

C++ Modules: What You Should Know

Gabriel Dos Reis
Microsoft

Modules Are Coming

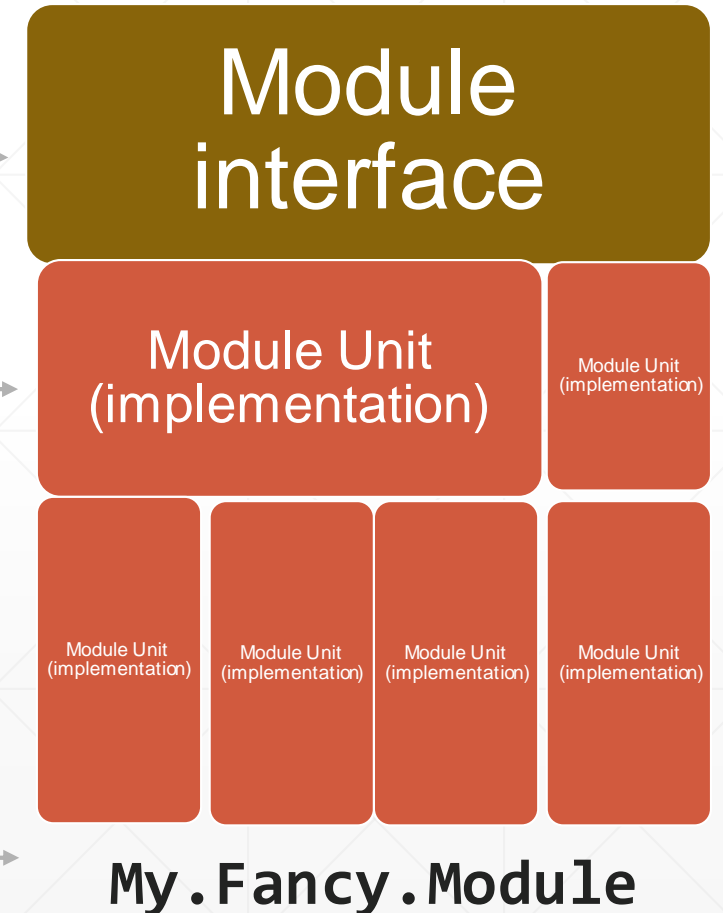
For the Impatients

- Designed in the North
 - Modules TS (lead effort from Microsoft)
 - Amendments from Google and others
 - Specification in the C++20 Working Draft – CD for ballot in July, after Cologne meeting
- Available from these major compiler providers
 - MSVC: <https://devblogs.microsoft.com/cppblog/>
 - GCC: <https://gcc.gnu.org/wiki/cxx-modules>
 - Clang: <https://clang.llvm.org/>

What Are They?

What is a Module?

- **Collection of related translation units, with a well-defined set of entry points**
- **Module interface:** set of declarations available to any consumer of a module
- **Module unit:** TU element of a module
- **Module name:** symbolic reference for a module



Production and Consumption of a Module

Calendar/Date.ixx

```
module;
#include <ostream>
#include <string>
import calendar.month;

export module calendar.date;

namespace Chrono {
    export struct Date {
        Date(int, Month, int);
        int day() const { return d; }
        Month month() const { return m; }
        int year() const { return y; }
    private:
        int d;
        Month m;
        int y;
    };

    export
    std::ostream& operator<<(std::ostream&, const Date&);
    export std::string to_string(const Date&);
}
```

Use-date.cxx

```
#include <iostream>
import calendar.date;

int main() {
    using namespace Chrono;
    Date date { 15, Month::Jun, 2019 };
    std::cout << "Today is " << date << std::endl;
}
```

Production and Consumption of a Module

Calendar/Date.ixx

```
module;
#include <ostream>
#include <string>
import calendar.month;

export module calendar.date;

export namespace Chrono {
    struct Date {
        Date(int, Month, int);
        int day() const { return d; }
        Month month() const { return m; }
        int year() const { return y; }
    private:
        int d;
        Month m;
        int y;
    };

    std::ostream& operator<<(std::ostream&, const Date&);
    std::string to_string(const Date&);
}
```

