# `tuple`: What's New, And How It Works

Stephan T. Lavavej ("Steh-fin Lah-wah-wade")

Principal Software Engineer, Visual C++ Libraries

`stl@microsoft.com`

`@StephanTLavavej`

# Getting Started

- Please hold your questions until the end
  - Write down the slide numbers
- Everything here is Standard
  - Unless otherwise specified
- Everything here is available in VS 2015 Update 3
  - Except C++17's `apply()`, `make_from_tuple()`, and `get(const tuple&&)`

# The Most Important Issue!

# Pronunciation!

- tuple is "too-pull", not "tuh-pull"
- Compare super vs. supper
- Imagine suple vs. supple
- Imagine suplex vs. supplex
- Imagine tuple vs. tupple
- Just follow the ordinary rules for pronunciation
  - It's not like V turning into "w" and J turning into "d"
  - That would make absolutely no sense whatsoever

# C++11 `tuple` Examples

# Multiple Return Values

```
tuple<string, int, string> starship() {
  return make_tuple("NCC", 1701, "D");
}
int main() {
  const auto t = starship();
  cout << get<0>(t) << "-" << get<1>(t)
    << "-" << get<2>(t) << endl;
}
// NCC-1701-D
```

# Data Structures

```
vector<tuple<string, string, int>> v{
  { "Armstrong", "Apollo 11", 1969 },
  { "Lovell", "Apollo 13", 1970 },
  { "Cernan", "Apollo 17", 1972 } };
sort(v.begin(), v.end());
for (const auto& t : v) {
  cout << get<0>(t) << ", " << get<1>(t)
    << ", " << get<2>(t) << endl; }
// Armstrong, Apollo 11, 1969
// Cernan, Apollo 17, 1972
// Lovell, Apollo 13, 1970
```

# Implementing Comparisons

```
struct Point {
  int x; int y; int z;
  bool operator<(const Point& p) const {
    return tie(x, y, z) < tie(p.x, p.y, p.z);
  }
};
Point a{ 11, 22, 99 };
Point b{ 11, 33, 0 };
cout << boolalpha << (a < b) << endl;
// true
```

# Piecewise Construction

```
pair<vector<int>, vector<int>>
  p(piecewise_construct,
    forward_as_tuple(),
    forward_as_tuple(3, 1729));
cout << p.first.size() << " "
  << p.second.size() << endl;
// 0 3
```

# Variadic Data Members

- `tuple` is the world's laziest `struct`
  - Multiple return values: lazy
  - Data structures: somewhat lazy
- Implementing comparisons: convenient
  - Core Language deficiency: no default comparisons
- Piecewise construction: only choice for `pair`
  - Core Language limitation: exactly one parameter list
- Variadic data members: only good choice
  - Core Language deficiency: no variadic data members
  - Obnoxious to use without C++14's `integer_sequence`

# C++11 `tuple` Features

# tuple Makers (template Omitted)

- `tuple<VTypes...> make_tuple(Types&&...);`
  - Like `make_pair()`, decays and unrefwraps
  - Don't use explicit template arguments
- `tuple<Types&...> tie(Types&...) noexcept;`
  - Can be used for unpacking assignment (see next slide)
- `tuple<T&&...> forward_as_tuple(T&&...) noexcept;`
  - Consume immediately, don't refrigerate
- `tuple<CTypes...> tuple_cat(Tuples&&...);`
  - Awesome, but impractical

# Unpacking Assignment

```
tuple<string, int, string> starship() {
  return make_tuple("NCC", 1701, "D");
}
string s;
int i = 0;
tie(s, i, ignore) = starship();
cout << s << "-" << i << endl;
// NCC-1701
```

# `tuple<T&>` Is Clever, But Scary

- Core Language: reference data members inhibit copy/move assignment
- Standard Library: "We don't care, we want `tie()`"
- `pair` and `tuple` frequently provide additional functionality beyond the Core Language
  - Conversions, comparisons, etc. are wonderful
- Reference assignment was probably a step too far
  - `vector<T&>` is forbidden (good!)
  - `vector<tuple<T&>>` will misbehave (bad!)

