SAVE TIME, SPACE AND A LITTLE SANITY WITH STD::FUNCTION_REF.

INTRODUCTION

o(✿￣▽￣)ブ

EMPLOYER

I am here on behalf of myself, and not my employer.

IMPLEMENTATIONS

- Standard libraries do not yet have avaliable implementations.
- Third party implementations will be used instead.

INFORMATION

- Code found here
- Working Draft
- P0792R1
- P0792R11
- Experiance.

WHAT I WILL COVER

- std::function_ref and background concepts.
- How to use std::function_ref.
- How not to use std::function_ref.
- How std::function_ref works.
- The cost of unerased type with example.
- Benchmarks of compile time and code size and runtime, now with graphs!

WHAT IS `std::function_ref`?

o(✿￣▽￣)o

WHAT IS `std::function_ref`?

a type-erased callable reference

— Vittorio Romeo, Zhihao Yuan, and Jarrad Waterloo; P0792R11

---

a non-owning reference to a Callable

— Vittorio Romeo; P0792R1

IT’S A FUNCTION WRAPPER!

- Create it with a callable of any type, and
- call it as a `std::function_ref`.

```
std::function_ref<int(int, int)> wrap = +[](int a, int b){ return a + b;
```

`std::function`

- From c++11.
- Owning.
- Allocates.
- Requires RTTI/typeid.
- Callables must be copyable.

`std::move_only_function`

- From c++23.
- Owning.
- Allocates.
- Cannot be copied.
- Functions can have qualifiers.

`std::copyable_function`

- From c++26.
- Owning.
- Allocates.
- Functions can have qualifiers.
- Callables must be copyable.

`std::function_ref`

- From c++26.
- Non-owning.
- Functions can have qualifiers.

REFERENCE TABLE

	`std::function`	`std::copyable_function`	`std::move_only_function`	`std::function_ref`
Owning	✔	✔	✔	✘
Special Member Functions	Copy & Move	Copy & Move	Move	Trivial Copy & Move
RTTI not needed	✘	✔	✔	✔
Express `noexcept` functions	✘	✔	✔	✔
Express `const` functions	✘	✔	✔	✔
Express ref-qualified functions	✘	✔	✔	✘
Supports move-only types	✘	✘	✔	✔

IT IS TWO NEW TYPES AND AN INLINE VARIABLE

<utility>:

```
template <auto V>
struct nontype_t { explicit nontype_t() = default; };

template <auto V>
inline constexpr nontype_t<V> nontype{};
```

<functional>:

```
template<class...>
class function_ref;

template<class R, class... ArgTypes>
class function_ref<R(ArgTypes...) cv noexcept(noex)>
```

TRIVIALLY COPYABLE

- With a size of two pointers.
- This has advantages for many ABIs, producing smaller and faster code!

BACKGROUND

o(✿￣﹀￣)o ┳┳

WHAT IS A CALLABLE?

- If it can be called with `std::invoke`, then it is a callable.
- 140 of 162 overload sets from `<algorithm>` had one or more callables.

CALLABLE: FUNCTION POINTERS/REFERENCES

```cpp
int add(int a, int b) { return a + b; }

int main() {
  int (*add_fn_ptr)(int, int) = &add;
  std::cout << std::invoke(add_fn_ptr, 1, 2) << "\n";

  int (&add_function_ref)(int, int) = add;
  std::cout << std::invoke(add_function_ref, 1, 2) << "\n";
}
```

CALLABLE: OVERLOADED CALL OPERATOR

```
struct OverloadedOperator {
  auto operator()(int a, int b) const -> int { return a +
};
int main() {
  auto const callable_object = OverloadedOperator{};
  std::cout << std::invoke(callable_object, 1, 2) << "\n"
}
```

- Lambdas,
- STL's `std::less`, `std::greater`, `std::less_equal`, `std::greater_equal` etc…
- STL's `std::plus`, `std::minus`, `std::multiplies`, `std::divides`, etc…
- and more…

CALLABLE: LAMBDAS ARE SYNTACTIC SUGAR

```
auto const lambda = [](int a, int b) {
  return a + b;
};

int do_thing() {
  return std::invoke(lambda, 1, 2);
}
```

```
struct lambda1 {
  constexpr int operator()(int a, int b) const {
    return a + b;
  }

  constexpr operator decltype(&impl) () const noexcept
    return &impl;
  }
private:
  static constexpr int impl(int a, int b) {
    return lambda1{}.operator()(a, b);
  }
};

auto const lambda = lambda1{};

int do_thing() {
  return std::invoke(lambda, 1, 2);
}
```

CALLABLE: MEMBER FUNCTION POINTER

```
struct Add {
  auto add(int a, int b) const -> int {
    return a + b;
  }
};

int main() {
  Add add;
  std::cout << std::invoke(&Add::add, add, 1, 2) << "\n";
}
```

CALLABLE: DATA MEMBER POINTER

```
struct Foo {
  int value = {};
};

int main() {
  Foo add{87};
  std::cout << std::invoke(&Foo::value, add) << "\n";
}
```

OWNERSHIP

- `std::function_ref` is non-owning.
- It does not maintain a copy of the callable.
- It maintains only a reference to the callable.

