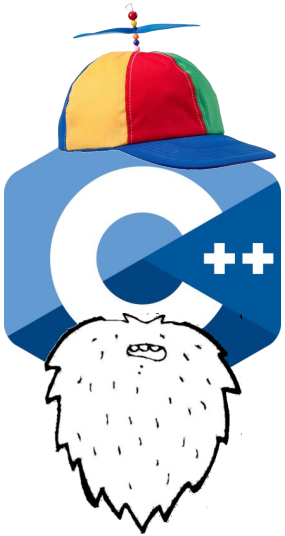
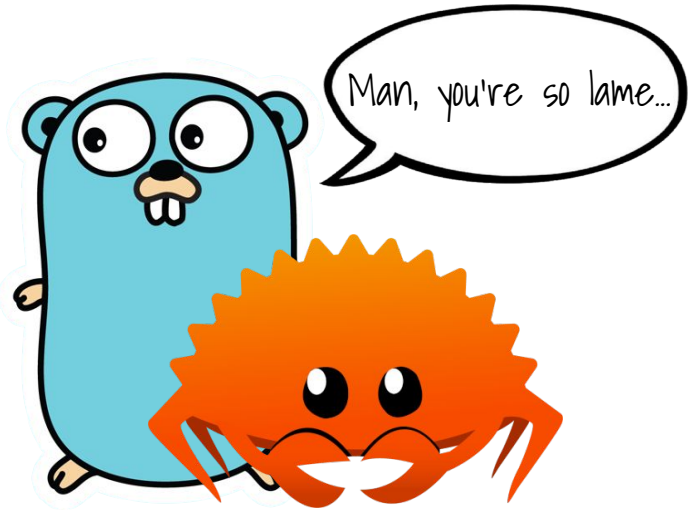
C++ Modules

conservative, but not entirely bad





Mundale Reality

Source files, header files, text inclusion and separate compilation:

- slow to compile
- ODR violations
- lack of encapsulation
- cyclic dependencies
- order dependencies

So, do modules solve our problems?
Are they *good*?

Yes.

Yes..

Yes...

Yes… and no.

Yes… and no. But yes.

State of the Art

Modules provide:

- modular interface
- proper encapsulation
- “top-down” and “bottom-up” isolation
- order independency of imports

And, as a *possible byproduct* of such design, shorter build times.

Basics

```
// main.cpp
import foo;
import bar;

int main() {
    TypeFromFoo x{};
    function_from_bar();
    internal_function_from_bar(); // error!
}

// foo.mi
export module foo;
export class TypeFromFoo {...};

// bar.mi
export module bar;
export void function_from_bar();
export inline void inline_function_from_bar() {...}
void internal_function_from_bar() {...}
```

```
// foo.m
module foo;
void TypeFromFoo::TypeFromFoo() {...}

// bar.m
module bar;
void function_from_bar() {...}
```