# tuple Type Traits

- `tuple_size<Tuple>::value`
- `tuple_element<I, Tuple>::type`
- They behave like the primary type categories
  - `cv tuple<A, B, C>` is acceptable
  - `tuple<A, B, C>&` is unacceptable
- When taking `T&&`, remember to `remove_reference`
  - `decay` is less typing, but performs more transformations
- They also accept `pair` and `array`

# tuple Type Traits (C++14)

```
template <typename Tuple>
  decltype(auto) get_back(Tuple&& t) {
  return get<tuple_size<
    remove_reference_t<Tuple>
  >::value - 1>(forward<Tuple>(t)); }
tuple<string, int, string> t("NCC", 1701, "D");
cout << get_back(t) << endl;
// D
```

# get<I>()

```
template <size_t I, typename... Types>
  E& get(tuple<Types...>&) noexcept;
template <size_t I, typename... Types>
  E&& get(tuple<Types...>&&) noexcept;
template <size_t I, typename... Types>
  const E& get(const tuple<Types...>&) noexcept;
```

- Where `E` is `typename tuple_element<I, tuple<Types...>>::type`
- `get()` propagates lvalueness/constness

# Why `get()` Is A Non-Member

- There's a reason, just not a very good reason
- In templated code, member `get()` would require `template` disambiguation:
  - `t.template get<0>()`
- `typename` is to `template` as wolf is to direwolf
- `boost::tuple` has non-member and member `get()`
- If you care enough, write a proposal

# C++14 `tuple` Features

# What's New In C++14 <tuple>

- `get<T>()` (N3670)
- `integer_sequence` (N3658)
  - Provided by `<utility>`, incredibly useful with `<tuple>`
- `constexpr` in `<tuple>` (N3471, LWG 2275/2301)
- `tuple_element_t` (N3887)

# get<T>()

```
const tuple<int, string, const char *, string>
  t(1729, "cute", "fluffy", "kittens");
cout << get<int>(t)
  << " " << get<1>(t)
  << " " << get<const char *>(t)
  << " " << get<3>(t) << endl;
// 1729 cute fluffy kittens
```

- `get<string>(t)` would emit a compiler error

# get() Overloads In C++14

- `template <size_t I, typename... Types> constexpr`
  - `E& get(tuple<Types...>&) noexcept;`
  - `E&& get(tuple<Types...>&&) noexcept;`
  - `const E& get(const tuple<Types...>&) noexcept;`
- `template <typename T, typename... Types> constexpr`
  - `T& get(tuple<Types...>&) noexcept;`
  - `T&& get(tuple<Types...>&&) noexcept;`
  - `const T& get(const tuple<Types...>&) noexcept;`
- `get()` still propagates lvalueness/constness
- More overloads for `pair` and `array`

# integer_sequence In <utility>

```
template <typename T, T...>
  struct integer_sequence;
template <typename T, T N>
  using make_integer_sequence
    = integer_sequence<T, 0, 1, etc, N - 1>;
template <size_t... I> using index_sequence
  = integer_sequence<size_t, I...>;
template <size_t N> using make_index_sequence
  = make_integer_sequence<size_t, N>;
template <typename... T> using index_sequence_for
  = make_index_sequence<sizeof...(T)>;
```

# make_integer_sequence Magic

- Previously, our implementation was O(N) recursive
- Users said that other vendors were O(log N) clever
- Now, actual implementation for C1XX/Clang/EDG:

```
template <class _Ty, _Ty _Size>
  using make_integer_sequence
    = __make_integer_seq<
      integer_sequence, _Ty, _Size>;
```

- Why do my job when compilers can do it better?

# Variadic Data Members, Part 1

```
template <typename F, typename... Args> struct MiniBinder {
  F m_f;
  tuple<Args...> m_t;
  explicit MiniBinder(F f, Args... args)
    : m_f(f), m_t(args...) { }
  template <size_t... Idxs>
    decltype(auto) call(index_sequence<Idxs...>) const {
    return m_f(get<Idxs>(m_t)...);
  }
  decltype(auto) operator()() const {
    return call(index_sequence_for<Args...>{});
  } };
```

# Variadic Data Members, Part 2

```
template <typename F, typename... Args>
  auto mini_bind(F f, Args... args) {
  return MiniBinder<F, Args...>(f, args...);
}
int add(int a, int b, int c, int d) {
  return a + b + c + d;
}
int main() {
  auto mb = mini_bind(add, 1000, 700, 20, 9);
  cout << mb() << endl;
}
// 1729
```

# C++17 `tuple` Features

# What's New In C++17 <tuple>

- `apply()` (P0220R1)
- `make_from_tuple()` (P0209R2)
- Logical Operator Type Traits (P0013R1)
  - Provided by <type_traits>, useful in implementing <tuple>
- Improving `pair` And `tuple` (N4387)

# apply() Implementation

```
template <typename F, typename Tuple, size_t... I>
  constexpr decltype(auto) apply_impl(
    F&& f, Tuple&& t, index_sequence<I...>) {
  return invoke(forward<F>(f),
    get<I>(forward<Tuple>(t))...);
}
template <typename F, typename Tuple>
  constexpr decltype(auto) apply(F&& f, Tuple&& t) {
  return apply_impl(forward<F>(f), forward<Tuple>(t),
    make_index_sequence<tuple_size_v<decay_t<Tuple>>>{});
} // ADL defenses omitted
```

