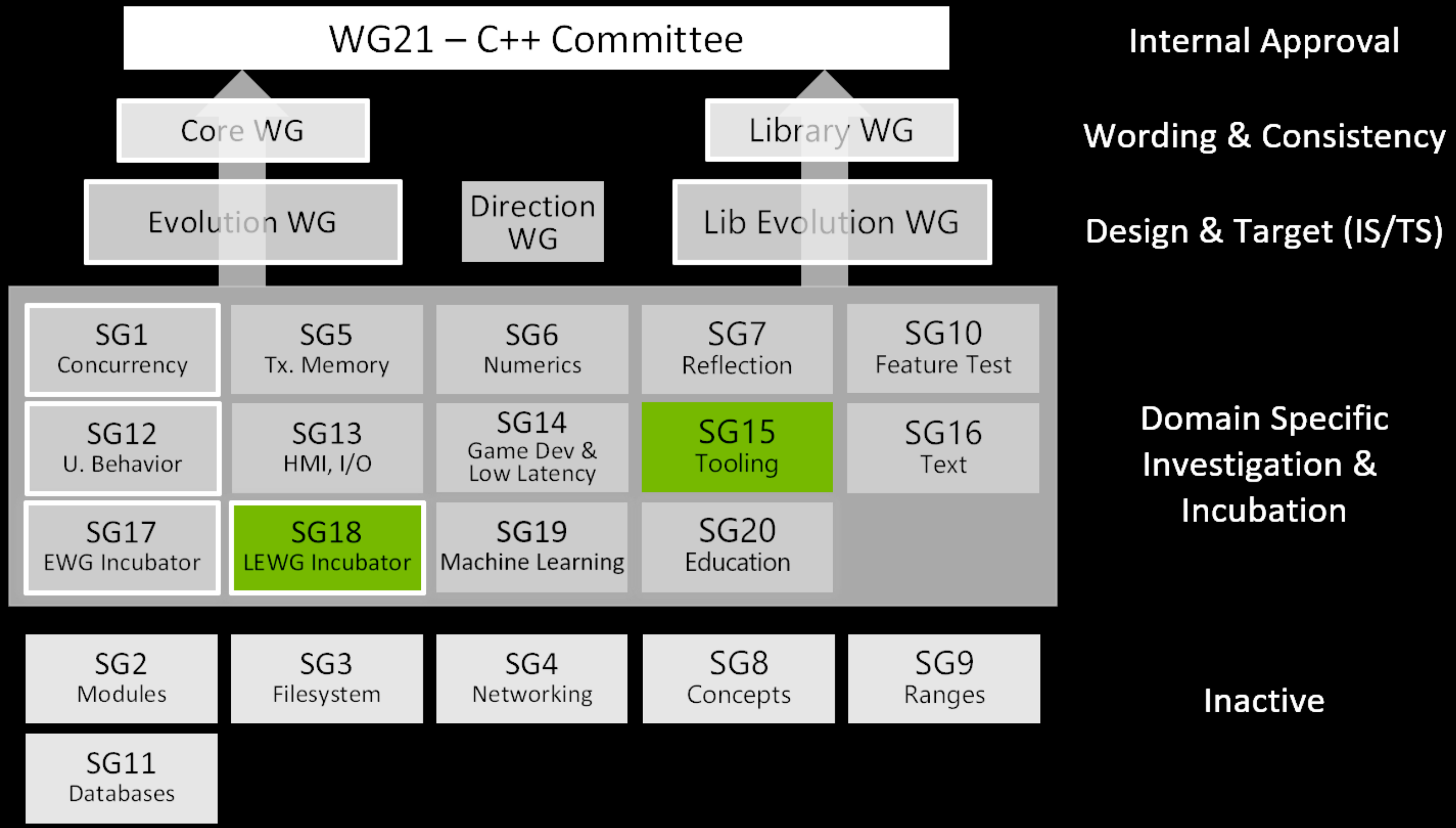
MODULES ARE COMING

Bryce Adelstein Lelbach, Core C++ 2019



WG21 – C++ Committee
Internal Approval
Core WG
Library WG
Wording & Consistency
Evolution WG
Direction WG
Lib Evolution WG
Design & Target (IS/TS)
SG1 Concurrency
SG5 Tx. Memory
SG6 Numerics
SG7 Reflection
SG10 Feature Test
SG12 U. Behavior
SG13 HMI, I/O
SG14 Game Dev & Low Latency
SG15 Tooling
SG16 Text
SG17 EWG Incubator
SG18 LEWG Incubator
SG19 Machine Learning
SG20 Education
Domain Specific Investigation & Incubation
SG2 Modules
SG3 Filesystem
SG4 Networking
SG8 Concepts
SG9 Ranges
SG11 Databases
Inactive

C++20 will have as big an impact as C++11.

- Concepts.
- Coroutines
- Improved `constexpr`.
- Ranges.
- Modules.

Modules are:

- A new compilation model for C++.
- A new way to organize C++ projects.

Textual Inclusion

math.hpp

```
#pragma once

int square(int a);
```

math.cpp

```
#include “math.hpp”

int square(int a) { return a * a; }
```

main.cpp

```
#include “math.hpp”

int main() { return square(42); }
```

Textual Inclusion

math.hpp

```
#pragma once

int square(int a);
```

math.cpp

```
#include “math.hpp”

int square(int a) { return a * a; }
```

main.cpp

```
#include “math.hpp”

int main() { return square(42); }
```

Modular Import

math.ixx

```
export module math;

export int square(int a);
```

math.mxx

```
module math;

int square(int a) { return a * a; }
```

main.cpp

```
import math;

int main() { return square(42); }
```

Modules will have a greater impact than any other feature added post C++-98.

