(Ab)using C++17

About Me

Jason Turner

- Co-host of CppCast http://cppcast.com
- Host of C++ Weekly https://www.youtube.com/c/JasonTurner-lefticus
- Co-creator of ChaiScript http://chaiscript.com
- Curator of http://cppbestpractices.com
- Microsoft MVP for C++ 2015-present
- C++ Trainer - Next Training is in Chicago July 12-14 https://www.eventbrite.com/e/better-c-chicago-registration-34084060342

(Ab)using C++17

- C++17 has a lot of little details and there's a lot to cover
- We're going to highlight features and move pretty quickly
- Be sure to interrupt and ask questions
- There's a lot of things we'll be leaving out also
- Keep in mind we're going to have fun with abusing some of these features, but hopefully always bring it back to something practical

Fold Expressions

Fold Expressions

```
( ... <op> <pack expression> ) // unary left fold
( <pack expression> <op> ... ) // unary right fold
( <init> <op> ... <op> <pack expression> ) // binary left fold
( <pack expression> <op> ... <op> <init> ) // binary right fold
```

Allows for "folding" of a variadic parameter pack into a single value.

Allowed operations:

`+` `-` `*` `/` `%` `^` `&` `|` `<<` `>>` `+=` `-=` `*=` `/=` `%=` `^=` `&=` `|=` `<<=` `>>=` `=` `==` `!=` `<` `>` `<=` `>=` `&&` `||` `,` `.*` `->*`

Unary left fold

```
( ... <op> <pack expression> ) // unary left fold
```

```
( ... && args )
```

Expands to:

```
( ( arg1 && arg2 ) && arg3 )
```

Unary right fold

```
( <pack expression> <op> ... ) // unary right fold
```

```
( args && ... )
```

Expands to:

```
( arg1 && ( arg2 && arg3 ) )
```

Binary left fold

```
( <init> <op> ... <op> <pack expression> )
```

```
( true && ... && args )
```

Expands to:

```
( ( ( true && arg1 ) && arg2 ) && arg3 )
```

Binary right fold

```
( <pack expression> <op> ... <op> <init> )
```

```
( args && ... && true )
```

Expands to:

```
( arg1 && ( arg2 && ( arg3 && true ) ) )
```

for_each_argument from Eric Niebler / Sean Parent

https://twitter.com/ericniebler/status/559119062895431680

```
#include <initializer_list>
#include <utility>

// Apply a function to each argument of a variadic template
template <class F, class... Ts>
void for_each_argument(F f, Ts&&... a) {
 (void)std::initializer_list<int>{(f(std::forward<Ts>(a)), 0)...};
}

int main()
{
  for_each_argument([](const auto &arg){ std::cout << arg; },
                    1,2,3,4);
}
```

for_each_argument in C++17

C++17's fold expressions help us clean this up.

```
#include <utility>

template<class F, class... Ts>
void for_each_argument(F &&f, Ts&& ...a) {
  // Note the cast to (void) which should be in the C++11 version also
  ( (void)f(std::forward<Ts>(a)), ... );
}
```

for_each_argument in C++17

```
#include <utility>

template<class F, class... Ts>
void for_each_argument(F &&f, Ts&& ...a) {
  //* is there any way we could keep the results of these functions?
  ( (void)f(std::forward<Ts>(a)), ... );
}
```

for_each_argument in the future?

```cpp
#include <utility>

template<class F, class... Ts>
auto for_each_argument(F &&f, Ts&& ...a) {

  return std::tuple{ f(std::forward<Ts>(a)), ... };
}
```

for_each_argument in the future?

```
#include <utility>

template<class F, class... Ts>
auto for_each_argument(F &&f, Ts&& ...a) {
  // oops, we're skipping ahead to template type deduction
  return std::tuple{ f(std::forward<Ts>(a)), ... };
}
```

for_each_argument in the future?

```
#include <utility>

template<class F, class... Ts>
auto for_each_argument(F &&f, Ts&& ...a) {
  // too bad we don't have regular void...
  return std::tuple{ f(std::forward<Ts>(a)), ... };
}
```

For Each Tuple Element

We can call a function on each element of a tuple with something like this:

```
template<typename T, typename Callable, size_t ... Indexes>
void for_each_elem_impl(T &&tuple, Callable &&callable,
                        std::index_sequence<Indexes...>) {
  ((void)callable(std::get<Indexes>(std::forward<T>(tuple))), ...);
}

template<typename T, typename Callable>
void for_each_elem(T &&tuple, Callable &&callable) {
  for_each_elem_impl(std::forward<T>(tuple),
                     std::forward<Callable>(callable),
                     std::make_index_sequence<std::tuple_size<T>{}>());
}
```

