What is a static function but isn’t a static function?

Anđelković Miloš

PSA

[Movie Guide - Countdown to Christmas 2024]

[Movie Guide - Countdown to Christmas 2023]

[Movie Guide - Countdown to Christmas 2022]

[Movie Guide - Countdown to Christmas 2021]

[Movie Guide - Countdown to Christmas 2020]

About me

- Started of as a teaching assistant
- Working as a software developer/engineer, programmer whatever for some time now
- Hope to get back to teaching

How much fun can you have in C++?

```
fun
```

voi


ruinedchildhood
Bland, boring Jessica.
ruinedchildhood
If she was a spice, she'd be flour.

```
template <typename T>
void fun() {}
```

```
template <typename … T>
void fun() {}
```

Infinite*

* Terms and conditions may apply

```
void fun(int) {}
```

BRO SCIENCE

IT'S SCIENCE, BRO.


DID SOMEONE SAY
PARTY???
imgflip.com

21:00

How does the compiler know which fun to go to?

- Unqualified name lookup
- Qualified name lookup
- Argument dependant lookup
- Overload resolution
- Viable candidates
- Best candidate

ABRIGED VERSION

Back To Basics: Overload Resolution - CppCon 2021

Back to Basics - Name Lookup and Overload Resolution in C++ - Mateusz Pusz - CppCon 2022

Overload resolution - cppreference.com

```cpp
#include <print>

void fun() { std::print("void fun()\n"); }

void fun(int) {
    std::print("void fun(int)\n"); }

void fun(int &) {
    std::print("void fun(int &)\n"); }

void fun(int &&) {
    std::print("void fun(int &&)\n"); }

void fun(int const &) {
    std::print("void fun(int const &)\n"); }

void fun(int const &&) {
    std::print("void fun(int const &&)\n"); }
```

```cpp
int main()
{
    int a = 0;
    int const b = 0;
    int & c = a;

    fun();
    fun(a);
    fun(b);
    fun(std::move(a));
    fun(std::move(b));
    fun(c);
    fun(std::move(c));
}
```

https://godbolt.org/z/Ezffrz93r

```cpp
#include <print>

void fun() { std::print("void fun()\n"); }

//void fun(int) {
//    std::print("void fun(int)\n"); }

void fun(int &) {
     std::print("void fun(int &)\n"); }

void fun(int &&) {
     std::print("void fun(int &&)\n"); }

void fun(int const &) {
     std::print("void fun(int const &)\n"); }

void fun(int const &&) {
     std::print("void fun(int const &&)\n"); }
```

```cpp
int main()
{
     int a = 0;
     int const b = 0;
     int & c = a;

     fun();
     fun(a);
     fun(b);
     fun(std::move(a));
     fun(std::move(b));
     fun(c);
     fun(std::move(c));
}
```

https://godbolt.org/z/Ezffrz93r

clang

```
fun():
fun(int):
fun(int&):
fun(int&&):
fun(int const&):
fun(int const&&):
```

msvc

```
void fun(void) PROC
void fun(void) ENDP
void fun(int) PROC
void fun(int &) PROC
void fun(int &&) PROC
void fun(int const &) PROC
void fun(int const &&) PROC
```

```
namespace club
{
    void fun() { std::print("void club::fun()\n"); }
    struct party
    {
        void fun() {
          std::print("void club::party::fun()\n"); }
    };
    void fun(party) { std::print("void club::fun(party)\n"); }
}

void fun() { std::print("void fun()\n"); }
```

```
int main()
{
    fun();
    fun(club::party{});
    club::fun();
    club::party p{};
    p.fun();
}
```

https://godbolt.org/z/Y1MqWxTM1

clang

```
club::fun():
club::party::fun():
club::fun(club::party):
fun():
```

msvc

```
void club::fun(void) PROC
void club::party::fun(void) PROC
void club::fun(club::party) PROC
void fun(void) PROC
```

Let’s play the fun game

Rules of the game:

