C++20 Modules

Kaspar Giger, bbv Software Services AG | April 17, 2024

Agenda

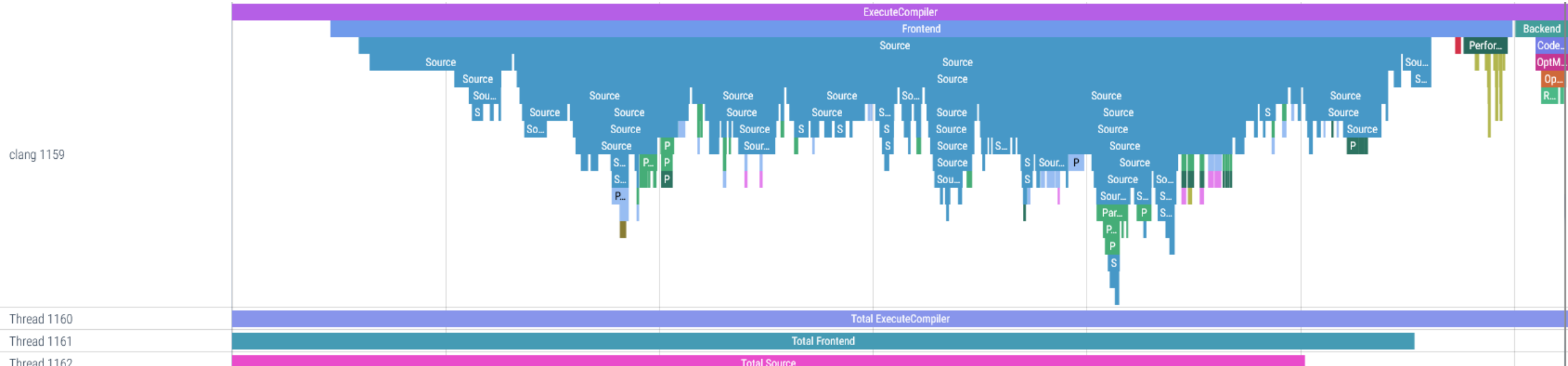
1. **History and Motivation**
   Why do we need modules at all?
2. **Modules**
   What are modules?
3. **Using Modules**
   Export, import, CMake
4. **Experience**
   Can they be used?
5. **Conclusion**
   My personal view on modules

History and Motivation

Compiling C++ Code

```cpp
#include <iostream>

int main()
{
    std::cout << "Hello, World!" << std::endl;
    return 0;
}
```

- Results in 32’000 lines to be compiled (Clang 18)!
- Heavy work for the pre-processor (“frontend”): (using Clang’s `-ftime-trace`)



Pre-processor

- Introduced (in C) to enable “code re-use” and modularization
  - Made C quite popular
  - Simple and elegant solution

- But (in the light of modern C++):
  - Limited safety
  - Poor readability
  - Debugging challenges
  - Namespace pollution

Problem

```cpp
// a.cpp
#include <memory>

void foo()
{
    std::shared_ptr<int> x;
    // ...
}
```

```cpp
// b.cpp
#include <memory>

void bar()
{
    std::unique_ptr<int> z;
    // ...
}
```

- Duplication of work:
  `memory` (and its dependants) are included each time!

- `#include` means: read the file, parse the code, optimize the code
- Some libraries available *“header-only”*

Motivation

Speed-up

- Avoid duplication

Better code

- Stick to “one definition rule”
- Avoid cyclic dependencies

History of Modules

- Proposal in 2004 (N1736)
- Study group in 2012
  - Early implementations in Clang and MSVC
- Technical specification in 2018
- Competing proposal by Google
- Merged both proposals in 2019

