Personal Log

Where No Init Has Gone Before

ANDREI ZISSU

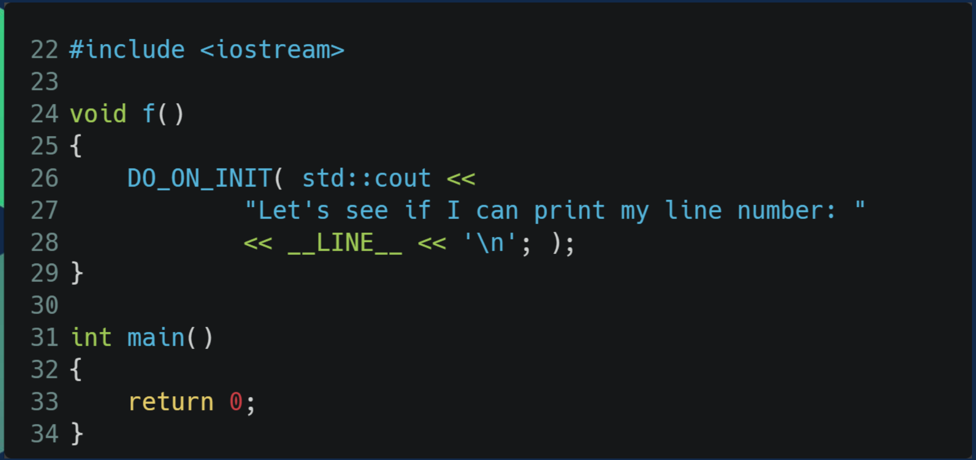
Cppcon The C++ Conference

2022 September 12th-16th

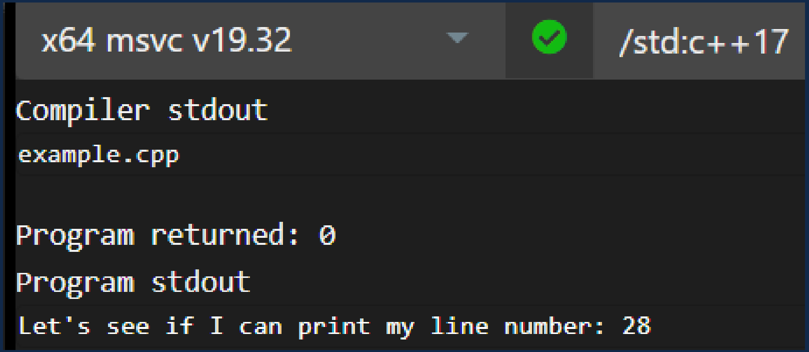
Personal Log:
Where No Init Has Gone Before

An Exploration on the C++ Fringe…

…Where the Uncalled for Happens Anyway



```
22 #include <iostream>
23
24 void f()
25 {
26     DO_ON_INIT( std::cout <<
27             "Let's see if I can print my line number: "
28             << __LINE__ << '\n'; );
29 }
30
31 int main()
32 {
33     return 0;
34 }
```



x64 msvc v19.32 ✓ /std:c++17

```
Compiler stdout
example.cpp

Program returned: 0
Program stdout
Let's see if I can print my line number: 28
```

Before We Go Any Further

I won’t be showing you actual logging stuff:

- File IO
- Parameter handling

Unless someone gives me a 90 minute slot… 😉

I will be showing you:

- Text encoding & decoding without any preprocessing
- The implementation of DO_ON_INIT
- In C++ 17

Who Am I?

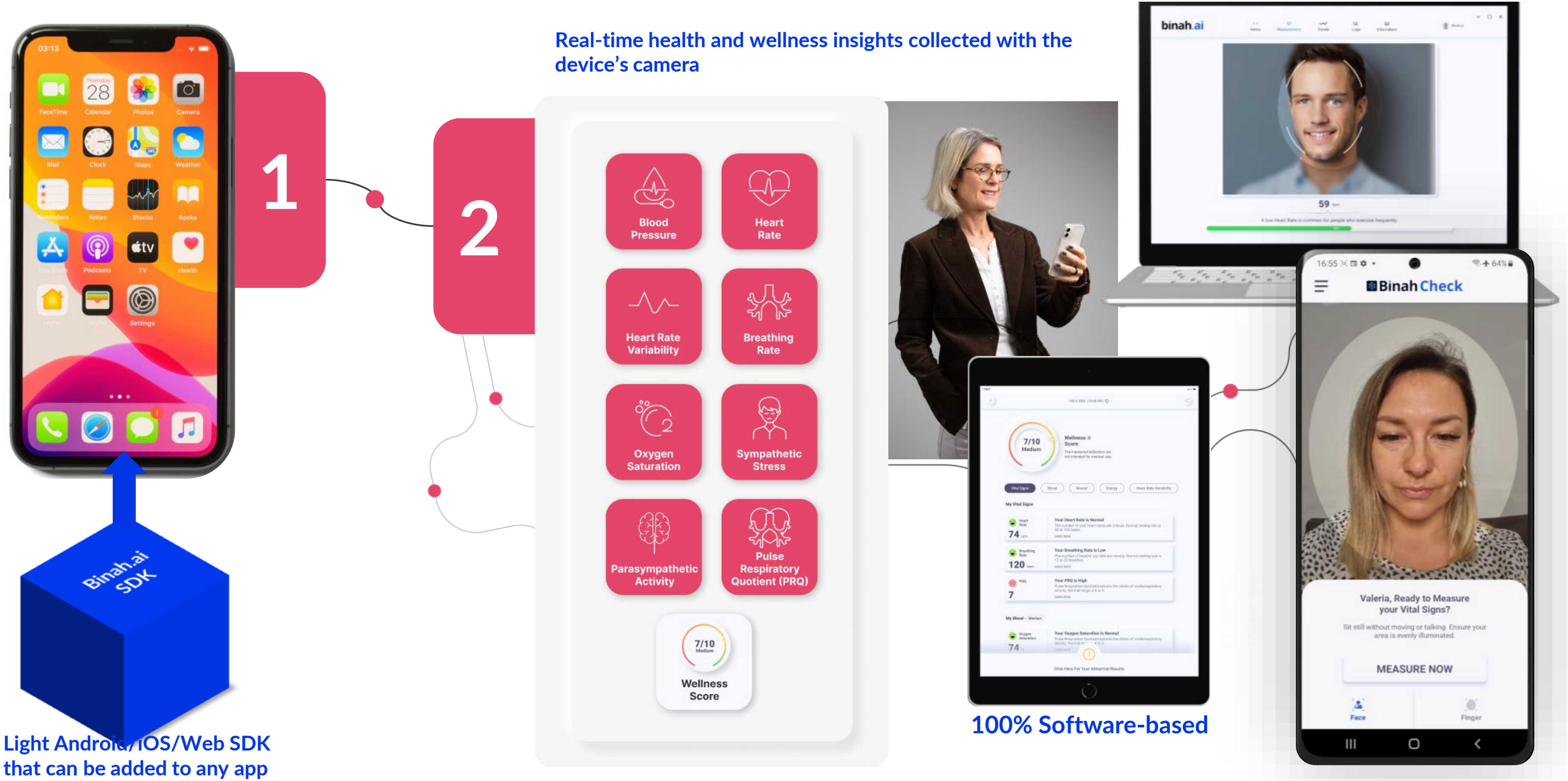
Andrei Zissu

- Israeli C++ programmer
- Multiple industries over the past 2 decades
  - Mobile, cyber, multimedia and more
- Member of WG21 Israeli NB
  - Special interest in reflection
- Working at Binah.ai

Binah.ai disrupts wellness and health monitoring

binah.ai
Health. Care. Anywhere.

Real-time health and wellness insights collected with the device's camera



100% Software-based

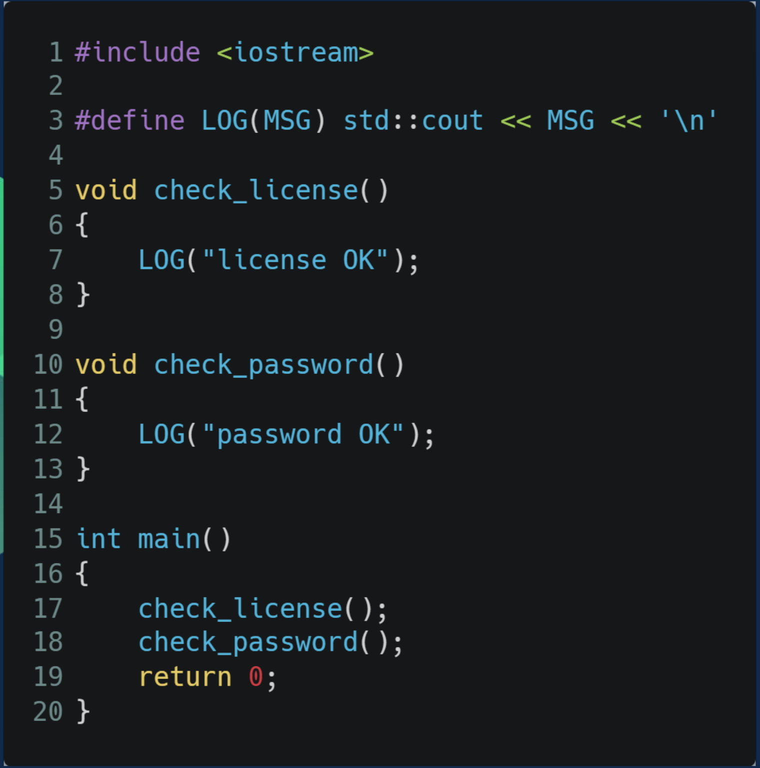
Light Android/iOS/Web SDK that can be added to any app

How It All Started

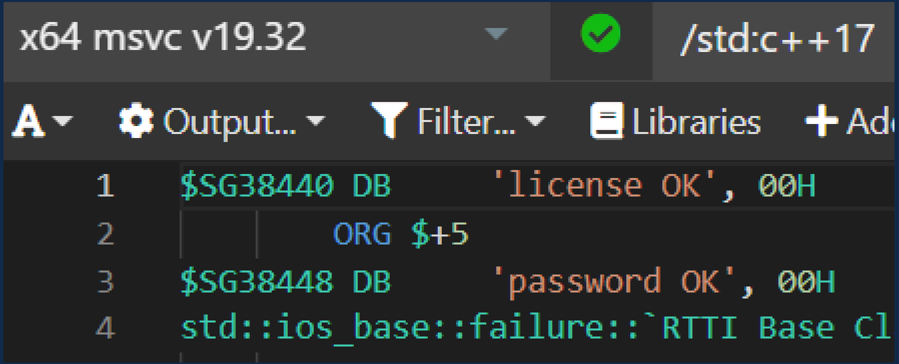
“Hi Andrei. Please remove sensitive log strings from our shipped binaries”

Simple…

Here’s the Problem



```cpp
#include <iostream>

#define LOG(MSG) std::cout << MSG << '\n'

void check_license()
{
    LOG("license OK");
}

void check_password()
{
    LOG("password OK");
}

int main()
{
    check_license();
    check_password();
    return 0;
}
```



x64 msvc v19.32 /std:c++17

```
$SG38440 DB      'license OK', 00H
        ORG $+5
$SG38448 DB      'password OK', 00H
std::ios_base::failure::`RTTI Base Cl
```

I’ll circle back later to why I’m showing this on msvc…

So How Do We Fix This (In C++17)?

- Replace strings with something else
- With what?
- Encrypted string?
  - But how would we produce one at compile time in C++17?
- Some numeric representation

Perhaps an enum With Log Msg Ids? Perhaps. Except…

- We'd need one for each unique message - upfront effort
- Lots of maintenance - when adding or modifying log messages
- Bug prone - stems directly out of the required maintenance

So Then…
Could We Do It Automatically?
Perhaps With Hashing?

Advantages:

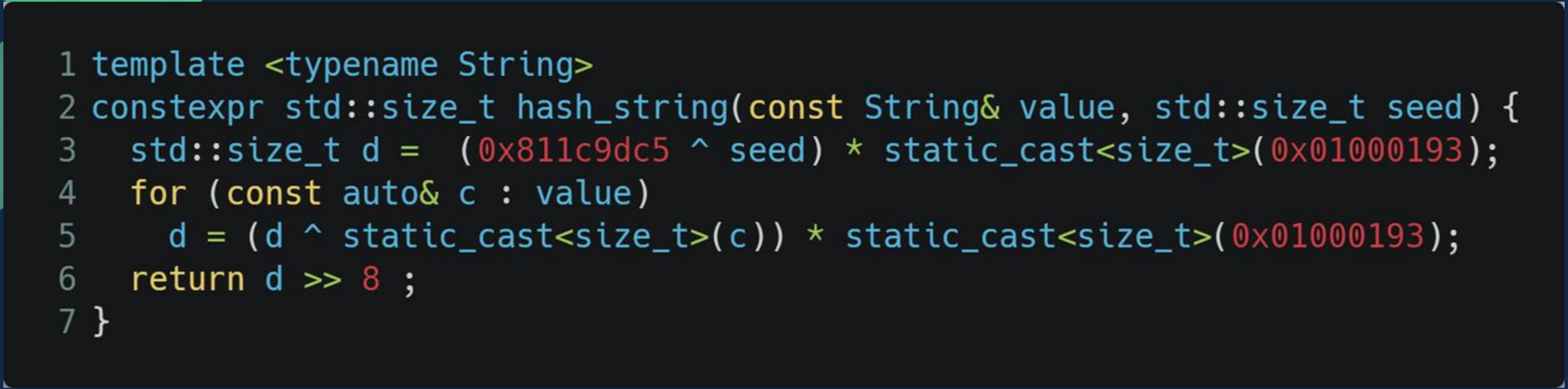
- Constant size regardless of input size - smaller binary, better security
- May be produced by a C++17 constexpr function (easy to find online, as I did)

Drawbacks:

- Can't be reversed (unlike encryption) - production code can't retrieve the original strings
- Hash collisions
  - Highly unlikely assuming a good hash function
  - Easily mitigated by guaranteed early detection (can just retry with a different hash key)

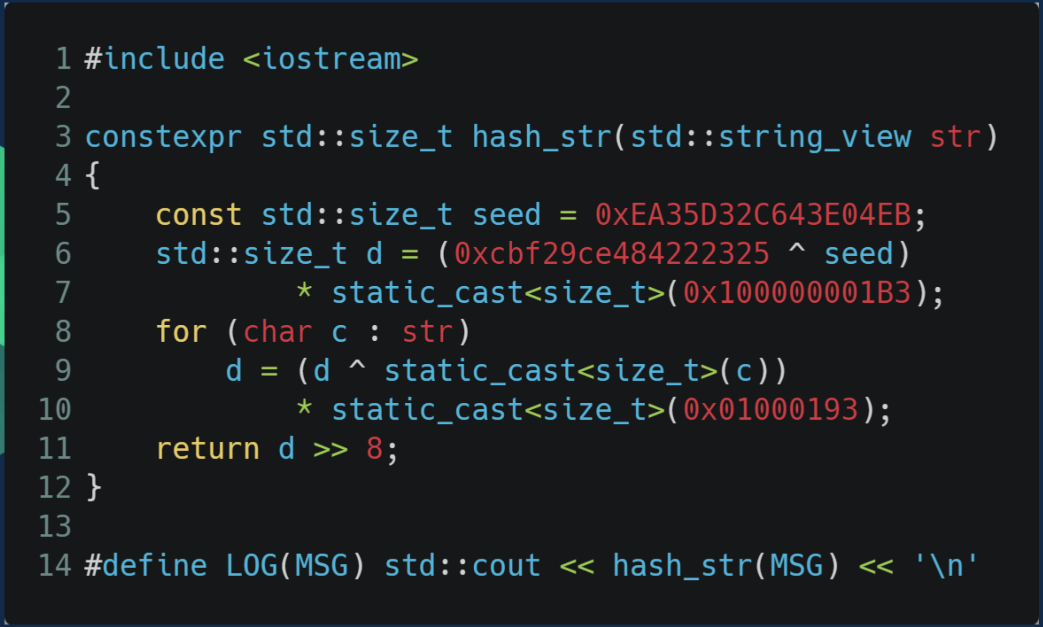
First Things First Though - Let’s Implement Log Hashing, We’ll Take Care of Decoding Later

Easily found constexpr hash function online (https://github.com/serge-sans-paille/frozen/blob/1f006e45adf600280bd3924513b80023e8dfdc80/include/frozen/bits/hash_string.h#L19)

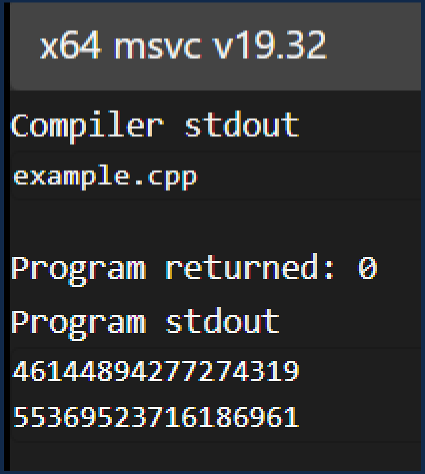


```cpp
template <typename String>
constexpr std::size_t hash_string(const String& value, std::size_t seed) {
  std::size_t d =  (0x811c9dc5 ^ seed) * static_cast<size_t>(0x01000193);
  for (const auto& c : value)
    d = (d ^ static_cast<size_t>(c)) * static_cast<size_t>(0x01000193);
  return d >> 8 ;
}
```

