THE ABI CHALLENGE

feat. inline namespaces

arvid@libtorrent.org

WHO AM I?

- writing C++ since 1997
- co-author of luabind (inspired by boost.python)
- author of libtorrent (bittorrent library)
- work at Blockstream

WHY TALK ABOUT ABI?

- A lot of people run into ABI issues, sometimes without knowing it

WHY TALK ABOUT ABI?

- A lot of people run into ABI issues, sometimes without knowing it
- Causes memory corruption or (at best) link errors

WHY TALK ABOUT ABI?

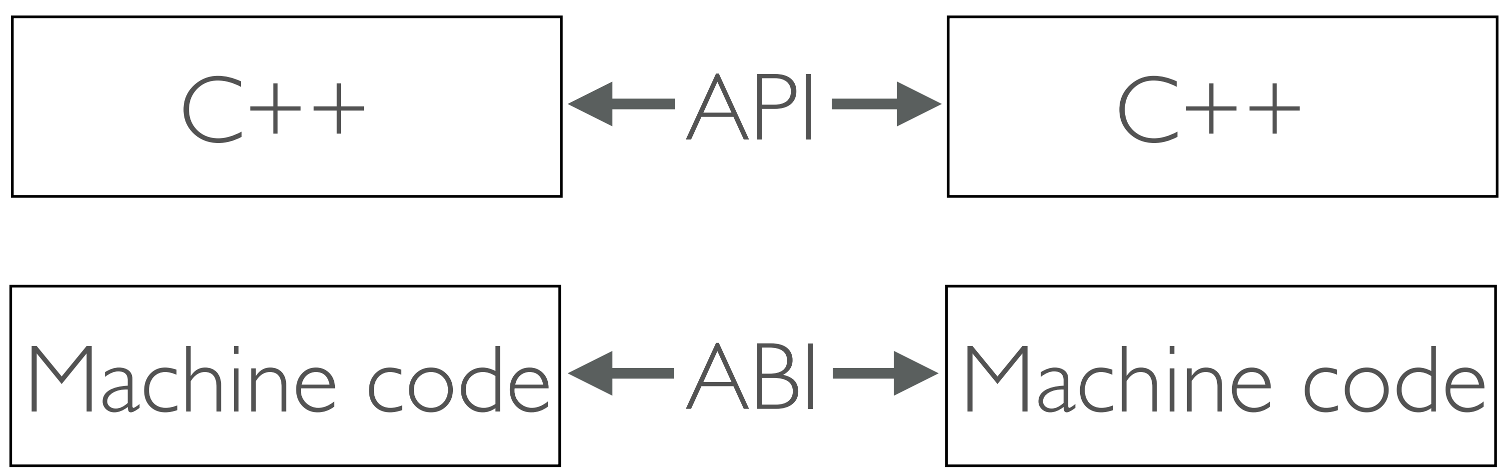
- A lot of people run into ABI issues, sometimes without knowing it
- Causes memory corruption or (at best) link errors
- Few people seem to appreciate the problem

AGENDA

- what is an ABI?
- linking symbols
- name mangling
- ABI errors in practice
- build systems
- inline namespaces
- forward declarations

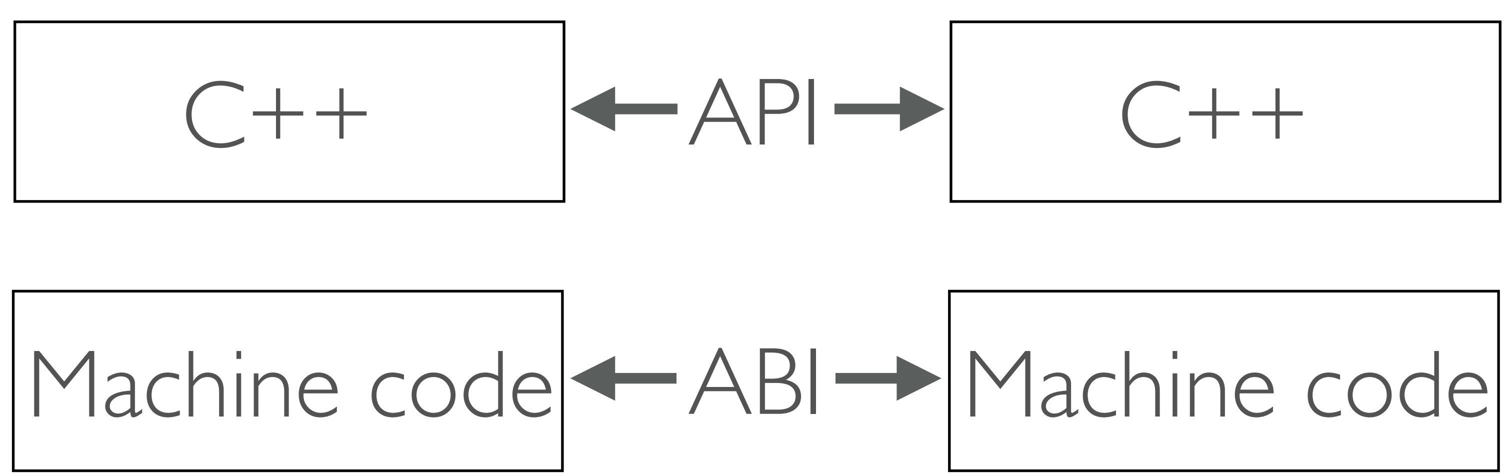
ABI

- Application Binary Interface. Like API but for machine code



ABI

- compatible API → recompilation
- compatible ABI → linking with an existing binary



ABI

- **calling convention**
  - which registers to pass arguments in
  - pass two 32bit arguments in 64 bit registers
  - split class and pass fields in registers
  - pass floats in special registers
  - pass arguments in SIMD registers
  - return value optimization [C++17]
  - how are exceptions implemented?

