

# Practical C++ Modules

Boris Kolpackov

[build2.org](http://build2.org)

v1.3, September 2019

***CODE  
SYNTHESIS***

# What & Why

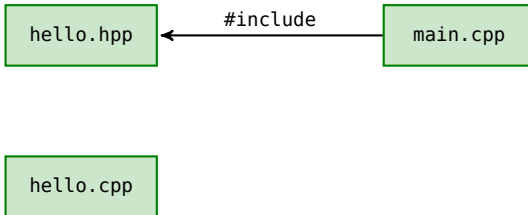
# Headers and Textual Inclusion

hello.hpp

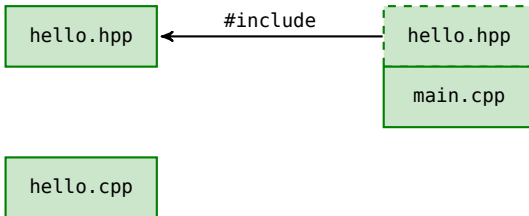
main.cpp

hello.cpp

# Headers and Textual Inclusion

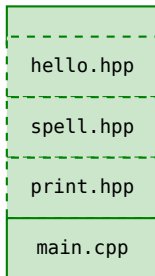


# Headers and Textual Inclusion



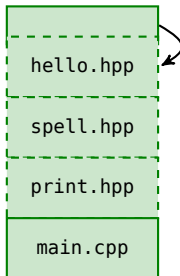
## Header Disadvantages

- Compilation speed
- Header/Source split
- Lack of isolation
  - Our code can change headers
  - Headers can change our code
  - Headers can change each other
  - Dependency on implementation
- ODR violations
  - *single definition*: non-inline
  - *identical definitions*: inline, types
- Order dependency and cycles
- Interfacing with C++



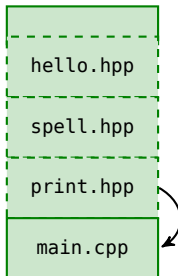
## Header Disadvantages

- Compilation speed
- Header/Source split
- Lack of isolation
  - Our code can change headers
  - Headers can change our code
  - Headers can change each other
  - Dependency on implementation
- ODR violations
  - *single definition*: non-inline
  - *identical definitions*: inline, types
- Order dependency and cycles
- Interfacing with C++



## Header Disadvantages

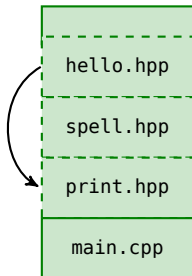
- Compilation speed
- Header/Source split
- Lack of isolation
  - Our code can change headers
  - Headers can change our code
  - Headers can change each other
  - Dependency on implementation
- ODR violations
  - *single definition*: non-inline
  - *identical definitions*: inline, types
- Order dependency and cycles
- Interfacing with C++





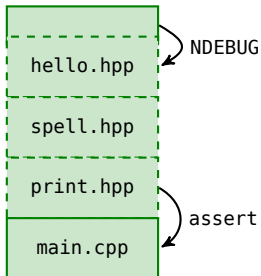
## Header Disadvantages

- Compilation speed
- Header/Source split
- Lack of isolation
  - Our code can change headers
  - Headers can change our code
  - Headers can change each other
  - Dependency on implementation
- ODR violations
  - *single definition*: non-inline
  - *identical definitions*: inline, types
- Order dependency and cycles
- Interfacing with C++



## Header Disadvantages

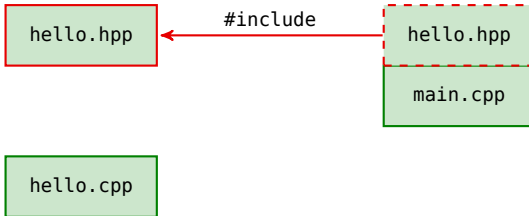
- Compilation speed
- Header/Source split
- Lack of isolation
  - Our code can change headers
  - Headers can change our code
  - Headers can change each other
  - Dependency on implementation
- ODR violations
  - *single definition*: non-inline
  - *identical definitions*: inline, types
- Order dependency and cycles
- Interfacing with C++