And Now With a Little Tweaking for My Needs…



```cpp
#include <iostream>

constexpr std::size_t hash_str(std::string_view str)
{
    const std::size_t seed = 0xEA35D32C643E04EB;
    std::size_t d = (0xcbf29ce484222325 ^ seed)
            * static_cast<size_t>(0x100000001B3);
    for (char c : str)
        d = (d ^ static_cast<size_t>(c))
            * static_cast<size_t>(0x01000193);
    return d >> 8;
}

#define LOG(MSG) std::cout << hash_str(MSG) << '\n'
```

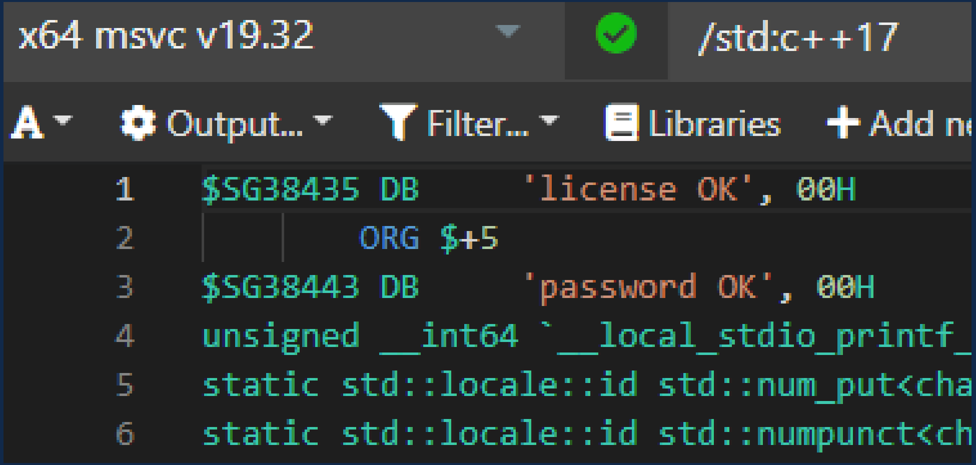


x64 msvc v19.32

```
Compiler stdout
example.cpp

Program returned: 0
Program stdout
46144894277274319
55369523716186961
```

Let’s just make sure the binary is indeed now clean of incriminating text…

Oops…

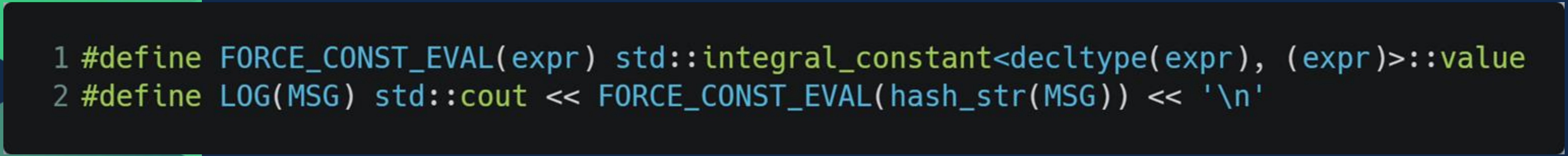


x64 msvc v19.32 /std:c++17

```
$SG38435 DB      'license OK', 00H
        ORG $+5
$SG38443 DB      'password OK', 00H
unsigned __int64 `__local_stdio_printf_
static std::locale::id std::num_put<cha
static std::locale::id std::numpunct<ch
```

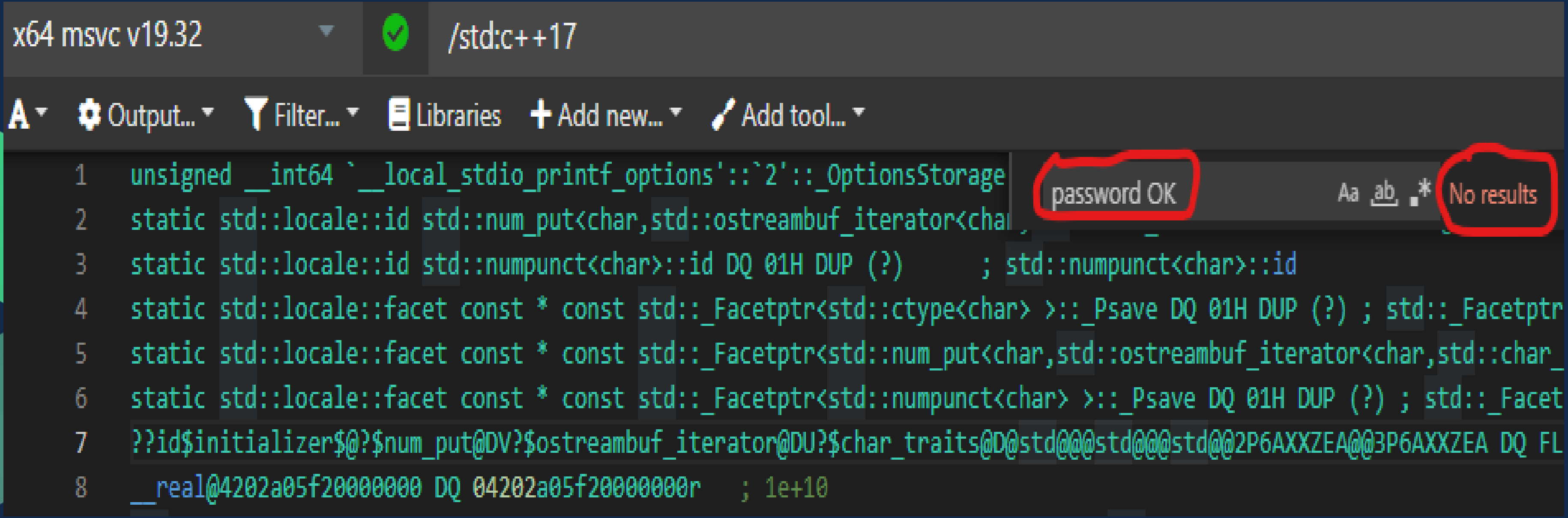
We might need to actually force a const evaluation…

…Which Only Takes a Single Extra Line of Code



```
#define FORCE_CONST_EVAL(expr) std::integral_constant<decltype(expr), (expr)>::value
#define LOG(MSG) std::cout << FORCE_CONST_EVAL(hash_str(MSG)) << '\n'
```

Good!



x64 msvc v19.32

/std:c++17

password OK

No results

```
unsigned __int64 `__local_stdio_printf_options'::`2'::_OptionsStorage
static std::locale::id std::num_put<char,std::ostreambuf_iterator<char
static std::locale::id std::numpunct<char>::id DQ 01H DUP (?)      ; std::numpunct<char>::id
static std::locale::facet const * const std::_Facetptr<std::ctype<char> >::_Psave DQ 01H DUP (?) ; std::_Facetptr
static std::locale::facet const * const std::_Facetptr<std::num_put<char,std::ostreambuf_iterator<char,std::char_
static std::locale::facet const * const std::_Facetptr<std::numpunct<char> >::_Psave DQ 01H DUP (?) ; std::_Facet
??id$initializer$@?$num_put@DV?$ostreambuf_iterator@DU?$char_traits@D@std@@@std@@@std@@2P6AXXZEA@@3P6AXXZEA DQ FL
__real@4202a05f20000000 DQ 04202a05f20000000r   ; 1e+10
```

Oh yeah, got rid of the log strings!
Oh, got rid of the log strings…
So… How do we get them back?

Decoding Hurdles

- Original strings are gone
- Hash functions are one way only
- Result: production code cannot access original strings
  - Log files contain only hash values, no strings
  - Preparing an offline dictionary is also impossible
- Conclusion: we need a separate decoding tool with access to the original strings
  - Which lucky for us are still there in the source code…

Decoder Design

What would the decoder tool do with the original strings?

- Calculate their hash values again, this time at run time

But where exactly would it get them from?

- The logger macros
- But we are not invoking that code in the decoder…
- And we need all of them…

What If…

What if we could somehow collect all the logged strings?

Without actual invocations…

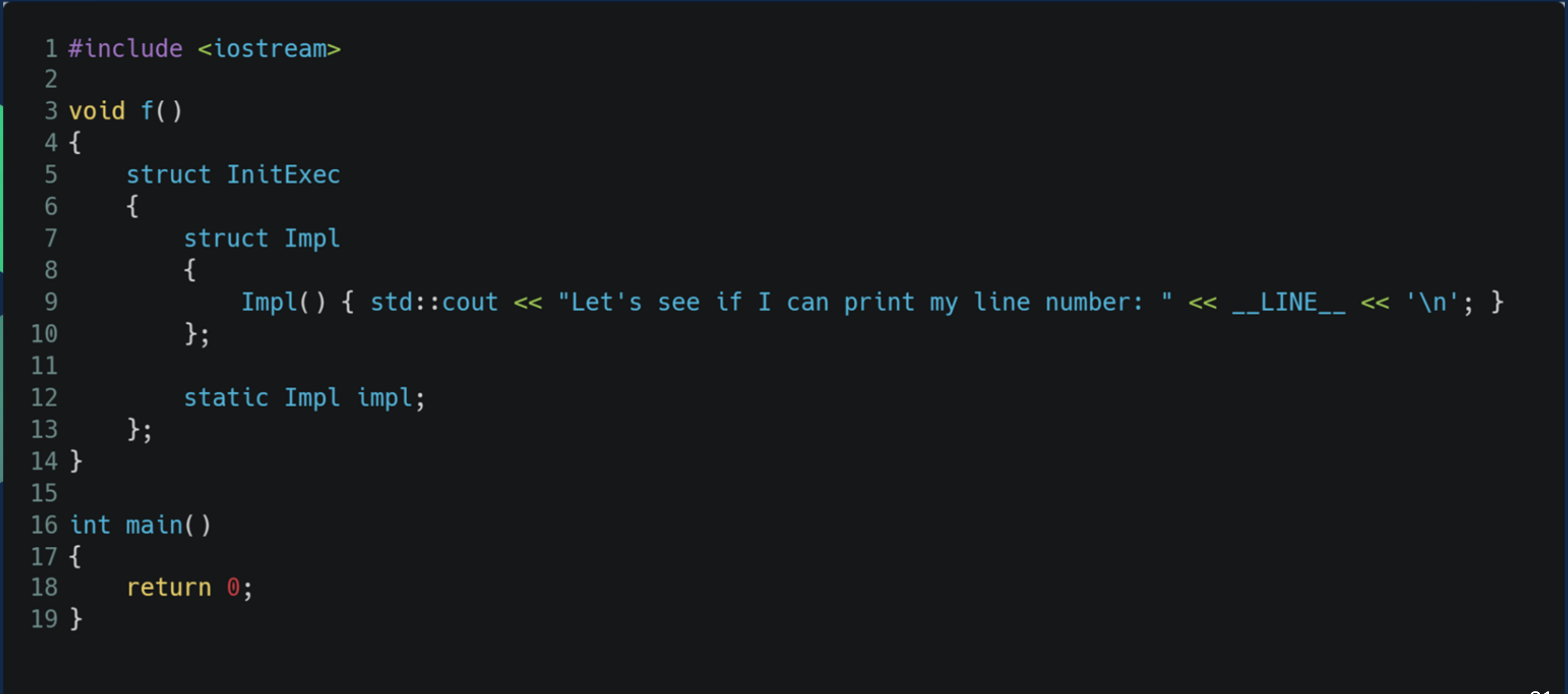
Before anything else happens…

What would it take to do that?

How Can You Do Something Automatically in C++?

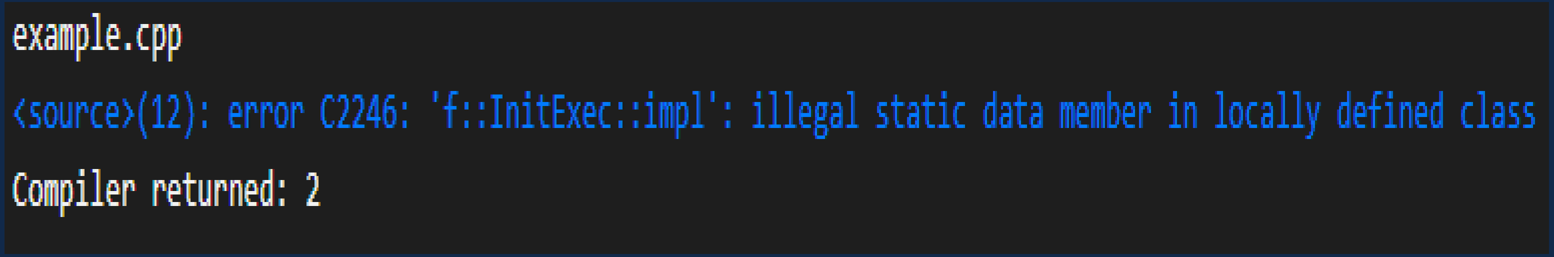
- Global object
  - But we need this done from local scopes. No way.
- Static data member of a class.
  - Locally defined classes are in the language (e.g. lambdas)
  - Let’s try this…

This Should Work. Right?



```cpp
#include <iostream>

void f()
{
    struct InitExec
    {
        struct Impl
        {
            Impl() { std::cout << "Let's see if I can print my line number: " << __LINE__ << '\n'; }
        };

        static Impl impl;
    };
}

int main()
{
    return 0;
}
```

Wrong!

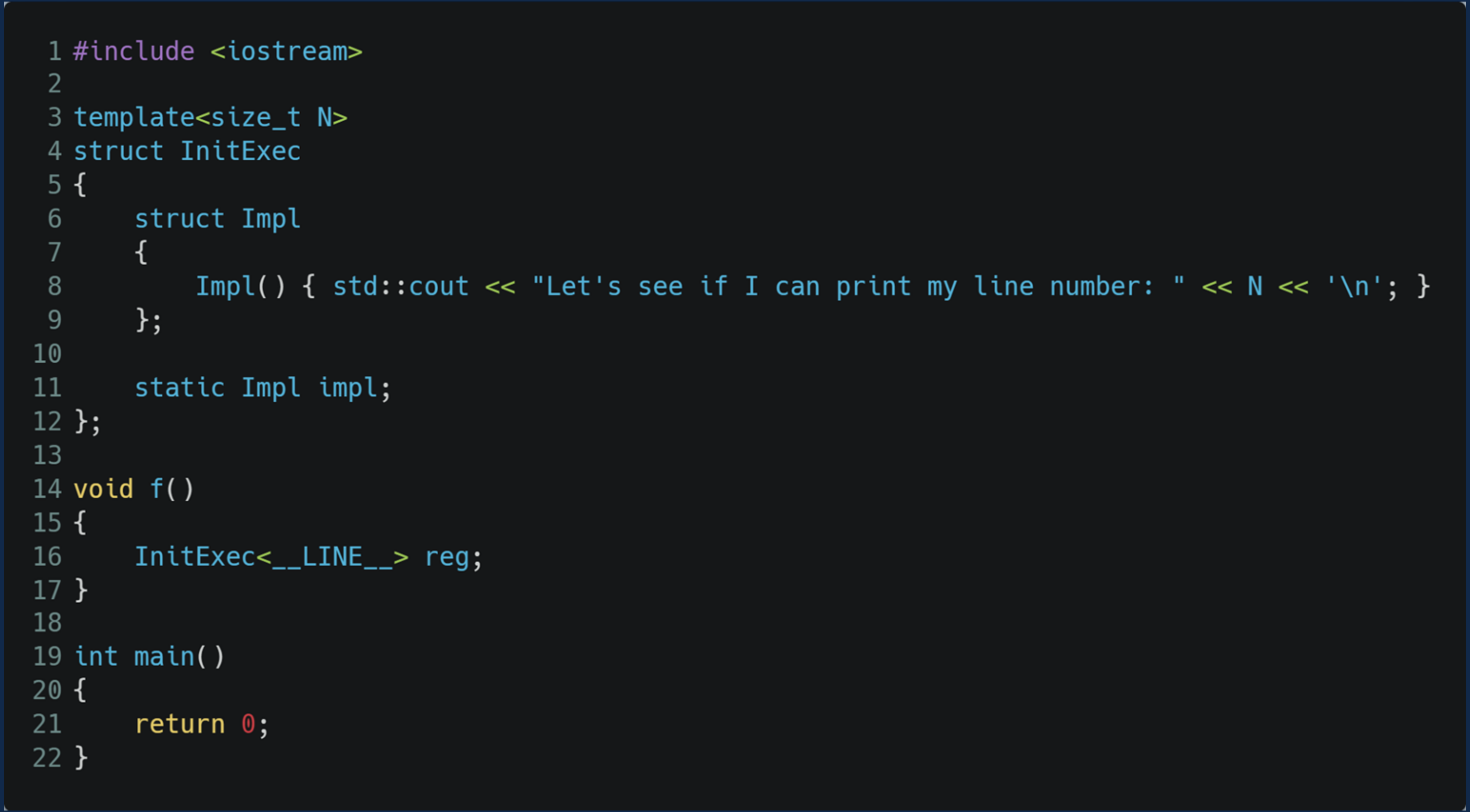


```
example.cpp
<source>(12): error C2246: 'f::InitExec::impl': illegal static data member in locally defined class
Compiler returned: 2
```

Locally defined classes cannot have static data members…

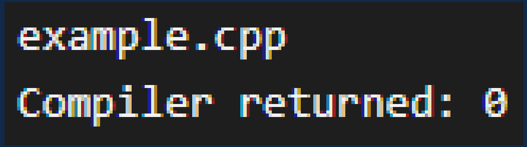
Now what?

How About We Extract InitExec Into a Template Class?