ABI

- **class layouts**
  - vtable layout (virtual inheritance)
  - where/how do we pad fields, alignment
  - empty base class optimization
    [[no_unique_address]]
  - std::pair (compressed_pair)
  - std::string (small string optimization)

ABI

- across library boundaries
  - C++ version

    may affect layout, calling convention, name mangling
  - defines

    may affect layout (e.g. _GLIBCXX_USE_CXX11_ABI)
  - any compiler flag that alters class layout or calling conventions

ABI

- across library boundaries
  - **C++ version**

    may affect layout, calling convention, name mangling
  - **defines**

    may affect layout (e.g. _GLIBCXX_USE_CXX11_ABI)
  - any compiler flag that alters class layout or calling conventions

LINKING

- Define `f()` in one translation unit, call it from another

LINKING

f.cpp

```
int f(int x) {
    return x*x;
}
```

main.cpp

```
int f(int);

int main() {
    std::cout
        << f(10)
        << "\n";
}
```

LINKING

`f.cpp`

`main.cpp`

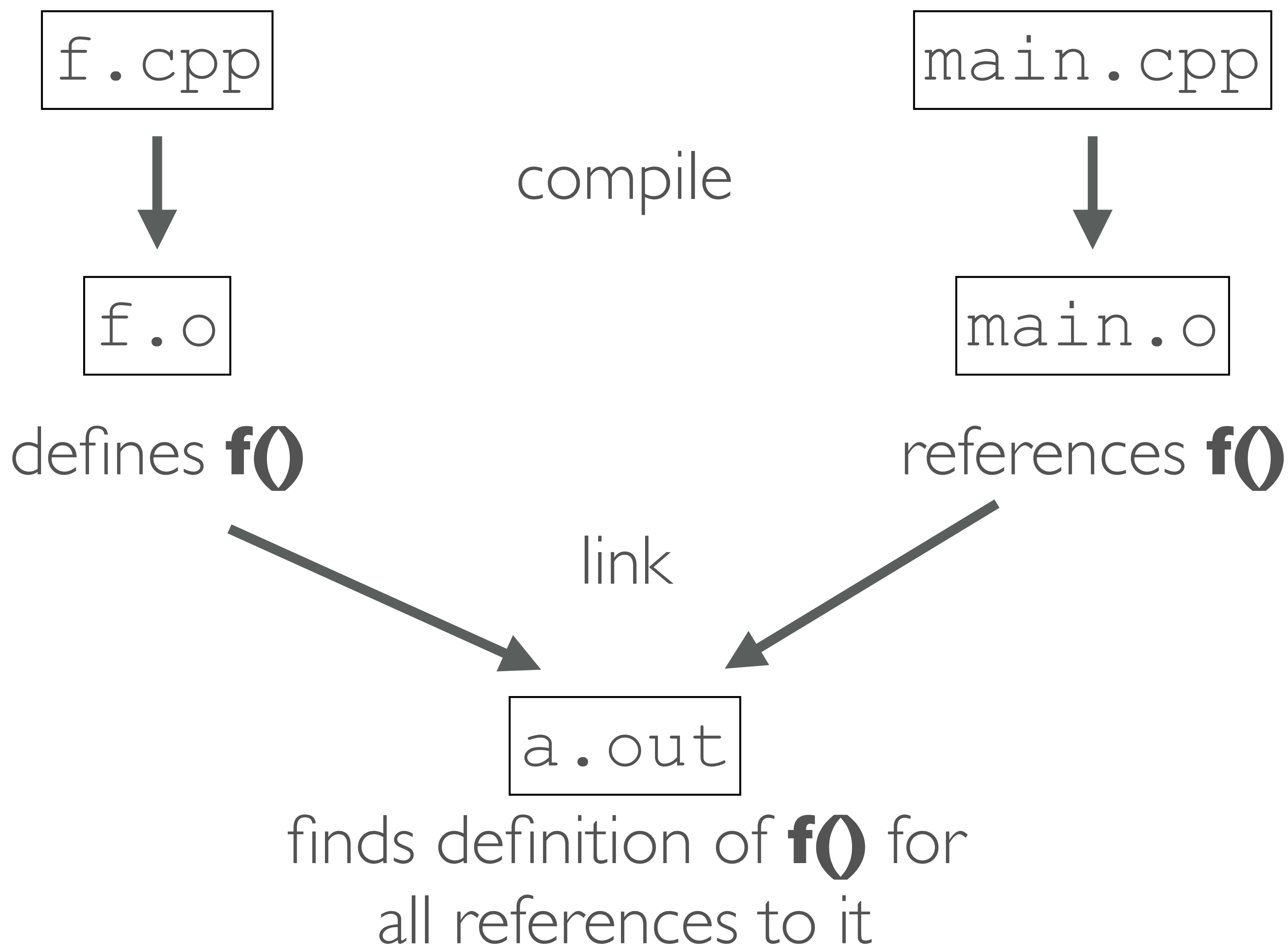
LINKING

`f.cpp` compile `main.cpp`

`f.o` `main.o`

defines **f()** references **f()**

LINKING


f.cpp
main.cpp
compile
f.o
main.o
defines **f()**
references **f()**
link
a.out
finds definition of **f()** for
all references to it

NAME MANGLING (C)

f.c

```
int f(int x)
{
 return x*x;
}
```

main.c

```
int f(int);
int main()
{
 printf("%d\n", f(10));
}
```

NAME MANGLING (C)

f.c

```
int f(int x)
{
 return x*x;
}
```

main.c

```
int f(int);
int main()
{
 printf("%d\n", f(10));
}
```

```
$ nm f.o
T _f
```

```
$ nm main.o
U _f
```

NAME MANGLING (C)

f.c

```
int f(int x)
{
 return x*x;
}
```

Defined symbol

```
$ nm f.o
T _f
```

main.c

```
int f(int);
int main()
{
 printf("%d\n", f(10));
}
```

External symbol

```
$ nm main.o
U _f
```

NAME MANGLING (C)

f.c

```c
float f(float x)
{
 return x*x;
}
```

main.c

```c
int f(int);
int main()
{
 printf("%d\n", f(10));
}
```

```
$ nm f.o
T _f
```

```
$ nm main.o
U _f
```

NAME MANGLING (C)

f.c

```c
float f(float x)
{
 return x*x;
}
```

main.c

```c
int f(int);
int main()
{
 printf("%d\n", f(10));
}
```

linked together: memory corruption at runtime!

```
$ nm f.o
T _f
```

```
$ nm main.o
U _f
```

NAME MANGLING (C++)

f.cpp

```
float f(float x)
{
 return x*x;
}
```

main.cpp

```
int f(int);
int main()
{
 printf("%d\n", f(10));
}
```

NAME MANGLING (C++)

f.cpp

```
float f(float x)
{
 return x*x;
}
```

main.cpp

```
int f(int);
int main()
{
 printf("%d\n", f(10));
}
```

```
$ nm f.o
T __Z1ff
```

```
$ nm main.o
U __Z1fi
```