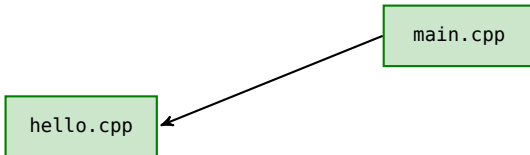
# Header Advantages

- Embarrassingly Parallel
- Familiar
- Flexible & Hackable
- Toolable (to a degree)

# From Headers to Modules

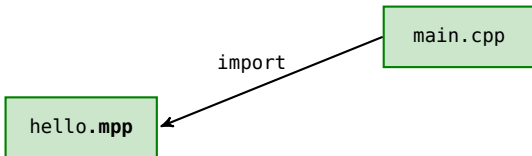


## From Headers to Modules



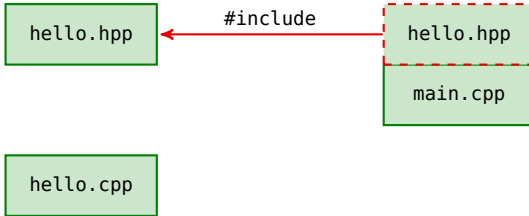
## Module Importation

## From Headers to Modules

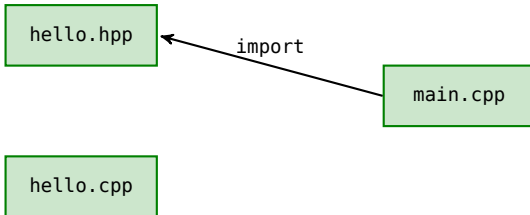


## Module Importation

# From Header Include to Import

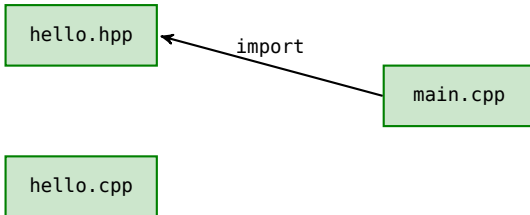


## From Header Include to Import



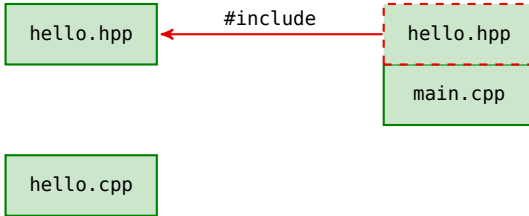


## From Header Include to Import

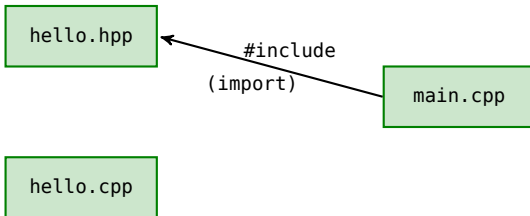


Header Importation

# From Header Include to Auto-Import



## From Header Include to Auto-Import

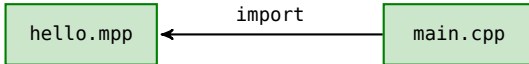


Include Translation

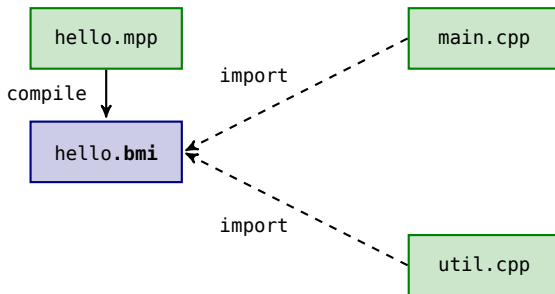
## Modularization Options

- Module importation
- Header importation
- Include translation (to Header importation)

# Modules Build Mechanics



# Modules Build Mechanics



## Binary Module Interface (BMI)

## Modularization Options

- Include translation (to Header importation)
- Header importation
- Module importation

## Include Translation

- No modification required on either side
- But header should be *importable*
- All C++20 std headers are importable...
- ... (except for <c\*> C wrapper headers)



## Include Translation

How does it actually work?