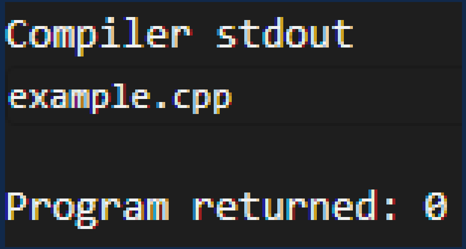
```cpp
#include <iostream>

template<size_t N>
struct InitExec
{
    struct Impl
    {
        Impl() { std::cout << "Let's see if I can print my line number: " << N << '\n'; }
    };

    static Impl impl;
};

void f()
{
    InitExec<__LINE__> reg;
}

int main()
{
    return 0;
}
```

Yes!!!
Managed to Build It



```
example.cpp
Compiler returned: 0
```

No…
Where's Our Log?



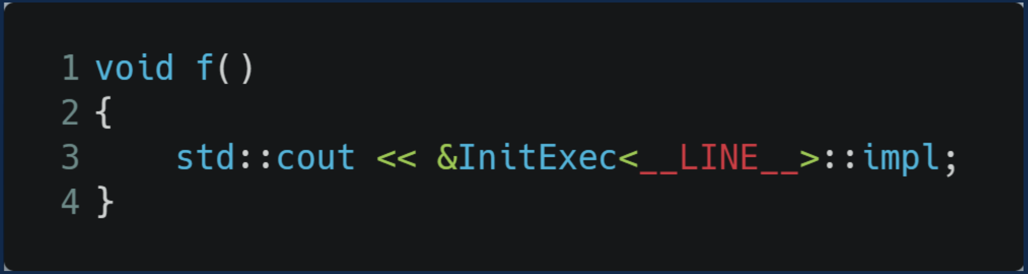
```
Compiler stdout
example.cpp


Program returned: 0
```

We just got optimized out.
Should have seen this one coming…

So What Might Force the Optimizer to Give Us a Break?

- Look for something that may not be optimized away
- An unused template instantiation won’t do, as we’ve just seen

Perhaps We Need to Force a Side Effect

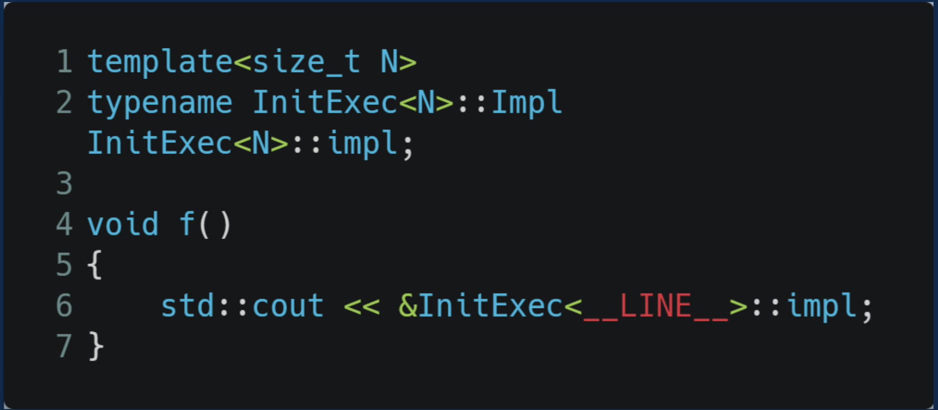


```cpp
void f()
{
    std::cout << &InitExec<__LINE__>::impl;
}
```

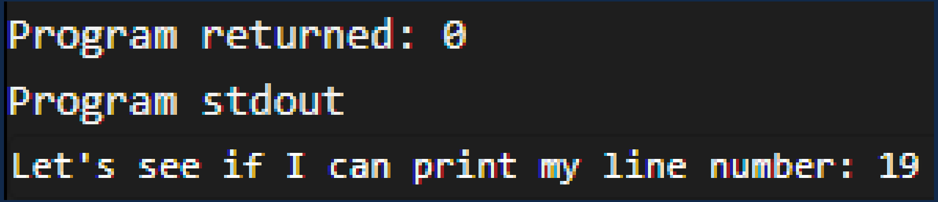
```
error LNK2019: unresolved external symbol "public: static struct
InitExec<16>::Impl InitExec<16>::impl" (?impl@?$InitExec@$0BA@@@2UImpl@1@A)
referenced in function "void __cdecl f(void)" (?f@@YAXXZ)
```

This is actually a good sign - we’ve passed the compilation phase

Let’s Calm Down the Linker Too

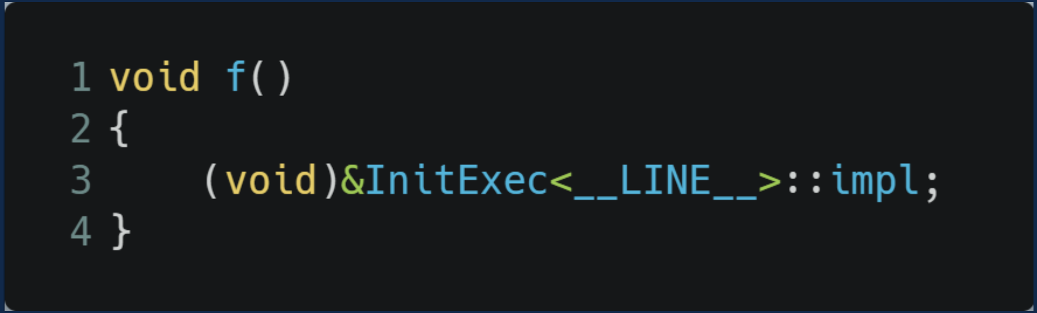


```cpp
template<size_t N>
typename InitExec<N>::Impl InitExec<N>::impl;

void f()
{
    std::cout << &InitExec<__LINE__>::impl;
}
```

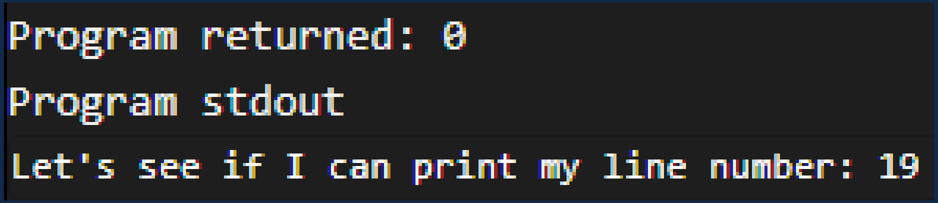


```
Program returned: 0
Program stdout
Let's see if I can print my line number: 19
```

But That’s Actually Quite Ugly
Let’s Get Rid of That Side Effect

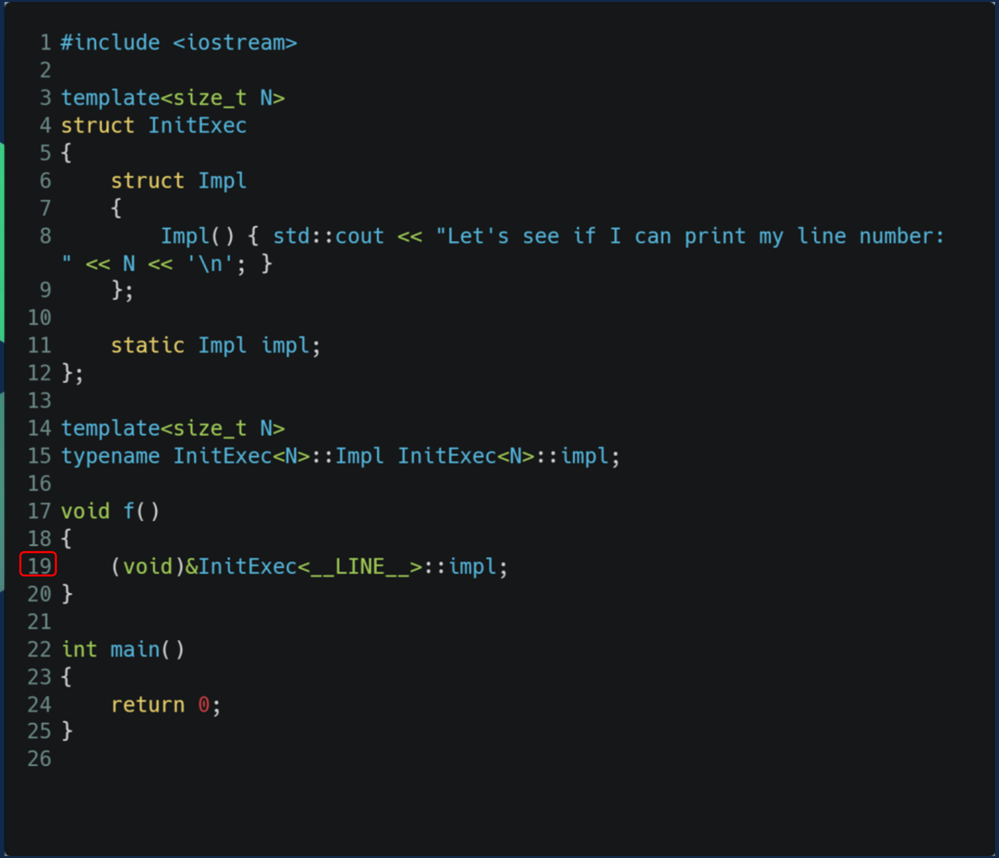


```
void f()
{
    (void)&InitExec<__LINE__>::impl;
}
```

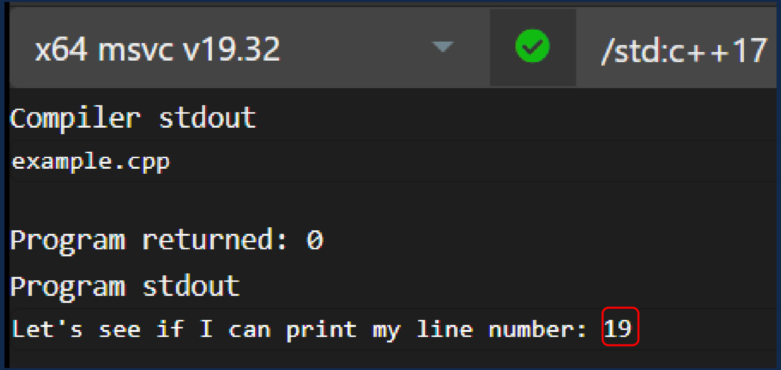


```
Program returned: 0
Program stdout
Let's see if I can print my line number: 19
```

Where We’re at So Far



```cpp
 1 #include <iostream>
 2
 3 template<size_t N>
 4 struct InitExec
 5 {
 6     struct Impl
 7     {
 8         Impl() { std::cout << "Let's see if I can print my line number: " << N << '\n'; }
 9     };
10
11     static Impl impl;
12 };
13
14 template<size_t N>
15 typename InitExec<N>::Impl InitExec<N>::impl;
16
17 void f()
18 {
19     (void)&InitExec<__LINE__>::impl;
20 }
21
22 int main()
23 {
24     return 0;
25 }
26
```



x64 msvc v19.32 /std:c++17

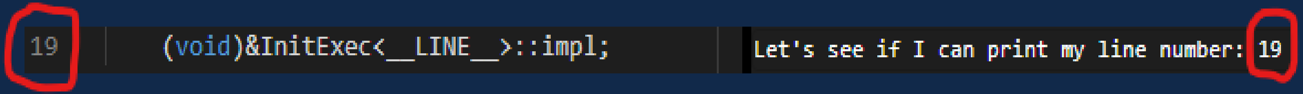
```
Compiler stdout
example.cpp

Program returned: 0
Program stdout
Let's see if I can print my line number: 19
```

That’s nice, but we’re not quite there yet

- We need the action (*cout* in this case) to be generated by local code!

That’s nice, but we’re not quite there yet

- We need the action (*cout* in this case) to be generated by local code!
- Actually we’ve already taken a small step…



```
19      (void)&InitExec<__LINE__>::impl;
```

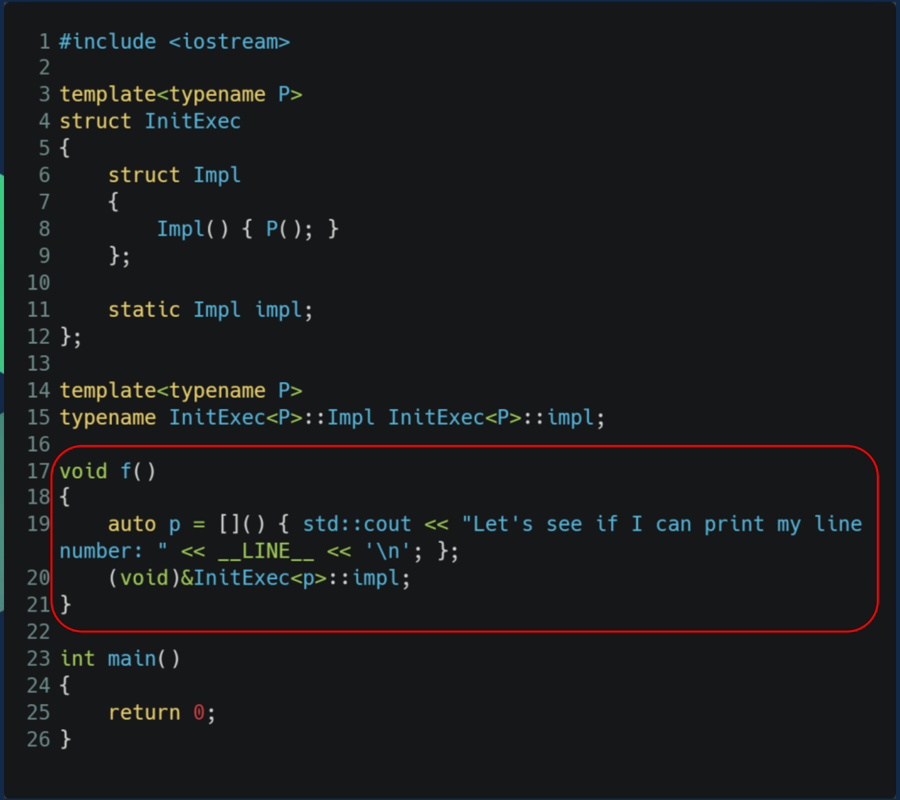
```
Let's see if I can print my line number: 19
```

- We’ve passed a small piece of state - the line number
- Could we go all the way and carry out any custom action?
- That sounds like… a lambda!

So How Do We Get a Lambda All the Way From f() To InitExec?

- As constructor parameter - static member, no way
- Template parameter it is then…

1st Try

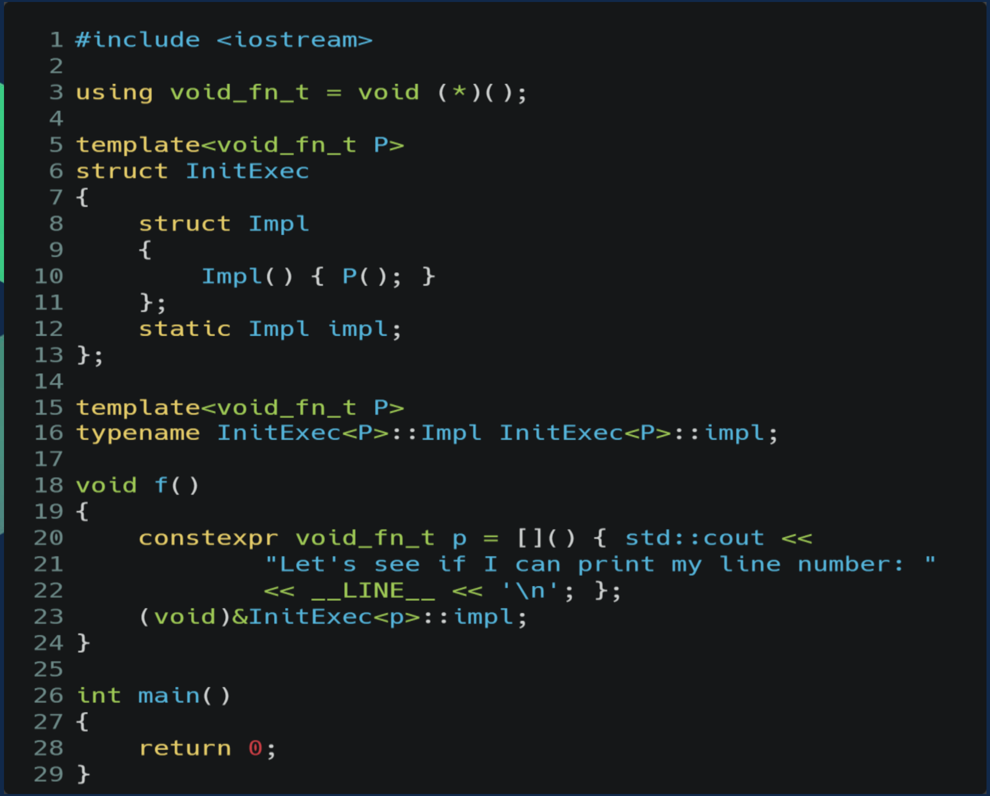


```cpp
#include <iostream>

template<typename P>
struct InitExec
{
    struct Impl
    {
        Impl() { P(); }
    };

    static Impl impl;
};

template<typename P>
typename InitExec<P>::Impl InitExec<P>::impl;

void f()
{
    auto p = []() { std::cout << "Let's see if I can print my line number: " << __LINE__ << '\n'; };
    (void)&InitExec<p>::impl;
}

int main()
{
    return 0;
}
```

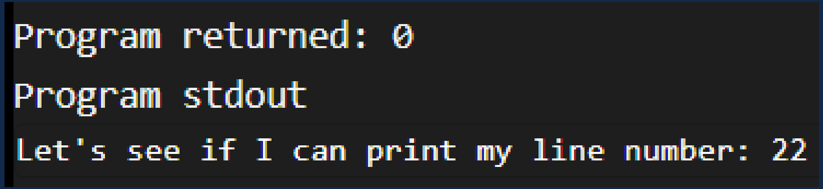
```
error C2923: 'InitExec': 'p' is not a valid
template type argument for parameter 'P'
error C2955: 'InitExec': use of class
template requires template argument list
```

