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

Gabriel Dos Reis

Microsoft

Overview

Where We Are (Today)

USE-DATE.CXX

```
#include <iostream>
#include “Calendar/date.h”

int main() {
      using namespace Chrono;
      Date date { 22, Month::Sep, 2015 };
      std::cout << “Today is “ << date << std::endl;
}
```

CALENDAR/DATE.H

```
#ifndef CHRONO_DATE_INCLUDED
#define CHRONO_DATE_INCLUDED
#include <iosfwd>
#include <string>
#include “Calendar/Month.h”

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&);
}

#endif  // CHRONO_DATE_INCLUDED
```

A Better Place to Be

USE-DATE.CXX

```
import std.io;
import calendar.date;

int main() {
      using namespace Chrono;
      Date date { 22, Month::Sep, 2015 };
      std::cout << “Today is “ << date << std::endl;
}
```

CALENDAR/DATE.CXX

```
import std.io;
import std.string;
import calendar.month;

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&);
}
```

A Better Place to Be

USE-DATE.CXX

```
import std.io;
import calendar.date;

int main() {
      using namespace Chrono;
      Date date { 18, Month::Sep, 2015 };
      std::cout << “Today is “ << date << std::endl;
}
```

CALENDAR/DATE.CXX

```
import std.io;
import std.string;
import calendar.month;

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&);
}
```

What Is The Big Deal?

› Well,

```
#include <iostream>
#include “Calendar/date.h”

int main() {
	using namespace Chrono;
	Date date { 18, Month::Sep, 2015 };
	std::cout << “Today is “ << date << std::endl;
}
```

– is 176 bytes of user-authored text file (what user sees)

What Is The Big Deal?

› Well,

```
#include <iostream>
#include “Calendar/date.h”

int main() {
      using namespace Chrono;
      Date date { 18, Month::Sep, 2015 };
      std::cout << “Today is “ << date << std::endl;
}
```

– is 176 bytes of user-authored text file (what user sees)
– expands to (what compiler sees)
  › 412,326 bytes with GCC 5.2.0 – or 234,276% ~~compression~~expansion
  › 1,203,953 bytes with Clang 3.6.1– or 684,064% inflation
  › 1,083,255 bytes with VC++ Dev14 – or 615,485% inflation

What Is The Big Deal?

› Well,

```
#include <iostream>
#include “Calendar/date.h”

int main() {
      using namespace Chrono;
      Date date { 18, Month::Sep, 2015 };
      std::cout << “Today is “ << date << std::endl;
}
```

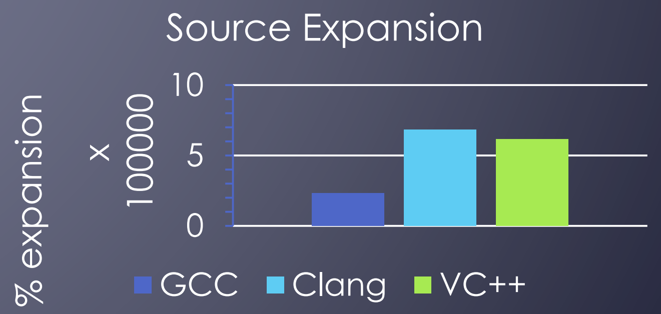
– is 176 bytes of user-authored text file (what user sees)

– expands to (what compiler sees)

› 412,326 bytes with GCC 5.2.0

› 1,203,953 bytes with Clang 3.6.1

› 1,083,255 bytes with VC++ Dev14



Why Is It a Big Deal?

```
#include <iostream>
#include “Calendar/date.h”
```

Why Is It a Big Deal?

```
#include <iostream>
#include “Calendar/date.h”
```

- Preprocessor directive `#include` ***is*** textual copy and paste
  - Compiler works hard to process the same entity multiple times
  - Linker works hard to throw all of them away, except one
  - Miserable build throughput

Why Is It a Big Deal?

```
#include <iostream>
#include “Calendar/date.h”
```

- Preprocessor directive `#include` ***is*** textual copy and paste
  - Compiler works hard to process the same entity multiple times
  - Linker works hard to throw all of them away, except one
  - Miserable build throughput
- **Copy**: No consistency guarantee
  - Hard to track bugs (famous “ODR” violation)
  - No component boundaries, brittle enforcement