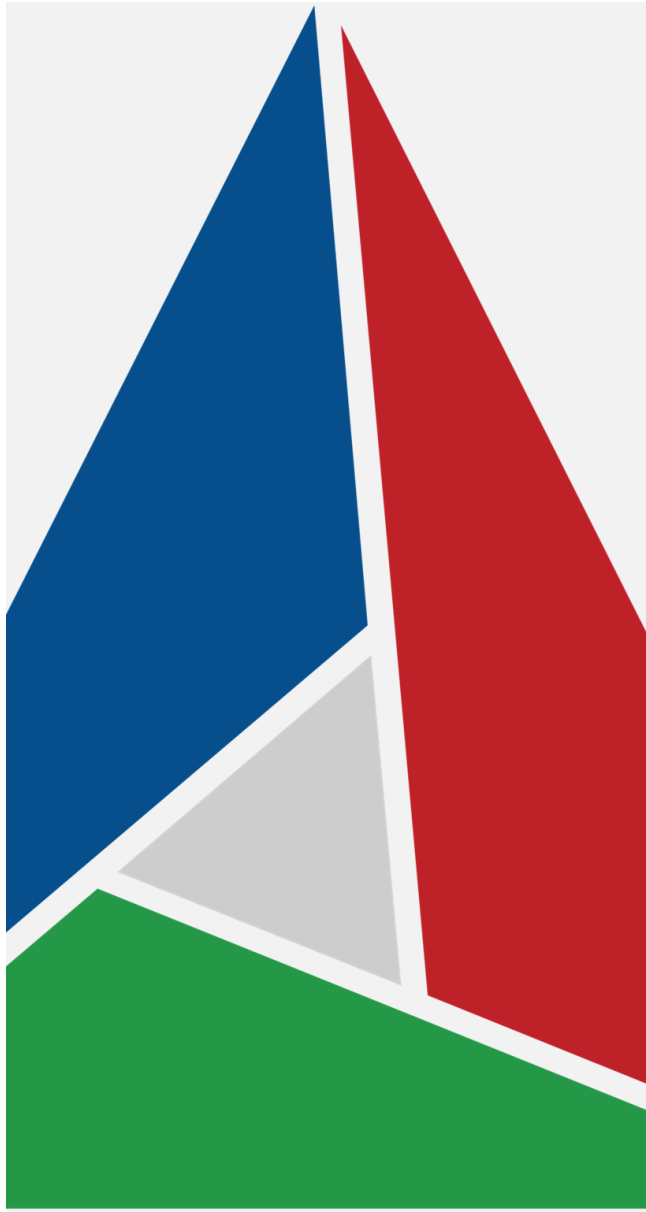
- Modules in C++20
  - GCC ≥ 11, Clang ≥ 8, MSVC ≥ 2015 [1]
- C++ standard library as modules in C++23
  - GCC -, Clang ≥ 17, MSVC ≥ 19.35 [1]

[1] https://en.cppreference.com/w/cpp/compiler_support

Compiling and Using Modules (Clang)

```
$> clang++ -std=c++20 my_module.cppm --precompile -o my_module.pcm

$> clang++ -std=c++20 main.cpp -fmodule-file=my_module=my_module.pcm \
       my_module.pcm -o my_program
```

- Very ugly, very intricate
- Needs support by the build system
  - Which file is a module?
  - Dependencies the file has (for re-compilation after changes!)?
- CMake 3.28 officially supports C++20 modules! (Dec. 2023) [2]
  - Needs
    - Ninja ≥ 1.11.1 (2022 August)
    - GCC ≥ 14 (2024 April)
    - Clang ≥ 16 (2023 April)
    - MSVC ≥ 14.34 (2022)



[2] https://www.kitware.com/import-cmake-the-experiment-is-over/

Modules

Modules?

bbv

Precompiled headers

- Compile time optimization
- Just for header inclusion
- Compiler specific

Modules

- Compile time optimization
- More towards code organization
- Standardized
- Dependency management
- Customize what’s exported

Module!

bbv

Binary compiler output

- Binary format
  - Fast to read and process
  - Not standardized!
  - No porting between compilers (how do you ship your library?)
- Contains:
  - Declarations (exported types, functions, variables)
  - Metadata (name, dependencies, versioning, …)
  - Optimized code (essentially the AST)

Using Modules

First Module

```
// libbar.cpp
export module bar;

namespace bar
{

export int someBarFunc()
{
    return 42;
}

} // namespace bar
```

```
// main.cpp
import bar;

int main()
{
    return bar::someBarFunc();
}
```

- `export module <some name>;`
  - Create a module <*some name*>
  - <*some name*> any valid identifier
    (you can see names with “.” - valid, no meaning)
  - One module per file!
- `export <some entity> ...`
  - What’s exported, i.e. visible outside
  - Every exported entity needs the `export` keyword
  - Can export:
    - Functions, variables, classes, enums, templates, concepts
    - Namespaces (entire namespace!)
    - Other modules (`export import my_module;`)

First Module

```
// libbar.cpp
export module bar;

namespace bar
{

export int someBarFunc()
{
    return 42;
}

} // namespace bar
```

```
// main.cpp
import bar;

int main()
{
    return bar::someBarFunc();
}
```

- `import <module name>;`
  - Imports the module *<module name>*
  - Equivalent to `#include <module name.h>`
  - Order of imports doesn't matter!
  - Be careful: same definition in multiple modules (didn't see a linker error!)

