# Practical C++17

# 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

# Jason Turner

Independent and available for training or contracting

- http://articles.emptycrate.com/idocpp
- http://articles.emptycrate.com/training.html
- Next training: 2 days of best practices @ CppCon 2017

# ChaiScript

- Embedded scripting language co-designed by me specifically for C++
- Supports Visual Studio, clang, GCC
- Runs on Windows, Linux, MacOS, FreeBSD, Haiku
- Currently requires C++14
- Header only - no external deps
- Designed for integration with C++
- All types are the same and directly shared between script and C++ (`double`, `std::string`, `std::function<>`, etc)

# ChaiScript

- My proving ground for testing best practices.
- About 25k lines of C++
- Has evolved from C++03 + Boost -> C++11 -> C++14 and onward to C++17
- Complex template usage for automatic function type deduction

# ChaiScript

Full example:

```
#include <chaiscript/chaiscript.hpp>

std::string greet(const std::string &t_name) {
  return "Hello " + t_name;
}

int main() {
  ChaiScript::chaiscript chai;
  chai.add(chaiscript::fun(&greet), "greet");
  chai.eval(R"(print(greet("Jason")))");
}
```

# About my Talks

- Move to the front!
- Please interrupt and ask questions

# Slack

# C++17 Overview

# Structured Bindings

# Structured Bindings

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

- Can be used to automatically split a structure into multiple variables
- Added in C++17

# if-init expressions

# if-init expressions

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

- Added in C++17
- Also works for `switch` conditions

# Standard Library Changes

# `emplace_back` (C++17)

```
container.emplace_back();
auto &val = container.back();
```

Becomes:

```
auto &val = container.emplace_back();
```

C++17 modified `emplace_back` and `emplace_front` to return references to the objects that were just created for sequence containers.

# `std::string_view` added in C++17

Provides a light weight `std::string`-like view into an array of characters.

It can be constructed from a `const char *` or automatically converted to from an `std::string` or from a pointer and a length.

```
void use_string(std::string_view sv);
```

```
use_string("hello world");
use_string(some_std_string);
```

Intended to be the new language for passing string-like things.

# Nested Namespaces

# Nested Namespaces

C++17 adds the ability to do:

```
namespace Namespace::Nested {
}
```

Which allows for more compact namespace declarations.

# Class Template Type Deduction

# Class Template Type Deduction

C++17 class template type deduction

```
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}; /// template parameters not needed
}
```

# Class Template Deduction Guides

Deduction guides tell the compiler how to deduce a class template from a set of parameters.

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

# Class Template Deduction Guides

Which helps for cases like this:

```
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
}
```

`if constexpr`

# `if constexpr`

A compile-time conditional block

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

# 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 ) ) )
```

# `noexcept` is now part of the type system

# `noexcept` In The Type System

```cpp
void use_func(void (*func)() noexcept);
void my_func();

use_func(&my_func);
```

This will no longer compile in C++17, a new overload is needed:

```cpp
void use_func(void (*func)());
```

`noexcept` function pointer can be converted to non-`noexcept`, but not the other way around.

# Feature Impact

# Which C++ 17 Had The Most Impact on ChaiScript?

What is important for defining impact?

- Compile Time?
- Runtime?
- Code Size?
- Readability?

I tried to review each of these.

# Which C++ 17 Had The Most Impact on ChaiScript?

Vote *https://strawpoll.com/w767k3kk*:

- Class template type deduction / guides?
- if / switch init expressions?
- fold expressions?
- `if constexpr`?
- structured bindings?
- `std::string_view`
- `std::vector::emplace_back`
- nested `namespace`s
- `noexcept` in the type system

# Results Least to Greatest

# `noexcept` In The Type System

# Impact of `noexcept` in the type system

- Caused a 2x duplication of my function type deduction code
- Required conditional compilation with `#if __cpp_noexcept_function_type >= 201510` to get it to compile in C++17 at all.
- This typed support for `noexcept` appears to give ~1% performance boost

# Impact of `noexcept` in the type system

Impact conclusions:

- Performance: Possible Minor Improvement
- Maintainability: Negative: more overloads necessary, conditional code
- Compile Time: None observed
- Readability: None

Score: -0.5 necessary but potentially annoying change.

# if-init Expressions

# if-init Expression Impact

I was excited about the possibility of limiting variable scope:

```
auto val = get_some_val();
if (val > 5) {
  // do something else
}
```

becomes

```
if (auto val = get_some_val(); val > 5) {
  // do something else
}
```

# if-init Expression Impact

But if your code is well modularized and has small functions...

