Building C++ Modules

Boris Kolpackov

Code Synthesis

v1.3, September 2017

Practical C++ Modules

"*acknowledged and acknowledgeable by the C++ standard*"

A Module System for C++ (P0142R0)

Why Modules?

- *Isolation* from macros and symbols
- *Physical design mechanism*
- Step towards not needing the preprocessor
- Reliable distributed compilation

Speed!!!

Photo credit: bhmpics

What is a Module?

Three Perspectives

- module consumer
- module producer
- build system

What is a Module?

Language-level mechanism:

```
import hello.core;
```

Not preprocessor-level:

```
#import hello.core
```

Modules from Consumer Perspective

- Collection of *external names*
- Called *module interface*
- Become *visible* once imported

```
import hello.core;
```

Modules from Consumer Perspective

What does *visible* mean?

An import-declaration makes exported declarations [...]
visible to name lookup in the current translation unit,
in the same namespaces and contexts [...].
Note: The entities are not redeclared [...]

Modules and Namespaces

Modules and Namespace are Orthogonal

- Module can export names from any namespace(s)
- Module name and namespace name need not be related
- `import` does not imply `using`-directive

Modules and Namespaces

Modules and Namespace are Orthogonal

- Module can export names from any namespace(s)
- Module name and namespace name need not be related
- `import` does not imply `using`-directive

```
import hello.core;          // Exports hello::f().
f ();                       // Error.
hello::f ();                // Ok.
using namespace hello;
f ();                       // Ok.
```

Module and Libraries

Modules provide *Names* not *Symbols*

- Satisfy symbols the usual way: link object file or library
- Make perfect sense in programs, not only libraries
- Library may have private/implementation modules

Modules from Producer Perspective

- *module interface unit*
- *module implementation unit*
- *non-module translation unit*

Modules from Producer Perspective

- Collection of module translation units
- Exactly one interface unit
- Zero or more implementation units
- Interface may define non-inline functions/variables

Modules Interface Unit

Contains *exporting module declaration*

```
export module hello.core;
```

Modules Implementation Unit

Contains *non-exporting module declaration*

```
module hello.core;
```

Interface File Extensions

Vendor suggested extensions:

Clang	`.cppm`
GCC	`.?`
VC	`.ixx`

Interface File Extensions

Vendor suggested extensions:

Clang	`.cppm`
GCC	`.?`
VC	`.ixx`

My recommendation:

`.hpp/.cpp`	`.mpp`
`.hxx/.cxx`	`.mxx`
Other?	Switch!

Module Purview

Module declaration starts *module purview*

Name declared in purview *belongs* to module

```
#include <string>                          // Not in purview.

export module hello.core;                  // Start of purview.

void say_hello (const std::string&);       // In purview.
```

Module Export

Name belonging to module is *invisible* unless *exported*

Name can only be exported in module interface unit

Export Specifier

```
export module hello.core;

export enum class volume {quiet, normal, loud};

export void say_hello (const char*, volume);
```

Exported Group

```
export module hello.core;

export
{
  enum class volume {quiet, normal, loud};

  void say_hello (const char*, volume);
}
```

Exported Namespace

```
export module hello.core;

export namespace hello
{
  enum class volume {quiet, normal, loud};

  void say (const char*, volume);
}

namespace hello
{
  void impl (const char*, volume); // Not exported.
}
```

Module Ownership Model

- For exported names symbols are unchanged
- Non-exported names have *module linkage*:
- ...can only be resolved from module purview
- ...no clash with identical names in other modules
- (implemented by decorating with module name)

Module Ownership Model

Library built as modules can be used via headers

and even the other way around

Modules and Preprocessor

Modules do not export macros, only C++ names

Macros do not affect interfaces being imported

Import order is insignificant

Import Visibility

```
// hello.extra interface
//
export module hello.extra;

import hello.core;    // Exports say_hello().


// hello.extra implementation
//
module hello.extra;

say_hello (”World”);  // Ok.
```

Import Visibility

```
// hello.extra interface
//
export module hello.extra;

import hello.core;     // Exports say_hello().


// hello.extra implementation
//
module hello.extra;

say_hello (”World”);  // Ok.
```

Import Visibility

```
// hello.extra interface
//
export module hello.extra;

import hello.core;    // Exports say_hello().


// hello.extra implementation
//
module hello.extra;

say_hello (”World”);  // Ok.
```

Import Visibility

```
// hello.extra interface
//
export module hello.extra;

import hello.core;    // Exports say_hello().


// hello.extra consumer
//
import hello.extra;

say_hello (”World”);  // Error, hello.core import.
```

Import Visibility

```
// hello.extra interface
//
export module hello.extra;

import hello.core;    // Exports say_hello().


// hello.extra consumer
//
import hello.extra;

say_hello (”World”);  // Error, hello.core import.
```

Re-Export

```
// hello.extra interface
//
export module hello.extra;

export import hello.core;


// hello.extra consumer
//
import hello.extra;

say_hello (”World”);  // Ok.
```

Re-Export and Submodules

Re-export is the mechanism for assembling bigger modules out of submodules

```
export module hello;

export
{
  import hello.core;
  import hello.basic;
  import hello.extra;
}
```

Modules from Build System Perspective

"*The compiler should not become a build system.*"

Richard Smith

- *Binary Module Interface (BMI)*
- Produced by compiling module interface unit
- Required when compiling importing translation units...
- ...as well as module's implementation units