Nope, we can’t just pass a lamda as a template argument

But Maybe There's Another Way After All…
(Thanks to the Almighty Internet 🖧)

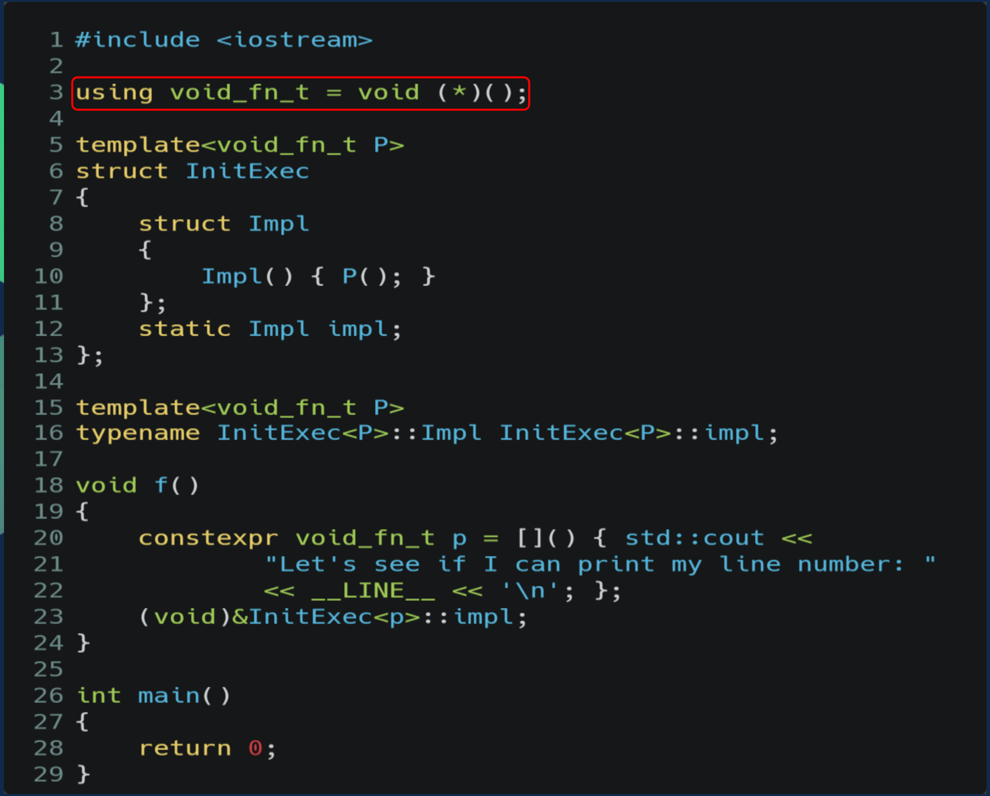


```cpp
#include <iostream>

using void_fn_t = void (*)();

template<void_fn_t P>
struct InitExec
{
    struct Impl
    {
        Impl() { P(); }
    };
    static Impl impl;
};

template<void_fn_t P>
typename InitExec<P>::Impl InitExec<P>::impl;

void f()
{
    constexpr void_fn_t p = []() { std::cout <<
            "Let's see if I can print my line number: "
            << __LINE__ << '\n'; };
    (void)&InitExec<p>::impl;
}

int main()
{
    return 0;
}
```

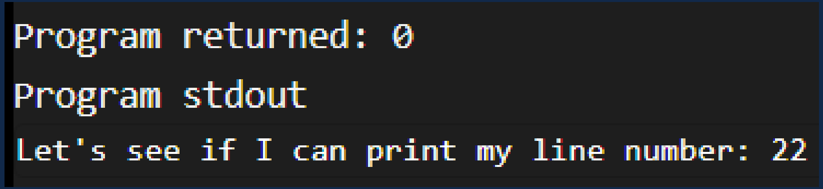


```
Program returned: 0
Program stdout
Let's see if I can print my line number: 22
```

But Maybe There's Another Way After All…
(Thanks to the Almighty Internet 🖧)

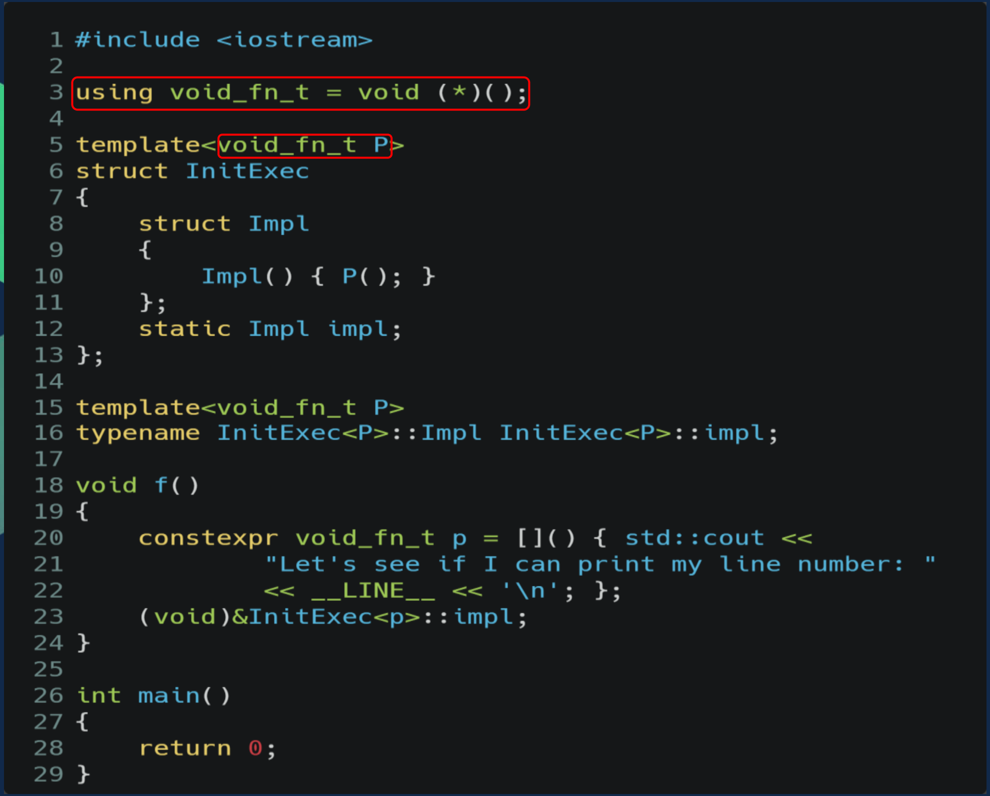


```cpp
#include <iostream>

using void_fn_t = void (*)();

template<void_fn_t P>
struct InitExec
{
    struct Impl
    {
        Impl() { P(); }
    };
    static Impl impl;
};

template<void_fn_t P>
typename InitExec<P>::Impl InitExec<P>::impl;

void f()
{
    constexpr void_fn_t p = []() { std::cout <<
            "Let's see if I can print my line number: "
            << __LINE__ << '\n'; };
    (void)&InitExec<p>::impl;
}

int main()
{
    return 0;
}
```

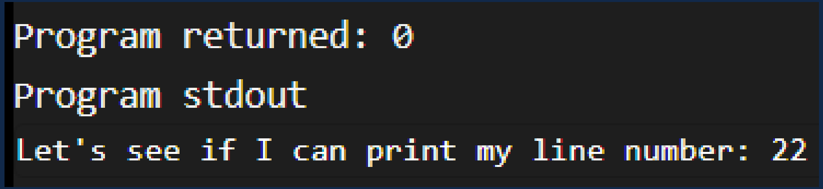


```
Program returned: 0
Program stdout
Let's see if I can print my line number: 22
```

But Maybe There’s Another Way After All… (Thanks to the Almighty Internet 🖧)

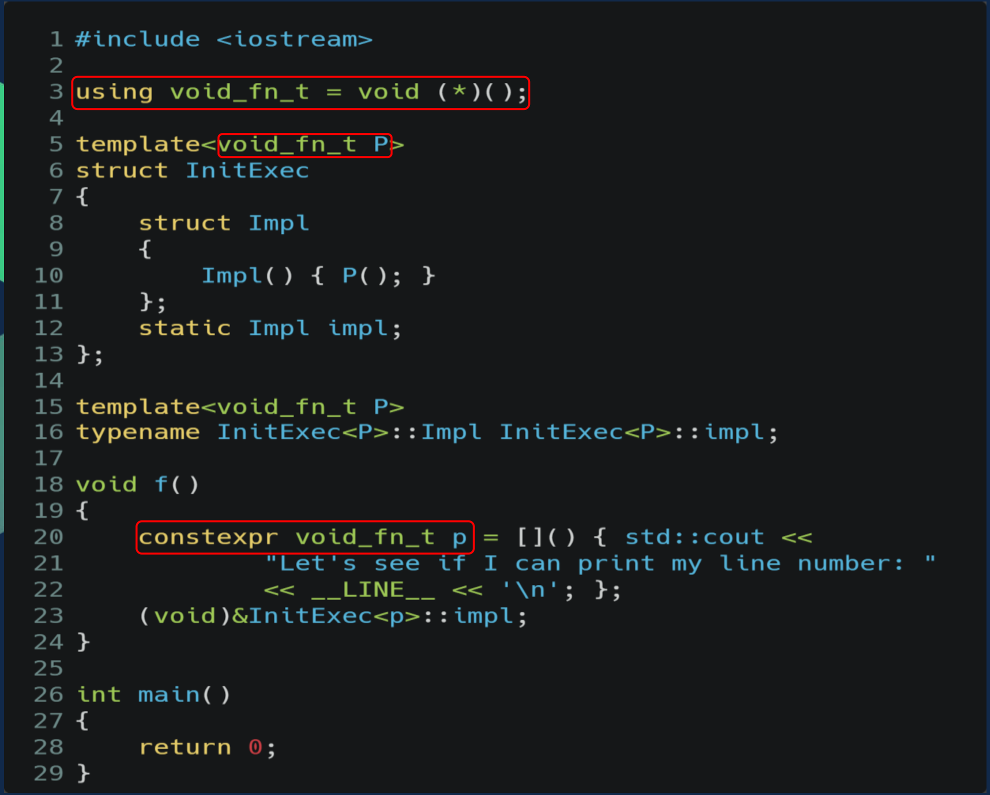


```
#include <iostream>

using void_fn_t = void (*)();

template<void_fn_t P>
struct InitExec
{
    struct Impl
    {
        Impl() { P(); }
    };
    static Impl impl;
};

template<void_fn_t P>
typename InitExec<P>::Impl InitExec<P>::impl;

void f()
{
    constexpr void_fn_t p = []() { std::cout <<
            "Let's see if I can print my line number: "
            << __LINE__ << '\n'; };
    (void)&InitExec<p>::impl;
}

int main()
{
    return 0;
}
```

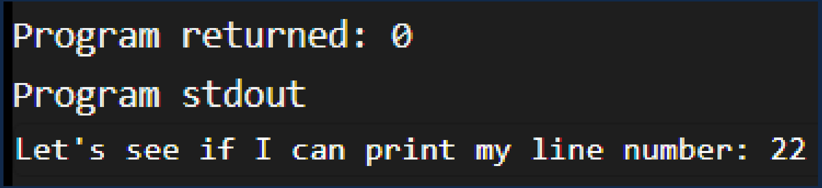


```
Program returned: 0
Program stdout
Let's see if I can print my line number: 22
```

But Maybe There's Another Way After All… (Thanks to the Almighty Internet 🖧)



```cpp
 1 #include <iostream>
 2 
 3 using void_fn_t = void (*)();
 4 
 5 template<void_fn_t P>
 6 struct InitExec
 7 {
 8     struct Impl
 9     {
10         Impl() { P(); }
11     };
12     static Impl impl;
13 };
14 
15 template<void_fn_t P>
16 typename InitExec<P>::Impl InitExec<P>::impl;
17 
18 void f()
19 {
20     constexpr void_fn_t p = []() { std::cout <<
21             "Let's see if I can print my line number: "
22             << __LINE__ << '\n'; };
23     (void)&InitExec<p>::impl;
24 }
25 
26 int main()
27 {
28     return 0;
29 }
```

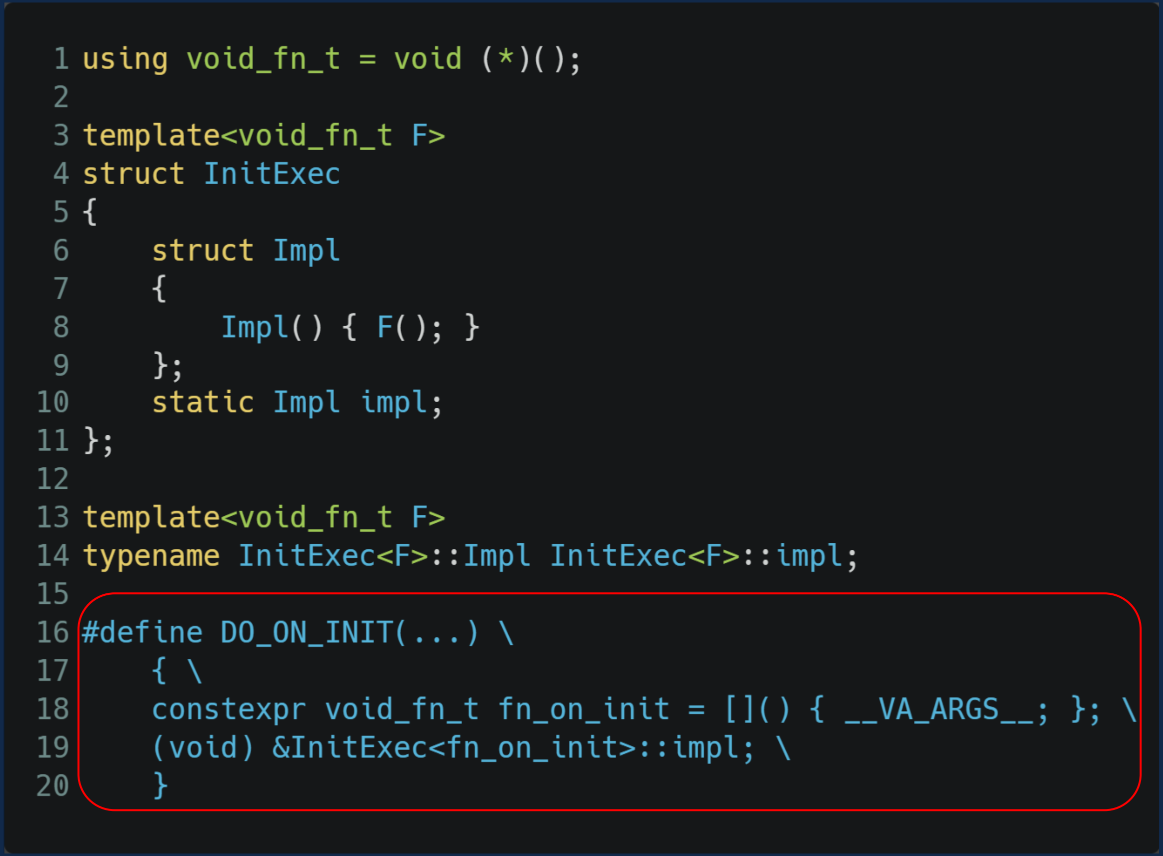


```
Program returned: 0
Program stdout
Let's see if I can print my line number: 22
```

And There You Have It!

