

КАША ИЗ ТОПОРА

Зачем нужны модули в C++ и почему это так больно?

К. Владимиров, Syntacore, 2025
mail-to: konstantin.vladimirov@gmail.com

Что не так с ЭТИМ КОДОМ?

```
while (dlink) {
    b = tip[dlink], dlink = next[dlink]; l = final_level;
enter_level:
    boy[l] = b;
advance:
    if (blink[b]) {
        g = tip[blink[b]], blink[b] = next[blink[b]];
        if (mark[imate[g]] < 0) goto advance;
        b = imate[g], l--; goto enter_level;
    }
    if (++l > final_level) continue;
    b = boy[l]; goto advance;
}
```

Дисциплина для меньшей сложности

- Убираем `goto`, предпочитаем структурные конструкции, декомпозируем код на понятные небольшие блоки.
- Стараемся убрать явные циклы, заменив их на алгоритмы стандартной библиотеки (или на собственные функции написанные в том же стиле).

Что не так с ЭТИМ КОДОМ?

```
int build_tree(cube_t*& cube, const coords_t& coords){
    cube = new cube_t{coords};
    int count_inside = 0, i = 0;
    for (auto triangle : triangles_) {
        if (cube->is_triangle_in(*triangle.first)) {
            cube->triangles_.push_front(triangle);
            count_inside++; in_chilids_parent[i] = true;
        }
        i++;
    }

    if (count_inside < CNT_TRESHOLD) { delete cube; return 0; }
    // [...]
```

Дисциплина для меньшей сложности

- Убираем `goto`, предпочитаем структурные конструкции, декомпозируем код на понятные небольшие блоки.
- Стараемся убрать явные циклы, заменив их на алгоритмы стандартной библиотеки (или на собственные функции написанные в том же стиле).
- Убираем явные `new`, предпочитаем контейнеры или умные указатели.
- Проектируем код для безопасности исключений, следим за гарантиями типов и за нейтральностью функций.

Что не так с ЭТИМ кодом?

```
// foo.cc
#include "bar.h"
#include "foo.h"
namespace A {
    int foo(int);
    int baz(int); // uses B::bar
}
```

```
// bar.cc
#include "bar.h"
namespace A { int foo(int); }
namespace B {
    int bar(int); // uses A::foo
}
```

- Кстати, здесь более одной проблемы. Что вас более всего тревожит?

Что не так с ЭТИМ кодом?

```
// foo.cc
#include "bar.h"
#include "foo.h"
namespace A {
    int foo(int);
    int baz(int); // uses B::bar
}
```

```
// bar.cc
#include "bar.h"
namespace A { int foo(int); }
namespace B {
    int bar(int); // uses A::foo
}
```

- Кстати, здесь более одной проблемы. Что вас более всего тревожит?
- Вас, видимо, тревожила, в том числе циклическая зависимость. Хм...
- > gcc foo.cc bar.cc [...] -lgcc -lc -lgcc

ОСНОВЫ КОМПОНЕНТНОГО ПОДХОДА

- Модуль включает свой хедер первой строчкой перед прочими хедерами.

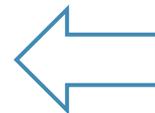
```
// foo.h  
#pragma once
```

```
// foo.cc  
#include "foo.h"  
#include "bar.h"
```

ОСНОВЫ КОМПОНЕНТНОГО ПОДХОДА

- Модуль включает свой хедер первой строчкой перед прочими хедерами.
- Циклические зависимости между компонентами должны быть исключены.

```
// bar.cc
#include "bar.h"
#include "foo.h"
namespace B {
    int bar(int); // uses A::foo
}
```



```
// foo.h
#pragma once
```

```
// foo.cc
#include "foo.h"
#include "bar.h"
namespace A {
    int foo(int);
    int baz(int); // uses B::bar
}
```

ОСНОВЫ КОМПОНЕНТНОГО ПОДХОДА

- Модуль включает свой хедер первой строчкой перед прочими хедерами.
- Циклические зависимости между компонентами должны быть исключены.
- Все внешние сущности, определённые в модуле объявлены в его хедере и наоборот.

```
// foo.h
#pragma once
namespace A {
    int foo(int); int baz(int);
}
```

```
// foo.cc
#include "foo.h"
#include "bar.h"
namespace A {
    int foo(int);
    int baz(int); // uses B::bar
}
```

ОСНОВЫ КОМПОНЕНТНОГО ПОДХОДА

- Модуль включает свой хедер первой строчкой перед прочими хедерами.
- Циклические зависимости между компонентами должны быть исключены.
- Все внешние сущности, определённые в модуле объявлены в его хедере и наоборот.
- Внешние пользователи получают доступ к функционалу компонента только через его хедер.

```
// foo.h
#pragma once
namespace A {
    int foo(int); int baz(int);
}
```

```
// foo.cc
#include "foo.h"
#include "bar.h"
namespace A {
    int foo(int);
    int baz(int); // uses B::bar
}
```

Компоненты и единицы трансляции

```
// component.h  
#pragma once  
#include "general.h"
```

```
// general.h  
#pragma once  
int foo(int x);  
int bar(int x);
```

```
// general_bar.cc  
#pragma once  
int bar(int x);
```

```
// component.cc  
// uses internal functions  
#include "component.h"  
#include "internals.h"  
int foo(int x) {  
    return baz(3) - bar(2);  
}
```

```
// internals.h  
#pragma once  
int baz(int x);
```

```
// internals.cc  
#pragma once  
int baz(int x);
```

Дисциплина для меньшей сложности

- Убираем `goto`, предпочитаем структурные конструкции, декомпозируем код на понятные небольшие блоки.
- Стараемся убрать явные циклы, заменив их на алгоритмы стандартной библиотеки (или на собственные функции написанные в том же стиле).
- Убираем явные `new`, предпочитаем контейнеры или умные указатели.
- Проектируем код для безопасности исключений, следим за гарантиями типов и за нейтральностью функций.
- Проектируем физические компоненты для уменьшения связности, определяем интерфейсы, через которые они используются.
- Избегаем циклических зависимостей по нашим компонентам.

