Embracing (and Also Destroying) Variant Types *Safely*

C++ Russia 2021

Andrei Alexandrescu, Ph.D.
andrei@erdani.com

2021-11-17

Coming soon to a bookstore near you!

Talks in this series:

- Embracing User Defined Literals *Safely* for Types that Behave as though Built-in
  - *Pablo Halpern, Tuesday 9am*
- Embracing (and also destroying) Variant Types *Safely*
  - *Andrei Alexandrescu, Thursday 9am*
- Embracing PODs *Safely* Until They Die
  - *Alisdair Meredith & Nina Ranns, Thursday 10:30am*
- *Embracing `noexcept` Operators and Specifiers Safely*
  - *John Lakos, Thursday 3:15pm*

You are here!

bloomberg.com/careers

I work with Bloomberg. I like them! They're hiring!

jobs.symmetryinvestments.dev

I work with Symmetry Investments. I like them! They're hiring!

Plan

- Prelude: understanding variadics
- Variant types: do we care?
- `std::variant`
- Destroying variants, or the Converse Factory Method pattern

Motivation of Variadics

- Class/function templates w/ arbitrary number of arguments
  - No limitations, simple boilerplate
  - Algebraic types: `tuple`, `variant`
- Efficiency: consolidates calculations, allocations, locking
  - `concat`, `print`
- Calling ctors with unknown arity
  - `make_unique`, `make_shared`, `emplace*`

Variadics Open New Opportunities

- `all_of`, `any_of`, `mismatch`, `equal`, `merge`, `set_union`, `set_intersection` could and should be variadic
- NOT equivalent to chaining calls!
- `static for` needed for simple implementation

Understanding Variadics

- Difficult rules, mostly by enumeration and fiat
- Interactions with the rest of C++
- Must memorize: *what* expansions are allowed
- Must memorize: *where* expansion is allowed

- Deduction rules: the most difficult by far

Variadics: #1 Thing to Understand

- Packs are a distinct *kind*
- All types belong to a kind
  - . . . or 14, depending how you count
- Template names belong to another kind
- There is a one-of-a-kind construct

- Adding new kinds is almost unprecedented
- Packs are unlike everything else in C++

Variadics: #2 Thing to Understand

- It's all about '...'
- '...' to the left of a name means "introduce"
  - `template <typename... Ts> class C;`
- '...' to the right of a pattern means "expand"
  - Pattern: expression or type, e.g. `C<Ts>...`
  - One or more packs
  - Multiple packs expand in lockstep

- That's literally it
- BTW this is a huge bummer!

(

Please add this ONE LINE to std C++!

```
template <typename... Ts> struct type_sequence {};
```

Please add this ONE LINE to std C++!

```
template <typename... Ts> struct type_sequence {};
```

- Allows manipulation of packs as types
  - What does `std::tr2::direct_bases` yield?
- btw `integer_sequence` really is `value_sequence`

```
using ehm = integer_sequence<double, 1.5, 3.14>;
```

Actually a couple more lines

```
template <typename... Ts> struct head;

template <typename T, typename... Ts>
struct head<type_sequence<T, Ts...>> {
   using type = T;
};

template <typename T>
using head_t = typename head<T>::type;
```

Actually a couple more lines

```
template <typename... Ts> struct tail;

template <typename T, typename... Ts>
struct tail<type_sequence<T, Ts...>> {
   using type = type_sequence<Ts...>; // AHA!
};

template <typename T>
using tail_t = typename tail<T>::type;
```

Actually a couple more lines

```
template <typename T, typename List> struct cons;

template <typename T, typename... Ts>
struct cons<T, type_sequence<Ts...>> {
   using type = type_sequence<T, Ts...>;
};

template <typename T, typename List>
using cons_t = typename cons<T, List>::type;
```

)

The Rule of Greedy Matching

Once a template parameter pack starts matching one explicitly specified template argument, it also matches everything after it.