Hello Module

```
// hello.mxx
export module hello;
export void say_hello (const char* name);

// hello.cxx
#include <iostream>

module hello;

void say_hello (const char* n)
{
  std::cout << ”Hello, ” << n << ’!’ << std::endl;
}

// driver.cxx
import hello;

int main () { say_hello (”Modules”); }
```

Hello Module

```
// hello.mxx
export module hello;
export void say_hello (const char* name);

// hello.cxx
#include <iostream>

module hello;

void say_hello (const char* n)
{
  std::cout << "Hello, " << n << '!' << std::endl;
}

// driver.cxx
import hello;

int main () { say_hello ("Modules"); }
```

Hello Module

```
// hello.mxx
export module hello;
export void say_hello (const char* name);
```

```
// hello.cxx
#include <iostream>

module hello;

void say_hello (const char* n)
{
  std::cout << "Hello, " << n << '!' << std::endl;
}
```

```
// driver.cxx
import hello;

int main () { say_hello ("Modules"); }
```

Hello Module

```
// hello.mxx
export module hello;
export void say_hello (const char* name);

// hello.cxx
#include <iostream>

module hello;

void say_hello (const char* n)
{
  std::cout << ”Hello, ” << n << ’!’ << std::endl;
}

// driver.cxx
import hello;

int main () { say_hello (”Modules”); }
```

Compiling Modules

```
cl /D_CRT_SECURE_NO_WARNINGS -IC:\tmp\build\libodb-sql
ite-2.5.0-b.6.1503567680.1432c5607115e465 -IC:\tmp\bui
ld\libodb-sqlite-2.5.0-b.6.1503567680.1432c5607115e465
-DLIBODB_SQLITE_BUILD2 -DLIBODB_SQLITE_SHARED_BUILD -I
C:\tmp\build\libodb-2.5.0-b.6.1503567043.6b15416ac28ad
a02-IC:\tmp\build\libodb-2.5.0-b.6.1503567043.6b15416a
c28ada02 -DLIBODB_BUILD2 -DLIBODB_SHARED -IC:\tmp\buil
d\libsqlite3-3.18.3-a.0.1503562393.3c9bf2b8ce40e258\li
bsqlite3 -DSQLITE_API=__declspec(dllimport) /W3 /WX /w
d4251/wd4275 /nologo /EHsc/MD /Fo: libodb-sqlite-2.5.0
-b.6.1503567680.1432c5607115e465\odb\sqlite\database.d
ll.obj /c /TP C:\tmp\build\libodb-sqlite-2.5.0-b.6.150
3567680.1432c5607115e465\odb\sqlite\database.cxx
```

Compiling with GCC

```
$ ls -1
hello.mxx
hello.cxx
driver.cxx

$ g++ -std=c++1z -fmodules -x c++ -o hello.nms.o \
-fmodule-output=hello.nms -c hello.mxx

$ g++ -std=c++1z -fmodules -x c++ -o hello.o     \
-fmodule-file=hello=hello.nms -c hello.cxx

$ g++ -std=c++1z -fmodules -x c++ -o driver.o    \
-fmodule-file=hello=hello.nms -c driver.cxx

$ g++ -o hello hello.nms.o driver.o hello.o
```

Compiling with GCC

```
$ ls -1
hello.mxx
hello.cxx
driver.cxx

$ g++ -std=c++1z -fmodules -x c++ -o hello.nms.o \
-fmodule-output=hello.nms -c hello.mxx

$ g++ -std=c++1z -fmodules -x c++ -o hello.o     \
-fmodule-file=hello=hello.nms -c hello.cxx

$ g++ -std=c++1z -fmodules -x c++ -o driver.o    \
-fmodule-file=hello=hello.nms -c driver.cxx

$ g++ -o hello hello.nms.o driver.o hello.o
```

Compiling with GCC

```
$ ls -1
hello.mxx
hello.cxx
driver.cxx

$ g++ -std=c++1z -fmodules -x c++ -o hello.nms.o \
-fmodule-output=hello.nms -c hello.mxx

$ g++ -std=c++1z -fmodules -x c++ -o hello.o     \
-fmodule-file=hello=hello.nms -c hello.cxx

$ g++ -std=c++1z -fmodules -x c++ -o driver.o    \
-fmodule-file=hello=hello.nms -c driver.cxx

$ g++ -o hello hello.nms.o driver.o hello.o
```

Compiling with GCC

```
$ ls -1
hello.mxx
hello.cxx
driver.cxx

$ g++ -std=c++1z -fmodules -x c++ -o hello.nms.o \
-fmodule-output=hello.nms -c hello.mxx

$ g++ -std=c++1z -fmodules -x c++ -o hello.o     \
-fmodule-file=hello=hello.nms -c hello.cxx

$ g++ -std=c++1z -fmodules -x c++ -o driver.o    \
-fmodule-file=hello=hello.nms -c driver.cxx

$ g++ -o hello hello.nms.o driver.o hello.o
```

Compiling with GCC

```
$ ls -1
hello.mxx
hello.cxx
driver.cxx

$ g++ -std=c++1z -fmodules -x c++ -o hello.nms.o \
-fmodule-output=hello.nms -c hello.mxx

$ g++ -std=c++1z -fmodules -x c++ -o hello.o     \
-fmodule-file=hello=hello.nms -c hello.cxx

$ g++ -std=c++1z -fmodules -x c++ -o driver.o    \
-fmodule-file=hello=hello.nms -c driver.cxx

$ g++ -o hello hello.nms.o driver.o hello.o
```