Use-date.cxx

```
#include <iostream>
import calendar.date;

int main() {
    using namespace Chrono;
    Date date { 15, Month::Jun, 2019 };
    std::cout << "Today is " << date << std::endl;
}
```


Production and Consumption of a Module (look no ‘#include’ Ma)

Calendar/Date.ixx

```
export module calendar.date;

import <ostream>;
import <string>;
import calendar.month;

export namespace Chrono {
    struct Date {
        Date(int, Month, int);
        int day() const { return d; }
        Month month() const { return m; }
        int year() const { return y; }
    private:
        int d;
        Month m;
        int y;
    };

    std::ostream& operator<<(std::ostream&, const Date&);
    std::string to_string(const Date&);
}
```

Use-date.cxx

```
import <iostream>;
import calendar.date;

int main() {
    using namespace Chrono;
    Date date { 15, Month::Jun, 2019 };
    std::cout << "Today is " << date << std::endl;
}
```

C++ Modules: Language Support for Software Design and Development

- **Componentization**
 - Software architecture
- **Isolation**
 - In particular from macros
- **Build throughput**
 - Structure understood and enforced by entire toolchain
- **Support for modern semantics-aware developer tools**
 - Enabling technology
- **Non Goals:**
 - Improve or remove the preprocessor

The Pedestrian's View

- **Modules are isolated from macros**
 - Interface is “compiled” set of exported entities
 - Not affected by macros defined in the importing TU
 - Conversely, macros defined in a module do not leak out
- **A unique place where exported entities are declared**
 - A module can be just one TU, or several TUs with distinguished TUs for exports (interface partitions)
- **Every entity is defined at exactly one place, and processed only once**
 - **Owned by the defining module**
 - Exception is made for “global module” (for seamless integration with existing code)
- **Few new name lookup rules**
 - We have too many already, and nobody knows how many
- **Modules do not replace header files**
 - Macro heavy interfaces are likely to continue using header files, with fairly modularized sub-components, header units
- **Build time is faster**

Programming with Modules

- Put header files `#include` first
- Put import declarations
- Module declaration
- Export declarations
- Other declarations to be shared with sibling module units
- ...

```
module;  
#include <assert.h>  
import <iostream>;  
import calendar.month;  
  
export module calendate.date;  
  
namespace Chrono {  
    export struct Date {  
        // ...  
    };  
}
```

Why Are We Doing This?