## Importable Headers

- Modular in the broader sense:
  - Does not rely on pre-definitions (macros, declarations)
  - Or post-undefinitions (macros)
- Example: header that requires pre-inclusion of another header
- Example: header that implements X-macro technique
- Internal linkage is Ok as long as not used outside header
- Example: Schwartz counter

## Include Translation: Problems Solved

- Compilation speed
- Header/Source split
- Lack of isolation
  - Our code can change headers
  - Headers can change our code
  - Headers can change each other
  - Dependency on implementation
- ODR violations
  - *single definition*: non-inline
  - *identical definitions*: inline, types
- Order dependency and cycles
- Interfacing with C++

## Include Translation: Problems Solved

- ~~Compilation speed~~
- Header/Source split
- Lack of isolation
  - Our code can change headers
  - Headers can change our code
  - Headers can change each other
  - Dependency on implementation
- ODR violations
  - *single definition*: non-inline
  - *identical definitions*: inline, types
- Order dependency and cycles
- Interfacing with C++

## Include Translation: Problems Solved

- ~~Compilation speed~~
- Header/Source split
- Lack of isolation
  - ~~Our code can change headers~~ (if translated)
  - Headers can change our code
  - ~~Headers can change each other~~ (if translated)
  - Dependency on implementation
- ODR violations
  - *single definition*: non-inline
  - *identical definitions*: inline, types
- Order dependency and cycles
- Interfacing with C++

## Include Translation: Problems Solved

- ~~Compilation speed~~
- Header/Source split
- Lack of isolation
  - ~~Our code can change headers~~ (if translated)
  - Headers can change our code
  - ~~Headers can change each other~~ (if translated)
  - Dependency on implementation
- ~~ODR violations~~ (if translated)
  - *single definition*: non-inline
  - ~~*identical definitions*: inline, types~~
- Order dependency and cycles
- Interfacing with C++

## Include Translation: Problems Solved

- ~~Compilation speed~~
- Header/Source split
- Lack of isolation
  - ~~Our code can change headers~~ (if translated)
  - Headers can change our code
  - ~~Headers can change each other~~ (if translated)
  - Dependency on implementation
- ~~ODR violations~~ (if translated)
  - *single definition*: non-inline
  - ~~*identical definitions*~~: inline, types
- ~~Order dependency~~ (if translated) and cycles
- Interfacing with C++

## Include Translation: Problems Solved

- ~~Compilation speed~~
- Header/Source split
- Lack of isolation
  - ~~Our code can change headers~~ (if translated)
  - Headers can change our code
  - ~~Headers can change each other~~ (if translated)
  - Dependency on implementation
- ~~ODR violations~~ (if translated)
  - *single definition*: non-inline
  - ~~*identical definitions*~~: inline, types
- ~~Order dependency~~ (if translated) and cycles
- ~~Interfacing with C++ and C!~~



# Header Importation

main.cpp

```
#include "hello.hpp"
```

```
int main ()  
{  
    // ...  
}
```

main.cpp

```
import "hello.hpp";
```

```
int main ()  
{  
    // ...  
}
```

# Header Importation

main.cpp

```
#include "hello.hpp"
```

```
int main ()  
{  
    // ...  
}
```

main.cpp

```
import "hello.hpp";
```

```
int main ()  
{  
    // ...  
}
```

## Header Importation

- Only consumer modifications required
- Header should be importable
- Remaining `#includes` are Ok...
- ...But not automatically translated

## Header Importation: Problems Solved

- Compilation speed
- Header/Source split
- Lack of isolation
  - Our code can change headers(if translated)
  - Headers can change our code
  - Headers can change each other (if translated)
  - Dependency on implementation
- ODR violations
  - *single definition*: non-inline
  - *identical definitions*: inline, types
- Order dependency (if translated) and cycles
- Interfacing with C++ and C!

