Clang Modules at Scale

Ian Anderson & Michael Spencer

LLVM Dev Meeting 2024

Agenda

10 Years of Modularizing Apple’s Platforms

Avoiding Module Pitfalls

Using the Preprocessor with Modules

Building Modules

What are Clang Modules?

Importable interfaces - similar to modules in Swift, C++, Python, etc

Not the same thing as C++20 modules

Designed for Swift to interact with existing C based code

Defined in module map files, comprised of headers

Built as independent translation units

10 Years of Modularizing Apple's Platforms

Modularization History

Original modules were made alongside Swift 1.0

- Most libraries got their own module
- Made big modules for usr/include, clang headers, libc++
- Made compiler kludges to shortcut the header/module map work

Library owners took possession of their modules

Early Swift adopters began using modules

Lessons Learned

Modules can be unintuitive, and have easily hit pitfalls

Header contents and uses are widely varied

Build performance is not a given

Avoiding Module Pitfalls

Module Basics

Modules are precompiled and reused

Modules build as independent translation units

`#include` statements are translated to module imports

Modules do not inherit the preprocessor environment from the includer

In Swift, the module name is part of the identifier for a declaration

Modularize Headers Bottom-Up

Modular headers can only include other modular headers

Non-modular includes cause a wide variety of confusing bugs

- `#include` doesn’t seem to do anything
- Type redeclaration errors
- Incompatible type errors
- Other seemingly nonsensical errors

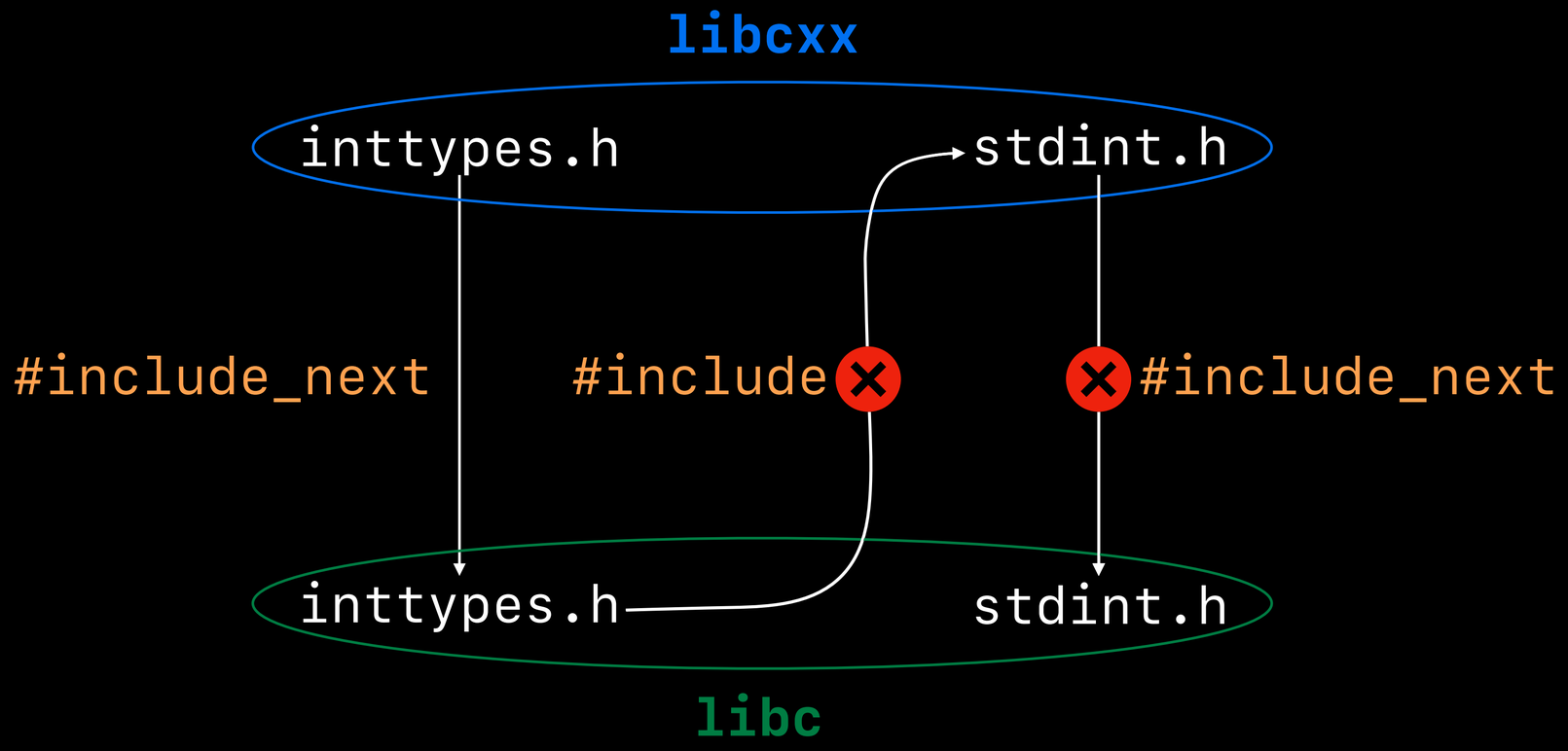
Non-modular Headers

```
// module.modulemap
module AModule [system] {
  module First {
    header "first.h"
    export *
  }
  module Second {
    header "second.h" // includes non_modular.h
    export *
  }
}

// non_modular.h
typedef int non_modular_t;



// unsuspecting_victim.c
#include <first.h>
#include <non_modular.h>
non_modular_t x; // ❌ unknown type'non_modular_t'
```