For Each Tuple Element

```
template<typename T, typename Callable, size_t ... Indexes>
void for_each_elem_impl(T &&tuple, Callable &&callable,
                        std::index_sequence<Indexes...>) {
  //* expands to:
  //* callable(std::get<0>(std::forward<T>(tuple))),
  //* callable(std::get<1>(std::forward<T>(tuple))), ...
  ((void)callable(std::get<Indexes>(std::forward<T>(tuple))), ...);
}

template<typename T, typename Callable>
void for_each_elem(T &&tuple, Callable &&callable) {
  for_each_elem_impl(std::forward<T>(tuple),
                     std::forward<Callable>(callable),
                     std::make_index_sequence<std::tuple_size<T>{}>());
}
```

For Each Tuple Element - Refined

```
template<typename Callable, typename Tuple>
void for_each_elem(Callable &&func, Tuple &&tuple)
{
  std::apply(
    [&func](auto&&... xs) {
      ((void)func(std::forward<decltype(xs)>(xs)), ...);
    },
    std::forward<Tuple>(tuple)
  );
}
```

I heard this one came from Vittorio?

It's obvious what it's doing, right?

For Each Tuple Element - Refined

```
template<typename Callable, typename Tuple>
void for_each_elem(Callable &&func, Tuple &&tuple)
{
  std::apply(
    [&func](auto&&... xs) {
      ((void)func(std::forward<decltype(xs)>(xs)), ...);
    },
    std::forward<Tuple>(tuple)
  );
}

for_each_elem([](const auto &v){std::cout << v;},
              std::tuple{1,2,3,4});
```

For Each Tuple Element - Refined

```
template<typename Callable, typename Tuple>        //*
void for_each_elem(Callable &&func, Tuple &&tuple)//*
{
  std::apply(
    [&func](auto&&... xs) {
      ((void)func(std::forward<decltype(xs)>(xs)), ...);
    },
    std::forward<Tuple>(tuple)
  );
}

for_each_elem([](const auto &v){std::cout << v;}, //*
              std::tuple{1,2,3,4});               //*
```

For Each Tuple Element - Refined

```
auto func = [](const auto &v){std::cout << v;}; //*
auto tuple = std::tuple{1,2,3,4};               //*

std::apply(
  [&func](auto&&... xs) {
    ((void)func(std::forward<decltype(xs)>(xs)), ...);
  },
  std::forward<Tuple>(tuple)
);
```

For Each Tuple Element - Refined

```cpp
auto func = [](const auto &v){std::cout << v;};

std::apply(
  [&func](auto&&... xs) {
    ((void)func(std::forward<decltype(xs)>(xs)), ...);
  },
  std::tuple{1,2,3,4}  //*
);
```

For Each Tuple Element - Refined

What does `std::apply` do?

```cpp
auto func = [](const auto &v){std::cout << v;};

std::apply(
  [&func](auto&&... xs) {
    ((void)func(std::forward<decltype(xs)>(xs)), ...);
  },
  std::tuple{1,2,3,4}
);
```

For Each Tuple Element - Refined

What does `std::apply` do?

answer: it calls a function by unpacking all of the elements of the tuple into function arguments.

```cpp
auto func = [](const auto &v){std::cout << v;};

std::apply(  //*
  [&func](auto&&... xs) {
    ((void)func(std::forward<decltype(xs)>(xs)), ...);
  },
  std::tuple{1,2,3,4}
);
```

For Each Tuple Element - Refined

```cpp
auto func = [](const auto &v){std::cout << v;};

auto application = [&func](auto &&...xs){
  ((void)func(std::forward<decltype(xs)>(xs)), ...);
};

application(1, 2, 3, 4);
```

For Each Tuple Element - Refined

```cpp
auto func = [](const auto &v){std::cout << v;};

((void)func(1), (void)func(2), (void)func(3), (void)func(4));
```

For Each Tuple Element - Refined

Awesome, but terrifying levels of template expansions and instantiations for code that's concerned about compile times.

```
template<typename Callable, typename Tuple>
void for_each_elem(Callable &&func, Tuple &&tuple)
{
  std::apply(
    [&func](auto&&... xs) {
      ((void)func(std::forward<decltype(xs)>(xs)), ...);
    },
    std::forward<Tuple>(tuple)
  );
}
```

For Each Tuple Element

What else is a "tuple" besides `std::tuple`?

- `std::pair`
- `std::array`

For Each Array Element

```
for_each_elem(do_thing, array);
// vs
std::for_each(std::begin(array), std::end(array), do_thing);
```

Why choose `for_each_elem` over `std::for_each`?

- `for_each` is a loop which might be unrolled
- `for_each_elem` is force-unrolled
- Depending on the application, `for_each_elem`'s forced unrolling could increase inlining and constant folding
- `for_each` gives the compiler more flexibility to choose between unrolling or not

Pointer to Member Folds

Did you notice that you can fold on `.*` and `->*` operators?