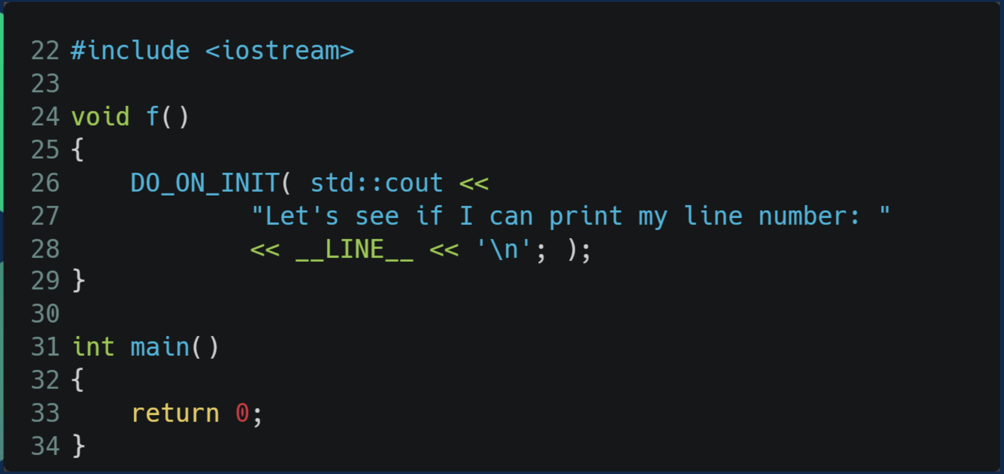
- Custom code executed at global init from a non-invoked context
- In C++ 17!
- Unfortunately not in all compilers (more about this later)
- This is the basis, now we’ll package it nicely as DO_ON_INIT

Summing It Up - DO_ON_INIT

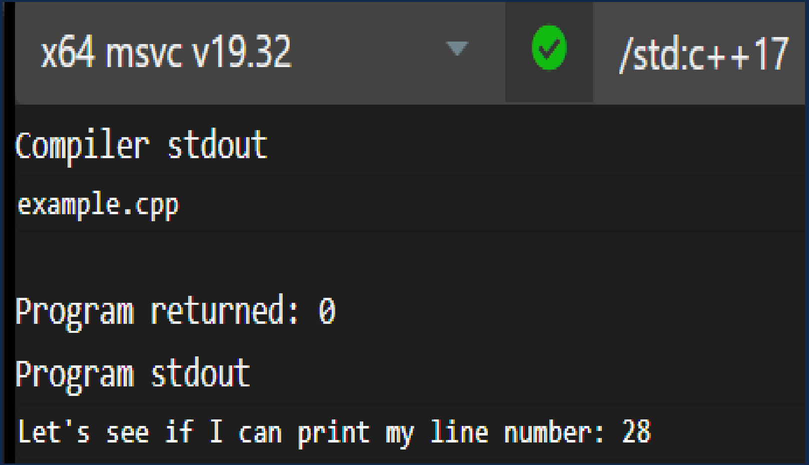


```
using void_fn_t = void (*)();

template<void_fn_t F>
struct InitExec
{
    struct Impl
    {
        Impl() { F(); }
    };
    static Impl impl;
};

template<void_fn_t F>
typename InitExec<F>::Impl InitExec<F>::impl;

#define DO_ON_INIT(...) \
    { \
    constexpr void_fn_t fn_on_init = []() { __VA_ARGS__; }; \
    (void) &InitExec<fn_on_init>::impl; \
    }
```

And Now This Is Finally Possible

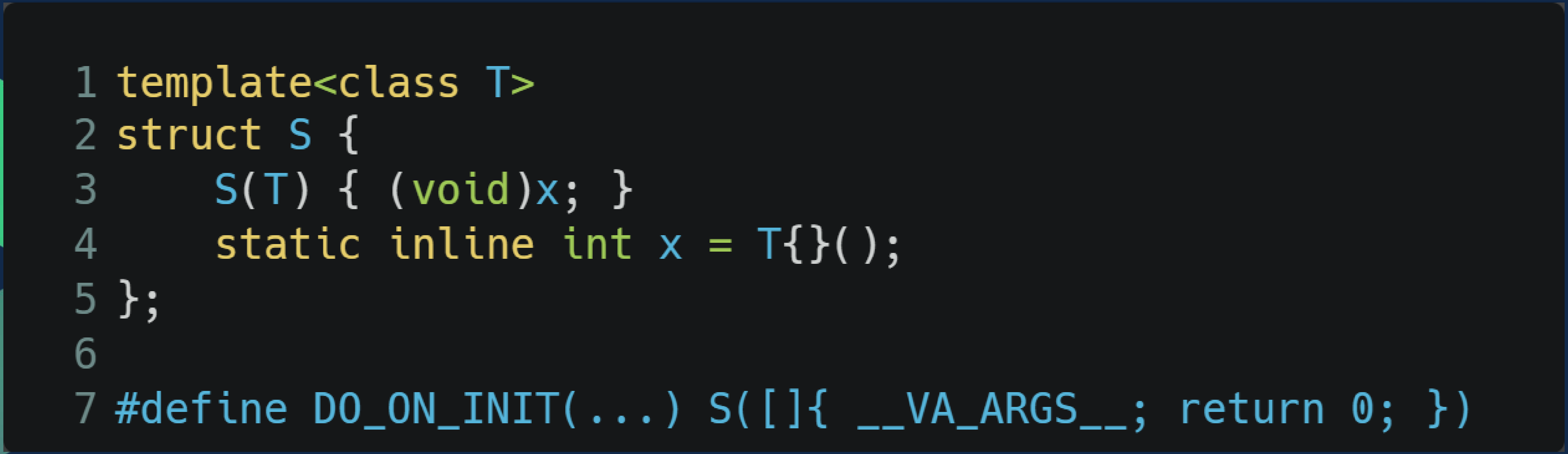


```
22 #include <iostream>
23
24 void f()
25 {
26     DO_ON_INIT( std::cout <<
27             "Let's see if I can print my line number: "
28             << __LINE__ << '\n'; );
29 }
30
31 int main()
32 {
33     return 0;
34 }
```



```
Compiler stdout
example.cpp

Program returned: 0
Program stdout
Let's see if I can print my line number: 28
```

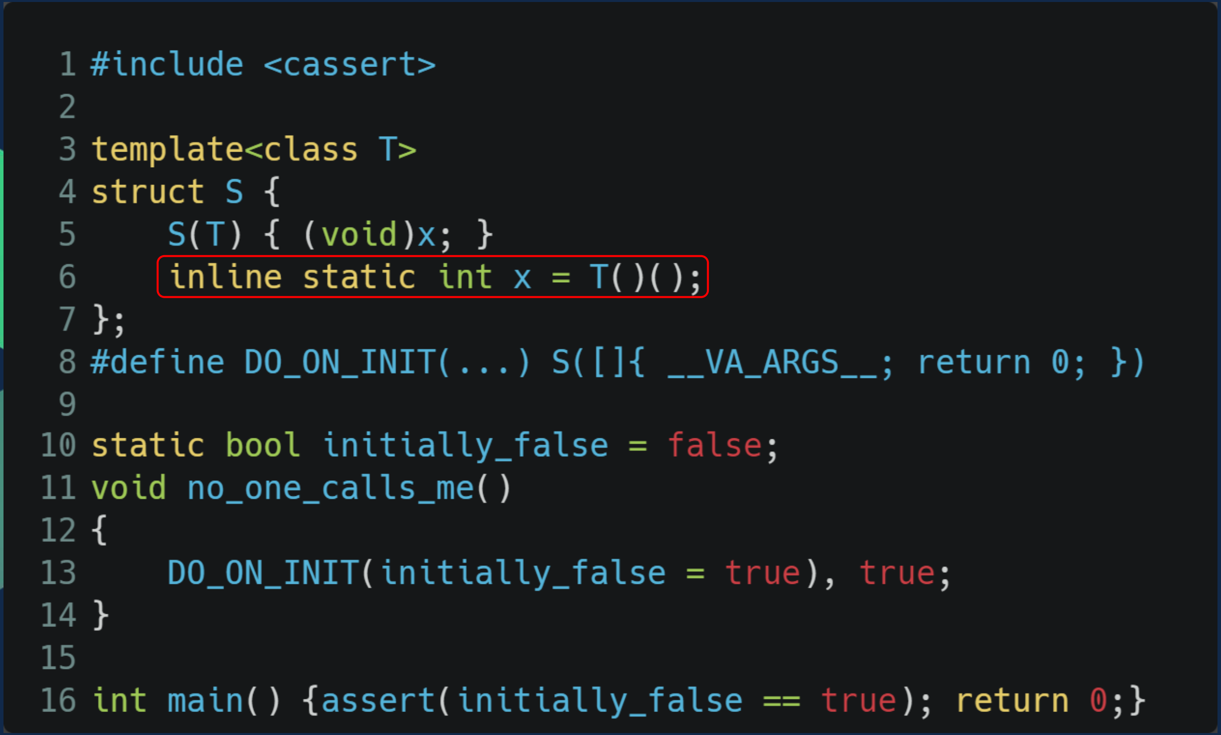
And It’s Even Easier In C++ 20



```
template<class T>
struct S {
    S(T) { (void)x; }
    static inline int x = T{}();
};

#define DO_ON_INIT(...) S([]{ __VA_ARGS__; return 0; })
```

* Based on code contributed by Arthur O'Dwyer

Inline Static Is Actually Available in C++17



```cpp
#include <cassert>

template<class T>
struct S {
    S(T) { (void)x; }
    inline static int x = T()();
};
#define DO_ON_INIT(...) S([]{ __VA_ARGS__; return 0; })

static bool initially_false = false;
void no_one_calls_me()
{
    DO_ON_INIT(initially_false = true), true;
}

int main() {assert(initially_false == true); return 0;}
```

But it crashed on me in VS 2019. Go figure…

Questions So Far?

Putting It All Together

We started off with encoding our log strings at compile time:

```
#define FORCE_CONST_EVAL(expr) std::integral_constant<decltype(expr), (expr)>::value
#define LOG(MSG) std::cout << FORCE_CONST_EVAL(hash_str(MSG)) << '\n'
```

So How Do We Build the Decoder Tool?

- Production code doesn’t have the original strings
- But the source code does!
- Same LOG macros, different implementation when built for decoding
- Doing what? - mapping the string hash values to the original strings
- When? - before all else, to have the mapping handy when needed
- How? - well, with DO_ON_INIT of course!

Overall Design

- BUILD_FOR_ENCODING compile-time switch
- If on
  - LOG macro substitutes logged string with hash at compile time
- If off
  - LOG macro uses DO_ON_INIT to register the logged string and its hash, at run time
  - Any hash encountered in the log file is replaced with the original string
- Out of scope in this talk
  - File IO (both ways)
  - Log parameters

Let’s Start, Top to Bottom

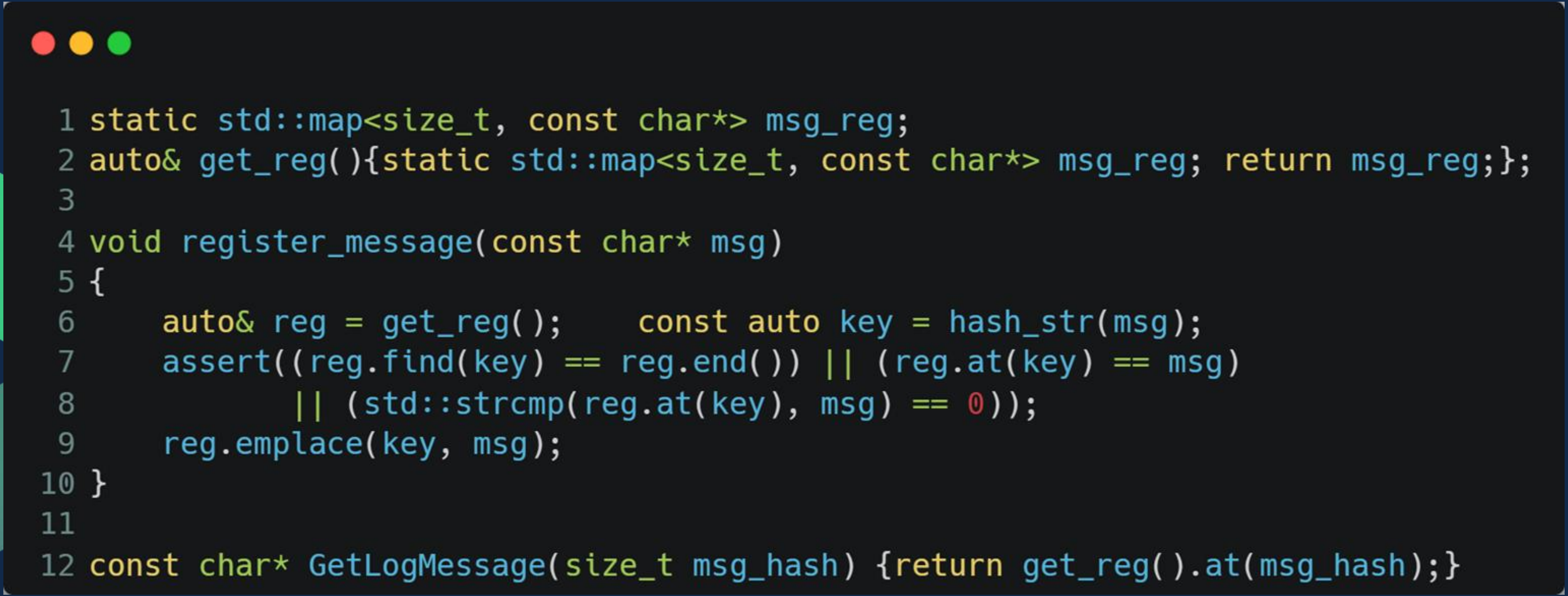
```
#ifdef BUILD_FOR_ENCODING
    #define LOG(MSG) std::cout << HASH(MSG) << '\n'
#else
    #define LOG(MSG) DO_ON_INIT(register_message(MSG))
#endif
```

Let’s Start, Top to Bottom

```
#ifdef BUILD_FOR_ENCODING
    #define LOG(MSG) std::cout << HASH(MSG) << '\n'
#else
    #define LOG(MSG) DO_ON_INIT(register_message(MSG))
#endif
```

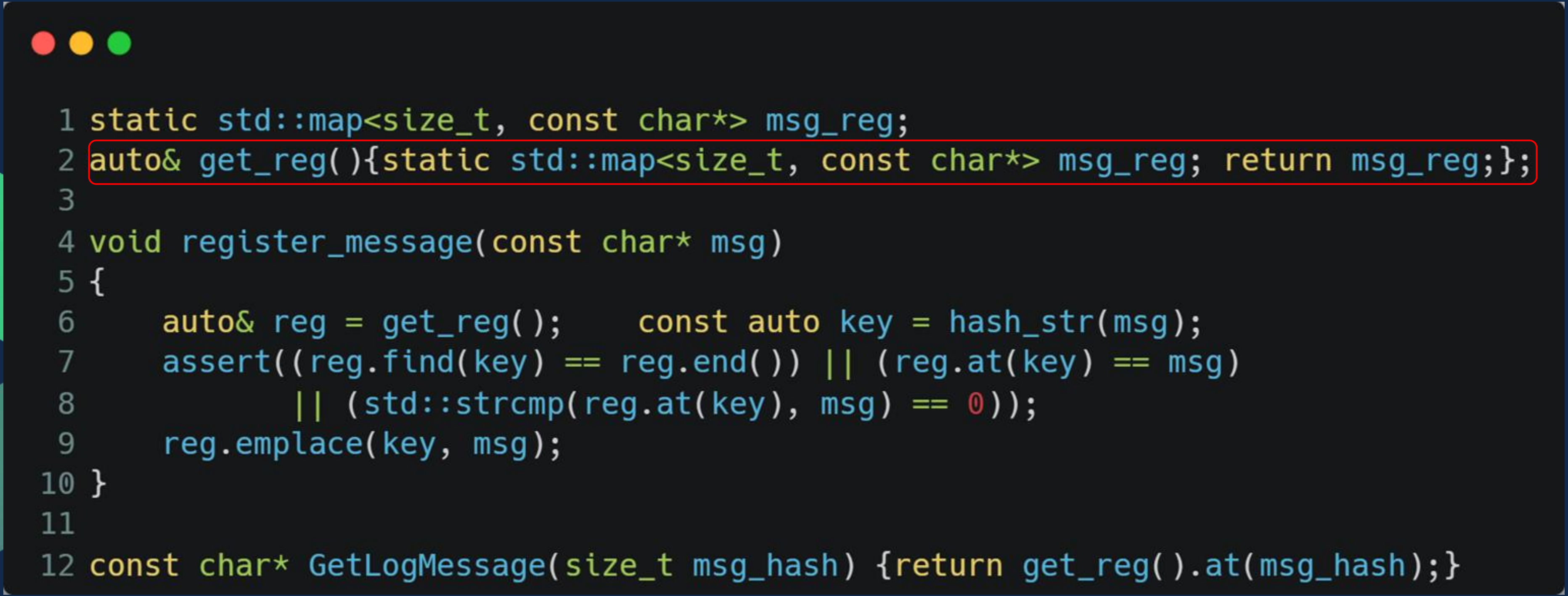
We’ve already seen the encoding part, so let’s just focus on the decoder

Registration Is Pretty Straightforward



```
static std::map<size_t, const char*> msg_reg;
auto& get_reg(){static std::map<size_t, const char*> msg_reg; return msg_reg;};

void register_message(const char* msg)
{
    auto& reg = get_reg();    const auto key = hash_str(msg);
    assert((reg.find(key) == reg.end()) || (reg.at(key) == msg)
           || (std::strcmp(reg.at(key), msg) == 0));
    reg.emplace(key, msg);
}

const char* GetLogMessage(size_t msg_hash) {return get_reg().at(msg_hash);}
```

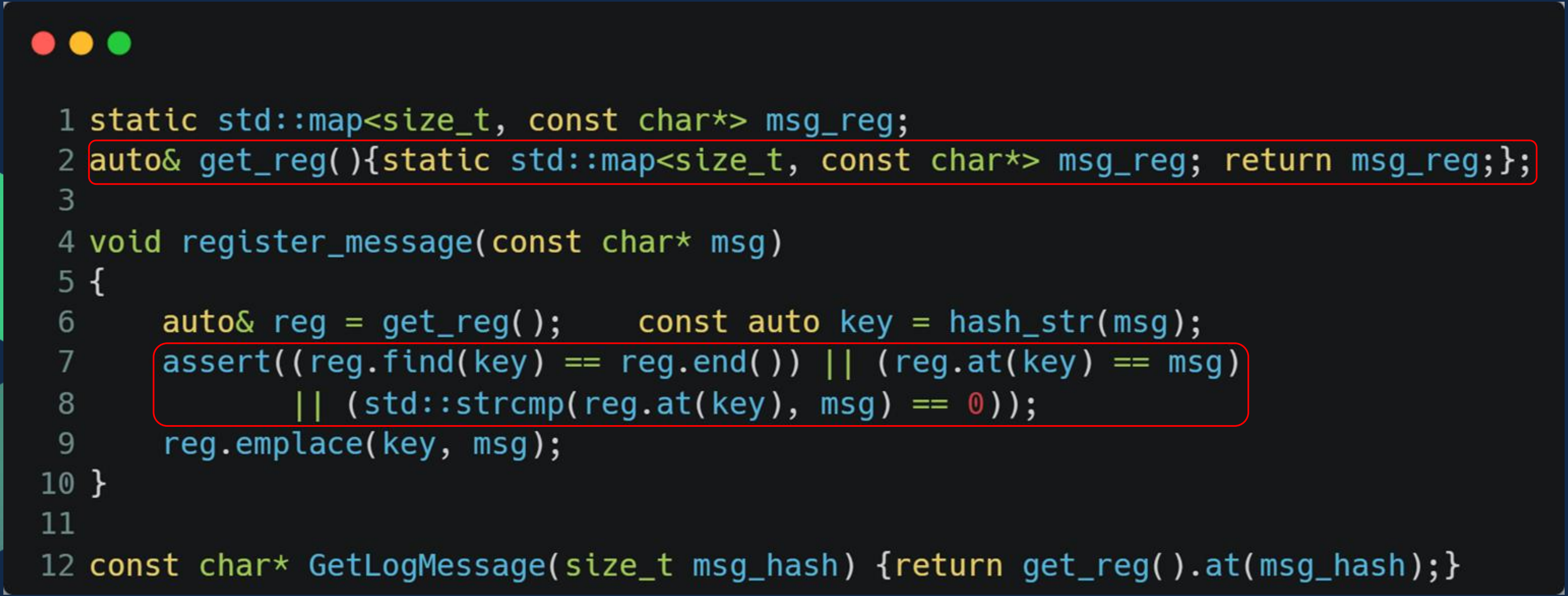
Registration Is Pretty Straightforward



```cpp
static std::map<size_t, const char*> msg_reg;
auto& get_reg(){static std::map<size_t, const char*> msg_reg; return msg_reg;};

void register_message(const char* msg)
{
    auto& reg = get_reg();    const auto key = hash_str(msg);
    assert((reg.find(key) == reg.end()) || (reg.at(key) == msg)
            || (std::strcmp(reg.at(key), msg) == 0));
    reg.emplace(key, msg);
}

const char* GetLogMessage(size_t msg_hash) {return get_reg().at(msg_hash);}
```