Compiling with GCC

```
$ ls -1
hello.mxx
hello.cxx
driver.cxx

$ g++ -std=c++1z -fmodules -x c++ -o hello.nms.o \
-fmodule-output=hello.nms -c hello.mxx

$ g++ -std=c++1z -fmodules -x c++ -o hello.o     \
-fmodule-file=hello=hello.nms -c hello.cxx

$ g++ -std=c++1z -fmodules -x c++ -o driver.o    \
-fmodule-file=hello=hello.nms -c driver.cxx

$ g++ -o hello hello.nms.o driver.o hello.o
```

Compiling with Clang

```
$ clang++ -std=c++2a -fmodules-ts --precompile       \
-x c++-module [...] -o hello.pcm hello.mxx

$ clang++ -std=c++2a -fmodules-ts -o hello.pcm.o     \
-c hello.pcm

$ clang++ -std=c++2a -fmodules-ts -x c++ -o hello.o  \
-fmodule-file=hello.pcm -c hello.cxx

$ clang++ -std=c++2a -fmodules-ts -x c++ -o driver.o \
-fmodule-file=hello=hello.pcm -c driver.cxx

$ clang++ -o hello hello.pcm.o driver.o hello.o

[...] = -Xclang -fmodules-embed-all-files \
        -Xclang -fmodules-codegen         \
        -Xclang -fmodules-debuginfo
```

Compiling with Clang

```
$ clang++ -std=c++2a -fmodules-ts --precompile        \
-x c++-module [...] -o hello.pcm hello.mxx

$ clang++ -std=c++2a -fmodules-ts -o hello.pcm.o      \
-c hello.pcm

$ clang++ -std=c++2a -fmodules-ts -x c++ -o hello.o   \
-fmodule-file=hello.pcm -c hello.cxx

$ clang++ -std=c++2a -fmodules-ts -x c++ -o driver.o \
-fmodule-file=hello=hello.pcm -c driver.cxx

$ clang++ -o hello hello.pcm.o driver.o hello.o

[...] = -Xclang -fmodules-embed-all-files \
        -Xclang -fmodules-codegen         \
        -Xclang -fmodules-debuginfo
```

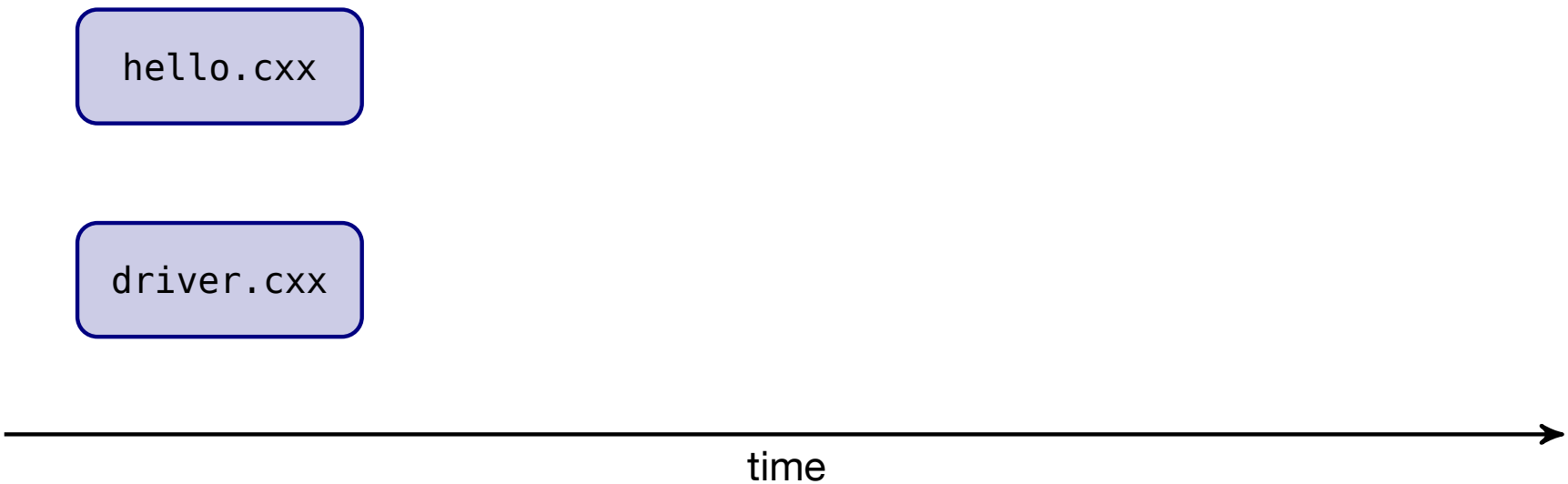
Compiling with VC

```
> cl.exe /std:c++latest /experimental:module /TP /EHsc ^
/MD /module:interface /Fo: hello.ifc.obj                ^
/module:output hello.ifc /c hello.mxx

> cl.exe /std:c++latest /experimental:module /TP /EHsc ^
/MD /module:reference hello.ifc /Fo: hello.obj          ^
/c hello.cxx

> cl.exe /std:c++latest /experimental:module /TP /EHsc ^
/MD /module:reference hello.ifc /Fo: driver.obj         ^
/c driver.cxx

> link.exe /OUT:hello.exe hello.ifc.obj driver.obj      ^
hello.obj
```