Maybe we can use this to build a method to chain function calls by name?

```
//goal, call obj.member1 then call obj.member2 on the result value
chain(obj, &Obj::member1, &Obj::member2);
```

Pointer to Member Folds

```
// Attempt:
template<typename O, typename ... T>
decltype(auto) chain_members(O &&obj, T && ... t)
{
  return (obj .* ... .* t)();
}
```

Pointer to Member Folds

```
struct S {
  S& do_thing() { return *this; }
  S& do_thing_2() { return *this; };
};

template<typename O, typename ... T>
decltype(auto) chain_members(O &&obj, T && ... t) {
  return (obj .* ... .* t)();
}

int main() {
  S s;
  chain_members(s, &S::do_thing);
}
```

Pointer to Member Folds

```
struct S {
  S& do_thing() { return *this; }
  S& do_thing_2() { return *this; };
};

template<typename O, typename ... T>
decltype(auto) chain_members(O &&obj, T && ... t) {
  return (obj .* ... .* t)();
}

int main() {
  S s;
  chain_members(s, &S::do_thing);
  //* parses to: (s.*do_thing)()
  //* works
}
```

Pointer to Member Folds

```cpp
struct S {
  S& do_thing() { return *this; }
  S& do_thing_2() { return *this; };
};

template<typename O, typename ... T>
decltype(auto) chain_members(O &&obj, T && ... t) {
  return (obj .* ... .* t)();
}

int main() {
  S s;
  chain_members(s, &S::do_thing, &::do_thing_2);
}
```

Pointer to Member Folds

```
struct S {
  S& do_thing() { return *this; }
  S& do_thing_2() { return *this; };
};

template<typename O, typename ... T>
decltype(auto) chain_members(O &&obj, T && ... t) {
  return (obj .* ... .* t)();
}

int main() {
  S s;
  chain_members(s, &S::do_thing, &::do_thing_2);
  //* parses to: ((s.*do_thing).*do_thing_2)()
  //* fails to compile
}
```

Pointer to Member Folds

I could not find any way to invoke a call to each member function, but it is possible to chain data members.

Pointer to Member Folds

```
struct S {
  void do_thing() { /* do thing */ }
  S* lhs;
  S* rhs;
};

template<typename O, typename ... T>
decltype(auto) walk_tree(O *o, T && ... t) {
  return ( o ->* ... ->* t )();
}

int main() {
  S s;
  walk_tree( &s, &S::lhs, &S::rhs, &S::rhs, &S::do_thing);
  // this works in the same way that C "works"
}
```

Pointer to Member Folds

Is anyone aware of any actual use for this ability?

Fold Expressions For Testing Template Performance

We can easily instantiate many 1000's of templates with a fold expression.

Fold Expressions For Testing Template Alias Performance

Which compiles faster, A or B?

```
#include <utility>
#include <type_traits>
template<std::size_t T> struct S {};

template<std::size_t ... Indexes>
void test_templates(std::index_sequence<Indexes...>) {
#ifdef TEST_V
  (std::is_nothrow_constructible_v<S<Indexes>>, ...);//* A
#else
  (std::is_nothrow_constructible<S<Indexes>>{}, ...);//* B
#endif
}
int main() { test_templates(std::make_index_sequence<8000>()); }
```

Fold Expressions For Testing Template Performance

g++ (GCC) 7.0.1 20170228

```
(std::is_nothrow_constructible_v<S<Indexes>>, ...);//* A
```

9.63s 824MB RAM

```
(std::is_nothrow_constructible<S<Indexes>>{}, ...);//* B
```

19.51s 1400MB RAM

Fold Expressions For Testing Conjunction

Which compiles faster, A or B?

```cpp
#include <utility>
#include <type_traits>
template<std::size_t T> struct S {};

template<std::size_t ... Indexes>
bool test_templates(std::index_sequence<Indexes...>) {
#ifdef TEST_CONJUNCTION
  return std::conjunction_v<
           std::is_nothrow_constructible<S<Indexes>>...>;      //* A
#else
  return (std::is_nothrow_constructible_v<S<Indexes>> && ...); //* B
#endif
}

int main() {
  test_templates(std::make_index_sequence<3000>());
}
```

Fold Expressions For Testing Conjunction

```
std::conjunction_v<std::is_nothrow_constructible<S<Indexes>>...>;//* A
```

/usr/local/opt/include/c++/7.0.1/type_traits:1577:67: fatal error: template instantiation depth exceeds maximum of 1000 (use -ftemplate-depth= to increase the maximum)

```
(std::is_nothrow_constructible_v<S<Indexes>> && ...);           //* B
```

10s 327MB RAM

Fold Expressions For Testing Conjunction

After adjusting `-ftemplate-depth`

```
std::conjunction_v<std::is_nothrow_constructible<S<Indexes>>...>;//* A
```

20s 1845MB RAM

```
(std::is_nothrow_constructible_v<S<Indexes>> && ...);              //* B
```

10s 327MB RAM

Anyone know why a recursive template would have even been used here instead of a C++11 style fold in the worse case?