Modules Must Be Acyclic

Particularly tricky with C Standard Library headers

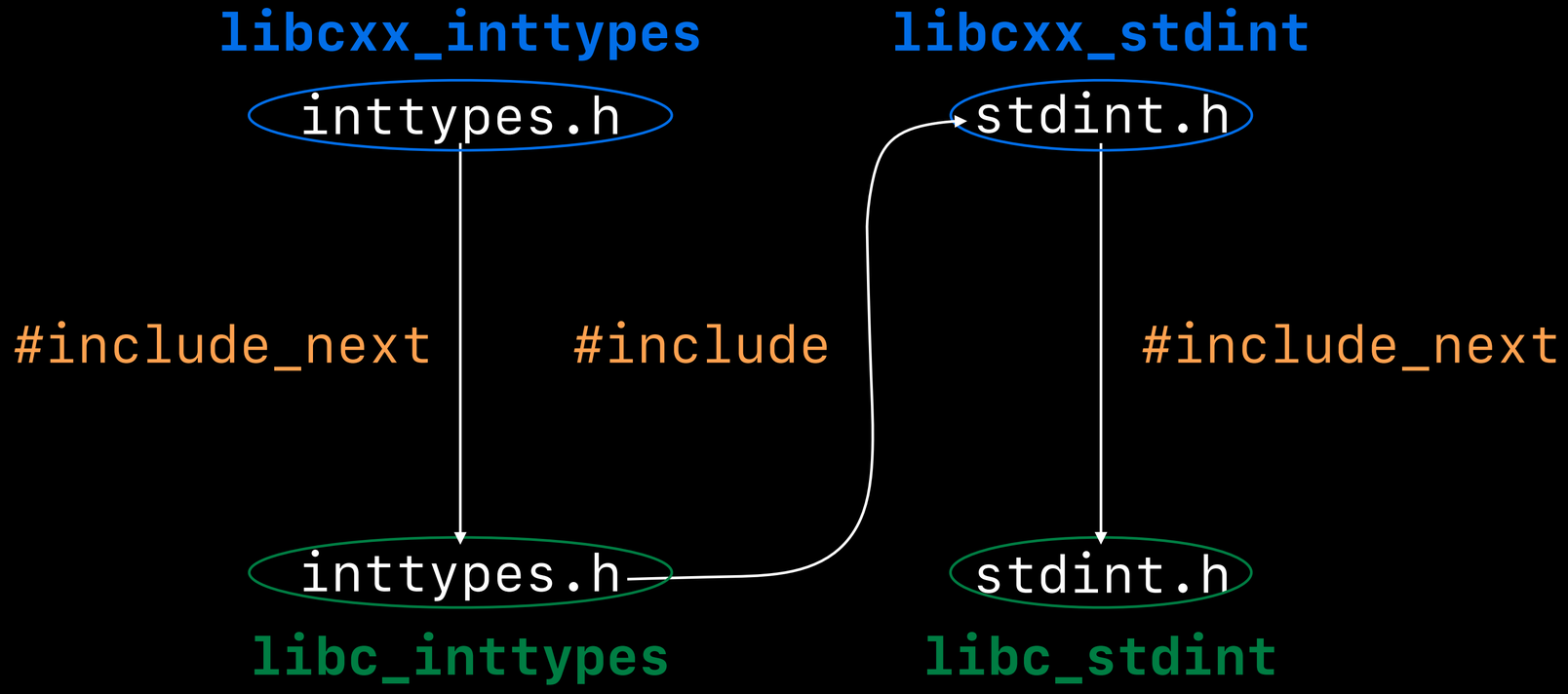
What looks like a simple #include goes up and down layers



Modules Must Be Acyclic

Originally worked around with compiler and module kludges

Extremely difficult to retroactively fix in this year’s releases



Make Sure Expected Macros Are Defined

👻 Modules build as independent translation units 👻

Headers cannot rely on the includer's preprocessor environment

User provided macros must be passed on the command line

```
// cool_module_header.h
#if EQUIP_SUNGLASSES
void be_cool(void);
#endif
```

```
// client_code.c
#define EQUIP_SUNGLASSES 1
#include <cool_header.h>
be_cool(); // ❌ use of undeclared function
```

```
// client_code.c
// clang -DEQUIP_SUNGLASSES=1
#include <cool_header.h>
be_cool(); // 😎
```

One Definition Rule

There can be only one definition of a macro, type, function, etc.