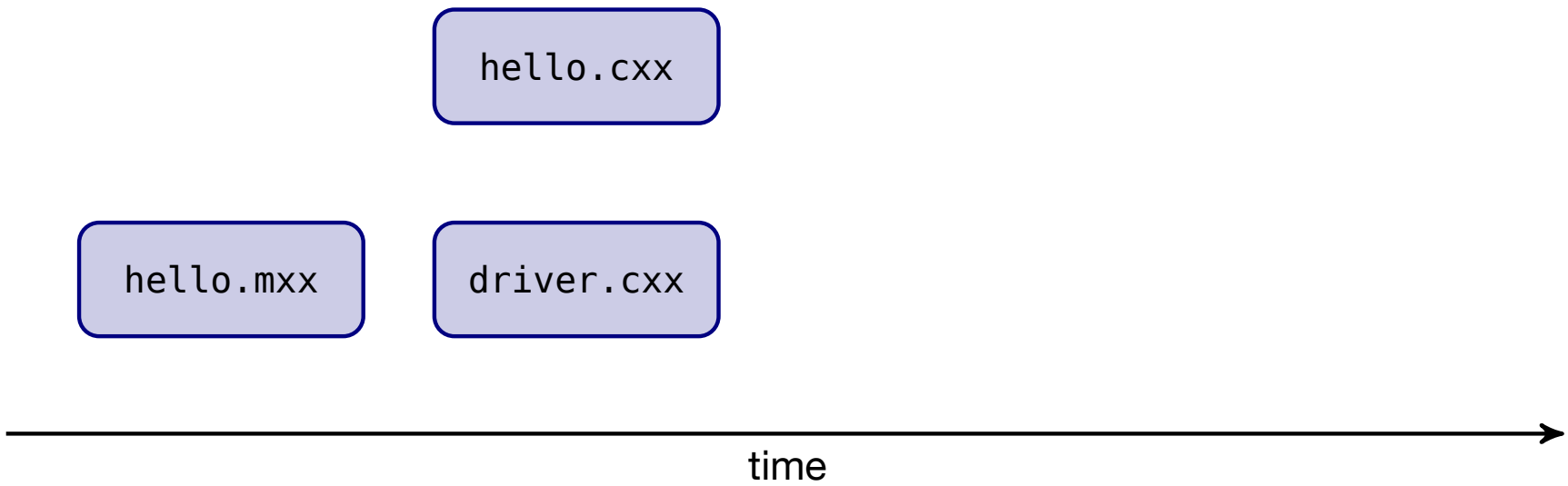
Modules from Build System Perspective

- Figure out the order of compilation
- Make sure every compilation can find BMIs it needs