Destructuring

Structured Bindings

```
auto [a,b,c] = <expression>;
```

Can be used to automatically split a structure into multiple variables

Structured Bindings

What value is returned?

```
int main() {
  std::tuple t{0,0};
  auto &[a,b] = t;
  a = 5;
  return std::get<0>(t); //*
}
```

Structured Bindings

What value is returned?

```
int main() {
  std::tuple t{0,0};
  auto &[a,b] = t;
  a = 5;
  return std::get<0>(t); //* returns 5
}
```

Structured Bindings

Also work for user defined types

```cpp
struct S {
  int i;
  int j;
};

int main() {
  S s;
  auto &[a, b] = s;
}
```

Structured Bindings

And arrays

```
int main() {
  std::array<int, 2> array{1,2};
  auto &[a, b] = array;
}
```

Structured Bindings

Also works for other types of initialization

```
int main() {
  std::tuple<int, int> t;
  auto &[a,b] = t;
  auto &[c,d](t); // these are almost unreadable in my mind
  auto &[e,f]{t};
}
```

Destructuring Standard Containers

```
template<std::size_t Start, typename T, std::size_t ... S>
auto destructure_impl(T &&t, std::index_sequence<S...>) {
  // returns references to contained things
  return std::forward_as_tuple(
      *std::next(std::begin(std::forward<T>(t)), Start+S)...);
}

template<std::size_t Start, std::size_t Count, typename T>
auto destructure(T &&t) {
  return destructure_impl<Start>(
      std::forward<T>(t), std::make_index_sequence<Count>());
}

template<std::size_t Count, typename T>
auto destructure(T &&t) {
  return destructure<0, Count>(std::forward<T>(t));
}
```

Destructuring Standard Containers

```
int main() {
  std::vector<int> v{1,2,3};
  auto [a,b] = destructure<2>(v);   //* returns references
  auto [x,y] = destructure<1,2>(v);
  x = 3;
  y = 12;
  return v[2] + a + b; //* returns?
}
```

Destructuring Standard Containers

```cpp
int main() {
  std::vector<int> v{1,2,3};
  auto [a,b] = destructure<2>(v);
  auto [x,y] = destructure<1,2>(v);
  x = 3;
  y = 12;
  return v[2] + a + b; //* 16
}
```

If-inits

if-init expressions

```
if (auto [key, value] = *my_map.begin(); key == "mykey"){}
```

- Added in C++17
- Also works for `switch` conditions
- The combination of structured bindings and if/switch init expressions will probably change the way we interact with and design libraries

C++ `using` Blocks

Give us a handy way of making a "using" block in C++ like other languages have:

```
if (auto var = someFunc(); &var /* pointer will evaluate as true */)
{
  // use var
}
```

This is a bit abusive, however.

C++ `using` Blocks

Maybe...?

```cpp
if (auto var = some_thing(); true)
{
  // var
}
```

C++ `using` Blocks

More practical applications.

```cpp
if (std::ofstream ofs(path); ofs.good())
{ /*...*/ }
```

```cpp
if (auto optionalResult = someFunction(); optionalResult)
{ /*...*/ }
```

```cpp
if (std::cmatch results; std::regex_match("str", regex, results))
{ /*...*/ }
```

Class Template Type Deduction

Class Template Type Deduction

In C++ pre-17 we have templates like this:

```cpp
template<typename First, typename Second>
struct Pair {
  Pair(First t_first, Second t_second)
    : first(std::move(t_first)), second(std::move(t_second))
  {}

  First first;
  Second second;
};

int main() {
  Pair<int, double> p(1, 2.3); //*
}
```

Class Template Type Deduction

So we added helpers:

```
template<typename First, typename Second>
struct Pair {
  Pair(First t_first, Second t_second)
    : first(std::move(t_first)), second(std::move(t_second))
  {}

  First first;
  Second second;
};

template<typename First, typename Second>
constexpr auto make_pair(First &&first, Second &&second) {
  return
    std::pair<std::decay_t<First>, std::decay_t<Second>>(
      std::forward<First>(first), std::forward<Second>(second));
}

int main() {
  auto p = std::make_pair(1, 2.3); //*
}
```

Class Template Type Deduction

C++17 adds class template type deduction, so the helper isn't needed now

```
template<typename First, typename Second>
struct Pair {
  Pair(First t_first, Second t_second)
    : first(std::move(t_first)), second(std::move(t_second))
  {}

  First first;
  Second second;
};

int main() {
  Pair p{1, 2.3}; //*
}
```

Class Template Type Deduction

Which can lead to...

```cpp
template<typename First, typename Second>
struct Pair {
  Pair(First t_first, Second t_second)
    : first(std::move(t_first)), second(std::move(t_second))
  {}

  First first;
  Second second;
};

int main() {
  Pair p{1, 2.3};
  Pair p2{2.3, 3.4};
  p = p2; //* what is p equal to?
}
```

Class Template Type Deduction

But automatic deduction only works if the constructor parameters match the class template parameters.

