Variadic Templates

Andrei Alexandrescu
Research Scientist
Facebook

Twitter stream highlight

"Don't think I'll be able to stay up to see whatever Cthuloid-template-horror Andrei Alexandrescu has in store. #GoingNative"

This talk

- Motivation and fundamentals
- True variadic functions
- `std::tuple`

Motivation and fundamentals

Motivation

- Define typesafe variadic functions
  - C99 macros safer than C++03 variadic functions?!
  - Forwarding with before/after hooks
- Define algebraic types without contortions
  - Sum types (`variant`)
  - Product types (`tuple`)
- Specify settings and parameters in policy-based designs

Fundamentals

```
template <typename... Ts>
class C {
    : : :
};

template <typename... Ts>
void fun(const Ts&... vs) {
    : : :
}
```

A New Kind: *Parameter Packs*

- `Ts` is not a type; `vs` is not a value!

```
typedef Ts MyList; // error!
Ts var; // error!
auto copy = vs; // error!
```

- `Ts` is an alias for a list of types
- `vs` is an alias for a list of values
- Either list may be potentially empty
- Both obey only specific actions

Using Parameter Packs

- Apply `sizeof...` to it

```
size_t items = sizeof...(Ts); // or vs
```

- Expand back

```
template <typename... Ts>
void fun(Ts&&... vs) {
  gun(3.14, std::forward<Ts>(vs)..., 6.28);
}
```

- That’s about it!

Expansion rules

Use	Expansion
`Ts...`	`T1,...,Tn`
`Ts&&...`	`T1&&,...,Tn&&`
`x<Ts, Y>::z...`	`x<T1, Y>::z,...,x<Tn, Y>::z`
`x<Ts&, Us>...`	`x<T1&, U1>,...,x<Tn&, Un>`
`func(5, vs)...`	`func(5, v1),...,func(5, vn)`

- (Please note: ellipses on the right are in a different font)

Expansion loci (1/2)

- Initializer lists

```
any a[] = { vs... };
```

- Base specifiers

```
template <typename... Ts>
struct C : Ts... {};
template <typename... Ts>
struct D : Box<Ts>... { : : : };
```

- Member initializer lists

```
// Inside struct D
template <typename... Us>
D(Us... vs) : Box<Ts>(vs)... {}
```

Expansion loci (2/2)

- Template argument lists

```
std::map<Ts...> m;
```

- Exception specifications
  - On second thought, scratch that
- Attribute lists

```
struct [[ Ts... ]] IAmFromTheFuture {};
```

- Capture lists

```
template <class... Ts> void fun(Ts... vs) {
   auto g = [&vs...] { return gun(vs...); }
   g();
}
```