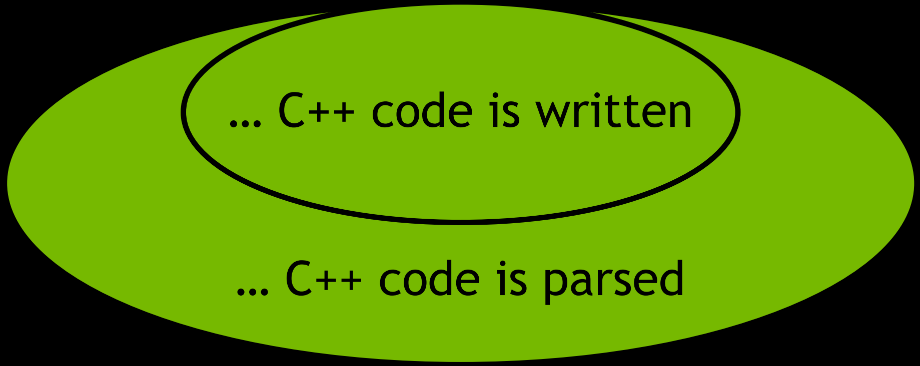
C++11’s <thread> changes how…

… C++ code is written

C++11's lambdas change how…

… C++ code is written

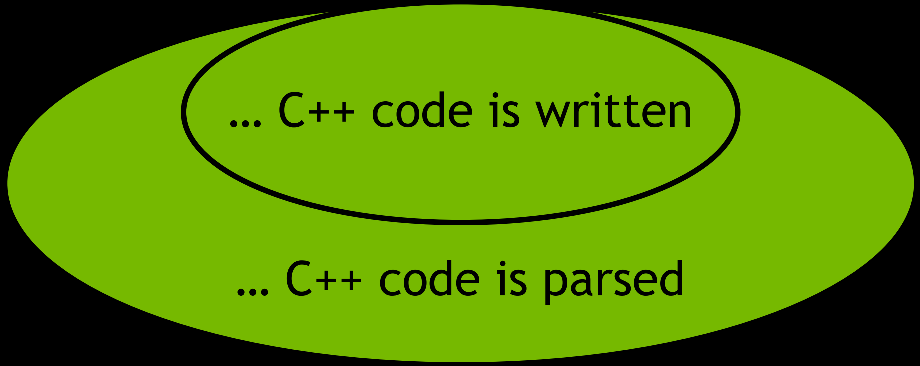
C++11’s lambdas change how…



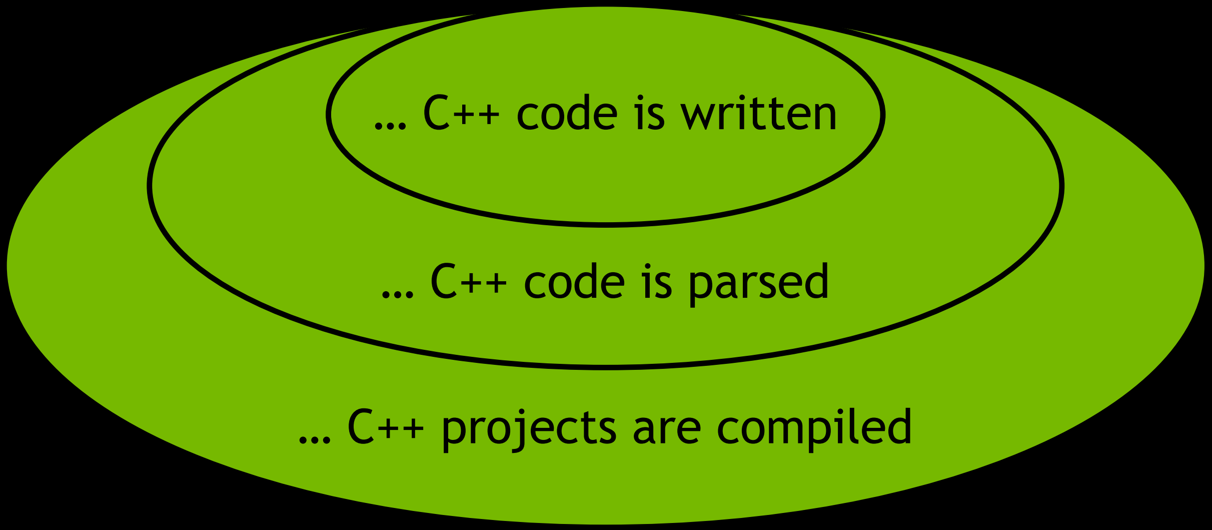
C++20's modules change how…

… C++ code is written

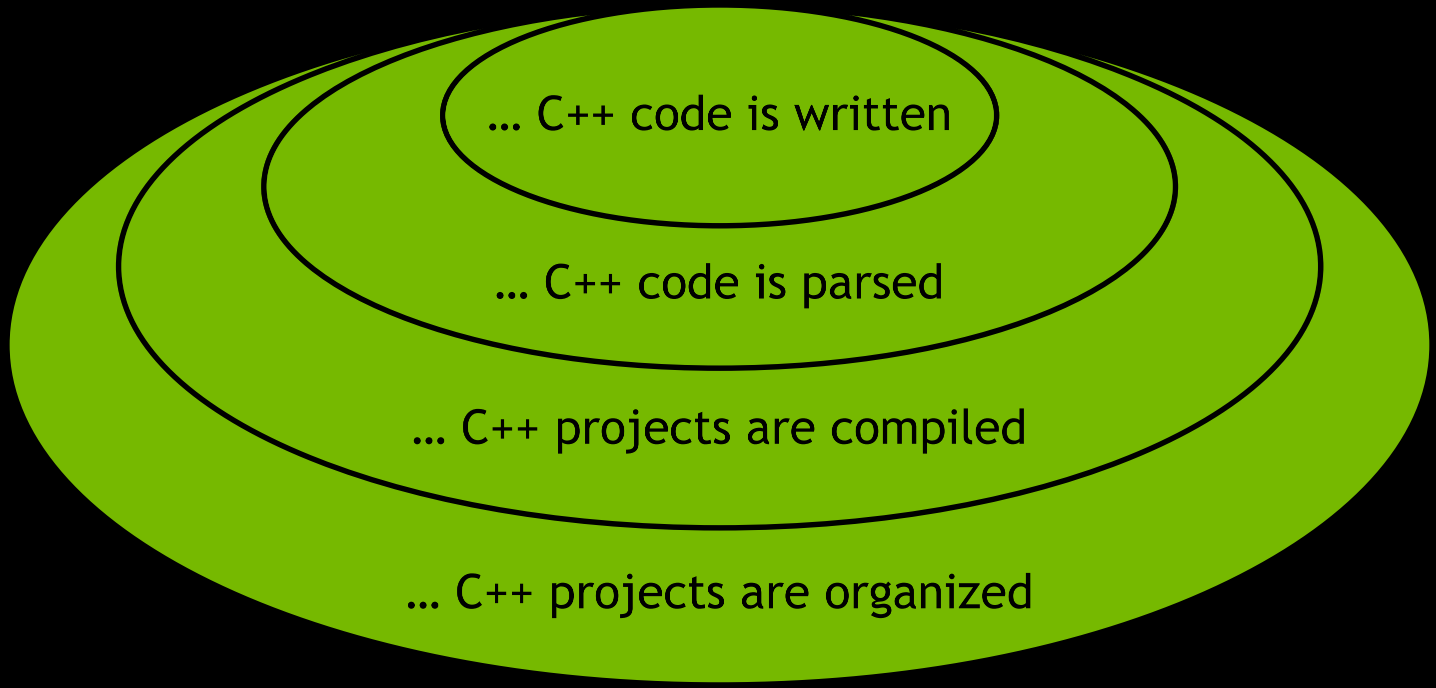
C++20’s modules change how…



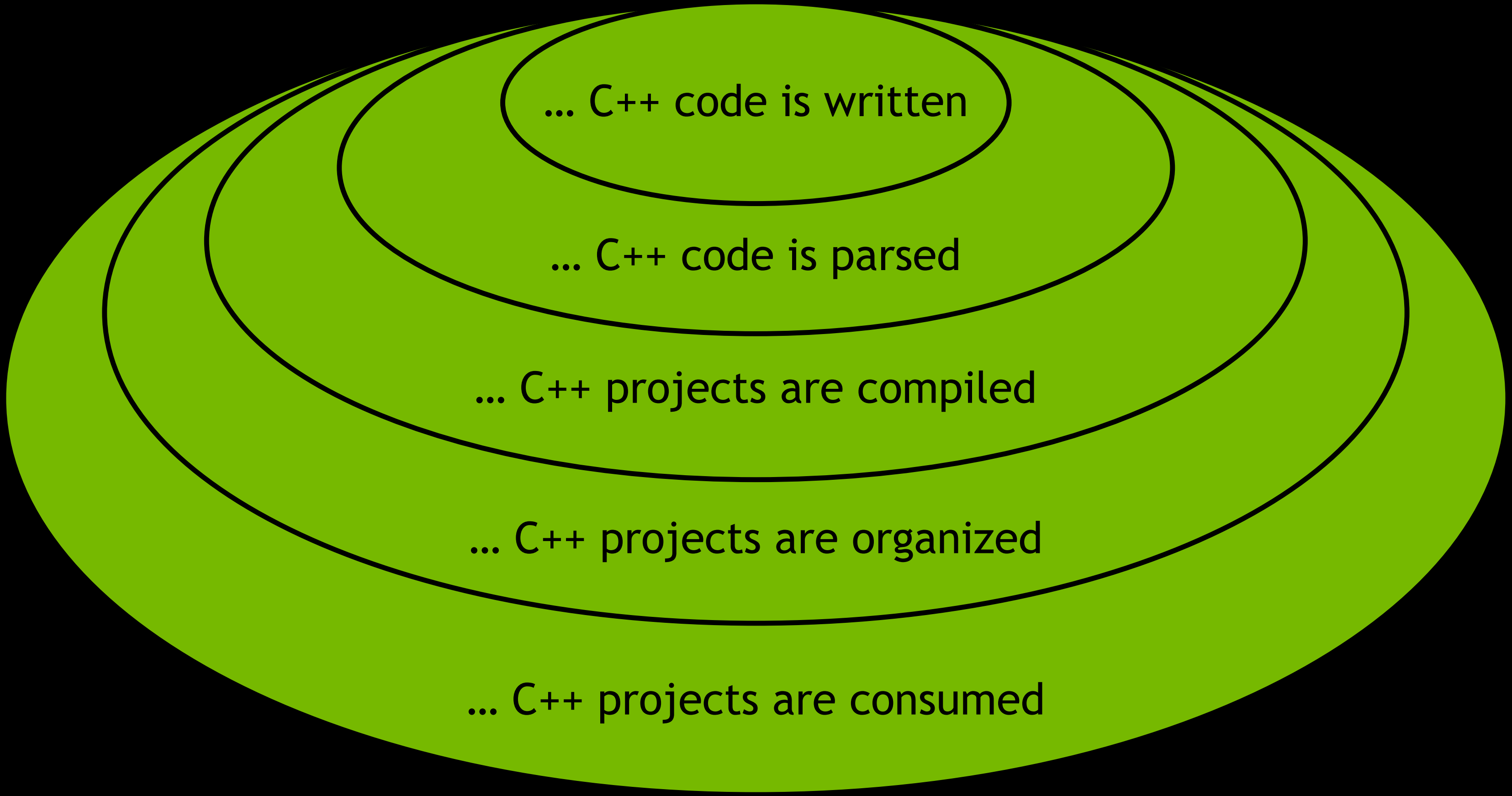
C++20's modules change how…



C++20's modules change how…



C++20’s modules change how…



Modules will have a greater impact than any other feature added post C++-98.

Today’s Compilation Model

What is C++’s compilation model today?

How do we organize C++ projects today?

"The text of the program is kept in units called ***source files*** in this International Standard."

[lex.separate] p1 s1

"A source file together with all the headers and source files included via the preprocessing directive `#include`, less any source lines skipped by any of the conditional inclusion preprocessing directives, is called a ***translation unit***."

[lex.separate] p1 s2

Kinds of Translation Units

	Example	Extension	Artifact	Notes
Non-Modular Unit	`#include “…”` …	`.cpp`	`.o`	

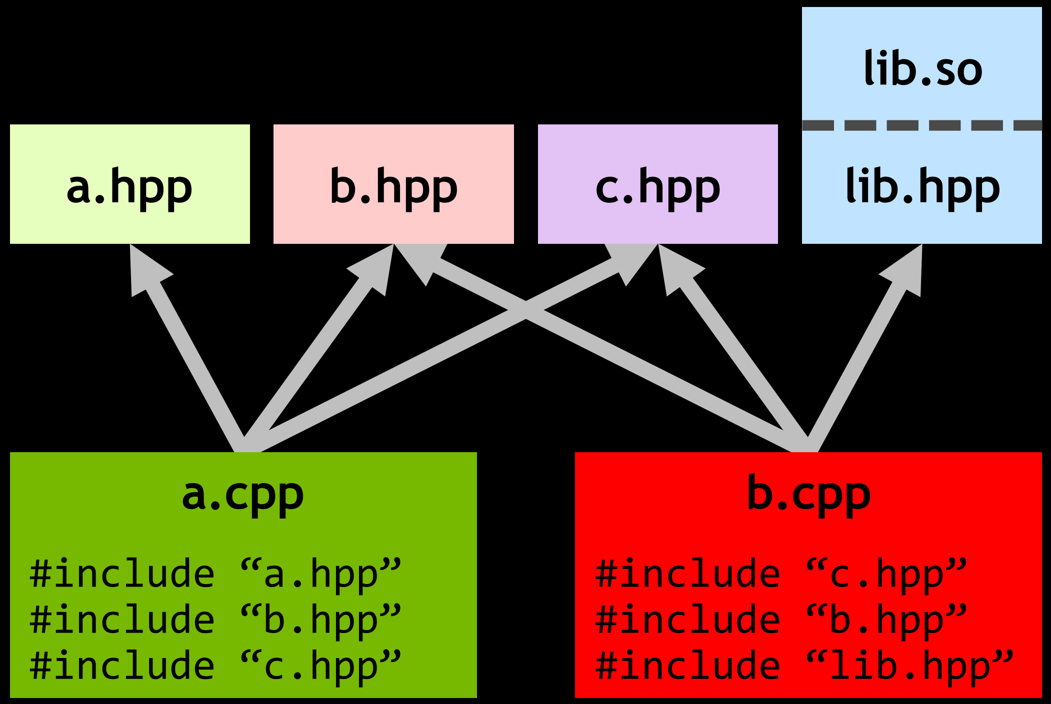
“A ***program*** consists of one or more *translation units* linked together.”

[basic.link] p1 s1

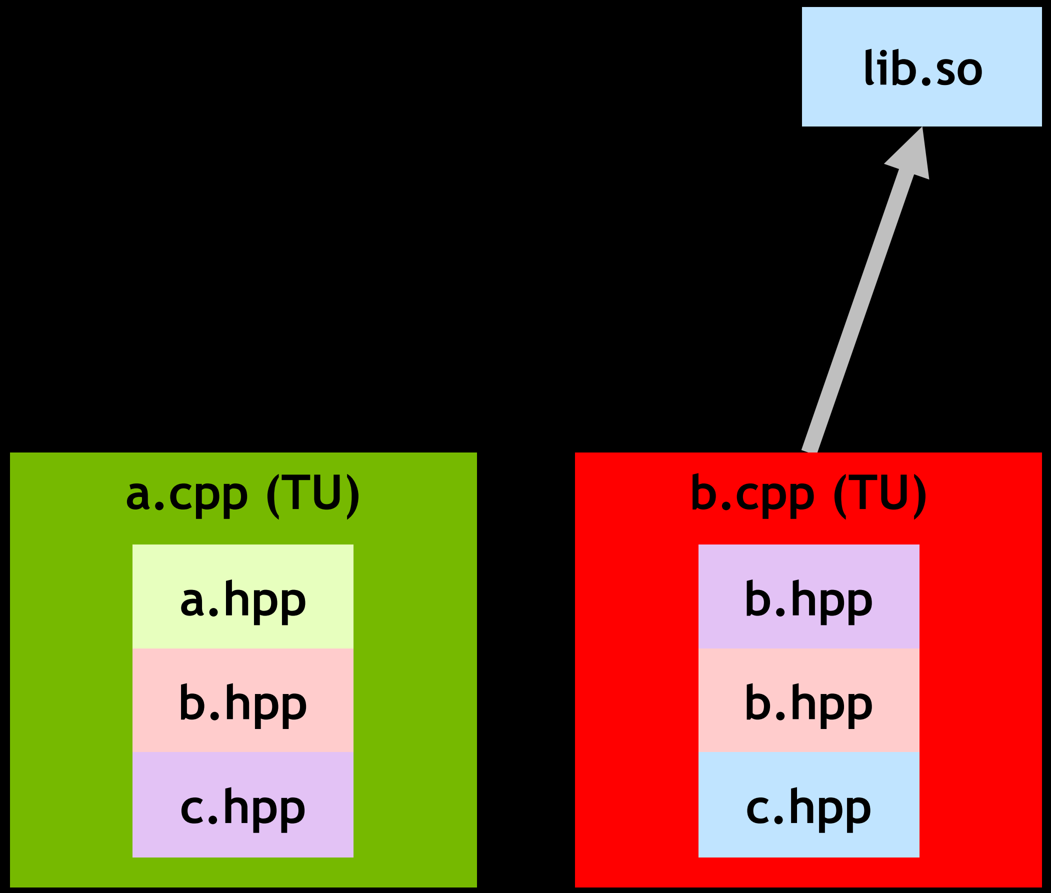
"Previously translated *translation units* and instantiation units can be preserved individually or in libraries."