WHAT IS AN OWNERSHIP?

- Owning examples: `std::unique_ptr`, `std::string`, and `std::vector`.
- Non-owning examples: `std::span`, `std::string_view`, and `std::weak_ptr`.

HOW TO USE

(✿◡‿◡) ┳┳

WITH FREE FUNCTIONS

```cpp
#include <functional>

int add(int a, int b) { return a + b; }

int main() {
    std::function_ref<int(int, int)> fn_from_pointer{ &add };
    std::function_ref<int(int, int)> from_nontype{ std::nontype<&add> };
}
```

WITH LAMBDAS/CALLABLE OBJECTS

```cpp
#include <functional>

auto const add = [](int a, int b){ return a + b; };

int main() {
    std::function_ref<int(int, int)> fn{ add };
}
```

WITH MEMBER FUNCTIONS POINTERS

```cpp
#include <functional>

struct Add {
    int Do(int a, int b) const {
        return a + b;
    }
};

int main() {
    Add add;
    std::function_ref<int(int, int)> fn1{ std::nontype<&Add::Do>, add };
    std::cout << fn1(1, 2) << "\n";

    std::function_ref<int(Add const &, int, int)> fn2{ std::nontype<&Add::Do> };
    std::cout << fn2(add, 1, 2) << "\n";
}
```

WITH DATA MEMBERS POINTERS

```cpp
#include <functional>

struct Add {
    int a = 0;
    int b = 0;
    int c = a + b;
};

int main() {
    Add add{1, 2};
    std::function_ref<int()> fn1{ std::nontype<&Add::c>, add };
    std::cout << fn1() << "\n";

    std::function_ref<int(Add const &)> fn2{ std::nontype<&Add::c> };
    std::cout << fn2(add) << "\n";
}
```

CALLING `std::function_ref`

```cpp
#include <functional>

void run_function(std::function_ref<int(int, int)> fn) {
    auto const result1 = fn(10, 24);
    auto const result2 = std::invoke(fn, 10, 24);
}
```

HOW NOT TO USE

(✿ᒧ‿^‿)ᒧ‿┻┻

IT CANNOT ALWAYS BE USED.

- We cannot use `std::function_ref` when the callable is destroyed before the `std::function_ref`.
- We cannot use `std::function_ref` when we need copies of our callable.

BE AWARE OF LIFETIMES

```
auto getPlus(int a) -> std::function_ref<int(int)> {
    auto plus = [a](int b){ return a + b; };
    return plus;
}

int main() {
    auto const fn = getPlus();
}
```

BE AWARE OF LIFETIMES

```
auto getPlus(int a) -> std::function_ref<int(int)> {
    auto plus = [a](int b){ return a + b; };
    return plus;
}

int main() {
    auto const fn = getPlus();
}
```

Calling `fn` is undefined behaviour.

IS THERE A BUG HERE?

```
auto foo(std::function_ref<int(int)> fn) -> std::function_ref<int(int)> {
    return fn;
}
```

WHAT ABOUT HERE?

```
int main() {
    auto const fn_ref = bar([accumulator = int(0)](int value) mutable {
        accumulator += value;
        return accumulator;
    });
    std::cout << fn_ref(1) << "\n";
}
```

BE CAREFUL DESIGNING FUNCTIONS USING `std::function_ref`

```cpp
auto identity(std::function_ref<int(int)> fn) -> std::function_ref<int(int)> {
    return fn;
}

int main() {
    auto const fn_ref = identity([accumulator = int(0)](int value){
        accumulator += value;
        return accumulator;
    });

    std::cout << fn_ref(1) << "\n";
}
```

- If you reviewed this, the PR may not contain the function `identity`. Would you always notice this error?

BE CAREFUL DESIGNING FUNCTIONS USING `std::function_ref`

```
auto identity(std::function_ref<int(int)> fn) -> std::function_ref<int(int)> {
    return fn;
}

int main() {
    auto const fn_ref = identity([accumulator = int(0)](int value){
        accumulator += value;
        return accumulator;
    });

    std::cout << fn_ref(1) << "\n";
}
```

- It is not kind is making the natural usage an error.
- Make it easy to write correct code and don’t write future bugs.

HOW IT WORKS

(✿◠‿◠) ┻┻

A CALLABLE CAN BE REFERENCED BY ONE OF TWO THINGS

- An object.
- A function pointer.