- **Lazy init in get_reg() insures lifetime control during global init sequence**

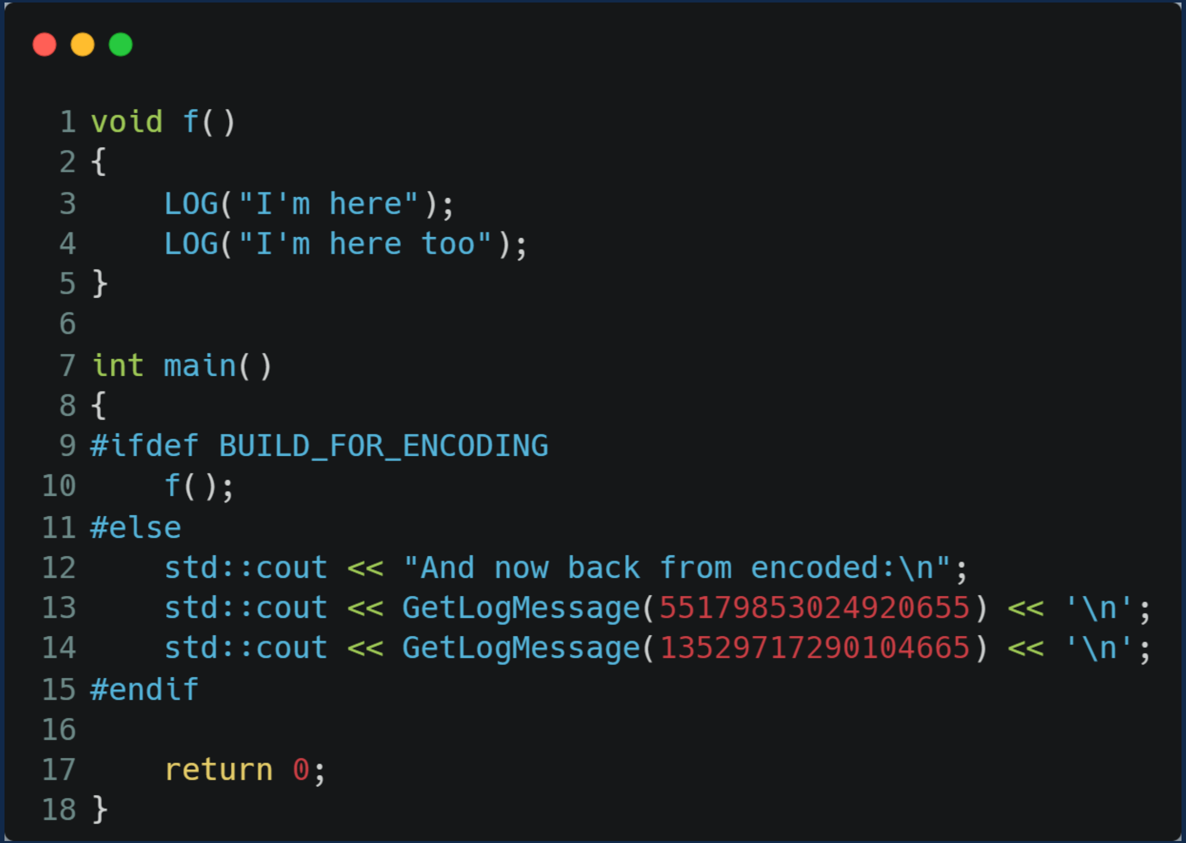
Registration Is Pretty Straightforward



```
static std::map<size_t, const char*> msg_reg;
auto& get_reg(){static std::map<size_t, const char*> msg_reg; return msg_reg;};

void register_message(const char* msg)
{
    auto& reg = get_reg();    const auto key = hash_str(msg);
    assert((reg.find(key) == reg.end()) || (reg.at(key) == msg)
           || (std::strcmp(reg.at(key), msg) == 0));
    reg.emplace(key, msg);
}

const char* GetLogMessage(size_t msg_hash) {return get_reg().at(msg_hash);}
```

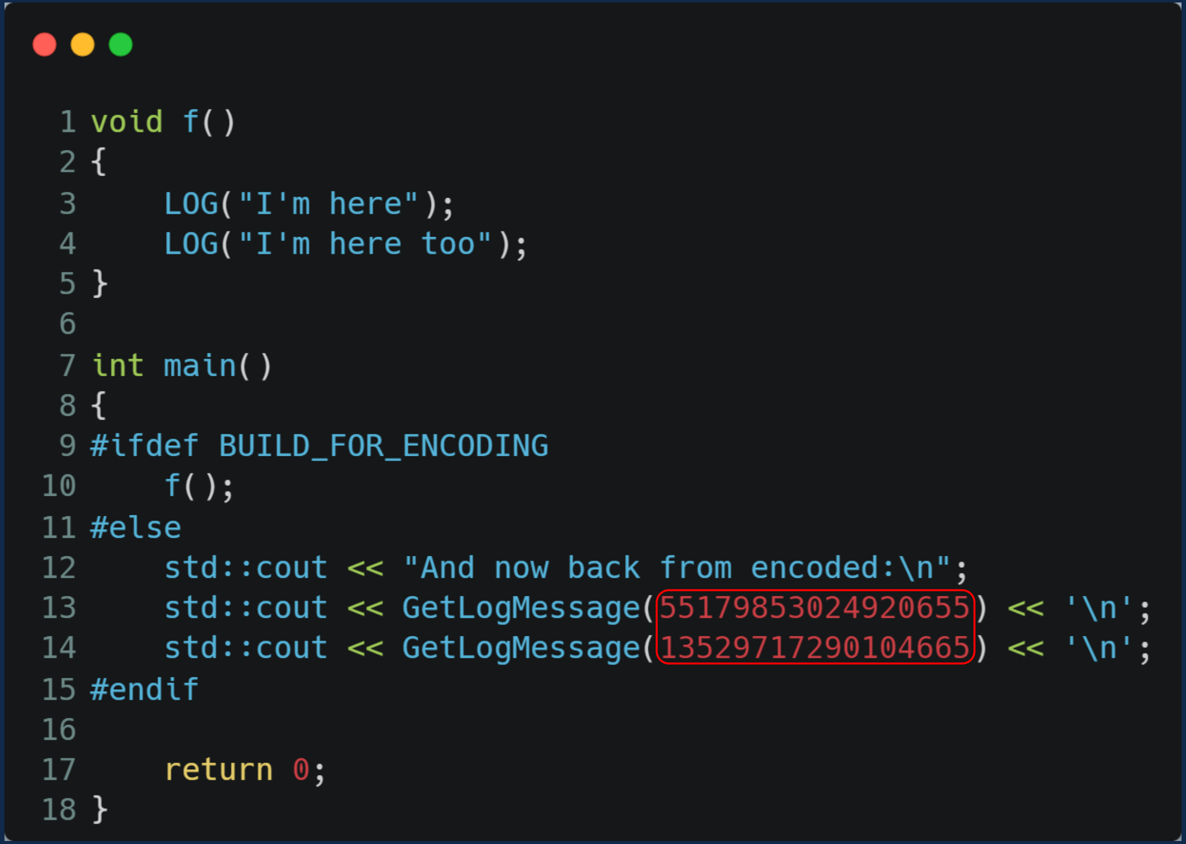
- **Lazy init in get_reg() insures lifetime control during global init sequence**
- **The assert is our safety net against hash collisions**
  - **Every log message is registered, so the assert is guaranteed to be checked for all logs (in debug builds)**
  - **Last check may never be reached if compiler does string pooling**

And Now Let’s Test It!



```cpp
void f()
{
    LOG("I'm here");
    LOG("I'm here too");
}

int main()
{
#ifdef BUILD_FOR_ENCODING
    f();
#else
    std::cout << "And now back from encoded:\n";
    std::cout << GetLogMessage(55179853024920655) << '\n';
    std::cout << GetLogMessage(13529717290104665) << '\n';
#endif

    return 0;
}
```

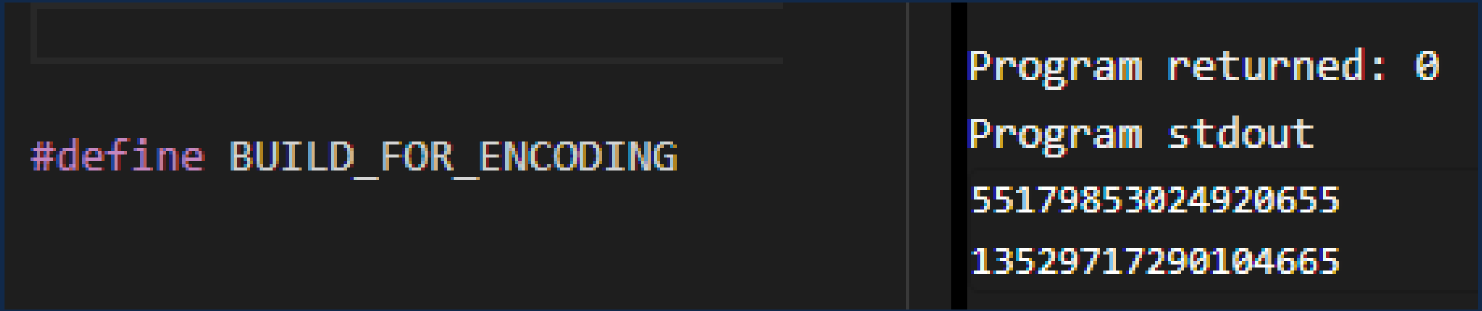
And Now Let’s Test It!



```cpp
void f()
{
    LOG("I'm here");
    LOG("I'm here too");
}

int main()
{
#ifdef BUILD_FOR_ENCODING
    f();
#else
    std::cout << "And now back from encoded:\n";
    std::cout << GetLogMessage(55179853024920655) << '\n';
    std::cout << GetLogMessage(13529717290104665) << '\n';
#endif

    return 0;
}
```

Where did these two hash values come from?

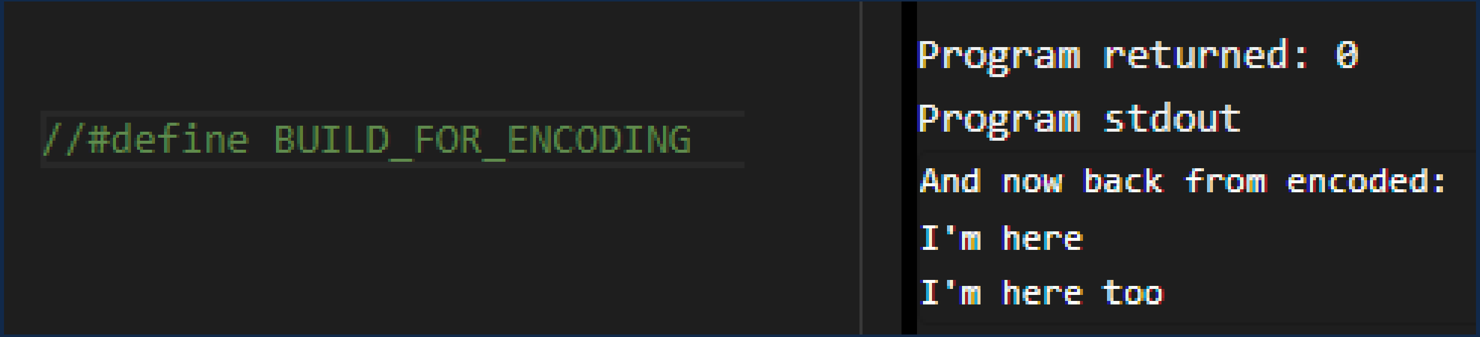
This Is Where:



```
#define BUILD_FOR_ENCODING
```

```
Program returned: 0
Program stdout
55179853024920655
13529717290104665
```

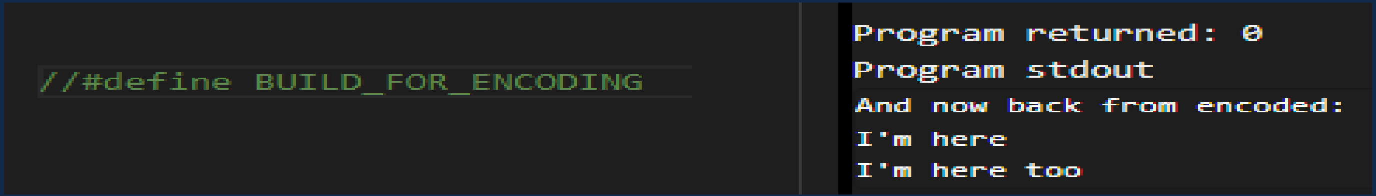
And Now Back to Decoder Mode:



```
//#define BUILD_FOR_ENCODING
```

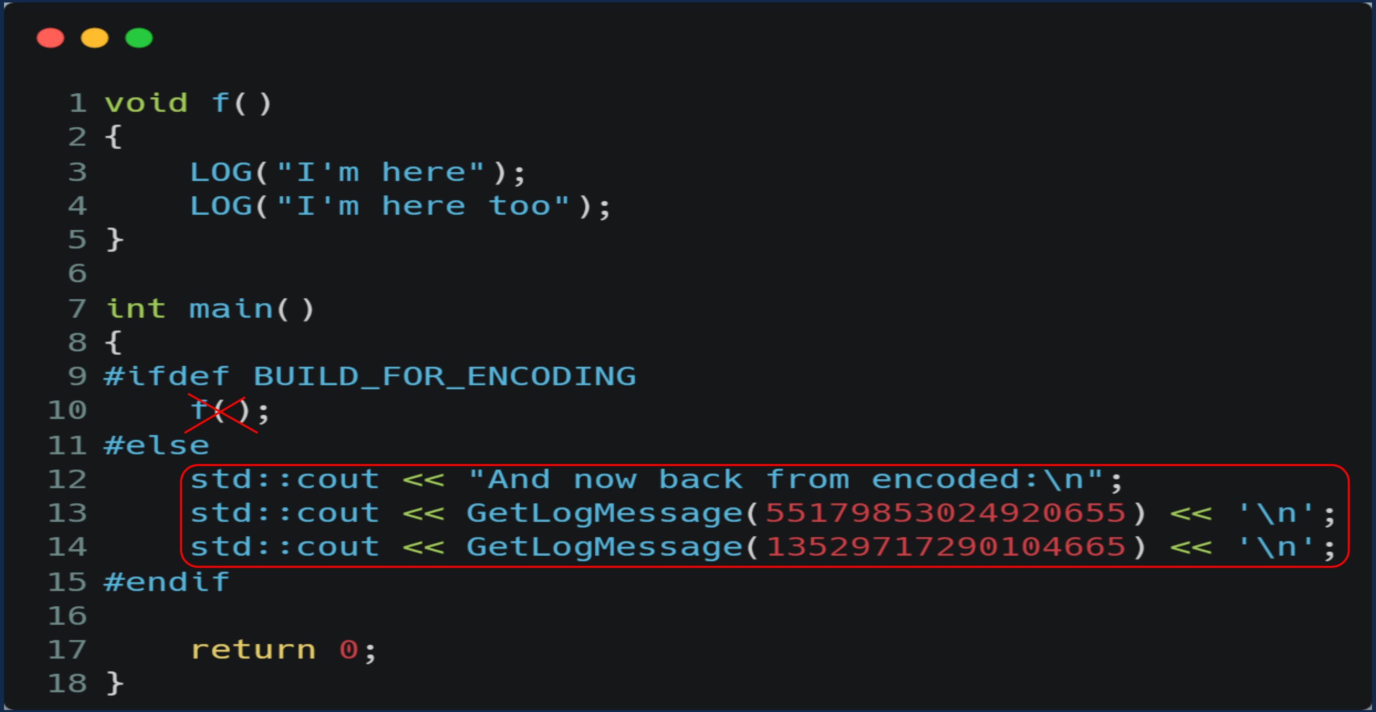
```
Program returned: 0
Program stdout
And now back from encoded:
I'm here
I'm here too
```

f() Is Not Called, Log Strings Materialize "Out of Nowhere"