Pressing Challenges for Modern C++

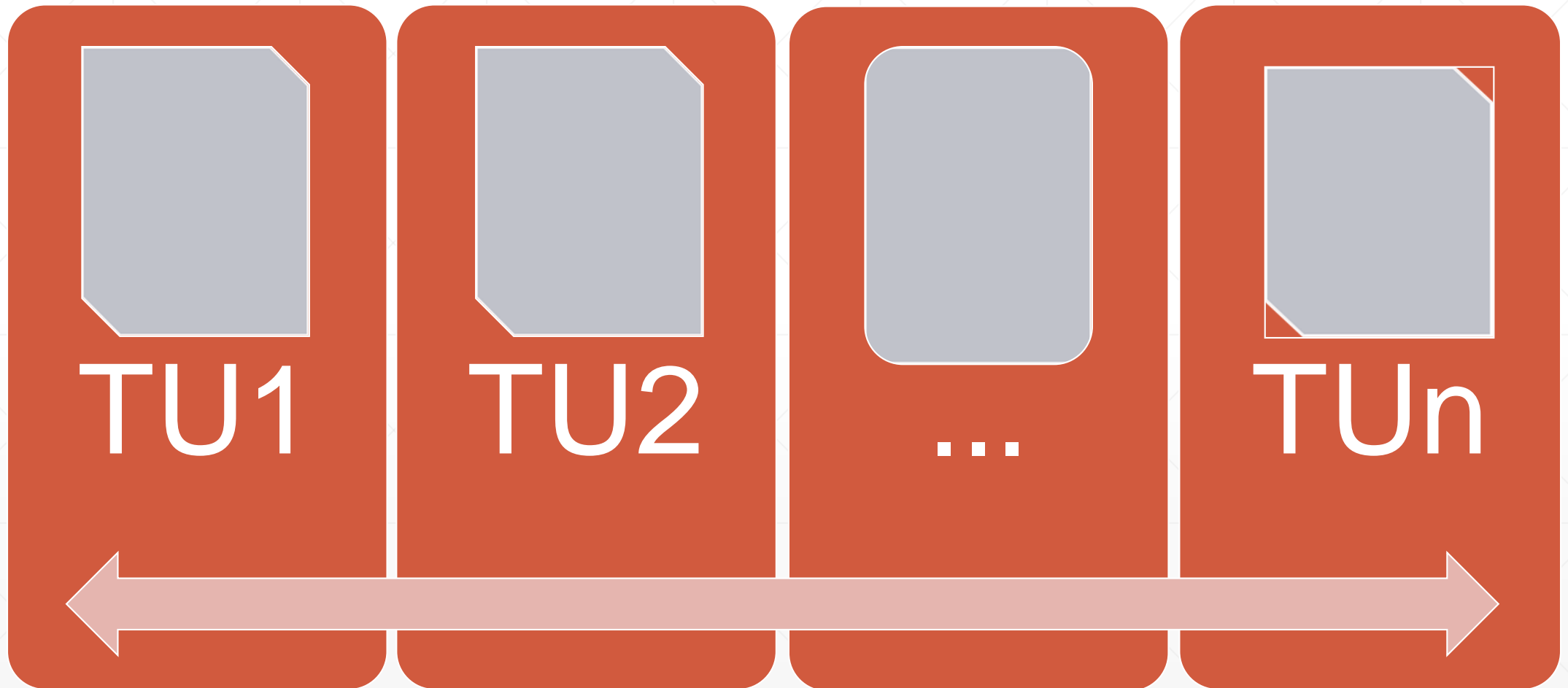
Pressing Challenges for Modern C++

- **Build time scalability of idiomatic C++**
 - Distributed build, cloud build, etc.
- **Paucity of semantics-aware developer tools**
 - Serious impediment to programmer productivity
 - Great disadvantage vis-à-vis contemporary languages (C#, Java, Ada, Rust, etc.)
 - Reason not to adopt C++
 - Reason to migrate away from C++
- **Source Code Organization at Large**
 - Scaling beyond the billion lines of codes
 - Producing, composing, consuming large components with well-defined semantics boundaries

Program Organization and Basic Linking Model

- **Program = Collection of *independently* translated units**
 - Each TU is processed in isolation, without knowledge of peer TUs
- **TUs communicate by brandishing declarations for *external names***
 - No explicit dependency on TU provider of external names
- **Linker resolves uses of external name to whichever definitions happen to match**
 - Type-safe linkage problems
 - One Definition Rule (ODR) violation

What is in a C++ Program?



Basic Linking Model

1.cc (producer of quant)

```
int quant(int x, int y) {  
    return x*x + y*y;  
}
```

2.cc (consumer of quant)

```
extern int quant(int, int);  
int main() {  
    return quant(3, 4);  
}
```

3.cc (another producer of quant)

