The Power and Pain of Hidden Symbols

Jason Turner

2025

@lefticus

emptycrate.com/idocpp

Power and Pain of Hidden Symbols

Jason Turner

C++ Weekly

- Weekly videos since March, 2016
- 115k+ subscribers, 455+ weeks straight

https://www.youtube.com/@cppweekly

Jason Turner

- Author
  - C++ Best Practices, C++23 Best Practices
  - OpCode, Copy and Reference, Object Lifetime Puzzlers
  - https://amzn.to/3xWh8Ox
  - https://leanpub.com/u/jason_turner

Jason Turner

- Developer
  - https://cppbestpractices.com
  - https://github.com/lefticus
  - https://github.com/cpp-best-practices
- Microsoft MVP for C++, 2015-present

Jason Turner - Training

https://articles.emptycrate.com/training.html

How to get my training:

1. Have me come to your company on-site for dynamic customized training where you already are - generally the most economical option for groups (CA, DE, NL, RO, CZ, JP, US, PL, SE, …)
2. Come to a conference workshop
   - C++ On Sea (Folkestone, UK, Late June)
   - CppCon (Aurora, CO, US, ~Sept)
   - NDC TechTown (Kongsberg, NO, ~Sept)
   - And possibly others

About my Talks

- Avoid sitting in the back
- Please interrupt and ask questions, yell things out, I'll repeat it for the room
- This is approximately how my training days look, as interactive as reasonable

Workshops!

- C++ On Sea
- NDC Tech Town
- CppCon

Having a good conference?

Feeling inspired?

Reflection

```
namespace lefticus::interface {
int myfloor(double d) { return static_cast<int>(d); }
void print(int v) { std::cout << "the value is: " << v; }
}  // namespace lefticus::interface

int main() {
  lefticus::cons_expr evaluator;

  // auto bind all members of the namespace with the scripting engine
  bind<^^lefticus::interface>(evaluator);

  evaluator.evaluate("(print (myfloor (+ 3.2 13.9)))");
}
```

https://godbolt.org/z/65fYej37d

https://godbolt.org/z/6Y17EG984

Power and Pain of Hidden Symbols

Power and Pain of Hidden Symbols

- Who uses Visual Studio (cl.exe, not VS Code)?

Power and Pain of Hidden Symbols

- Who uses Visual Studio (cl.exe, not VS Code)?
- Who uses GCC and/or Clang?

Power and Pain of Hidden Symbols

- Who uses Visual Studio (cl.exe, not VS Code)?
- Who uses GCC and/or Clang?
- Who writes shared libraries (.so, .dylib, .dll?)

Power and Pain of Hidden Symbols

- Who uses Visual Studio (cl.exe, not VS Code)?
- Who uses GCC and/or Clang?
- Who writes shared libraries (.so, .dylib, .dll?)
- Who uses CMake?

Power and Pain of Hidden Symbols

- Who uses Visual Studio (cl.exe, not VS Code)?
- Who uses GCC and/or Clang?
- Who writes shared libraries (.so, .dylib, .dll?)
- Who uses CMake?
- Who can share something about the types of shared libraries they are creating?

Power and Pain of Hidden Symbols

- Who uses Visual Studio (cl.exe, not VS Code)?
- Who uses GCC and/or Clang?
- Who writes shared libraries (.so, .dylib, .dll?)
- Who uses CMake?
- Who can share something about the types of shared libraries they are creating?
- Do you rely on ABI and/or API compatibility?

Power and Pain of Hidden Symbols

- Who uses Visual Studio (cl.exe, not VS Code)?
- Who uses GCC and/or Clang?
- Who writes shared libraries (.so, .dylib, .dll?)
- Who uses CMake?
- Who can share something about the types of shared libraries they are creating?
- Do you rely on ABI and/or API compatibility?
- Do you do anything to check or enforce your assumptions (abidiff?)

Power and Pain of Hidden Symbols

This talk lives at the intersection between what the standard requires or allows and what linkers and operating systems expect.

What Does It Mean For a Function To Be Inlined?

What Does It Mean For a Function To Be Inlined?

```
[[gnu::noinline]] int f1() {
  return 42;
}

int f2(int x) {
  return f1() + x;
}
```

https://godbolt.org/z/9oP7q9xxE

```
f1():
        mov     eax, 42
        ret
f2(int):
        call    f1()
        add     eax, edi
        ret
```

What Does It Mean For a Function To Be Inlined?

```
int f1() {
  return 42;
}

int f2(int x) {
  return f1() + x;
}
```

https://godbolt.org/z/Ths3PeMav

```
f1():
        mov     eax, 42
        ret
f2(int):
        lea     eax, [rdi+42]
        ret
```

Sometimes we think of inlining as increasing binary size, but it often actually makes the binary smaller, like this.