NAME MANGLING (C++)

f.cpp

```
float f(float x)
{
 return x*x;
}
```

main.cpp

```
int f(int);
int main()
{
   intf(“%d\n”, f(10));
```

symbol name mismatch
link error!

```
$ nm f.o
T __Z1ff
```

```
$ nm main.o
U __Z1fi
```

NAME MANGLING

- C++ encode signature in symbol names
  - to support overloading
  - increases type safety

NAME MANGLING

- C++ encode signature in symbol names
  - to support overloading
  - increases type safety
- **BUT**
  - encodes user types by name
  - return type (normally) not included

NAME MANGLING

```
struct foobar;
void f(foobar const& x)
{
  // …
}
```

```
$ nm f.o
T __Z1fRK6foobar
```

NAME MANGLING

```
struct foobar;
void f(foobar const& x)
{
  // ...
}
```

"foobar" encoded in symbol

```
$ nm f.o
T __Z1fRK6foobar
```

NAME MANGLING

- Considering the following definition

```
struct foobar {
 int x;
#ifndef NDEBUG
 int debug_state;
#endif
};
```

NAME MANGLING

- One-definition rule (ODR) violation [basic.def.odr]

> Every program shall contain exactly one definition of every non-inline function or variable that is odr-used in that program outside of a discarded statement;
> no diagnostic required.

NAME MANGLING

- One-definition rule (ODR) violation [basic.def.odr]

> Every program shall contain exactly one definition of every non-inline function or variable that is odr-used in that program outside of a discarded statement;
> **no diagnostic required**.

ABI

- It's hard to maintain ABI compatibility across C++ versions

ABI

- It's hard to maintain ABI compatibility across C++ versions

"At some point, we'll need to acknowledge the problem which results from building Boost with one -std and user code with another"

— Peter Dimov (boost mailing list)

ABI

"The thing that should work and the only thing that can be guaranteed to work is when your **whole code base is compiled with the same compiler with the same standard**"

— degski (boost mailing list)

(emphasis mine)

ABI

- What happens in the wild when using shared libraries
  - library in release-mode, client in debug mode
  - library in C++11, client in C++14
  - library headers may be newer than library binary
    the system building the client may have a newer version of the library than the system running it

EXAMPLE 1

Peter Dimov's example from boost mailing list:

```
struct X;
struct Y {
    void f( X const& x );
#if !defined(BOOST_NO_CXX11_RVALUE_REFERENCES)
    void f( X&& x );
#endif
};
```

EXAMPLE 2

std::string may be different types depending on language version

```
std::string error_code::message() const {
    // ...
   return msg;
};
```

EXAMPLE 2

std::string may be different types depending on language version

```
std::string error_code::message() const {
    // ...
  return msg;
};
```

std::string may depend on language version. Not encoded in mangled name

EXAMPLE 3

```
template <class T>
using map_string
 = std::map<std::string, T, strview_less>;
```

EXAMPLE 3

```
#if (__cplusplus > 201103)

template <class T>
using map_string
 = std::map<std::string, T, strview_less>;

#else

template <class T>
struct map_string : std::map<std::string, T> {
// ...
};

#endif
```

EXAMPLE 3

```
#if (__cplusplus > 201103)

template <class T>
using map_string
 = std::map<std::string, T, strview_less>;

#else

template <class T>
struct map_string : std::map<std::string, T> {
// ...
};

#endif
```

