МОДУЛИ В C++20

Дмитрий Кожевников

МОДУЛИ В C++20

- Самая ожидаемая фича
- Самая непонятая фича
- Самая революционная фича
- Не добавляет новых возможностей для прикладного кода

ЗАЧЕМ НУЖНЫ МОДУЛИ?

СУЩЕСТВУЮЩАЯ МОДЕЛЬ КОМПИЛЯЦИИ

- Пришла из C
- Использует препроцессор для переиспользования кода
- Препроцессор не знает о семантике языка

ПРОБЛЕМА: СКОРОСТЬ КОМПИЛЯЦИИ

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

- 30 тысяч строчек (на моей машине)
- Все 30 тысяч должны быть обработаны компилятором
- Бывает миллион сточек в больших проектах
- Время сборки растет нелинейно с ростом проекта:
  - Увеличивается количество TU
  - Увеличиваетя размер TU

ПРОБЛЕМА: ПРОТЕКАЮЩИЕ ИМЕНА

Пример кода из реализации stdlib:

```
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
        __annotator.__done();
        ++this->__end_;
    }
    else
        __push_back_slow_path(__x);
}
```

ПРОБЛЕМА: ПРОТЕКАЮЩИЕ ИМЕНА

Почему это так ужасно?

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

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

ПРОБЛЕМА: ПРОТЕКАЮЩИЕ ИМЕНА

- Все имена из ранее включенных хедеров - в зоне видимости
  - Макросы
  - Классы
  - Глобальные переменные
- Особенно плохо для авторов библиотек
- Авторы стандартной библиотеки могу использовать зарезервированные имена

ПРОБЛЕМА: НАРУШЕНИЕ ODR

One Definition Rule (упрощенно):

- Только одно определение в TU
- Все определения в "программе" должны быть идентичными
- При нарушении: ill-formed, no diagnostics required

ПРОБЛЕМА: НАРУШЕНИЕ ODR

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

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

- Влияет на member layout, размер объектов, типы, ...
- Легко ошибиться случайно

ПРОБЛЕМА: СЛУЧАЙНЫЕ ЗАВИСИМОСТИ

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

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

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

МОДУЛИ ПРИЗВАНЫ УЛУЧШИТЬ:

- Время сборки
- Изоляцию имен
- Борьбу с ODR
- Отслеживание зависимостей

МОДУЛИ: HELLO, WORLD!

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

- Модуль std.io парсится отдельно
- Интерфейс модуля после парсинга хранится в бинарном файле
- Import declarations загружают интерфейс модуля без репарса

МОДУЛИ: HELLO, WORLD!

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

- Точки в имени модуля ничего не значат
- Модули не связаны с namespaces
- Информация в модуле эквивалентна исходному коду:
  - Inline функции
  - Шаблоны
  - Сообщения об ошибках

ИНФОРМАЦИЯ В МОДУЛЕ ЭКВИВАЛЕНТНА КОДУ:

- Inline функции
- Шаблоны
- Сообщения об ошибках

БИБЛИОТЕКА: HEADER-BASED

```
// 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;
  }
}

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

БИБЛИОТЕКА: НА МОДУЛЯХ

```
// 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);
}
```

ЭКСПОРТ И ИМПОРТ

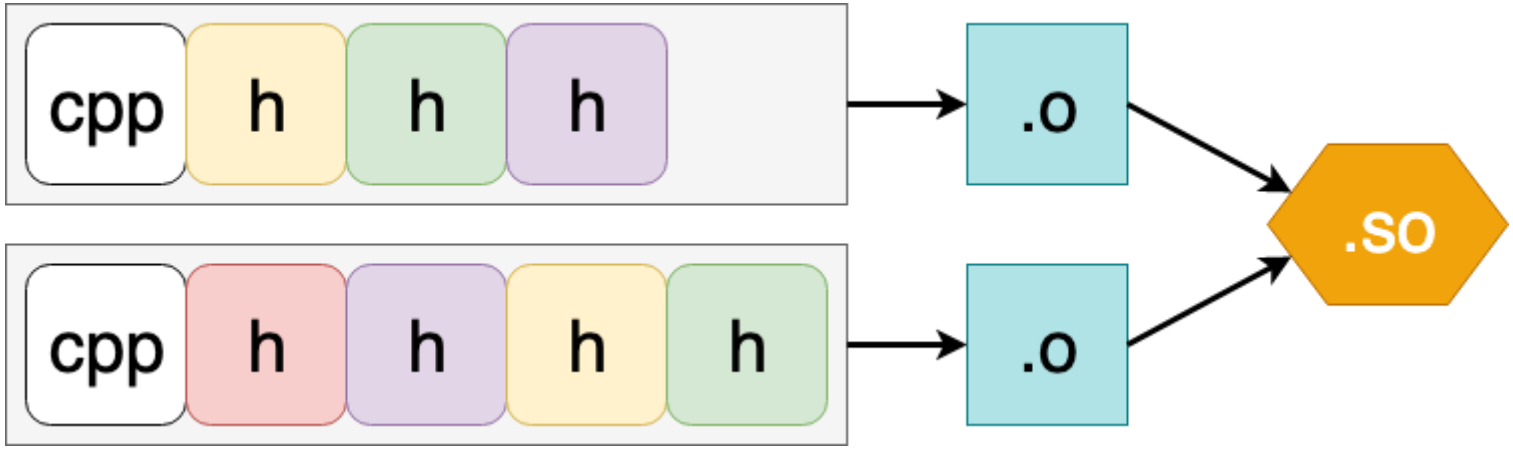
```
// module1.cppm
export module Module1;
struct S { int x; };
export namespace N {
  S foo();
}
export S bar();
```