Will `work` Be Inlined?

```
1  int work(int x) {
2    return x;
3  }
4
5  int main(const int argc, const char *[])
6  {
7    return work(argc);
8  }
```

https://godbolt.org/z/xEdrzjcoW

Will `work` Be Inlined?

```
1  int work(int x) {
2    return x;
3  }
4
5  int main(const int argc, const char *[])
6  {
7    return work(argc);
8  }
```

https://godbolt.org/z/xEdrzjcoW

Trick Question!

What decides if `work` is inlined?

```
int work(int x) {
  return x;
}

int main(const int argc, const char *[])
{
  return work(argc);
}
```

What decides if `work` is inlined?

```
int work(int x) {
  return x;
}

int main(const int argc, const char *[])
{
  return work(argc);
}
```

- Optimizer settings

What decides if `work` is inlined?

```
1  int work(int x) {
2    return x;
3  }
4
5  int main(const int argc, const char *[])
6  {
7    return work(argc);
8  }
```

- Optimizer settings
- Code generation settings

What decides if `work` is inlined?

```
1  int work(int x) {
2    return x;
3  }
4
5  int main(const int argc, const char *[])
6  {
7    return work(argc);
8  }
```

- Optimizer settings
- Code generation settings
- Linker settings…?

What does “code generation settings” mean?

Check with `-fPIC`, `-O3`, etc.

```
1  int work(int x) {
2    return x;
3  }
4
5  int main(const int argc, const char *[])
6  {
7    return work(argc);
8  }
```

Semantic Interposition

Simply: If a symbol is public, and your code is compiled as a shared object then:

- The symbol will not be inlined… usually.
- The symbol will be emitted into the binary

This is so that the symbol can be swapped out with another at run link time.

Semantic Interposition

Clang cheats and GCC has several other flags controlling things

- `-fno-semantic-interposition`
- `-fvisibility-inlines-hidden`
- etc.

Notes:

Semantic Interposition

Clang cheats and GCC has several other flags controlling things

- `-fno-semantic-interposition`
- `-fvisibility-inlines-hidden`
- etc.

Notes:

- This appears to be an ELF rule, not a related to the C++ or C standards.

Semantic Interposition

Clang cheats and GCC has several other flags controlling things

- `-fno-semantic-interposition`
- `-fvisibility-inlines-hidden`
- etc.

Notes:

- This appears to be an ELF rule, not a related to the C++ or C standards.
- ⚠️ you might be asking for ODR violations if a public symbol is also inlined, so you end up with 2 different versions called.

Is `work` inlined? (`inline`, what decides?)

```
1  inline int work(int x) {
2    return x;
3  }
4
5  int main(const int argc, const char *[])
6  {
7    return work(argc);
8  }
```

https://godbolt.org/z/eGbh4vG3x

Is `work` inlined? (`inline`, what decides?)

```
1  inline int work(int x) {
2    return x;
3  }
4
5  int main(const int argc, const char *[])
6  {
7    return work(argc);
8  }
```

https://godbolt.org/z/eGbh4vG3x

- Optimizer settings

Is `work` inlined? (`static`, what decides?)

```
1  static int work(int x) {
2    return x;
3  }
4
5  int main(const int argc, const char *[])
6  {
7    return work(argc);
8  }
```

https://godbolt.org/z/Y6a44hr1h

Is `work` inlined? (`static`, what decides?)

```
static int work(int x) {
  return x;
}

int main(const int argc, const char *[])
{
  return work(argc);
}
```

https://godbolt.org/z/Y6a44hr1h

- Optimizer settings

Is `work` inlined? (anonymous namespace, what decides?)

```
1  namespace {
2    int work(int x) {
3      return x;
4    }
5  }
6
7  int main(const int argc, const char *[])
8  {
9    return work(argc);
10 }
```

https://godbolt.org/z/v5EP7bE4e

Is `work` inlined? (anonymous namespace, what decides?)

```
1  namespace {
2    int work(int x) {
3      return x;
4    }
5  }
6
7  int main(const int argc, const char *[])
8  {
9    return work(argc);
10 }
```

https://godbolt.org/z/v5EP7bE4e

- Optimizer settings

`inline` vs `static`

- `inline`: definitions are merged
- `static`: each translation gets its own definition
- `inline`: definition must be available
- `static` || `inline`: compiler won't emit a definition unless it needs to

NOTE: this is why I say to `inline` large global `constexpr` `static` data

Are we going to make all of our shared library code `inline`, `static`, or put it in an anonymous `namespace`?