# Header Importation: Problems Solved

- ~~Compilation speed~~
- Header/Source split
- Lack of isolation
  - ~~Our code can change headers (if translated)~~
  - Headers can change our code
  - ~~Headers can change each other (if translated)~~
  - Dependency on implementation
- ~~ODR violations~~
  - *single definition*: non-inline
  - *identical definitions*: inline, types
- ~~Order dependency (if translated)~~ and cycles
- ~~Interfacing with C++ and C!~~

# Module Importation

hello.hpp

```
#pragma once
#include <string>
namespace hello {
    void say (std::string);
}
```

hello.cpp

```
#include "hello.hpp"
#include <iostream>
namespace hello {
    void say (std::string) {
        ... }
}
```

main.cpp

```
#include "hello.hpp"
int main () {
    hello::say ("World");
}
```

hello.mpp

```
export module hello;
import <string>;
```

```
import <iostream>;
namespace hello {
    export void say (std::string) {
        ... }
}
```

main.cpp

```
import hello;
int main () {
    hello::say ("World");
}
```

# Module Importation

hello.hpp

```
#pragma once
#include <string>
namespace hello {
    void say (std::string);
}
```

hello.cpp

```
#include "hello.hpp"
#include <iostream>
namespace hello {
    void say (std::string) {
        ... }
}
```

main.cpp

```
#include "hello.hpp"
int main () {
    hello::say ("World");
}
```

hello.mpp

```
export module hello;
import <string>;
```

```
import <iostream>;
namespace hello {
    export void say (std::string) {
        ... }
}
```

main.cpp

```
import hello;
int main () {
    hello::say ("World");
}
```

# Module Importation

hello.hpp

```
#pragma once
#include <string>
namespace hello {
    void say (std::string);
}
```

hello.cpp

```
#include "hello.hpp"
#include <iostream>
namespace hello {
    void say (std::string) {
        ... }
}
```

main.cpp

```
#include "hello.hpp"
int main () {
    hello::say ("World");
}
```

hello.mpp

```
export module hello;
import <string>;
```

```
import <iostream>;
namespace hello {
    export void say (std::string) {
        ... }
}
```

main.cpp

```
import hello;
int main () {
    hello::say ("World");
}
```



# Module Importation

hello.hpp

```
#pragma once
#include <string>
namespace hello {
    void say (std::string);
}
```

hello.cpp

```
#include "hello.hpp"
#include <iostream>
namespace hello {
    void say (std::string) {
        ... }
}
```

main.cpp

```
#include "hello.hpp"
int main () {
    hello::say ("World");
}
```

hello.mpp

```
export module hello;
import <string>;
```

```
import <iostream>;
namespace hello {
    export void say (std::string) {
        ... }
}
```

main.cpp

```
import hello;
int main () {
    hello::say ("World");
}
```

# Module Importation

hello.hpp

```
#pragma once
#include <string>
namespace hello {
    void say (std::string);
}
```

hello.cpp

```
#include "hello.hpp"
#include <iostream>
namespace hello {
    void say (std::string) {
        ... }
}
```

main.cpp

```
#include "hello.hpp"
int main () {
    hello::say ("World");
}
```

hello.mpp

```
export module hello;
import <string>;
```

```
import <iostream>;
namespace hello {
    export void say (std::string) {
        ... }
}
```

main.cpp

```
import hello;
int main () {
    hello::say ("World");
}
```

# Module Importation

hello.hpp

```
#pragma once
#include <string>
namespace hello {
    void say (std::string);
}
```

hello.cpp

```
#include "hello.hpp"
#include <iostream>
namespace hello {
    void say (std::string) {
        ... }
}
```

main.cpp

```
#include "hello.hpp"
int main () {
    hello::say ("World");
}
```

hello.mpp

```
export module hello;
import <string>;
```

```
import <iostream>;
namespace hello {
    export void say (std::string) {
        ... }
}
```