```
// module2.cppm
export module Module2;
export import Module1;

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

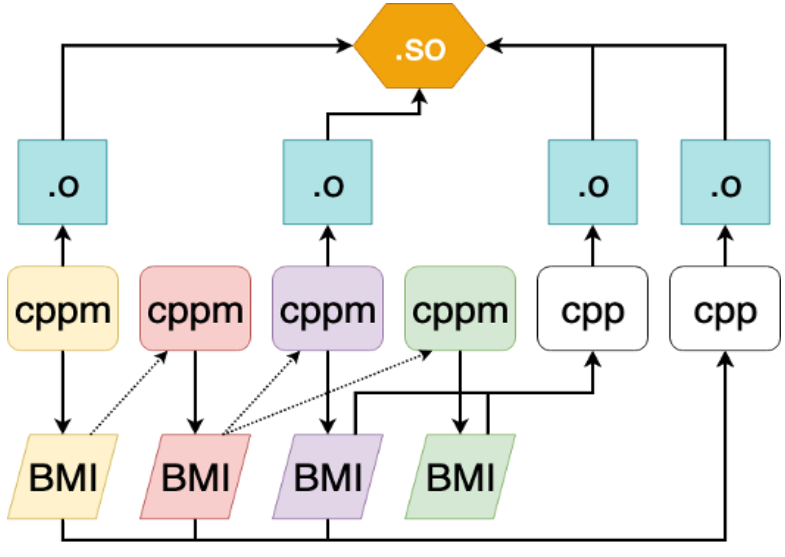
```
// main.cpp
import Module2;
int main() {
  baz(); // ok
  N::foo(); // ok
  auto s1 = bar(); // ok
  s1.x; // ok;
  S s2; // error
}
```

СУЩЕСТВУЮЩАЯ МОДЕЛЬ КОМПИЛЯЦИИ



- Хедера включаются в TU препроцессором
- Для хедеров нет отдельных правил и артефактов в системе сборки

МОДЕЛЬ КОМПИЛЯЦИИ С МОДУЛЯМИ



- cppm: исходны код Module Translation Unit
- BMI: Binary Module Interface
- Каждый модуль компилируется отдельно в BMI и (возможно) объектный файл

BMI - НЕ АРТЕФАКТЫ ДЛЯ РАСПРОСТРАНЕНИЯ!

BMI - НЕ АРТЕФАКТЫ ДЛЯ РАСПРОСТРАНЕНИЯ!

- BMI зависит от компилятора
- BMI зависит от флагов компилятора
- Распространяться должен исходный код интерфейса модуля
- BMIs можно распространять в особых ситуациях
  - Стандартная библиотека вместе с тулчейном (Visual C++, Xcode)
  - BMI для фиксированного окружения внутри компании

РАСПРОСТРАНЕНИЕ МОДУЛЬНОГО КОДА

- Исходный код — для module interface units
- Скомпилированные бинарники — для библиотек (скомпилированы из implementation units)
- Модули не помогают (и не мешают) проблеме пакетных менеджеров!

LINKAGE

Контролирует локальность символа

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

Что делать с символами из модулей?