# apply() Usage

```
int add(int a, int b, int c, int d) {
  return a + b + c + d;
}
int main() {
  tuple<int, int, int, int> t(1000, 200, 30, 4);
  cout << apply(add, t) << endl;
  pair<string, string> p("me", "ow");
  cout << apply(plus<>{}, p) << endl;
}
// 1234
// meow
```

# make_from_tuple() Implementation

```
template <typename T, typename Tuple, size_t... I>
  constexpr T make_from_tuple_impl(
    Tuple&& t, index_sequence<I...>) {
  return T(get<I>(forward<Tuple>(t))...);
}
template <typename T, typename Tuple>
  constexpr T make_from_tuple(Tuple&& t) {
  return make_from_tuple_impl<T>(forward<Tuple>(t),
    make_index_sequence<tuple_size_v<decay_t<Tuple>>>{});
} // ADL defenses omitted
```

# make_from_tuple() Usage

```
int main() {
  tuple<int, int> t(3, 1729);
  auto v = make_from_tuple<vector<int>>(t);
  for (const auto& e : v) {
    cout << e << " ";
  }
  cout << endl;
}
// 1729 1729 1729
```

# Logical Operators In <type_traits>

```
template <typename... B> struct conjunction
  : first false trait, otherwise last trait { };
```

- Short-circuiting logical AND
- `true_type` when empty

```
template <typename... B> struct disjunction
  : first true trait, otherwise last trait { };
```

- Short-circuiting logical OR
- `false_type` when empty

```
template <typename B> struct negation
  : bool_constant<!B::value> { };
```

# Short-Circuiting

```
conjunction<
  negation<is_same<A, B>>,
  is_constructible<A, X>
>::value
```

- Metaprogramming doesn't usually short-circuit
- `conjunction` and `disjunction` are special
- Minor benefit: possibly improves throughput
- Major benefit: sometimes avoids infinite recursion

# Improving `pair` And `tuple` (N4387)

- C++14 perfect forwarding constructor:

```
template <typename... UTypes> constexpr
  explicit tuple(UTypes&&...);
```

- C++17 perfect forwarding constructor:

```
template <typename... UTypes> constexpr
  EXPLICIT tuple(UTypes&&...);
```

- `pair` and `tuple` now have conditionally-explicit constructors, implemented without Core support

# Implicit Construction

```
void print(const tuple<string, string, string>& t) {
  cout << get<0>(t) << get<1>(t) << get<2>(t) << endl;
}
tuple<string, string, string> rgb() {
  return { "Red", "Green", "Blue" };
}
int main() {
  print(rgb());
  print({ "One", "Two", "Three" });
}
// RedGreenBlue
// OneTwoThree
```

# Explicit Construction

```
void print(const tuple<string, vector<int>>& t) {
  cout << get<string>(t) << " ";
  for (const auto& e : get<vector<int>>(t)) {
    cout << e << " ";
  }
  cout << endl;
}
tuple<string, vector<int>> x("Good", 3);
print(x);
// Good 0 0 0
```

- `print({ "Bad", 4 })` would emit a compiler error

# Infinite Diversity...

```
template <typename... Types> class tuple {
public:
  constexpr tuple();
  tuple(const tuple&) = default;
  tuple(tuple&&) = default;

  // allocator_arg_t ctors not shown here
```

# … In Infinite Constructors

- `constexpr`
  - *EXPLICIT* `tuple(const Types&...);`
- `template <typename... U> constexpr`
  - *EXPLICIT* `tuple(U&&...);`
  - *EXPLICIT* `tuple(const tuple<U...>&);`
  - *EXPLICIT* `tuple(tuple<U...>&&);`
- `template <typename X, typename Y> constexpr`
  - *EXPLICIT* `tuple(const pair<X, Y>&);`
  - *EXPLICIT* `tuple(pair<X, Y>&&);`

# Implementing *EXPLICIT*

# How Do I SFINAE Thee, Part 1

```
template <typename T> auto add0(T t, int i)
  -> decltype(t + i) { return t + i; }
template <typename T> auto add1(T t, int i)
  -> enable_if_t<is_integral_v<T>, decltype(t + i)>
  { return t + i; }
template <typename T> auto add2(T t, int i,
  enable_if_t<is_integral_v<T>, void **> = nullptr)
  -> decltype(t + i) { return t + i; }
template <typename T> auto add3(T t,
  enable_if_t<is_integral_v<T>, int> i)
  -> decltype(t + i) { return t + i; }
```