Splitting Into Multiple Files

- Can implement module in multiple files
- One file (with `export module <name>;`) defines the interface (=all exports)
- Other files contribute to the implementation
  - Just start with `module <name>;`

- Essentially similar to 1 .h-file, N .cpp-files

```cpp
// libfoo1.cpp
export module foo;

export int foo1()
{
    return 314159;
}
export int foo2();
```

```cpp
// libfoo2.cpp
module foo;

int foo2()
{
    return 17;
}
```

Headers and Modules

```
// libbar_header_units.cpp
export module bar;

import <iostream>;

export void someBarFunc()
{
    std::cout << "hi bar\n";
}
```

```
// libbar_headers.cpp
module;

#include <iostream>

export module bar;

export void someBarFunc()
{
    std::cout << "hi bar\n";
}
```

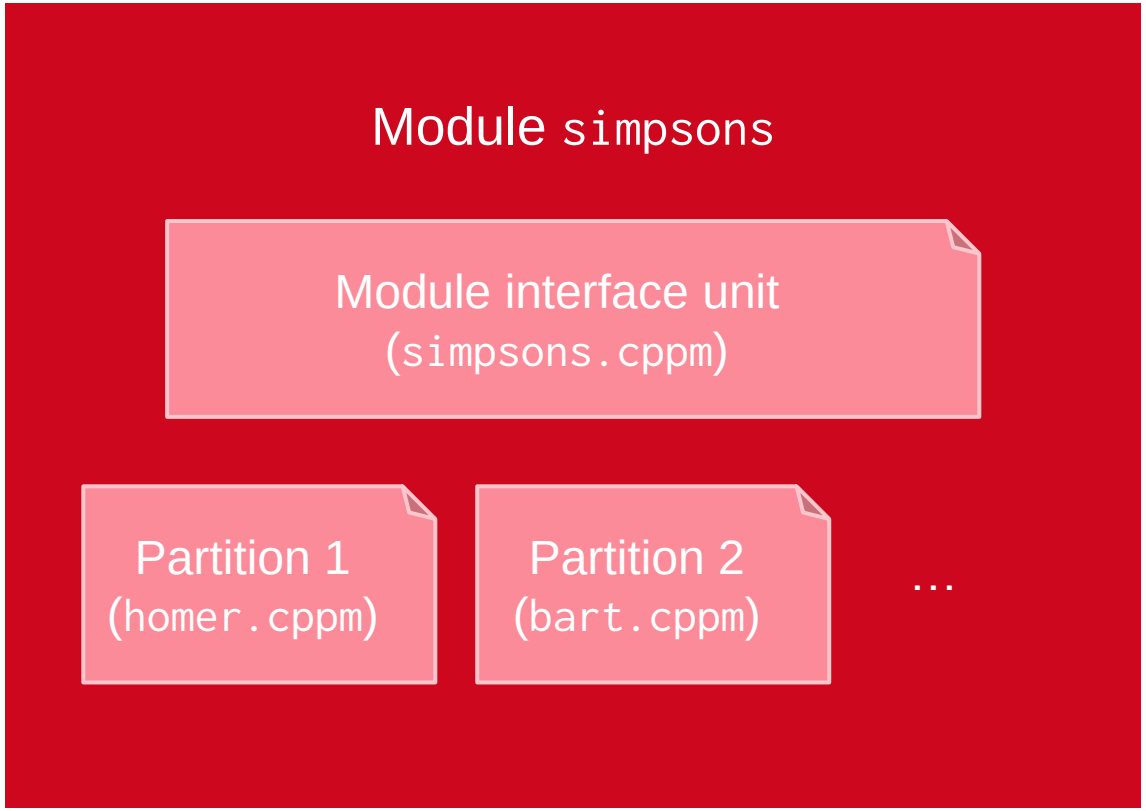
- Actually: prefer header units
  (because headers are old-fashioned)
- Header units = *“A header unit is a binary representation of a header file”* [3]
- Currently not supported by build tools