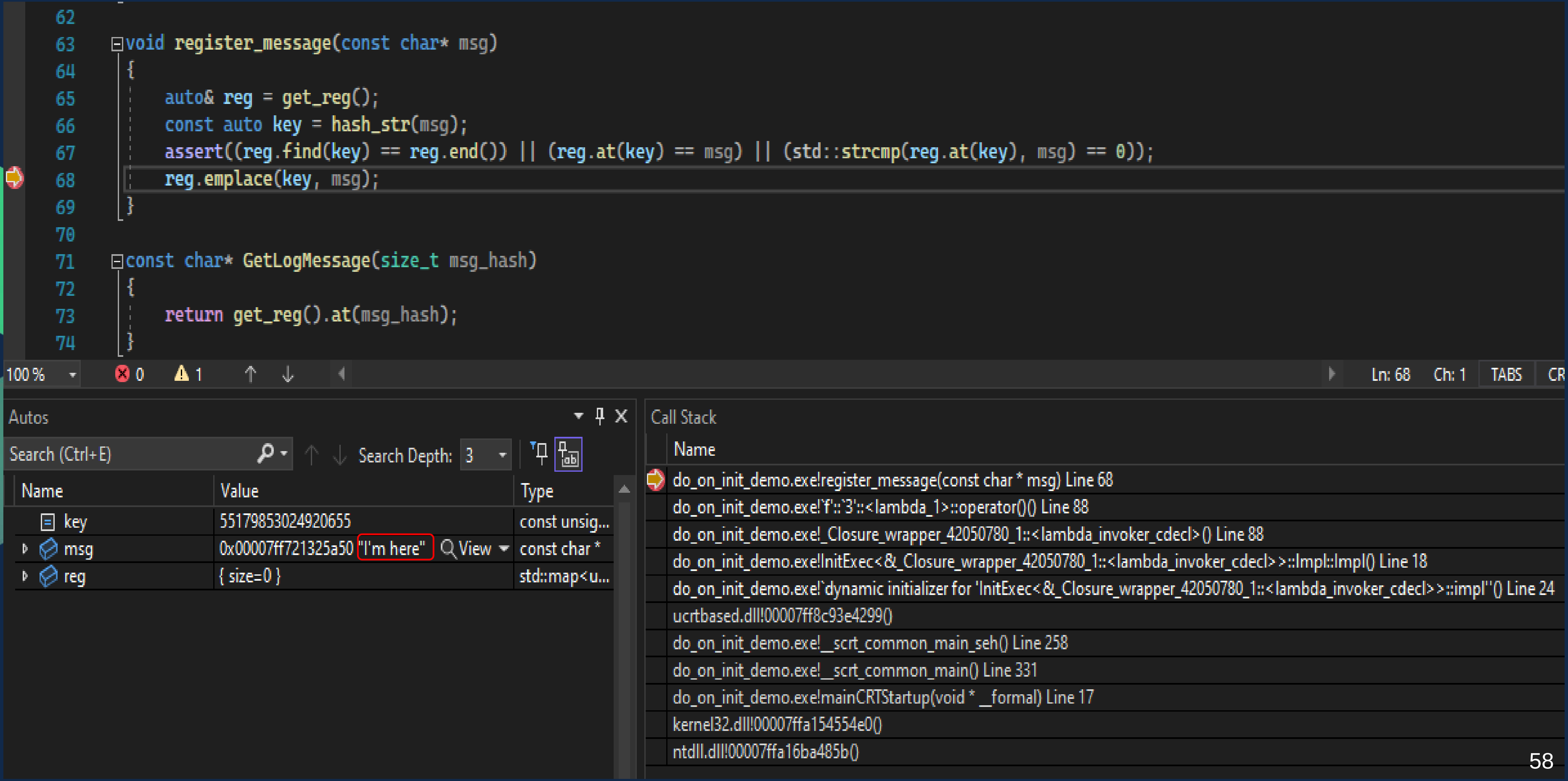
```
//#define BUILD_FOR_ENCODING
```

```
Program returned: 0
Program stdout
And now back from encoded:
I'm here
I'm here too
```



```
void f()
{
    LOG("I'm here");
    LOG("I'm here too");
}

int main()
{
#ifdef BUILD_FOR_ENCODING
    ~~f(~~);
#else
    std::cout << "And now back from encoded:\n";
    std::cout << GetLogMessage(55179853024920655) << '\n';
    std::cout << GetLogMessage(13529717290104665) << '\n';
#endif

    return 0;
}
```

“Magic” Call Stack



```
62
63  void register_message(const char* msg)
64  {
65      auto& reg = get_reg();
66      const auto key = hash_str(msg);
67      assert((reg.find(key) == reg.end()) || (reg.at(key) == msg) || (std::strcmp(reg.at(key), msg) == 0));
68      reg.emplace(key, msg);
69  }
70
71  const char* GetLogMessage(size_t msg_hash)
72  {
73      return get_reg().at(msg_hash);
74  }
```

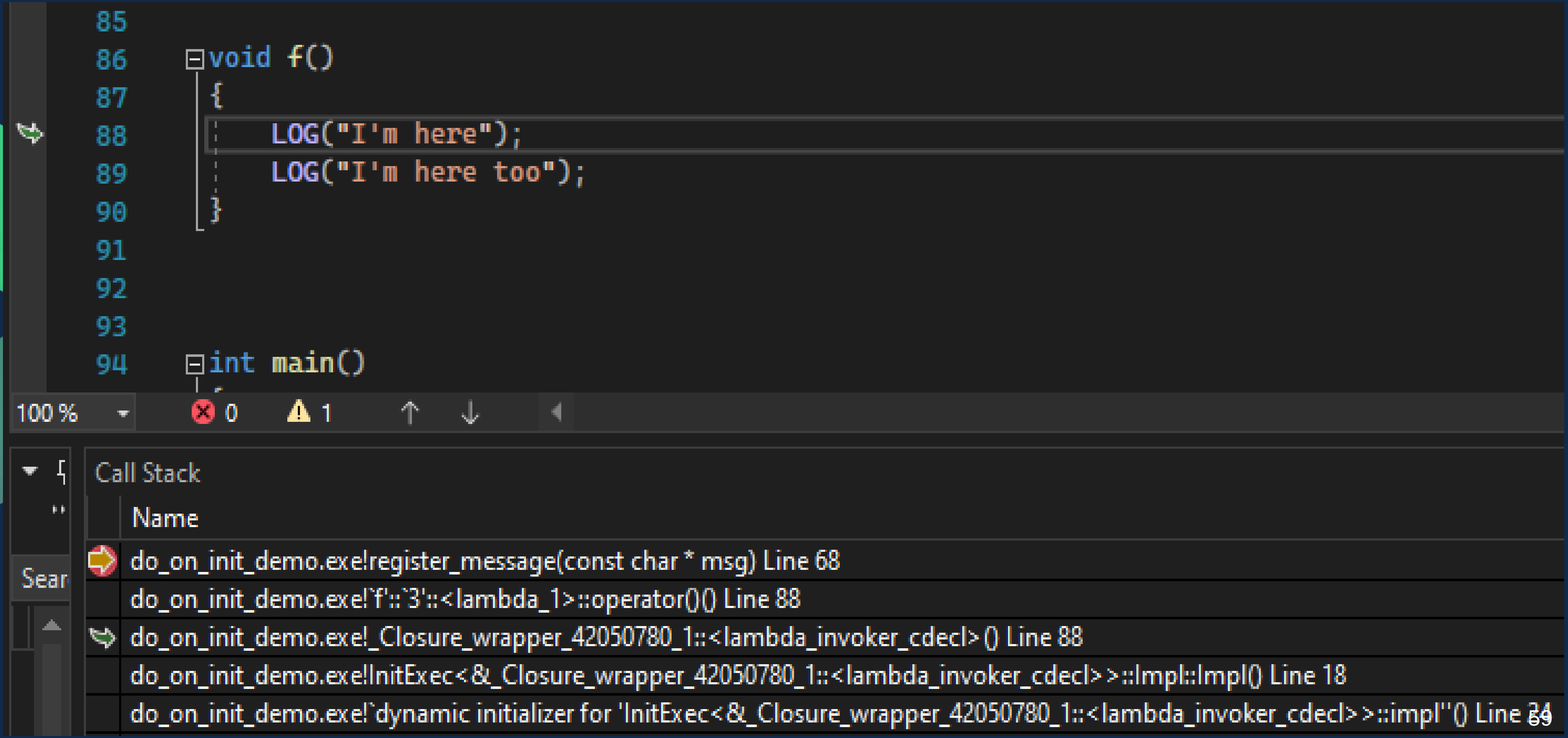
Autos

Name	Value	Type
key	55179853024920655	const unsig...
msg	0x00007ff721325a50 "I'm here" View	const char *
reg	{ size=0 }	std::map<u...

Call Stack

```
Name
do_on_init_demo.exe!register_message(const char * msg) Line 68
do_on_init_demo.exe!`f'::`3'::<lambda_1>::operator()() Line 88
do_on_init_demo.exe!_Closure_wrapper_42050780_1::<lambda_invoker_cdecl>() Line 88
do_on_init_demo.exe!InitExec<&_Closure_wrapper_42050780_1::<lambda_invoker_cdecl>>::Impl::Impl() Line 18
do_on_init_demo.exe!`dynamic initializer for 'InitExec<&_Closure_wrapper_42050780_1::<lambda_invoker_cdecl>>::impl''() Line 24
ucrtbased.dll!00007ff8c93e4299()
do_on_init_demo.exe!__scrt_common_main_seh() Line 258
do_on_init_demo.exe!__scrt_common_main() Line 331
do_on_init_demo.exe!mainCRTStartup(void * __formal) Line 17
kernel32.dll!00007ffa154554e0()
ntdll.dll!00007ffa16ba485b()
```

Which Log Is It?
Just Inspect the Call Stack!

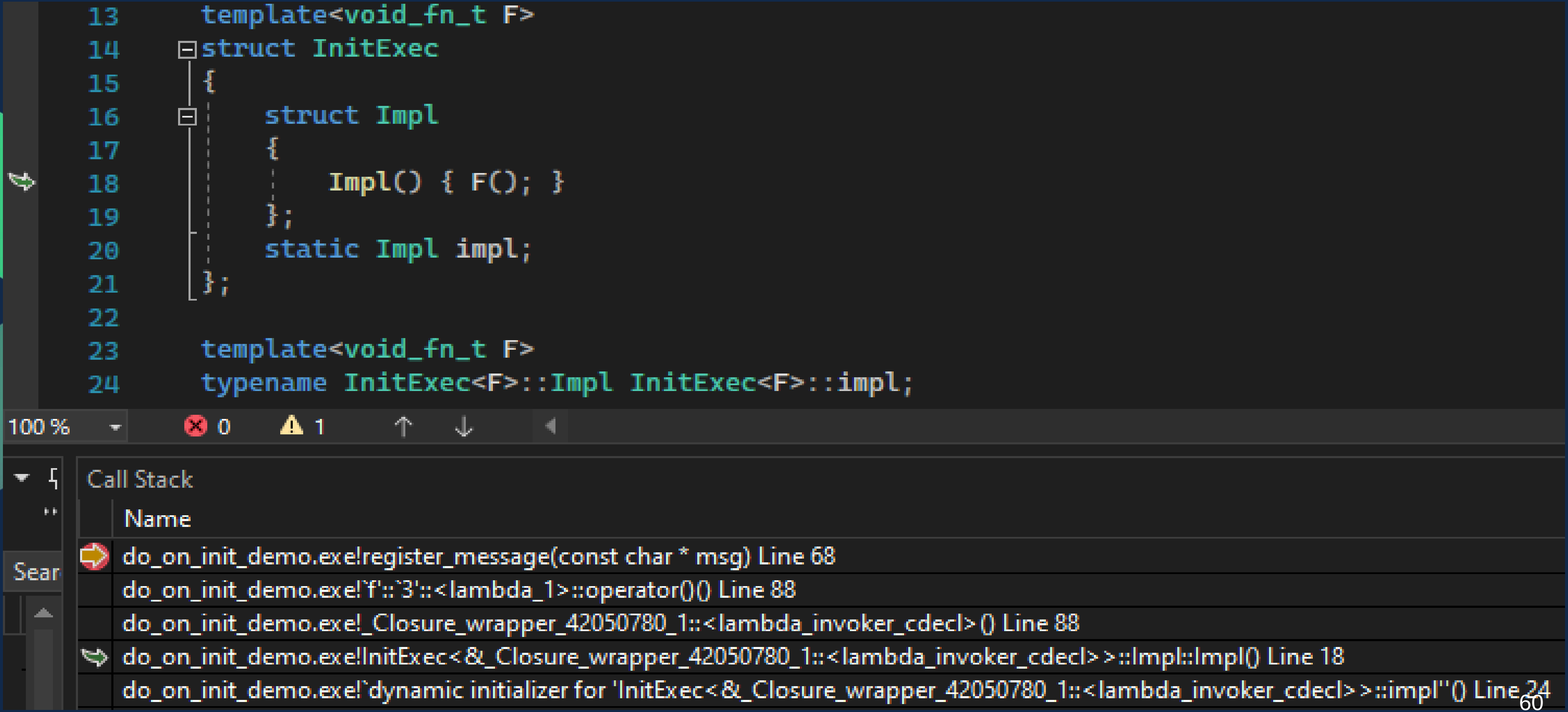


```
85
86  void f()
87  {
88      LOG("I'm here");
89      LOG("I'm here too");
90  }
91
92
93
94  int main()
```

Call Stack

Name
do_on_init_demo.exe!register_message(const char * msg) Line 68
do_on_init_demo.exe!`f'::`3'::<lambda_1>::operator()() Line 88
do_on_init_demo.exe!_Closure_wrapper_42050780_1::<lambda_invoker_cdecl>() Line 88
do_on_init_demo.exe!InitExec<&_Closure_wrapper_42050780_1::<lambda_invoker_cdecl>>::Impl::Impl() Line 18
do_on_init_demo.exe!`dynamic initializer for 'InitExec<&_Closure_wrapper_42050780_1::<lambda_invoker_cdecl>>::impl''() Line 24

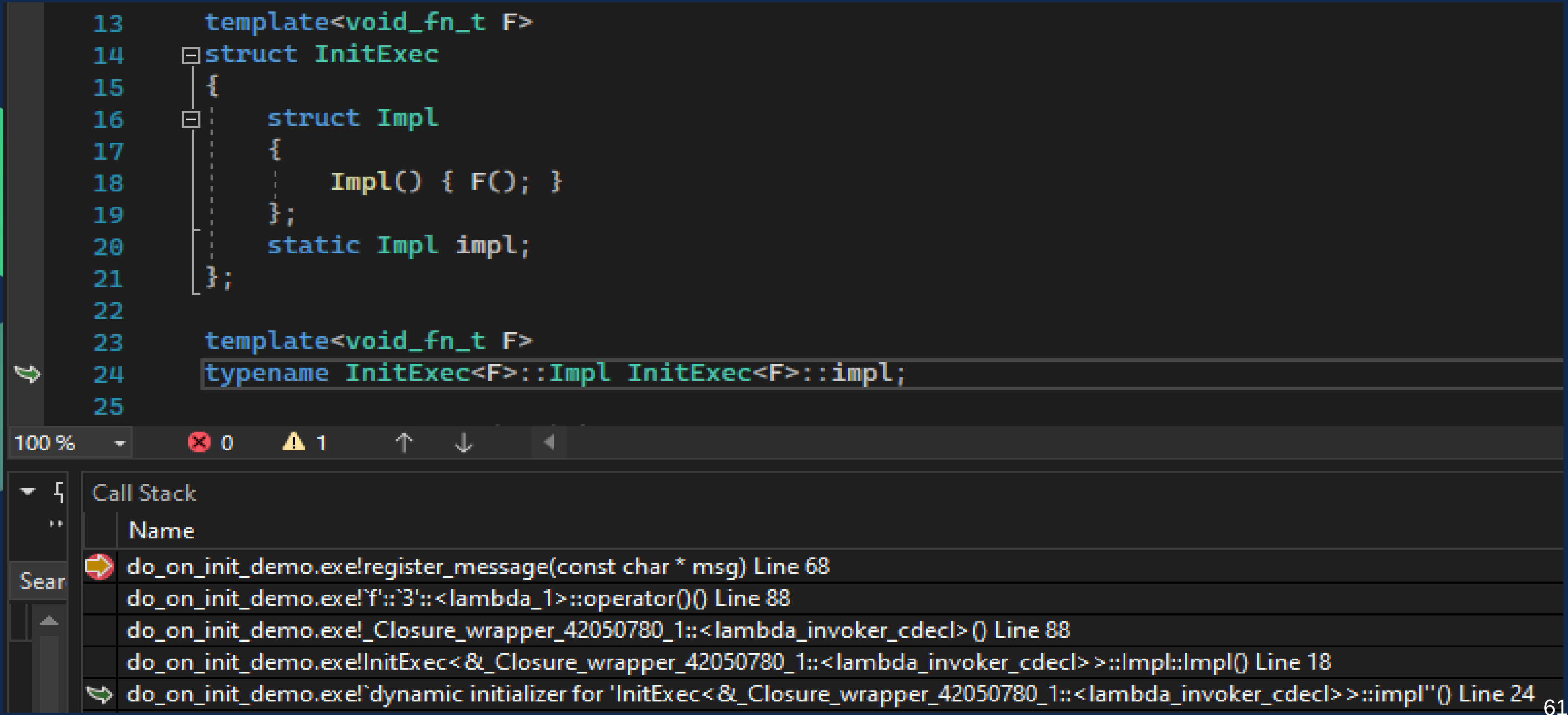
Next up in the Call Stack:
Nested Class Constructor Call



```
13      template<void_fn_t F>
14      struct InitExec
15      {
16          struct Impl
17          {
18              Impl() { F(); }
19          };
20          static Impl impl;
21      };
22
23      template<void_fn_t F>
24      typename InitExec<F>::Impl InitExec<F>::impl;
```

```
do_on_init_demo.exe!register_message(const char * msg) Line 68
do_on_init_demo.exe!`f'::`3'::<lambda_1>::operator()() Line 88
do_on_init_demo.exe!_Closure_wrapper_42050780_1::<lambda_invoker_cdecl>() Line 88
do_on_init_demo.exe!InitExec<&_Closure_wrapper_42050780_1::<lambda_invoker_cdecl>>::Impl::Impl() Line 18
do_on_init_demo.exe!`dynamic initializer for 'InitExec<&_Closure_wrapper_42050780_1::<lambda_invoker_cdecl>>::impl''() Line 24
```

And Last but Not Least:
The Static Member Global Initialization



```
13      template<void_fn_t F>
14      struct InitExec
15      {
16          struct Impl
17          {
18              Impl() { F(); }
19          };
20          static Impl impl;
21      };
22
23      template<void_fn_t F>
24      typename InitExec<F>::Impl InitExec<F>::impl;
25
```

```
do_on_init_demo.exe!register_message(const char * msg) Line 68
do_on_init_demo.exe!`f'::`3'::<lambda_1>::operator()() Line 88
do_on_init_demo.exe!_Closure_wrapper_42050780_1::<lambda_invoker_cdecl>() Line 88
do_on_init_demo.exe!InitExec<&_Closure_wrapper_42050780_1::<lambda_invoker_cdecl> >::Impl::Impl() Line 18
do_on_init_demo.exe!`dynamic initializer for 'InitExec<&_Closure_wrapper_42050780_1::<lambda_invoker_cdecl> >::impl''() Line 24
```

Questions So Far?