# How Do I SFINAE Thee, Part 2

```
template <typename T,
  typename = enable_if_t<is_integral_v<T>>>
  auto add4(T t, int i)
  -> decltype(t + i) { return t + i; }
template <typename T,
  enable_if_t<is_integral_v<T>, int> = 0>
  auto add5(T t, int i)
  -> decltype(t + i) { return t + i; }
```

# SFINAE Dispatch

```
template <typename T,
  enable_if_t<is_integral_v<T>, int> = 0>
  decltype(auto) add(T t, int i) {
    return t + i + 1000; }
template <typename T,
  enable_if_t<!is_integral_v<T>, int> = 0>
  decltype(auto) add(T t, int i) {
    return t + i + 2000; }
cout << add(11, 22) << " " << add(0.5, 16) << endl;
// 1033 2016.5
```

# Implementing *EXPLICIT*, Part 1

```
template <typename T> struct Wrap {
  template <typename U> using NotSelf =
    negation<is_same<Wrap,
    remove_const_t<remove_reference_t<U>>>>;
  template <typename U> using EnableImplicit =
    enable_if_t<conjunction_v<NotSelf<U>,
    is_constructible<T, U>, is_convertible<U, T>>, int>;
  template <typename U> using EnableExplicit =
    enable_if_t<conjunction_v<NotSelf<U>,
    is_constructible<T, U>,
    negation<is_convertible<U, T>>>, int>;
```

# Implementing *EXPLICIT*, Part 2

```
  template <typename U, EnableImplicit<U> = 0>
             Wrap(U&& u) : t(forward<U>(u)) { }
  template <typename U, EnableExplicit<U> = 0>
    explicit Wrap(U&& u) : t(forward<U>(u)) { }
  T t;
};
Wrap<string> a("meow");
Wrap<string> b = "purr";
Wrap<vector<int>> x(11);
```

- `Wrap<vector<int>> y = 22;` would emit a compiler error

# C++17 tuple Issues

# LWG 2485 get(const tuple&&)

- template <size_t I, typename... Types> constexpr
  - E& get(tuple<Types...>&) noexcept;
  - E&& get(tuple<Types...>&&) noexcept;
  - const E& get(const tuple<Types...>&) noexcept;
  - const E&& get(const tuple<Types...>&&) noexcept;
- template <typename T, typename... Types> constexpr
  - T& get(tuple<Types...>&) noexcept;
  - T&& get(tuple<Types...>&&) noexcept;
  - const T& get(const tuple<Types...>&) noexcept;
  - const T&& get(const tuple<Types...>&&) noexcept;
- Fixes a subtle hole in the type system

# LWG 2367 `pair()`, `tuple()`

- Constrains `pair` and `tuple`'s default constructors to exist iff all of their types are default constructible
- Fixes `is_default_constructible<tuple<A, B, C>>`
- Recurring theme: in highly generic code, properly constraining constructors is **really** important

# LWG 2312 tuple Arity Constraints

```
void meow(tuple<long, long>) { puts("Two"); }
void meow(tuple<long, long, long>) {
  puts("Three"); }
int main() {
  meow({ 2, 2 });
  tuple<int, int, int> t(3, 3, 3);
  meow(t);
}
// Two
// Three
```

# LWG 2549 Perfect Vs. Converting

- Complicated issue involving dangling references
  - Only 1-`tuples` are affected
- `template <typename... U> constexpr`
  - `EXPLICIT tuple(U&&...);`
  - `EXPLICIT tuple(const tuple<U...>&);`
  - `EXPLICIT tuple(tuple<U...>&&);`
- Resolved by preferring perfect forwarding
- Properly constrain `YourType`'s constructors!

# LWG ???? Perfect Vs. Copy/Move

- For 1-`tuple`s, the perfect forwarding constructor can outcompete the copy/move constructors:

```
template <typename... U> constexpr
  EXPLICIT tuple(U&&...);
tuple(const tuple&) = default;
tuple(tuple&&) = default;
```

- Happens for `tuple<T>&` and `const tuple<T>&&`
  - When `T` is omni-constructible
- All STL implementers agree to prefer copy/move

# More Info

# More Info

- Everything here is available in VS 2015 Update 3
  - Except for three C++17 things
  - `get(const tuple&&)` implemented in VS "15"
  - `apply()`, `make_from_tuple()` implemented locally
    - libc++'s tests found compiler bugs in `constexpr`, etc.
    - Should be available in VS "15", but no promises yet
- C++17 Working Paper
  - `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/n4606.pdf`

# Questions?

stl@microsoft.com

@StephanTLavavej