these are different types, not compatible

SOLUTION

SOLUTION

- better build systems

  don't link incompatible objects

SOLUTION

- better build systems

  don't link incompatible objects

- better libraries

  turn memory corruption into link errors

SOLUTION

- better build systems

  don't link incompatible objects

- better libraries

  turn memory corruption into link errors

- increased awareness

BUILD SYSTEMS

- Build all your dependencies from source

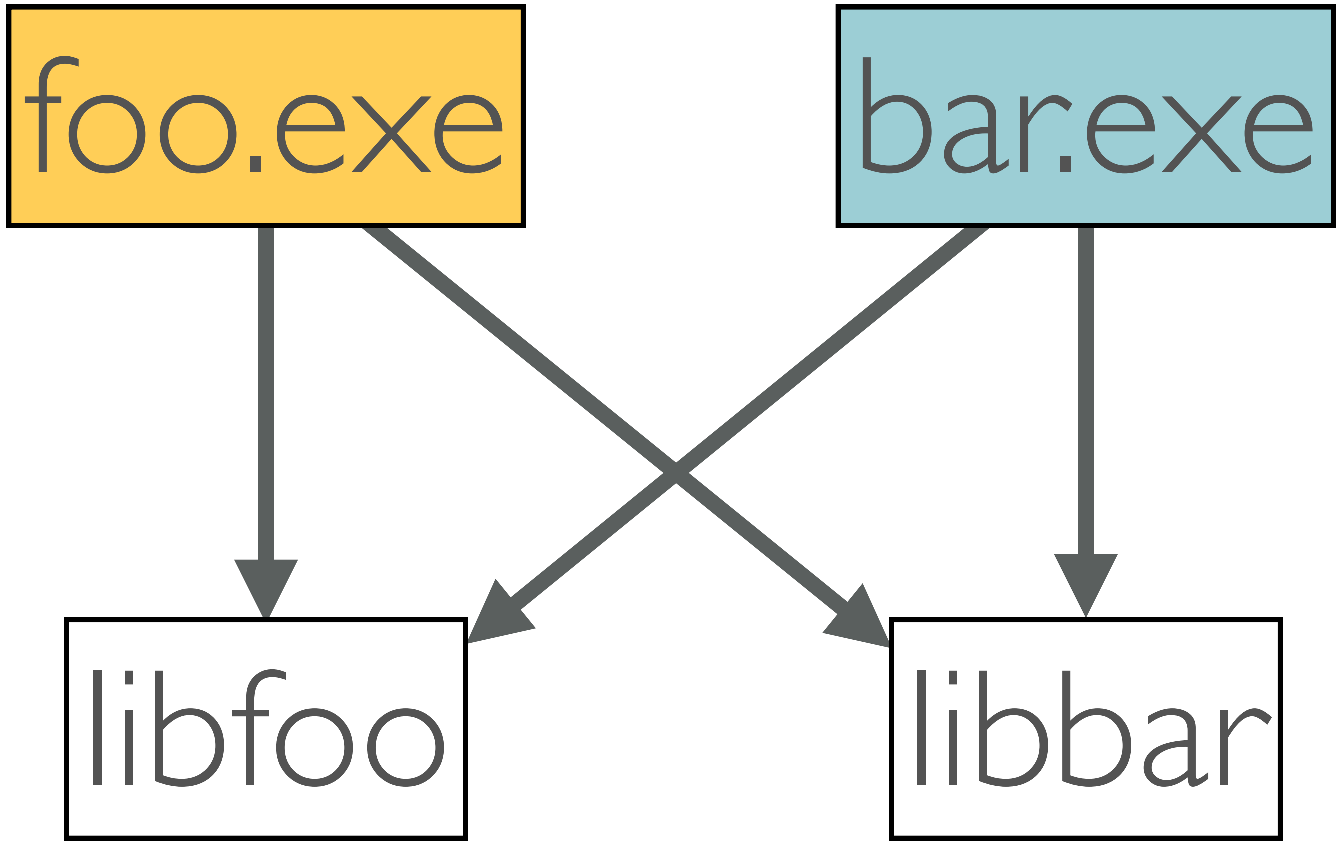
BUILD SYSTEMS

- Build all your dependencies from source
- Use a build system that ensures link-compatibility
  (or at least doesn't encourage you to create link incompatibilities)