Class Template Type Deduction

```cpp
template<typename First, typename Second>
struct Pair {
  template<typename P1, typename P2>
  Pair(P1 &&p1, P2 &&p2)
    : first(std::forward<P1>(p1)), second(std::forward<P2>(p2))
  {}

  First first;
  Second second;
};

int main() {
  Pair p{1, 2.3}; //* ambiguous: deduction now fails
}
```

Class Template Type Deduction

So C++17 also added deduction guides

```
template<typename First, typename Second>
struct Pair {
  template<typename P1, typename P2>
  Pair(P1 &&p1, P2 &&p2)
    : first(std::forward<P1>(p1)), second(std::forward<P2>(p2))
  {}

  First first;
  Second second;
};

template<typename P1, typename P2>                                    //*
Pair(P1 &&p1, P2 &&p2) -> Pair<std::decay_t<P1>, std::decay_t<P2>>;//*

int main() {
  Pair p{1, 2.3}; //* deduction now succeeds
}
```

Abusing Deduction Guides

C++ still doesn't have a `function_traits` helper, but C++17's deduction guides now work with `std::function`

```
int main()
{
  std::function f = [](const int) {} //* yay deduction
}
```

Abusing Deduction Guides

```cpp
int main()
{
  std::function f = [](const int) {};
  // The type of `f` contains all the information we want
}
```

Abusing Deduction Guides

```
int main()
{
  std::function f = [](const int) {}
  // The type of `f` contains all the information we want
  // how can we abuse this?
}
```

Abusing Deduction Guides

```
#include <functional>
#include <tuple>

template<typename Signature> struct Function_Traits_Impl;

template<typename Ret, typename ... Param>
struct Function_Traits_Impl<Ret (Param...)> {
  using Return_Type = Ret;
  constexpr static auto arity = sizeof...(Param);
  using Param_Types = std::tuple<Param...>;
};

template<typename Sig>
struct Function_Traits_Impl<std::function<Sig>>
  : Function_Traits_Impl<Sig> { };

template<typename Func>
struct Function_Traits
  : Function_Traits_Impl<
      decltype(std::function{std::declval<Func>()})
    >
{
};
```

Abusing Deduction Guides

```cpp
#include <functional>
#include <tuple>

template<typename Signature> struct Function_Traits_Impl;

template<typename Ret, typename ... Param>
struct Function_Traits_Impl<Ret (Param...)> {
  using Return_Type = Ret;
  constexpr static auto arity = sizeof...(Param);
  using Param_Types = std::tuple<Param...>;
};

template<typename Sig>
struct Function_Traits_Impl<std::function<Sig>>
  : Function_Traits_Impl<Sig> { };

template<typename Func>
struct Function_Traits
  : Function_Traits_Impl<
      decltype(std::function{std::declval<Func>()}) //* 1) deduce
    >
{
};
```

Abusing Deduction Guides

```cpp
#include <functional>
#include <tuple>

template<typename Signature> struct Function_Traits_Impl;

template<typename Ret, typename ... Param>
struct Function_Traits_Impl<Ret (Param...)> {
  using Return_Type = Ret;
  constexpr static auto arity = sizeof...(Param);
  using Param_Types = std::tuple<Param...>;
};

template<typename Sig>
struct Function_Traits_Impl<std::function<Sig>>
  : Function_Traits_Impl<Sig> { }; //* 2) extract

template<typename Func>
struct Function_Traits
  : Function_Traits_Impl<
      decltype(std::function{std::declval<Func>()}) // 1) deduce
    >
{
};
```

Abusing Deduction Guides

```cpp
#include <functional>
#include <tuple>

template<typename Signature> struct Function_Traits_Impl;

template<typename Ret, typename ... Param>
struct Function_Traits_Impl<Ret (Param...)> {//* 3) unwrap
  using Return_Type = Ret;
  constexpr static auto arity = sizeof...(Param);
  using Param_Types = std::tuple<Param...>;
};

template<typename Sig>
struct Function_Traits_Impl<std::function<Sig>>
  : Function_Traits_Impl<Sig> { }; // 2) extract

template<typename Func>
struct Function_Traits
  : Function_Traits_Impl<
      decltype(std::function{std::declval<Func>()}) // 1) deduce
    >
{
};
```

Abusing Deduction Guides

```
int free_func() {};

int main()
{
  auto l = [i = 5](const int j, const double d) {};
  static_assert(Function_Traits<decltype(l)>::arity == 2); //* 4) Use
  static_assert(Function_Traits<decltype(&free_func)>::arity == 0);
}
```

Abusing Deduction Guides

Deduction guides must be in the form of:

```
Type(Args) -> Type<concrete template type>
```