`#ifndef` would normally get around this requirement, however...

🎃 Modules build as independent translation units 🎃

```
// ubrk.h
#ifndef UBRK_TYPEDEF_UBREAK_ITERATOR
#define UBRK_TYPEDEF_UBREAK_ITERATOR
typedef struct UBreakIterator UBreakIterator;
#endif

// ustring.h
#ifndef UBRK_TYPEDEF_UBREAK_ITERATOR
#define UBRK_TYPEDEF_UBREAK_ITERATOR
typedef struct UBreakIterator UBreakIterator;
#endif
```

#undef can’t rewrite headers

#undef is sometimes used to fix other headers

<float.h> and <limits.h> in clang do this, but...

🧟 Modules build as independent translation units 🧟

```
#include_next <limits.h>

/* Many system headers try to "help us out" by defining these.
   No really, we know how big each datatype is. */
#undef  INT_MAX // Doesn't remove the define from the system module
#define INT_MAX __INT_MAX__ // ❌ 'INT_MAX' was redeclared
```

extern "C" {…} can’t rewrite headers

Like `#undef`, `extern "C"` is often used to fix other headers

🤷 Modules build as independent translation units 🤷

```
// bad_header.h
typedef int a_type_t;
// ⬇️ needs to be `extern "C"`
extern a_type_t get_a_type(int arg);
```

```
// well_intentioned_header.h
extern "C" { // ⬇️ doesn't get affected
#include <bad_header.h>
#include <stdatomic.h>
void print_a_type(a_type_t arg);
}
```

extern "C" {...} can’t rewrite headers

Treated as an error when building modules in C++ mode

Originally worked around with a module kludge

Using the Preprocessor with Modules

Textual Headers

Headers can be marked `textual` in a module map

Such headers do not build with their module

Including a textual header does not translate to a module import

Textual headers do NOT build as independent translation units

Dangers of Textual Headers

Similar to non-modular headers

Must not allow declarations to exist in multiple modules

<assert.h> cannot be used by modular headers

X Macro Generators

```
// LangOptions.def
LANGOPT(Modules, 1, 0, "modules semantics")
LANGOPT(CPlusPlusModules, 1, 0, "C++ modules syntax")
LANGOPT(BuiltinHeadersInSystemModules, 1, 0, "builtin headers ...")
```

```
// LangOptions.h
class LangOptions {
public:
#define LANGOPT(Name, Bits, DefaultValue, Description) \
  unsigned Name : Bits;
#include "LangOptions.def"
};
```

Private Implementation Headers

Some headers are split up for length

Others are split up for parallel implementation (e.g. arm/x86)

Such headers must have a single includer, and can be marked `private textual`

Split Interesting Headers Into Textual and Modular Parts

GCC style <stddef.h> uses `__need` macros to support partial inclusion

Top level `textual` header to handle the preprocessor environment

Normal modular header to provide declarations

Example: stddef.h

```
// stddef.h
#if defined(__need_ptrdiff_t)
#include <__stddef_ptrdiff_t.h>
#undef __need_ptrdiff_t
#endif

#if defined(__need_size_t)
#include <__stddef_size_t.h>
#undef __need_size_t
#endif
```

```
// module.modulemap
module _Builtin_stddef [system] {
  textual header "stddef.h"

  explicit module ptrdiff_t {
    header "__stddef_ptrdiff_t.h"
    export *
  }
  explicit module size_t {
    header "__stddef_size_t.h"
    export *
  }
}
```

Building Modules

Module Building Basics

Compiled to on disk PCMs

Contents depend on compiler flags like `-target`, and `-D`

Subset of the command line forms the *module context hash*

PCMs for the same module with differing hashes are called *variants*

PCMs can only be safely reused between compiles if the hash matches

Initial Solution - Implicitly Built Modules

Modules built by each compiler process when needed

Other processes block while that module builds

Already built modules are independently checked for changes

Common to have different module context hashes across translation units in large projects