No, that kind of goes against the point of shared libraries!

Is `get_data()` inlined? (what decides?)

```
1  struct S {
2    int get_data() { return 42; }
3  };
4
5  int main()
6  {
7    S s;
8    return s.get_data();
9  }
```

https://godbolt.org/z/1doq7GGT3

Is `get_data()` inlined? (what decides?)

```
1  struct S {
2    int get_data() { return 42; }
3  };
4
5  int main()
6  {
7    S s;
8    return s.get_data();
9  }
```

https://godbolt.org/z/1doq7GGT3

- Optimizer settings

Is `get_data()` inlined? (what decides?)

```
1  struct S {
2    int get_data() const;
3  };
4
5  int main()
6  {
7    S s;
8    return s.get_data();
9  }
10
11 int S::get_data() const { return 42; }
```

https://godbolt.org/z/61nddn3GP

Is `get_data()` inlined? (what decides?)

```
1   struct S {
2     int get_data() const;
3   };
4
5   int main()
6   {
7     S s;
8     return s.get_data();
9   }
10
11  int S::get_data() const { return 42; }
```

https://godbolt.org/z/61nddn3GP

- Optimizer settings

Is `get_data()` inlined? (what decides?)

```
1   struct S {
2     int get_data() const;
3   };
4
5   int main()
6   {
7     S s;
8     return s.get_data();
9   }
10
11  int S::get_data() const { return 42; }
```

https://godbolt.org/z/61nddn3GP

- Optimizer settings
- Code generation settings (function is no longer implicitly inline)

`static` vs `inline` vs anonymous namespace

```
// my.hpp file
#include <cstdio>

void do_work() {
  std::puts("Hello World!");
}
```

https://godbolt.org/z/1rEEdEMrd

What happens when I include this file in more than one .cpp file and link them together?

`static` vs `inline` vs anonymous namespace

```
// my.hpp file
#include <cstdio>

void do_work() {
  std::puts("Hello World!");
}
```

https://godbolt.org/z/1rEEdEMrd

What happens when I include this file in more than one .cpp file and link them together?

ODR violation (more on that later).

`static` vs `inline` vs anonymous namespace

```
// my.hpp file
#include <cstdio>

static void do_work() {
  std::puts("Hello World!");
}
```

https://godbolt.org/z/qsrso44nG

What happens when I include this file in more than one .cpp file and link them together?

`static` vs `inline` vs anonymous namespace

```
// my.hpp file
#include <cstdio>

static void do_work() {
  std::puts("Hello World!");
}
```

https://godbolt.org/z/qsrso44nG

What happens when I include this file in more than one .cpp file and link them together?

Each object file gets its own copy.

`static` vs `inline` vs anonymous namespace

```
// my.hpp file
#include <cstdio>

inline void do_work() {
  std::puts("Hello World!");
}
```

https://godbolt.org/z/z1anf6K8c

What happens when I include this file in more than one .cpp file and link them together?

`static` vs `inline` vs anonymous namespace

```
// my.hpp file
#include <cstdio>

inline void do_work() {
  std::puts("Hello World!");
}
```

https://godbolt.org/z/z1anf6K8c

What happens when I include this file in more than one .cpp file and link them together?

Copies across object files are merged

`static` vs `inline` vs anonymous namespace

```cpp
// my.hpp file
#include <cstdio>

namespace {
void do_work() {
  std::puts("Hello World!");
}
}
```

https://godbolt.org/z/T6xjedWTY

What happens when I include this file in more than one .cpp file and link them together?

`static` vs `inline` vs anonymous namespace

```cpp
// my.hpp file
#include <cstdio>

namespace {
void do_work() {
  std::puts("Hello World!");
}
}
```

https://godbolt.org/z/T6xjedWTY

What happens when I include this file in more than one .cpp file and link them together?

Each gets its own copy and ODR violations are impossible.

anonymous namespaces

From clang-tidy documentation

misc-use-anonymous-namespace

Finds instances of static functions or variables declared at global scope that could instead be moved into an anonymous namespace.