- Include headers in some “global module fragment”
  (code between an initial `module;` and module declarations)
  - This way included headers are not considered part of the module

[3] https://learn.microsoft.com/en-us/cpp/build/walkthrough-header-units

Partitions

- Implement module in multiple files
- Each file defines a *«partition»* of the module (some parts of the module)
- One *«module interface unit»* assembles the partions and serves as public interface



Partitions

- Partitions separated by ":"
  - Before: module name
  - After: partition name
- Can import other partition(s) (of the same module)
  - All definitions are imported!
    (no `export class Person` !!)
- One «module interface unit» with the
  `export import :<partition name>;`
- There can be «purely internal» partitions
  (see `simpsons:person`)

```
// person.cppm
export module simpsons:person;

class Person
{
    // ...
};
```

```
// homer.cppm
export module simpsons:homer;

import :person;

export class Homer : public Person
{
    // ...
};
```

```
// simpsons.cppm
export module simpsons;

export import :homer;
export import :bart;
```

CMake Configuration

Library with header (legacy way)

```
set(NAME libbar_header)

add_library(${NAME})
add_library(${NAME}::${NAME} ALIAS ${NAME})

target_sources(${NAME}
    PRIVATE
        include/libbar.h
        src/libbar.cpp
)

target_include_directories(${NAME}
    PUBLIC
        include
)
```

Library as module

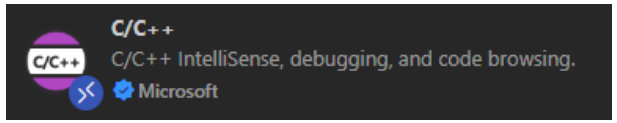
- Define `FILE_SET` for files exporting modules (in `target_sources(…)`)

```
set(NAME libbar_module)

add_library(${NAME})
add_library(${NAME}::${NAME} ALIAS ${NAME})

target_sources(${NAME}
    PUBLIC
        FILE_SET CXX_MODULES FILES
            libbar.cppm
)
```

Experiences with Modules

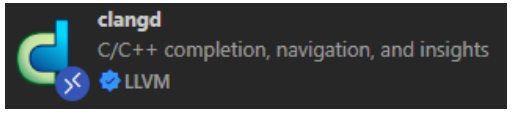
Tooling (VS-Code)