```
bool my_func() {
  auto val = get_some_val();
  if (val > 5) {
    return something(val);
  } else {
    return something_else(val);
  }
}
```

vs

```
bool my_func() {
  if (auto val = get_some_val(); val > 5) {
    return something(val);
  } else {
    return something_else(val);
  }
}
```

# if-init Expression Impact

Then how do you format it for readability?

```
bool my_func() {
  if (auto val = get_some_val(); val > 5) {
    return something(val);
  } else {
    return something_else(val);
  }
}
```

vs

```
bool my_func() {
  if (auto val = get_some_val();
      val > 5) {
    return something(val);
  } else {
    return something_else(val);
  }
}
```

# if-init Expression Impact

Impact conclusions:

- Performance: None
- Maintainability: Small
- Compile Time: None
- Readability: Possibly Hurt

Score: 0.5 disappointing

# Nested Namespaces

# Impact of nested namespaces

```
namespace chaiscript {
  namespace dispatch {
    namespace detail {
      // some thing
    }
    // some other thing
  }
}
```

becomes

```
namespace chaiscript::dispatch {
  namespace detail {
    // some thing
  }
  // some other thing
}
```

# Impact of nested namespaces

I found these changes quite satisfying to implement.

However:

- Causes large whitespace diffs
- Can cause large diffs again if you need to insert something in a middle namespace

# Impact of nested namespaces

Impact conclusions:

- Performance: none
- Maintainability: possibly negative
- Compile time: none
- Readability: large

Score: 1.5 surprisingly gratifying to implement. Might make us rethink some of our coding and formatting standards

# Class Template Type Deduction

# Class Template Type Deduction Impact

C++17 adds a complete complement of deduction guides for standard library types.

What impact did this have on my existing code base?

Almost 0

# Class Template Type Deduction Impact

Class template type deduction doesn't help at the class level

```
struct S
{
  // no practical way to use type deduction here
  std::vector<int> myvec;
};
```

# Class Template Type Deduction Impact

I almost never create local containers, and `auto` picks up the rest.

Which is better (pretend they do more)?

```
void myfunc() {
  /// A
  auto f = [](){ /* something */ };
  f();
}
```

```
void myfunc() {
  /// B
  std::function f = [](){ /* something */ };
  f();
}
```

# Class Template Type Deduction Impact

Who thinks these are good uses of class template type deduction?

```
std::vector some_strings{
  "This is a list"s,
  "of somewhat longish"s,
  "strings to operate on."s
  };
```

```
std::vector some_objs{
    std::make_unique<Obj>(1),
    std::make_unique<Obj>(2),
    std::make_unique<Obj>(3),
  };
```

# Class Template Type Deduction Impact

Impact conclusions:

- Performance: Possible negative if not considering object lifetime
- Maintainability: Small
- Compile Time: None
- Readability: Small

Score: 1.5

# Structured Bindings

# Impact of structured bindings

Good, helps clean up code.

```cpp
void dump_types() const {
  for (const auto &v : get_types()) {
    std::cout << v.first << ": " << v.second->name() << '\n';
  }
}
```

becomes

```cpp
void dump_types() const {
  for (const auto &[name, type] : get_types()) {
    std::cout << name << ": " << type->name() << '\n';
  }
}
```

# Impact of structured bindings

Is there any performance impact?

# Impact of structured bindings

Which is better?

```
std::string func() {
  auto val = get_some_val();
  return val;
}
```

or

```
std::string func() {
  auto val = get_some_val();
  return std::move(val);
}
```

# Impact of structured bindings

Why?