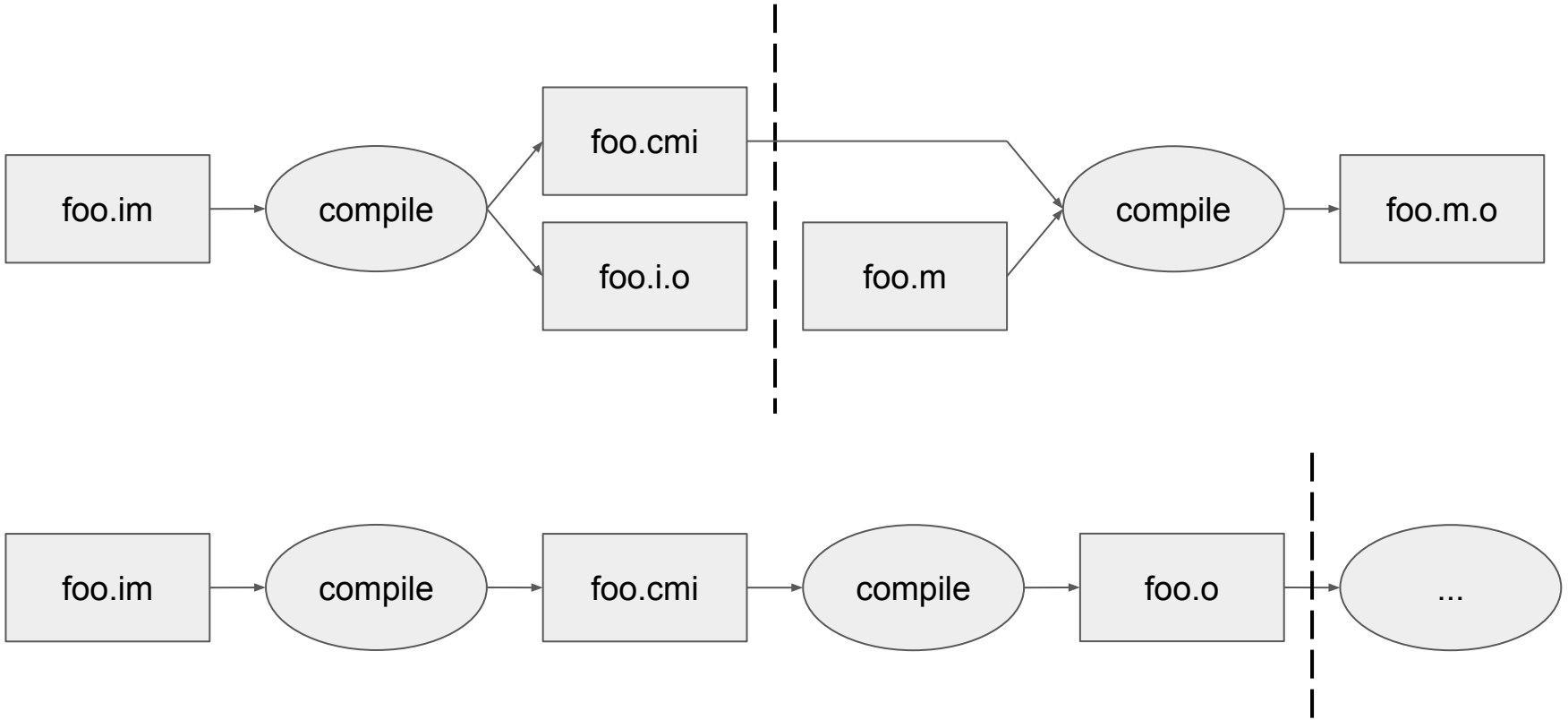
Translation Units

- Non-Modular Unit
  no module-declaration

- Module Interface Unit
  `export module <module-name>;`
  1 Module Interface Unit per module

- Module Implementation Unit
  `module <module-name>;`
  `0 or 1 Module Implementation Unit per module`

Compilation Model



Problems with Modules Compilation Model

- order dependencies
- modules lookup

More or less everything regarding build process is *implementation-defined*. And there are no consensus on how to best implement it yet.

But TR is promised post-C++20 release.

All About Exports

```
// foo.mi
export module foo;

import bar;

// export declarations
export const int x = 10;
export void f() {...}
export struct S;

// when using export declaration
// to export templates, 'export'
// should be placed before 'template'
export template<typename T>
void ft(T t) {...}

// export block
export {
    int y;
    void g();
    class D {...};
}

// ...
```

```
// ...

// changing linkage of previously
// defined names is an error
void h() {...}
export void h(); // error!

// (re-)definition of names owned by
// other module is an error
void bar_f() {...} // error!
class BarC {...}; // error!

// exporting using-declarations
// is fine (cause they don't define
// anything, only introduce an alias)
// as long as name being aliased
// have external linkage
export using BarC;
export using BarS; // error!
using FooS = BarS; // error!

// re-exporting other modules
export import bar;
```

```
// bar.mi
export module bar;

export void bar_f() {...}
export class BarC {...};
struct BarS {...};
```