Anonymous namespaces are the "superior alternative" according to the C++ Standard. static was proposed for deprecation, but later un-deprecated to keep C compatibility [1]. static is an overloaded term with different meanings in different contexts, so it can create confusion.

anonymous namespaces

The following uses of static will not be diagnosed:

Functions or variables in header files, since anonymous namespaces in headers is considered an antipattern. Allowed header file extensions can be configured via the global option HeaderFileExtensions.

const or constexpr variables, since they already have implicit internal linkage in C++.

```
0000000000001143 <(anonymous namespace)::print()>:
    1143:       f3 0f 1e fa             endbr64
    1147:       55                      push   %rbp
    1148:       48 89 e5                mov    %rsp,%rbp
    114b:       48 8d 05 ba 0e 00 00    lea    0xeba(%rip),%rax # 200c <_fini+0xe9c>
    1152:       48 89 c7                mov    %rax,%rdi
    1155:       e8 f6 fe ff ff          call   1050 <puts@plt>
    115a:       90                      nop
    115b:       5d                      pop    %rbp
    115c:       c3                      ret

000000000000115d <go2()>:
    115d:       f3 0f 1e fa             endbr64
    1161:       55                      push   %rbp
    1162:       48 89 e5                mov    %rsp,%rbp
    1165:       e8 d9 ff ff ff          call   1143 <(anonymous namespace)::print()>
    116a:       90                      nop
    116b:       5d                      pop    %rbp
    116c:       c3                      ret
```

`inline namespace`

Really unrelated concept, allows stuff like this:

```
#include <iostream>

namespace A {
void print(int x) { std::cout << x << '\n'; }
  inline namespace B {
    void print(double x) { std::cout << x << '\n'; }
  }
}

int main() {
  A::print(42);
  A::print(1.3);
}
```

https://godbolt.org/z/K3EsnbsME

- The namespace name is still part of the symbol
- See also Ep 320 “Using `inline namespace` To Save Your ABI https://youtu.be/rUESOjhvLw0

The Point

```cpp
// awesomelibrary.hpp file
int get_value_impl();
int return_value();
```

```cpp
// library_file1.cpp file
int get_value_impl() { return 42; }

int return_value() {
  return get_value_impl(); // you really want this inlined, right?
}
```

https://godbolt.org/z/jzfdboeKe

```cpp
// library_file2.cpp file
int use_value() {
  return get_value_impl() * 10; // nice to have inlining!
}
```

https://godbolt.org/z/dfecnqhcr

- We could make `get_value_impl()` `inline`, but then we must provide the definition for everyone!
- Same for `static`. Each TU gets its own copy, so it must have an implementation available. This could also lead to code bloat

What we want is a clean way to specify private interfaces vs public interfaces and let the compiler decide what's best!

Windows Devs … what's missing here?

```
// awesomelibrary.hpp file
int get_value_impl();
int return_value();
```

```
// library_file1.cpp file
int get_value_impl() {
  return 42;
}

int return_value() {
  return get_value_impl(); // you really want this inlined, right?
}
```

https://godbolt.org/z/zGT914fGq

```
// library_file2.cpp file
int use_value() {
  return get_value_impl() * 10; // nice to have inlining!
}
```

https://godbolt.org/z/e4xaxMxTz

dllexport macro

```
1  #ifdef _WIN32
2    #ifdef MYLIB_EXPORTS // defined during library compilation
3      #define MYLIB_API __declspec(dllexport)
4    #else
5      #define MYLIB_API __declspec(dllimport)
6    #endif
7  #else
8    #define MYLIB_API
9  #endif
10
11 // Example function declaration
12 int get_value_impl();
13 MYLIB_API int return_value();
```

https://godbolt.org/z/P1PYbbEcE

`dllexport` macro

On Windows you `/DMYLIB_EXPORTS` during compilation of the .dll file to make sure the symbols are exported, then they default to import symbols with nothing defined.

```
#ifdef _WIN32
  #ifdef MYLIB_EXPORTS // defined during library compilation
    #define MYLIB_API __declspec(dllexport)
  #else
    #define MYLIB_API __declspec(dllimport)
  #endif
#else
  #define MYLIB_API
#endif
```

https://godbolt.org/z/Y5Tze1Tfc

Now we just have to do what the Windows devs have always done…

The Plan

The Plan

1. Make symbols hidden by default

The Plan

1. Make symbols hidden by default
2. Export only the symbols we want to be part of our public interface

The Plan

1. Make symbols hidden by default
2. Export only the symbols we want to be part of our public interface
3. Profit! (from smaller binaries and increased inlining of code!)