Microsoft C++ Intellisense



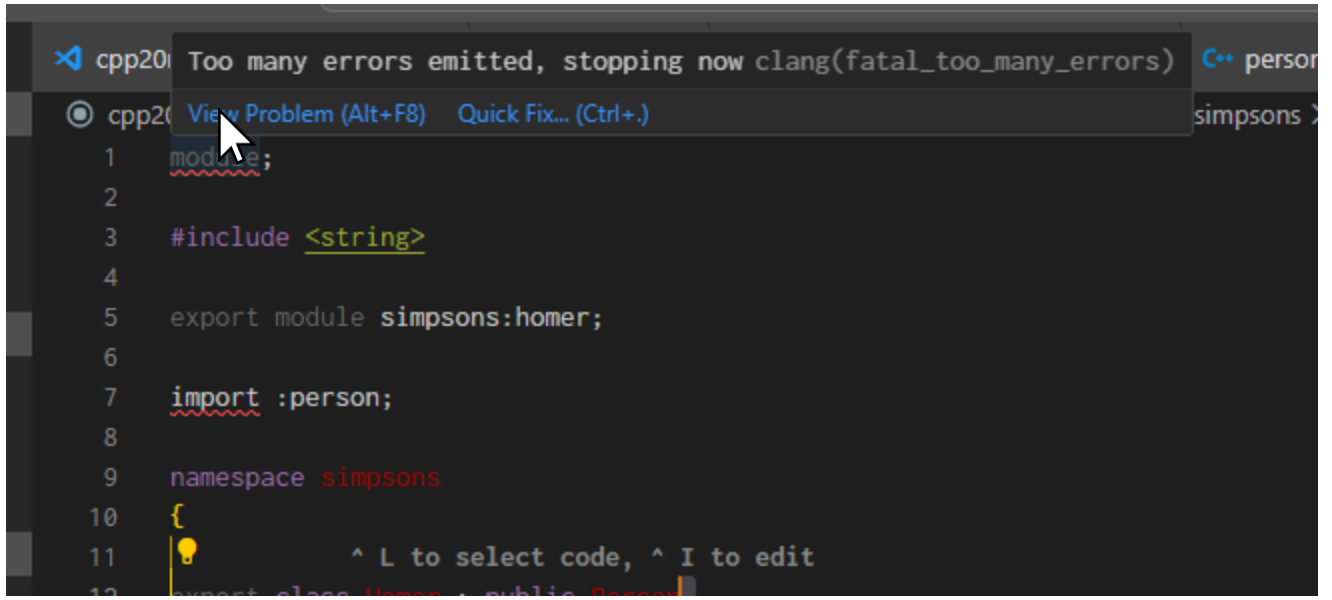
- Doesn't really work

Clangd extension



- Works better (Clang 17 and 18)
  - Still weird errors…

➔ Demo…



Modules in an Existing C++ Repo

Experimented in large C++ code base with ~11’000 translation units

Costly library

- Maximum performance gain
- ✘ Failed desperately due to
  - dependency hell
  - forward declarations
  - macro magic

Small library

- (✓) Some migrated successfully
- Some failed because of magic compiler errors (Clang 16)

```
Zs/unitZZZZExtended/CMakeFiles/UnitZZZZExtended_ZZZZ.dir/src/ZZZZ_PresentCheck.cpp.o
Files/UnitZZZZExtended_ZZZZ.dir/src/ZZZZ_PresentCheck.cpp.o
est ccache /usr/bin/clang++-16 -DBOOST_MPL_CFG_NO_PREPROCESSED_HEADERS ......
/user/Test/Platform/ZZZZ/include/ZZZZ/ImgProc/Contour.h:8:
./include/c++/10/compare:355:22: error: 'std::__detail::__common_cmp_cat' has different definitions in different
                                      modules; definition in module 'utility.regression.<global>' first differenc
                                      is function body


./include/c++/10/compare:355:22: note: but in '' found a different body
```

Modules in an Existing C++ Repo

Experimented in large C++ code base with ~11’000 translation units

Costly library

- Maximum performance gain
- ✘ Failed desperately due to
  - dependency hell
  - forward declarations
  - macro magic

Small library

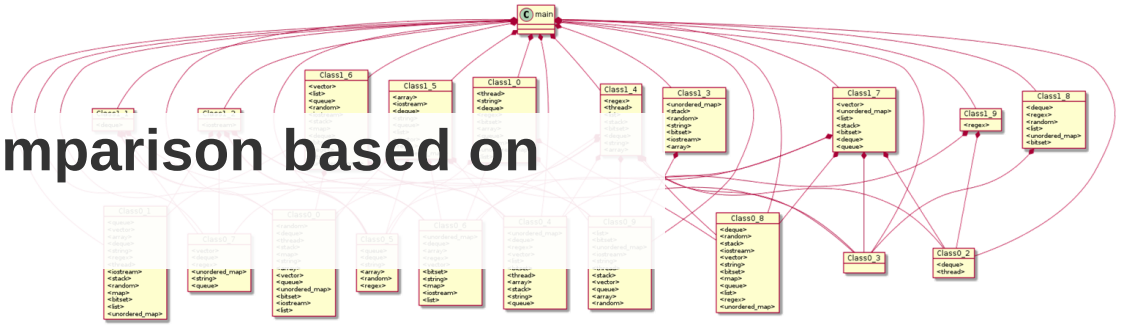
- (✓) Some migrated successfully
- Some failed because of magic compiler errors (Clang 16)