Why Is It a Big Deal?

```
#include <iostream>
#include “Calendar/date.h”
```

- Preprocessor directive `#include` ***is*** textual copy and paste
  - Compiler works hard to process the same entity multiple times
  - Linker works hard to throw all of them away, except one
  - Miserable build throughput
- **Copy**: No consistency guarantee
  - Hard to track bugs (famous “ODR” violation)
  - No component boundaries, brittle enforcement
- **C Preprocessor technology**: Impossible to correctly parse/analyze a component
  - Need to know all macros

Pressing Challenges for Modern C++

- **Source Code Organization at Large**
  - Scaling beyond billions of lines of code
  - Producing, composing, consuming components with well-defined semantics boundaries
- **Paucity of Semantics-Aware Developer Tools**
  - Serious impediment to programmer productivity
  - Great disadvantage vis-à-vis contemporary languages (C#, Java, Ada, etc.)
    - Reason not to adopt C++
    - Reason to migrate away from C++
- **Build time Scalability of Idiomatic C++**
  - Distributed build, cloud build, etc.
    - Use ***semantics*** difference to accelerate build

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

When Can I Use It?

When Can I Use It?

- **MS VC 2015 Update 1 Timeframe**
  - Experimental implementation of the module proposal
  - Feedback based on use
  - Evidence of feasibility (for C++17)

When Can I Use It?

- **MS VC 2015 Update 1 Timeframe**
  - Experimental implementation of the module proposal
  - Feedback based on use
  - Evidence of feasibility (for C++17)
- Clang has an implementation based on “module maps”

What Hand Are We Dealt?

Program Organization

- **Program = Collection of independently translated units**
  - Each TU processed in isolation, ***without knowledge of peer TUs***
- **TUs communicate by brandishing declarations for external names**
  - No explicit dependency on TU or component provider
  - No good scalable way to check/verify consistency
- **Linker resolves uses of external names to whichever definitions happen to match (somehow)**
  - Type-safe linkage problems
  - Opportunities for One Definition Rule (ODR) violation

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 (YAPoQ)

```
#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**
- Useful, effective, but low-level and brittle
  - Leak implementation details to language specification

The Root Cause: One Definition Rule (ODR)

- What is ODR anyway?
  - Bjarne Stroustrup:
    - "I asked Dennis when I started in 1979."
  - [DMR]:
    - "**as if there was exactly one section of source text**"
- In the C++ standards
  - Several pages of opaque text about token-for-token comparison, name lookup, overload resolution, template instantiation contexts, etc.
  - Bjarne Stroustrup:
    - "**Every single word about "token comparison" is there to workaround absence of a real module system**"

Modules: 33,000ft view

Consumption

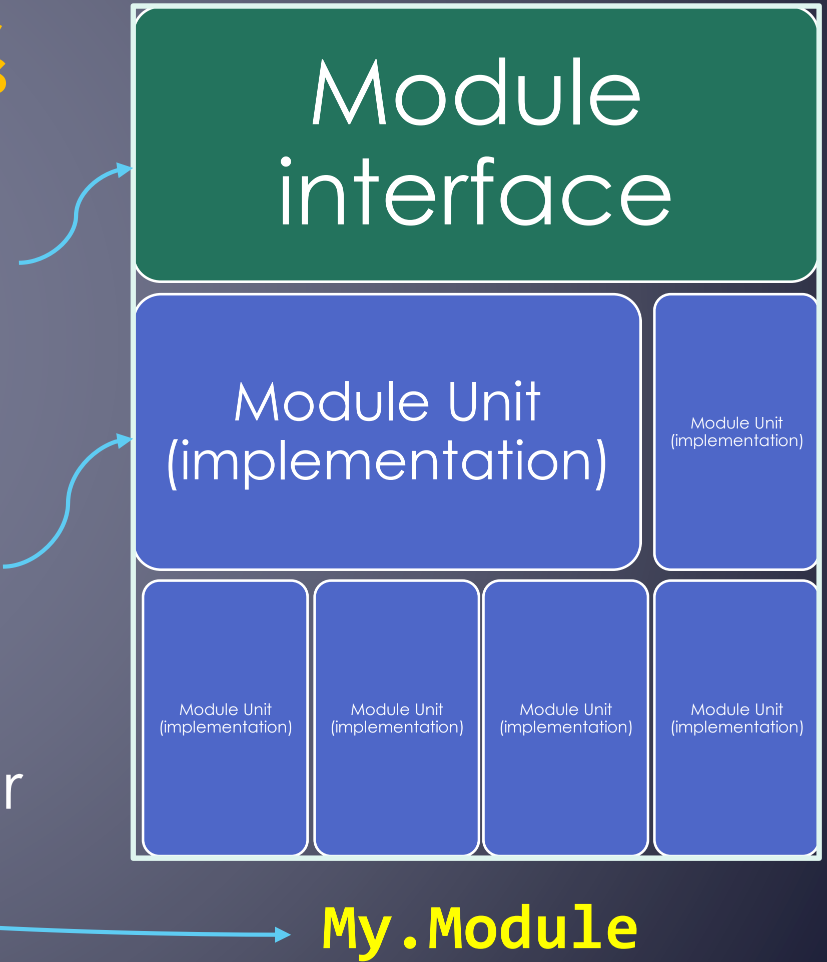
› As simple as