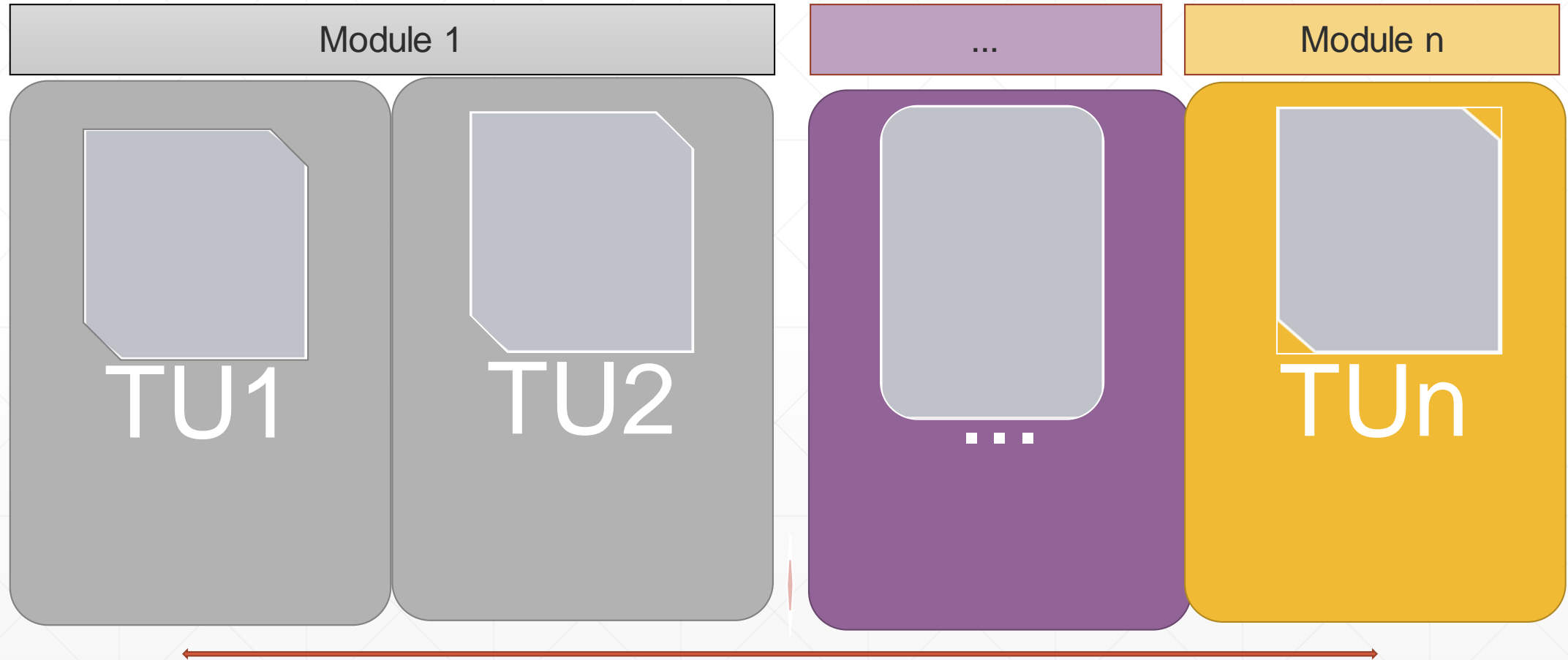
```
#include <stdlib.h>  
int quant(int x, int y) {  
    return abs(x) + abs(y);  
}
```

- Valid programs: (a) 1.cc and 2.cc; (b) 2.cc and 3.cc
- Effective, but low-level and brittle
 - Leak implementation details to language specification

Header Files and Macros

- **Enough has been said**
 - See Stroustrup's writings and recommendations
- **Source file inclusion is textual copy and paste**
 - Force repeated processing of definitions in header files
 - “duplicate” definitions need to be matched and “uniqued”
 - Redundant throw-away work by the compiler
 - Lead to elaborate specification of otherwise simple foundational principle (ODR)
- **Macros stifle developments of semantics-aware dev tools**