Общее направление языка

- **Чем сложнее язык, тем проще на нём писать.**
- Мы действительно со всех сторон прижаты идиомами, не учитывать которые себе дороже.
- Мы сами ограничиваем своё "хочу" (хочу и поставлю goto, хочу и выделю память) ради качества кода.
- Важно, что мы всегда можем в этом самоограничении рассчитывать на поддержку со стороны языка программирования.
- И сейчас в языке C++ есть ровно одна область, где это не так.

Improving physical design – forward declarations (1)

```
// zoo.h
#pragma once
class animal;

class zoo
{
public:
    animal get_random_animal();
    void add_animal(animal);

    zoo(); // ←
    ~zoo(); // ←

private:
    std::vector<animal> _animals;
};
```

```
// zoo.cpp
#pragma once
#include "zoo.h"
#include "animal.h"

animal zoo::get_random_animal() { /* ... */ }

void zoo::add_animal(animal x)
{
    _animals.push_back(x);
}

zoo() = default; // ←
~zoo() = default; // ←
```

- The code above is now completely valid!
 - `animal.h` is only included in `zoo.cpp`
- Using forward declarations can greatly reduce header dependencies
 - Libraries should provide “*forward headers*” for their users

vittorioromeo.com | mail@vittorioromeo.com | vromeo5@bloomberg.net | @supahvee1234 | github.com/vittorioromeo/accu2023 | (C) 2023 Bloomberg Finance L.P. All rights reserved.



ACCU.ORG



Vittorio Romeo

Думаете это частное мнение?

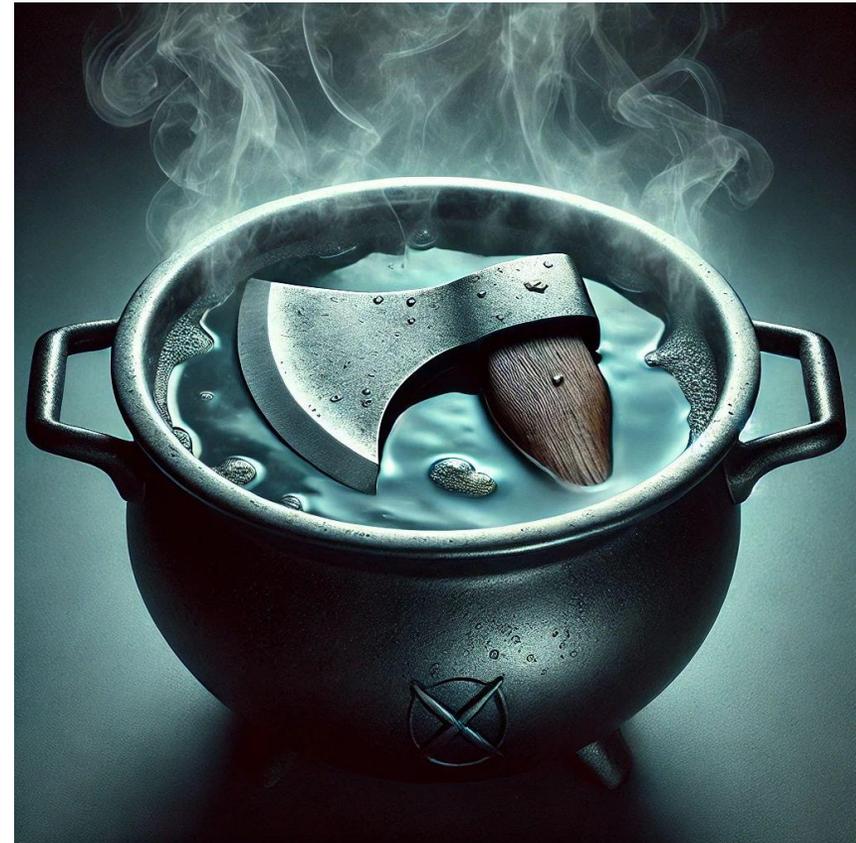
`#include` as Little as Possible

`#include` hurts compile time performance. Don't do it unless you have to, especially in header files.

But wait! Sometimes you need to have the definition of a class to use it, or to inherit from it. In these cases go ahead and `#include` that header file. Be aware however that there are many cases where you don't need to have the full definition of a class. If you are using a pointer or reference to a class, you don't need the header file. If you are simply returning a class instance from a prototyped function or method, you don't need it. In fact, for most cases, you simply don't need the definition of a class. And not `#include`ing speeds up compilation.

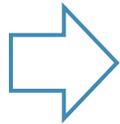
И вот у нас появился топор

- «**Not #include'ing speeds up compilation**»
- Точно ли мы **не умеем** решать эту проблему?
- Точно ли мы **хотим** решить именно эту проблему?
- В любом случае начать варить хотя бы топор это хороший старт.



Precompiled headers

```
// stdafx.h
#include <cassert>
#include <iostream>
#include <string>
#include <vector>
```



```
// stdafx.h.pch
CPCH^A^H^@^@&^K^@^@
^GÁ³Đ<8c>Â+, ^B-
ÈÂ+°Â/<88>^B+¼Â(-ÂC
^R´B)ĐB(<90>B(ĐB(<0
```



```
// mypch.cc
#include "stdafx.h"
std::vector<int> v;
std::string s;
```

- Основная идея в том, что мы делаем предварительную компиляцию до синтаксического дерева.

```
$ clang++ -x c++-header -c stdafx.h -o stdafx.h.pch
```

```
$ clang++ -S -include stdafx.h test.cc
```

- Но тут сразу возникают вопросы. Например, что если разойдутся опции?