```
import std.io;                                                        // #include <iostream>
import calendar.date;                                                 // #include “Calendar/Date.h”

int main() {
      using namespace Chrono;
      Date date { 18, Month::Sep, 2015 };
      std::cout << “Today is “ << date << std::endl;
}
```

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


Module interface
Module Unit (implementation)
Module Unit (implementation)
Module Unit (implementation)
Module Unit (implementation)
Module Unit (implementation)
Module Unit (implementation)
My.Module

Production

- As simple as

```
import std.io;
import std.string;
import calendar.month;

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& os, const Date& d)
	{
	    // …
	}
	// …
	export std::string to_string(const Date& d)
	{
	    // …
	}
}
```

Production

› As simple as

```
import std.io;
import std.string;
import calendar.month;

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& os, const Date& d)
      {
         // …
      }
      // …
      std::string to_string(const Date& d)
      {
         // …
      }
}
```

Production

› As simple as

```
#include <iostream>
import std.string;
import calendar.month;

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& os, const Date& d)
      {
          // …
      }
      // …
      export std::string to_string(const Date& d)
      {
          // …
      }
}
```

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 a distinguished TU for exports
- **Every entity is defined at exactly one place, and processed only once**
  - **Owned by the defining module**
  - Except full semantics analysis of templates
  - Exception is made for "global module" (for seamless integration)
- **No 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
- **Build time is faster (goal)**

The Rules of Engagement

What To Expect

- Module owns entities in its purview
  - **ODR: every entity is defined exactly once**
- Order of consecutive import declarations is irrelevant
- Modules are isolated from macros
- Import declarations only makes name available
  - **You don't pay for what you don't use**

- Module metadata suitable for use by packaging systems
- Modules provide ownership

Anatomy of a Module Unit

Module purview

```
#include <iostream>

import Enum.Utils;                          // for bits::rep().

module Calendar.Month;

namespace Chrono {
     export enum class Month { Jan = 1, Feb, Mar, Apr, May, Jun, /*… */ };

     constexpr const char* month_name_table[] = {
          “January”, “February”, /* … */
     };

     export std::ostream& operator<<(std::ostream& os, Month m)
     {
         assert(m >= Month::Jan and m <= Month::Dec);
         return os << month_name_table[bits::rep(m) - 1];
     }
}
```

Anatomy of a Module Unit

Module purview

```
#include <iostream>

import Enum.Utils;                          // for bits::rep().