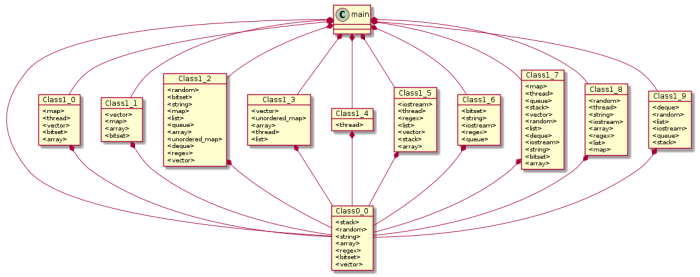
```
xtended_ZZZZ.dir/src/ZZZZ_PresentCheck.cpp.o
Z_PresentCheck.cpp.o
T_MPL_CFG_NO_PREPROCESSED_HEADERS ......
ZZ/ImgProc/Contour.h:8:
./include/c++/10/compare:355:22: error: 'std::__detail::__common_cmp_cat' has different definitions in different
                                 modules; definition in module 'utility.regression.<global>' first differenc
                                 is function body


./include/c++/10/compare:355:22: note: but in '' found a different body
```

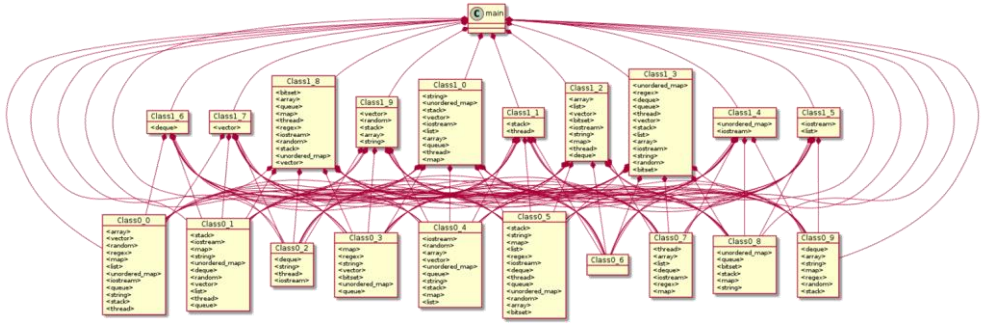
A View on Build Times

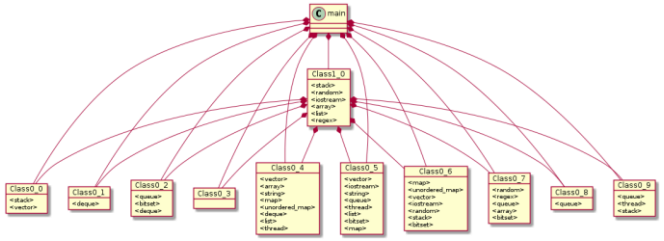
Build performance comparison based on generated C++ code

- Very artificial!
- Various topologies
- Generated code:
  - 1 file = 1 class
  - Include random STL headers
  - Random class members, function, comments
  - 1x with headers (traditional)
  - 1x with modules