```
std::string func() {
  auto val = get_some_val();
  return val;
}
```

or

```
std::string func() {
  auto val = get_some_val();
  return std::move(val); /// move forced, breaks NRVO
}
```

# Impact of structured bindings

Which is better?

```
std::string func() {
  auto pair = get_some_pair();
  return pair.second;
}
```

or

```
std::string func() {
  auto pair = get_some_pair();
  return std::move(pair.second);
}
```

# Impact of structured bindings

Why?

```
std::string func() {
  auto pair = get_some_pair();
  return pair.second; /// NRVO cannot apply to subobject
}
```

or

```
std::string func() {
  auto pair = get_some_pair();
  return std::move(pair.second);
}
```

# Impact of structured bindings

Which is better?

```
std::string func() {
  auto [succeeded, value] = get_some_pair();
  return value;
}
```

or

```
std::string func() {
  auto [succeeded, value] = get_some_pair();
  return std::move(value);
}
```

# Impact of structured bindings

Why?

```
std::string func() {
  auto [succeeded, value] = get_some_pair();
  return value; /// NRVO cannot apply to sub object
}
```

or

```
std::string func() {
  auto [succeeded, value] = get_some_pair();
  return std::move(value);
}
```

# Impact of structured bindings

This is (effectively) what the compiler is doing for us

```cpp
std::string func() {
  auto e = get_some_pair();
  auto &succeeded = e.first;
  auto &value = e.second;
  return value;
}
```

# Impact of structured bindings

Impact conclusions:

- Performance: potentially very negative. Now you have to consider the source of the variable.
- Maintainability: Minor
- Compile Time: None
- Readability: Large

Score: 2 potentially really helpful, but mess with our understand of object lifetime

# Fold Expressions

# Fold Expression Impact

*Almost* everything that is possible with fold expressions, is possible with initializer_lists

```
(void)std::initializer_list<int>{
  (static_cast<T&>(*this).trace(ds, node), 0)...
};
```

became

```
(static_cast<T&>(*this).trace(ds, node), ... );
```

# Fold Expression Impact

*Everyone agrees both versions have a well defined order, right?*

```
(void)std::initializer_list<int>{
  (static_cast<T&>(*this).trace(ds, node), 0)...
};
```

became

```
(static_cast<T&>(*this).trace(ds, node), ... );
```

# Fold Expression Impact

Used in this context:

```
void do_trace(const Dispatch_State &ds,
              const AST_Node_Impl<Tracer<T...>> *node)
{
  (static_cast<T&>(*this).trace(ds, node), ... );
}
```

# Fold Expression Impact

Impact conclusions:

- Performance: None
- Maintainability: Small
- Compile Time: None
- Readability: Small

Score 2: no negatives, minor help in some cases

# Emplace Back

# Impact of C++17 `emplace_back`

This:

```
Boxed_Value &add_get_object(const std::string &t_name,
                            Boxed_Value obj, Stack_Holder &t_holder)
{
  auto &stack_elem = get_stack_data(t_holder).back();

  if (std::any_of(stack_elem.begin(), stack_elem.end(),
      [&](const std::pair<std::string, Boxed_Value> &o) {
        return o.first == t_name;
      }))
  {
    throw chaiscript::exception::name_conflict_error(t_name);
  }

  stack_elem.emplace_back(t_name, std::move(obj))
  return stack_elem.back().second;
}
```

# Impact of C++17 `emplace_back`

Became:

```
Boxed_Value &add_get_object(const std::string &t_name,
                            Boxed_Value obj, Stack_Holder &t_holder)
{
  auto &stack_elem = get_stack_data(t_holder).back();

  if (std::any_of(stack_elem.begin(), stack_elem.end(),
      [&](const std::pair<std::string, Boxed_Value> &o) {
        return o.first == t_name;
      }))
  {
    throw chaiscript::exception::name_conflict_error(t_name);
  }

  return stack_elem.emplace_back(t_name, std::move(obj)).second;
}
```

# Impact of C++17 `emplace_back`

Impact Conclusions:

- Performance: tiny possibility of performance improvement (I did measure some)
- Maintainability: minor
- Compile time: none
- Readability: minor

Score: 2.5 possible performance, readability, and maintainability helps

# `std::string_view`

# Impact of `std::string_view`

The main impact of using `std::string_view` is in looking up of identifiers and avoiding temporary strings.

But this comes with a complexity cost.

# Impact of `std::string_view`

Cost of `std::string_view`:

What does this `string_view` constructor need to do?

