WHY MODULES?

It's not about build time.

WHAT ARE MODULES FOR?

Modules are about controlling visibility and access to names and definitions.

Supporting "large-scale" software development.

WHAT WE'RE GOING TO TALK ABOUT

I will show how to use various features of Modules and the kinds of module units to use to provide access to the features of your library, while hiding the details you don't want clients to depend on.

I will also cover some of the limitations and how clients may still end up depending on your details in ways that constrain your ability to maintain ABI compatibility.

THE PROBLEM

`#include` is awful.

MACROS are worse.

HYRUM'S LAW

With a sufficient number of users of an API,
it does not matter what you promise in the contract:
all observable behaviors of your system
will be depended on by somebody.

Hyrum Wright

https://www.hyrumslaw.com/

OBLIGATORY XKCD

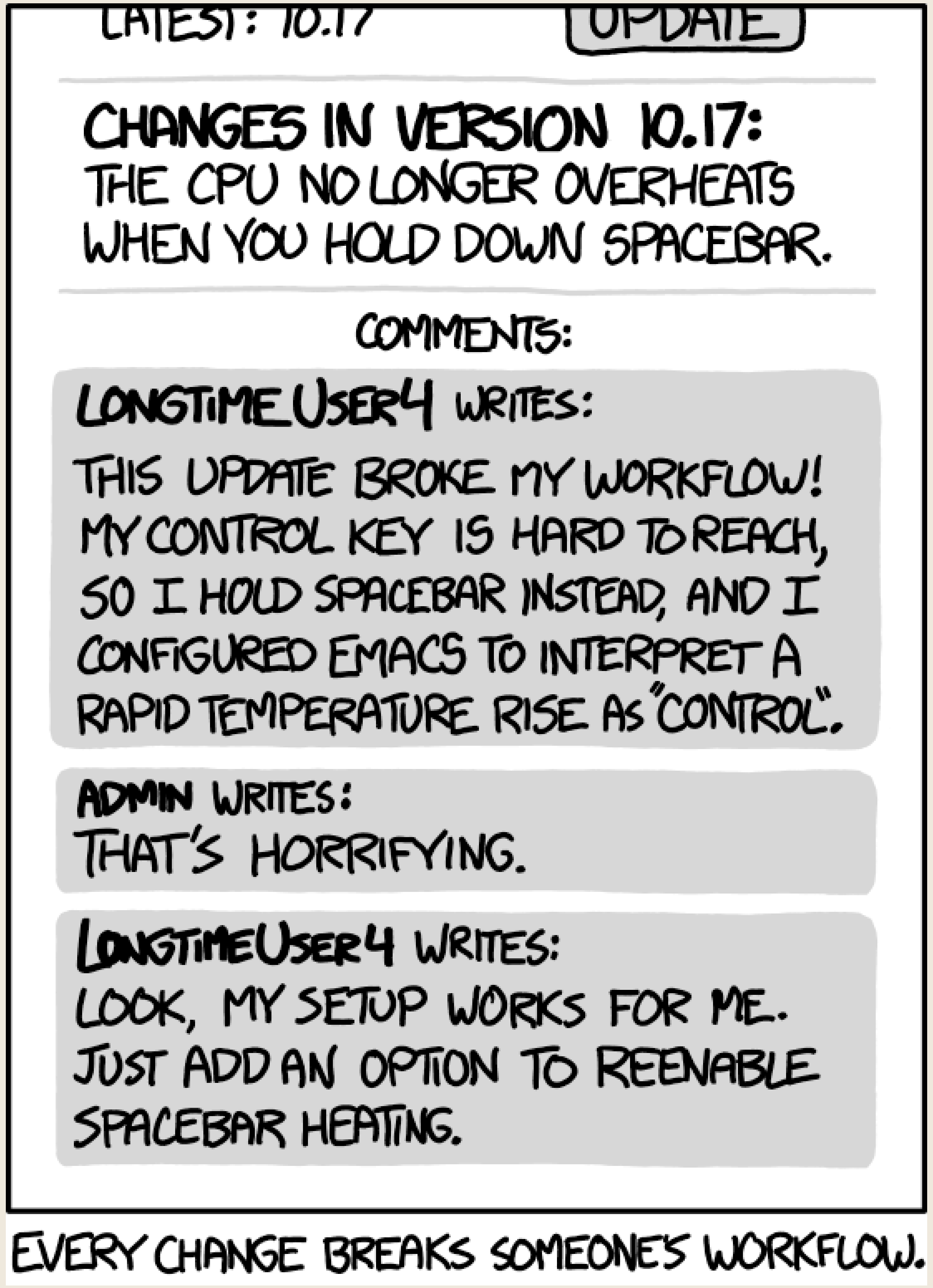
LATEST: 10.17
UPDATE
CHANGES IN VERSION 10.17:
THE CPU NO LONGER OVERHEATS WHEN YOU HOLD DOWN SPACEBAR.
COMMENTS:
LONGTIMEUSER4 WRITES:
THIS UPDATE BROKE MY WORKFLOW! MY CONTROL KEY IS HARD TO REACH, SO I HOLD SPACEBAR INSTEAD, AND I CONFIGURED EMACS TO INTERPRET A RAPID TEMPERATURE RISE AS "CONTROL".
ADMIN WRITES:
THAT'S HORRIFYING.
LONGTIMEUSER4 WRITES:
LOOK, MY SETUP WORKS FOR ME. JUST ADD AN OPTION TO REENABLE SPACEBAR HEATING.
EVERY CHANGE BREAKS SOMEONE'S WORKFLOW.