Состав PCH и их структура.

```
$ clang++ -module-file-info stdafx.h.pch
```

```
Information for module file 'stdafx.h.pch':
```

```
Module format: raw
```

```
Generated by this Clang: (1ubuntu1)
```

```
Language options:
```

```
  C++17: Yes
```

```
  C++20: No
```

```
  C++ exceptions: Yes
```

```
Target options:
```

```
  Triple: x86_64-pc-linux-gnu
```

```
Preprocessor options:
```

```
  Uses compiler/target-specific predefines [-undef]: Yes
```

```
Predefined macros:
```

```
  -D__GCC_HAVE_DWARF2_CFI_ASM=1
```

```
Input file: /usr/include/features.h [System]
```

- METADATA
- SOURCE INFO
- PREPROCESSOR STATE
- Serialized AST
 - Types
 - Decls

Контроль корректности для PCH

```
$ rm stdafx.h
$ clang++ -S -include stdafx.h mypch.cc
fatal error: malformed or corrupted AST file stdafx.h
referenced by AST file stdafx.h.pch

$ svn revert stdafx.h
$ clang++ -S -include stdafx.h mypch.cc
fatal error: file stdafx.h has been modified since the
precompiled header 'stdafx.h.pch' was built: mtime changed

$ clang++ -x c++-header -c stdafx.h -o stdafx.h.pch
$ clang++ -S -O2 -include stdafx.h mypch.cc
error: __OPTIMIZE__ predefined macro was disabled in PCH file
but is currently enabled
```

Немного убогих трюков

```
$ clang++ -x c++-header -DMYDEF=1 -c stdafx.h -o stdafx.h.pch
```

```
$ clang++ -S -DMYDEF=0 -include stdafx.h mypch.cc
```

```
error: definition of macro 'MYDEF' differs between the  
precompiled header ('1') and the command line ('0')
```

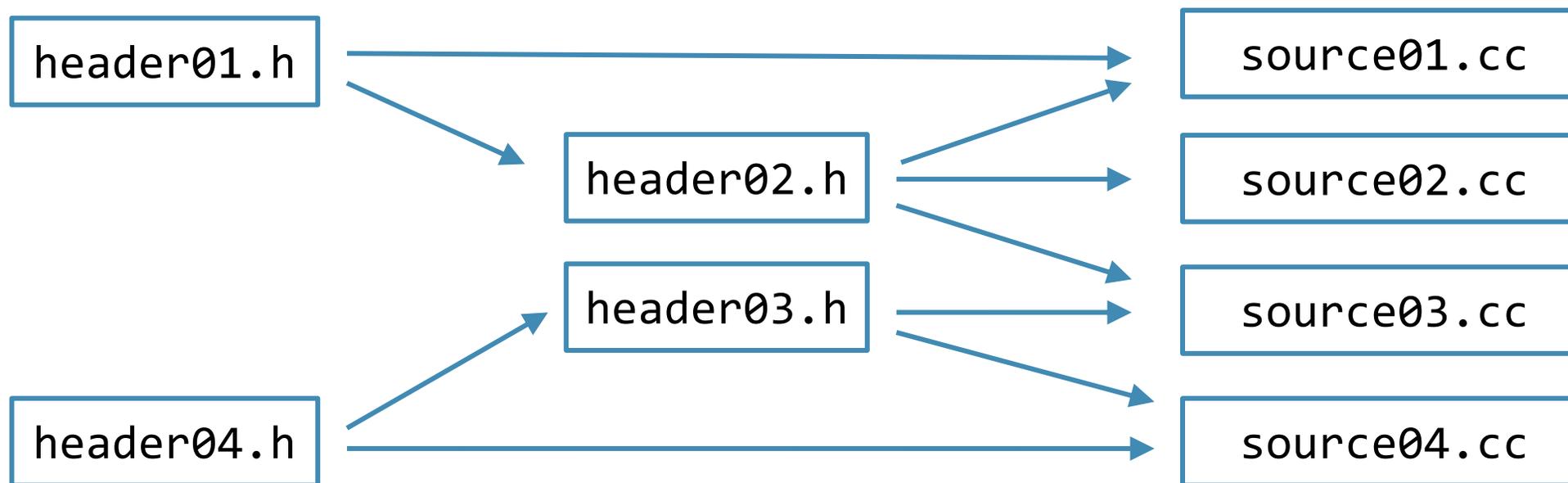
```
$ mkdir stdafx.h.pch
```

```
$ clang++ -x c++-header -DMYDEF=1 -c stdafx.h \  
-o stdafx.h.pch/stdafx.h.1.pch
```

```
$ clang++ -x c++-header -DMYDEF=0 -c stdafx.h \  
-o stdafx.h.pch/stdafx.h.0.pch
```

```
$ clang++ -S -DMYDEF=0 -include stdafx.h mypch.cc
```

Исходная картина: DAG



PCN: неизбежный fork-join



Проблема: неконтролируемый экспорт

```
// afx2.h
namespace _detail {
    int foo();
}
int bar(int* a) {
    a[0] = _detail::foo();
    // .....
}
```

- Прекомпилированный хедер один и в таком качестве идёт первым, поскольку состояние препроцессора нельзя разумно смержить.
- Невозможно по настоящему скрыть служебную функцию foo.
- И тут к нашему топору надо бросить кусочек мяса...

Кусочек мяса: контроль экспорта

```
// afx2.h
namespace _detail {
    int foo();
}
int bar(int* a) {
    a[0] = _detail::foo();
    // .....
}
```



```
// afx2.cppm
export module afx2;
int foo() { ..... }
export int bar(int* a) {
    a[0] = foo();
    // .....
}
```

Работа с модулями

```
// afxm.cppm  
// exports: bar  
// hides: foo
```



```
// afx.pcm  
CPCH^A^H^@^@&^K^@^@  
^GÁ³Đ<8c>Â+, ^B-  
ÈÂ+°Â/<88>^B+¼Â(-ÂC
```



```
// muser.cc  
import afxm;  
// uses: bar
```