- `void fun() {}` is given
- No changing the name of the function
- No templates
- No adding of arguments
- No changing the return type
- Anything else is free game as long as the code compiles
- How much `fun()` can we have?

```
struct party
{
    void fun() {}
};
```

```
void fun() {}
```

```
void fun() & {}
void fun() && {}
void fun() const & {}
void fun() const && {}
void fun() volatile & {}
void fun() volatile && {}
void fun() const volatile & {}
void fun() const volatile && {}
```

```
int main()
{
    party plain_party{};
    party const const_party{};
    party volatile volatile_party{};
    party const volatile const_volatile_party{};

    plain_party.fun();
    std::move(plain_party).fun();

    const_party.fun();
    std::move(const_party).fun();

    volatile_party.fun();
    std::move(volatile_party).fun();

    const_volatile_party.fun();
    std::move(const_volatile_party).fun();
}
```

You can have 8 fun

https://godbolt.org/z/YxooTeGMT

22:00

this

this

What is `this`?

- Object on which the member function is invoked
- Implicitly the first argument to member function calls
- The type of `this` in a member function of class **X** is **X*** (pointer to **X**). If the member function is declared with a cv-qualifier sequence cv, the type of this is cv **X*** (pointer to identically cv-qualified **X**)

~~frie~~ ~~) { }~~

stat) { }


I'VE FINALLY FOUND IT... AFTER 15 YEARS
THE SCROLL OF TRUTH!
RobotatertotComics
real treasure was the friends we made along the way
NYEHHH
imgflip.com

`static` member functions

- Static members of a class are not associated with the objects of the class: they are independent variables with static or thread storage duration or regular functions.
- When called, they have no this pointer.
- Static member functions cannot be virtual, const, volatile, or ref-qualified.
- The address of a static member function may be stored in a regular pointer to function, but not in a pointer to member function.

```
struct party
{
    static void fun()              { std::print("static void fun()\n"); }









};

int main()
{
    party plain_party{};

    plain_party.fun();
    party::fun();
}
```

https://godbolt.org/z/sxxWT1EMh

```
struct party
{
    static void fun()             { std::print("static void fun()\n"); }

    /*
    void fun() &                  { std::print("void fun() &\n"); }
    void fun() &&                 { std::print("void fun() &&\n"); }
      . . .
    */

};

int main()
{
    party plain_party{};

    plain_party.fun();
    party::fun();
}
```

https://godbolt.org/z/sxxWT1EMh

Brief aside

What is unified function call syntax anyway? | Barry's C++ Blog

UFCS: Customization and Extension | Barry's C++ Blog

Unified function call syntax (UFCS)

UFCS is a breaking change, of the absolutely worst kind

clang

```
party::fun():

call party::fun()
call party::fun()
```

msvc

```
static void party::fun(void) PROC

call static void party::fun(void)
; party::fun
call static void party::fun(void)
; party::fun
```

The address of a static member function may be stored in a **regular pointer to function**, but not in a **pointer to member function**.

23:00

HIGHER-ORDER FUNCTIONS

Higher-order functions

- Term used by the senior developers to impress the juniors
- Term used by the programmers/developers/software engineers who studied maths

Source: Dude trust me

Higher-orde

In mathematics F) is a function that does at lea

- takes one c arameter, which is a parame
- returns a fu

All other functio

Source: Wikipe

“How do I write a higher-order function?”

```
auto first_order_fun() {
    std::print("inferior first_order_fun\n");
    return 3;
}

auto higher_order_fun(auto first_order_fun) {
    std::print("superior higher_order_fun\n");
    return first_order_fun();
}

int main()
{
    return higher_order_fun(first_order_fun);
}
```

“How do I write a higher-order function?”

```
auto first_order_fun() {
    std::print("inferior first_order_fun\n");
    return 3;
}

template<typename T>
auto higher_order_fun(T first_order_fun) {
    std::print("superior higher_order_fun\n");
    return first_order_fun();
}
int main()
{
    return higher_order_fun(first_order_fun);
}
```

`std::invoke`

invoke, std::invoke_r - cppreference.com

Pointers to functions?