BOUND ENTITY

The bound-entity, an object that stores a pointer to the object or referenced free-function:

```
union BoundEntity {
  void cv * ptr;
  void (*fn)();
};
BoundEntity m_bound_entity = {};
```

BOUND ENTITY DATA

Bound entity stores either a pointer to the function, or a pointer to the underlying object:

```
union BoundEntity {
  void cv * ptr;
  void (*fn)();
};
BoundEntity m_bound_entity = {};
```

THUNK

The thunk, a function pointer that knows how to call the referenced callable:

```
union BoundEntity {
  void const * ptr;
  void (*fn)();
};
BoundEntity m_bound_entity = {};
Ret (*m_thunk)(BoundEntity bound_entity, Args...);
```

CALL OPERATOR

Calling `std::function_ref` means calling the thunk with the bound entity and provided arguments:

```
constexpr auto operator()(Args...args) const -> Ret {
  return m_thunk(m_bound_entity, std::forward<Args>(args)...);
}
```

CONSTRUCT FROM FREE FUNCTION

We can easily construct from a free function:

```
template <typename Fn> requires (std::is_function_v<Fn>)
function_ref(Fn * fn) noexcept
  : m_bound_entity{ .fn = reinterpret_cast<void (*)()>(fn) }
  , m_thunk(+[](BoundEntity bound_entity, Args... args) -> Ret {
    return std::invoke_r<Ret>(reinterpret_cast<Fn *>(bound_entity.fn), args...);
  })
{}
```

CONSTRUCT FROM A CALLABLE OBJECT

Or, we can store a pointer to the callable object, and cast back to that object type in our thunk:

```
template <typename Obj>
constexpr function_ref(Obj && obj) noexcept
  : m_bound_entity{ .ptr = std::addressof(obj) }
  , m_thunk(+[](BoundEntity bound_entity, Args... args) -> Ret {
    return std::invoke_r<Ret>(*static_cast<Obj cv *>(bound_entity.ptr), args...);
  })
{}
```

CONSTRUCT FROM A NTTP AND THE FIRST CALLABLE ARGUMENT

Or, we can store a member function pointer:

```
template <auto V, typename Obj>
constexpr function_ref(nontype_t<V>, Obj && obj) noexcept
  : m_bound_entity{ .ptr = std::addressof(obj) }
  , m_thunk(+[](BoundEntity bound_entity, Args... args) -> Ret {
    return std::invoke_r<Ret>(V, *static_cast<Obj cv*>(bound_entity.ptr), std::forward<Args>(args)...);
  })
{}
```

CONSTRUCT FROM A NTTP

Or, we can store a pointer to the callable object, and cast back to that object type in our thunk:

```
template <auto V>
constexpr function_ref(nontype_t<V>) noexcept
  : m_bound_entity{ .ptr = nullptr }
  , m_thunk(+[](BoundEntity, Args... args) -> Ret {
    return std::invoke_r<Ret>(V, std::forward<Args>(args)...);
  })
{}
```

OMITTED SOME OF THE FIDDLY DETAILS…

That’s it!

THE COST OF TYPES

(✿>﹏<) ┻━┻

MORE TYPES

If we have a function, with two template arguments a and b:

```
template <typename A, typename B>
void print_csv_row(A a, B b);
```

MORE TYPES

With only two types, O and N, there are 4 types:

```
template <>
void print_csv_row<O, O>(O a, O b);
template <>
void print_csv_row<O, N>(O a, N b);
template <>
void print_csv_row<N, O>(N a, O b);
template <>
void print_csv_row<N, N>(N a, N b);
```

AND MORE DEFINITIONS…

And 4 different function definitions…

```
template <>
void print_csv_row<O, O>(O a, O b) {
  std::cout << a << ", " << b << "\n";
}
```

FUNCTION DEFINITION CODE SPACE

Each combination has a its own different definition:

signature	Size (Bytes)
`void print_csv_row<int, int>(int, int)`	???
`void print_csv_row<char, int>(char, int)`	???
`void print_csv_row<int, char>(int, char)`	???
`void print_csv_row<char, char>(char, char)`	???

Godbolt

FUNCTION DEFINITION CODE SPACE

Each combination has a its own different definition:

signature	Size (Bytes)
`void print_csv_row<int, int>(int, int)`	96
`void print_csv_row<char, int>(char, int)`	144
`void print_csv_row<int, char>(int, char)`	144
`void print_csv_row<char, char>(char, char)`	192

Godbolt

LAMBDAS ARE SPECIAL

They have a unique type, per lambda:

```
auto const a = [](int x, int y){ return x + y; };
auto const b = [](int x, int y){ return x + y; };
```

- a and b have different types.

THERE ARE TWO DIFFERENT foo DEFINITION!

```
void foo(auto a, auto b) {
  a();
  b();
}

int main() {
  foo( []{ std::cout << "hello, "; }, []{ std::cout << "world!\n"; } );
  foo( []{ std::cout << "hello, "; }, []{ std::cout << "world!\n"; } );
}
```

NOW THERE IS ONE foo DEFINITION!

```
void foo(auto a, auto b) {
  a();
  b();
}

int main() {
  auto const a = []{ std::cout << "hello, "; };
  auto const b = []{ std::cout << "world!\n"; };
  foo( a, b );
  foo( a, b );
}
```

NOW THERE IS ONE foo DEFINITION!

```
void foo(std::function_ref<void()> a, std::function_ref<void()> b) {
  a();
  b();
}

int main() {
  foo( []{ std::cout << "one, "; }, []{ std::cout << "two,\n"; } );
  foo( []{ std::cout << "three, "; }, []{ std::cout << "four!\n"; } );
}
```

CODE SIZE BREAKDOWN EXAMPLE

(✿'◠‿◠) ┻━┻ (◠‿◠▲)

LETS BUILD A THING!