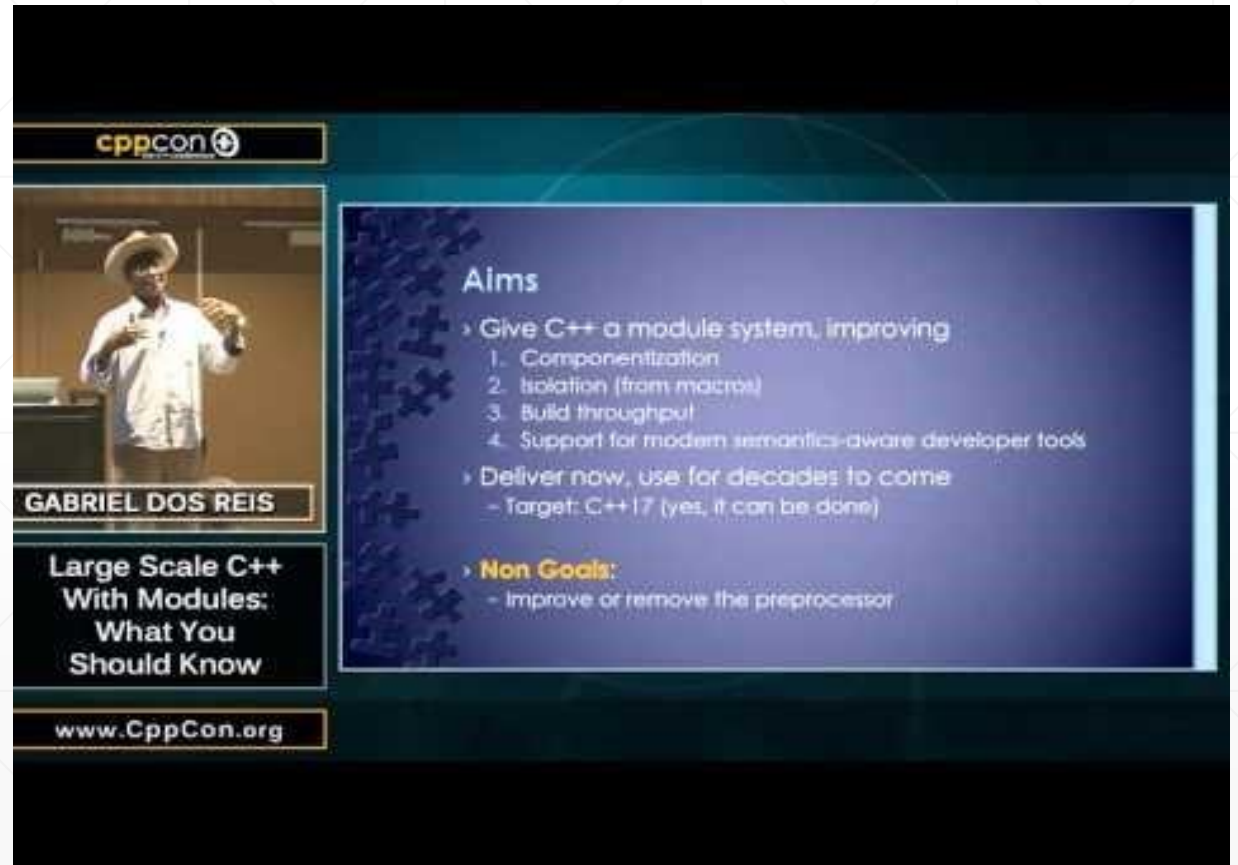
Solution: Componentization



How Are We Doing It?

CppCon2015 presentation on Modules

Large Scale C++ With Modules: What You Should Know



The screenshot shows a presentation slide from CppCon2015. On the left, a video inset shows Gabriel Dos Reis, a man wearing a hat and a light-colored shirt, gesturing while speaking. The slide content is on the right, featuring a dark blue background with a puzzle piece pattern. The title 'Large Scale C++ With Modules: What You Should Know' is displayed in white text. Below the title is the speaker's name 'GABRIEL DOS REIS' and the website 'www.CppCon.org'. The main content area is titled 'Aims' and lists several goals and non-goals for C++ modules.