- A pointer to function can be initialized with an address of a non-member function or a static member function
- Unlike functions or references to functions, pointers to functions are objects and thus can be stored in arrays, copied, assigned, etc.
- A pointer to function may be initialized from an overload set which may include functions, function template specializations, and function templates, if only one overload matches the type of the pointer

Pointers

```
int a{};
int* ptr = nullptr;
party p{};
party* ptr = nullptr
```

Pointers

```
int a{};
int* ptr = nullptr;
party p{};
party* ptr = nullptr
```

Pointers

```
int a{};
int* ptr = nullptr;
party p{};
party* ptr = nullptr
```

Pointers

```
int a{}, *ptr = nullptr;
party p{}, *ptr = nullptr;
```

Pointers

```
int *ptr = nullptr, a{};
party *ptr = nullptr, p{};
```

Pointers to functions

```
void fun(int a) {}
```

Pointers to functions

```
void fun(int a)
```

Pointers to functions

```
void fun(int a)
```

Pointers to functions

```
void *fun(int a)
```

Pointers to functions

```
void *fun(int a)
```

Pointers to functions

```
void (*fun)(int a)
```

Pointers to functions

```
void (*ptr)(int)
```

Pointers to functions

```
void fun(int a) {}

int main() {
    void (*ptr)(int) = fun;
}
```

Pointers to functions

```
void fun(int a) {}

int main() {
    using fun_ptr_type = void(*)(int);
    fun_ptr_type ptr = fun;
}
```

Pointers to functions

```
void fun(int&) {}
void fun(int&&) {}

int main() {
    using fun_ptr_type_1 = void(*)(int&);
    using fun_ptr_type_2 = void(*)(int&&);
    fun_ptr_type_1 ptr_1 = fun;
    fun_ptr_type_2 ptr_2 = fun;
}
```

```
void fun(int) { std::print("void fun(int)\n"); }


void fun(int&) { std::print("void fun(int&)\n"); }

int main()
{
    int a = 0;
    fun(a);
}
```

```
<source>:10:5: error: call to 'fun' is ambiguous
 10 |     fun(a);
    |     ^~~
<source>:3:6: note: candidate function
  3 | void fun(int) { std::print("void fun(int)\n"); }
    |      ^
<source>:5:6: note: candidate function
  5 | void fun(int&) { std::print("void fun(int&)\n"); }
    |      ^
1 error generated.
Compiler returned: 1
```

```
example.cpp
<source>(10): error C2668: 'fun': ambiguous call to
overloaded function
<source>(5): note: could be 'void fun(int &)'
<source>(3): note: or       'void fun(int)'
<source>(10): note: while trying to match the argument list
'(int)'
Compiler returned: 2
```

```
void fun(int) { std::print("void fun(int)\n"); }

void fun(int&) { std::print("void fun(int&)\n"); }

int main()
{
    int a = 3;
    void (*fun_ptr_int) (int) = &fun;
    void (*fun_ptr_int_ref) (int&) = &fun;
    std::invoke(fun_ptr_int, a);
    std::invoke(fun_ptr_int_ref, a);
}
```

https://godbolt.org/z/h7P4W3vvz

Pointers to functions?

```
void fun(int);
void (*p1)(int) = &fun;
void (*p2)(int) = fun; // same as &fun


void (a[10])(int);  // Error: array of
                    // functions
void (&a[10])(int); // Error: array of
                    // references
void (*a[10])(int); // OK: array of pointers
                    // to functions
```

```
int fun();
int (*p)() = fun;  // pointer p is pointing to f
int (&r)() = *p; // the lvalue that
                 // identifies f is bound to
                 // a reference
r();             // function f invoked
                 // through lvalue reference
(*p)();          // function f invoked
                 // through the function
                 // lvalue
p();             // function f invoked
                 // directly through the
                 // pointer
```

Pointers to member functions?

- A pointer to non-static member function f which is a member of class C can be initialized with the expression `&C::fun` exactly.
- Such a pointer may be used as the right-hand operand of the pointer-to-member access operators `operator.*` and `operator->*`.
- Pointer to member function of a base class can be implicitly converted to pointer to the same member function of a derived class
- Conversion from a pointer to member function of a derived class to a pointer to member function of an unambiguous non-virtual base class, is allowed with `static_cast` and explicit cast, even if the base class does not have that member function.