[lex.separate] p2 s1

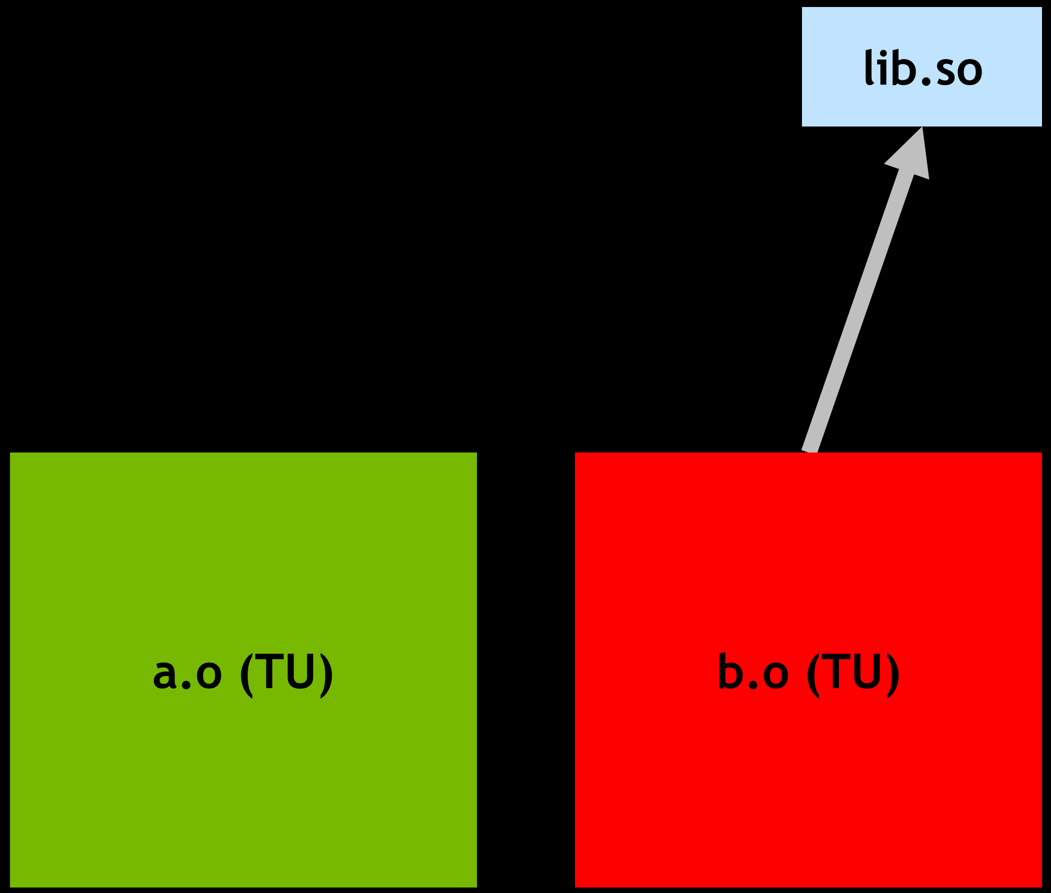
Textual Inclusion: Preprocess



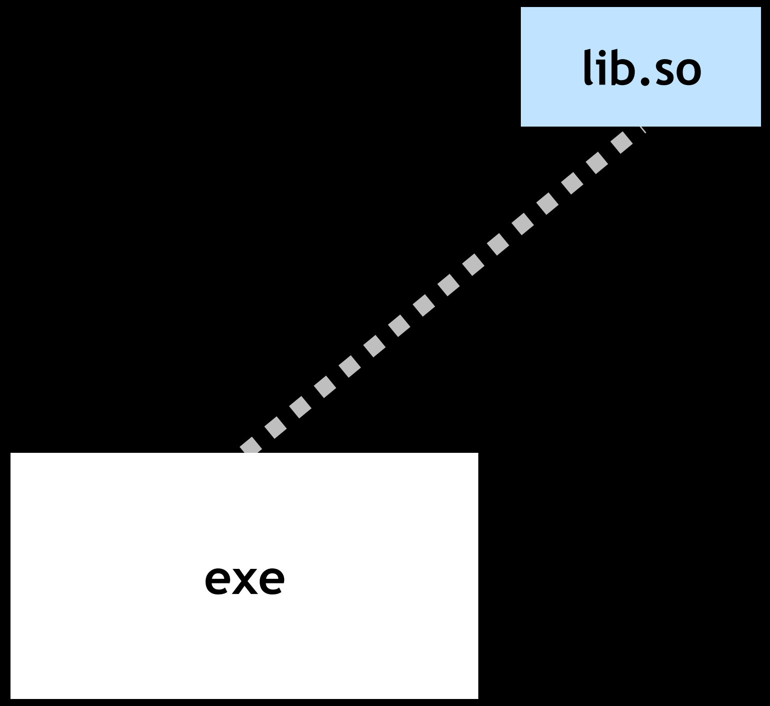
Textual Inclusion: Compile



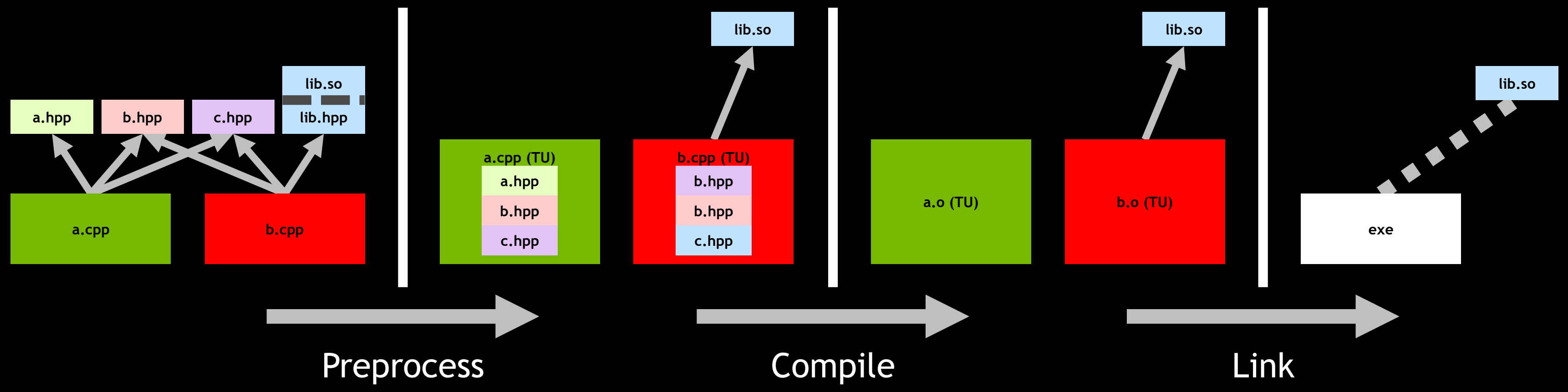
Textual Inclusion: Link



Textual Inclusion



Textual Inclusion



```
a.o: a.cpp
        $(CC) -c a.cpp -o a.o

b.o: b.cpp
        $(CC) -c b.cpp -o b.o

exe: a.o b.o
        $(CC) a.o b.o lib.so -o exe
```

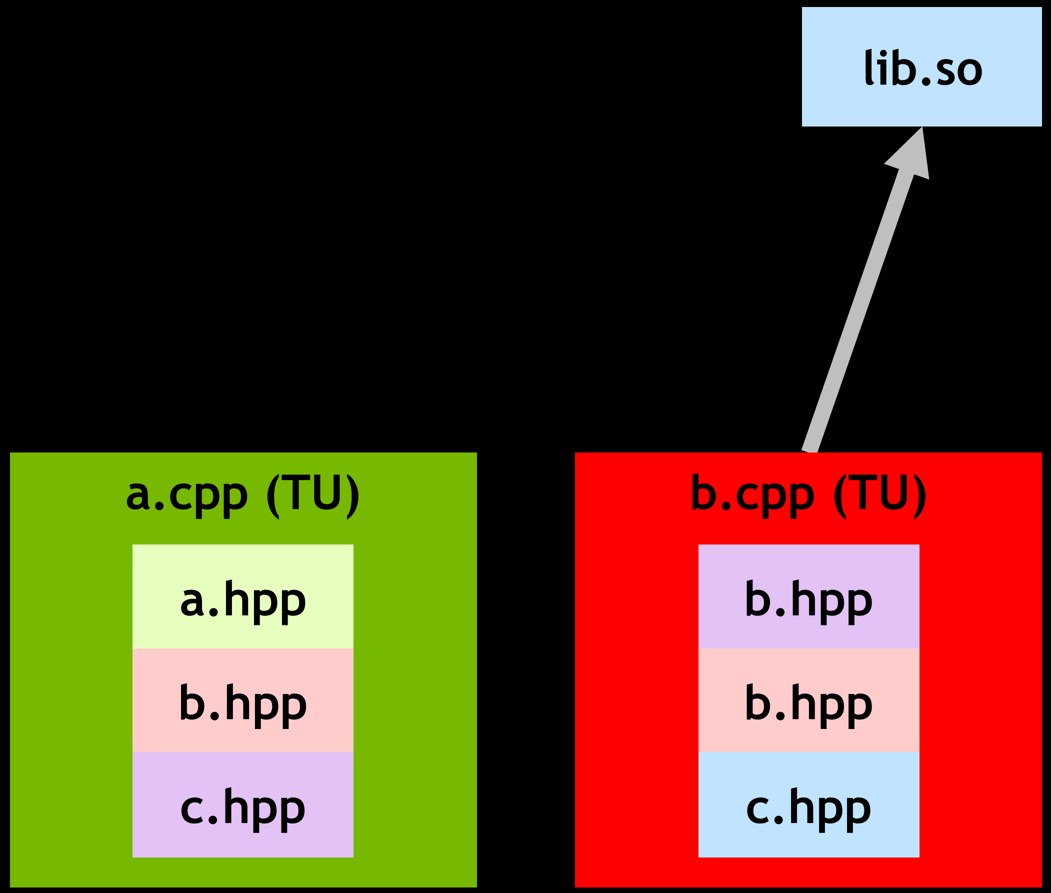
Headers are terrible:

- Slow to compile.
- ODR violations.
- Lack of encapsulation.
- Cyclic dependencies.
- Order dependent.

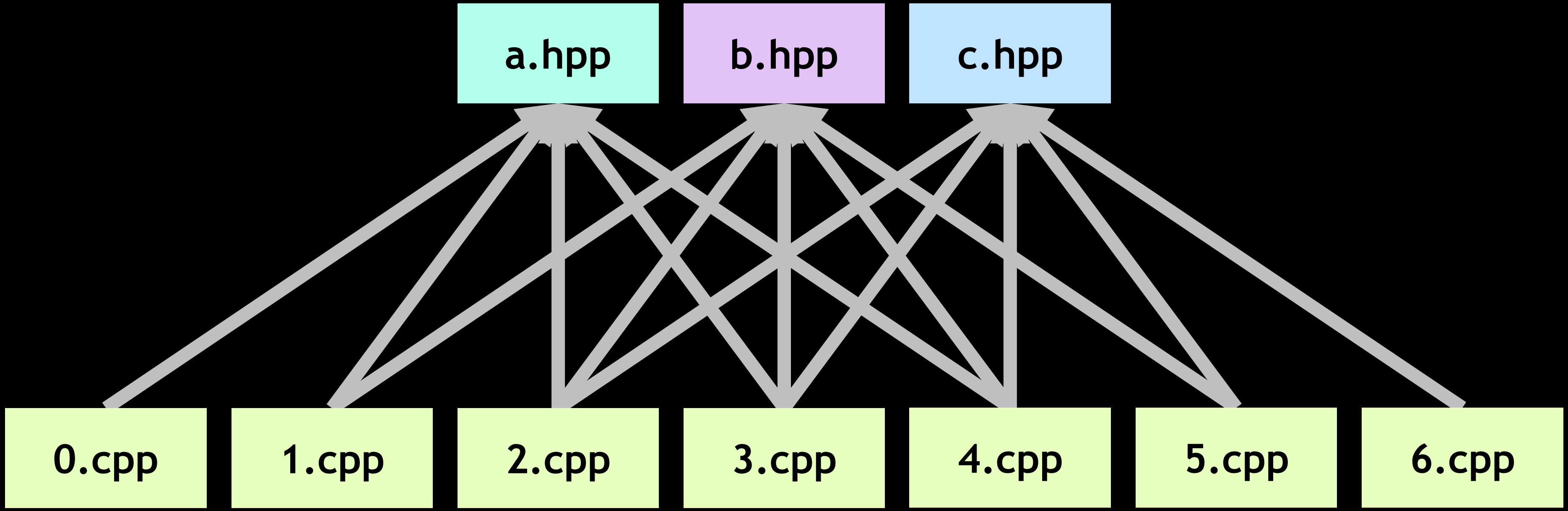
Headers are terrible:

- Slow to compile.
- ODR violations.
- Lack of encapsulation.
- Cyclic dependencies.
- Order dependent.

Textual Inclusion: Compile



Textual Inclusion



Pro: Embarrassingly parallel.

Con: a.hpp, b.hpp, and c.hpp are compiled 7 times.

Headers are terrible:

- Slow to compile.
- ODR violations.
- Lack of encapsulation.
- Cyclic dependencies.
- Order dependent.

"A variable, function, class type, enumeration type, or template shall not be defined where a prior definition is necessarily reachable; no diagnostic is required if the prior declaration is in another *translation unit*."