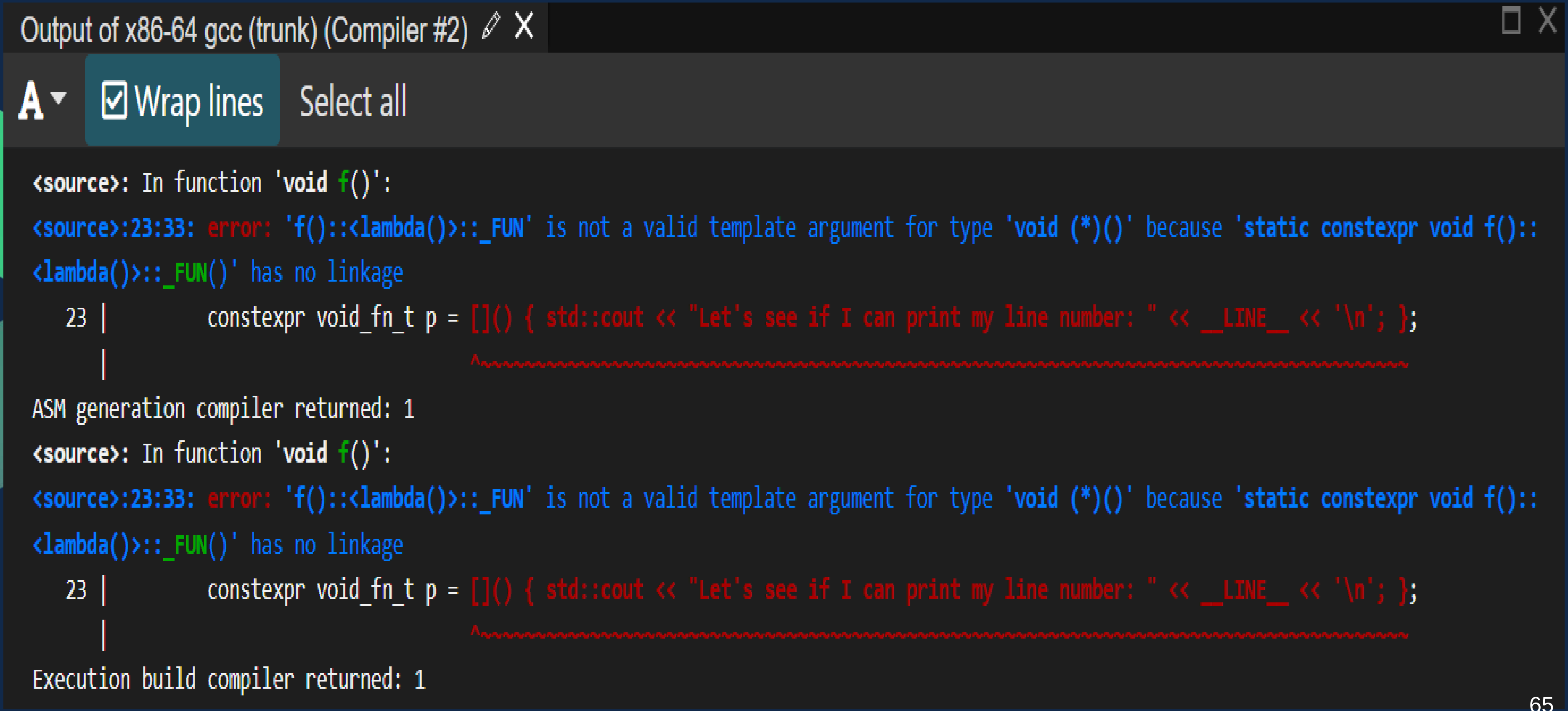
Demo Time!

https://github.com/cppal/hashed_logger

Special Circumstances Which Made This Possible

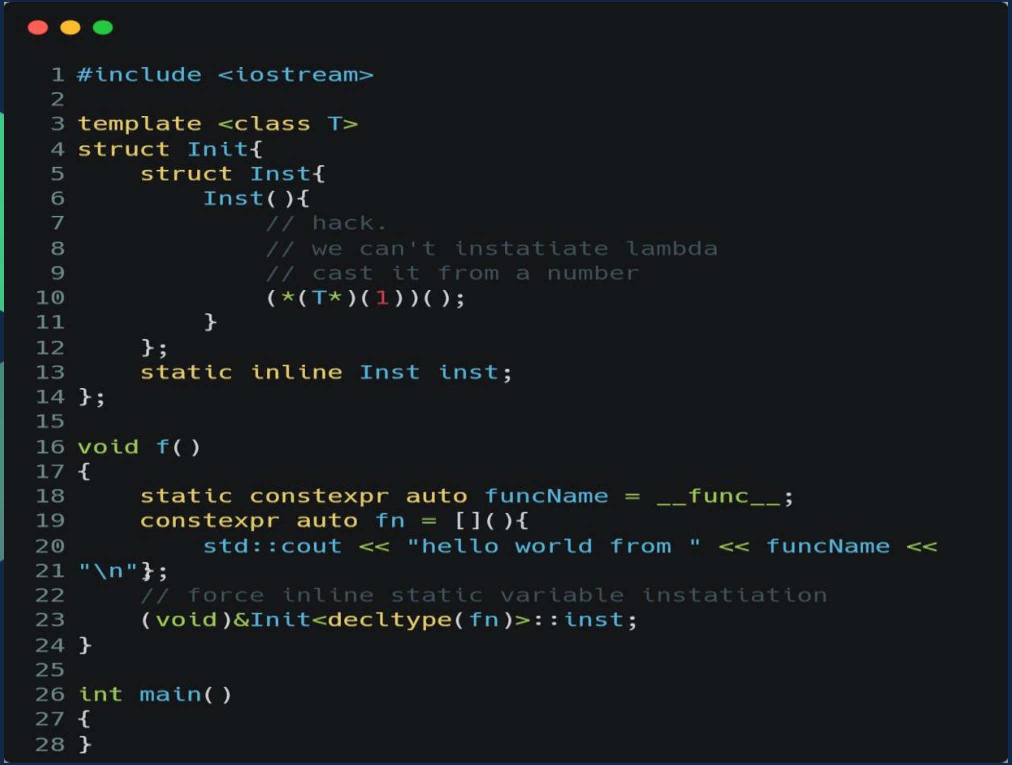
- A need that presented itself.
- This “hack” just recently happened to become possible in C++17.
- I didn’t know locally defined classes can’t have static data members.
- I happened to try this out first with the right compiler (msvc)…
    - gcc - can’t compile this at all (more on the next slide)
    - clang - segfaulted due to the dangers of the global init context…

Lucky I Didn’t Try This First in Gcc…

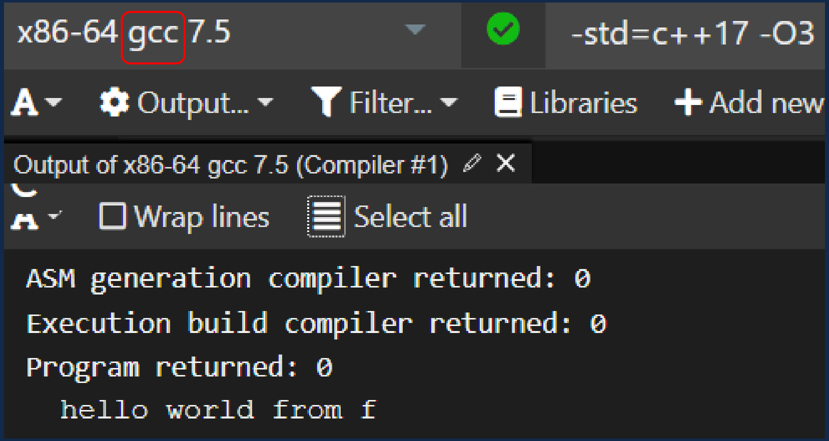


```
<source>: In function 'void f()':
<source>:23:33: error: 'f()::<lambda()>::_FUN' is not a valid template argument for type 'void (*)()' because 'static constexpr void f()::<lambda()>::_FUN()' has no linkage
   23 |         constexpr void_fn_t p = []() { std::cout << "Let's see if I can print my line number: " << __LINE__ << '\n'; };
      |                                 ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
ASM generation compiler returned: 1
<source>: In function 'void f()':
<source>:23:33: error: 'f()::<lambda()>::_FUN' is not a valid template argument for type 'void (*)()' because 'static constexpr void f()::<lambda()>::_FUN()' has no linkage
   23 |         constexpr void_fn_t p = []() { std::cout << "Let's see if I can print my line number: " << __LINE__ << '\n'; };
      |                                 ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Execution build compiler returned: 1
```

… Or at Least Not Without Some Hacks Even Crazier Than Mine



```cpp
#include <iostream>

template <class T>
struct Init{
    struct Inst{
        Inst(){
            // hack.
            // we can't instatiate lambda
            // cast it from a number
            (*(T*)(1))();
        }
    };
    static inline Inst inst;
};

void f()
{
    static constexpr auto funcName = __func__;
    constexpr auto fn = [](){
        std::cout << "hello world from " << funcName <<
"\n"};
    // force inline static variable instatiation
    (void)&Init<decltype(fn)>::inst;
}

int main()
{
}
```



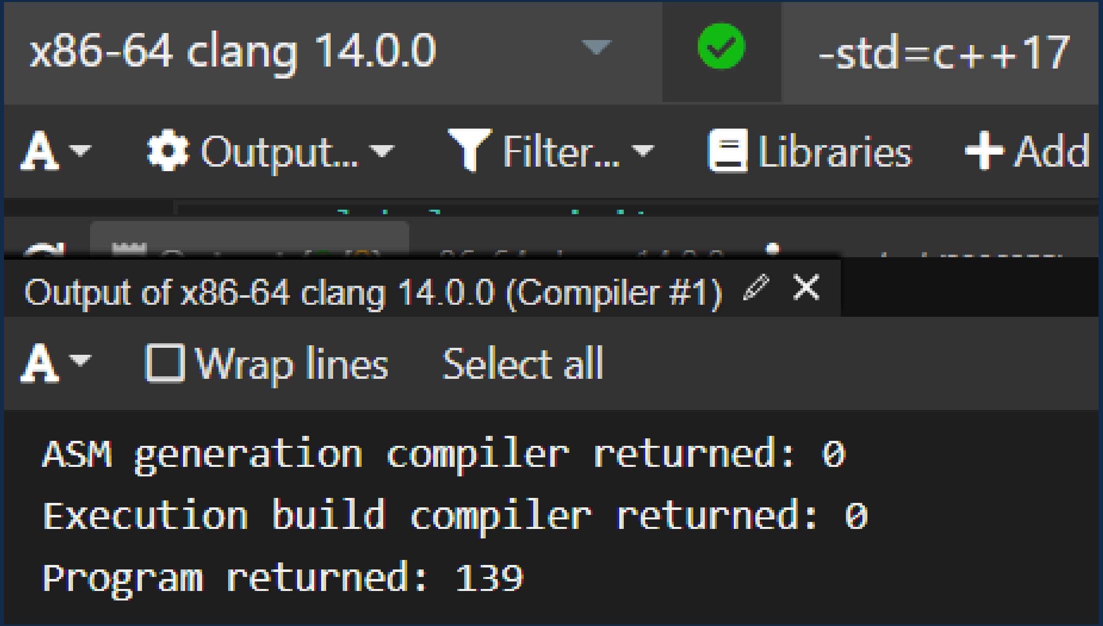
```
ASM generation compiler returned: 0
Execution build compiler returned: 0
Program returned: 0
  hello world from f
```

*Contributed by Alexander Vaisman

Digging a Bit Deeper Into Gcc

- https://gcc.gnu.org/bugzilla/show_bug.cgi?id=83258
  - **Bug 83258 - Rejecting function pointer non-type template parameter without linkage**
- **https://gcc.gnu.org/bugzilla/show_bug.cgi?id=92320**
  - **"Generally speaking it seems that GCC is perfectly happy instantiating a template with a constexpr (as you would hope) and with a constexpr function pointer even, but only if that function pointer derives from a free function."** (Joshua Leahy)
- Gcc also has __attribute__((__used__, section(".init_array") as a vendor-specific extension. Not sure that works on all platforms. (Kudos Erez Strauss for pointing me to this)

Lucky I Didn't Try This First in Clang…



```
ASM generation compiler returned: 0
Execution build compiler returned: 0
Program returned: 139
```

As It Turns Out, I Was Doing It Wrong

```
DO_ON_INIT( std::cout <<
            "Let's see if I can print my line number: "
            << __LINE__ << '\n'; );
```

```
DO_ON_INIT( printf( "Let's see if I can print my line
number: %d\n", __LINE__ ) );
```

***Figured out thanks to Alexander Vaisman**

Recap

- Used constexpr hash function to obfuscate log messages
- DO_ON_INIT implementation
  - Class templated on constexpr function pointer NTTP
  - Executes NTTP function via constructor of nested class
  - Invoked during construction of nested class static instance
  - Template class is instantiated with local lambda
  - Via constexpr function pointer
  - Forced into ODR use by (void)
- DO_ON_INIT is used to map text hash values back to original texts

Analysis

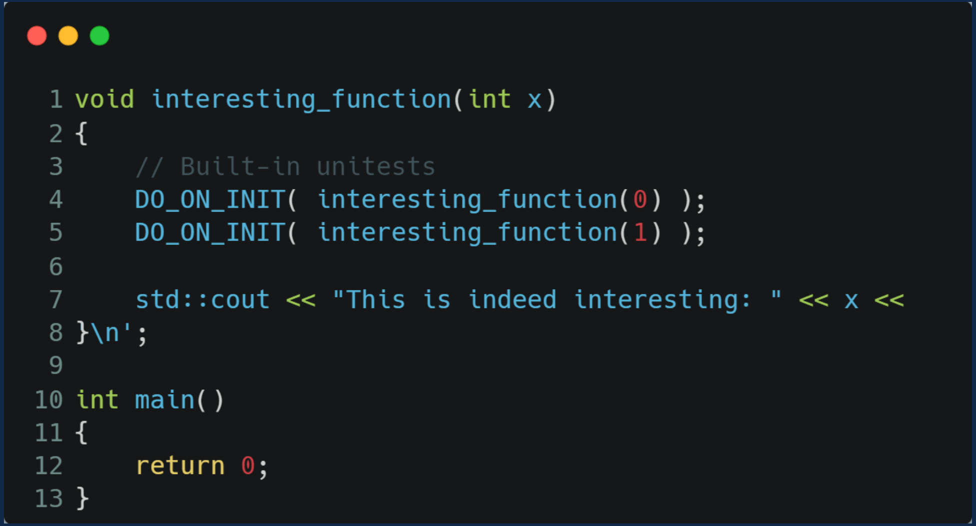
- Main drawback - be careful with this in production code
    - Not on all compilers (gcc in particular)
    - May encounter compiler limitations
        - Perhaps even UB?
- But it can be great for internal tools (e.g. log decoder)
- Secondary drawback - this technique requires macros
- Be careful what you do with DO_ON_INIT (cout as cautionary tale)

Analysis - Performance Impact

- No serious performance/memory footprint
  - Production code may actual benefit on both counts
  - Decoding tool has proven small and fast (on our 200+ logs)
- Impact on production code
  - Small hash values instead of full strings
  - May need to be converted back to strings if warranted by underlying logger - but those can be cached with statics
  - Impact on build times should be negligible - depending on the hash function

What Else Could DO_ON_INIT Be Used For?

- Default initial API call - probably not the best idea until we're sure we can trust DO_ON_INIT in production code
- Built-in unitests:



```
void interesting_function(int x)
{
    // Built-in unitests
    DO_ON_INIT( interesting_function(0) );
    DO_ON_INIT( interesting_function(1) );

    std::cout << "This is indeed interesting: " << x <<
}\n';

int main()
{
    return 0;
}
```

* Unitests can easily be left out of production code via #ifdef

Live, Log and Prosper 🖖
Thank You!

Get in touch:

- andrziss@gmail.com
- https://www.linkedin.com/in/andreizissu/

* Many thanks to Inbal Levi, Dafna Mordechai
and other good people
for all the first timer advice!