cppcon

GABRIEL DOS REIS

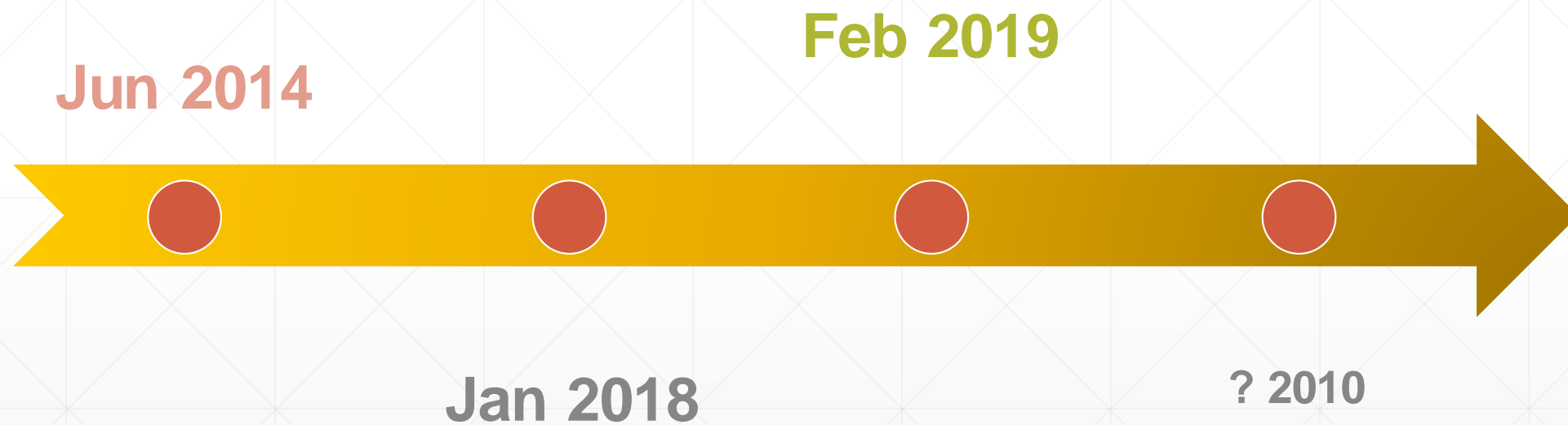
Large Scale C++
With Modules:
What You
Should Know

www.CppCon.org

Aims

- › Give C++ a module system, improving
 1. Componentization
 2. Isolation (from macros)
 3. Build throughput
 4. Support for modern semantics-aware developer tools
- › Deliver now, use for decades to come
 - Target: C++17 (yes, it can be done)
- › **Non Goals:**
 - Improve or remove the preprocessor

Simplified Timeline



Modules TS

Programming Languages — Extensions to C++ for Modules

Langage de programmation — Extension C++ pour les modules

Warning

This document is not an ISO International Standard. It is distributed for review and comment. It is subject to change without notice and may not be referred to as an International Standard.

Recipients of this draft are invited to submit, with their comments, notification of any relevant patent rights of which they are aware and to provide supporting documentation.

Document type: Technical Specification

Document stage: (30) Committee

Document Language: E

Merged Modules

- At the Winter 2019 Kona meeting
- Modules TS + Elements of ATOM proposal (Google) + Various feedback
- New **essential additions**
 - Header units
 - Module partitions
 - Notion of preamble
 - **import** and **module** made context-sensitive keywords

Header Unit

- Formalization of Precompiled Header (PCH)

Production and Consumption of a Module

Calendar/Date.ixx

```
export module calendar.date;

import <ostream>;
import <string>;
import calendar.month;

export namespace Chrono {
    struct Date {
        Date(int, Month, int);
        int day() const { return d; }
        Month month() const { return m; }
        int year() const { return y; }
    private:
        int d;
        Month m;
        int y;
    };

    std::ostream& operator<<(std::ostream&, const Date&);
    std::string to_string(const Date&);
}
```

Use-date.cxx

```
import <iostream>;
import calendar.date;

int main() {
    using namespace Chrono;
    Date date { 15, Month::Jun, 2019 };
    std::cout << "Today is " << date << std::endl;
}
```

Module Partitions

- Modules TS has a major inconvenient restriction
 - Only one single monolithic TU for module interface
 - Developers like to use separate files for conceptually separate abstractions
- A module partition can be interface unit
 - Example: `export module BinarySearchTree : Data;`
- A module partition can be implementation unit
 - Example: `module BinaryTree: Data;`
- Still a single TU for module interface
 - Must re-export all partition interface units

Module Partitions

```
export module BinarySearchTree;  
  
export import BinarySearchTree : Data;  
export import BinarySearchTree : Algo;
```

```
export module BinarySearchTree : Data;  
  
export template<typename T>  
struct BinaryTree {  
    // ...  
};
```

```
export module BinarySearchTree : Algo;  
  
export template<typename T>  
void insert(BinaryTree<T>&, const T&);  
  
template<typename T>  
void rebalance(BinaryTree<T>&) {  
    // ...  
}
```