[basic.def.odr] p1

Ill formed, no diagnostic required

(IFNDR)

tree_node.hpp

```
#pragma once

template <typename T>
struct tree_node {
  T value;
  std::vector<tree_node*> children;
 #ifdef DEBUG
  tree_node* parent;
 #endif
};
```

a.cpp

```
#define DEBUG
#include "tree_node.hpp"

// …
```

b.cpp

```

#include "tree_node.hpp"

// …
```

Headers are terrible:

- Slow to compile.
- ODR violations.
- Lack of encapsulation.
- Cyclic dependencies.
- Order dependent.

a.hpp

```
#pragma once

namespace c {
 struct A {
  private:
   template <typename T>
   void impl();
 };

 namespace detail::unsupported {
  template <typename T>
  void __please_dont_use();
 }
}
```

Headers are terrible:

- Slow to compile.
- ODR violations.
- Lack of encapsulation.
- Cyclic dependencies.
- Order dependent.

a.hpp

```
#pragma once
#include “b.hpp”

struct Y;
struct X { Y* y; };
```

b.hpp

```
#pragma once
#include “a.hpp”

struct X;
struct Y { X* x; };
```

Headers are terrible:

- Slow to compile.
- ODR violations.
- Lack of encapsulation.
- Cyclic dependencies.
- Order dependent.

a.hpp

```
#pragma once

struct S { /* … */ };
```

b.hpp

```
#pragma once

void foo(S s);
```

c.cpp

```
#include “a.hpp”
#include “b.hpp”
```

a.hpp

```
#pragma once

struct S { /* … */ };
```

b.hpp

```
#pragma once

void foo(S s);
```

c.cpp

```
#include “a.hpp”
#include “b.hpp”
```

d.cpp

```
#include “b.hpp”
#include “a.hpp”
```

Headers are terrible:

- Slow to compile.
- ODR violations.
- Lack of encapsulation.
- Cyclic dependencies.
- Order dependent.

Using Modules

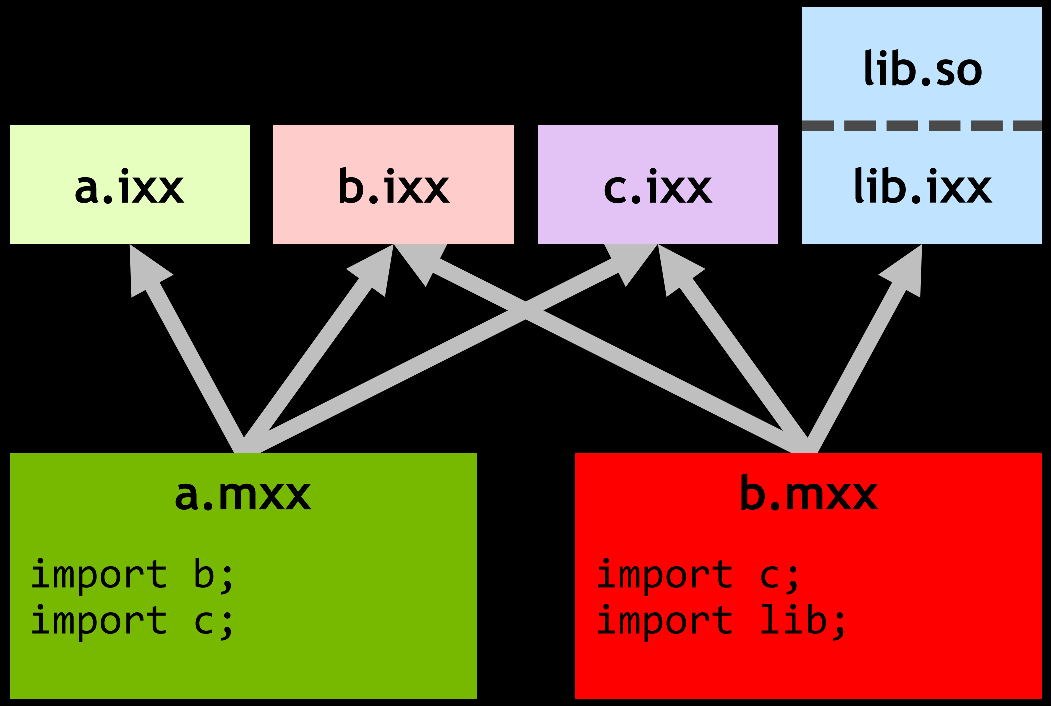
Textual Inclusion

```
#include “foo.hpp”
```