BUILD SYSTEMS

Example 1

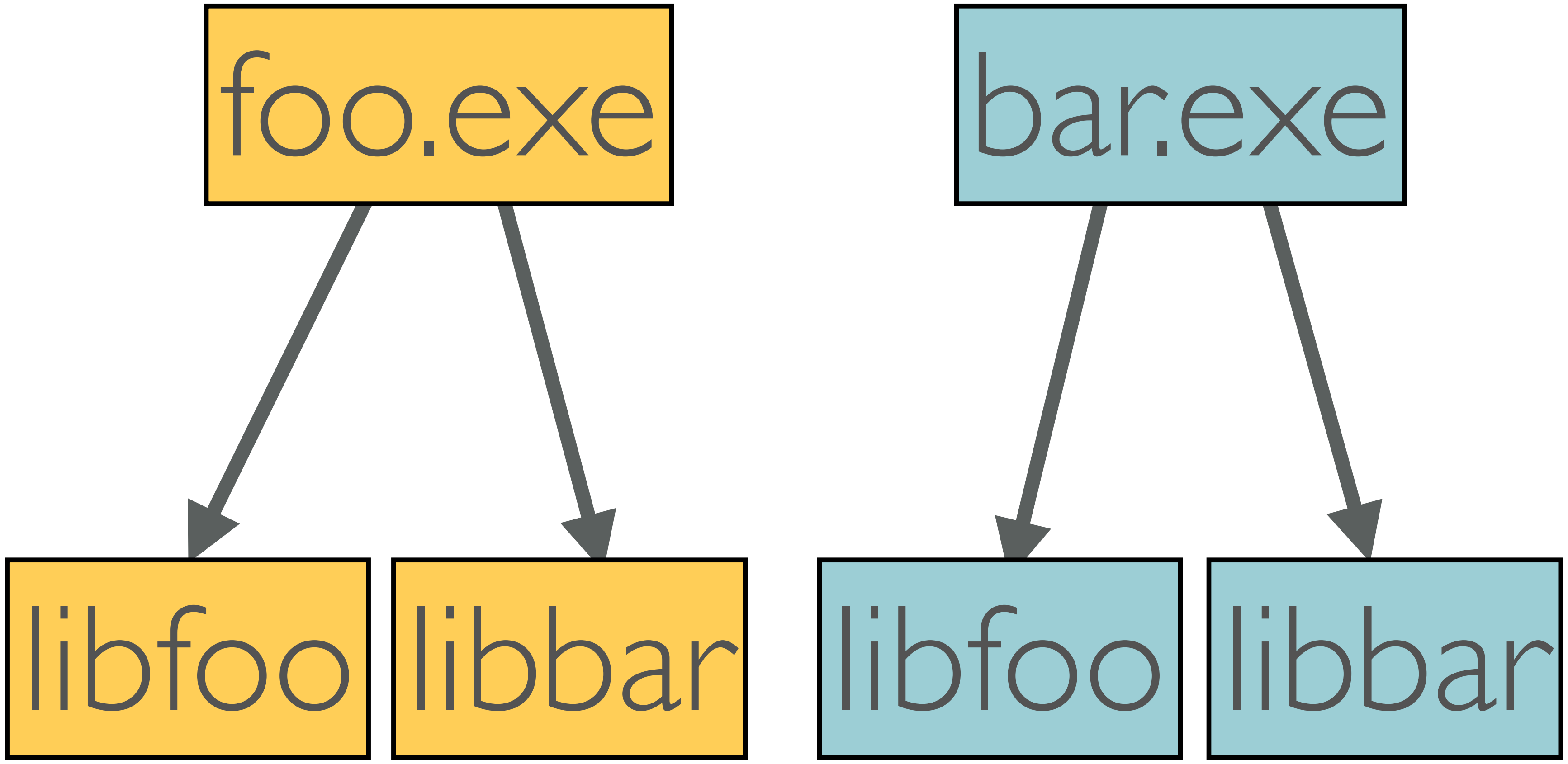
color indicates ABI



BUILD SYSTEMS

Example 1

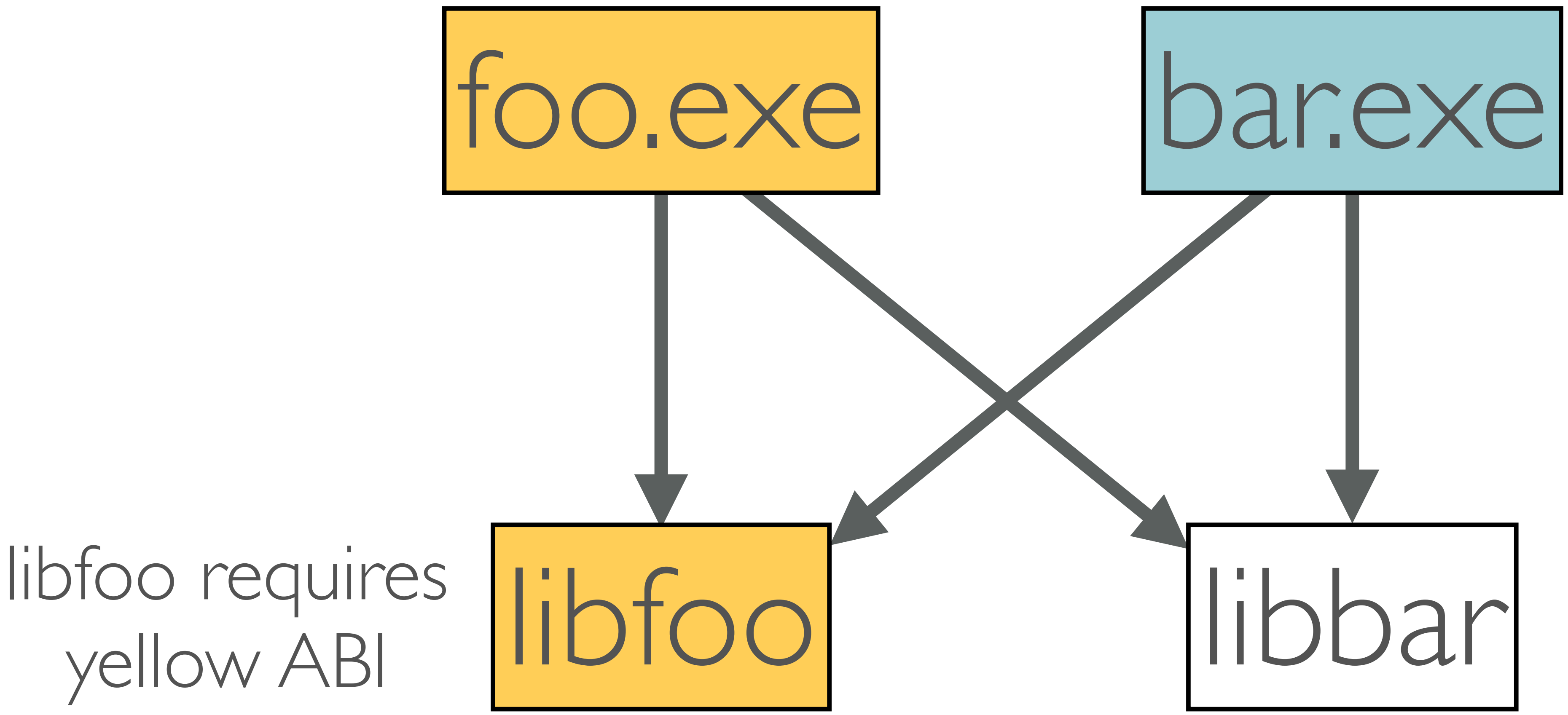
color indicates ABI