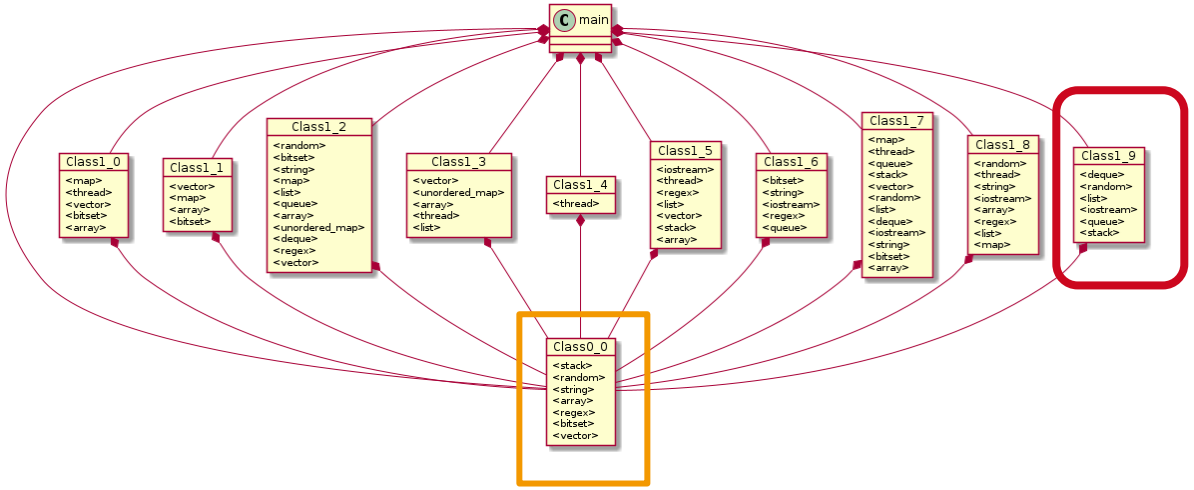






```
module;
#include <regex>

import class0_2;
import class0_4;

export module class1_9;

export class Class1_9
{
public:
    Class1_9() = default;
    ~Class1_9() = default;

    unsigned bingkxs0(unsigned, char, double, float, int);
    double xheirvgbutyxgvy1();
    short gykjbbqfk2(float, double, short, float, float);
    int frr3(char, short, short, unsigned);
    char lnhwsfd4(int, short, float);
    float pfenviwudzlsvjaubpd5(char);
    double uaclxxvinprasu6(char, int);
    float kslkmdwsorugff7(float, float, short);
    double xhppk8(int, int, char, double);
    double hg9(int, double, short, int);
    int wxewlihxjguqxugj10(float, int, int, char);
    double zyaarvjssdt11(float, float);
    unsigned iylvl2(float, unsigned, int, float, double);
    char brxwtbwltmos13(short);
    float dabnhumrqqzggcy14(unsigned, float, short);
    float jluudtakknprm15(char, short, unsigned, char);
    char ivaisiwjxhexteew16(short);

private:
    int pj1(double, float, short, char);
    float qcv2(float, float);
    float ybjspiubmgu3(short, double, short, int, unsigned);
    unsigned azl4();
    char wawsjghlsivtnpdr5(float);
    int lcsayyhtjgh6(double, char, int, float);
    short epdbdfk7();
    int ynacqjtydvswjbn8(double, float, short, float);
    char yguevnfw9(short, double, int, char, int);
    double ccddhhhnb10(double, double, double, float);
    unsigned uan11(short, short, char, double, short);
    char pxlfvuvzxmdyivllrh12(char, float, double, double);
    int p13(double, unsigned);

    // some random doxygen docu 0
    Class0_4 m_member_internal0_0{};
    // some random doxygen docu 0
    Class0_2 m_member_internal1_0{};
    std::regex m_member_stl0_0{};
    std::regex m_member_stl0_1{};
    std::regex m_member_stl0_2{};
    std::regex m_member_stl0_3{};
    std::regex m_member_stl0_4{};
    std::regex m_member_stl0_5{};
    std::regex m_member_stl0_6{};
    std::regex m_member_stl0_7{};
    std::regex m_member_stl0_8{};
};
```

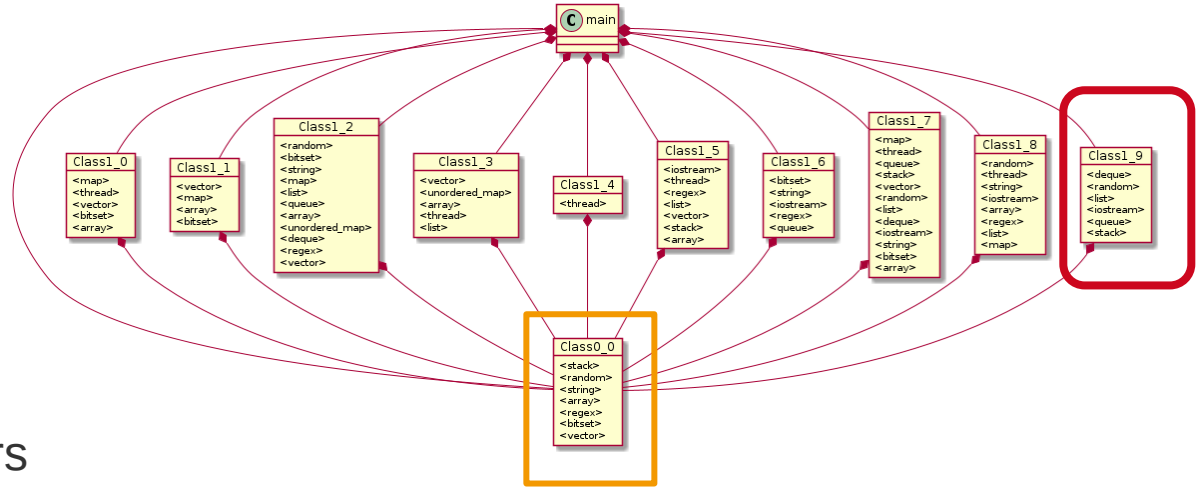
A View on Build Times



Detailed view (headers)