module Calendar.Month;

namespace Chrono {
     export enum class Month { Jan = 1, Feb, Mar, Apr, May, Jun, /*… */ };

     constexpr const char* month_name_table[] = {
          “January”, “February”, /* … */
     };

     export std::ostream& operator<<(std::ostream& os, Month m)
     {
         assert(m >= Month::Jan and m <= Month::Dec);
         return os << month_name_table[bits::rep(m) - 1];
     }
}
```

Namespace partition

A Word on Ownership

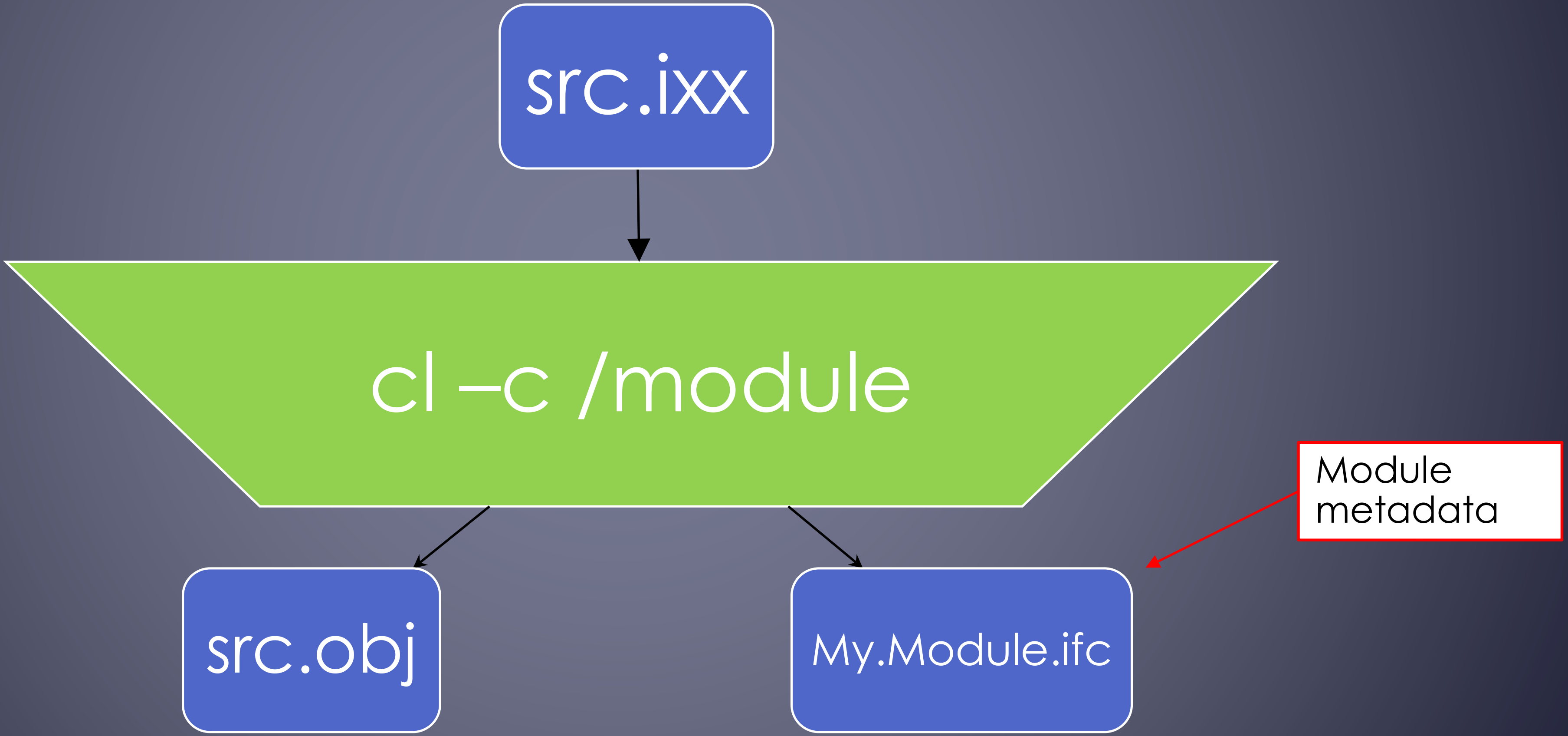
Impoverished Linking Abstractions

- **Strings and bytes**
  - Name “mangling” or name “decoration”
  - Unfortunate leakage to language specification