build configurations propagate down

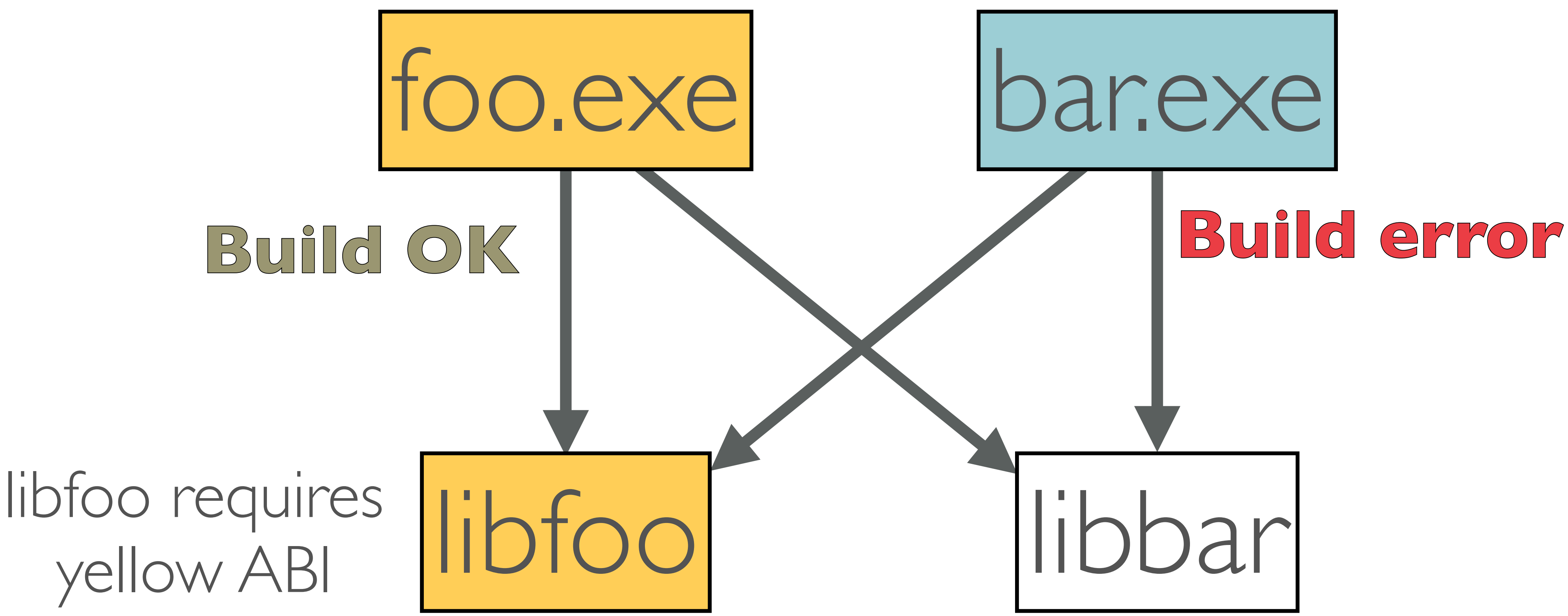
BUILD SYSTEMS

Example 2



BUILD SYSTEMS

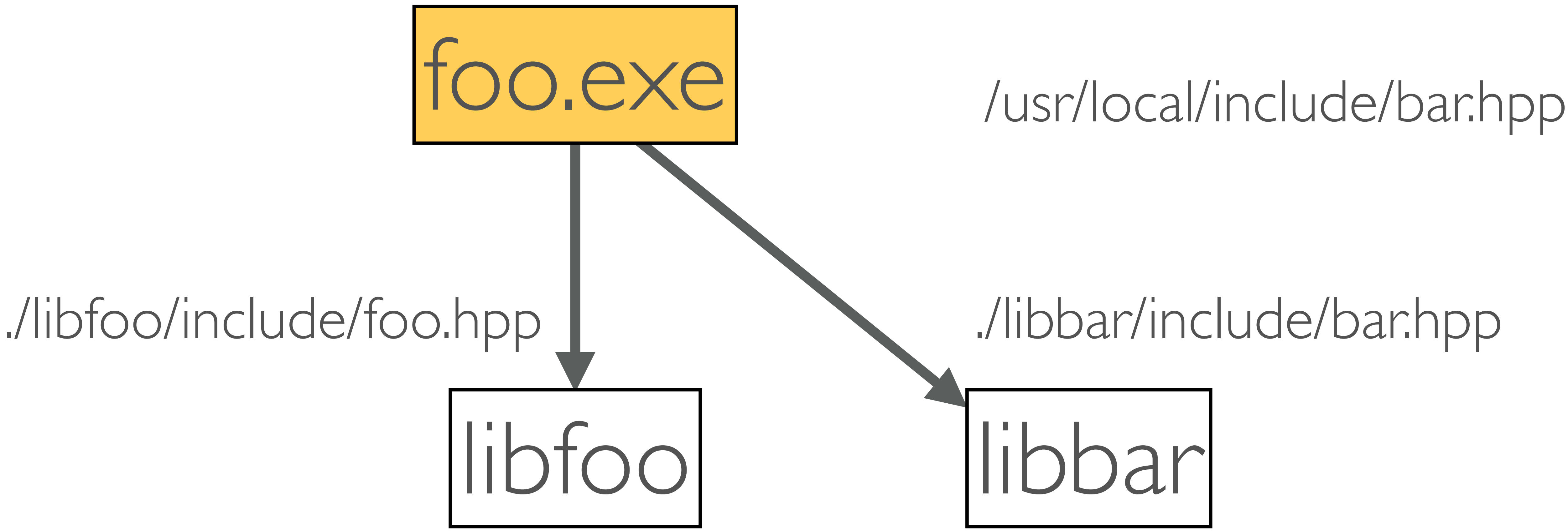
Example 2



build restrictions propagate "up"