Example:

```
template<typename P1, typename P2>
Pair(P1 &&p1, P2 &&p2) -> Pair<std::decay_t<P1>, std::decay_t<P2>>;
```

Using Deduction Guides

Real function type traits:

```
// Free functions
template<typename Ret, typename ... Param>
Function_Signature(Ret (*f)(Param...))
     -> Function_Signature<Ret, Function_Params<Param...>>;

template<typename Ret, typename ... Param>
Function_Signature(Ret (*f)(Param...) noexcept)
     -> Function_Signature<Ret, Function_Params<Param...>,true>;

// no reference specifier
template<typename Ret, typename Class, typename ... Param>
Function_Signature(Ret (Class::*f)(Param ...) volatile)
     -> Function_Signature<Ret, Function_Params<volatile Class &, Param...>>

template<typename Ret, typename Class, typename ... Param>
Function_Signature(Ret (Class::*f)(Param ...) volatile noexcept)
     -> Function_Signature<Ret, Function_Params<volatile Class &, Param...>,

/// etc etc etc etc...
/// etc.
```

Using Deduction Guides

After we've made deduction guides for all the free function and member function types, what kind of callable thing is left?

Using Deduction Guides

```
// Assuming we have a Function_Signature for all possible matches
// but still want to match on a callable with operator()
template<typename Func>
Function_Signature(Func &&) -> Function_Signature<
  typename decltype(Function_Signature{&std::decay_t<Func>::operator()})
                   ::Return_Type,
  typename decltype(Function_Signature{&std::decay_t<Func>::operator()})
                   ::Param_Types,
  decltype(Function_Signature{&std::decay_t<Func>::operator()})
                   ::is_noexcept,
  false,
  true
 >;
```

`if constexpr`

`if constexpr`

A compile-time conditional block

```
if constexpr( /* constant expression */ ) {
  // if true, this block is compiled
} else {
  // if false, this block is compiled
}
```

if constexpr

```
#include <type_traits>

template<typename T, typename U>
auto remainder(T dividend, U divisor ) {
  if constexpr(std::is_floating_point_v<T>
               || std::is_floating_point_v<U>) {
    return 0.0;
  } else {
    return dividend % divisor;
  }
}
```

if constexpr

```cpp
#include <type_traits>

template<typename T, typename U>
auto remainder(T dividend, U divisor ) {
  if constexpr(std::is_floating_point_v<T>       //*
                  || std::is_floating_point_v<U>) { //*
    return 0.0;
  } else {
    return dividend % divisor;
  }
}
```

if constexpr

```cpp
#include <type_traits>

template<typename T, typename U>
auto remainder(T dividend, U divisor ) {
  if constexpr(std::is_floating_point_v<T>
               || std::is_floating_point_v<U>) {
    return 0.0; //* what is the remainder of float / float?
  } else {
    return dividend % divisor;
  }
}
```

`if constexpr`

Note that it does not short circuit as you might expect

```
#include <type_traits>

template<typename T>
void do_thing(T t) {
  if constexpr (std::is_integral_v<T>
                && std::is_same_v<std::make_signed_t<T>, int>) { //*
    // this is a signed or unsigned int equivalent
  } else {
    // it's something else entirely
  }
}

int main() {
  do_thing(0.0); //* oops compile time error
}
```

`if constexpr` vs SFINAE

`if constexpr`

```cpp
#include <type_traits>
template<typename T, typename U>
auto remainder(T dividend, U divisor ) {
  if constexpr(std::is_floating_point_v<T>
               || std::is_floating_point_v<U>) {
    return 0.0;
  } else {
    return dividend % divisor;
  }
}
```

`if constexpr` vs SFINAE

SFINAE (I don't like SFINAE)

```
template<typename T, typename U>
auto remainder(T dividend, U divisor,
               std::enable_if_t<std::is_floating_point_v<T>
                          || std::is_floating_point_v<U>> * = nullptr) {
  return 0.0;
}

template<typename T, typename U>
auto remainder(T dividend, U divisor,
               std::enable_if_t<!std::is_floating_point_v<T>
                         && !std::is_floating_point_v<U>> * = nullptr) {
  return dividend % divisor;
}
```

`if constexpr` vs SFINAE

But which one is faster to compile? SFINAE or `if constexpr` usage?

We'll use our `for_each_elem` to help us test

```
int main() {
  std::tuple<std::uint8_t, std::uint16_t, std::uint32_t, std::uint64_t,
             std::int8_t, std::int16_t, std::int32_t, std::int64_t,
             short, int, long, long long,
             char, unsigned char,
             unsigned short, unsigned int, unsigned long, unsigned long long,
             float, double, long double> t{};

  for_each_elem(t,
    [inner = t](const auto lhs){
      for_each_elem(inner,
        [lhs](const auto rhs) {
          remainder(lhs, rhs);
        }
      );
    }
  );
}
```

`if constexpr` vs SFINAE

`if constexpr`

1.76s 72128k

SFINAE

1.78s 73032k

slight advantage to `if constexpr` for this test

Lambdas with multiple signatures

Lambdas With Multiple Signatures!