- **Standard “linkage” far behind the practice and needs of our time**
- Examples:
  - GCC and Clang support linkage “visibility”
    - default
    - hidden
    - internal
    - protected
  - VC++ supports:
    - dllimport
    - dllexport

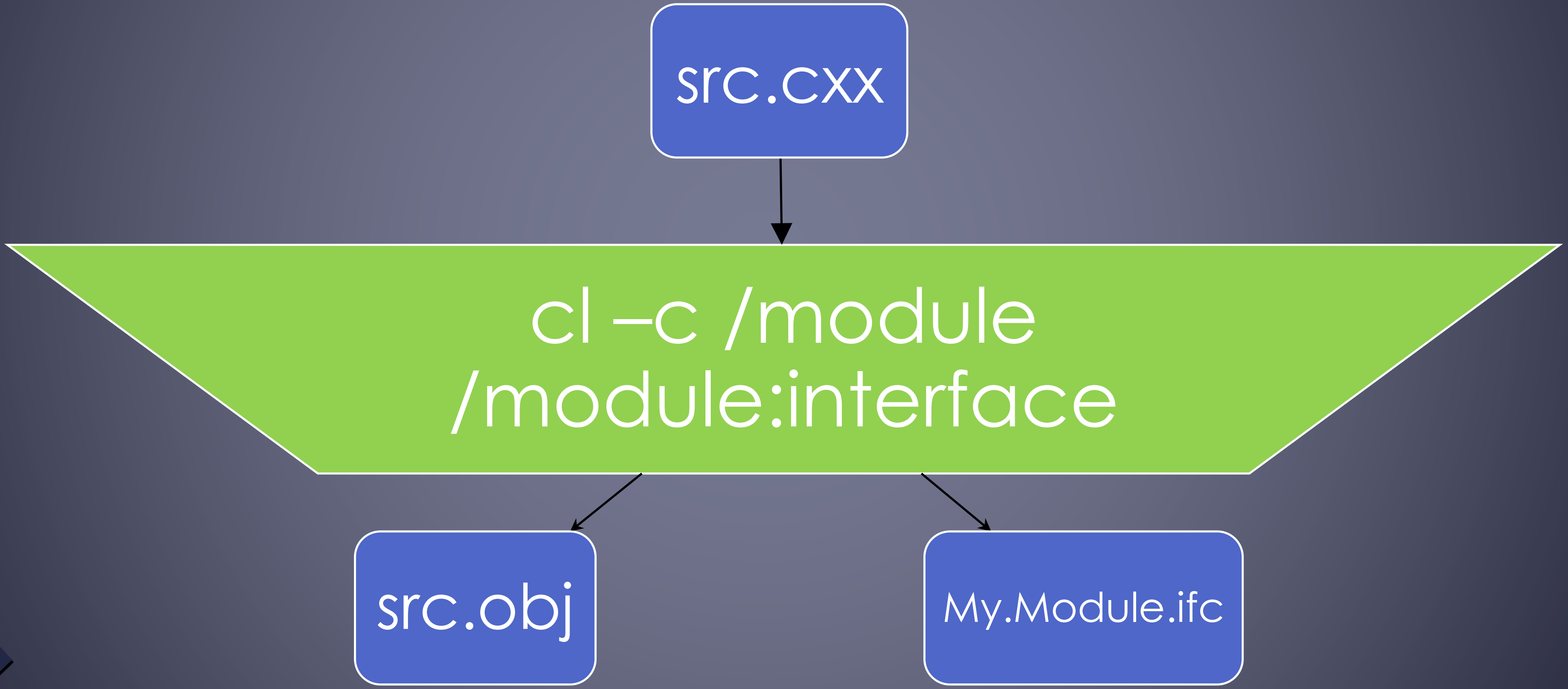
Tool Support

VC++ ongoing implementation

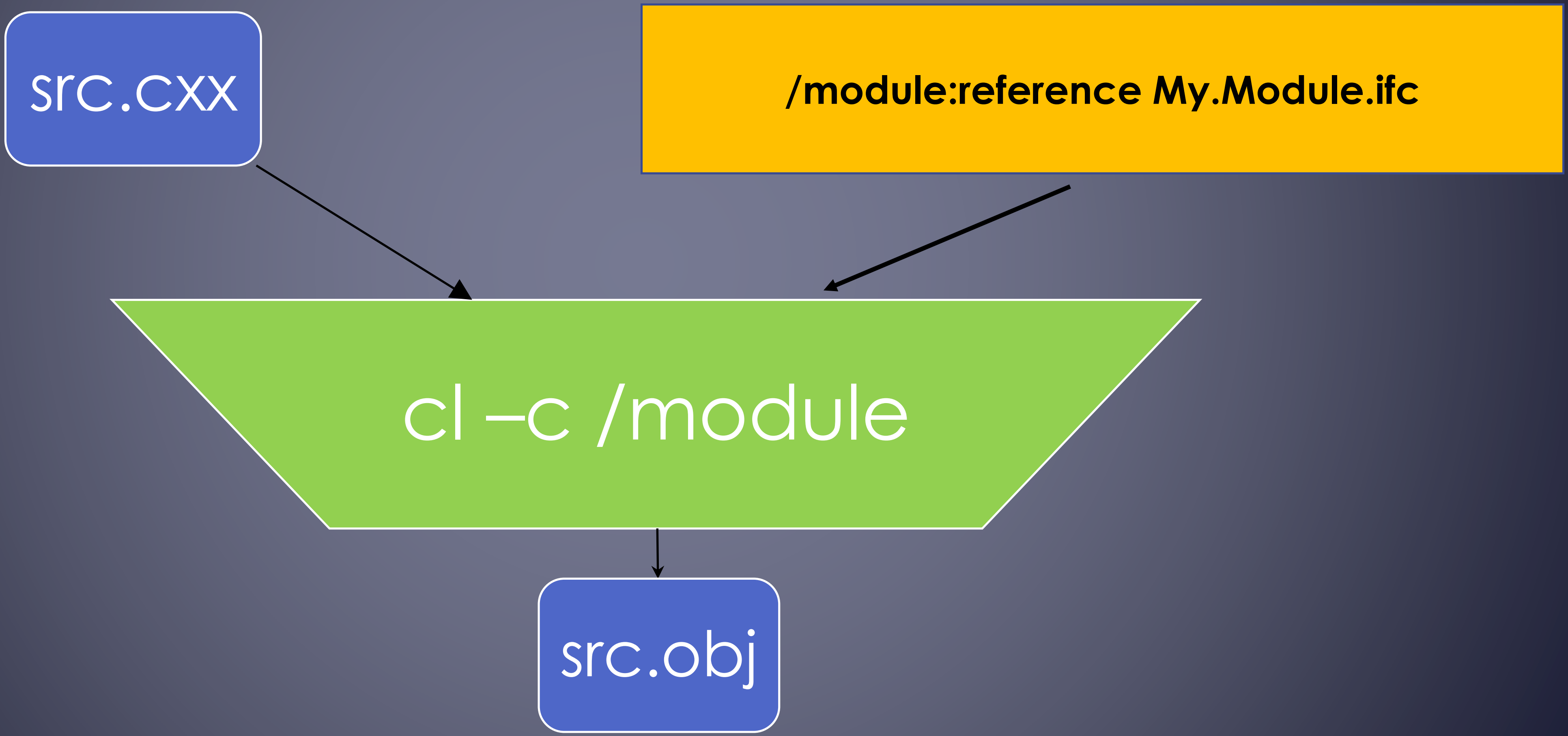
Production: Compiling a module interface



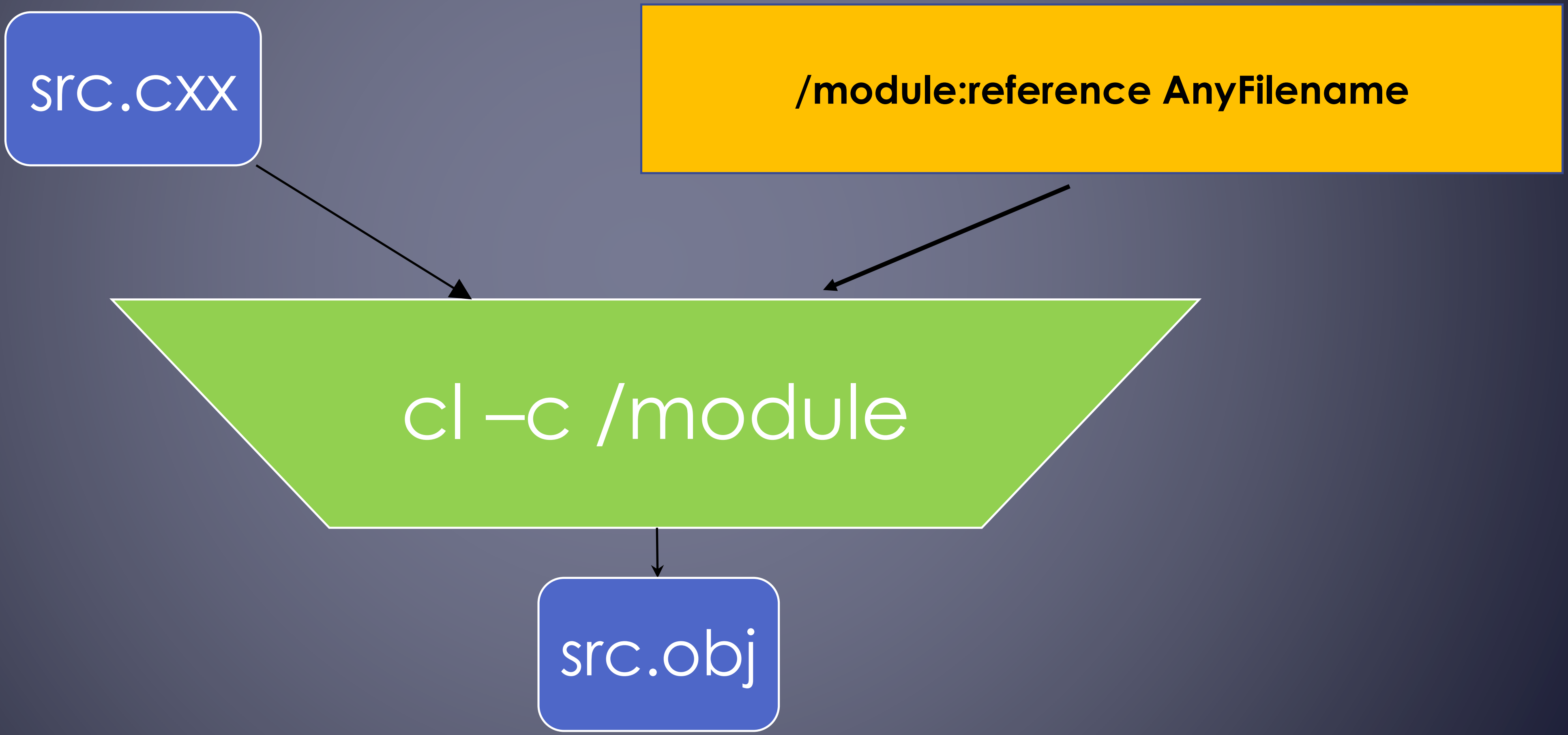
Production: Compiling a module interface



Consumption



Consumption



Compiler Options

- **/module**
  - Turn on module support
  - New keywords: module, import, export
- **/module:interface**
  - Force the compiler to interpret source code as module interface definition