```
std::string_view(const char *);
```

It must calculate the length of the passed in string. Which might happen at compile-time or might not.

# Impact of `std::string_view`

Provided conversions:

- `std::string` automatically converts to `std::string_view` via conversion operator
- `const char *` automatically converts to `std::string` via non-`explicit` constructor
- `const char *` automatically converts to `std::string_view` via non-`explicit` constructor

Provided comparisons:

- `operator<(const std::string &lhs, const std::string &rhs)`
- `operator<(std::string_view lhs, std::string_view rhs)`
- etc

# Impact of `std::string_view`

Is this code OK?

```
std::map<std::string_view, int> map;

void add(const std::string &t_str, int val)
{
  map[t_str] = val;
}
```

# Impact of `std::string_view`

Bad - possible pointer to temporary stored

```
std::map<std::string_view, int> map;

void add(const std::string &t_str, int val)
{
  map[t_str] = val; ///
}
```

# Impact of `std::string_view`

Storing a `map` of `string_view` is very risky, and you won't get any compiler warnings.

# Impact of `std::string_view`

Does this compile?

```
std::map<std::string, int> map;

int get(std::string_view t_sv) {
  return map.at(t_sv);
}
```

# Impact of `std::string_view`

Does this compile? No.

```
std::map<std::string, int> map;

int get(std::string_view t_sv) {
  return map.at(t_sv);  // no conversion to std::string
}
```

# Impact of `std::string_view`

We need to use `is_transparent` comparitor

# Impact of `std::string_view`

`is_transparent` is provided by `std::less<>`

```
std::map<std::string, int, std::less<>> map;

int get(std::string_view t_sv) {


  if (auto itr = map.find(t_sv); itr != map.end()) {
    return *itr;
  }
  throw std::range_error("Key not found");
}
```

# Impact of `std::string_view`

`is_transparent` is provided by `std::less<>`

```
std::map<std::string, int, std::less<>> map;

int get(std::string_view t_sv) {

  /// What happens for this find?
  if (auto itr = map.find(t_sv); itr != map.end()) {
    return *itr;
  }
  throw std::range_error("Key not found");
}
```

# Impact of `std::string_view`

`is_transparent` is provided by `std::less<>`

```
std::map<std::string, int, std::less<>> map;

int get(std::string_view t_sv) {
  /// a temporary string_view is need for each comparison
  /// by using the operator<(string_view, string_view)
  if (auto itr = map.find(t_sv); itr != map.end()) {
    return *itr;
  }
  throw std::range_error("Key not found");
}
```

# Impact of `std::string_view`

`is_transparent` is provided by `std::less<>`

```cpp
std::map<std::string, int, std::less<>> map;

int get(std::string_view t_sv) {

  /// How do we do better?
  if (auto itr = map.find(t_sv); itr != map.end()) {
    return *itr;
  }
  throw std::range_error("Key not found");
}
```

# Impact of `std::string_view`

`std::less<void>` looks something like this:

```
struct less {
  template<typename LHS, typename RHS>
  constexpr bool operator()(const LHS &lhs, const RHS &rhs) const
  {
    return lhs < rhs;
  }
  struct is_tansparent{};
};
```

# Impact of `std::string_view`

My version is:

```
struct str_less {
  constexpr bool operator()(
    const std::string &t_lhs, const std::string &t_rhs) const noexcept
  {
    return t_lhs < t_rhs;
  }

  template<typename LHS, typename RHS>
  constexpr bool operator()(
    const LHS &t_lhs, const RHS &t_rhs) const noexcept
  {
    return std::lexicographical_compare(
      t_lhs.begin(), t_lhs.end(), t_rhs.begin(), t_rhs.end());
  }
  struct is_transparent{};
};
```

# Impact of `std::string_view`

I'm sure this is highly dependent on compiler and optimization settings.

This was G++7.2 for me.