Multiple expansions

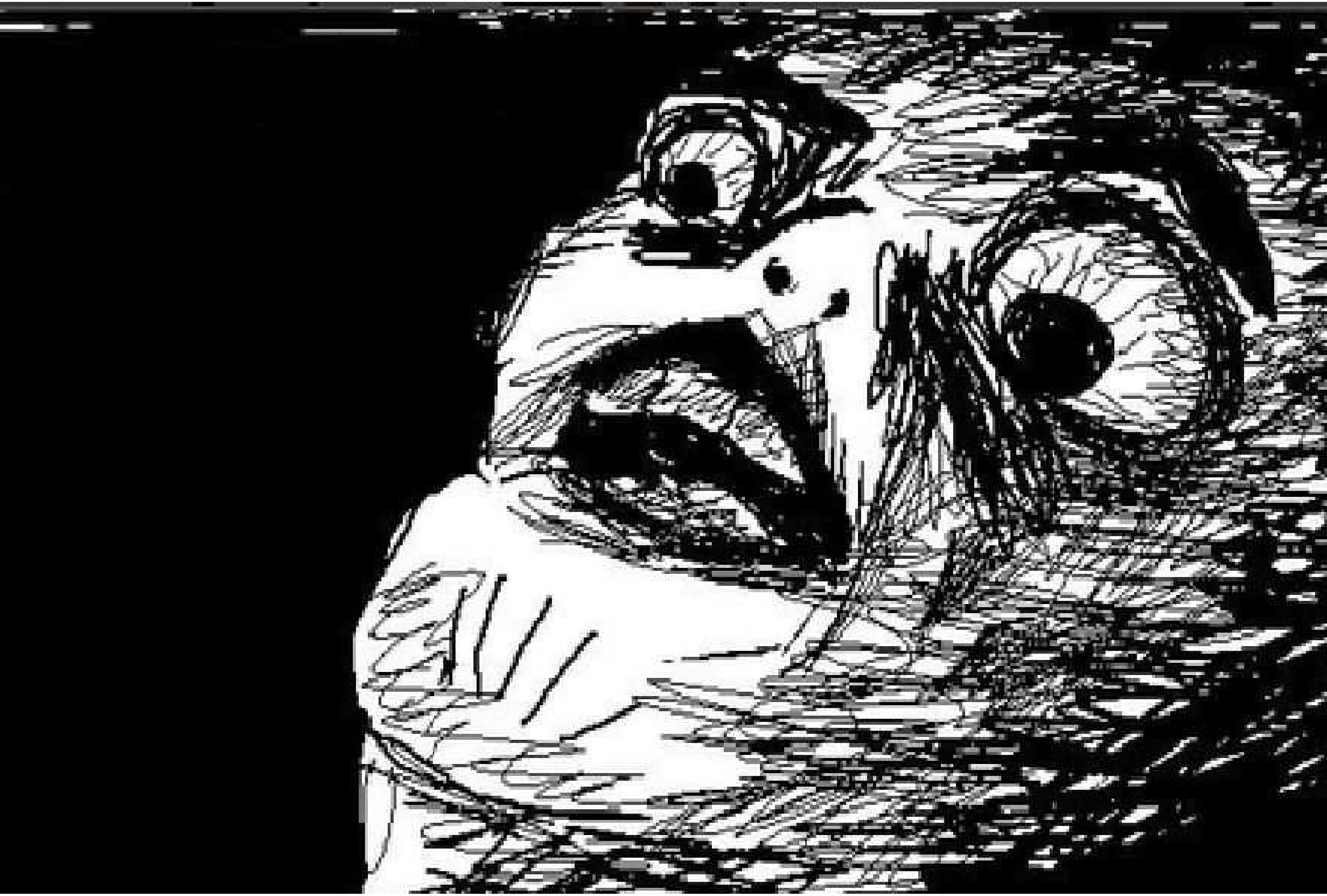
- Expansion proceeds outwards
- These are different expansions!

```
template <class... Ts> void fun(Ts... vs) {
   gun(A<Ts...>::hun(vs)...);
   gun(A<Ts...>::hun(vs...));
   gun(A<Ts>::hun(vs)...);
}
```

Per popular demand: VVTTs

```
template <
    typename T,
    template <
        template<class...> class... Policies
    >
>
class ICantBelieveItsNotButter;
```

- Yup, this works.



How to use variadics?

- Pattern matching!

```
template <class T1, class T2>
bool isOneOf(T1&& a, T2&& b) {
   return a == b;
}
template <class T1, class T2, class... Ts>
bool isOneOf(T1&& a, T2&& b, Ts&&... vs) {
   return a == b || isOneOf(a, vs...);
}
assert(isOneOf(1, 2, 3.5, 4, 1, 2));
```

True Variadic Functions

Typesafe `printf`

- Stock `printf`:
  - Fast
  - Thread-safe
  - Convenient
  - Ubiquitously known
  - Utterly unsafe
- Conventionally: reimplement from first principles

- Here: Add verification and adaptation code

Step 1: Add adaptation routines

```
template <class T>
typename enable_if<is_integral<T>::value, long>::type
normalizeArg(T arg) { return arg; }

template <class T> typename
enable_if<is_floating_point<T>::value, double>::type
normalizeArg(T arg) { return arg; }

template <class T>
typename enable_if<is_pointer<T>::value, T>::type
normalizeArg(T arg) { return arg; }

const char* normalizeArg(const string& arg) {
   return arg.c_str();
}
```