Leads to slower builds due to poor PCM reuse and duplicate work

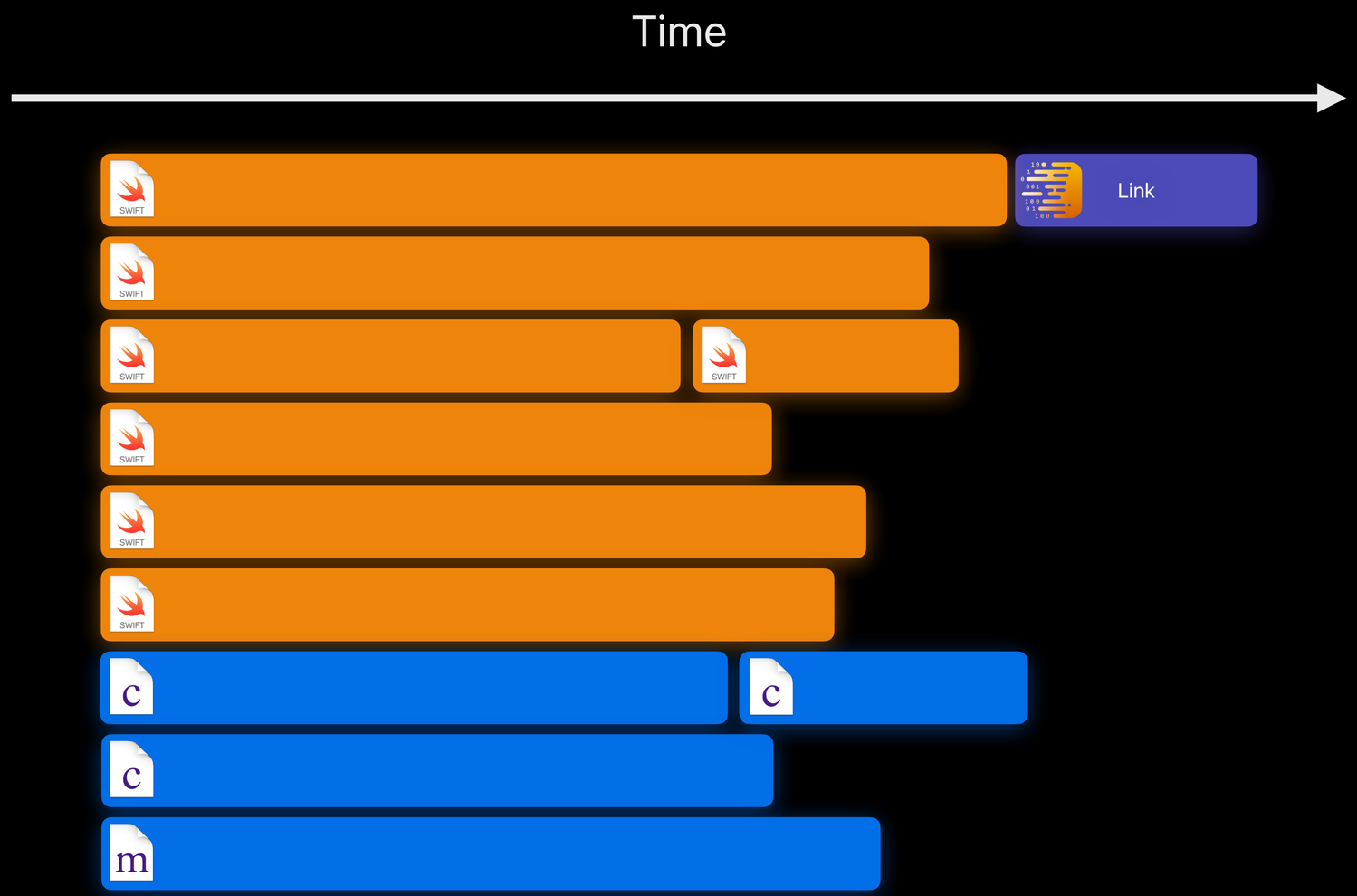
Initial Issues and Workarounds

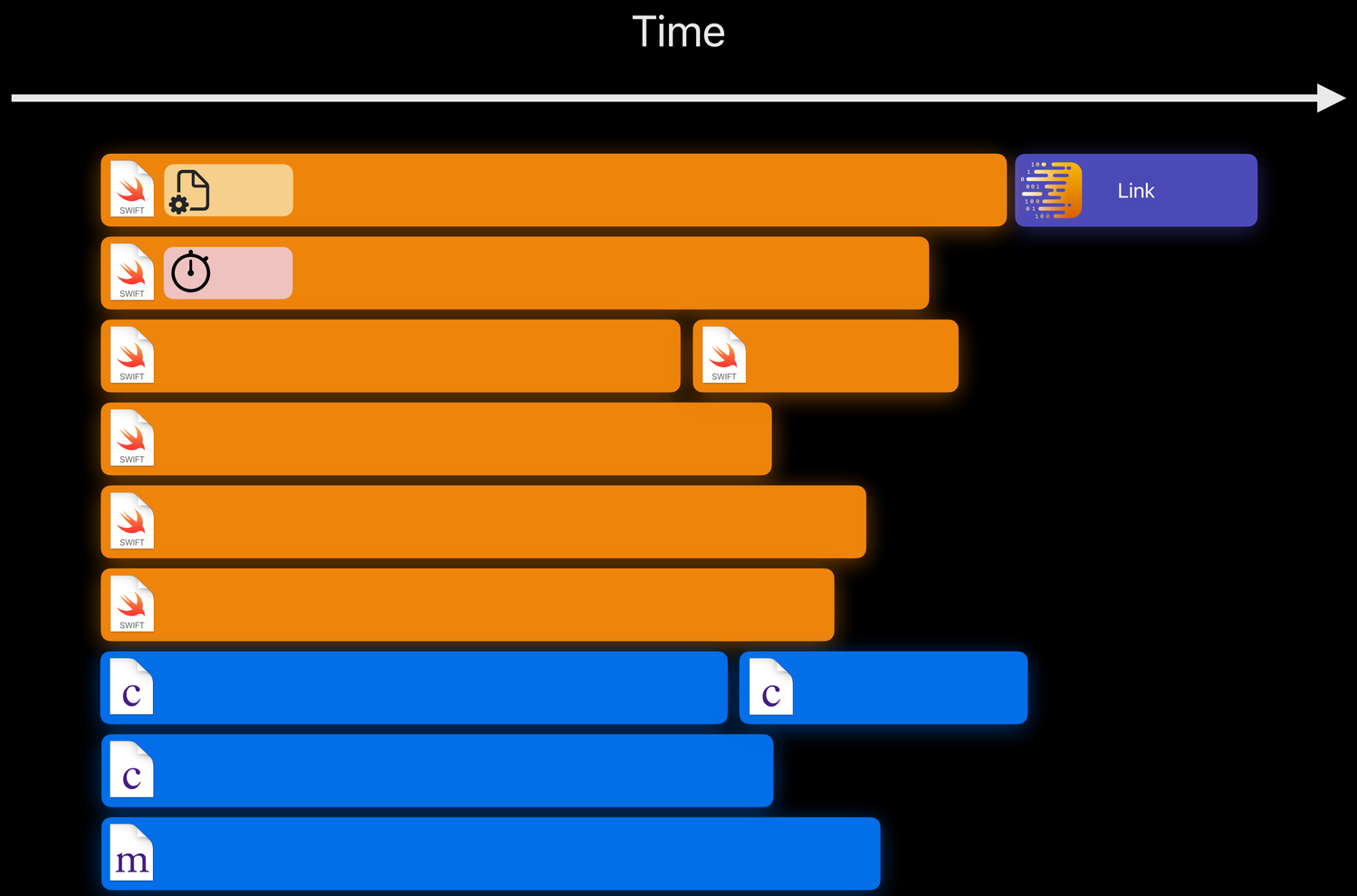