```cpp
/// using my str_less saved 2% performance
/// across ChaiScript
std::map<std::string, int, str_less> map;

int get(std::string_view t_sv) {
  if (auto itr = map.find(t_sv); itr != map.end()) {
    return *itr;
  }
  throw std::range_error("Key not found");
}
```

# Impact of `std::string_view`

Also, don't do this:

```cpp
struct S {
  S(std::string_view sv) : m_string(std::move(sv)) {}
  std::string m_string;
};
```

# Impact of `std::string_view`

Or worse:

```
S make_s(std::string s) {
  return S(std::move(s));
}

struct S {
  S(std::string_view sv) : m_string(std::move(sv)) {}
  std::string m_string;
};
```

# Impact of `std::string_view`

Which can become: (not actually contrived)

```
S make_s(std::string s) {
  return S(std::move(s));
}

struct S {
  S(std::string_view sv) : m_string(std::move(sv)) {}
  std::string m_string;
};

make_s("Hello World");
```

# Impact of `std::string_view`

- Performance: Possibly large either good or bad
- Maintainability: minor: remove_prefix is helpful with parsing
- Compile time: minor
- Readability: minor (you know that it's only a "view")

Score: 3 can help in virtually every way, but be aware of pitfalls

# Impact of `std::string_view`

Notes for deployment:

- Use consistently when you move to it, otherwise expensive conversions back and forth
- If you know you're going to ultimately need a `std::string` use `std::string` through the whole call chain
- Can have huge impact if you have string data that can be `constexpr`
- I'm still seeing buggy / incomplete `constexpr` implementations
- Understand the impact of using `string_view` with lookup for `std::map` / `std::set` keys

`if constexpr`

# if constexpr Impact

This:

```
template<typename Type>
auto do_call_impl(Class *o) const
    -> std::enable_if_t<std::is_pointer_v<Type>, Boxed_Value>
{
  return Handle_Return<Type>
      ::handle(o->*m_attr);
}

template<typename Type>
auto do_call_impl(Class *o) const
    -> std::enable_if_t<!std::is_pointer_v<Type>, Boxed_Value>
{
  return Handle_Return<std::add_lvalue_reference_t<Type>>
      ::handle(o->*m_attr);
}
```

# if constexpr Impact

Can become:

```
template<typename Type>
auto do_call_impl(Class *o) const
{
  if constexpr(std::is_pointer_v<Type>) {
    return detail::Handle_Return<Type>
      ::handle(o->*m_attr);
  } else {
    return detail::Handle_Return<std::add_lvalue_reference_t<Type>>
      ::handle(o->*m_attr);
  }
}
```

# if constexpr Impact

Which eliminated about 1/2 of my SFINAE usage, but why not 100%?

constructors and other mixed templated / typed overloads

# if constexpr Impact

```cpp
Any(const Any &t_any)
  : m_data(t_any.empty() ? nullptr : t_any.m_data->clone())
{ }

template<typename ValueType,
typename = std::enable_if_t<
    !std::is_same_v<Any, std::decay_t<ValueType>>>>
explicit Any(ValueType &&t_value)
  : m_data(std::make_unique<Data_Impl<std::decay_t<ValueType>>>(
                std::forward<ValueType>(t_value)))
{
}
```

# if constexpr Impact

```
template<Typename T>
static std::unique_ptr<Data> make_data(T &&t) {
  if constexpr (std::is_same_v<Any, std::decay_t<T>>) {
    if (!t_any.empty()) { return t_any.m_data->clone(); }
    else { return nullptr; }
  } else {
    return make_unique<Data_Impl<std::decay_t<ValueType>>>(
              std::forward<ValueType>(t_value));
  }
}

template<typename ValueType> explicit Any(ValueType &&t_value)
  : m_data(make_data(std::forward<ValueType>(t_value)))
{ }
```

What problem (if any) does this version have?

# if constexpr Impact

Covering move construction

