MODULES IN C++20

Dmitry Kozhevnikov

HOST

POLITECNICO MILANO

PATRON

Community Crumbs

SPONSORS

KDAB

JFrog | CONAN C/C++ package manager

JET BRAINS

develer

AIV ACCADEMIA ITALIANA VIDEOGIOCHI

sigeo Servizi Informativi Geografici

HEXAGON | Leica Geosystems

MODULES IN C++20

- Most anticipated
- Most misunderstood
- Most revolutionary
- Doesn't change the way we write programs
  (normally)

WHY MODULES?

PRESENT COMPILATION MODEL

- Inherited from C
- Uses preprocessor for code reuse
- Preprocessor doesn't know language semantics

ISSUE: SPEED

```cpp
#include <iostream>
int main() {
    std::cout << "Hello, world!" << std::endl;
    return 0;
}
```

- 30k lines on my machine, preprocessed
- Needs to pe parsed by the compiler
- 1M lines after preprocessing is not unheard of
- Superlinear growth of build times
  - Number of TUs grows
  - Number of used stuff grows

ISSUE: LEAKING NAMES

Some code from an stdlib implementation:

```
template <class _Tp, class _Allocator>
inline _LIBCPP_INLINE_VISIBILITY void
vector<_Tp, _Allocator>::push_back(const_reference __x)
{
    if (this->__end_ != this->__end_cap())
    {
        __RAII_IncreaseAnnotator __annotator(*this);
        __alloc_traits::construct(this->__alloc(),
                                  _VSTD::__to_raw_pointer(this->__end_), __x);
        __annotator.__done();
        ++this->__end_;
    }
    else
        __push_back_slow_path(__x);
}
```

ISSUE: LEAKING NAMES

Why it's so ugly?

```
//user-header.h
#define annotator 1
class RAII_IncreaseAnnotator{};
```

```
#include "user-header.h"
#include <vector>
```

ISSUE: LEAKING NAMES

- All names from previous headers are in scope
  - Macros
  - Classes
  - Global variables
- Especially bad for library writers
- Stdlib implementers can afford reserved names; others don't have this luxury

ISSUE: ODR VIOLATIONS

One Definition Rule (simplified):

- Only one definition in TU
- All definitions across the program should be identical
- Otherwise: ill-formed, no diagnostics required

ISSUE: ODR VIOLATIONS

```
// a.cpp
#include "header.h"
```

```
// b.cpp
#define _NDEBUG
#include "header.h"
```

- This could affect member layout, object size, type properties, ...
- Easy to do accidentally

ISSUE: ACCIDENTAL DEPENDENCIES

```
// lib1.h
struct Lib1Widget{};
```

```
// lib2.h
struct Lib2Gadget{
   Lib2Gadget(Lib1Widget const& w);
};
```

```
// main.cpp
#include "lib1.h"
#include "lib2.h"
```

MODULES ARE MEANT TO IMPROVE:

- Build speed
- Name isolation
- ODR violation prevention
- Source dependencies tracking

MODULES: HELLO, WORLD!

```
import std.io;
int main() {
   std::cout << "Hello, world!" << std::endl;
}
```

- Contents of std.io module is parsed separately
- Parsed module interface stored separately in compiler-specific format
- Import declaration loads pre-parsed module

MODULES: HELLO, WORLD!

```
import std.io;
int main() {
   std::cout << "Hello, world!" << std::endl;
}
```

- Dots in module name has no defined meaning
- Modules are orthogonal to namespaces
- Import declarations go on top of the file
- All relevant source information is captured:
  - Inline functions
  - Templates
  - Error messages

SIMPLE LIBRARY: HEADER-BASED

```cpp
// simple_lib.h
int foo();
template<typename T> T bar(T x) {
  return detail::baz(x);
}

namespace detail {
  template<typename T> T baz(T x) {
    return x + 1;
  }
}
```

```cpp
// simple_lib.cpp
#include "header.h"
int foo() {
  return bar(42);
}
```

SIMPLE LIBRARY: MODULAR

```cpp
// simple_lib.cppm
export module simple_lib;

template<typename T>
export T bar(T x) {
  return baz(x);
}

template<typename T> T baz(T x) {
  return x + 1;
}

export int foo() {
  return bar(42);
}
```