Builds are too slow

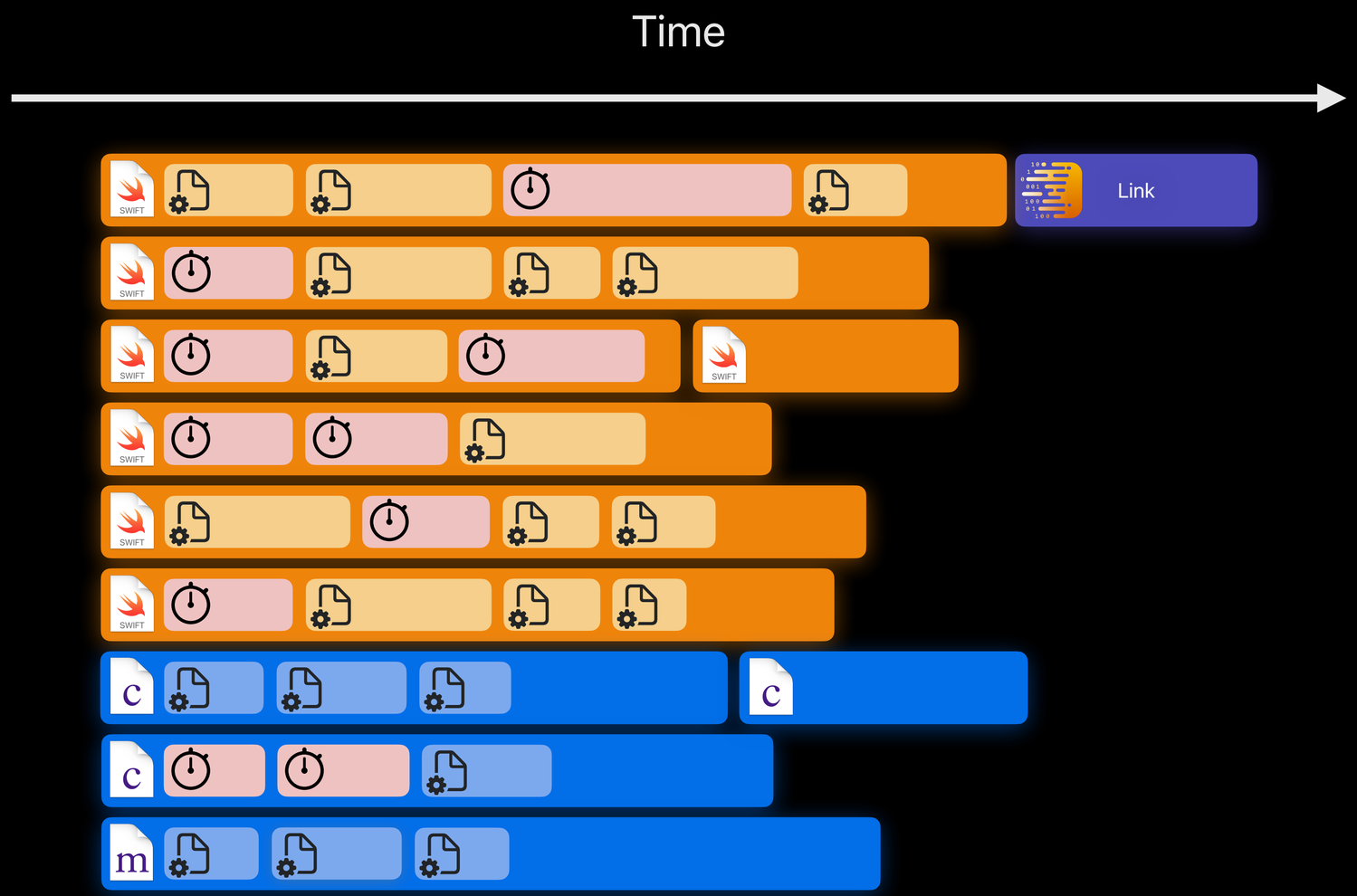
Cheat and use *non-strict* module context hashing to get more frequent reuse