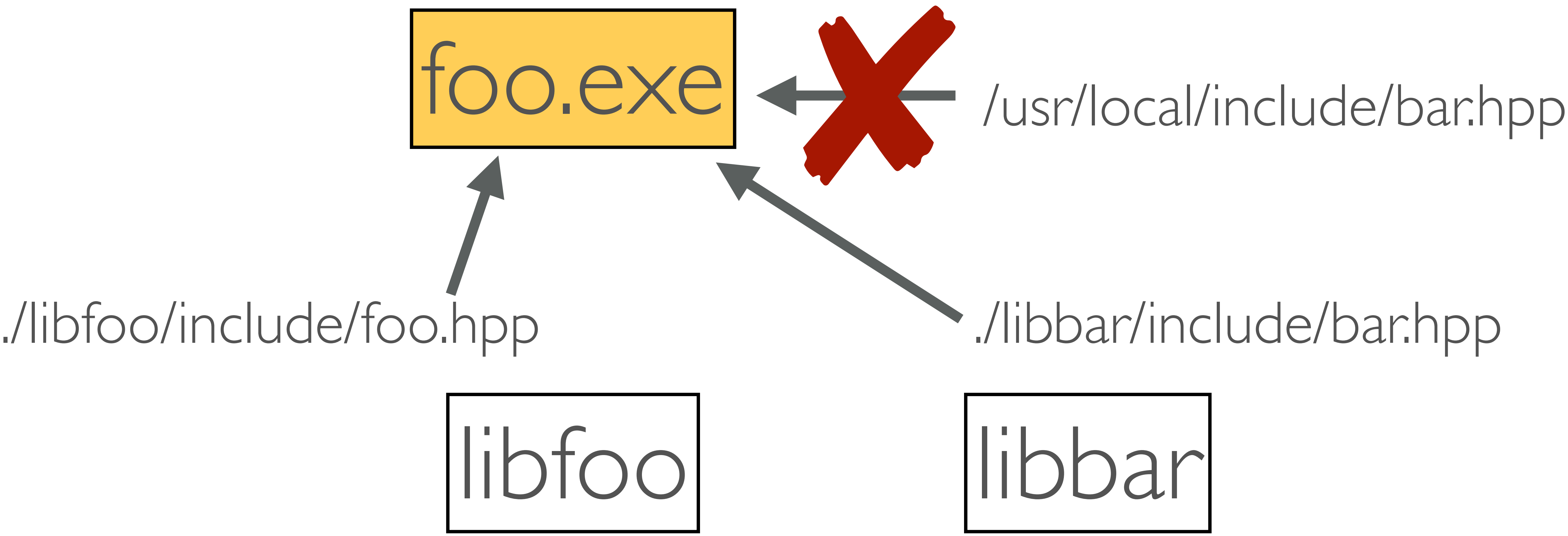
BUILD SYSTEMS

Example 3



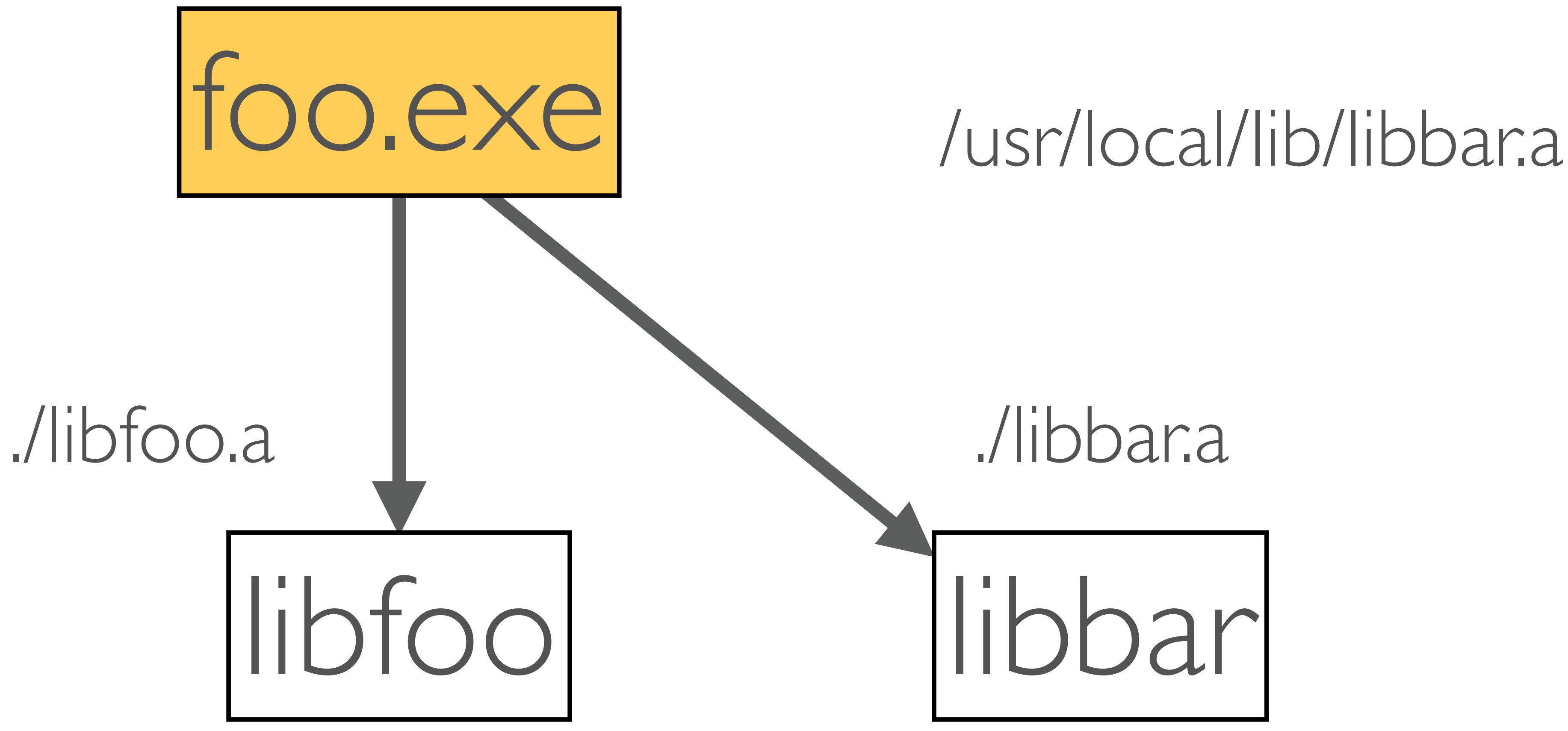
BUILD SYSTEMS

Example 3



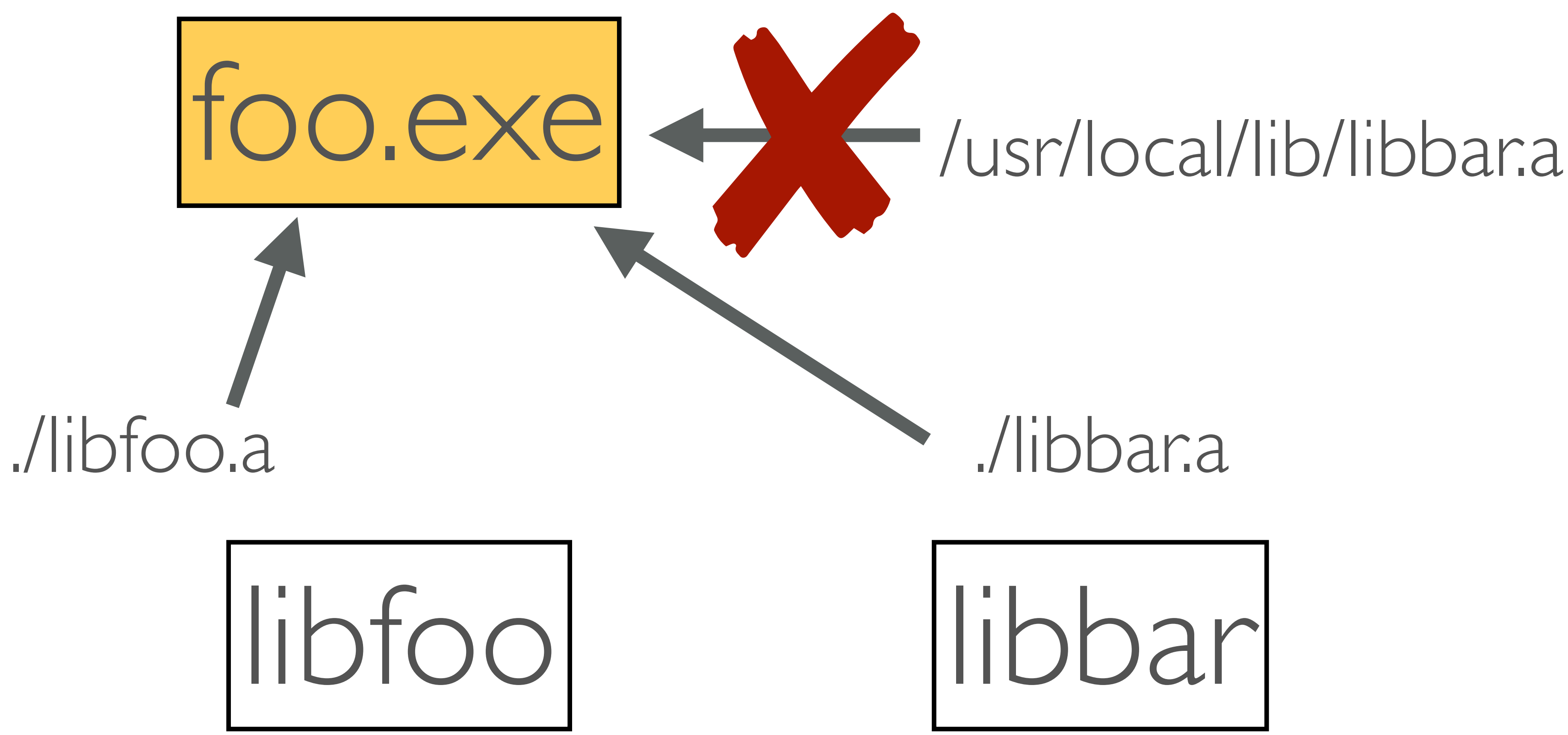
BUILD SYSTEMS

Example 4



BUILD SYSTEMS

Example 4



avoid -L linker option

BUILD SYSTEMS

- boost-build gets close to this

BUILD SYSTEMS

- boost-build gets close to this
- 3rd party dependencies is still a challenge. example: openssl

BUILD SYSTEMS

Your library may be distributed as a binary, whether you like it or not.

what to do?