main.cpp

```
import hello;
int main () {
    hello::say ("World");
}
```

# Module Importation

hello.hpp

```
#pragma once
#include <string>
namespace hello {
    void say (std::string);
}
```

hello.cpp

```
#include "hello.hpp"
#include <iostream>
namespace hello {
    void say (std::string) {
        ... }
}
```

main.cpp

```
#include "hello.hpp"
int main () {
    hello::say ("World");
}
```

hello.mpp

```
export module hello;
import <string>;
```

```
import <iostream>;
namespace hello {
    export void say (std::string) {
        ... }
}
```

main.cpp

```
import hello;
int main () {
    hello::say ("World");
}
```

# Module Importation

hello.hpp

```
#pragma once
#include <string>
namespace hello {
    void say (std::string);
}
```

hello.cpp

```
#include "hello.hpp"
#include <iostream>
namespace hello {
    void say (std::string) {
        ... }
}
```

main.cpp

```
#include "hello.hpp"
int main () {
    hello::say ("World");
}
```

hello.mpp

```
export module hello;
import <string>;
```

```
import <iostream>;
namespace hello {
    export void say (std::string) {
        ... }
}
```

main.cpp

```
import hello;
int main () {
    hello::say ("World");
}
```

## Module Importation: Problems Solved

- Compilation speed
- Header/Source split
- Lack of isolation
  - Our code can change modules
  - Modules can change our code
  - Modules can change each other
  - Dependency on implementation
- ODR violations
  - *single definition*: non-inline
  - *identical definitions*: inline, types
- Order dependency and cycles
- Interfacing with C++

## Module Importation: Problems Solved

- ~~Compilation speed~~
- Header/Source split
- Lack of isolation
  - Our code can change modules
  - Modules can change our code
  - Modules can change each other
  - Dependency on implementation
- ODR violations
  - *single definition*: non-inline
  - *identical definitions*: inline, types
- Order dependency and cycles
- Interfacing with C++

## Module Importation: Problems Solved

- ~~Compilation speed~~
- ~~Header/Source split~~
- Lack of isolation
  - Our code can change modules
  - Modules can change our code
  - Modules can change each other
  - Dependency on implementation
- ODR violations
  - *single definition*: non-inline
  - *identical definitions*: inline, types
- Order dependency and cycles
- Interfacing with C++



# Module Importation: Problems Solved

- ~~Compilation speed~~
- ~~Header/Source split~~
- ~~Lack of isolation~~
  - ~~Our code can change modules~~
  - ~~Modules can change our code~~
  - ~~Modules can change each other~~
  - ~~Dependency on implementation~~
- ODR violations
  - *single definition*: non-inline
  - *identical definitions*: inline, types
- Order dependency and cycles
- Interfacing with C++

# Module Importation: Problems Solved

- ~~Compilation speed~~
- ~~Header/Source split~~
- ~~Lack of isolation~~
  - ~~Our code can change modules~~
  - ~~Modules can change our code~~
  - ~~Modules can change each other~~
  - ~~Dependency on implementation~~
- ~~ODR violations~~
  - ~~*single definition*: non-inline~~
  - ~~*identical definitions*: inline, types~~
- Order dependency and cycles
- Interfacing with C++

# Module Importation: Problems Solved

- ~~Compilation speed~~
- ~~Header/Source split~~
- ~~Lack of isolation~~
  - ~~Our code can change modules~~
  - ~~Modules can change our code~~
  - ~~Modules can change each other~~
  - ~~Dependency on implementation~~
- ~~ODR violations~~
  - ~~*single definition*: non-inline~~
  - ~~*identical definitions*: inline, types~~
- ~~Order dependency and cycles~~
- ~~Interfacing with C++~~

# Module Importation: Problems Solved

- ~~Compilation speed~~
- ~~Header/Source split~~
- ~~Lack of isolation~~
  - ~~Our code can change modules~~
  - ~~Modules can change our code~~
  - ~~Modules can change each other~~
  - ~~Dependency on implementation~~
- ~~ODR violations~~
  - ~~*single definition*: non-inline~~
  - ~~*identical definitions*: inline, types~~
- ~~Order dependency and cycles~~
- ~~Interfacing with C++~~