Sacrifices correctness, sometimes manifests as compiler errors

Fixed duplicate validation with build sessions


Time
SWIFT
SWIFT
SWIFT
SWIFT
SWIFT
SWIFT
SWIFT
Link
c
c
c
m


Time
SWIFT
Link
SWIFT
SWIFT
SWIFT
SWIFT
SWIFT
SWIFT
c
c
c
m


Time
SWIFT
Link
c
m

Better Solution - Explicitly Built Modules

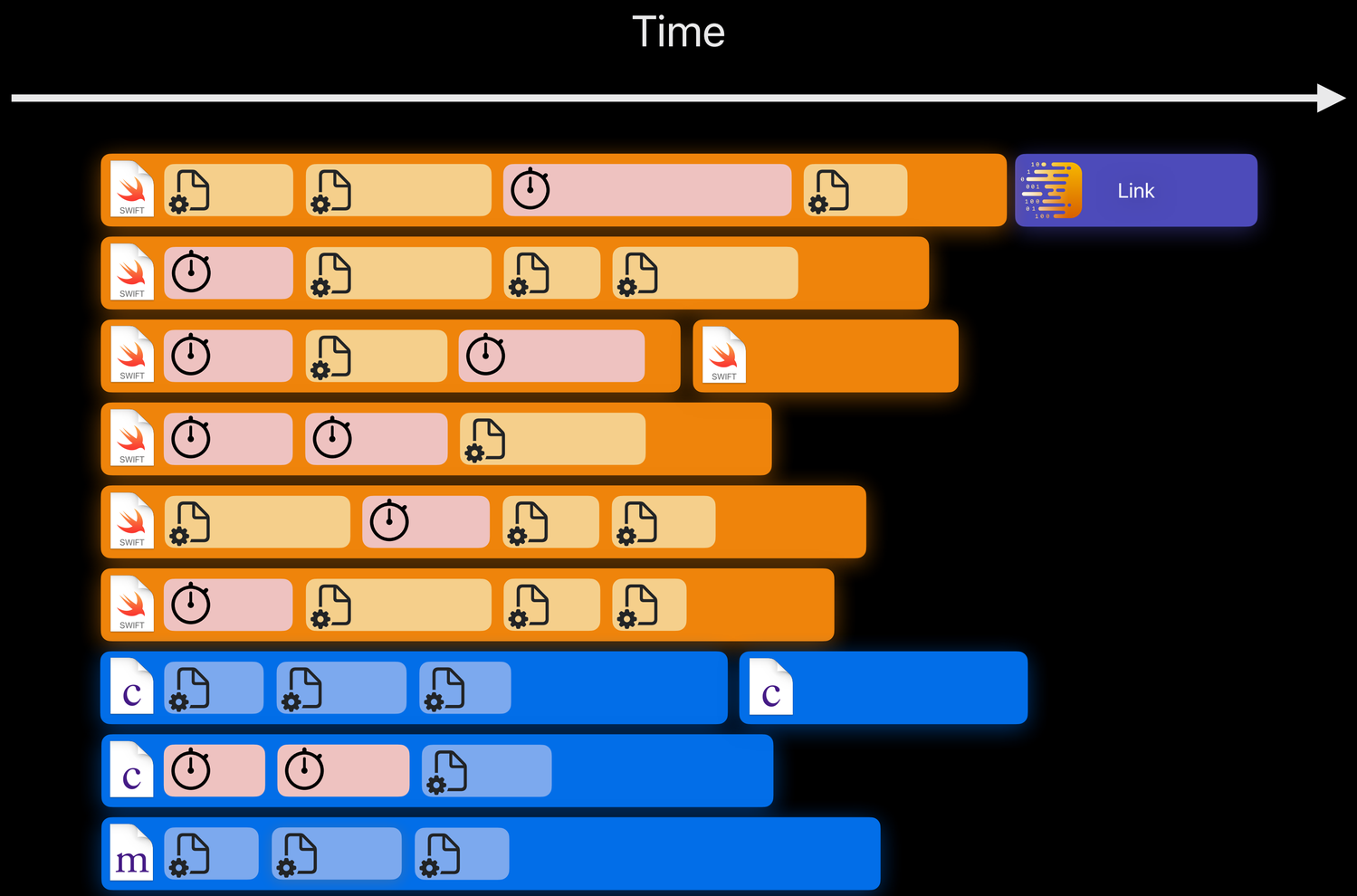
Build system scans for needed modules and builds them up front