Greedy Matching

- A parameter *after* a template parameter pack can *never* be explicitly specified
  - i.e., it is *always* deduced

```
template <typename T, typename... Ts, typename U>
int f(T, U, const Ts&...);
```

- `U` is *always* deduced

```
f(1, 2);   // Ts=<>
f<int, int>(1, 2, 3);   // Ts=<int>
f<int, int, double>(1, 2, 3, "4"); // Ts=<int, double>
```

Quiz: What Does This Do?

```
template <typename... Ts, typename T = int>
T puzzle(Ts... values);
```

The Rule of Fair Matching

A function parameter pack that's not at the end of a function's parameter list can never have its corresponding type pack deduced.

Fair Matching

- Not competing with Greedy Matching
  - They apply to different packs

```
template <typename... Ts, typename T>
int f(Ts... values, T value);
```

- `Ts` is *always* explicit

```
f(1);   // Ts=<>
f<>(1);   // Ts=<>
f<int, double>(1, "2", 3); // Ts=<int, double>
f<int, int>(1, 2); // Error
```

Quiz

```
template <typename... Ts, typename... Us>
int f(Ts... ts, Us... us);
```

- `Us` is *always* deduced (greedy matching)
- `Ts` is *always* explicit (fair matching)

```
f(1);
f(1, "2");
f<int, char>(1, '2');
f<int, char>(1, '2', "three");
```

The Corner Case from Hell

```
template <typename... Ts, typename... Us, class T>
int f(Ts..., Us..., T);

int a = f(42);
int b = f(1, 2.5);
int c = f<int, double>(1, 2.0, "three");
```

So. . .

Variant types: why do we care?

Variants types recap

- aka discriminated unions, sum types
- Can hold one value of a restricted set of types
  - Not to be confused with `std::any`

- `variant<nullptr_t, string, double, Obj*, bool, void*>`

Turns out, they're everywhere!

- Databases: columns
- Spreadsheets: cell types
- JSON & other data formats/protocols: values
- Interpreters: values
- DSLs: values
- Compilers: expressions etc
- Herb's keynote! (seven times)

- Good implementation crucial to performance
- Good *destructor* implementation crucial to performance

One Destructor Inlined

Experiment-relative percents (I

pagestats

Page	Count	CPU Instructions	CPU Time	Memcache KBytes	Memcache Keys	Memcache Ops
WWW Endpoints	416/420	1,561 -0.3% ±122 ±0.2%	838 +0.4% ±58 ±0.6%	177 +0% ±67 ±0%	674 +0% ±84 ±0%	102 +0% ±5.8 ±0%
API Home	141/150	914 -0.1% ±70 ±0.2%	432 +0.5% ±35 ±0.7%	60 -0% ±12 ±0%	130 +0% ±18 ±0%	43 +0% ±2.3 ±0.1%
API Timeline	145/150	470 -0.1% ±51 ±0.3%	240 +0.5% ±24 ±0.8%	22 -0% ±5.5 ±0%	70 +0% ±14 ±0%	26 +0% ±2.5 ±0.1%
API Notifications	99/100	175 -0.1% ±42 ±0.4%	100 +0.7% ±24 ±1.3%	15 +0% ±6.3 ±0%	76 +0% ±22 ±0%	14 +0% ±1.9 ±0.1%
/home.php	99/100	2,103 -0.3% ±162 ±0.2%	1,131 +0.4% ±80 ±0.5%	203 +0% ±76 ±0%	832 -0% ±104 ±0%	127 -0% ±8.7 ±0%
/profile_book.php	198/200	1,529 -0.3% ±57 ±0.1%	835 +0.6% ±27 ±0.4%	217 0% ±69 ±0%	800 0% ±68 ±0%	121 0% ±2.5 ±0%
/ajax/pagelet/generic.php:PhotoViewerPagelet	30/30	580 -0.6% ±116 ±0.6%	308 +0.4% ±53 ±0.9%	19 0% ±13 ±0%	53 0% ±17 ±0%	25 0% ±3.8 ±0%
WebPermalinkController	29/30	1,324 -0.4% ±221 ±0.8%	646 +0.4% ±94 ±1.1%	251 +0.1% ±142 ±0.2%	820 +0% ±202 ±0.4%	83 +0% ±9.3 ±0.1%
/widgets/like.php	60/60	72 -0.1% ±3.6 ±0.6%	41 +1.1% ±2.1 ±1.4%	2.8 0% ±0.54 ±0%	21 0% ±3.8 ±0%	8.4 0% ±0.30 ±0%
/ajax/chat/buddy_list.php	50/50	7.9 -0.3% ±0.065 ±0.2%	4.5 +0.5% ±0.092 ±1.4%	0.012 0% ±0.014 ±0%	0.064 0% ±0.078 ±0%	0.064 0% ±0.078 ±0%
/ajax/typeahead/search.php:search	48/50	129 -0.4% ±14 ±0.5%	71 +0.2% ±10 ±1.2%	12 0% ±1.7 ±0%	35 0% ±10 ±0%	9.8 0% ±0.83 ±0%
TypeaheadFacebarQueryController	50/50	144 -0.3% ±14 ±0.8%	78 +0.2% ±8.7 ±1.4%	18 0% ±2.8 ±0%	28 0% ±1.2 ±0%	7.8 0% ±0.59 ±0%
/wap/home.php:basic	50/50	830 -0.1% ±80 ±0.2%	422 +0.4% ±30 ±0.7%	68 -0% ±14 ±0%	1,020 0% ±263 ±0%	60 0% ±2.1 ±0%
/wap/home.php:touch	48/50	1,250 -0.1% ±95 ±0.3%	696 +0.3% ±51 ±0.9%	58 +0% ±11 ±0.1%	282 -0% ±52 ±0.1%	80 +0% ±5.7 ±0.1%
/wap/profile_timeline.php	32/40	699 -0.2% ±112 ±0.4%	348 +0.7% ±58 ±1.1%	28 +0% ±5.0 ±0.1%	488 +0% ±204 ±0%	50 +0% ±4.6 ±0.1%
/wap/profile_tribe.php	36/40	397 -0.2% ±98 ±0.5%	225 +0.9% ±49 ±1.1%	33 +0% ±9.2 ±0%	324 +0% ±113 ±0%	28 +0% ±3.2 ±0%