```
template<Typename T> std::unique_ptr<Data> make_data(T &&t) {
  if constexpr (std::is_same_v<Any, std::decay_t<T>>) {
    if constexpr (std::!is_const_v<decltype(t)>
                  && std::is_rvalue_reference_v<decltype(t)>) {
      return std::move(t.m_data);
    } else {
      if (!t_any.empty()) { return t_any.m_data->clone(); }
      else { return nullptr; }
    }
  } else {
    return make_unique<Data_Impl<std::decay_t<ValueType>>>
                 (std::forward<ValueType>(t_value))
  }
}

template<typename ValueType> explicit Any(ValueType &&t_value)
  : m_data(make_data(std::forward<ValueType>(t_value)))
{ }
```

# if constexpr Impact

Now, I don't like SFINAE, but I'm sticking with it for eliminating options for overload resolution.

# if constexpr Impact

Side note: Boxed_Number

- ChaiScript allows for all operators for all possible arithmetic types
- While maintaining the proper types, following C++ standards
- Example: `float * int` returns `float`, `uint64_t * int` returns `uint64_t`
- This means all possible combinations must be generated at compile time

`if constexpr` enabled a rewrite of this code which removed about 160 LOC and allowed for more linear flow

# if constexpr Impact

Impact conclusions:

- Performance: none
- Maintainability: large
- Compile Time: minor (for my use cases)
- Readability: large

Score: 5 generally very helpful, no real caveats.

# Class Template Deduction Guides

# Class Template Deduction Guide Impact

While class template type deduction specifically had almost no impact, deduction *guides* had a significant impact

# Class Template Deduction Guide Impact

This:

```
template<typename Ret, typename ... Param>
Function fun(Ret (*func)(Param...)) {
  auto fun_call = Fun_Caller<Ret, Param...>(func);
  return Function(make_shared<Function_Callable_Impl<
      Ret (Param...), decltype(fun_call)>>(fun_call));
}

template<typename Ret, typename Class, typename ... Param>
Function fun(Ret (Class::*t_func)(Param...) const) {
  auto call = Const_Caller<Ret, Class, Param...>(t_func);
  return Function(make_shared<Function_Callable_Impl<
      Ret (const Class &, Param...), decltype(call)>>(call));
}
```

Plus `noexcept`, non-`const` etc

# Class Template Deduction Guide Impact

Is replaced with a series of deduction guides:

```
template<typename Ret, typename Class, typename ... Param>
Signature(Ret (Class::*f)(Param ...) volatile) ->
  Signature<Ret, Params<volatile Class &, Param...>, false, true>;

template<typename Ret, typename Class, typename ... Param>
Signature(Ret (Class::*f)(Param ...) volatile noexcept) ->
  Signature<Ret, Params<volatile Class &, Param...>, true, true>;

template<typename Ret, typename Class, typename ... Param>
Signature(Ret (Class::*f)(Param ...) volatile const) ->
  Signature<Ret, Params<volatile const Class &, Param...>, false, true>;

template<typename Ret, typename Class, typename ... Param>
Signature(Ret (Class::*f)(Param ...) volatile const noexcept) ->
  Signature<Ret, Params<volatile const Class &, Param...>, true, true>;
```

Plus one simple function

# Class Template Deduction Guide Impact

Net result:

- no change in LOC
- 8x better support for C++ functions (`&&`, `&`, `volatile`)
- eliminated mostly redundant code.

# Class Template Deduction Guide Impact

Impact conclusions:

- Performance: Minor
- Maintainability: Large
- Compile Time: Minor
- Readability: Large

Score: 6 possibility of large help, including performance increases with rewritten code

# Conclusions

Copyright Jason Turner @lefticus

# Conclusions

- There are many `constexpr` changes which we did not address
- The little details have the biggest impact
- `[[maybe_unused]]` gets an honorable mention for removing code like `(void)variable`
- Being able to do `std::decay_t<Type>` and `std::is_trivially_destructible_v<Type>` instead of `typename std::decay<Type>::type` and `std::is_trivially_destructible<Type>::value` goes a long way to making templated code more readable

# 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

# Jason Turner

Independent and available for training or contracting

- http://articles.emptycrate.com/idocpp
- http://articles.emptycrate.com/training.html
- Next training: 2 days of best practices @ CppCon 2017