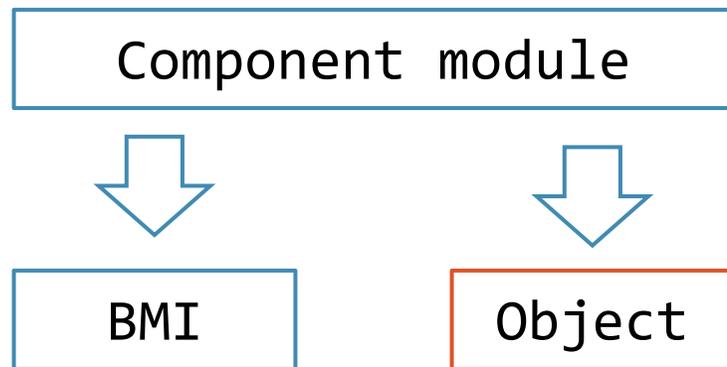
```
$ clang++ -std=c++23 --precompile afxm.cppm -o afxm.pcm
```

```
$ clang++ -std=c++23 muser.cc -fmodule-file=afxm=afxm.pcm afxm.pcm
```

- Обратите внимание `-fmodule-file=name=file` даёт возможность посмотреть в файл `file` за модулем с именем `name`.
- Но теперь импорт можно повторить сколько угодно раз.

Многоликий ВМІ

- Для clang стандартное расширение ВМІ файлов это PCM
- Для gcc это GCM
- Для cl (MSVC) это IFC



- The artifact created by a compiler to represent a module unit or header unit.
- The format for this representation is implementation specific and holds C++ entities, which can be represented in the form of
 - compiler specific data structures (e.g. ASTs, metadata, etc),
 - machine code (object files)
 - any intermediate representation chosen by the implementer.

Немного соли: collate

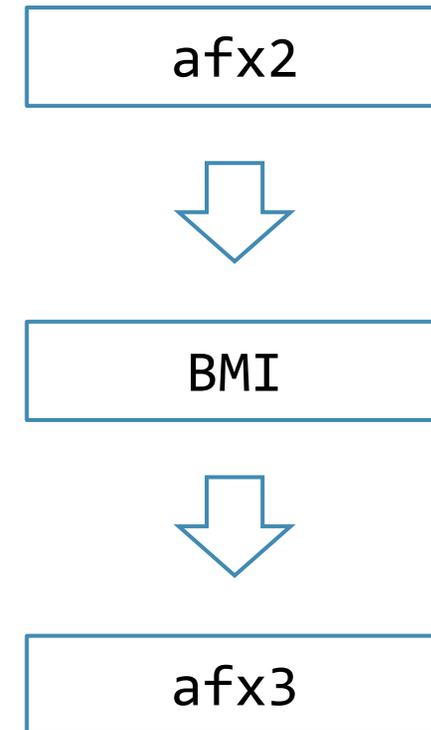
- Документ p1838, упомянутый на прошлом слайде это очень важный документ, который определяет терминологию (BMI, Dependency metadata).
- Но ещё важнее документ p1696, который определяет общий формат информации о зависимостях и позволяет её собирать.

```
> clang-scan-deps -format=p1689 -- clang++ -stdlib=libc++  
afx3.cppm -c -o afx3.cppm.o > afx3.cppm.o.ddi
```

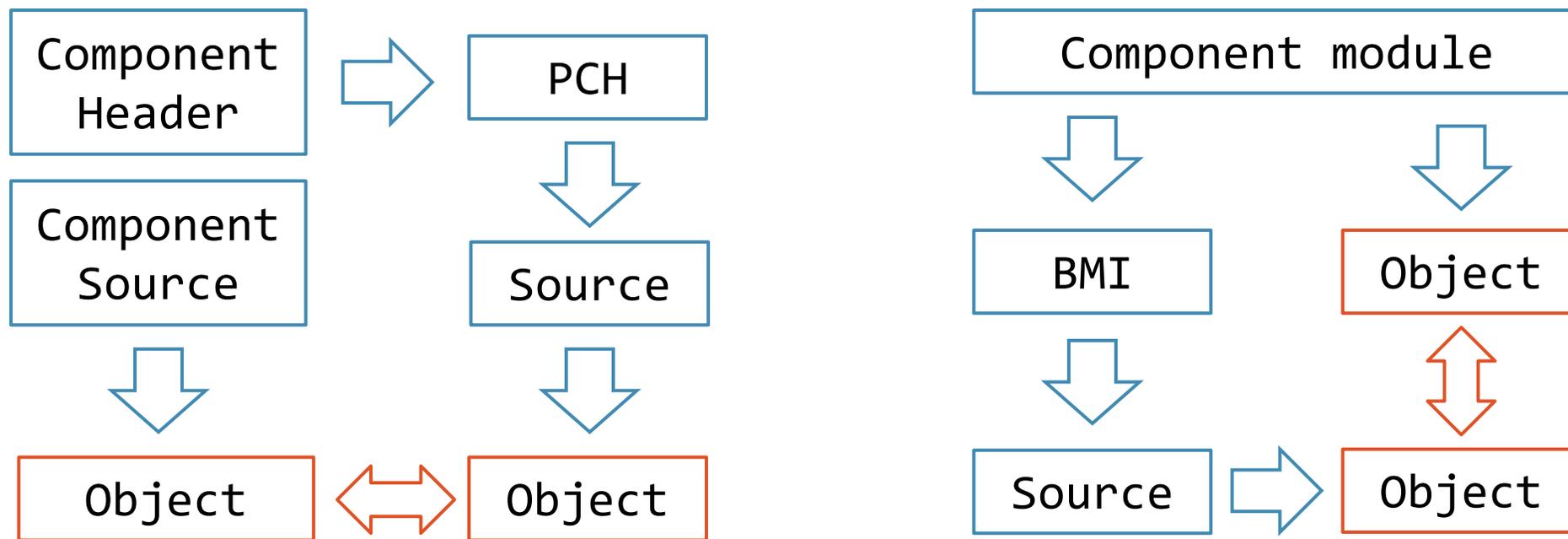
- Если препроцессорные зависимости собирались без фазы collate, то зависимости по модулям требуют специального синтаксического разбора.