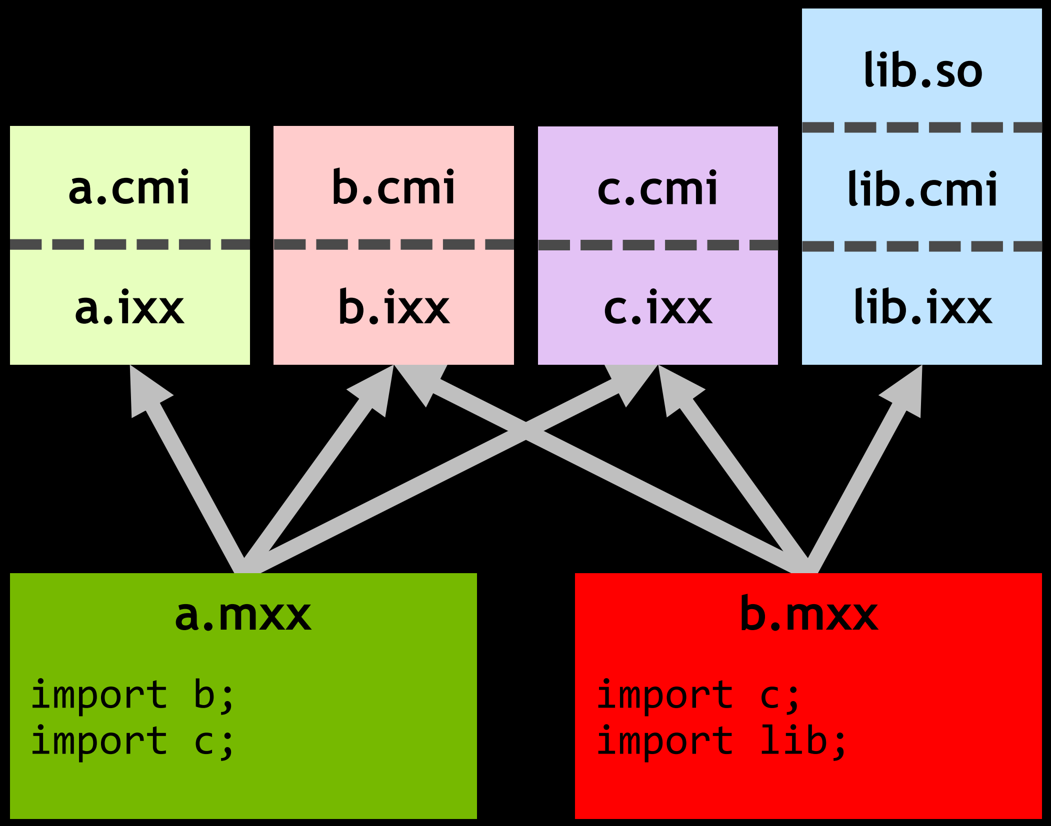
Modular Import

```
import foo;
```

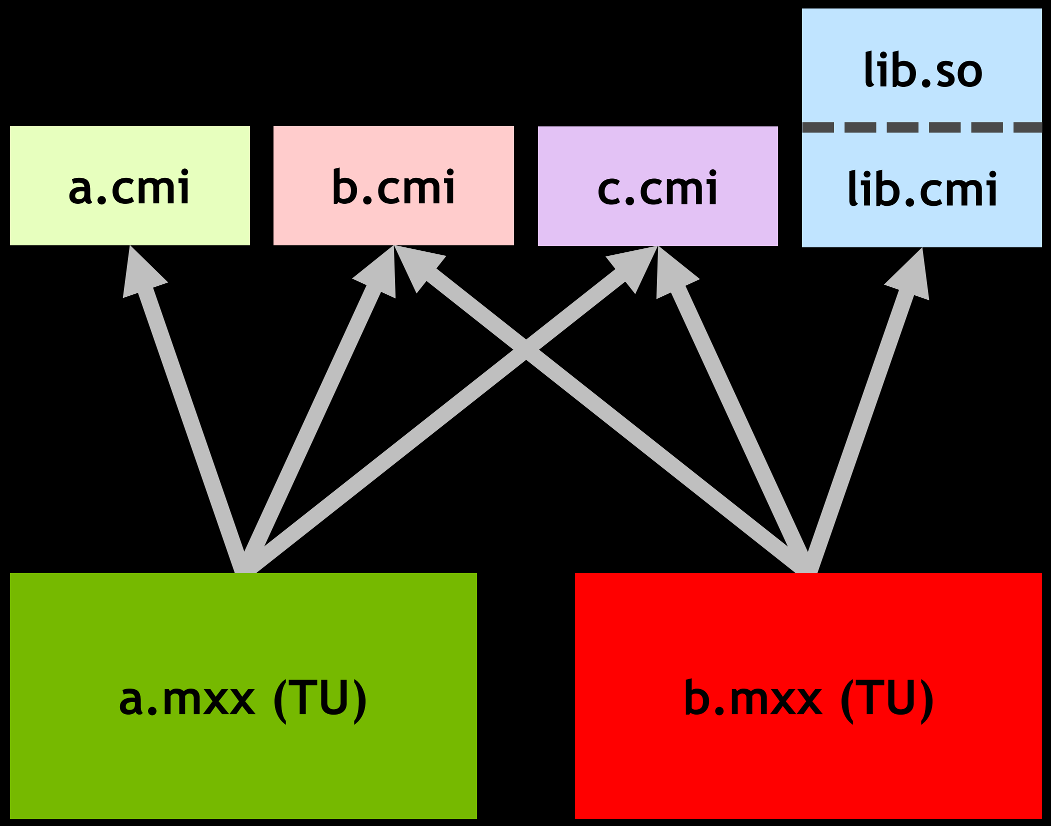
Modular Import: Precompile



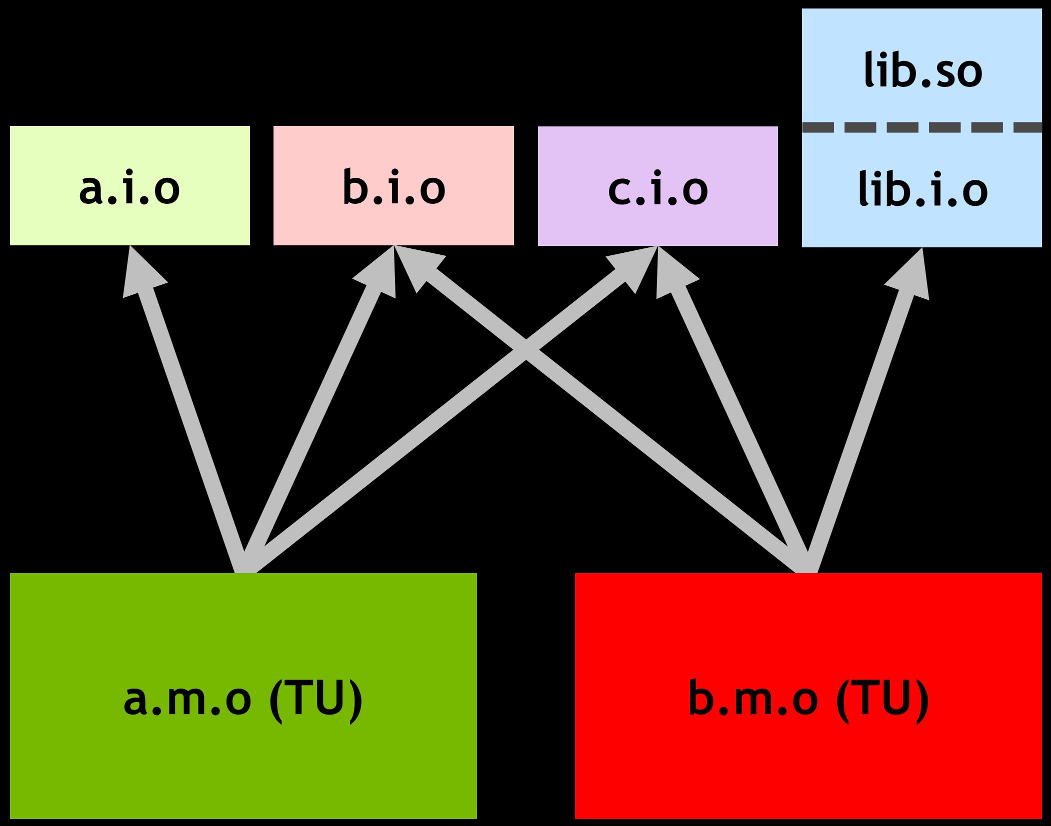
Modular Import: Preprocess



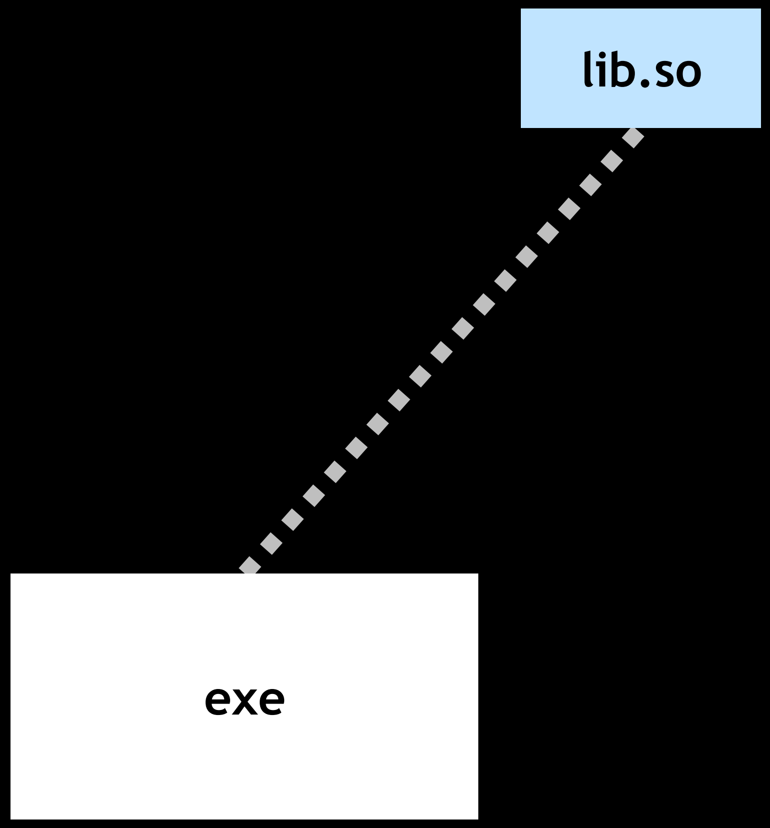
Modular Import: Compile



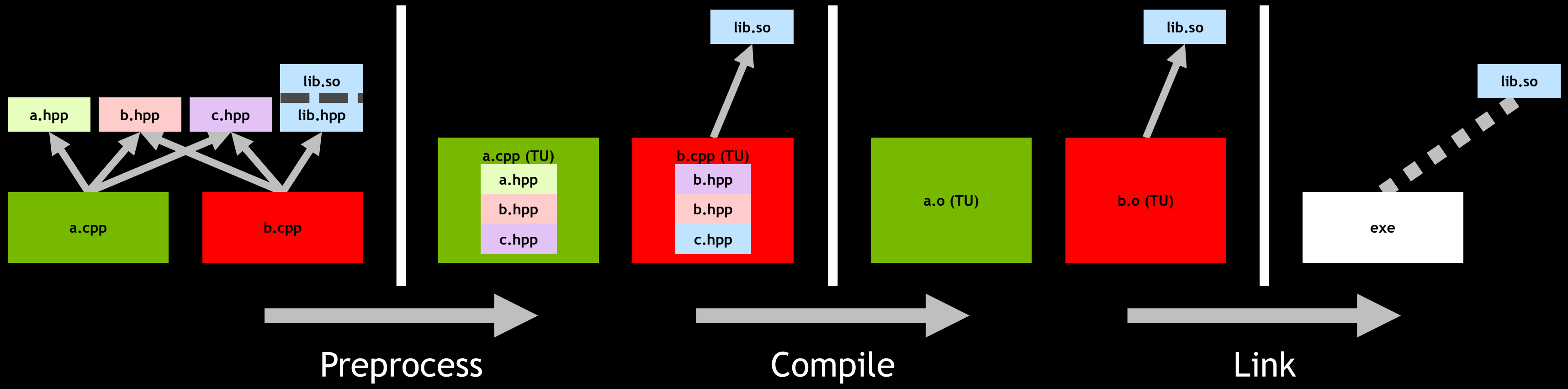
Modular Import: Link



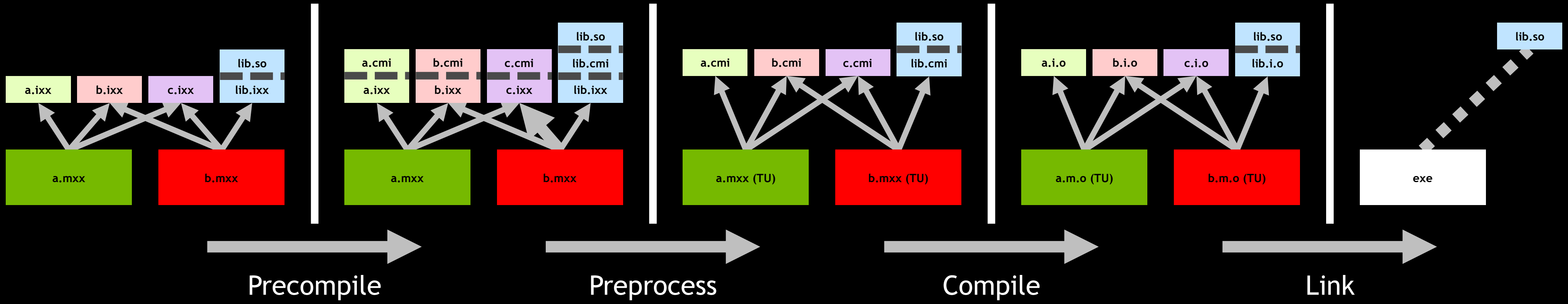
Modular Import



Textual Inclusion



Modular Import



Module Precompilation

```
a.cmi: a.ixx
        $(CC) --precompile a.ixx -o a.cmi

b.cmi: b.ixx
        $(CC) --precompile b.ixx -o b.cmi

c.cmi: c.ixx
        $(CC) --precompile c.ixx -o c.cmi

lib.cmi: lib.ixx
        $(CC) --precompile lib.ixx -o lib.cmi
```

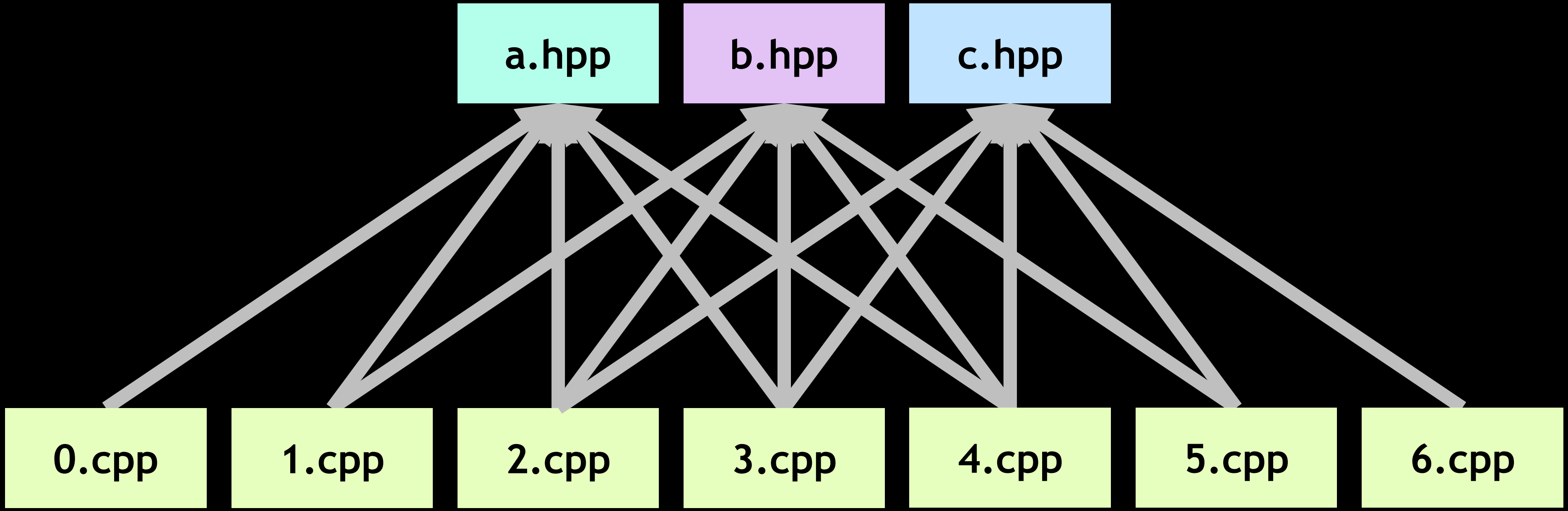
Module Interface Unit Compilation

```
a.i.o: a.cmi
        $(CC) -c a.cmi -o a.i.o

b.i.o: b.cmi
        $(CC) -c b.cmi -o b.i.o

c.i.o: c.cmi
        $(CC) -c c.cmi -o c.i.o

lib.i.o: lib.cmi
        $(CC) -c lib.cmi -o lib.i.o
```

Compilation and Link

```
a.m.o: a.mxx a.cmi b.cmi c.cmi
        $(CC) -c a.mxx -o a.m.o

b.m.o: b.mxx b.cmi c.cmi lib.cmi
        $(CC) -c b.mxx -o b.m.o

exe: a.m.o b.m.o a.i.o b.i.o c.i.o lib.i.o
        $(CC) a.o b.o a.i.o b.i.o c.i.o lib.i.o lib.so -o exe
```

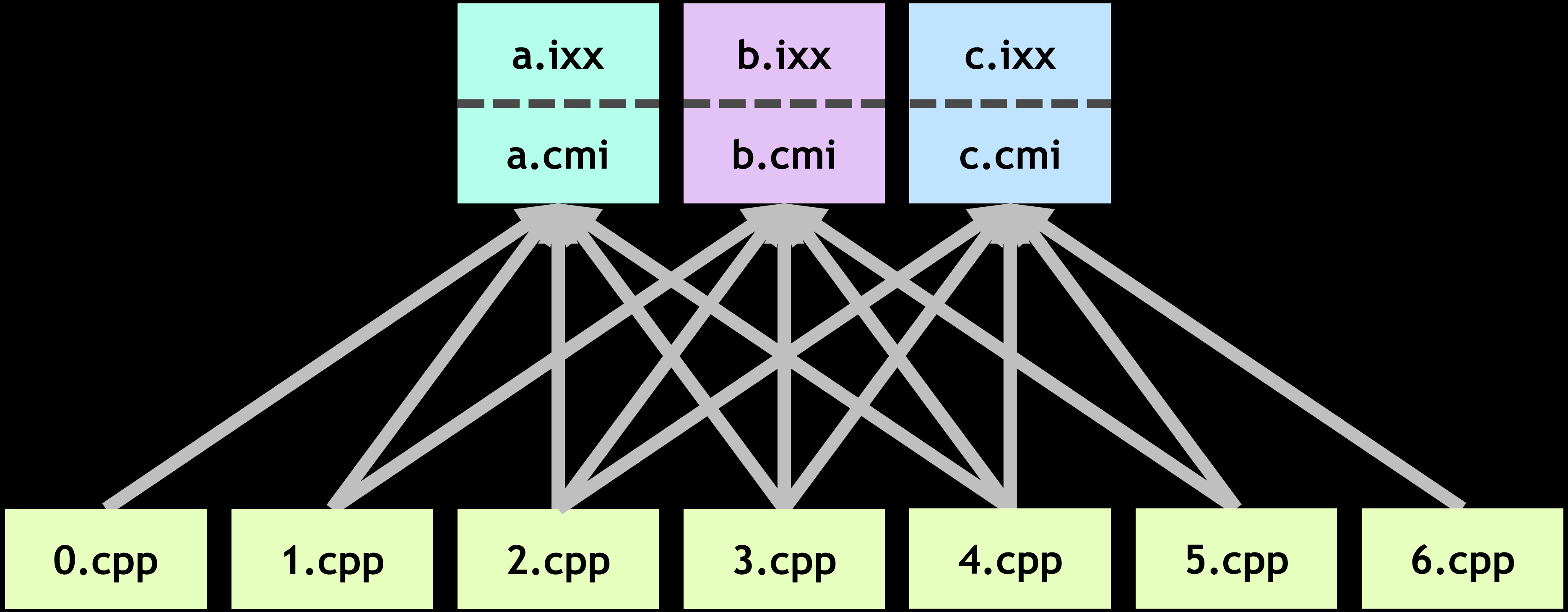
Textual Inclusion



Pro: Embarrassingly parallel.

Con: a.hpp, b.hpp, and c.hpp are compiled 7 times.

Modular Inclusion



Pro: a.ixx, b.ixx, and c.ixx are precompiled once.