Say we want a lambda with 3 different signatures:

```
callback()
```

```
callback(uint8_t)
```

```
callback(uint16_t)
```

```
auto callback = [](/* what here? */){
};
```

Lambdas With Multiple Signatures!

```
auto callback = [](auto ... p){
};
```

Lambdas With Multiple Signatures!

We want to accept calls with 0 or 1 parameters

```
auto callback = [](auto ... p){
  static_assert(sizeof...(p) == 0 || sizeof...(p) == 1);
};
```

Lambdas With Multiple Signatures!

```
auto callback = [](auto ... p){
  static_assert(sizeof...(p) == 0 || sizeof...(p) == 1);

  if constexpr(sizeof...(p) == 0) {
    return std::uint8_t(0);  // if 0 params, return a 0
  } else if constexpr(sizeof...(p) == 1) {
    // handle the 1 parameter cases
  }
};
```

Lambdas With Multiple Signatures!

```
auto callback = [](auto ... p){
  static_assert(sizeof...(p) == 0 || sizeof...(p) == 1);

  if constexpr(sizeof...(p) == 0) {
    return std::uint8_t(0);  // if 0 params, return a 0
  } else if constexpr(sizeof...(p) == 1) {
    // limit to our allowed possibilities
    static_assert(std::is_same_v<decltype(p)..., std::uint16_t> //*
                  || std::is_same_v<decltype(p)..., std::uint8_t>); //*
  }
};
```

Lambdas With Multiple Signatures!

```
auto callback = [](auto ... p){
  static_assert(sizeof...(p) == 0 || sizeof...(p) == 1);

  if constexpr(sizeof...(p) == 0) {
    return std::uint8_t(0);  // if 0 params, return a 0
  } else if constexpr(sizeof...(p) == 1) {
    static_assert(std::is_same_v<decltype(p)..., std::uint16_t>
                  || std::is_same_v<decltype(p)..., std::uint8_t>);
    if constexpr(std::is_same_v<decltype(p)..., std::uint8_t>) {
      return p * 2; //* but this cannot compile, unexpanded
    }
  }
};
```

Lambdas With Multiple Signatures!

```
auto callback = [](auto ... p){
  static_assert(sizeof...(p) == 0 || sizeof...(p) == 1);

  if constexpr(sizeof...(p) == 0) {
    return std::uint8_t(0);  // if 0 params, return a 0
  } else if constexpr(sizeof...(p) == 1) {
    static_assert(std::is_same_v<decltype(p)..., std::uint16_t>
                  || std::is_same_v<decltype(p)..., std::uint8_t>);
    if constexpr(std::is_same_v<decltype(p)..., std::uint8_t>) {
      return (p + ...) * 2; //* or maybe fold expression?
    }
  }
};
```

Lambdas With Multiple Signatures!

```
auto callback = [](auto ... p){
  static_assert(sizeof...(p) == 0 || sizeof...(p) == 1);

  if constexpr(sizeof...(p) == 0) {
    return std::uint8_t(0);  // if 0 params, return a 0
  } else if constexpr(sizeof...(p) == 1) {
    static_assert(std::is_same_v<decltype(p)..., std::uint16_t>
                  || std::is_same_v<decltype(p)..., std::uint8_t>);
    if constexpr(std::is_same_v<decltype(p)..., std::uint8_t>) {
      return std::get<0>(std::forward_as_tuple(p...)) * 2;//* or...
    }
  }
};
```

Lambdas With Multiple Signatures!

There, it takes 0 or 1 parameter of type `uint16_t` or `uint8_t`

```
auto callback = [](auto ... p){
  static_assert(sizeof...(p) == 0 || sizeof...(p) == 1);

  if constexpr(sizeof...(p) == 0) {
    return std::uint8_t(0);  // if 0 params, return a 0
  } else if constexpr(sizeof...(p) == 1) {
    static_assert(std::is_same_v<decltype(p)..., std::uint16_t>
                  || std::is_same_v<decltype(p)..., std::uint8_t>);
    if constexpr(std::is_same_v<decltype(p)..., std::uint8_t>) {
      return std::get<0>(std::forward_as_tuple(p...)) * 2;
    } else { // uint16_t is the only option left
      return std::get<0>(std::forward_as_tuple(p...)) * 4;
    }
  }
};
```

Better lambdas with multiple signatures

Variadic Using

Quite simply, allows `using` declarations to be variadic

```
template<typename ... Base>
struct Merged : Base...  {
  constexpr Merged(Base ... base) : Base(std::move(base))...
  { }
  using Base::operator()...;
};
```