Формат p1689

```
{
  "revision": 0,
  "rules": [
    {
      "primary-output": "afx3.cppm.o",
      "provides": [
        {
          "is-interface": true,
          "logical-name": "strutil",
          "source-path": "afx3.cppm"
        }
      ],
      "requires": [
        {
          "logical-name": "afx2"
        }
      ]
    }
  ]
}
```



Обращение зависимостей



Экспорт и экспорт

- Ключевое слово `export` в том числе означает две вещи:
 - Экспортную единицу модуля.
 - Внешнее связывание (`linkage`) у **экспортируемой** сущности.
- Какое тогда связывание у прочих сущностей?
- Хорошая ли идея сказать что у функции `foo` справа **внутреннее** связывание?

```
// экспортная единица
export module afx2;
// экспортируемая сущность
export int bar();
// не экспортируемая сущность
int foo();
```



```
// реализация
module afx2;
```

Важное отступление: linkage

- **external linkage** – имя может быть использовано в другой единице трансляции.
 - extern "C" linkage – гарантии по манглированию имён как в C.
 - extern "C++" linkage – гарантии по манглированию имён как в C++.
 - extern linkage имена являются субъектами ODR если не выполнен ряд условий , для их исключения оттуда.
- **internal linkage** – имя может быть использовано в других областях видимости только в той же единице трансляции.
- **no linkage** – имя не может быть использовано в других областях видимости.
- И у нас добавляется ещё одно: **module linkage**.

Дурная репутация internal linkage

```
// afx2.h
namespace _detail {
    static int foo() {
        //.....
    }
}

int bar(int* a) {
    a[0] = _detail::foo();
    // .....
}
```



```
// afx2.cppm
export module afx2;
int foo() { ..... }
export int bar(int* a) {
    a[0] = foo();
    // .....
}
```

Правила для экспорта

- Экспорт должен находиться исключительно в purview модуля.

```
// export unit of afx2  
export module afx2;
```

```
// purview of afx2  
export module afx2;  
int foo() { ..... }  
export int bar(int* a);
```

Правила для экспорта

- Экспорт должен находиться исключительно в `public` модуля.
- Экспорт не должен менять связывание имени.

```
export module afx2;
```

```
namespace {  
    export int a; // ERROR: internal linkage to external  
}
```

```
int b;
```

```
export namespace N {  
    using ::b; // ERROR: module linkage to external  
}
```

Правила для экспорта

- Экспорт должен находиться исключительно в `public` модуля.
- Экспорт не должен менять связывание имени.
- Экспорт **сам по себе не создаёт пространства имён.**

```
export module afx2;
```

```
export int n; // Это глобальная переменная с extern linkage  
             // и обычными ODR-based ограничениями.
```

Когда мясо гниловато

- Увы, самодеятельность отдельных вендоров очень сильно компрометирует идею модулей, как она заложена в стандарт.

```
export module lib1;  
export int foo() { ..... }
```

A.cc



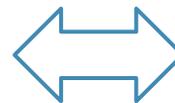
```
import lib1;  
int bar() { uses foo }
```

```
export module lib2;  
export int foo() { ..... }
```

B.cc

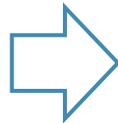


```
import lib2;  
int buz() { uses foo }
```



Большая проблема: экспорт макросов

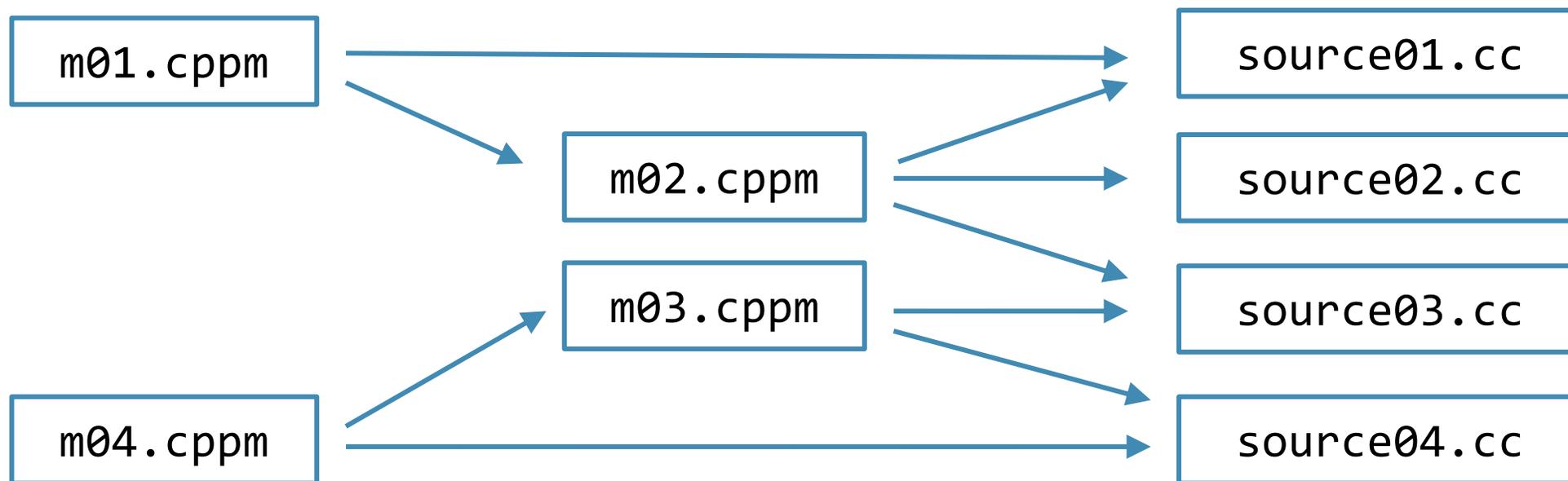
```
// afx2.h
#include <array>
#ifndef X
#define X 1
#endif
#define ARRSZ (X + 6)
```



```
// pch2.cc
#define X 2
#include "stdafx.h"
std::array<int, ARRSZ> a;
```