Con: Not embarrassingly parallel.

Kinds of Translation Units

	Example	Extension	Artifact	Notes
Non-Modular Unit	`#include "…"` `…`	`.cpp`	`.o`	
Module Interface Unit	`export module …;` `…`	`.ixx`	`.cmi` `.o (optional)`	Exactly one per module.
Module Implementation Unit	`module …;` `…`	`.mxx`	`.o`	At most one per module.

Textual Inclusion

math.hpp

```
#pragma once

int square(int a);
```

math.cpp

```
#include “math.hpp”

int square(int a) { return a * a; }
```

main.cpp

```
#include “math.hpp”

int main() { return square(42); }
```

Modular Import

math.ixx

```
export module math;

export int square(int a);
```

math.mxx

```
module math;

int square(int a) { return a * a; }
```

main.cpp

```
import math;

int main() { return square(42); }
```

Textual Inclusion

math.hpp

```
#pragma once

template <typename T>
T square(T a) { return a * a; }
```

main.cpp

```
import square;

int main() { return square(42); }
```

Modular Import

math.ixx

```
export module math;

export template <typename T>
T square(T a) { return a * a; }
```

main.cpp

```
import math;

int main() { return square(42); }
```

Textual Inclusion

```
#include <foo.hpp>
#include “foo.hpp”
```

Modular Import

```
import foo;
import <foo.hpp>;
import “foo.hpp”;
```

Importable headers:

- Most C++ standard library headers*.
- Some system headers.
- Headers you proclaim importable†.

* C standard library headers (`<cfoo>`, `<foo.h>`) are not required to be importable.

† The mechanism for indicating which headers are importable is implementation defined.

assert.h

```
#ifdef NDEBUG
#define assert(condition) ((void)0)
#else
#define assert(condition) /* … */
#endif
```

cstddef

```
namespace std { /* ... */ }
#define NULL /*see definition*/
#define offsetof(P, D) /*see definition*/
```

“If the header identified by the *header-name* denotes an *importable header*, the preprocessing directive is instead replaced by the *preprocessing-tokens*

```
import header-name ;
```

”

[cpp.include] p7

Your Code

```
#include <vector>
#include <iostream>

// …
```

Your Code

```
#include <vector>
#include <iostream>

// …
```

Compiler Interpretation

```
import <vector>;
import <iostream>;

// …
```

Kinds of Translation Units

	Example	Extension	Artifact	Notes
Non-Modular Unit	#include "…" …	.cpp	.o	
Header Unit	// Created by: import <…>;	.hpp	.cmi .o (optional)	
Module Interface Unit	export module …; …	.ixx	.cmi .o (optional)	Exactly one per module.
Module Implementation Unit	module …; …	.mxx	.o	At most one per module.

Writing Modules

```
import boost.spirit;
import ctre;
import blas.level1;
```

Module names are dot-separated identifiers.

```
import boost.spirit;
import ctre;
import blas.level1;
```

Module names are dot-separated identifiers.
Dots in module names have no semantic meaning.

“A ***module unit*** is a *translation unit* that contains a *module-declaration*.”

[module.unit] p1 s1

```
export module a;
// …
```

Module Interface Unit

```
module a;
// …
```

Module Implementation Unit

```
module a;
// …
module b;
// …
```

Only one module declaration per translation unit.

```
export module a;
// …
export module b;
// …
```

Only one module declaration per translation unit.

“In a *module unit*, all *module-import-declarations* shall precede all other *top-level-declarations* in the *top-level-declaration-seq* of the *translation-unit* …”

[module.import] p1 s1

Module Unit Structure

```
export module …;
import …;

…
```

```
export declaration

export {
  declaration …
}
```

```
export void f();

export struct A;

export int i{0};
```

```
export {
  void f();
  struct A;
  int i{0};
}
```

```
export template <typename T>
T square(T t) { return t * t; }
```

```
export template <typename T>
struct is_const : false_type {};

export template <typename T>
struct is_const<T const> : true_type {};
```

```
export namespace foo { struct A; }

namespace foo { struct B; }
```

```
export namespace foo { struct A; }

namespace foo { struct B; }
```

Only `foo::A` is exported.

```
export typedef int int32_t;

export using unsigned uint32_t;
```

```
struct A { /* … */ };

// ...

export using A;
```

```
export import a;
```

square.ixx

```cpp
export module square;

export template <typename T>
T square(T a) { return a * a; }
```

add.ixx

```cpp
export module add;

export template <typename T>
T add(T a, T b) { return a + b; }
```

math.ixx

```cpp
export module math;

export import square;
export import add;
```

a.ixx

```cpp
export module a;

struct S { int m; };
export S foo();
```

main.cpp

```cpp
import a;

int main() {
  auto s0 = foo();
  s0.m = 42;
}
```

a.ixx

```
export module a;

struct S { int m; };
export S foo();
```

main.cpp

```
import a;

int main() {
  auto s0 = foo();
  s0.m = 42;

  S s1{};
}
```

a.ixx

```
export module a;

struct S { int m; };
export S foo();
```

main.cpp

```
import a;

int main() {
  auto s0 = foo();
  s0.m = 42;

  decltype(foo()) s1{};
}
```

Textual Inclusion

functional

```
#pragma once




namespace std {
 export template <typename F, typename... Args>
 /* unspecified */    bind(F&& f, Args&&... args);
}
```

Textual Inclusion

functional

```
#pragma once

namespace std {
 template <typename F, typename... Args>
 struct __binder;

 template <typename F, typename... Args>
 __binder<F, Args...> bind(F&& f, Args&&... args);
}
```

Textual Inclusion

main.cpp

```
#include <functional>

int main()
{
  using namespace std::placeholders;

  auto add_four0 = std::bind(std::plus{}, _1, 4);

  std::__binder<std::plus<>, …, int> add_four1
    = std::bind(std::plus{}, _1, 4);
}
```

Modular Import

functional.ixx

```
export module std.functional;




namespace std {
 export template <typename F, typename... Args>
 /* unspecified */    bind(F&& f, Args&&... args);
}
```

Modular Import

functional.ixx

```
export module std.functional;

namespace std {
 template <typename F, typename... Args>
 struct __binder;

 export template <typename F, typename... Args>
 __binder<F, Args...> bind(F&& f, Args&&... args);
}
```

Modular Import

main.cpp

```
import std.functional;

int main()
{
  using namespace std::placeholders;

  auto add_four0 = std::bind(std::plus{}, _1, 4);

  std::__binder<std::plus<>, …, int> add_four1
    = std::bind(std::plus{}, _1, 4);
}
```

Visible: In scope, can be named.

Reachable: In scope, not necessarily namable.

a.ixx

```
export module a;

struct S { int m; };
export S foo();
```

main.cpp

```
import a;
```

In `main.cpp`:

- `S` is reachable.
- `foo` is reachable and visible.

Modules enable true encapsulation.

Textual Inclusion

a.hpp

```
#pragma once

// Implementation detail/not part of the API.
template <typename T>
T id(T t) { return t; }
```

b.hpp

```
#pragma once

// Implementation detail/not part of the API.
template <typename T>
struct id { using type = T; };
```

main.cpp

```
#include “a.hpp”;
#include “b.hpp”;
```

Textual Inclusion

a.hpp

```
#pragma once

// Implementation detail/not part of the API.
template <typename T>
T id(T t) { return t; }
```

b.hpp

```
#pragma once

// Implementation detail/not part of the API.
template <typename T>
struct id { using type = T; };
```

main.cpp

```
#include “a.hpp”;
#include “b.hpp”;
```

Textual Inclusion

a.hpp

```
#pragma once

namespace a::detail {
 template <typename T>
 T id(T t) { return t; }
}
```

b.hpp

```
#pragma once

namespace b::detail {
 template <typename T>
 struct id { using type = T; };
}
```

main.cpp

```
#include “a.hpp”;
#include “b.hpp”;
```

Textual Inclusion

a.hpp

```
#pragma once

namespace a::detail {
 template <typename T>
 T __dont_use_id(T t) { return t; }
}
```

b.hpp

```
#pragma once

namespace b::detail {
 template <typename T>
 struct __dont_use_id { using type = T; };
}
```

main.cpp

```
#include “a.hpp”;
#include “b.hpp”;
```

Textual Inclusion

a.hpp

```
#pragma once

namespace a::detail {
 template <typename T>
 T __dont_use_id(T t) { return t; }
}
```

b.hpp

```
#pragma once

namespace b::detail {
 template <typename T>
 struct __dont_use_id { using type = T; };
}
```

main.cpp

```
#include “a.hpp”;
#include “b.hpp”;
```

Modular Import

a.ixx

```
export module a;

template <typename T>
T id(T t) { return t; }
```

b.ixx

```
export module b;

template <typename T>
struct id { using type = T; };
```

main.cpp

```
import a;
import b;
```

No more `detail`/`impl` namespaces.

No more uglifying identifiers.

"A name is said to have ***linkage*** when it might denote the same object, reference, function, type, template, namespace or value as a name introduced by a declaration in another scope:

- When a name has ***external linkage***, the entity it denotes can be referred to by names from scopes of other *translation units* or from other scopes of the same *translation unit*.
- When a name has ***module linkage***, the entity it denotes can be referred to by names from other scopes of the same *module unit* or from scopes of other module units of that same module.
- When a name has ***internal linkage***, the entity it denotes can be referred to by names from other scopes in the same *translation unit*.
- When a name has ***no linkage***, the entity it denotes cannot be referred to by names from other scopes."

[basic.link] p4

Kinds of Linkage

	Example	Visible From	Notes

External Linkage	`extern void foo();`
	`export void bar();`
	`extern int i{};`
	`export bool b{};`
		Other translation units.	

Module Linkage	`struct S;`
	`int foo();`
	`int i{};`
		This module.	In non-modular units, entities with module linkage have external linkage.

Internal Linkage	`static void foo();`
	`static int i{};`
	`bool const b{};`
	`namespace { /* … */ }`
		This translation unit.	

No Linkage	`int main() {`
	`  int i{};`
	`}`
		This scope.	

Modules are sandboxed.

Textual Inclusion

a.cpp

```
#pragma once

struct foo { /* … */ };
```

b.hpp

```
#pragma once

void bar(foo f);
```

main.cpp

```
#include “a.hpp”
#include “b.hpp”
```

Textual Inclusion

a.hpp

```
#pragma once

struct foo { /* … */ };
```

b.hpp

```
#pragma once

void bar(foo f);
```

main.cpp

```
#include “a.hpp”
#include “b.hpp”
```

Modular Import

a.ixx

```
export module a;

export struct foo { /* … */ };
```

b.ixx

```
export module b;

export void bar(foo f);
```

main.cpp

```
import a;
import b;
```

Textual Inclusion

a.hpp

```
#pragma once

struct foo { /* … */ };
```

b.hpp

```
#pragma once

void bar(foo f);
```

main.cpp

```
#include “a.hpp”
#include “b.hpp”
```

Modular Import

a.ixx

```
export module a;

export struct foo { /* … */ };
```

b.ixx

```
export module b;

export void bar(foo f);
```

main.cpp

```
import a;
import b;
```

a.ixx

```
export module a;

export struct foo{};
```

main.cpp

```
#define foo bar
import a;
#undef foo

foo f{};
```

a.ixx

```
export module a;

export struct foo{};
```

main.cpp

```
#define foo bar
import a;
#undef foo

foo f{};
```

The definition of `foo` isn’t seen by the imported module.

a.ixx

```
export module a;

#if defined(DEBUG)
  // …
#else
  // …
#endif
```

main.cpp

```
#define DEBUG
import a;
```

a.ixx

```
export module a;

#if defined(DEBUG)
  // …
#else
  // …
#endif
```

main.cpp

```
#define DEBUG
import a;
```

The definition of DEBUG isn’t seen by the imported module.

```
#define _LIBCPP_NO_EXCEPTIONS
import <vector>;
```

```
#define _LIBCPP_NO_EXCEPTIONS
import <vector>;
```

The definition of `_LIBCPP_NO_EXCEPTIONS` isn’t seen by `<vector>`.

“If the header identified by the *header-name* denotes an *importable header*, the preprocessing directive is instead replaced by the *preprocessing-tokens*

```
import header-name ;
```

”

[cpp.include] p7

```
#define _LIBCPP_NO_EXCEPTIONS
#include <vector>;
```

```
#define _LIBCPP_NO_EXCEPTIONS
#include <vector>;
```

The definition of `_LIBCPP_NO_EXCEPTIONS` isn't seen by `<vector>`.

```
#define _LIBCPP_NO_EXCEPTIONS
#include <vector>;

#define NDEBUG
#include <assert.h>;

// For `readlink`.
#define _XOPEN_SOURCE
#include <unistd.h>
```

How do we deal with these non-modular headers?