Compilation of Header Project



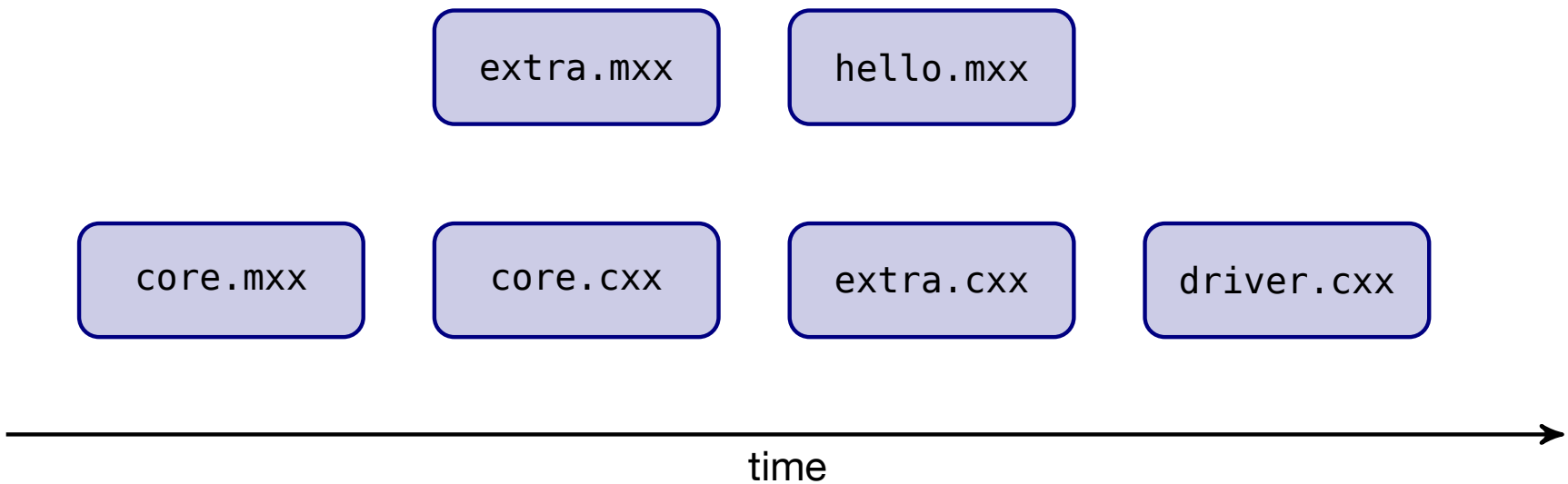
Compilation of Module Project



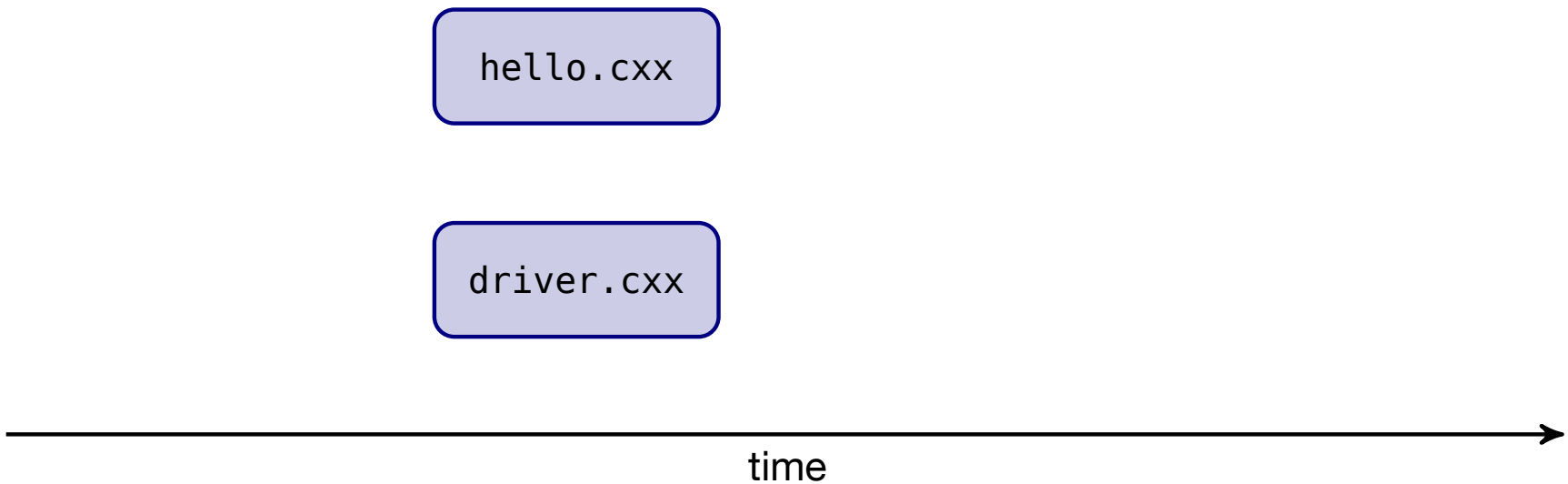
More Complex Module Project

```
$ ls -1
core.mxx
extra.mxx   # imports hello.core
hello.mxx   # re-exports hello.core and hello.extra

core.cxx
extra.cxx

driver.cxx  # imports hello
```