# How

# Module Structure

hello.mpp

```
|  
.  
  
export module hello;  
  
import <string>;  
import <iostream>;  
  
export namespace hello {  
    void say (std::string) {  
        ...  
    }  
}
```

# Module Structure

hello.mpp

```
module;  
  
#include <cassert>  
  
export module hello;  
  
import <string>;  
import <iostream>;  
  
export namespace hello {  
    void say (std::string) {  
        ...  
    }  
}
```

# Module Structure

hello.mpp

**module;**

**#include** <cassert>

**export module** hello;

**import** <string>;

**import** <iostream>;

**export namespace** hello {  
    **void** say (std::string) {  
        ...  
    }  
}

**module;**

global module fragment  
(preprocessor directives only)

**export module** *name*;

module preamble  
(import declarations only)

module purview  
(exported declarations, etc)



# Module Structure

hello.mpp

```
module;
```

```
#include <cassert>
```

```
export module hello;
```

```
import <string>;
```

```
import <iostream>;
```

```
export namespace hello {  
    void say (std::string) {  
        ...  
    }  
}
```

```
module;
```

global module fragment  
(preprocessor directives only)

```
export module name;
```

module preamble  
(import declarations only)

module purview  
(exported declarations, etc)

# Module Structure

hello.mpp

```
module;
```

```
#include <cassert>
```

```
export module hello;
```

```
import <string>;
```

```
import <iostream>;
```

```
export namespace hello {  
    void say (std::string) {  
        ...  
    }  
}
```

**module;**

global module fragment  
(preprocessor directives only)

**export module *name*;**

module preamble  
(import declarations only)

module purview  
(exported declarations, etc)

# Module Structure

hello.mpp

```
module;
```

```
#include <cassert>
```

```
export module hello;
```

```
import <string>;
```

```
import <iostream>;
```

```
export namespace hello {  
    void say (std::string) {  
        ...  
    }  
}
```

**module;**

global module fragment  
(preprocessor directives only)

**export module *name*;**

module preamble  
(import declarations only)

module purview  
(exported declarations, etc)

## Module Structure

What's wrong with this?

hello.mpp

```
export module hello;
```

```
#include <string>
```

```
...
```

## Module Structure

What's wrong with this?

hello.mpp

```
export module hello;
```

```
#include <string>
```

```
...
```

Including headers in module purview is a bad idea

(Unless modularizing that header)

# Module Interface and Implementation

hello.mpp

```
module;  
  
#include <cassert>  
  
export module hello;  
  
import <string>;  
import <iostream>;  
  
export namespace hello {  
    void say (std::string) {  
        ... }  
}
```

# Module Interface and Implementation

hello.mpp

```
module;
```

```
#include <cassert>
```

```
export module hello;
```

```
import <string>;
```

```
import <iostream>;
```

```
export namespace hello {  
    void say (std::string) {  
        ... }  
}
```

# Module Interface and Implementation

hello.mpp

```
module;  
  
#include <cassert>  
  
export module hello;  
  
import <string>;  
import <iostream>;  
  
export namespace hello {  
    void say (std::string) {  
        ... }  
}
```

hello.mpp

```
export module hello;  
import <string>;  
  
export namespace hello {  
    void say (std::string);  
}
```

hello.cpp

```
module;  
#include <cassert>  
  
module hello;  
import <iostream>;  
  
namespace hello {  
    void say (std::string) {  
        ... }  
}
```



# Module Interface and Implementation

hello.mpp

```
module;  
  
#include <cassert>  
  
export module hello;  
  
import <string>;  
import <iostream>;  
  
export namespace hello {  
    void say (std::string) {  
        ... }  
}
```

hello.mpp

```
export module hello;  
import <string>;  
  
export namespace hello {  
    void say (std::string);  
}
```

hello.cpp

```
module;  
#include <cassert>  
  
module hello;  
import <iostream>;  
  
namespace hello {  
    void say (std::string) {  
        ... }  
}
```