Preliminary tests

```
// Not really safe yet
template <typename... Ts>
int safe_printf(const char * f,
      const Ts&... ts) {
   return printf(f, normalizeArg(ts)...);
}
```

Step 2: Define test for arg-less call

```
void check_printf(const char * f) {
    for (; *f; ++f) {
        if (*f != '%' || *++f == '%') continue;
        throw Exc("Bad format");
    }
}
```

Step 3: Define recursive test

```
template <class T, typename... Ts>
void check_printf(const char * f, const T& t,
      const Ts&... ts) {
   for (; *f; ++f) {
      if (*f != '%' || *++f == '%') continue;
      switch (*f) {
      default: throw Exc("Invalid format char: %", *f);
      case 'f': case 'g':
         ENFORCE(is_floating_point<T>::value);
         break;
      case 's':  ...
      }
      return check_printf(++f, ts...); // AHA!!!
   }
   throw Exc("Too few format specifiers.");
}
```

Step 4: Integration

```
template <typename... Ts>
int safe_printf(const char * f,
      const Ts&... ts) {
   check_printf(f, normalizeArg(ts)...);
   return printf(f, normalizeArg(ts)...);
}
```

Further improvements

- Extend to all types (easy)
- Add flags, precision etc (easy but $>$1 slide)
- Allow odd cases (e.g. print `long` as pointer)
- Define `safe_scanf`
- Guard the check:

```
#ifndef NDEBUG
   check_printf(f, normalizeArg(ts)...);
#endif
```

`std::tuple`

`std::tuple`

- Largest variadics-related offering in `std`
- “Product type” packing together any number of values of heterogeneous types
- Generalizes, plays nice with `std::pair`
- Store layout not specified
  - $+$ Implementation is free to choose optimally
    - $-$ Currently neither does
  - $-$ No prefix/suffix property

`std::tuple` introduction

```
tuple<int, string, double> t;
static_assert(tuple_size<decltype(t)>::value
    == 3, "Rupture in the Universe.");
get<0>(t) = 42;
assert(get<0>(t) == 42);
get<1>(t) = "forty-two";
get<2>(t) = 0.42;
```

The usual suspects

- Constructors, assignment
- `make_tuple`
- Equality and ordering comparisons
- `swap`

Less usual suspects

- `pack_arguments`
- `tie`
- `tuple_cat`
- Allocator constructors, `uses_allocator`
- Range primitives `begin`, `end`

`std::tuple` structure

```
template <class... Ts> class tuple {};
template <class T, class... Ts>
class tuple<T, Ts...> : private tuple<Ts...> {
private:
   T head_;
   ...
};
```

- Head is a *suffix* of the structure
- No prescribed layout properties

Implementing `std::get`

- Let's first implement the kth type

```
template <size_t, class> struct tuple_element;
template <class T, class... Ts>
struct tuple_element<0, tuple<T, Ts...>> {
   typedef T type;
};
template <size_t k, class T, class... Ts>
struct tuple_element<k, tuple<T, Ts...>> {
   typedef
      typename tuple_element<k-1, tuple<Ts...>>::type
      type;
};
```

Implementing `std::get`: base case

- Shouldn't be a member!
- `t.template get<1>()`, ew

```
template <size_t k, class... Ts>
typename enable_if<k == 0,
   typename tuple_element<0,
      tuple<Ts...>>::type&>::type
get(tuple<Ts...>& t) {
   return t.head();
}
```

Implementing `std::get`: recursion

```
template <size_t k, class T, class... Ts>
typename enable_if<k != 0,
   typename tuple_element<k,
      tuple<T, Ts...>>::type&>::type
get(tuple<T, Ts...>& t) {
   tuple<Ts...> & super = t; // get must be friend
   return get<k - 1>(super);
}
```

Summary

Summary

- Familiar approach: pattern matching with recursion
- Yet new, too: expansion rules and loci
- True variadic functions finally possible
- `std::tuple` a useful abstraction for product types