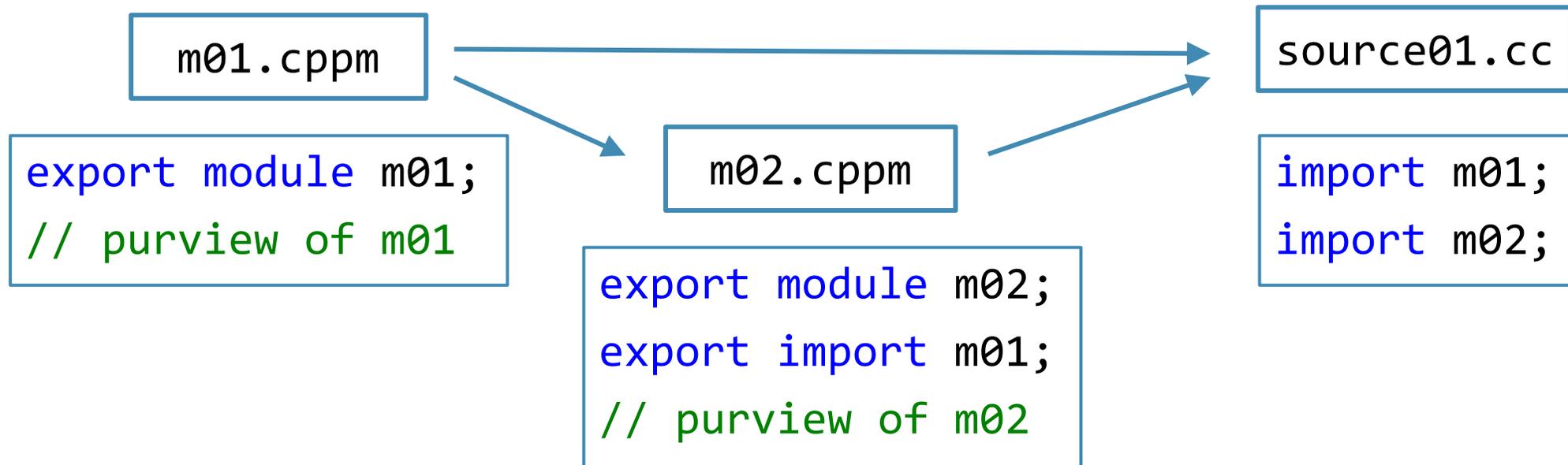
- Прекомпилированный хедер **один** и в таком качестве идёт первым, поскольку состояние препроцессора нельзя разумно сжать.
- Что если мы откажемся экспортировать состояние препроцессора?
- От чего мы тогда ещё откажемся? Что мы при этом получим?

Мы получим снова DAG



Немного крупы: транзитивный импорт

- Возможен благодаря отказу от экспорта макросостояния
- Обратите внимание: каждый модуль предкомпилирован, а стрелки это зависимости по экспорту. То есть **по ВМІ**.



Каша суховата без маслица

- Что если мы хотим улучшить наш модуль какими-то существующими хедерами?

```
#include <array>

namespace _detail {
    inline int foo() { .... }
}

int bar() {
    std::array<int, 2> a;
    a[0] = _detail::foo();
    // .....
}
```

Одна очень плохая идея

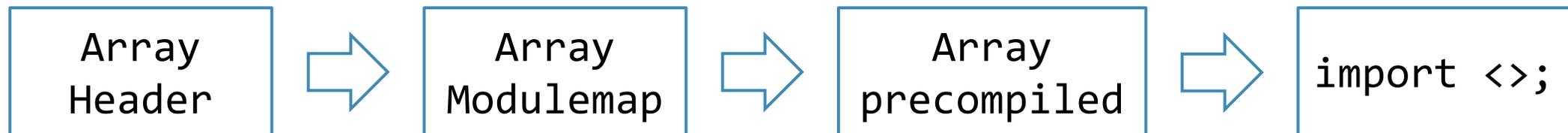
```
export module afxm;  
#include <array> // вы кинули масло куда-то не туда!
```

- Проблемы тут в том, что:
- Ни один символ из `std::array` не попадает в список экспорта, но все они обязаны оказаться в модуле.
- Если там были символы с явно заданным `extern C linkage`, это может вызвать проблемы с ABI.

Немного про importable headers

```
export module afxm;  
import <array>; // importable header
```

- В стандарт это залетело из попыток сохранить предкомпилированность стандартной библиотеки.
- Ключевое слово `import` здесь это новая директива препроцессора, которая **предкомпилирует хедер** и вставляет содержимое с внутренним связыванием.



Иногда каша подгорает

- Это не так просто заставить работать даже в clang.

```
export module helloworld;
```

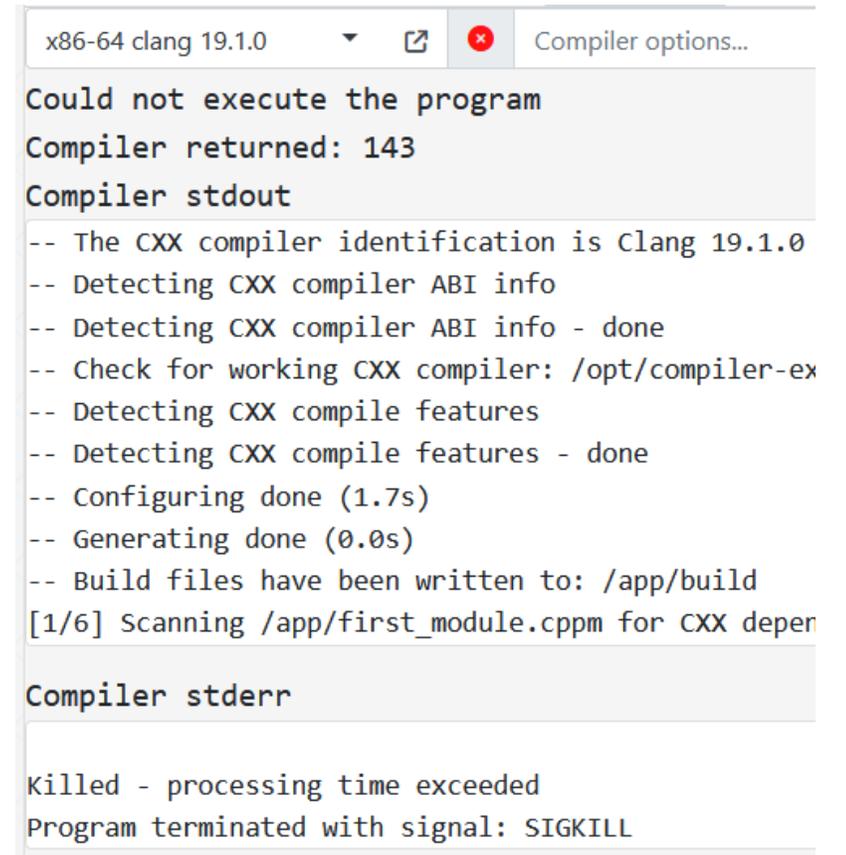
```
import <iostream>; // importable header
```

```
export void hello();
```