MODULE UNITS

Module interface unit:

```
// simple_lib.cppm
export module simple_lib;

template<typename T>
export T bar(T x) {
  return baz(x);
}

template<typename T> T baz(T x) {
  return x + 1;
}

export int foo();
```

Module implementation unit:

```
// simple_lib_impl.cppm
module simple_lib;
int foo() {
  return bar(42);
}
```

EXPORTING AND IMPORTING

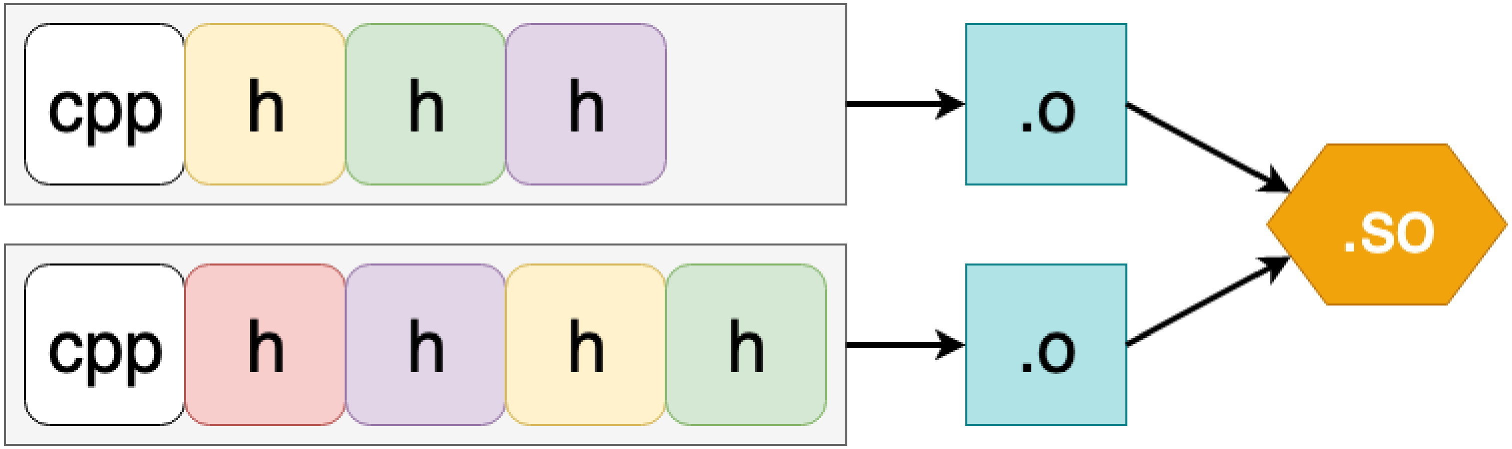
```cpp
// module1.cppm
export module Module
struct S { int x; };
export namespace N {
  S foo();
}
export S bar();
```

```cpp
// module2.cppm
export module Module
export import Module

export void baz() {
  bar();
}
```