The goal will be to:

- Break down the code size cost of various function wrappers, and
- Create a reusable library that can be used for compile time benchmarking.

CALCULATOR

- Arithmetic operations,
- Only integers.
- Function call syntax.

```
add(1, 2)
sub(2,1)
mul(3, 5 )
div( 50, 5)
mod( 5 , 4 )
```

CODE: PARSABLE

A `Parsable` is an object that has the `parse` function, it takes and returns a `std::string_view`:

```
template <typename T>
concept Parsable = requires(T const & object, std::string_view input) {
  { object.parse(input) } -> std::same_as<std::string_view>;
};
```

CODE: HELPER FUNCTIONS

Identify a string with a value as a `valid_string`:

```
constexpr bool valid_string(std::string_view input) {
  return input.data() != nullptr;
}
```

Do less then operation on the size of two `std::string_view`:

```
constexpr bool less_size(std::string_view lhs, std::string_view rhs) noexcept {
  return std::tuple{valid_string(lhs), lhs.size()} < std::tuple{valid_string(rhs), rhs.size()};
}
```

Get the result of parsing an input with several `Parsables`:

```
constexpr auto parse_all(std::string_view input, Parsable auto const & ... parsables) noexcept {
  return std::array<std::string_view, sizeof...(parsables)>{
    parsables.parse(input)
    ....
  };
}
```

CODE: PREFIX

Eat a given prefix from a string:

```
struct Prefix {
  std::string_view prefix;

  auto parse(std::string_view input) const -> std::string_view {
    if (!input.starts_with(prefix)) { return {}; }
    input.remove_prefix(prefix.size());
    return input;
  }
};
```

CODE: WHITESPACE

Eat any leading whitespace from a string:

```
struct Whitespaces {
  auto parse(std::string_view input) const -> std::string_view {
    size_t i = 0;
    for (; i < input.size(); ++i) {
      if (!IsSpace(input[i])) { break; }
    }
    return std::string_view{ input.begin() + i, input.end() };
  }
};
```

CODE: SEQ

Eat a series of other tokens:

```
template <Parsable ... Ts>
struct Seq {
  std::tuple<Ts...> elements;

  auto parse(std::string_view input) const -> std::string_view {
    auto const parse_one = [&](auto const & arg) {
      return valid_string(input = args.parse(input));
    };
    auto const parse_all = [&](Ts const & ... args) {
      return (parse_one(args) && ...)
        ? input : std::string_view{};
    };
    return apply(parse_all, elements);
  }
};
```

CODE: COMMA

Eat a comma a whitespaced comma:

```
struct Comma
{
  constexpr auto parse(std::string_view input) const -> std::string_view{
    return Seq{Whitespaces{}, Prefix{","}, Whitespaces{}}.parse(input);
  }
};
```

CODE: ANY

Greedily parse several `Parsable`:

```
template <Parsable ... Ts>
  requires (sizeof...(Ts) >= 1)
struct Any
{
  std::tuple<Ts...> elements;

  constexpr auto parse(std::string_view input) const -> std::string_view
  {
    return apply(
      [input](Ts const & ... args) -> std::string_view
      {
        const auto results = parse_all(input, args...)
        return *std::min_element(results.begin(), results.end(), less_size);
      },
      elements
    );
  }
};
```

CODE: IDENTIFYING OUR VALUE TYPES

Concept for any integer:

```
template <typename T>
concept AnyInt = AnyOf<
  T,
  unsigned char,      signed char,
  unsigned short,     signed short,
  unsigned int,       signed int,
  unsigned long int,  signed long int,
  unsigned long long, signed long long
>;
```

CODE: PARSING INTEGERS

Parsing integers:

```
template <typename T, CallableWith<T> F>
struct ValueParser;
template <AnyInt T, CallableWith<T> F>
struct ValueParser<T, F>
{
  F callable;
  int base = 10;

  constexpr auto parse(std::string_view input) const -> std::string_view
  {
    T value = {};
    auto const [ptr, ec] = std::from_chars(input.begin(), input.end(), value, base);
    if (ec != std::errc()) { return {}; }
    std::invoke(callable, value);
    return std::string_view{ ptr, input.end() };
  }
};
```

CODE: COPY ASSIGN CALLABLE

Copy assign a value on invokation:

```
template <typename T>
struct CopyAssigner
{
  T & value;

  constexpr void operator()(T const & result) const {
    value = result;
  }
};
```

CODE: PARSE TUPLE

Store each parsed element of a tuple, each seperated by a comma:

```
template <typename ... Args>
struct TupleParser<std::tuple<Args...>> {
  std::tuple<Args...> & args{};

  constexpr auto parse(std::string_view input) const -> std::string_view {
    auto const parse_arg = [&](size_t arg_index, auto & arg) {
      auto const arg_callable = CopyAssigner{ arg };
      input = ValueParser{ arg_callable }.parse(input);
      if (!valid_string(input)) { return false; }
      if ((arg_index + 1) != sizeof...(Args)) {
        input = Comma{}.parse(input);
      }
      return valid_string(input);
    };
    auto const handle_args = [&](auto & ... args) {
      auto i = size_t{0};
      return (parse_arg(i++, args) && ...);
    };
    if (!std::apply(handle_args, args)) { return {}; }
    return Whitespaces{}.parse(input);
  }
};
```