Figure 1: Workflow: https://xkcd.com/1172/

BREAKING THE BUILD

TODAY IS A BAD DAY

```cpp
struct Days {
    inline static constexpr int Sunday    = 0;
    inline static constexpr int Monday    = 1;
    inline static constexpr int Tuesday   = 2;
    inline static constexpr int Wednesday = 3;
    inline static constexpr int Thursday  = 4;
    inline static constexpr int Friday    = 5;
    inline static constexpr int Saturday  = 6;

};
```

```
example.cpp
<source>(7): error C2061: syntax error: identifier 'TUESDAY'
<source>(7): error C2258: illegal pure syntax, must be '= 0'
<source>(7): error C2253: 'Days::lookup_day': pure specifier or abstract override specifier only
                          allowed on virtual function
Compiler returned: 2
```

ELSEWHERE

```
const int TUESDAY = 2;
#define Tuesday lookup_day(TUESDAY)
```

HOW DID I BREAK THE BUILD?

```
--- a/dict.hpp  2024-07-08 11:26:20.346434057 -0400
+++ b/dict.hpp  2024-07-08 11:25:56.461562259 -0400
@@ -2,13 +2,13 @@
 #ifndef INCLUDED_MAP_EXAMPLE
 #define INCLUDED_MAP_EXAMPLE

-#include <map>
+#include <unordered_map>

 namespace example {

 class mine {
   private:
-    std::map<std::string, int> map_;
+    std::unordered_map<std::string, int> map_;

   public:
     /// More stuff
```

ELSEWHERE

```
/// snip
#include <dict.hpp>

/// snip

std::map<std::string, int>::iterator lookupThing(std::string const& key);
```

WHY?

```diff
--- a/detail/view.hpp   2024-07-08 11:42:52.542003117 -0400
+++ b/detail/view.hpp   2024-07-08 11:42:35.534632242 -0400
@@ -2,11 +2,12 @@
 #ifndef INCLUDED_DETAIL_VIEW
 #define INCLUDED_DETAIL_VIEW

+#include <cstddef>
 namespace detail {

 struct view {
     char* begin_;
-    char* end_;
+    std::ptrdiff_t distance_;
 };

 } // namespace detail
```

ELSEWHERE - IN CLIENT CODE

```
std::cout << "begin:" << (void*)v.begin_
          << "\tend:" << (void*)v.end_ << '\n';
```

ACCIDENTAL COMPLEXITY

Our code is coupled because of accidental information leakage.