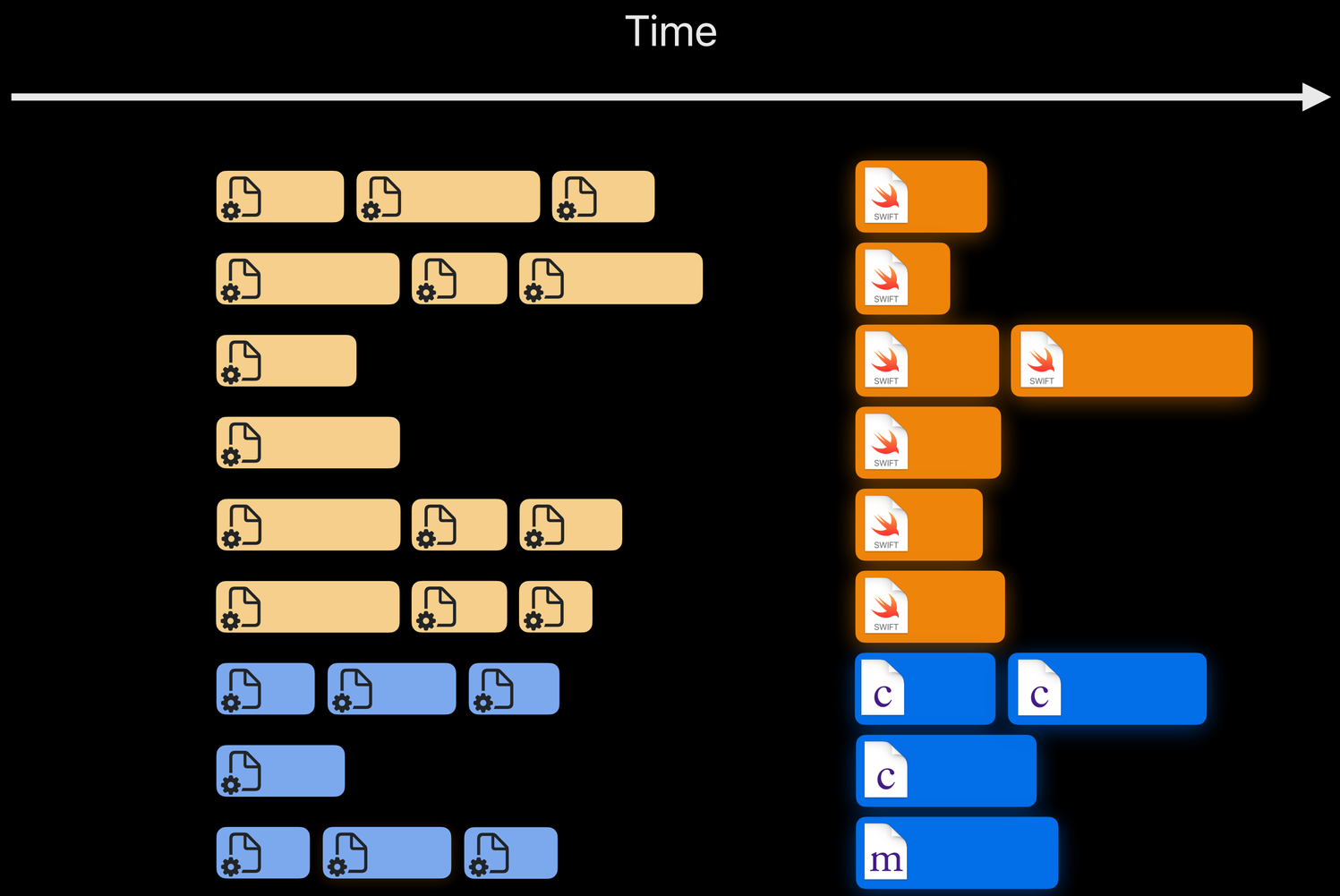
Scanning can detect some config differences that don’t matter and remove them