- Можно заставить это работать через:

```
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS}  
    -fmodules -stdlib=libc++")
```

- И это может быть долго.



The screenshot shows a terminal window titled "x86-64 clang 19.1.0" with a red error icon and a "Compiler options..." button. The main content of the terminal is a message: "Could not execute the program". Below this, it states "Compiler returned: 143" and "Compiler stdout". The stdout content is a series of diagnostic messages from Clang, including "The CXX compiler identification is Clang 19.1.0", "Detecting CXX compiler ABI info", "Check for working CXX compiler: /opt/compiler-ex", "Detecting CXX compile features", "Configuring done (1.7s)", "Generating done (0.0s)", and "Build files have been written to: /app/build". The final line of the stdout is "[1/6] Scanning /app/first_module.cppm for CXX dependen". Below the stdout, there is a section for "Compiler stderr" which contains the message "Killed - processing time exceeded" and "Program terminated with signal: SIGKILL".

```
x86-64 clang 19.1.0  Compiler options...
Could not execute the program
Compiler returned: 143
Compiler stdout
-- The CXX compiler identification is Clang 19.1.0
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Check for working CXX compiler: /opt/compiler-ex
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done (1.7s)
-- Generating done (0.0s)
-- Build files have been written to: /app/build
[1/6] Scanning /app/first_module.cppm for CXX dependen
Compiler stderr
Killed - processing time exceeded
Program terminated with signal: SIGKILL
```

Долго только в первый раз

- Модуляризованная стандартная библиотека образует кеш из модулей.

`$HOME/.cache/clang/ModuleCache`

- При первой компиляции с соответствующими флагами и макроопределениями туда кешируется вся предкомпилированная std.
- Clang поддерживает специальный язык определения отображений хедеров на модули.

```
module assert {  
    textual header "assert.h"  
    header "bits/assert-decls.h"  
    export *  
}
```

Не используйте `importable headers`

- Вы используете легаси механизм большинства компиляторов, пережиток до-стандартных модулей.
- Поскольку это директива препроцессора, она норовит затащить к вам макросостояние.
- Разница между этими строчками немыслимо огромна:

```
import mymodule; // обычный контролируемый импорт
```

```
import <mymodule>; // мрачная и непредсказуемая дыра в языке
```

- Но если это не использовать, что тогда делать?

Кусочек масла: глобальный фрагмент

- В него мы помещаем не экспортируемое легаси состояние.

```
module;  
#include <array>  
export module afxm;  
// purview
```



```
export module afxm;  
import <array>;  
// purview
```



- Теперь всё поддерживается максимально прозрачными методами языка.
- В глобальном фрагменте запрещено что бы то ни было кроме препроцессора.
- Глобальный фрагмент позволяет **иметь** в модуле макросостояние, но при этом **не распространять** его на пользователей.

Контролируемый экспорт + GMF

```
// afx2.h
#include <array>
#define SZ 15
namespace _detail {
    inline int foo() { ..... }
}
int bar() {
    std::array<int, SZ> a;
    a[0] = _detail::foo();
    // .....
}
```



```
// afx2.cppm
module;
#include <array>
#define SZ 15
export module afx2;
int foo() { ..... }
export int bar() {
    std::array<int, SZ> a;
    a[0] = foo();
    // .....
}
```

Кашу маслом не испортишь?

```
module;                                     // afx3user.cc

namespace N {
    struct S{};
    template <typename T>
    T foo(S, T) { return 1; }
}

export module afx3;

export template <typename T>
int bar(T x) {
    return foo(N::S{}, x);
}
```

Экскурсия в достижимость

- Достижимость это самое фундаментальное свойство языка.
- Имя может быть достижимо даже если оно не экспортировано.

```
// afxuser.cppm
export module A;

struct X {};
export using Y = X;

// afxuser.cc
import A;
Y y; // ok, X reachable
```

- A declaration **D** is reachable from a point **P** if
 - **D** appears prior to **P** in the same translation unit, or
 - **D is not discarded**, appears in a translation unit that is reachable from **P**, and does not appear within a private-module-fragment.
- A declaration is reachable if it is reachable from any point in the instantiation context

Отменённые определения

- Достижимость это самое фундаментальное свойство языка.
- Имя может быть достижимо даже если оно не экспортировано.
- Имя может быть достижимо даже если оно невидимо для поиска имён.
- Но увы, имя foo в точке инстанцирования не достижимо, так как вычеркнуто из этого списка (discarded).
- A declaration D in a global module fragment of a module unit is **discarded** if D is not **decl-reachable** from any declaration in the declaration-seq of the translation-unit.