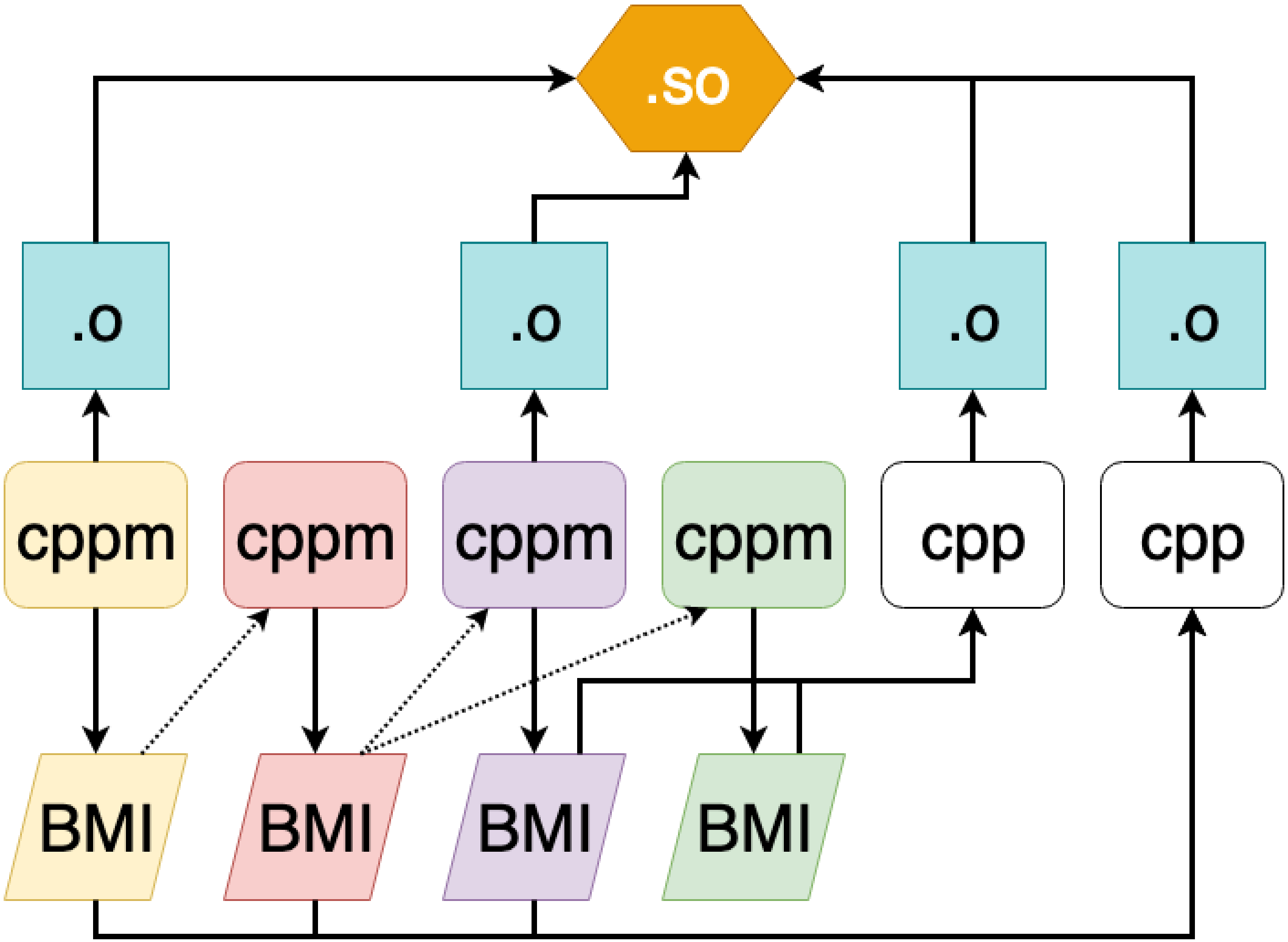
```cpp
// main.cpp
import Module2;
int main() {
  baz(); // ok
  N::foo(); // ok
  auto s1 = bar(); // ok
  s.x; // ok;
  S s2; // error
}
```

HEADER-BASED COMPILATION MODEL



- Headers are included into TUs
- No build step/artifacts for headers

MODULAR COMPILATION MODEL



- BMI: Binary Module Interface

MODULAR CODE DISTRIBUTION

BMIS ARE NOT DISTRIBUTION ARTIFACTS!

BMIS ARE NOT DISTRIBUTION ARTIFACTS!

- BMI depends on the compiler
- BMI depends on compiler flags
- Interface source should be distributed (similar to header files)
- BMIs can be distributed in limited situations
  - STL for vendor-provided toolchains (Visual C++, Xcode)
  - Known builds inside companies

MODULAR CODE DISTRIBUTION

- Distribute source for module interface units
- Distribute compiled binaries for libraries (compiled from implementation units)
- Don't distribute BMIs (except rare cases)
- Modules don't solve package distribution in any way!

LINKAGE

Controls symbol locality

- External linkage
- Internal linkage (static, unnamed namespace)
- No linkage (type aliases, ...)

What about exported symbols? Not exported?