# Module Interface and Implementation

hello.mpp

```
module;  
  
#include <cassert>  
  
export module hello;  
  
import <string>;  
import <iostream>;  
  
export namespace hello {  
    void say (std::string) {  
        ... }  
}
```

hello.mpp

```
export module hello;  
import <string>;  
  
export namespace hello {  
    void say (std::string);  
}
```

hello.cpp

```
module;  
#include <cassert>  
  
module hello;  
import <iostream>;  
  
namespace hello {  
    void say (std::string) {  
        ... }  
}
```

# Module Interface and Implementation

- Interface can (still) define non-inline functions/variables
- We can have multiple implementation units...
- ...But only one (primary) interface unit
- Interface partitions: split interface
- Implementation partitions: “module-private interface”

## To Split or Not to Split

### Pros:

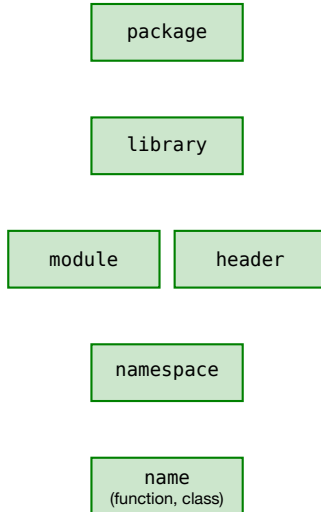
- DRY
- *Module interface-only* libraries

### Cons:

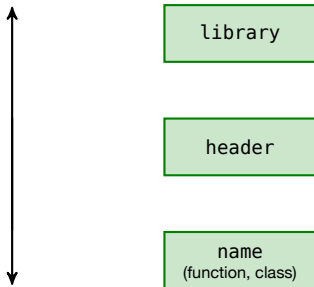
- Unnecessary recompilation
- Reduced interface readability
- Extra dependencies (implementation imports)
- Reduced interface compilation speed

Judgment Call

# Physical Design Mechanisms



# Module Granularity



Cost of importing modules is negligible

# Module Granularity

## Too big:

- Unnecessary recompilations
- Hard to navigate

## Too small:

- Tedious to import
- Also hard to navigate

## Module Granularity

Combine related and commonly-used entities  
(generally good design)



## Module Granularity

Combine related and commonly-used entities  
(generally good design)

Use re-export to create “aggregate modules”

hello.mpp

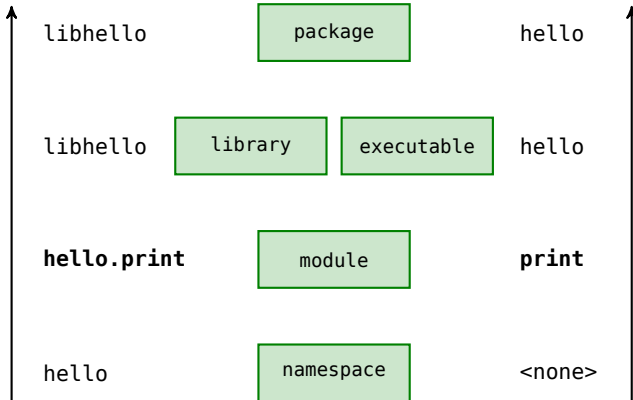
```
export module hello;  
  
export import hello.format;  
export import hello.print;
```

## Module Name

```
export module hello;  
  
export module hello.format;  
export module hello.print;  
  
export module hello.print.iostream;
```

- Sequence of dot-separated identifiers
- On a separate “name plane”
- Do not collide with namespace/type/function names
- No specified hierarchical semantics (yet)

# Naming Modules



## Naming Modules

- Start with the library/project top namespace (if any)
- Finish with a name describing the module's functionality
- If for a single/primary entity (class, etc), use its name
- Provide “aggregate modules” for hierarchy