CODE: PARSE FUNCTION ARGUMENTS

Parse the arguments of a function and run a callback if the parse is successful:

```
template <typename F, typename Ret, typename ... Args>
struct ArgParser<F, Ret, std::tuple<Args...>> {
  F callback;
  Ret return_value;

  constexpr auto parse(std::string_view input) const -> std::string_view {
    std::tuple<Args...> args{};
    input = Seq{
      Prefix{"("}, Whitespaces{},
      TupleParser<std::tuple<Args...>>{args},
      Whitespaces{}, Prefix{")"}
    }.parse(input);
    if (!valid_string(input)) { return {}; }
    return_value(std::apply(callback, args));
    return input;
  }
};
```

CODE: PARSE A FUNCTION CALL

Parse a function's arguments, call a callback with those arguments, provide the return value via a callback to the caller:

```
template <typename F, typename R>
struct CallableParser {
  using return_type = typename FnTraits<F>::ret;
  using args_type = typename FnTraits<F>::args;

  std::string_view name;
  F callable;
  R ret;

  constexpr auto parse(std::string_view input) const -> std::string_view {
    return Seq{ Prefix{name}, Whitespaces{}, ArgParser<F, R, args_type>{callable, ret} }.parse(input);
  }
};
```

CODE: OPERATIONS

Each operations will be defined as follows:

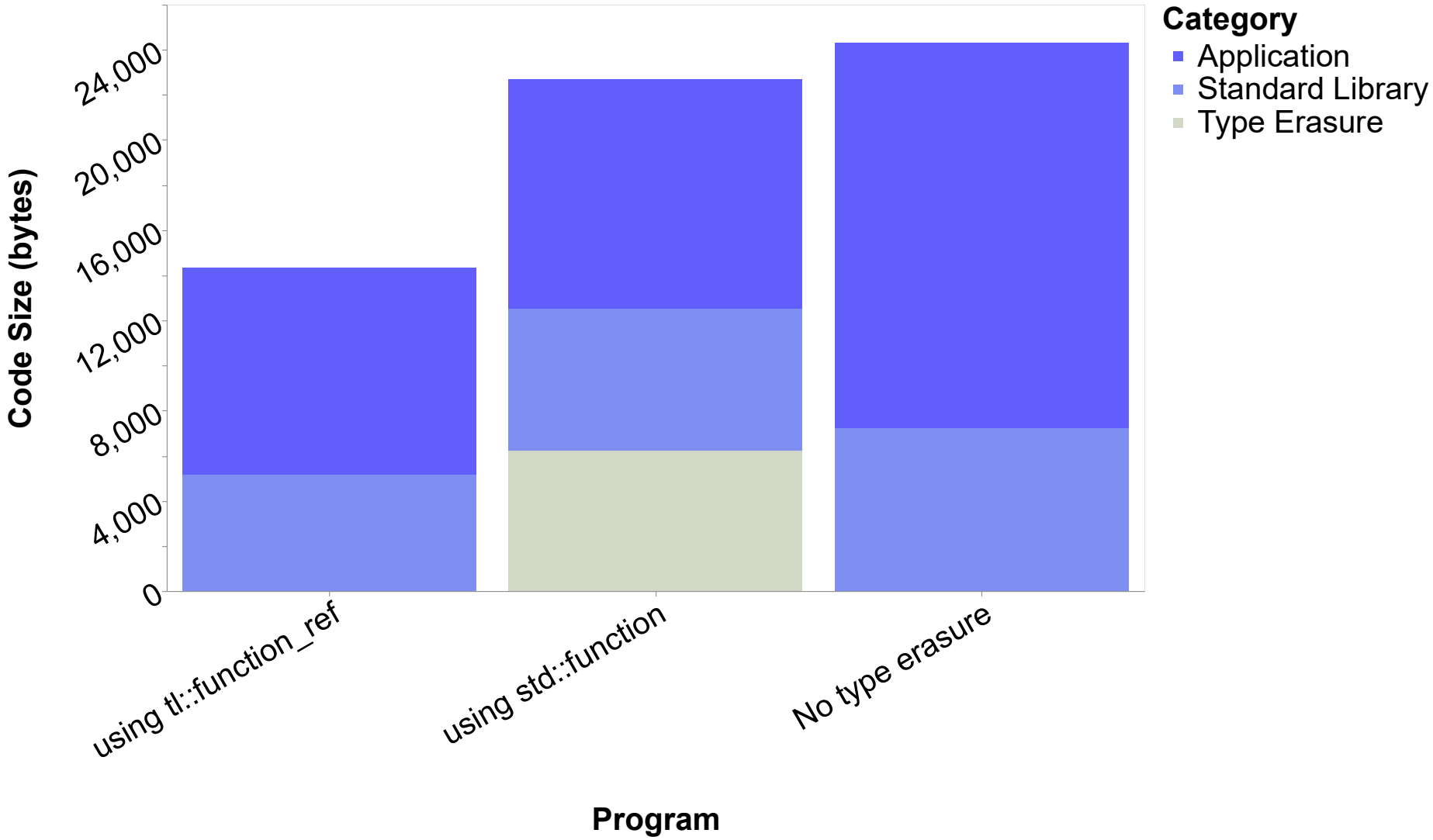
```
auto const add = CallableParser{
  "add",
  FN([](int a, int b){ return a + b; }),
  FN([](int ret){ std::cout << "add: " << ret << "\n"; })
};
```