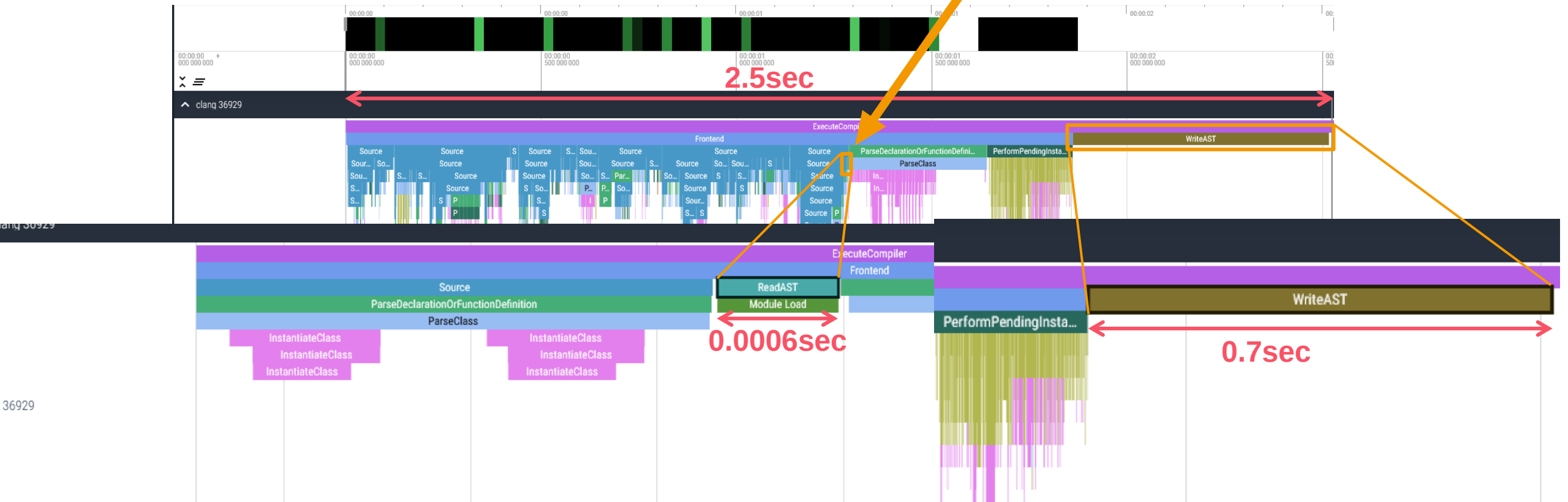
- `class1_9.cpp` includes some STL headers and `class0_0.h`

A View on Build Times



Detailed view (headers)

- `class1_9.cppm` includes some STL headers and imports `class0_0`



A View on Build Times

Speedup measurements

- 5 files (per layer): [-14..+14%]
- 10 files (per layer): [-13..+18%]
- 20 files (per layer): [-90..+2%]

(+ → speedup, - → slowdown)

Conclusion

Conclusions

bbv

Performance

- Importing modules is fast!
- Overall gain: depends…
  ⇨ don’t expect miracles
  - Maybe few %!
    (confirms with others)

Tooling

- Use latest tools
  - Clang ≥ 17 (my recommendation)
  - CMake ≥ 3.28 (needed)
  - Ninja ≥ 1.11.1 (needed)
- VS-Code
  - Use Clangd extension for Intellisense

General

- Still ongoing discussions
- `import std;` will have introduce some momentum

My recommendations

- Great concept!
  - Helps keeping your code modern
- Existing project ⇨ just don’t use modules (at least not now…)
  - Not enough gain, painful migration, external libraries!?

- New project ⇨ maybe
  - External libraries?
  - Latest, latest tools

- Wait for `import std;` - at least!
- Wait for better tool support
- Wait for the community to establish best practices (partitions? Module size? …)