GCC / Clang

- `-fvisibility=hidden`
- `[[gnu::visibility("default")]]`

```
1  #ifdef _WIN32
2    #ifdef MYLIB_EXPORTS // defined during library compilation
3      #define MYLIB_API __declspec(dllexport)
4    #else
5      #define MYLIB_API __declspec(dllimport)
6    #endif
7  #else
8    // no difference required for import vs export
9    #define MYLIB_API [[gnu::visibility("default")]]
10 #endif
11
12 int get_value_impl();
13 MYLIB_API int return_value();
```

https://godbolt.org/z/5aW4MP8o6

`-fvisility=hidden` + `visibilty("default")`

Niall Douglas: https://www.nedprod.com/programs/gccvisibility.html

https://gcc.gnu.org/wiki/Visibility

`-fvisility=hidden` + `visibilty("default")`

- Cleaner ABI - we only export what we want to

Niall Douglas: https://www.nedprod.com/programs/gccvisibility.html

https://gcc.gnu.org/wiki/Visibility

`-fvisility=hidden` + `visibilty("default")`

- Cleaner ABI - we only export what we want to
- Smaller binaries - symbols that don’t need to be emitted will not be emitted

Niall Douglas: https://www.nedprod.com/programs/gccvisibility.html

https://gcc.gnu.org/wiki/Visibility

`-fvisility=hidden` + `visibilty("default")`

- Cleaner ABI - we only export what we want to
- Smaller binaries - symbols that don’t need to be emitted will not be emitted
- Better inlining - the compiler knows which symbols don’t need to be dynamic and it will more freely inline them

Niall Douglas: https://www.nedprod.com/programs/gccvisibility.html

https://gcc.gnu.org/wiki/Visibility

That’s The Power, What’s the Pain?

Pain of Hidden Symbols

Simply: you cannot use symbols that have not been exported when creating the shared library!

Pain of Hidden Symbols

Simply: you cannot use symbols that have not been exported when creating the shared library!

- Is this a bad thing?

Pain of Hidden Symbols

Simply: you cannot use symbols that have not been exported when creating the shared library!

- Is this a bad thing?
- How does this affect testing?

Pain of Hidden Symbols

```cpp
// Mylib.hpp
struct ExceptionType {};

MYLIB_API void dowork();

// Mylib.cpp

void dowork() {
  throw ExceptionType{}; // surprise UB!
}
```

https://godbolt.org/z/n1Mbbo5Kv

make sure you export exceptions that cross DLL boundaries.

Pain of Hidden Symbols

When you test…

Pain of Hidden Symbols

When you test…

- Do you link to your shared library?

Pain of Hidden Symbols

When you test…

- Do you link to your shared library?
- Do you only test your public facing functions?

Pain of Hidden Symbols

When you test…

- Do you link to your shared library?
- Do you only test your public facing functions?
- Do you hate building your shared library twice?

Pain of Hidden Symbols

When you test…

- Do you link to your shared library?
- Do you only test your public facing functions?
- Do you hate building your shared library twice?

Link a static library of your project and use that for tests

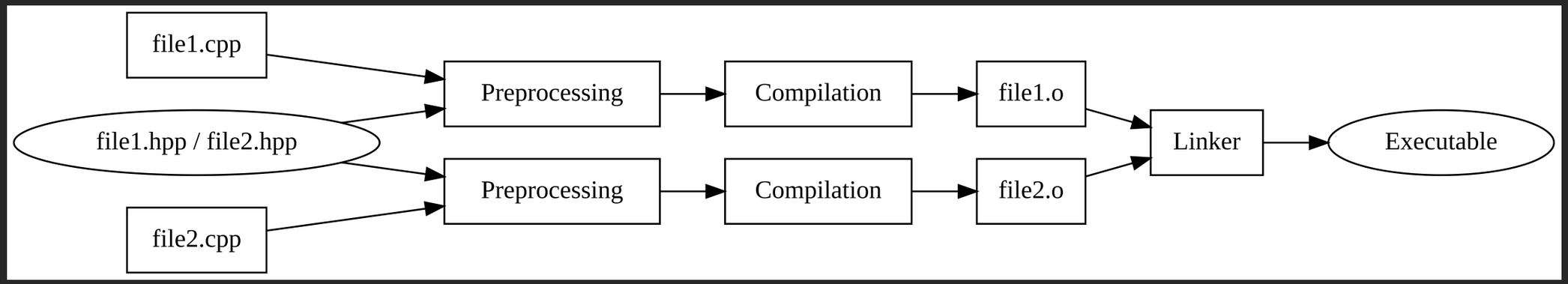
```
add_library(mylib STATIC mylib.cpp) # or internal?
add_library(mydll SHARED)
target_link_libraries(mydll mylib)
target_link_libraries(mytests mylib)
```

How does this interact with modules?