ENTRY POINT

Run several different parsers in order to include them in the program:

```
auto const any_parser = Any{ std::tuple{ add, sub, mul, div, mod } };
any_parser.parse("add(1, 2)");
any_parser.parse("sub(2, 3)");
any_parser.parse("mul(45, 2)");
any_parser.parse("div(123,3)");
any_parser.parse("mod(23, 3)");
```

CALCULATOR: CODE SIZE



BENCHMARKING (GRAPHS ON SLIDES!)

(✿◠‿◠) ☆┳┳◠(◡‿◡▲)

BENEFITS OF TYPE ERASURE

- Type erasure can have benefits for:
  - Code size,
  - Compile time, and
  - Runtime
- Benchmarking will be done by extending the calculator example from before.

BENCHMARKING SOURCE GENERATION

- Randomly generated source files, each representing a complete application.
- We will generate source files, each with:
- 100, 200, 400, 600, 800, 1000, 1200, 1400, 1600, 1800, 2000, 2200, 2400, 2600, 2800 and 3000 random callables.

RANDOM SOURCE GENERATION

- Random callables will be generated with each callable having:
  - a random number of arguments,
  - a random callable type (lambda, classes or free functions).
  - evaluating a random expression composed of infix expressions (+, -, and * etc.…).

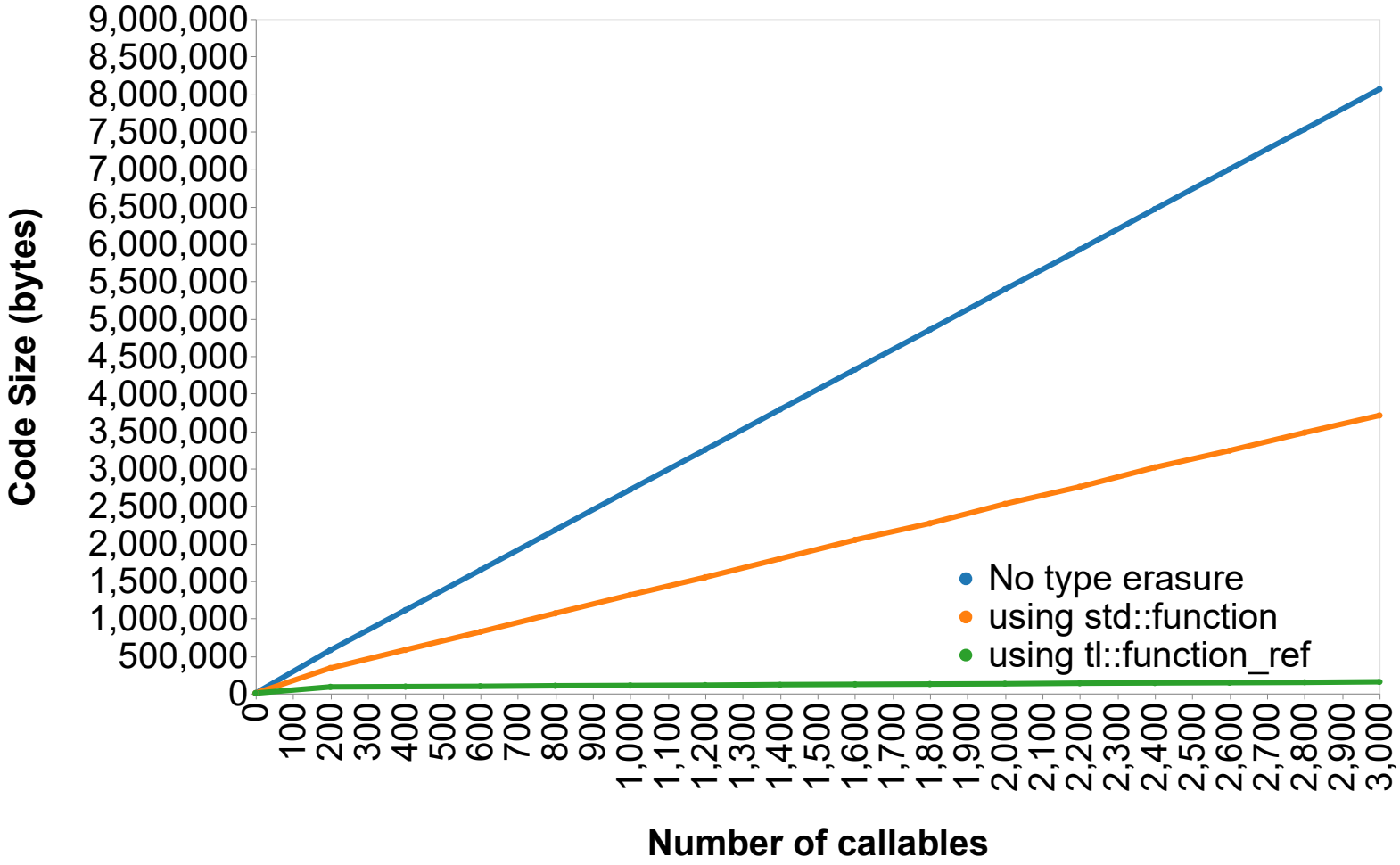
BENCHMARKING CODE SIZE

- The size of the code will be calculated based on relevant sections from the the output of `nm`.

```
nm -C -t d --size-sort --print-size <executable_path> >> symbols.txt
```

  - This was done for every generated source file.

CODE SIZE VS CALLABLES



COMPILE TIME

- Compile time will be measured by modifying CMake's `RULE_LAUNCH_COMPILE` target property.
- This allows the script to time our compilation.

This does something to the effect of:

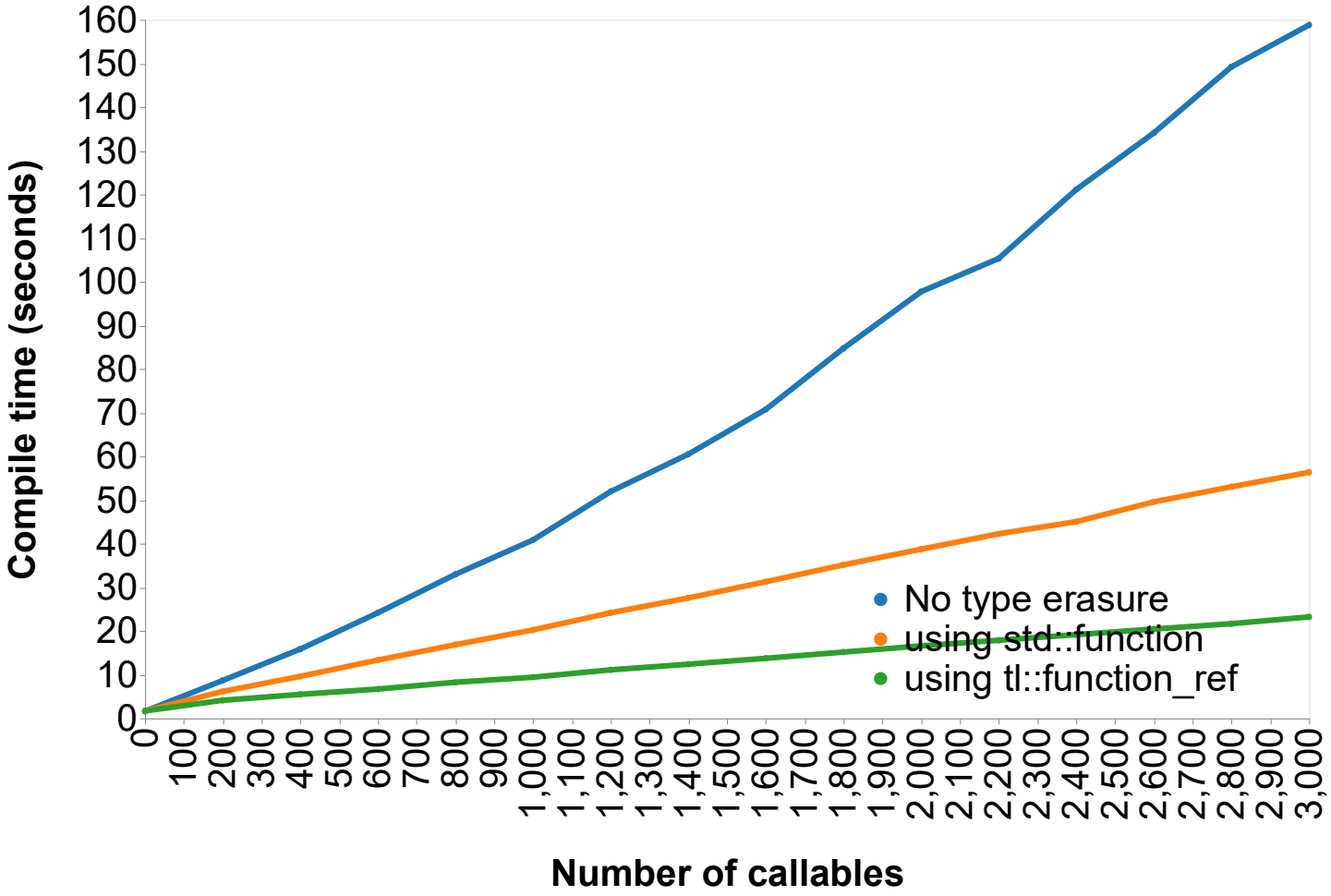
```
gcc args...
```

as:

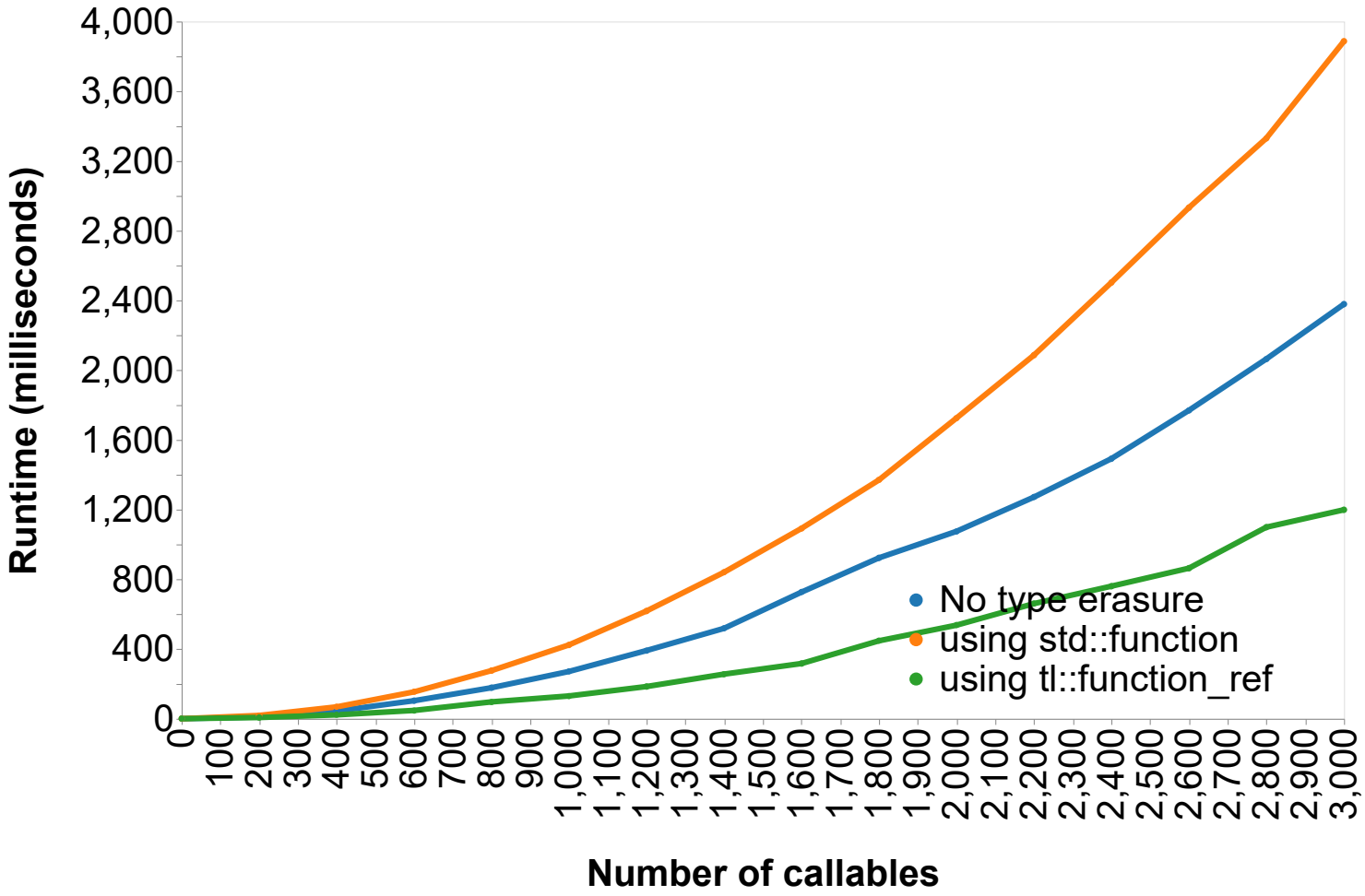
```
time gcc args...
```

Producing a timing for each invokation of the compiler.

COMPILE TIME VS CALLABLES



RUNTIME VS CALLABLES



CONCLUSION

- Try out `std::function_ref`, you may find your application improves in size and compile time.
- By replacing some templates, you can improve the expressiveness of your interfaces.

SLIDES AND CODE

Slides: https://seppeon.gitlab.io/cpptalks/function_ref

AVALIABLE IMPLEMENTATIONS

- Sy Byand's: https://github.com/TartanLlama/function_ref
- Vittorio Romeo's: https://github.com/vittorioromeo/Experiments/blob/master/function_ref.cpp
- Zhihao Yuan's: https://github.com/zhihaoy/nontype_functional

RELATED TALKS

- C++ Type Erasure Demystified by Fedor Pikus; Tommorow (Wednesday July 24, 2024 13:00 - 14:00 EDT); Track C.