Well-intentioned programmers trying to get their jobs done will use whatever is available to them.

Some days, *you* are that programmer.

THIS IS THE PROBLEM MODULES TRIES TO SOLVE

WHAT IF IT COULD ALSO MAKE BUILDING FASTER?

PROBLEM STATEMENT

The lack of direct language support for componentization of C++ libraries and programs, combined with increasing use of templates, has led to serious impediments to compile-time scalability, and programmer productivity. It is the source of lackluster build performance and poor integration with cloud and distributed build systems. Furthermore, the heavy-reliance on header file inclusion (i.e., copy-and-paste from compilers perspective) and macros stifle flowering of C++ developer tools in increasingly semantics-aware development environments.

A Module System for C++ (Revision 4)

Gabriel Dos Reis Mark Hall Gor Nishanov

2016-02-15

https://wg21.link/p0142

THE GOALS

> 1. *componentization;*
> 2. *isolation from macros;*
> 3. *scalable build;*
> 4. *support for modern semantics-aware developer tools.*
>
> *Furthermore, the proposal reduces opportunities for violations of the One Definition Rule (ODR) and increases practical type-safe linking.*

https://wg21.link/p0142

WE WANT WHAT MODERN LANGUAGES HAVE

- Don't Repeat Yourself
- Increase Cohesion
- Decrease Coupling

MODULE SYNTAX AND SEMANTICS

MODULE UNITS

All parts of modules are distinct translation units.

They are compiled separately.

- Module Unit
- Module Interface Unit
- Module Implementation Unit
- Module Partition
- Primary Module Interface Unit

MODULE UNIT

A translation unit that has a module declaration.

MODULE INTERFACE UNIT

A *module unit* that begins with `export`

The interface is what other translation units can see.

MODULE IMPLEMENTATION UNIT

Any *module unit* that isn't an interface unit.

MODULE PARTITION

A *module unit* that has a *module partition* in its declaration.

A *module partition* can only be imported within the module.

PRIMARY MODULE INTERFACE UNIT

The one *module unit* for a module that exports everything visible in the interface.

EXAMPLE

Primary Interface Unit

```
export module A;
export import :Foo;
export int baz();
```

Module partition A:Foo which is an interface unit

```
export module A:Foo;
import :Internals;
export int foo() { return 2 * (bar() + 1); }
```

Module partition A:Internals which is not part of the interface of A

```
module A:Internals;
int bar();
```

Module implementation unit

```
module A;
import :Internals;
int bar() { return baz() - 10; }
int baz() { return 30; }
```

MODULE "PURVIEWS"

Everything from the *module declaration* to the end of the translation unit.

The text before the *module declaration* is not within the purview of the module.

EXPORT

Export is how we make declarations, names and definitions, available to be imported by translation units not in the module.

- `export` must be in the *purview* of a module
- You can't `export` things with internal linkage

WORKS

```
export int f();                 // OK
export namespace N { }          // OK
export using namespace N;       // OK

struct S;
export using T = S;             // OK, exports name T denoting type S
```

DOES NOT WORK

```
export namespace {}               // error: namespace has internal linkage
namespace {
  export int a2;                  // error: export of name with internal linkage
}
export static int b;              // error: b explicitly declared static
```

SURPRISING THINGS THAT WORK

```cpp
namespace {
  struct S { };
}
export void f(S);                // OK
struct T { };
export T id(T);                  // OK
```

This means if you can produce an object of type S or T, you can call the function.

You can't name them or construct them yourself.

IMPORT

Import is how a translation unit gets access to the declarations that a module exports.

Only `import` a module interface that was `export` -ed.

GLOBAL MODULE FRAGMENT

Between `module;` and the module declaration.