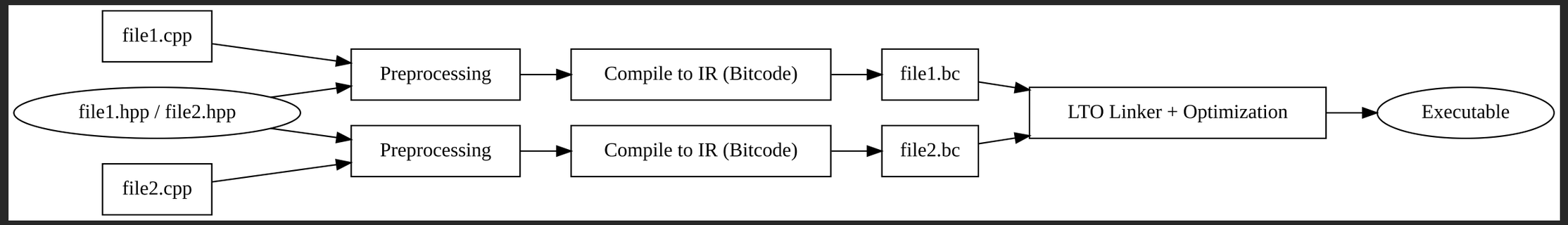
Anyone have insight here with modules + DLLs?

LTO vs Traditional Compilation

Traditional Compilation



LTO Compilation



Inlining With LTO

LTO allows for cross-translation unit inlining

```
// file1.hpp
int get_value();
```

```
// file1.cpp
int get_value() { return 42; }
```

```
// file2.hpp
int use_value();
```

```
// file1.cpp
#include "file1.hpp"
int use_value() { return 42; }
```

Bringing It All Together

Bringing It All Together

Bringing It All Together

- default hidden symbols

Bringing It All Together

- default hidden symbols
- export only the things you want public (minimize ABI surface)

Bringing It All Together

- default hidden symbols
- export only the things you want public (minimize ABI surface)
- export transitive dependencies also!

Bringing It All Together

- default hidden symbols
- export only the things you want public (minimize ABI surface)
- export transitive dependencies also!
- enable LTO

Case Study

Case Study

Original:

- 55004864
- 1:40

LTO:

- 50448528 (8.3% smaller)
- 1:40 (0% faster)

(this is an unexpected result)

Case Study

Original:

- 55004864
- 1:40

Hidden:

- 51614936 (6.2% smaller)
- 1:37 (3% faster)

(this was a surprising result)

Case Study

Original:

- 55004864
- 1:40

Hidden + LTO:

- 47247568 (14.1% smaller)
- 1:30 (10% faster)

(this is a clear winner)

Case Study

This code was already written for cross-platform MSVC, so we got 10% faster runs *with only 10 lines of changes*.

Before this revelation, LTO was deemed to be “not worth it” on this project because we gained no performance advantage.

Key Takeaways

Key Takeaways

- 💡 This mostly applies to shared libraries
- ✅ Enable `-fvisibility=hidden` to make symbols hidden by default
- ✅ Explicitly export symbols you want visible
- ✅ Limit the surface of your interfaces
- ✅ Enable LTO with your hidden symbols to get cross-translation unit inlining
- ⚠️ Make sure you properly export all dependent types
- 💡 Using CMake to do all this for you (example if we have time)

Let’s Talk About LTO

Who has tried to use LTO/IPO with a project?

Who has succeeded?

Who has failed?

Who knows what ODR is?

ODR (One Definition Rule)

[basic.def.odr]

> *No translation unit shall contain more than one definition of any definable item.*

(and basically the definitions across translation units must also match each other)

Aside on LTO

While creating an example for this project, I accidentally wrote code like this:

```cpp
// data_provider.hpp
std::span<std::string_view, 255> get_data();
```

```cpp
// data_provider.cpp
std::span<const std::string_view, 255> get_data() {
  return data;
}
```

https://godbolt.org/z/xc1frWKdb

This code appeared to work.

Then I compiled with `-flto`

ODR + LTO

```
data_provider.hpp:4:34: warning: ‘get_data’ violates the
  C++ One Definition Rule [-Wodr]
    4 | std::span<std::string_view, 255> get_data();
      |                                  ^
data_provider.cpp:13:40: note: return value type mismatch
   13 | std::span<const std::string_view, 255> get_data()
      |                                        ^
/usr/include/c++/14/span:102:11: note: type name
  ‘span<basic_string_view<char, char_traits<char> > const, 255ul>’
  should match type name ‘span<basic_string_view<char, char_traits<char> >, 255ul>’
  102 |     class span
      |           ^
data_provider.cpp:13:40: note: ‘get_data’ was previously declared here
   13 | std::span<const std::string_view, 255> get_data()
      |                                        ^
data_provider.cpp:13:40: note: code may be misoptimized unless
  ‘-fno-strict-aliasing’ is used
```

✓ Utilize LTO to find bugs

✓ Utilize LTO to find bugs

Even if you don’t ship with LTO, put an LTO build on your CI to find additional bugs.