This creates a callable object that merges the passed in callable objects

Variadic Using

Constructs an object with 3 different call operators

```
template<typename ... Base>
struct Merged : Base...  {
  constexpr Merged(Base ... base) : Base(std::move(base))...
  { }
  using Base::operator()...;
};

Merged m {
  []{ return std::uint8_t(0); },
  [](const std::uint8_t v){ return v * 2; },
  [](const std::uint16_t v){ return v * 4; }
};
```

Variadic Using

More completely we should specify the `noexcept`

```
template<typename ... Base>
struct Merged : Base... {
  constexpr Merged(Base ... base)
      noexcept((std::is_nothrow_move_constructible_v<Base> && ...))
    : Base(std::move(base))...
  { }
  using Base::operator()...;
};
```

Returning Multiple Values

Returning multiple values

C++17 now allows us to return multiple values from a function!

```
auto get_values()
{
  std::string s1 = someFunction();
  std::string s2 = someOtherFunction();
  return std::tuple{s1, s2};
}
```

What's not ideal about this?

Returning multiple values

Should we move the values out?

```
auto get_values()
{
  std::string s1 = someFunction();
  std::string s2 = someOtherFunction();
  return std::tuple{std::move(s1), std::move(s2)};
}
```

Returning multiple values

But this isn't ideal either.

```cpp
auto get_values()
{
  std::string s1 = someFunction();
  std::string s2 = someOtherFunction();
  return std::tuple{std::move(s1), std::move(s2)};
}
```

Returning multiple values

Remember that moved-from objects still must be destructed

```
auto get_values()
{
  std::string s1 = someFunction();      //*
  std::string s2 = someOtherFunction(); //*
  return std::tuple{std::move(s1), std::move(s2)};
}
```

Returning multiple values

Remember that moved-from objects still must be destructed

```
auto get_values()
{
  return std::tuple{someFunction(), someOtherFunction()};
}
```

Returning multiple values

I think that efficiently utilizing multiple return values will be difficult.

Avoiding Dynamic Allocations While Allowing Runtime Polymorphism

Polymorphism Without Dynamic Allocation

C++17 adds `std::variant` which can be any one type:

```
std::variant<int, double, string> v;
v = std::string("Hello World");
```

Polymorphism Without Dynamic Allocation

Because variant knows the sizes of all possible types it might hold, it does not need to do any internal dynamic allocation

```
std::variant<int, double, MyStaticallySizedStruct> v;
v = 3.2;
v = MyStaticallySizedStruct(); //* no allocs happen
```

Polymorphism Without Dynamic Allocation

Variant also has a `visit` mechanism

```
std::variant<int, double, MyStaticallySizedStruct> v;
v = 3.2;
v = MyStaticallySizedStruct();
std::visit(Merged{
  [](const double d){ /* do something with double */ },
  [](const int i){ /* do something with int */ },
  [](const MyStaticallySizedstrung &){ /* else */ }
}, v);
```

Polymorphism Without Dynamic Allocation

Variant also has a `visit` mechanism

… which can be a generic lambda

```
std::variant<int, double, MyStaticallySizedStruct> v;
v = 3.2;
v = MyStaticallySizedStruct();
// compile time error if any type fails to support
// cout << <Type>
std::visit([](const auto &o) {
  std::cout << o; //* assuming everything has support
}, v);
```

Polymorphism Without Dynamic Allocation

So as long as every contained type can support the same interface...

```
#include <numeric>
#include <variant>
#include <array>

struct S {
  virtual int get_val() const { return 2; }
};
struct D : S {
  int get_val() const override { return 4; }
};

using variant_t = std::variant<S,D>;

//* returns get_val from whatever is contained
auto get_val(const variant_t &v) {
  return std::visit([](const auto &val){
    return val.get_val();
  }, v);
}
```

Polymorphism Without Dynamic Allocation

```
int main() {
  // Imagine a vector that is added to at runtime
  std::array<variant_t, 2> a { D(), S() };
  auto result =
    std::accumulate(std::begin(a), std::end(a), 0,
      [](const auto previous, const auto &variant) {
        return get_val(variant) + previous;
      }
    );
  return result;
}
```

Polymorphism Without Dynamic Allocation

How well does it optimize though?

(Compiler Explorer Time)

Polymorphism Without Dynamic Allocation

How well does it optimize though?

In the cases where the code is not inlined and folded, we've essentially turned every function call into a virtual function call.

For environments that don't allow or very much limit dynamic allocation this might be an acceptable trade off however.

End

Jason Turner

- Co-host of CppCast http://cppcast.com
- Host of C++ Weekly https://www.youtube.com/c/JasonTurner-lefticus
- Co-creator of ChaiScript http://chaiscript.com
- Curator of http://cppbestpractices.com
- Microsoft MVP for C++ 2015-present
- C++ Trainer - Next Training is in Chicago July 12-14 https://www.eventbrite.com/e/better-c-chicago-registration-34084060342