- **/module:reference** <filename>
  - Look for a compiled module interface (IFC) in the file designated by the path
- **/module:search** <directory>
  - Search directory for referenced files

Transition paths

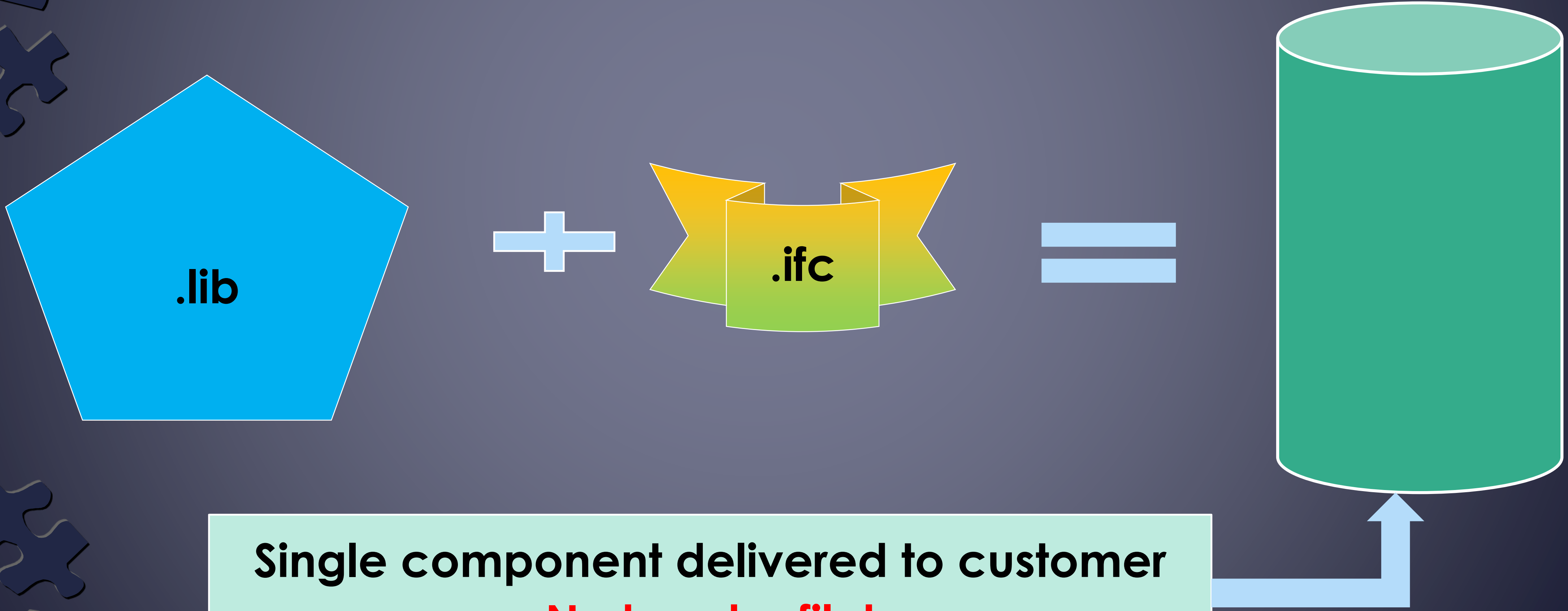
- **Can one consume a header file as a module?**
  - Only if the header file “behaves” well
    - **/module:export** vec.cxx **/module:name** std.vector

- **Ablity to selectively export “well behaved” macros**
  - **/module:exportMacro** <macroName>

Compiled Module Interface (IFC)

- **Binary Format Designed to represent C++**
  - Intended to be open (ideally used by all C++ implementations)