Can only have preprocessor directives.

Not attached to the module, but may be reachable if used.

```
module;
#include "foo.h"
export module M;
```

PRIVATE MODULE FRAGMENT

To support single translation unit modules, the *private module fragment* is unable to affect other translation units.

It can provide definitions of things used within a module.

```
static void fn_s();
export struct X;
export void g(X *x) {
  fn_s();                         // OK, call to static function in same translation unit
}
export X *factory();              // OK

module :private;
struct X {};                      // definition not reachable from importers of A
X *factory() {
  return new X ();
}
```

REACHABILITY

You can only use declarations that are *reachable*. A declaration is *reachable* at a point if:

- It appears before the point in the same translation unit
- It is in a reachable translation unit and is not in the PMF.

A translation unit is reachable from a point if there is an interface dependency on it or a transitive dependency.

You might not be able to use the name, but the type is usable by the compiler.

EXAMPLES

Module Interface Partition

```
export module M:A;
export struct B;
```

Module Implementation Partition

```
module M:B;
struct B {
  operator int();
};
```

Module Implementation Partition

```
module M:C;
import :A;
B b1;                                // error: no reachable definition of struct B
```

Primary Module Interface

```
export module M;
export import :A;
import :B;
B b2;
export void f(B b = B());
```

Plain Old Source

```
import M;
B b3;                                // error: no reachable definition of struct B
void g() { f(); }                    // error: no reachable definition of struct B
```

ORGANIZING YOUR MODULE

DEPENDENCY CYCLES ARE FORBIDDEN

This is explicit – but also, how else would it work?

Bloomberg Component rules (a.k.a. Lakosian rules) give us this already!

FORWARD DECLARATION

You can't forward declare a type in a different module.

Cycle breaking must be within modules.

SINGLE FILE UNIT

There is some support for the equivalent of a header only library.

It may still need to produce an object file that you link.

Your project may need to build it.

MODULE PARTITIONS

Partitions must be acyclic, too.

Useful for separating parts of a library internally.

IMPLEMENTATION PARTITIONS

An *Implementation Partition*

- Is very much like normal source files for a library.
- Can not contribute to interface.
- Has access to all declarations from the primary module interface.

`export import`: SUB-MODULES

You can re-export an entire module from your module.

Units that `import` your module have access just as if they had themselves.

Can partially hide the internal structure of a module.

PLANNING AHEAD

DO YOU CARE ABOUT ABI OR API?

ABI stability means some source code reorganizations will change ABI. You can move between partitions, but not modules.

Module attachment is not visible in the API.

Bloomberg cares about API stability.

We always rebuild and link consistently, so we are only affected by API changes.

`inline` MEANS INLINE

The `inline` keyword finally means something about inlining. Functions must be marked `inline` to have their bodies exported, although this is more complicated for template instantiations in importers.

MODULE ATTACHMENT AND MANGLING

```
module A:B;

export struct foo {
    int i_;
    foo(int i) : i_(i) {
    }
};
```

```
foo@A::foo(int) [base object constructor]:         # @foo@A::foo(int) [base object constructor]
        mov     dword ptr [rdi], esi
        ret
initializer for module A:B:                         # @initializer for module A:B
        ret
```

Mangled:

```
_ZNW1A3fooC2Ei:                                    # @_ZNW1A3fooC2Ei
        movl    %esi, (%rdi)
        retq
_ZGIW1AWP1B:                                       # @_ZGIW1AWP1B
        retq
```

TESTING MODULES

PUBLIC INTERFACE

If you can test via the public interface, that is the most straightforward, and will probably make your clients happiest.

TEST IMPLEMENTATION UNITS

It is possible to write implementation units that have access to the internals of a module; write tests there to link into your test driver.

Module implementations are not closed, similar to namespace.

It is not a security feature. But they can't change your interface.

THANK YOU!

QUESTIONS?

THANKS AGAIN!