MODULE LINKAGE

Символ виден внутри модуля

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

На практике достигается новыми правилами манглинга

МОДУЛИ И ПРЕПРОЦЕССОР

- Существующие макросы не влияют на компиляцию модуля
- Макросы из модуля не экспортируются

```
// M.cppm
export module M;
#ifdef A
  #define B
#endif
#define C
// A и B не определены

// main.cpp
#define A
import M;

// B и C не определены
```

ПЕРЕХОДНЫЙ ПУТЬ

- Включить существующий header в модуль - скорее всего, некорректно
- Переписать весь код на модули - нереалистично

HEADER UNITS

- Пометить "хорошие" хедеры как модули
- Не требует изменений исходного кода
- Все символы экспортируются
- Макросы видны после импорта!

```
import normal.module;
import <vector>
```

Все хедера стандартной библиотеки - "хорошие"

GLOBAL MODULE FRAGMENT

```
module;
#include <windows.h>
export module WinStuff;
export HRESULT DoStuff(HANDLE h);
export using MessageBox; // MessageBoxW =)
```

- Можно использовать любой хедер
- Ничего не экспортируется автоматически

- Header units: постепенная модуляризация существующей кодовой базы
- Global module fragment: новая модульная кодовая база с использованием старых хедеров

ЧТО С ВРЕМЕНЕМ КОМПИЛЯЦИИ?

ЧТО СО ВРЕМЕНЕМ КОМПИЛЯЦИИ?

- Многие ждут модули исключительно ради ускорения билдов
- Количество работы для компилятора уменьшается
- Параллельность билда ухудщается

ПАРАЛЛЕЛЬНОСТЬ БИЛДОВ

- Нужно анализировать текст модулей для определения зависимостей
- Анализ не параллелится
- Копирование BMI в распределенных билдах не бесплатное

КАКИХ УЛУЧШЕНИЙ ОЖИДАТЬ?

50%-1000% УМЕНЬШЕНИЕ ВРЕМЕНИ БИЛДА

На основе:

- Существующих реализаций
- Экспериментов
- Спекуляций

ПЛОХО РАБОТАЮТ В МОДУЛЬНЫХ БИЛДАХ:

- Небольшие простые хедера
- Много шаблонного метапрограммирования

ХОРОШО РАБОТАЮТ В МОДУЛЬНЫХ БИЛДАХ:

- Большие хедера, много inline кода
- Много метапрограммирования на constexpr

НОВЫЕ ВОЗМОЖНОСТИ ДЛЯ БИЛДОВ:

- Ускорение инкрементальной компиляции
- Параллелизм сборки одного TU

ТУЛИНГ

СЛОЖНОСТИ ДЛЯ СИСТЕМ СБОРКИ

- Где искать модули?
- В каком порядке их компилировать?

ГДЕ ИСКАТЬ МОДУЛИ?

Имя модуля написано в исходном коде:

- Не связано с именем файла с кодом
- Не связано с именем файла с BMI

ГДЕ ИСКАТЬ МОДУЛИ?

РЕШЕНИЯ:

- Договориться про стандартную схему именования (ха-ха)
- Явно передавать информацию компилятору
- Огранизовать общение компилятора и системы сборки

В КАКОМ ПОРЯДКЕ КОМПИЛИРОВАТЬ МОДУЛИ?

```
export module A

import B;

export module B
```

```
export module A

export module B

import A;
```

РЕШЕНИЯ

- Вручную указывать зависимости
- Анализировать код перед билдом
- Строить BMI на лету

IDE И СТАТИЧЕСКИЕ АНАЛИЗАТОРЫ

Есть проблемы с хедерами

- В каком контексте анализировать хедера?
- Что, если они не самодостаточные?
- Что делать с диагностиками из чужих хедеров?
- Существующие IDE используют техники оптимизаций, похожие на модули

Модули помогают с этим, но могут потребовать более тесной интеграции с компиляторами и системами сборки

ТЕКУЩИЙ СТАТУС ПОДДЕРЖКИ КОМПИЛЯТОРАМИ

- clang: -fmodules-ts
  - -fmodules это НЕ C++20 модули!
- Visual C++: /experimental:module
- GCC: cxx-modules бранч

СТАНДАРТНАЯ БИБЛИОТЕКА

- C++23
- Доступны реализации от некоторых вендоров
- Header Units в C++20

ТЕКУЩИЙ СТАТУС ПОДДЕРЖКИ СИСТЕМАМИ СБОРКИ

- build2 провел первые эксперименты
- CMake: активная работа в процессе
- Сложности с Make-based системами (если не задавать зависимости явно)

РЕЗЮМЕ

- Модули приняты в C++20
- Более аккуратный способ переиспользовать код
- Ускорение сборки (но не такое большое, как некоторые надеялись)
- Авторы компиляторов и тулинга в процессе поддержки

СПАСИБО ЗА ВНИМАНИЕ!

ВОПРОСЫ?