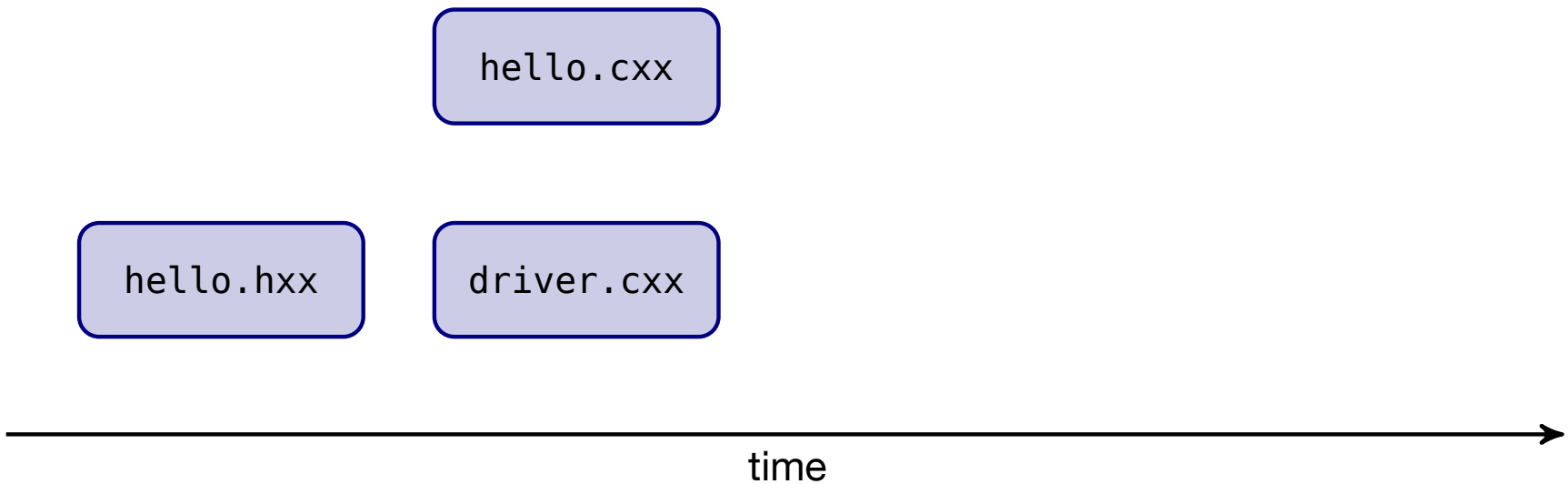
Compilation of Complex Module Project



Compilation of Generated Header Project



Compilation of Generated Header Project

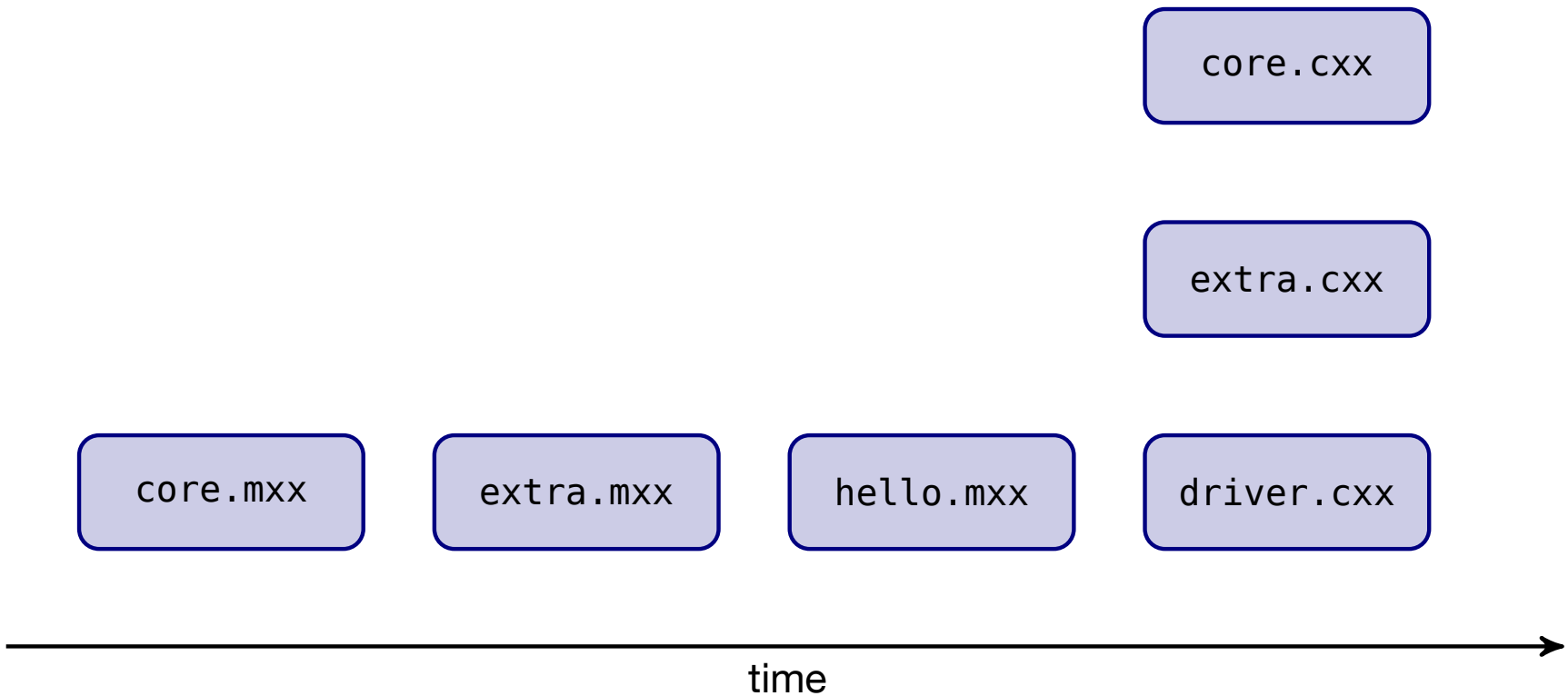


Modules Support via Ad Hoc Pre-Build Step

Does not work well

- Binary module interfaces depend on each other
- Manual module dependency specification does not scale

Compilation via Ad Hoc Pre-Build Step



What's in a BMI?

- Compiler specific, can be anything between
- ...stream of preprocessed tokens and
- ...something close to object code
- Sensitive to compiler options
- Build system may have to *side-build*

What to Install/Distribute?

Not a *distribution mechanism*

- BMIs should not be installed/distributed
- Install/distribute module interface units instead
- Build system has to *side-build*

Standard Library Modules (P0581R0)

Note: Very Experimental

`std.fundamental`	`<cstd*>` `<utility>` `<type_traits>` ...
`std.core`	`<string>` `<functional>` containers ...
`std.io`	`<iostream>` `<locale>` ...
`std.threading`	`<mutex>` `<thread>` ...

Module Design Guidelines

"*brave new module world*"

A Module System for C++ (P0142R0)

- Explicit exportation
- Module purview
- Module building

Module Granularity

- Cost of importing modules is negligible
- Mega-Modules cause unnecessary recompilation
- Mini-Modules are tedious to import

Module Granularity

Combine related and commonly-used entities

(generally good design)

- `xml.parser`
- `xml.serializer`
- `xml` (aggregate module)

Module Partitioning into Units

Opportunity to get rid of header/source divide?

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

Module Partitioning into Units

Have a separate module implementation unit

(except for simple/inline/template implementations)

Module-Only Libraries

The Holy Grail?

```
export module hello;

import std.core;
import std.io;

using namespace std;

export void say_hello (const string& n)
{
  cout << ”Look, ” << n << ”, I am not inline!” << endl;
}
```

Module-Only Libraries

Or a better Foot Gun?

- ODR violations
- Incompatible versions
- Where are the tests?

First Real Module

- Where to put `#include` directives?
- Where to put `import` declarations?
- In what order?