## Module Naming Examples

- Library name: libbutl
- Library namespace: butl
- Library modules:

butl.base64  
butl.char\_scanner  
butl.const\_ptr  
butl.diagnostics  
butl.fdstream  
butl.filesystem  
butl.manifest\_parser  
butl.manifest\_serializer  
butl.multi\_index  
butl.openssl  
butl.pager

butl.path  
butl.path\_io  
butl.path\_map  
butl.process  
butl.sha256  
butl.small\_vector  
butl.string\_parser  
butl.string\_table  
butl.target\_triplet  
butl.timestamp  
butl.vector\_view

## Naming Module Files

- No mapping between module names and file names
- But clearly makes sense for them to be related

## Naming Module Files: Extensions

Source/Header	Module Interface	Module Implementation
.cpp/ .hpp/ .h	.mpp	.cpp
.cxx/ .hxx/ .h	.mxx	.cxx
.c++/ .h++/ .h	.m++	.c++
.cc/ .hh/ .h	switch	switch
.C/ .H/ .h	switch	switch

**Have a separate extension for interfaces**

## Naming Module Files: Base

```
export module hello;  
  
export module hello.format;  
export module hello.print;  
  
export module hello.print.iostream;
```

```
libhello/  
├─ hello.mpp  
├─ hello-format.mpp  
├─ hello-print.mpp  
└─ hello-print-iostream.mpp
```

```
libhello/  
├─ hello.mpp  
├─ format.mpp  
├─ print.mpp  
└─ print-iostream.mpp
```



## Naming Module Files

Embed sufficient amount of module name “tail”  
into file names to unambiguously distinguish modules  
within a library/project

# Distributing Modules

## What's in a BMI?

- Compiler specific, can be anything between
- ...stream of preprocessed tokens
- ...dump of an AST
- ...something close to object code
- Sensitive to most compiler options (even warning)

## What to Install/Distribute?

**BMIs are not a *distribution mechanism***

- BMIs should not be installed/distributed (maybe cached)
- Install/distribute module interfaces instead
- BTW, another reason to split interface/implementation

# When

# Modularization Options

How far do you want to go?

- Include translation
- Header importation
- Module importation

## Types of C++ Projects

Single-platform  
End-product

Single-platform  
Reusable

Cross-platform  
End-product

Cross-platform  
Reusable

# Single-platform End-product

## Single-platform End-product:

- Include translation (can do better)
- Header importation
- Module importation

Single-platform  
Reusable

Cross-platform  
End-product

Cross-platform  
Reusable

# Single-platform Reusable

Single-platform  
End-product

## Single-platform Reusable:

- Include translation (can do better)
- Header importation
- Module importation

Cross-platform  
End-product

Cross-platform  
Reusable



# Cross-platform End-product

Single-platform  
End-product

Single-platform  
Reusable

Cross-platform End-product:

- Include translation
- Header importation (complexity)
- Module importation (portability)

Cross-platform  
Reusable

# Cross-platform Reusable

Single-platform End-product	Single-platform Reusable
Cross-platform End-product	<b>Cross-platform Reusable:</b> <ul style="list-style-type: none"><li>• Include translation</li><li>• Header importation (portability)</li><li>• Module importation (portability)</li></ul>

## Cross-platform Reusable

Dual header/module interface?

Just say No!

See CppCon 2017 “Building C++ Modules” for details

## When: the Standard

- Modules in C++20
- Importable standard library headers in C++20
- Modular standard library in C++23
- What about system headers?

## When: the Compilers

- Still incomplete but improving rapidly
- There are bugs, especially in header importation
- Support for build systems is still lacking

## When: the Build Systems

	Modules	Headers	Include Translation
build2	Yes	Yes (GCC)	Yes (GCC)
CMake	WIP		
Meson		“wait and see”	
autotools		“unlikely?”	
Bazel		?	
Buck		?	
IDEs		?	

## Questions?

### Areas Not Covered:

- Visibility vs Reachability
- Private module fragment (`module :private;`)
- Interface and Implementation partitions
- Exported using declarations (`export using X;`)
- What about `main()`?
- What about module versioning? Inline modules anyone?