When to Use '#include' vs. Header Units

Production and Consumption of a Module

Calendar/Date.ixx

```
module;
#include <ostream>
#include <string>
import calendar.month;

export module calendar.date;

export namespace Chrono {
    struct Date {
        Date(int, Month, int);
        int day() const { return d; }
        Month month() const { return m; }
        int year() const { return y; }
    private:
        int d;
        Month m;
        int y;
    };

    std::ostream& operator<<(std::ostream&, const Date&);
    std::string to_string(const Date&);
}
```

Use-date.cxx

```
#include <iostream>
import calendar.date;

int main() {
    using namespace Chrono;
    Date date { 15, Month::Jun, 2019 };
    std::cout << "Today is " << date << std::endl;
}
```

Production and Consumption of a Module

Calendar/Date.ixx

```
export module calendar.date;

import <ostream>;
import <string>;
import calendar.month;

export namespace Chrono {
    struct Date {
        Date(int, Month, int);
        int day() const { return d; }
        Month month() const { return m; }
        int year() const { return y; }
    private:
        int d;
        Month m;
        int y;
    };

    std::ostream& operator<<(std::ostream&, const Date&);
    std::string to_string(const Date&);
}
```

Use-date.cxx

```
import <iostream>;
import calendar.date;

int main() {
    using namespace Chrono;
    Date date { 15, Month::Jun, 2019 };
    std::cout << "Today is " << date << std::endl;
}
```


When to Use '#include' vs. Header Units

- Use '#include' when you want to project a modular view over a “messy” header file”, or you know what you are using
 - Usually, smaller footprint of metadata
- Use **header unit** when the header file is relatively well-structured and modular, and you know really know what you are using
 - Be mindful about header files that are included for their side-effects

Implications for the C++ Ecosystem

New Reality

- Build systems
 - Code Analyzers
 - IDEs
 - Education Materials
 - C++ in the 21st Century
-
- New WG21 Study Group: SG-15 – Tooling
 - Produce a TR for C++ tool providers

Enabling Technology

Food for Thought

- Any good, scalable, production-grade implementation of C++ Modules persists compiled module interface in some form
- Disastrous for the C++ community to have N compilers on the same platform use N distinct formats
- Disastrous for the C++ community to miss opportunity to share same format across wide selection (all?) of platforms
 - Fragmentation of the tooling community and tooling ecosystem

Module Interface Metadata: Constraints

- Must represent completely elaborated C++
 - E.g. types, expressions, etc. (template arguments involve both types and non-types)
- Must represent templates and associated constraints
 - Syntactic constructs, dependent names, non-dependent names
- Input source code **NOT** a good representation
 - Requires a C++ compiler
 - Requires repeating work already done
- Conclusion: ***Must represent all of C++ faithfully***

A Common Module Interface Format

- Not standardization of an ABI
 - Rather, a **common structure** to describe a C++ source program after elaboration
- Ideal: Binary format developed by the C++ community
 - Not necessarily WG21 (although WG21 involvement would be nice)
- Starting point (suggestion): the Internal Program Representation
 - Work done a decade ago by GDR and B. Stroustrup: “**A Principled, Complete, and Efficient Representation of C++**”
 - Open source with BSD-type license: <https://github.com/GabrielDosReis/ipr>
- Description of the complete **Abstract Semantics Graph**

Call to Action

Help Us Help You

- Implementations
 - MSVC: <https://devblogs.microsoft.com/cppblog/> with `/experimental:modules`
 - GCC: <https://gcc.gnu.org/wiki/cxx-modules> with `-fmodules-ts`
 - Clang: <https://clang.llvm.org/> with `-fmodules-ts`
- Feedback to C++ Standards Committee (WG21)
 - SG2 (Modules): modules@isocpp.org
 - SG15 (Tooling): tooling@isocpp.org
 - Your NB Reps: AFNOR, DIN, BSI, PL.16, etc.

?