Pointer to member function?

```
struct party
{
    void fun(int n) { std::print("{}\n", n); }
};

int main()
{
    void (party::* ptr)(int) = &party::fun; // pointer to member function f of
                                            // struct party
    party p;
    (p.*ptr)(1);                    // prints 1
    party* pp = &p;
    (pp->*ptr)(2);                  // prints 2
}
```

Pointer to member function?

https://godbolt.org/z/n3Y47PWjq

```
struct party{
    static void static_fun () {}
    void regular_fun() {}
};

int main() {
    using static_fun_ptr_t = void(*)();
    using regular_member_fun_ptr_t = void(party::*)();

    static_fun_ptr_t static_ptr = party::static_fun;
    regular_member_fun_ptr_t regular_ptr = &party::regular_fun;
}
```

Pointer to member function?

https://godbolt.org/z/n3Y47PWjq

```
struct party{
    static void static_fun () {}
    void regular_fun() {}
};

int main() {
    using static_fun_ptr_t = void(*)();
    using regular_member_fun_ptr_t = void(party::*)();

    static_fun_ptr_t static_ptr = party::static_fun;
    regular_member_fun_ptr_t regular_ptr = party::regular_fun;
}
```

Pointer to member function?

GCC:

<source>: In function '**int main**()':

<source>:12:51: error: invalid use of non-static member function '**void party::regular_fun**()'

```
 12 |     regular_member_fun_ptr_t regular_ptr = party::regular_fun;
```

CLANG:

<source>:12:51: error: call to non-static member function without an object argument

```
 12 |     regular_member_fun_ptr_t regular_ptr = party::regular_fun;
```

MSVC:

<source>(12): error C3867: 'party::regular_fun': non-standard syntax; use '&' to create a pointer to member

“How do I write a higher-order function?”

```
auto first_order_fun() {
    std::print("inferior first_order_fun\n");
    return 3;
}

auto higher_order_fun(int (*first_order_fun)()) {
    std::print("superior higher_order_fun\n");
    return first_order_fun();
}

int main()
{
    return higher_order_fun(first_order_fun);
}
```

https://godbolt.org/z/jsMGxnaKb

00:00

Ok, cool.
But what is a static function but isn’t a static function?

But what is a static member function but isn’t a static member function?

[Deducing this]
[Deducing this Patterns - Ben Deane - CppCon 2021]
[C++ Weekly - Ep 326 - C++23's Deducing `this`]
[How C++23 Changes the Way We Write Code - Timur Doumler - CppCon 2022]

Explicit object parameter

- A non-static member function can be declared to take as its first parameter an explicit object parameter, denoted with the prefixed keyword **`this`**
- For member function templates, explicit object parameter allows deduction of type and value category, this language feature is called "deducing this"
- Inside the body of a function with explicit object parameter, the **`this`** pointer cannot be used
- A pointer to a member function with explicit object parameter is an ordinary pointer to function

Quadruplication

```
template <typename T>
class optional {
  // ...
  constexpr T& value() & {
    if (has_value()) {
      return this->m_value;
    }
    throw bad_optional_access();
  }

  constexpr T const& value() const& {
    if (has_value()) {
      return this->m_value;
    }
    throw bad_optional_access();
  }

  constexpr T&& value() && {
    if (has_value()) {
      return move(this->m_value);
    }
    throw bad_optional_access();
  }

  constexpr T const&&
  value() const&& {
    if (has_value()) {
      return move(this->m_value);
    }
    throw bad_optional_access();
  }
  // ...
};
```

Delegation to 4th

```
template <typename T>
class optional {
  // ...
  constexpr T& value() & {
    return const_cast<T&>(
      static_cast<optional const&>(
        *this).value());
  }

  constexpr T const& value() const& {
    if (has_value()) {
      return this->m_value;
    }
    throw bad_optional_access();
  }

  constexpr T&& value() && {
    return const_cast<T&&>(
      static_cast<optional const&>(
        *this).value());
  }

  constexpr T const&&
  value() const&& {
    return static_cast<T const&&>(
      value());
  }
  // ...
};
```