✓ Utilize LTO to find bugs

Even if you don’t ship with LTO, put an LTO build on your CI to find additional bugs.

If you cannot compile with LTO and modern tooling, you likely have ODR / UB in your project!

What About `constexpr`?

What About `constexpr`?

```
// data_provider.hpp
constexpr auto data = make_data();
```

```
// consumer_1.cpp
#include "data_provider.hpp"
void data_consumer_1();
```

```
// consumer_2.cpp
#include "data_provider.hpp"
void data_consumer_2();
```

Thoughts?

What About `constexpr`?

```
// data_provider.hpp
constexpr auto data = make_data();
```

```
// consumer_1.cpp
#include "data_provider.hpp"
void data_consumer_1();
```

```
// consumer_2.cpp
#include "data_provider.hpp"
void data_consumer_2();
```

Thoughts?

1. each translation unit has to recompute the `data` object

What About `constexpr`?

```
// data_provider.hpp
constexpr auto data = make_data();
```

```
// consumer_1.cpp
#include "data_provider.hpp"
void data_consumer_1();
```

```
// consumer_2.cpp
#include "data_provider.hpp"
void data_consumer_2();
```

Thoughts?

1. each translation unit has to recompute the `data` object
2. each translation unit has its own copy of the `data` object

What About `constexpr`?

```
// data_provider.hpp
constexpr inline auto data = make_data();
```

```
// consumer_1.cpp
#include "data_provider.hpp"
void data_consumer_1();
```

```
// consumer_2.cpp
#include "data_provider.hpp"
void data_consumer_2();
```

Thoughts?

What About `constexpr`?

```cpp
// data_provider.hpp
constexpr inline auto data = make_data();
```

```cpp
// consumer_1.cpp
#include "data_provider.hpp"
void data_consumer_1();
```

```cpp
// consumer_2.cpp
#include "data_provider.hpp"
void data_consumer_2();
```

Thoughts?

1. each translation unit has to recompute the `data` object

What About `constexpr`?

```
// data_provider.hpp
constexpr inline auto data = make_data();
```

```
// consumer_1.cpp
#include "data_provider.hpp"
void data_consumer_1();
```

```
// consumer_2.cpp
#include "data_provider.hpp"
void data_consumer_2();
```

Thoughts?

1. each translation unit has to recompute the `data` object
2. only one copy of the `data` after linking

What If I Hide The Data?

What About Hidden `constexpr`?

```
// data_provider.hpp
std::span<float, 1024> get_data();
```

```
// data_provider.cpp
std::span<float, 1024> get_data() {
  constexpr static auto data = make_data();
  return data;
}
```

https://godbolt.org/z/WecG8xx3K

```
// consumer_1.cpp
#include "data_provider.hpp"
void data_consumer_1() {
  // use get_data()
}
```

https://godbolt.org/z/rj89PaveG

What About Hidden `constexpr`?

```cpp
// data_provider.hpp
std::span<float, 1024> get_data();
```

```cpp
// data_provider.cpp
std::span<float, 1024> get_data() {
  constexpr static auto data = make_data();
  return data;
}
```

https://godbolt.org/z/WecG8xx3K

```cpp
// consumer_1.cpp
#include "data_provider.hpp"
void data_consumer_1() {
  // use get_data()
}
```

https://godbolt.org/z/rj89PaveG

1. `data` is computed once

What About Hidden `constexpr`?

```cpp
// data_provider.hpp
std::span<float, 1024> get_data();
```

```cpp
// data_provider.cpp
std::span<float, 1024> get_data() {
  constexpr static auto data = make_data();
  return data;
}
```

https://godbolt.org/z/WecG8xx3K

```cpp
// consumer_1.cpp
#include "data_provider.hpp"
void data_consumer_1() {
  // use get_data()
}
```

https://godbolt.org/z/rj89PaveG

1. `data` is computed once
2. There is only 1 copy of `data`

What About Hidden `constexpr`?

```
// data_provider.hpp
std::span<float, 1024> get_data();
```

```
// data_provider.cpp
std::span<float, 1024> get_data() {
  constexpr static auto data = make_data();
  return data;
}
```

https://godbolt.org/z/WecG8xx3K

```
// consumer_1.cpp
#include "data_provider.hpp"
void data_consumer_1() {
  // use get_data()
}
```

https://godbolt.org/z/rj89PaveG

1. `data` is computed once
2. There is only 1 copy of `data`
3. Functions no longer have visibility into `data`

What About Hidden `constexpr`?