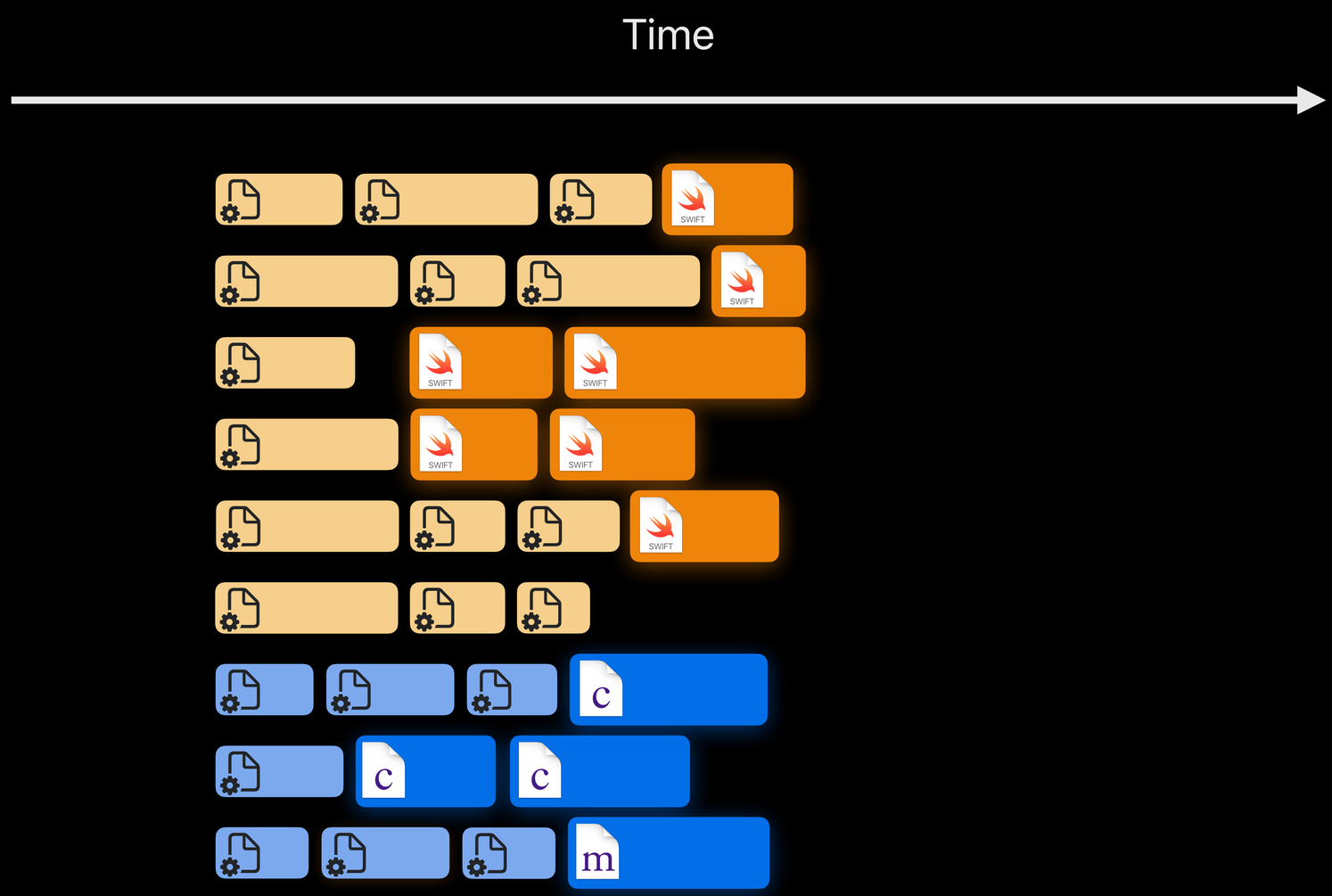
This allows for strict module context hashes

Fixes correctness and many bugs

More precise ⇒ more modules built in some cases


Time
SWIFT
Link
c
m


Time
SWIFT
SWIFT
SWIFT
SWIFT
SWIFT
SWIFT
SWIFT
c
c
c
m


Time
SWIFT
SWIFT
SWIFT
SWIFT
SWIFT
SWIFT
SWIFT
C
C
C
m

Performance

Building a module currently has a large overhead

Lots of variants and small modules decrease scanning performance

Globally unique config removes variants

Combine headers where they don’t create cycles

Single Module

Module per Header

Conclusions

- Modules change how the C preprocessor works
- Header layering should be set before modularizing
- Fixing modularization after client adoption is very difficult
- Modules affect build performance, requires work on the build system and from clients