First Real Module

```
// Old rules.

[export] module hello; // Start or module purview.

// New rules.
```

First Real Module

What's wrong with this?

```
export module hello;

#include <string>

...
```

Where to Include?

Including headers in module purview is bad idea

(except for certain special headers)

First Real Module

```
#include <string>

export module hello;

#include <libhello/export.hxx>

export namespace hello
{
  ...
}

#include <libhello/hello.ixx>
```

Where to Import?

What about `import`?

- Merely makes names visible
- For implementation units exact location does not matter
- For interface units only imports in module purview:
- ... are visible in implementation units
- ... can be re-exported

Where to Import?

In interface units import in module purview

(unless a good reason not too)

Do likewise in implementation for consistency

First Real Module

```
#include <cassert>

export module hello;

import std.core;

#include <libhello/export.hxx>

export namespace hello
{
  ...
}

#include <libhello/hello.ixx>
```

Module Interface Template

```
<header includes>

export module <name>;        // Start of module purview.

<module imports>

<special header includes>  // Config, export, etc.

export
{
  <module interface>
}

<inline/template includes>
```

Module Interface Template

```
<header includes>

export module <name>;        // Start of module purview.

<module imports>

<special header includes>  // Config, export, etc.

export
{
  <module interface>
}

<inline/template includes>
```

Module Interface Template

```
<header includes>

export module <name>;       // Start of module purview.

<module imports>

<special header includes>  // Config, export, etc.

export
{
  <module interface>
}

<inline/template includes>
```

Module Interface Template

```
<header includes>

export module <name>;        // Start of module purview.

<module imports>

<special header includes>  // Config, export, etc.

export
{
  <module interface>
}

<inline/template includes>
```

Module Interface Template

```
<header includes>

export module <name>;        // Start of module purview.

<module imports>

<special header includes>  // Config, export, etc.

export
{
  <module interface>
}

<inline/template includes>
```

Module Interface Template

```
<header includes>

export module <name>;        // Start of module purview.

<module imports>

<special header includes>  // Config, export, etc.

export
{
  <module interface>
}

<inline/template includes>
```

Module Implementation Template

```
<header includes>

module <name>;                  // Start of module purview.

<extra module imports>  // Only additional to interface.

<module implementation>
```

Module Implementation Template

```
<header includes>

module <name>;                  // Start of module purview.

<extra module imports>  // Only additional to interface.

<module implementation>
```

Module Implementation Template

```
<header includes>

module <name>;                  // Start of module purview.

<extra module imports>  // Only additional to interface.

<module implementation>
```

Module Implementation Template

```
<header includes>

module <name>;                  // Start of module purview.

<extra module imports>  // Only additional to interface.

<module implementation>
```

Module Naming

- Module names in a separate “name plane”
- Do not collide with namespace/type/function names
- No prescribed hierarchical semantics
- Customary for `hello.core` to be a submodule of `hello`

Module Naming

- Start with the library/project namespace
- Finish with a name describing the module's functionality
- If dedicated to a single/primary entity, use its name

Module Naming Examples

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

```
butl.base64                  butl.path
butl.char_scanner            butl.path_io
butl.const_ptr               butl.path_map
butl.diagnostics             butl.process
butl.fdstream                butl.sha256
butl.filesystem              butl.small_vector
butl.manifest_parser         butl.string_parser
butl.manifest_serializer     butl.string_table
butl.multi_index             butl.target_triplet
butl.openssl                 butl.timestamp
butl.pager                   butl.vector_view
```

When to Re-Export?

```
export module hello;

export import std.core; // Good idea?

export void say_hello (const std::string&);
```

When to Re-Export?

Let’s talk about this another day ;-)

```
export module hello;

export import std.core; // Good idea?

export void say_hello (const std::string&);
```

Modularizing Existing Code

- Build system with proper modules support
- Well modularized (in the general sense) headers

Modularizing Existing Code

Bad Idea!

```
export module hello;

export
{
#include ”hello.hxx”
}
```

Guerrilla Modularization

```
#include <string> // Pre-include out of purview.

export module hello;

export
{
#include ”hello.hxx”
}
```

Modularizing Existing Code

No mixing of inclusion and importation

in the same translation unit

Modularized Standard Library

Two Plausible Strategies:

- First switch your entire codebase to modularized `std`
- First complete modularization of your entire codebase

Modularized Standard Library

```
#include <libhello/core.hxx> // Includes <iostream>?

module hello.extra;

import std.io;
```

Modularizing Own Code

Modularize inter-dependent sets of headers

one set at a time

starting from low-level components

(newly modularized only depends on already modularized)

Modularizing Existing Code

Modularizing one component at a time?

```
#include <libhello/impl.hxx> // Imports hello.extra?

module hello.extra;
```

Backwards Compatibility

- *modules-only*
- *modules-or-headers*
- *modules-and-headers*

Modules-Only

- Follow the template and guidelines discussed above
- Seriously consider only supporting modularized `std`

Modules-or-Headers

- Expect consumers to be adjusted
- Module interface files used as headers
- FTM: `__cpp_modules` `__cpp_lib_modules`

Modules-or-Headers Interface

```
#ifndef __cpp_modules
#pragma once
#endif

// C includes, if any.

#ifndef __cpp_lib_modules
<std includes>
#endif

// Other includes, if any.

#ifdef __cpp_modules
export module <name>;
#ifdef __cpp_lib_modules
<std imports>
#endif
#endif
```