Functions no longer have visibility into `data`

What About Hidden `constexpr`?

Functions no longer have visibility into `data`

… or do they?

Let’s look at a single standalone example

Final Summary

Final Summary

Final Summary

- ✔ Enable `-fvisibility=hidden` to make symbols hidden by default

Final Summary

- ✔ Enable `-fvisibility=hidden` to make symbols hidden by default
- ✔ Explicitly export symbols you want visible

Final Summary

- ✔ Enable `-fvisibility=hidden` to make symbols hidden by default
- ✔ Explicitly export symbols you want visible
- ✔ Limit the surface of your interfaces

Final Summary

- ✅ Enable `-fvisibility=hidden` to make symbols hidden by default
- ✅ Explicitly export symbols you want visible
- ✅ Limit the surface of your interfaces
- ✅ Enable LTO with your hidden symbols to get cross-translation unit inlining

Final Summary

- ✓ Enable `-fvisibility=hidden` to make symbols hidden by default
- ✓ Explicitly export symbols you want visible
- ✓ Limit the surface of your interfaces
- ✓ Enable LTO with your hidden symbols to get cross-translation unit inlining
- ✓ Build a compilation firewall around `constexpr` data tables

Final Summary

- ✓ Enable `-fvisibility=hidden` to make symbols hidden by default
- ✓ Explicitly export symbols you want visible
- ✓ Limit the surface of your interfaces
- ✓ Enable LTO with your hidden symbols to get cross-translation unit inlining
- ✓ Build a compilation firewall around `constexpr` data tables
  - Functions that need constant expressions get the `constexpr` data

Final Summary

- ✓ Enable `-fvisibility=hidden` to make symbols hidden by default
- ✓ Explicitly export symbols you want visible
- ✓ Limit the surface of your interfaces
- ✓ Enable LTO with your hidden symbols to get cross-translation unit inlining
- ✓ Build a compilation firewall around `constexpr` data tables
  - Functions that need constant expressions get the `constexpr` data
  - Functions that need runtime views, get `view` or `span` types.

Final Summary

- ✓ Enable `-fvisibility=hidden` to make symbols hidden by default
- ✓ Explicitly export symbols you want visible
- ✓ Limit the surface of your interfaces
- ✓ Enable LTO with your hidden symbols to get cross-translation unit inlining
- ✓ Build a compilation firewall around `constexpr` data tables
  - Functions that need constant expressions get the `constexpr` data
  - Functions that need runtime views, get `view` or `span` types.
- ✓ link “internal” or “static” libraries to help with testing framework

Final Summary

- ✓ Enable `-fvisibility=hidden` to make symbols hidden by default
- ✓ Explicitly export symbols you want visible
- ✓ Limit the surface of your interfaces
- ✓ Enable LTO with your hidden symbols to get cross-translation unit inlining
- ✓ Build a compilation firewall around `constexpr` data tables
  - Functions that need constant expressions get the `constexpr` data
  - Functions that need runtime views, get `view` or `span` types.
- ✓ link “internal” or “static” libraries to help with testing framework
- ✓ Use CMake to do the boilerplate stuff for you

Final Example

Final Example

https://github.com/lefticus/hidden_symbol_example_library

Jason Turner

C++ Weekly

- Weekly videos since March, 2016
- 115k+ subscribers, 455+ weeks straight

https://www.youtube.com/@cppweekly

Jason Turner

- Author
  - C++ Best Practices, C++23 Best Practices
  - OpCode, Copy and Reference, Object Lifetime Puzzlers
  - https://amzn.to/3xWh8Ox
  - https://leanpub.com/u/jason_turner

Jason Turner

- Developer
  - https://cppbestpractices.com
  - https://github.com/lefticus
  - https://github.com/cpp-best-practices
- Microsoft MVP for C++, 2015-present

Jason Turner - Training

https://articles.emptycrate.com/training.html

How to get my training:

1. Have me come to your company on-site for dynamic customized training where you already are - generally the most economical option for groups (CA, DE, NL, RO, CZ, JP, US, PL, SE, …)
2. Come to a conference workshop
   - C++ On Sea (Folkestone, UK, Late June)
   - CppCon (Aurora, CO, US, ~Sept)
   - NDC TechTown (Kongsberg, NO, ~Sept)
   - And possibly others

Workshops!

- C++ On Sea
- NDC Tech Town
- CppCon