Macros defined on the command line (`-DFOO=…`) are seen.

Textual Inclusion

a.hpp

```
#pragma once
#define _XOPEN_SOURCE // For `readlink`.
#include <unistd.h>

// …
```

Modular Import

a.ixx

```
export module a;
#define _XOPEN_SOURCE // For `readlink`.
import <unistd.h>;

// …
```

Textual Inclusion

a.hpp

```
#pragma once
#define _XOPEN_SOURCE // For `readlink`.
#include <unistd.h>

// …
```

Modular Import

a.ixx

```
export module a;
#define _XOPEN_SOURCE // For `readlink`.
#include <unistd.h>;

// …
```

Textual Inclusion

a.hpp

```
#pragma once
#define _XOPEN_SOURCE // For `readlink`.
#include <unistd.h>

// …
```

Modular Import

a.ixx

```
module;
#define _XOPEN_SOURCE // For `readlink`.
#include <unistd.h>;
export module a;

// …
```

```
module;
#include <boost/circular_buffer>
export module boost.circular_buffer;
namespace boost {
  export using ::boost::circular_buffer;
}
```

Module Unit Structure

```
module;
#pp-directive …;
export module …;
import …;
…
```

Modules are order independent.

```
import a;
import b;
```

==

```
import b;
import a
```

Modules cannot have cycles.

Textual Inclusion

a.hpp

```
#pragma once
#include “b.hpp”

struct Y;
struct X { Y* y; };
```

b.hpp

```
#pragma once
#include “a.hpp”

struct X;
struct Y { X* x; };
```

Textual Inclusion

Modular Import

a.hpp

```
#pragma once
#include “b.hpp”

struct Y;
struct X { Y* y; };
```

b.hpp

```
#pragma once
#include “a.hpp”

struct X;
struct Y { X* x; };
```

a.ixx

```
export module a;
import b;

struct Y;
export struct X { Y* y; };
```

b.ixx

```
export module b;
import a;

struct X;
export struct Y { X* x; };
```

Textual Inclusion

a.hpp

```
#pragma once
#include “b.hpp”

struct Y;
struct X { Y* y; };
```

b.hpp

```
#pragma once
#include “a.hpp”

struct X;
struct Y { X* x; };
```

Modular Import

a.ixx

```
export module a;
import b;

struct Y;
export struct X { Y* y; };
```

b.ixx

```
export module b;
import a;

struct X;
export struct Y { X* x; };
```

Textual Inclusion

a.hpp

```
#pragma once
#include “b.hpp”

struct Y;
struct X { Y* y; };
```

b.hpp

```
#pragma once
#include “a.hpp”

struct X;
struct Y { X* x; };
```

Modular Import

a.ixx

```
export module a;
import b;

struct Y;
export struct X { Y* y; };
```

b.ixx

```
export module b;
import a;

struct X;
export struct Y { X* x; };
```

Modular Import

a.ixx

```
export module a;

struct Y;
export struct X { Y* y; };
```

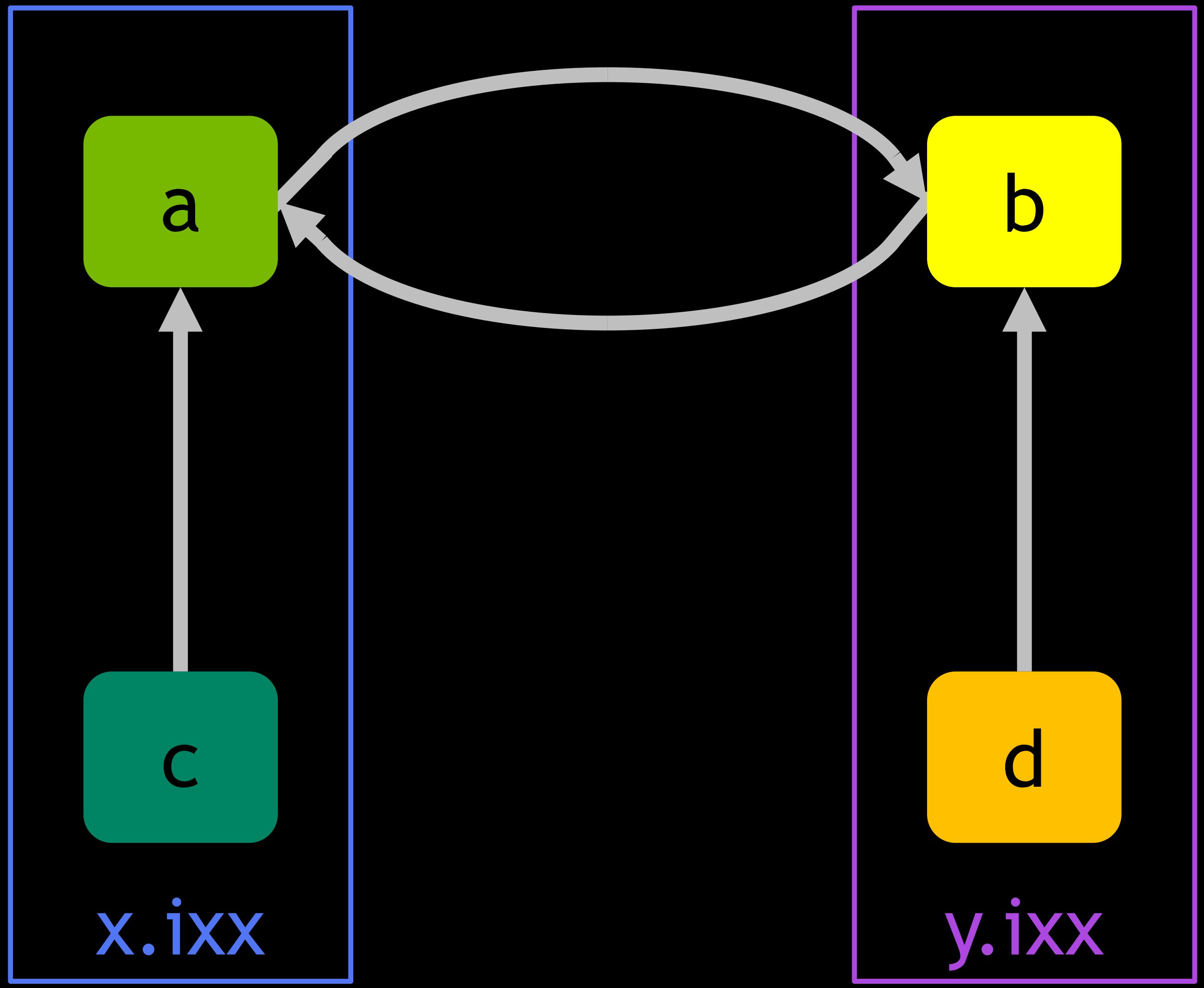
b.ixx

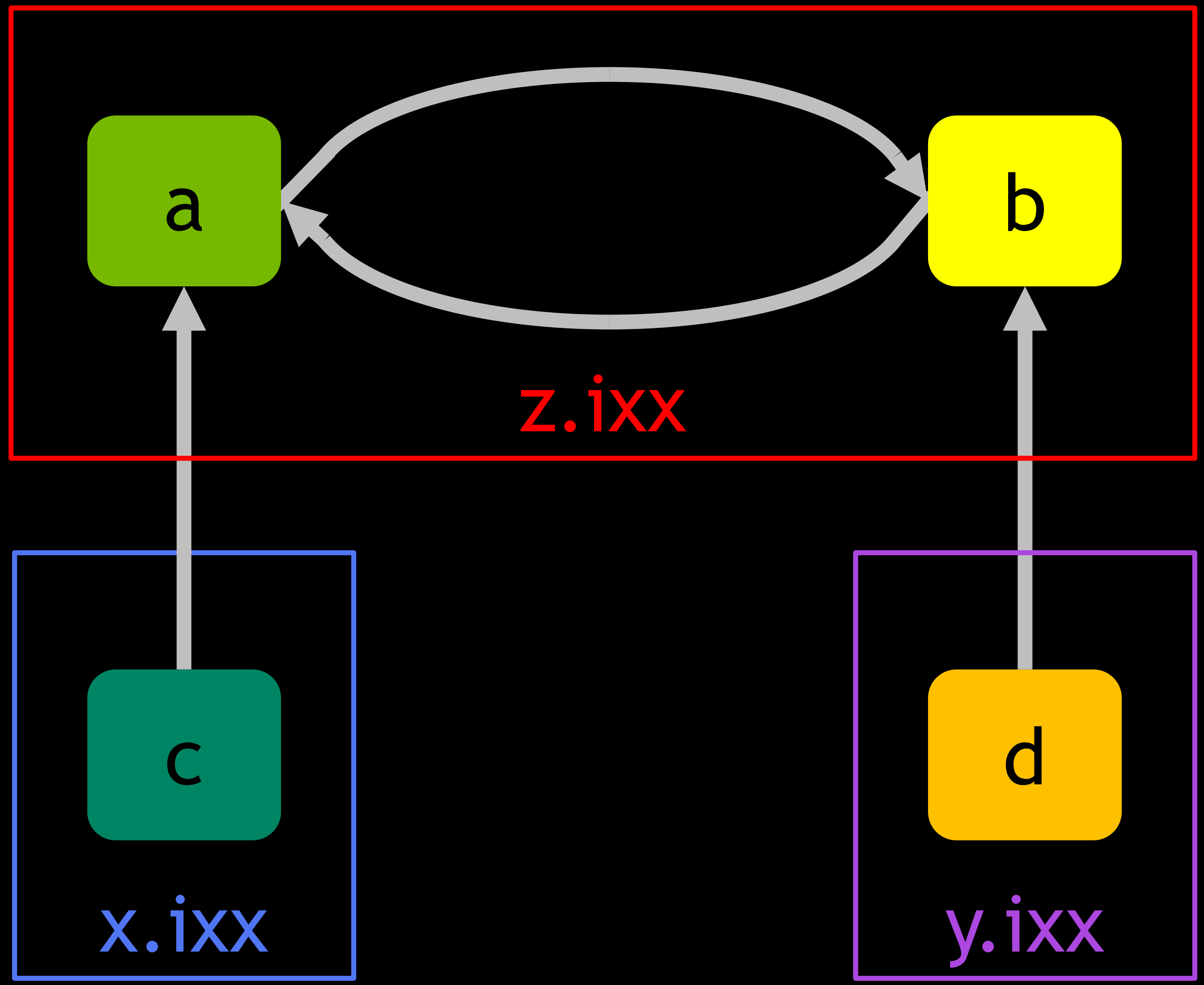
```
export module b;

struct X;
export struct Y { X* x; };
```

main.cpp

```
import a;
import b;
```

How do we break cycles?


a
b
c
d
x.ixx
y.ixx


a
b
z.ixx
c
x.ixx
d
y.ixx

Modules own their declarations.

Textual Inclusion

a.hpp

```
#pragma once

void foo();
```

a.cpp

```
#include “a.hpp”

void foo() { /* … */ }
```

b.cpp

```
#include “a.hpp”

static void foo() { /* … */ }
```

Modular Import

a.ixx

```
export module a;

export void foo();
```

a.cpp

```
module a;

void foo() { /* … */ }
```

b.cpp

```
import a;

static void foo() { /* … */ }
```

Textual Inclusion

a.hpp
```
#pragma once

void foo();
```

a.cpp
```
#include “a.hpp”

void foo() { /* … */ }
```

b.cpp
```
#include “a.hpp”

static void foo() { /* … */ }
```

Modular Import

a.ixx
```
export module a;

export void foo();
```

a.cpp
```
module a;

void foo() { /* … */ }
```

b.cpp
```
import a;

static void foo() { /* … */ }
```

"If a declaration would redeclare a reachable declaration attached to a different module, the program is ill-formed. As a consequence of these rules, all declarations of an entity are attached to the same module; the entity is said to be ***attached*** to that module."

[basic.link] p12

"If multiple declarations of the same name with *external linkage* would declare the same entity except that they are attached to different modules, the program is ill-formed; no diagnostic is required."