Modules and Namespaces

```
// foo.mi
export module foo;

namespace foo {
    export void f1(); // external linkage
    void f2();        // module linkage
    static void f3(); // internal linkage
}

export namespace foo {
    void g1();
    static void g2(); // error!
}

namespace foo {
    class MapImpl {
        ...
    };

    export class Map {
    public:
        ...
    private:
        ...
        MapImpl map_impl;
    };
}
```

```
// main.cpp
import foo;

int main() {
    foo::f1();
    foo::f2(); // error!
    foo::f3(); // error!
    foo::g1();

    foo::Map map{};
    foo::MapImpl map_impl{}; // error!
}
```

Modules and Static Linkage

```
// foo.mi

const int x = 10;
static int y = 20;

static void f() {...};

namespace {
    int z = 30;
    void g() {...};
    class C {...};
}
```

Visibility and Reachability

```
// foo.mi
export module foo;

// visible: in scope, can be named
// reachable: in scope, can't be directly named

// reachable
class C {
public:
    int x, y, z;
    void f(int whatever);
    ...
};

// not visible and not reachable
namespace {
    struct S {...};
}

// visible
// C is reachable, so this is allowed
export C wait_what() {
    return C{};
}

// S is not reachable, so this is not
// or, maybe, it is, but it should have not
// don't do this
export S you_cant_do_this() {
    return S{};
}
```

```
// main.cpp
import foo;

int main() {
    auto c1 = wait_what();
    c.f(c.x + c.y + c.z);

    C c2{}; // error!

    // you can do this, but you should not
    decltype(wait_what()) c3;
}
```

Module Private Fragments

```
// foo.mi
export module foo;

export class pimpl;

// foo.m
module foo;

class pimpl {
    ...
};
```

Module Private Fragments

```
// foo.mi
export module foo;

export class pimpl;

// foo.m
module foo;

class pimpl {
    ...
};
```

```
// foo.mi
export module foo;

export class pimpl;

module : private;

class pimpl {
    ...
};
```

Splitting Modules: Module Partitions

```
// foo.mi
export module foo;

export import :a;
export import :b;
export import :c;

export void f();

// foo-a.mi
export module foo:a;

import foo;

export void a1();

export void a2() {
    return f();
}

// foo-b.mi
export module foo:b;
export void b();
```

```
// foo.m
module foo;

void f() {...}
void a1() {...}

// foo-b.m
module foo:b;
void b() {...}

// foo-c.m
module foo:c;
void c() {...}
```

Turning Headers into Modules: Header Units

```
// main.cpp
import <iostream>;
import "whatever.hpp";

int main() {
    std::cout << f() << std::endl;
}

// whatever.hpp
import <string>;
function std::string f() {...}
```

Mixing Modules and Headers: Global Module Fragment

```
// foo.mi
module ;

#undef NDEBUG
#include <cassert>

#define WHATEVER
#include "whatever.hpp"

export module foo;

export int f(int i) {
    assert(i);
    return QWERTY;
}

// whatever.hpp
#ifdef WHATEVER
#define QWERTY 10
#endif
```

How Can I Try Modules Right Now?

- GCC “modules” branch: svn://gcc.gnu.org/svn/gcc/branches/c++-modules
- Clang 8.0 have -fmodules-ts flag, but everything is completely undocumented
- Clang 9.0 (currently in development) have tests for modules, so one can infer how they could be used

C++ Modules: GCC

- Modules support should be enabled with -fmodules-ts
- GCC auto-generates compiled module interface, when you compile module interface units
- Compile your module interfaces before you compile your module implementations

C++ Modules: Clang *(UNTESTED)*

- Modules support should be enabled with -fmodules-ts
- Clang requires you to explicitly compile your module interfaces:
  `clang++ -fmodules-ts` ***`-emit-module-inteface`*** `<module-interface-unit> -o <compiled-module-interface>`
- Clang requires you to explicitly specify all required compiled module interface files, when building module implementation or non-module units:
  `clang++ -fmodules-ts` ***`-fmodule-file`***`=<compiled-module-interface> <translation-unit>`

Questions?