Кашу маслом не испортишь?

```
module;

namespace N {
    struct S{};
    template <typename T>
    T foo(S, T) { return 1; }
}

export module afx3;

export template <typename T>
int bar(T x) {
    return foo(N::S{}, x);
}
```

```
// afx3user.cc
```

```
import afx3;
```

```
auto x = bar(1); // FAIL?
```

- Формально в точке инстанцирования достижимо имя `bar`, но не достижимо имя `foo`.
- Оно не принадлежит единице трансляции
- Увы, в clang компиляция проходит.

Недостижимость руками

- Приватный фрагмент модуля это место куда из `priview` относят недостижимые имена.

```
export module afx2;  
export struct pimpl;  
  
module : private;  
struct pimpl { int x, y; };
```

- Приватный фрагмент не принадлежит `priview` и расположен всегда в конце экспортной единицы.

- A declaration `D` is reachable from a point `P` if
 - `D` appears prior to `P` in the same translation unit, or
 - `D` is not discarded, appears in a translation unit that is reachable from `P`, and **does not appear within a private-module-fragment.**
- A declaration is reachable if it is reachable from any point in the instantiation context

Теперь немного овощей

- В начале мы рассматривали компонентный подход.
- Не станут ли наши модули лучше, если они его поддержат?

Компоненты и единицы трансляции

```
// component.h
#pragma once
#include "general.h"
```

```
// general.h
#pragma once
int foo(int x);
int bar(int x);
```

```
// general_bar.cc
#pragma once
int bar(int x);
```

```
// component.cc
// uses internal functions
#include "component.h"
#include "internals.h"
int foo(int x) {
    return baz(3) - bar(2);
}
```

```
// internals.h
#pragma once
int baz(int x);
```

```
// internals.cc
#pragma once
int baz(int x);
```

Отделяем экспорт и реализацию

- Один модуль может распространяться на несколько единиц трансляции.
- Из них только одна является его интерфейсной единицей.

```
// component.cppm
export module component;
int bar(int x);
int baz(int x);
int foo(int x) { return bar(x) - baz(x); }
```

```
// component-bar.cc
module component;
int bar(int x) { ..... }
```

```
// component-baz.cc
module component;
int baz(int x) { ..... }
```

Разбиваем интерфейсы

- Разбиение модуля позволяет комбинировать компоненты.
- У каждого разбиения есть единственная экспортная единица разбиения.
- Интересное перегруженное использование двоеточия, раньше им не злоупотребляли.

```
// component.cppm  
export module component;  
export import :general;
```

```
// component.cc  
module component;  
import :internals;
```

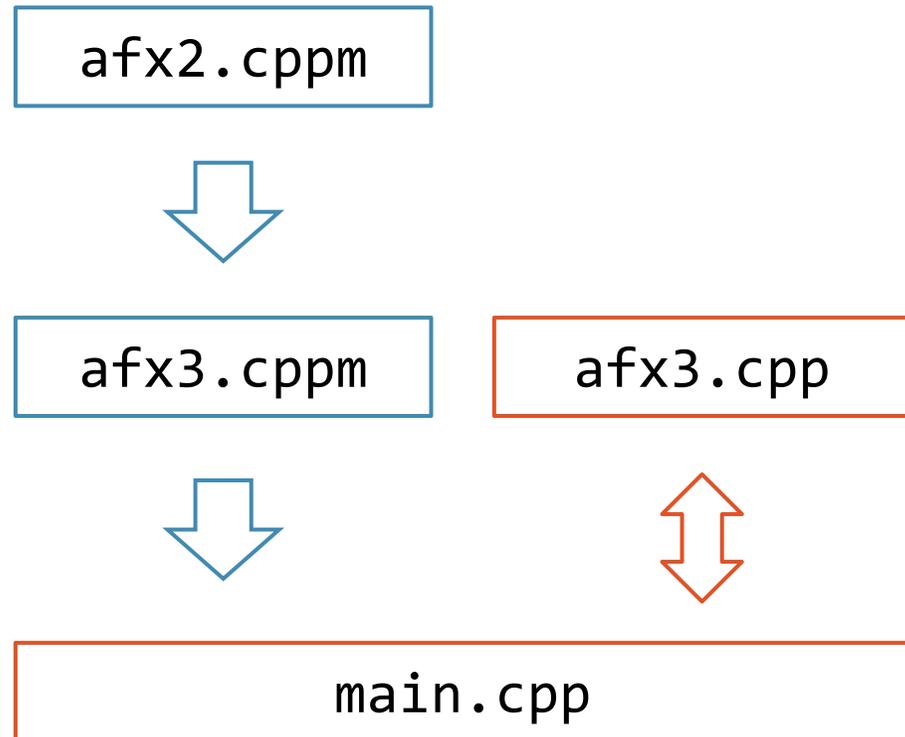
```
// internals.cppm  
export module component:internals;
```

```
// general.cppm  
export module component:general;  
import :internals;
```

Немного перца: поддержка в stake

```
target_sources(main
  PUBLIC
  FILE_SET CXX_MODULES
  FILES
    afx2.cppm
    afx3.cppm
)
```

```
target_sources(main
  PUBLIC
  main.cpp
  afx3.cpp
)
```



Обсуждение

- Теперь язык поддерживает нас в нашем стремлении к хорошему компонентному коду.
- Очень долгое время были проблемы с инструментами и системами сборки, которые теперь в принципе решены.
- Самое время попробовать модули – сейчас.
- Не ожидайте от ваших модулей, что они будут просто прекомпилированными хедерами и просто увеличат вам скорость компиляции.
- Вам придётся изменить свой подход к декомпозиции ваших исходников.

Финальный рецепт

- **Топор:** предкомпилированные хедера, скорость сборки.
- **Мяско:** контролируемый экспорт.
- **Крупка:** транзитивный импорт, отказ от импорта макросов.
- **Маслице:** глобальные фрагменты и правила reachability.
- **Овоци:** компонентный подход и его поддержка.
- **Соль и перец:** общие правила описания зависимостей и поддержка в системах сборки.
- А дальше можно выбросить топор.



ВСЕМ СПАСИБО

А теперь ваши вопросы