Variant Layout

```
template <typename... Ts> class variant;

template <typename T, typename... Ts>
class variant<T, Ts...> {
   enum : size_t {
      size = std::max({sizeof(T), sizeof(Ts)...})
   };
   alignas(T) alignas(Ts...) char d_data[size];
   unsigned d_active;
   // ...
};
```

Variant Implementation

- Construction, assignment, `emplace`, `swap`, comparison: the usual suspects
- Query: `index`, `valueless_by_exception`
  - Nonmembers: `holds_alternative`, `visit`, `get`, `get_if`

- Most are $O(1)$ (at runtime)
- `visit` and `~variant` are (naïvely) $O(n)$ in the number of types held

The Easy Part

```
template <size_t i, typename... Ts>
constexpr auto& get(const variant<Ts...>& v) {
   if (i != v.d_active)
      throw bad_variant_access("You're fired.");
   using T = variant_alternative_t<i, Ts...>;
   return *static_cast<const T*>(&v.d_data);
}
```

The Easy Part

```
template <unsigned N, typename T, typename... Ts>
struct variant_alternative
   : variant_alternative<N - 1, Ts...> {
};

template <typename T, typename... Ts>
struct variant_alternative<0, T, Ts...> {
   using type = T;
};
```

The Hard Part

- `visit`, `~variant`: perform a search
- `visit`: full-bore multiple dispatch!
- Destruction is the Converse Factory Method pattern
  - You know what you're making, but not what you have
  - Typical for artifacts using type erasure

- Here we discuss `variant<Ts...>::~variant()`

Mission: Destruction

```
template <unsigned>
void destroyLinear(unsigned, void*) { }

template <unsigned i, typename T, typename... Ts>
void destroyLinear(unsigned n, void* p) {
   if (n == i) static_cast<T*>(p)->~T();
   else destroyLinear<i + 1, Ts...>(n, p);
}
```

Mission: Destruction

```
template <typename... Ts>
variant<Ts...>::~variant() {
   if (d_active < sizeof...(Ts))
      destroyLinear<0, Ts...>(d_active, &d_data);
}
```