INLINE NAMESPACES

- inline namespaces let you inject information into the mangled name (**ABI**), without altering the **API**

INLINE NAMESPACES

- inline namespaces let you inject information into the mangled name (**ABI**), without altering the **API**

- inline namespaces affect the *linker names* of symbols, while preserving the API

INLINE NAMESPACES

- inline namespaces let you inject information into the mangled name (**ABI**), without altering the **API**

- inline namespaces affect the *linker names* of symbols, while preserving the API

- e.g. the two **`foobar`** types could be given different names

NAME MANGLING

- foobar is defined as:

```
struct foobar {
 int x;
#ifndef NDEBUG
 int debug_state;
#endif
};
```

NAME MANGLING

- foobar is defined as:

```
#ifdef NDEBUG
inline namespace rel {
#else
inline namespace dbg {
#endif

struct foobar {
 int x;
#ifndef NDEBUG
 int debug_state;
#endif
};
} // inline namespace
```

INLINE NAMESPACES

When built in debug mode

```
$ nm f.o
T __Z1fRKN3dbg6foobarE
```

When built in release mode

```
$ nm f.o
T __Z1fRKN3rel6foobarE
```

NAME MANGLING

```
Undefined symbols for architecture x86_64:
  "f(dgb::foobar const&)", referenced from:
      _main in main.o
ld: symbol(s) not found for architecture
x86_64
clang-7: error: linker command failed with
exit code 1 (use -v to see invocation)
```

INLINE NAMESPACES

- inline namespaces make all its declarations available in its surrounding scope

INLINE NAMESPACES

```
namespace my_library {

inline namespace v1 {

int f(int) { … };

} // v1
} // my_library
```

library code

```
…
my_library::f(10);
…
using my_lbrary::f;
f(10)
```

client code

INLINE NAMESPACES

```
namespace my_library {

inline namespace v1 {

int f(int) { … };

} // v1
} // my_library
```

library code

namespace **v1** not necessary in user code

```
…
my_library::f(10);
…
using my_lbrary::f;
f(10)
```

client code

INLINE NAMESPACES

- you can specialise templates

INLINE NAMESPACES

```
namespace my_library {

inline namespace v1 {

  template <typename T>
  struct foobar {};

} // v1
} // my_library
```

library code

```
namespace my_library {

  template <>
  struct foobar<int> { … };

} // my_library
```

client code

INLINE NAMESPACES

```
namespace my_library {

inline namespace v1 {

  template <typename T>
  struct foobar {};

} // v1
} // my_library
```

defined in **my_library::v1**
specialized in **my_library**

```
namespace my_library {

  template <>
  struct foobar<int> { … };

} // my_library
```

client code

INLINE NAMESPACES

- From the user's point of view, as if the inline namespace isn't there
- Very useful for versioning of functions and types in libraries

INLINE NAMESPACES

```
namespace std {









namespace cxx11 {
  template <…> class basic_string {…}; // SSO
}

namespace cxx98 {
  template <…> class basic_string {…}; // COW
}

} // std
```

INLINE NAMESPACES

```
namespace std {

#if defined _GLIBCXX_USE_CXX11_ABI
inline namespace cxx11 {}
#else
inline namespace cxx98 {}
#endif

namespace cxx11 {
  template <…> class basic_string {…}; // SSO
}

namespace cxx98 {
  template <…> class basic_string {…}; // COW
}

} // std
```

INLINE NAMESPACES

```
namespace std {

#if defined _GLIBCXX_USE_CXX11_ABI
inline namespace cxx11 {}
#else
inline namespace cxx98 {}
#endif

namespace cxx11 {
  template <…> class basic_string {…}; // SSO
}

namespace cxx98 {
  template <…> class basic_string {…}; // COW
}

} // std
```

inlineness depends on *first* time namespace is opened

INLINE NAMESPACES

Inline namespaces to provide backwards ABI compatibility

INLINE NAMESPACES

```cpp
namespace library {


inline namespace v1 {}




namespace v1 {
  void f(std::string const&);
}




} // library
```

header

```cpp
namespace library {

namespace v1 {
 void f(std::string const& s){
  // do some work
 }
}







} // library
```

C++ file

INLINE NAMESPACES

```
namespace library {

#ifdef USE_OLD_API
inline namespace v1 {}
#else
inline namespace v2 {}
#endif

namespace v1 {
  void f(std::string const&);
}

namespace v2 {
  void f(std::string_view);
}

} // library
```

header

```
namespace library {

namespace v1 {
 void f(std::string const& s){
  v2::f(s);
 }
}

namespace v2 {
 void f(std::string_view){
    // do some work
  }
}

} // library
```

C++ file

FORWARD DECLARATIONS

- A symbol's actual namespace is an *implementation detail*
- clients may not forward declare 3rd party symbols

FORWARD DECLARATIONS

```
namespace third_party {

   struct foo;

} // third_party


struct bar
{
   …
   third_party::foo* the_foo_;
};
```

client code

FORWARD DECLARATIONS

```
namespace third_party {

    struct foo;

} // third_party


struct bar
{
    …
    third_party::foo* the_foo_;
};
```

foo does not need to be a complete type

client code

FORWARD DECLARATIONS

```
namespace third_party {

    struct foo;

} // third_party


struct bar
{
    …
    third_party::foo* the_foo_;
};
```

may be tempting to forward declare it, despite coming from a 3rd party

client code

FORWARD DECLARATIONS

```
namespace third_party {

   struct foo;

} // third_party


struct bar
{
   …
   third_party::foo* the_fo
};
```

client code

```
namespace third_party {
inline namespace v2 {
   struct foo;
} // v2
} // third_party
```

library code

FORWARD DECLARATIONS

```
namespace third_party {

    struct foo;

} // third_party


struct bar
{
    …
    third_party::foo* the_fo
};
```

client code

```
namespace third_party {
inline namespace v2 {
    struct foo;
} // v2
} // third_party
```

library code

namespace v2 is an implementation detail

FORWARD DECLARATIONS

```
#include "third_party/fwd.hpp"

struct bar
{
    …
    third_party::foo* the_foo_;
};
```

FORWARD DECLARATIONS

```
#include "third_party/fwd.hpp"

struct bar
{
    …
    third_party::foo* the_foo_;
};
```

forward declaration
controlled by library

FORWARD DECLARATIONS

non-trivial libraries should provide
forward declarations headers

FORWARD DECLARATIONS

- <iosfwd>
- <netfwd>
- <boost/numeric/ublas/fwd.hpp>
- <boost/math/distributions/fwd.hpp>

FORWARD DECLARATIONS

surprisingly few libraries
provide fwd headers

SUMMARY

- Build your dependencies from source
- Use a good build system
- Use **`inline namespace`** for
  - backward compatible upgrades to your ABI
  - ABI-safe build configurations
- Library authors, provide forward declaration headers!

THANK YOU

github.com/arvidn

arvid@libtorrent.org