  - Recognizable by C++ programmers and implementers alike
  - An open-source reference implementation (e.g. on GitHub)
  - Compact, efficient, complete
- **Structure**:
  - Set of homogeneous tables representing all relevant entities
  - "pointers" are represented by typed indices (all 32-bit wide)
  - A "header" describing table locations, size, etc.
  - Principle: every index is well typed.
  - Deterministically produced by input module source file
- **Tooling**
  - Inspecting; embedding into static LIB or DLL; etc.
  - IDE integration

Enhancing Libraries with Module Interfaces



Single component delivered to customer

No header file!

User Feedback

Syntax

- “This is simple and elegant. Please do not make it ugly because it has to be C++”
  - OK. I will try.

Standardization

- "Can I get it in C++17?"
  - We are trying.
- "Pretty please, give me modules now"
  - We are trying
- "What about the IFC format"
  - After modules.
- "Really??? Are you kidding?"
  - No, but I can use some help

Controlling Visibility

- “What about inaccessible members? Are they exported too?”
  - Yes, but I hope we find a good solution
- “Come on!”

- See talks
  - by Steve and Ayman: "**What is New in Visual C++ 2015 and Future Directions**", Thursday
  - Neil MacIntosh on Static Analysis and Safe Buffer Types
  - gdr on "**Contracts**", tomorrow

Thanks!

Questions?