_sequence<T, Ns...>>
{};
```

Exclusive Scan

```
template<typename T, T... Ns>
struct exclusive_scan<
  integer_sequence<T, Ns...>
>
  : exclusive_scan<T, Ns...>
{};
```

Exclusive Scan

```
template<typename T, T... Ns>
using exclusive_scan_t =
  typename
    exclusive_scan<T, Ns...>::type;
```

Inclusive Scan

```
template<typename T, T... Ns>
struct inclusive_scan
  : plus<
      exclusive_scan_t<T, Ns...>,
      integer_sequence<T, Ns...>
    >
{};
```

Inclusive Scan

```
template<typename T, T... Ns>
struct inclusive_scan<
  integer_sequence<T, Ns...>
>
  : inclusive_scan<T, Ns...>
{};
```

Inclusive Scan

```
template<typename T, T... Ns>
using inclusive_scan_t =
  typename
    inclusive_scan<T, Ns...>::type;
```

Select

```
template<typename T, T... Ns>
struct _select
{
  static constexpr T arr[] = {Ns...};
};
```

Select

```
template<size_t I,
  typename T, T... Ns>
struct select
  : integral_constant<T,
      _select<T, Ns...>::arr[I]>
{};
```

Select

```
template<size_t I,
  typename T, T... Ns>
struct select<
  I, integer_sequence<T, Ns...>
>
  : select<I, T, Ns...>
{};
```

Map

```
template<typename, typename>
struct map;
```

Map

```
template<size_t... Ns, typename M>
struct map<index_sequence<Ns...>, M>
{
  using type =
    integer_sequence<
      typename M::value_type,
      integer_select<Ns, M>::value...
    >;
};
```

Map

```
template<typename S, typename M>
using map_t =
  typename map<S, M>::type;
```

Appendix D

`type_select`

Type By Index

```
template<size_t>
struct any
{
  any(...);
};
```

The above ellipsis does not denote a variadic template, it is a C-style variadic parameter.

Type By Index

```
template<typename>
struct get_nth;

template<size_t... Is>
struct get_nth<index_sequence<Is...>>
{
  template<typename T>
    static T deduce(
      any<Is & 0>..., T*, ...);
};
```

Type By Index

```
template<typename>
struct wrapper;

template<typename>
struct unwrap;

template<typename T>
struct unwrap<wrapper<T>>
{
  using type = T;
};
```

Type By Index

```
template<size_t I, typename... Ts>
using type_select =
  unwrap<
    decltype(
      get_nth<
        make_index_sequence<I>
      >::deduce(
        declval<wrapper<Ts>*>()...
      )
    )
  >;
```

Type By Index

```
template<size_t I, typename... Ts>
using type_select_t =
  typename
    type_select<I, Ts...>::type;
```

Appendix E

A better `tuple_cat`

Tuple 🐈

```
template<size_t M, size_t... Ns>
struct count_less_or_equal
  : sum<size_t, ((Ns <= M)?1:0)...>
{};
```

The above should be implemented as a type-alias, but that triggers a bug in GCC 4.x.

Tuple 🐈

```
template<typename, typename>
struct expand;

template<typename I, typename S>
using expand_t =
  typename expand<I, S>::type;
```

Tuple 🐈

```
template<size_t... Is, size_t... Ns>
struct expand<
  index_sequence<Is...>,
  index_sequence<Ns...>>
{
  using type =
    index_sequence<
      count_less_or_equal<Is, Ns...>
        ::value...>;
};
```

Tuple 🐈

```
template<size_t, typename>
struct tuple_select;

template<size_t I, typename... Ts>
struct tuple_select<I, tuple<Ts...>>
  : type_select<I, Ts...>
{};

template<size_t I, typename T>
using tuple_select_t =
  typename tuple_select<I, T>::type;
```

Tuple 🐈

```
template<typename...>
struct tuple_cat_result;

template<typename... Ts>
using tuple_cat_result_t =
  typename
    tuple_cat_result<Ts...>::type;
```

Tuple 🐈

```
template<
  size_t... Os, size_t... Is,
  typename... Ts>
struct tuple_cat_result<
  index_sequence<Os...>,
  index_sequence<Is...>, Ts...>
{
  using type = tuple<
    tuple_select_t<Is,
      type_select_t<Os, Ts...>>...>;
};
```

Tuple 🐈

```cpp
template<typename... Ts>
struct tuple_cat_h
{
  using S =
    index_sequence<
      tuple_size<Ts>::value...>;
  // …
};
```

Tuple 🐈

```
template<typename... Ts>
struct tuple_cat_h
{
  // …
  using R =
    make_index_sequence<
      sum<S>::value>;
  // …
};
```

Tuple 🐈

```
template<typename... Ts>
struct tuple_cat_h
{
  // …
  using O =
    expand_t<R, inclusive_scan_t<S>>;
  // …
};
```

Tuple 🐈

```
template<typename... Ts>
struct tuple_cat_h
{
  // …
  using I =
    minus_t<R,
      map_t<O, exclusive_scan_t<S>>>;
  // …
};
```

Tuple 🐈

```
template<typename... Ts>
struct tuple_cat_h
{
  // …
  using type =
    tuple_cat_result_t<0, I, Ts...>;
};
```

Tuple 🐈

```
template<typename R,
   size_t... Os, size_t... Is,
   typename Tuples>
R _tuple_cat(
   index_sequence<Os...>,
   index_sequence<Is...>,
   Tuples t)
{
   return R(get<Is>(get<Os>(t))...);
}
```

Tuple 🐈

```
template<typename... Ts,
   typename H = tuple_cat_h<Ts...>,
   typename R = typename H::type>
R tuple_cat(Ts&&... ts)
{
   return _tuple_cat<R>(
      typename H::O(),
      typename H::I(),
      forward_as_tuple(
         forward<Ts>(ts)...));
}
```