Modules-or-Headers Interface

```
#ifndef __cpp_modules
#pragma once
#endif

// C includes, if any.

#ifndef __cpp_lib_modules
<std includes>
#endif

// Other includes, if any.

#ifdef __cpp_modules
export module <name>;
#ifdef __cpp_lib_modules
<std imports>
#endif
#endif
```

Modules-or-Headers Interface

```
#ifndef __cpp_modules
#pragma once
#endif

// C includes, if any.

#ifndef __cpp_lib_modules
<std includes>
#endif

// Other includes, if any.

#ifdef __cpp_modules
export module <name>;
#ifdef __cpp_lib_modules
<std imports>
#endif
#endif
```

Modules-or-Headers Interface

```
#ifndef __cpp_modules
#pragma once
#endif

// C includes, if any.

#ifndef __cpp_lib_modules
<std includes>
#endif

// Other includes, if any.

#ifdef __cpp_modules
export module <name>;
#ifdef __cpp_lib_modules
<std imports>
#endif
#endif
```

Modules-or-Headers Interface

```
#ifndef __cpp_modules
#pragma once
#endif

// C includes, if any.

#ifndef __cpp_lib_modules
<std includes>
#endif

// Other includes, if any.

#ifdef __cpp_modules
export module <name>;
#ifdef __cpp_lib_modules
<std imports>
#endif
#endif
```

Modules-or-Headers Interface

```
#ifndef __cpp_modules
#pragma once
#endif

// C includes, if any.

#ifndef __cpp_lib_modules
<std includes>
#endif

// Other includes, if any.

#ifdef __cpp_modules
export module <name>;
#ifdef __cpp_lib_modules
<std imports>
#endif
#endif
```

Modules-or-Headers Interface

```
#ifndef __cpp_modules
#pragma once
#endif

// C includes, if any.

#ifndef __cpp_lib_modules
<std includes>
#endif

// Other includes, if any.

#ifdef __cpp_modules
export module <name>;
#ifdef __cpp_lib_modules
<std imports>
#endif
#endif
```

Modules-or-Headers Implementation

```
#ifndef __cpp_modules
#include <module interface file>
#endif

// C includes, if any.

#ifndef __cpp_lib_modules
<std includes>
<extra std includes>
#endif

// Other includes, if any

#ifdef __cpp_modules
module <name>;
#ifdef __cpp_lib_modules
<extra std imports>  // Only additional to interface.
#endif
#endif
```

Modules-or-Headers Implementation

```
#ifndef __cpp_modules
#include <module interface file>
#endif

// C includes, if any.

#ifndef __cpp_lib_modules
<std includes>
<extra std includes>
#endif

// Other includes, if any

#ifdef __cpp_modules
module <name>;
#ifdef __cpp_lib_modules
<extra std imports>  // Only additional to interface.
#endif
#endif
```

Modules-or-Headers Implementation

```
#ifndef __cpp_modules
#include <module interface file>
#endif

// C includes, if any.

#ifndef __cpp_lib_modules
<std includes>
<extra std includes>
#endif

// Other includes, if any

#ifdef __cpp_modules
module <name>;
#ifdef __cpp_lib_modules
<extra std imports>  // Only additional to interface.
#endif
#endif
```

Modules-or-Headers Implementation

```
#ifndef __cpp_modules
#include <module interface file>
#endif

// C includes, if any.

#ifndef __cpp_lib_modules
<std includes>
<extra std includes>
#endif

// Other includes, if any

#ifdef __cpp_modules
module <name>;
#ifdef __cpp_lib_modules
<extra std imports>  // Only additional to interface.
#endif
#endif
```

Modules-or-Headers Implementation

```
#ifndef __cpp_modules
#include <module interface file>
#endif

// C includes, if any.

#ifndef __cpp_lib_modules
<std includes>
<extra std includes>
#endif

// Other includes, if any

#ifdef __cpp_modules
module <name>;
#ifdef __cpp_lib_modules
<extra std imports>  // Only additional to interface.
#endif
#endif
```

Modules-or-Headers Implementation

```
#ifndef __cpp_modules
#include <module interface file>
#endif

// C includes, if any.

#ifndef __cpp_lib_modules
<std includes>
<extra std includes>
#endif

// Other includes, if any

#ifdef __cpp_modules
module <name>;
#ifdef __cpp_lib_modules
<extra std imports>  // Only additional to interface.
#endif
#endif
```

Modules-or-Headers Implementation

```
#ifndef __cpp_modules
#include <module interface file>
#endif

// C includes, if any.

#ifndef __cpp_lib_modules
<std includes>
<extra std includes>
#endif

// Other includes, if any

#ifdef __cpp_modules
module <name>;
#ifdef __cpp_lib_modules
<extra std imports>  // Only additional to interface.
#endif
#endif
```

Modules-or-Headers Consumer

```
#ifdef __cpp_modules
import hello;
#else
#include <libhello/hello.mxx>
#endif
```

Modules-and-Headers

- Old consumers must work unmodified
- Keep module interface and header files
- Slight complication over *modules-and-headers*

Questions?

build2.org

Build System Manual → C++ Modules Support

Acknowledgements

Andrew Pardoe (VC/Microsoft)
David Blaikie (Clang/Google)
Gabriel Dos Reis (VC/Microsoft)
Nathan Sidwell (GCC/Facebook)
Richard Smith (Clang/Google)