Delegation to helper

```
template <typename T>
class optional {
  // ...
  constexpr T& value() & {
    return value_impl(*this);
  }

  constexpr T const& value() const& {
    return value_impl(*this);
  }

  constexpr T&& value() && {
    return value_impl(move(*this));
  }

  constexpr T const&&
  value() const&& {
    return value_impl(move(*this));
  }

private:
  template <typename Opt>
  static decltype(auto)
  value_impl(Opt&& opt) {
    if (!opt.has_value()) {
      throw bad_optional_access();
    }
    return forward<Opt>(opt).m_value;
  }
  // ...
};
```

```
template <typename T>
struct optional {
 template <typename Self>
 constexpr auto&& value(this Self&& self) {
   if (!self.has_value()) {
     throw bad_optional_access();
   }

   return forward<Self>(self).m_value;
 }
}
```

```
struct party
{
    using lets = void;
    lets  get(this party started)  {}
};
```

clang msvc

```
party::get(this party):  static void party::get(UNKNOWN,party) PROC
```

```
party p{};
p.get();
void (*ptr) (party)  = &party::get;
void (*ptr1) (party) = party::get; // msvc ok, clang no, gcc no
std::invoke(ptr, p);
```

```
void fun(this party &) {}
void fun(this party &&) {}
void fun(this party const &) {}
void fun(this party const &&) {}
void fun(this party volatile &) {}
void fun(this party volatile &&) {}
void fun(this party const volatile &) {}
void fun(this party const volatile &&) {}
```

https://godbolt.org/z/e6qqz95jf

01:00


THERE IS A POINT WHERE WE NEEDED TO STOP
AND WE HAVE CLEARLY PASSED IT
BUT LET'S KEEP GOING
AND SEE WHAT HAPPENS

```
void fun(this party *) {}
void fun(this party const *) {}
void fun(this party volatile *) {}
void fun(this party const * const *) {}
void fun(this party const * volatile *) {}
```

https://godbolt.org/z/e58TroeMK

How do we call these overloads?

https://godbolt.org/z/bEKvMMrGd

```
struct party
{
    void fun(this party *) { std::print("void fun(this party *)\n"); }

};

int main()
{
    party p{};

    void (*fun_ptr) (party *) = &party::fun;
    std::invoke(fun_ptr, p);
}
```

How do we call these overloads?

https://godbolt.org/z/bEKvMMrGd

```
struct party
{
    void fun(this party *) { std::print("void fun(this party *)\n"); }

};

int main()
{
    party p{};
    p.fun();
    void (*fun_ptr) (party *) = &party::fun;
    std::invoke(fun_ptr, p);
}
```

How do we call these overloads?

https://godbolt.org/z/bEKvMMrGd

```
struct party
{
    void fun(this party *) { std::print("void fun(this party *)\n"); }
    operator party*() { return this; }
};

int main()
{
    party p{};
    p.fun();
    void (*fun_ptr) (party *) = &party::fun;
    std::invoke(fun_ptr, p);
}
```

??:??


Otvorili su mi se novi horizonti
gifs.com

```
struct party
{
    void fun(this int)
    {
        fmt::print("void fun(this int)\n");
    }
};
```

https://godbolt.org/z/5Kc8bszGP


```
struct party{
    static void fun() {}
    void fun(this party) {}
};
```

<source>:3:10: error: static and non-static member functions with the same parameter types cannot be overloaded

```
   3 |     void fun(this party) {}
     |          ^
```

<source>:2:17: note: previous definition is here

```
   2 |     static void fun() {}
     |                 ^
```

1 error generated.

Compiler returned: 1

<source>(3): error C7675: cannot overload static member function with member function declaring the same non-object parameter types

<source>(2): note: could be 'void party::fun(void)'

<source>(3): note: or   'void party::fun(party)'

<source>(4): error C2059: syntax error: '}'

<source>(4): error C2143: syntax error: missing ';' before '}'

Compiler returned: 2

```
party::fun():                  static void party::fun(void) PROC
party::fun(this party):        static void party::fun(UNKNOWN,party) PROC
```

What have we learned?

- Drink responsibly
- Getting into fights is no way to get things into standard

Wasn’t this fun?

misasedam  

an.milos94@gmail.com