If Linear is Good, Log is Better

- Parameter packs can be walked only linearly
- Numbers can be halved at compilation/runtime

- Linear search at compilation, binary search at runtime

Mission: Destruction

```
template <unsigned lo, unsigned hi, typename... Ts>
void destroyLog(unsigned n, void* p) {
   static_assert(lo < hi);   // static precondition
   assert(n >= lo && n < hi);   // precondition
   constexpr size_t m = lo + (hi - lo) / 2;
   if (n < m) {
      destroyLog<lo, lo==m?hi:m, Ts...>(n, p);
   } else if (n > m) {
      destroyLog<m+1==hi?lo:(m+1), hi, Ts...>(n, p);
   } else { // (n == m)
      using Tn = variant_alternative_t<m, Ts...>;
      static_cast<Tn*>(p)->~Tn();
   }
}
```

Mission: Destruction

```
template <typename... Ts>
variant<Ts...>::~variant() {
   if (d_active < sizeof...(Ts))
      destroyLog<0, sizeof...(Ts), Ts...>(
         d_active, &d_data);
}
```

If Log is Better, is Constant Best?

- Given an index:
- How can we call in $O(1)$ a function depending on that index?

Mission: Destruction

```
template <typename T>
static void destroyElement(void* p) {
   static_cast<T*>(p)->~T();
}

template <typename... Ts>
void destroyCtTime(unsigned n, void* p) {
   static const auto dt[] =
      { &destroyElement<Ts>... };
   dt[n](p);
}
```

Analysis of `destroyCtTime`

- Constant time
- Small
- Simple
- Easy to understand
- No massive template instantiations
- But: indirect calls (gcc, clang) very slow
- But: wasteful for trivial dtors (often)

- True solution: *algorithm selection*

Qualitatively New Challenge

- New era of C++: mixed-mode algorithms
- Some compile-time, some run-time
- Created by increase in introspection
- Linear pack iteration $\Rightarrow O(n^2)$ instantiations

- This is just the beginning

Breaking news!

Hat tip to Eduardo Madrid

- Snap (of Snapchat fame) optimized their variant type heavily
- ‘switch‘ with only two branches expands in ’“Russian Doll” manner
- clang converts it to a jump table
- gcc doesn’t

Hat tip to Eduardo Madrid

```
template<typename R, typename Visitor, typename V,
   int Current, typename Head, typename... Tail>
struct Visit_impl {
   static R execute(Visitor &&visitor, V &&v, int typeSwitch) {
      switch(typeSwitch) {
      case Current: return visitor(*v.template as<Head>());
      default:
         return
            Visit_impl<R, Visitor, V, Current + 1, Tail...
            >::execute(
               std::forward<Visitor>(visitor),
               std::forward<V>(v),
               typeSwitch
            );
      }
   }
};
```

Hat tip to Andrey Petrushenko

```
template <typename... Ts, int... Is>
constexpr void destroy_variant(int idx, void* ptr,
      std::integer_sequence<int, Is...>) {
   (void)((idx == Is &&
      (static_cast<Ts*>(ptr)->~Ts(), true)) || ...);
}
template <typename... Ts>
constexpr void destroy_variant(int idx, void* ptr) {
   destroy_variant<Ts...>(idx, ptr,
      std::make_integer_sequence<int,
         sizeof...(Ts)>{});
}
```

Summary

- Understand type deduction in variadic functions
  - Rule of Greedy Matching
  - Rule of Fair Matching
- Variant destructors are difficult
- Get ready for mixed-mode algorithms

- Add that line to the standard!

Thank You!

Destroy the speaker
safely.

From The Press Tomorrow

“Though the talk was purportedly about variants, the speaker used the opportunity to shamelessly push for three useless list primitives. . . ” ♦ “Looks like larger parts of the C++ compiler get moved into the standard library.” ♦ “. . . yet we still don’t understand how ‘safely’ fits here. . . ” ♦ “Legolas has been forever ruined for me.” ♦ “The book is available for preorder on Amazon for $79.99. Talk about inflation.” ♦ “After `static if` now `static for`? When will he stop?. . . ”