[basic.link] p11 s2

“[*Note*: *using-declarations*, *typedef declarations*, and *alias-declarations* do not declare entities, but merely introduce synonyms. Similarly, *using-directives* do not declare entities. — *end note*]”

[basic.link] p11 s3, s4

a.ixx

```
export module a;

export void foo();
```

b.ixx

```
export module b;

export void foo();
```

a.ixx

```
export module a;

export void foo();
```

b.ixx

```
export module b;

export void foo();
```

Ill formed, no diagnostic required (IFNDR).

Modules can be contained in one file.

Textual Inclusion

a.hpp

```
#pragma once

struct pimpl;
```

a.cpp

```
#include “a.hpp”

struct pimpl { /* … */ };
```

Modular Import

a.ixx

```
export module a;

export struct pimpl;
```

a.cpp

```
module a;

struct pimpl { /* … */ };
```

Textual Inclusion

a.hpp

```
#pragma once

struct pimpl;
```

a.cpp

```
#include “a.hpp”

struct pimpl { /* … */ };
```

Modular Import

a.ixx

```
export module a;

export struct pimpl;

module : private;

struct pimpl { /* … */ };
```

Module Unit Structure

```
module;
#pp-directive …;
export module …;
import …;
…
module : private;
…
```

Modules can be split across multiple files.

square.ixx

```
export module square;

export template <typename T>
T square(T a) { return a * a; }
```

add.ixx

```
export module add;

export template <typename T>
T add(T a, T b) { return a + b; }
```

math.ixx

```
export module math;

export import square;
export import add;
```

square.ixx

```
export module math:square;

export template <typename T>
T square(T a) { return a * a; }
```

add.ixx

```
export module math:add;

export template <typename T>
T add(T a, T b) { return a + b; }
```

math.ixx

```
export module math;

export import :add;
export import :square;
```

math_vector.ixx

```
export module math.vector;
export import :dot_product;
float sqrt(float x);
```

math_vector.ixx

```
export module math.vector;
export import :dot_product;
float sqrt(float x);
```

math_vector_dot_product.ixx

```
export module math.vector:dot_product;
import :sum_of_squares;
import <span>;
export float vector_norm(std::span<float> x) {
  return sqrt(sum_of_squares(x));
}
```

math_vector.ixx

```
export module math.vector;
export import :dot_product;
float sqrt(float x);
```

math_vector_dot_product.ixx

```
export module math.vector:dot_product;
import :sum_of_squares;
import <span>;
export float vector_norm(std::span<float> x) {
  return sqrt(sum_of_squares(x));
}
```

math_vector_sum_of_squares.mxx

```
module math.vector:sum_of_squares;
import <span>;
import <algorithm>;
float sum_of_squares(std::span<float> x) {
  return std::transform_reduce(x);
}
```

math_vector.ixx

```
export module math.vector;
export import :dot_product;
float sqrt(float x);
```

math_vector_dot_product.ixx

```
export module math.vector:dot_product;
import :sum_of_squares;
import <span>;
export float vector_norm(std::span<float> x) {
  return sqrt(sum_of_squares(x));
}
```

math_vector_sum_of_squares.mxx

```
module math.vector:sum_of_squares;
import <span>;
import <algorithm>;
float sum_of_squares(std::span<float> x) {
  return std::transform_reduce(x);
}
```

math_vector.mxx

```
module math.vector;
float sqrt(float x) { /* ... */ }
```

Kinds of Translation Units

	Example	Extension	Artifact	Notes
Non-Modular Unit	`#include "…"` `…`	`.cpp`	`.o`	
Header Unit	`// Created by:` `import <…>;`	`.hpp`	`.cmi` `.o (optional)`	
Module Interface Unit	`export module …;` `…`	`.ixx`	`.cmi` `.o (optional)`	Exactly one per module.
Module Implementation Unit	`module …;` `…`	`.mxx`	`.o`	At most one per module.
Module Partition Interface Unit	`export module …:…;` `…`	`.ixx`	`.cmi` `.o (optional)`	
Module Partition Implementation Unit	`module …:…;` `…`	`.mxx`	`.o`	

Modules do not force a file layout on you.

Ecosystem Impact

math.hpp

```
#pragma once

int square(int a);
```

math.cpp

```
#include “math.hpp”

int square(int a) { return a * a; }
```

How are headers found?

- Not specified by the standard.
- In practice, all implementations assume a mapping between file names and #includes.
- The file is searched for in a set of include paths.

COMPILER EXPLORER

Add... More | Share Other Policies

C++ source #1

A Save/Load + Add new... CppInsights | C++

```cpp
#include <https://compile-time.re/dfa.hpp>

bool match(std::string_view subject) {
    return ctre::fast_match<"aloha|[a-z]*">(subject);
}
```

x86-64 gcc 9.1 (Editor #1, Compiler #1) C++

x86-64 gcc 9.1 | -std=c++2a -Os

11010 | .LX0: | lib.f: | .text | // | \s+ | Intel | Demangle

Libraries + Add new... Add tool...

```asm
match(std::basic_string_view<c
        add     rdi, rsi
.L3:
        cmp     rdi, rsi
        je      .L4
        movsx   eax, BYTE PTR
        sub     eax, 97
        cmp     eax, 25
        ja      .L5
        inc     rsi
        jmp     .L3
.L4:
        mov     al, 1
        ret
.L5:
        xor     eax, eax
        ret
```

Output (0/0) x86-64 gcc 9.1 - cached (1569389B)

bar.ixx

```
export module foo;

// …
```

How are modules found?

- Not specified by the standard.
- Unlike headers, modules are programmatically named.
- A file name <-> module name mapping is not straightforward.
    - Modules have to be precompiled.
    - Partitions span multiple files.

Compiled Module Interface (CMI) Configuration

- CMIs are built with a certain set of compiler options and global macro definitions (the CMI configuration).
    - Ex: -Wall, -O3, -DDEBUG
- The CMI may only be used when compiling with the same set of compiler options and global macro definitions.
- Module lookup isn't just a matter of mapping a module name to a module interface unit (MIU) + a CMI.
- It's mapping a module name + CMI configuration to a MIU + a CMI.

Strategies for Module Lookup

- Assume File Name == Module Name and Search
  - When importing “foo”, the compiler looks for “foo.ixx” and “foo.cmi” in a set of search directories.
  - If a “foo.cmi” with the wrong CMI configuration is found, an error is produced or a new CMI is built on the fly.

Strategies for Module Lookup

- Search.
  - When importing “foo”, the compiler looks for all “.ixx” files in a set of search directories.
  - When the compiler finds a match it looks for a corresponding “.cmi” file with the same prefix.
  - If the found CMI has the wrong CMI configuration is found, an error is produced or a new CMI is built on the fly.

Strategies for Module Lookup

- Explicitly Passed.
  - The user specifies all of the MIU and CMI files needed for compilation of a particular TU to the compiler.
  - If the specified CMI has the wrong CMI configuration is found, an error is produced.
  - This is one of the approaches supported by Clang today.

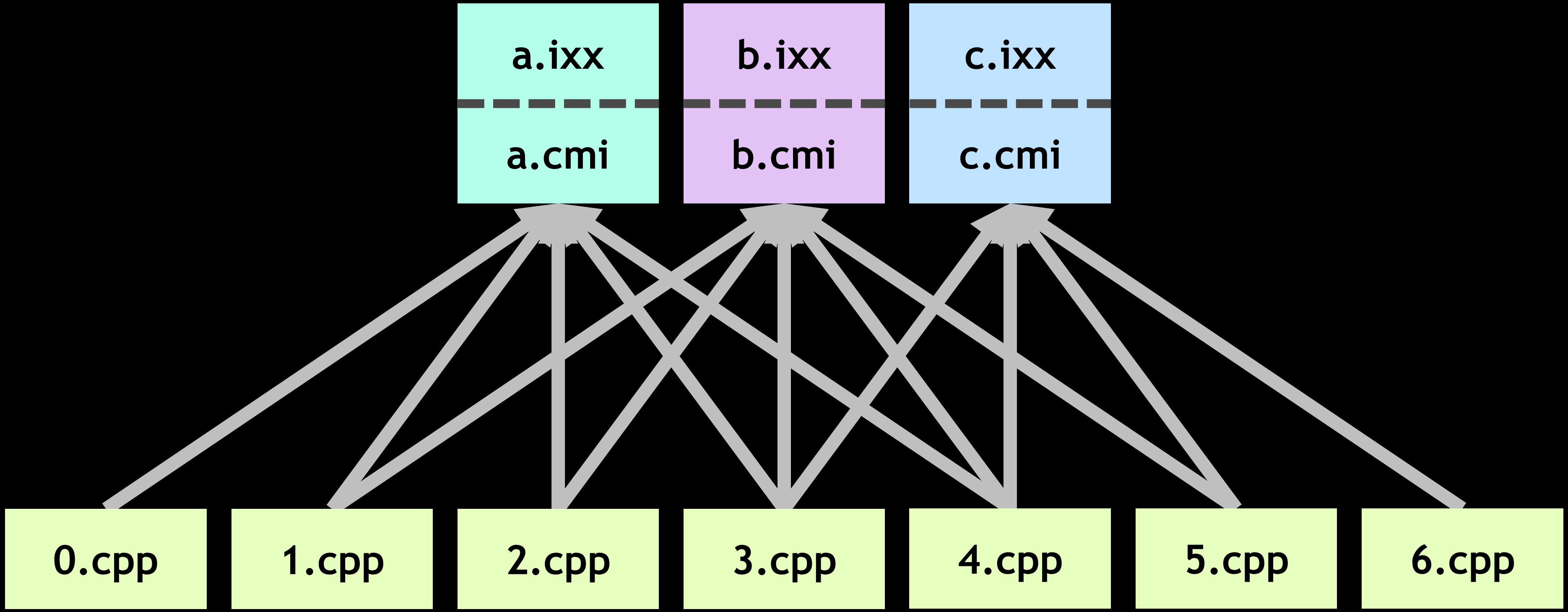
Strategies for Module Lookup

- Explicit Static Mapping.
  - The user specifies a file that describes a mapping from module name + CMI configuration to MIU + CMI.
  - If the specified CMI has the wrong CMI configuration is found, an error is produced.
  - This is one of the approaches supported by Clang today (for header units at least).

Strategies for Module Lookup

- Client/Server Mapper Daemon
  - The compiler communicates with a daemon to either request the location of an existing CMI or register a new CMI.
  - This is one of the approaches being explored by GCC today.

Implicit Precompilation is Problematic



If precompilation is implicit and builds are parallel, how do we decide who builds the CMIs?

Implicit Precompilation is Problematic

Assuming the existence of fast dependency scanners, you can implicitly discover what CMIs need to be built.

This comes with a cost, however; you'll need to do this scanning as an additional pass prior to compilation.

Explicit Precompilation is Ideal, But…

Implicit

```
exe: a.o b.o
        $(CC) a.o b.o lib.so -o exe

a.o: a.cpp
        $(CC) -c a.cpp -o a.o

b.o: b.cpp
        $(CC) -c b.cpp -o b.o
```

Explicit

```
a.i.o: a.cmi
        $(CC) -c a.cmi -o a.i.o

b.i.o: b.cmi
        $(CC) -c b.cmi -o b.i.o

c.i.o: c.cmi
        $(CC) -c c.cmi -o c.i.o

lib.i.o: lib.cmi
        $(CC) -c lib.cmi -o lib.i.o

a.cmi: a.ixx
        $(CC) --precompile a.ixx -o a.cmi

b.cmi: b.ixx
        $(CC) --precompile b.ixx -o b.cmi

c.cmi: c.ixx
        $(CC) --precompile c.ixx -o c.cmi

lib.cmi: lib.ixx
        $(CC) --precompile lib.ixx -o lib.cmi

a.m.o: a.mxx a.cmi b.cmi c.cmi
        $(CC) -c a.mxx -o a.m.o

b.m.o: b.mxx b.cmi c.cmi lib.cmi
        $(CC) -c b.mxx -o b.m.o

exe: a.m.o b.m.o a.i.o b.i.o c.i.o lib.i.o
        $(CC) a.o b.o a.i.o b.i.o c.i.o lib.i.o lib.so -o exe
```

Tools can no longer rely on simple lookup mechanism (include directories and header file names) to understand C++ projects.

Dependency scanning now requires a C++ parser, not just a C preprocessor.

Tools that want to understand C++ code will need to interface with compilers.

C++ Ecosystem Technical Report

Modules are coming:

- Modules will bring substantial algorithmic build throughput improvements.
- Modules offer true encapsulation and proper sandboxing which will eliminate many structural challenges with writing C++ at scale.
- Transitioning to modules will not be free.

Thanks:

Hana Dusíková

Richard Smith

Michael Spencer

Gašper Ažman