MODULE LINKAGE

Symbol is local for the module

- External linkage (exported symbols)
- Module linkage (not exported symbols)
- Internal linkage (as before)
- No linkage (as before)

Module identity is mangled into symbol name

MODULES AND PREPROCESSOR

- Outside macros don't leak into modules
- Macros defined by a module are visible right after the import

```
// a.cppm
export module A
#define A_MACRO
```

```
// b.cppm
export module B;
#ifdef A_MACRO
  #define B_MACR
#endif
```

```
// main.cpp
import A;
import B;

// A_MACRO is defined
// B_MACRO is not defined
```

TRANSITION PATH

- Including existing header into a module might be expensive/incorrect
- Rewriting all existing code is unrealistic

HEADER UNITS

- Mark "good" header as module
- No source code change required

```
import normal.module;
import "good.h";
```

```
#include "normal/header.h"
#include "good.h" //turned into import!
```

GLOBAL MODULE FRAGMENT

```
module;
#include <windows.h>
#include <cstdio>
export module A;
```

```
module;
#include <cstdlib>
#include <windows.h>
export module B;
```

- Any header can be included
- Declarations are stored separately
- Unused declarations are discarded
- Declarations are merged across multiple modules

- Header units: graduate modularization of existing codebase
- Global module fragment: use old header in new modular code

HOW FAST ARE MODULES?

HOW FAST ARE MODULES

- Many people want modules just for build time improvements
- Workload for the compiler is aysmptotically reduced
- Build parallelism is reduced

BUILD PARALLELISM

- Pre-scanning is not parallel
- Build DAG can have points of contention
- Transferring BMIs across distributed builds is not free

WHAT BUILD TIME IMPROVEMENTS SHOULD WE EXPECT?

- For clean builds
- On real-world projects
- On real-world hardware
- For common case

WHAT BUILD TIME IMPROVEMENTS SHOULD WE EXPECT?

50%-300% BUILD TIME REDUCTION

Based on:

- Existing implementations
- Experiments
- Speculation

BAD FOR MODULAR BUILDS:

- Small, simple headers
- Little extra declarations in headers
- Heavy template-based metaprogramming
- A lot of header-only libraries with a single TU

GOOD FOR MODULAR BUILDS:

- Big headers, a lot of inline code
- A lot of extra declarations in headers
- Constexpr-based metaprogramming

BUILD TIME: OPPORTUNITIES

- Incremental compilation might be faster: less stuff invalidated with interface changes
- Edit-and-compile cycle might be faster: imported modules could be built in parallel

TOOLING STORY

BUILD SYSTEM COMPLICATIONS

- Where to look for modules?
- What to compile first?

WHERE TO LOOK FOR MODULES?

Module name is spelled inside module unit:

- Not related to module source file name
- Not related to BMI name

SOLUTIONS

- Pass explicit mapping to the compiler
- Filename-based lookup
- Discovery server

WHAT MODULE TO COMPILE FIRST?

```
export module A
import B;
```

```
export module B
```

```
export module A
```

```
export module B
import A;
```

SOLUTIONS

- Manually specify dependencies in the build script
- Pre-scan sources
- Build BMIs on the fly

MODULE PREAMBLE

Import declarations should be placed in the beginning.

- Still affected by preprocessor
- Need to figure out where it ends
- Not applicable for header units

ANALYZERS AND IDES

Headers cause troubles

- In what context to analyze?
- What to do with non-self-contained?
- What to do with diagnostics from headers
- A lot of existing IDEs use modular-like optimizations for performance reasons

Modules solve most of this, but might require deeper integration with compiler and build systems

CURRENT STATE OF IMPLEMENTATIONS

- clang: ongoing development, experimental support available (-fmodules-ts)
  - Note: "Clang modules" (-fmodules) are NOT C++20 modules!
- Visual C++: experimental support available
- GCC: ongoing development in a branch
- Modularized STL is not planned for C++20, some implementation exist

CURRENT STATE OF BUILD SYSTEM SUPPORT

- build2 pioneered the effort
- CMake: work in progress
- Ninja: a patch has landed recently to make it possible
- Complicated with Make (unless explicitly specifying dependencies)

SUMMARY

- Modules are accepted in C++20
- More hygienic code reuse model
- Build time improvements (maybe not that large)
- Implementation and tooling are being worked on, but not there yet

THANK YOU!

QUESTIONS?