C++

Spring 2021

compscicenter.ru

Philipp Grabovoy

t.me/phil-grab

Lecture VIII

Modules

Notes

- `CppReference` core modules page: incomplete
- `Video`: Bryce Adelstein :: Modules are Coming
- `Alternatives`: clang modules, ...

headers in compiling model

artifacts

- source files
- translation units
- executables/libraries

actions

- sources [→ PreProcessing →]
- TUs [→ Compilation →]
- also TUs [→ Linking/Archiving/.. →]
- executables/libraries

PreProcessing

`#include`-директивы *подставляются* в исходные файлы (textual inclusion)

issues with headers

compiling

- пример:
  - `a.cpp` и `b.cpp` ← `#include<somelib.hpp>`
- ⇒ код заголовочника анализируется/компилируется несколько раз
- ⇒ можно делать параллельно

ODR violations (macros deps)

- foo.hpp:

```
struct foo {
    char* data;
#ifdef DEBUG
    char* debug_info;
#endif
};
```

- a.cpp:

```
#define DEBUG
#include "foo.hpp"
```

- b.cpp:

```
#include "foo.hpp"
```

no encapsulation

- в хедерах нет сокрытия деталей
- → `namespace details { … }`
- → `void __inner_do_no_use_func();`

order dependent

- foo.hpp:

```
struct foo { /* ... */ };
```

- bar.hpp:

```
void bar(foo s) { /* ... */ };
```

- source.cpp:

```
#include "foo.hpp"
#include "bar.hpp"
```

using modules

example

- math.ixx:

```
export module math;
export int square(int a);
```

- math.mxx:

```
module math;
int square(int a) { return a * a; }
```

- main.cpp:

```
import math;
int main() { return square(42); }
```

changes in files

- `math.hpp` → `math.ixx` — Module Interface Unit
- `math.cpp` → `math.mxx` — Module Implementation Unit
- `math.ixx` [on *precompile*] → `math.cmi` — Compiled Module Interface
  - используется *разделяемо* cpp-сорсами (не-модулями) и другими модулями (по зависимостям)
  - `math.cmi` формируется из `math.ixx` изолированно от `*.cpp`

changes in processing

- появляется прекомпиляция
- препроцессинг → работает с объявлениями из `cmi`
- компиляция:
  - определения из `.ixx` — отдельно → `.i.o`
  - `main.cpp`, `math.mxx` — *слабо изменились* (→ `main.o`, `math.m.o`)
- линковка: вместе `main.o` + `math.i.o` + `math.m.o`

pros and cons

- `.cmi` файлы разделяются между сорсами
  - их сборка: происходит единожды независимо от количества импортов
- сборка теперь с зависимостями ⇒ нет параллелизма из коробки, нужен граф и т.д.
- макросы и импорты плохо сочетаются вместе [*]:
  - `#define` + `import` не работают
  - `#define` внутри модуля наружу не работает

headers as modules [*]

- `import <foo.hpp>`, `import "foo.hpp"`
- Importable headers:
  - Большинство стандартных библиотек C++
  - Некоторые системные заголовочники
  - Заголочники, обозначенные импортибельными
- Компилятор может заменять `#include` на `import`

creating modules

basic syntax

- Interface Unit:

```
export module csc.cpp;

import csc.common;  // deps here

// some exports or ...
```

- Implementation Unit:

```
module csc.cpp;
/* ... */
```

exporting

```
export struct Foo { /* ... */ };

export {
    void f();
    struct Bar { /* ... */ };
}

export namespace some_ns { /* ...*/ }

export import homeworks;
export import cls_tasks;
```

visible vs reachable

- a.ixx:

```
export module a;

struct S { int m; };  // not exported
export S foo();
```

- main.cpp:

```
import a;

int main() {
    auto s0 = foo();  // ok, _reachable_
    s0.m = 42;

    // *not ok*, not visible for _naming_
    S s1;
}
```

pre C++20 linkage

- `external linkage`
  - global name + `extern`
- `internal linkage`
  - `static`, global `const`, anonymous namespace
- `no linkage`
  - local variables

post C++20 linkage

- `external linkage`
  - `extern` + `export`
- `module linkage` → `external`, если не модуль
  - global (no modifiers)
- `internal linkage`
  - `static`, global `const`, anonymous namespace
- `no linkage`
  - local variables

private in module

- объединяет декларацию и реализацию в одном файле
  - позволяет скрывать детали реализации от использования

```
export module foo;

export struct pimpl;          // declared

module :private;
struct pimpl{ /* ... */; }  // not _reachable_ from oute
```

modules and macros

- флаги компиляции `-D...` — работают
- `#define` перед `import` — не работают
- внутри модуля можно делать `#include` "под" макросом:

```
module;

#define _XOPEN_SOURCE // For `readlink`
#include <unistd.h>;

export module a;
/* ... */
```

modules in production

for developers

- переезд headers → modules не бесплатен
  - отказ от некоторых фичей препроцессора
  - разворачивание циклических зависимостей
  - ....

compilers

- надо расширяться по фичам, характерным для систем сборки:
  - modules' lookup
  - caching (per configuration)
  - build graph

developers' tools

- взаимдействиями с модулями (садджест/навигация по коду):
  - через новые (специальные) интерфейсы компилятора
  - C парсер (не C PreProcessor)