Implementing C++ Modules: Lessons Learned, Lessons Abandoned

Gabriel Dos Reis & Cameron DaCamara
Microsoft

Overview

C++ Modules Design Goals

1. **Componentization**
   – Compositional semantics of independently developed libraries or program parts
   – Explicit expression of boundaries and dependencies in code
2. **Isolation**
   – Self-contained semantics of a component
   – Semantic barriers respected
3. **Scalable build**
   – Improved "inner loop" experience, support distributed/cloud builds
   – Generally obtained as result of (1) + (2)
4. **Modern semantics-aware developer tools**
   – Bring C++ development experience to next level
   – Semantics-based browsing, transformation, completion, etc.

What to Take Away from this Talk?

- **How a C++ toolset can deliver on those promises**
  - Sharing experience from a complete implementation in Visual C++
  - Increased safety, better ODR violation mitigation
  - Focus on high level semantics; see GDR's talk on C++ representations

- **What a C++ programmer can do to benefit from these achievements**
  - Source code organization
  - Code hygiene to get 10x compile-time speed up

- **The One Definition Rule is your friend**
  - Named modules provide ODR guarantees, by design
  - Header files and header units require (expensive) incomplete ODR checks, and demand complete trust

Modules and ODR

One Definition Rule (ODR) and Implications

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

- Safety:
  - A toolchain should try its best to diagnose ODR violations
    - e.g. linker errors for duplicate definitions, mismatching types
- Speed:
  - Matching a function call to its definition begins "the great hunt"

The Global Module

- Everything we used to do before Modules
  - Merry mess
- Provide no ODR guarantees
  - Responsibility of the programmer
- Complete ODR check nearly impossible with conventional tools
  - Source files processed with "one-translation-unit-at-a-time" view
  - No reliable, formalized way to share/vehicle semantics across the set of source files making up a component

Module Ownership

Module Ownership

- Supporting programming in the large (componentization)
  - Non-interference between module-local entities (module linkage)
  - Interface provenance:
    - Weak ownership – good
    - Strong ownership – better
- ODR guarantees
  - An entity is owned by exactly one module
  - Reachability, instead of “redeclaration”
- Allows implementation freedom to enforce ownership
  - Throughout the entire toolchain, from parser to linker
  - MSVC implements a ‘strong ownership’ model.
  - Hope: More C++ toolsets offer this superior model

Weak Module Ownership

```
export module m1;

export
int f() { return 42; }
```

lib_m1.cpp

```
import m1;

int lib_m1() {
    return f();
}
```

```
export module m2;

export
int f() { return 1337; }
```

lib_m2.cpp

```
import m2;

int lib_m2() {
    return f();
}
```

Independently developed libraries

main.cpp

```
int lib_m1();
int lib_m2();

int main() {
    if (lib_m1() != 42) return 1;
    if (lib_m2() != 1337) return 1;
}
```

Linker error: duplicate definition for exported symbols '`f()`', needed by '`lib_m1()`' and '`lib_m2()`'.

Non-compositional semantics

Strong Module Ownership

```cpp
export module m1;

export
int f() { return 42; }
```

lib_m1.cpp

```cpp
import m1;

int lib_m1() {
    return f();
}
```

```cpp
export module m2;

export
int f() { return 1337; }
```

lib_m2.cpp

```cpp
import m2;

int lib_m2() {
    return f();
}
```

main.cpp

```cpp
int lib_m1();
int lib_m2();

int main() {
    if (lib_m1() != 42) return 1;
    if (lib_m2() != 1337) return 1;
}
```

Independently developed libraries

Linker happy: each reference to exported symbol '`f()`', needed by '`lib_m1()`' and '`lib_m2()`' traced back its owner in the context of use.

Compositional semantics

Linker: ownership and dependencies

- Enforce ownership beyond conventional name mangling
- Dynamic initialization follows dependencies as established by modules uses (through *import-declarations*)
- Performance is left on the table when linker *NOT* involved

ODR: front-end responsibility

- Before C++20:
  - Classic ODR detection when using text (defining entities with the same name in the same scope with the same signature is forbidden)
  - Only a single translation unit is considered
  - End of list?...

- Also true for any declaration attached to the global module in C++20

ODR: front-end responsibility

- From C++20 onward: The front-end gets a more complete picture of the program
- The MSVC toolset relies on the Abstract Semantics Graph (ASG) of a TU, persisted in the IFC format, Binary Module Interface (BMI)!
  - ASG contains all meaningful semantics information from the input source file (exports, ownership, dependency, etc)
- Lots of graph-based algorithms apply
  - How to check ODR with modules:

$$A \cap B = \emptyset$$

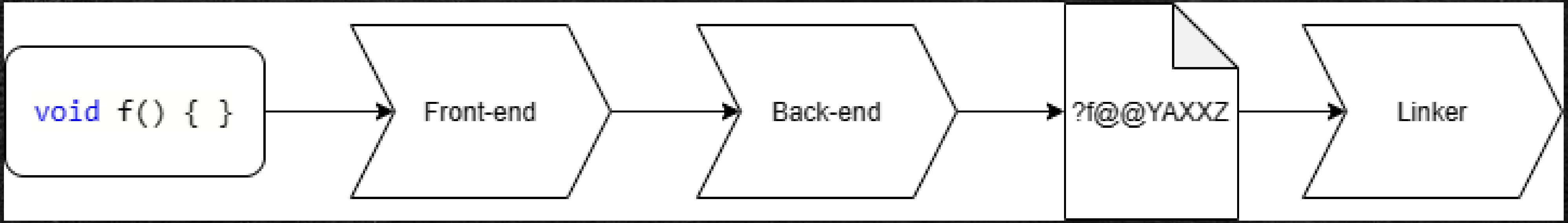
    - Where A and B are the exported entities from two named modules

ODR: linker responsibility

- Before C++20:
  - For internal linkage names the process is simple: only search that symbol
  - The linker must build two sets:
    - References to external linkage symbols
    - Definitions of external linkage symbols
  - For external linkage names the “great hunt” begins…
  - If two definitions of the same external linkage name, issue an error

ODR: linker responsibility

Before C++20



Name	Definition
A	a.obj
B	null
C	c.obj
D	d.obj

ODR: linker responsibility

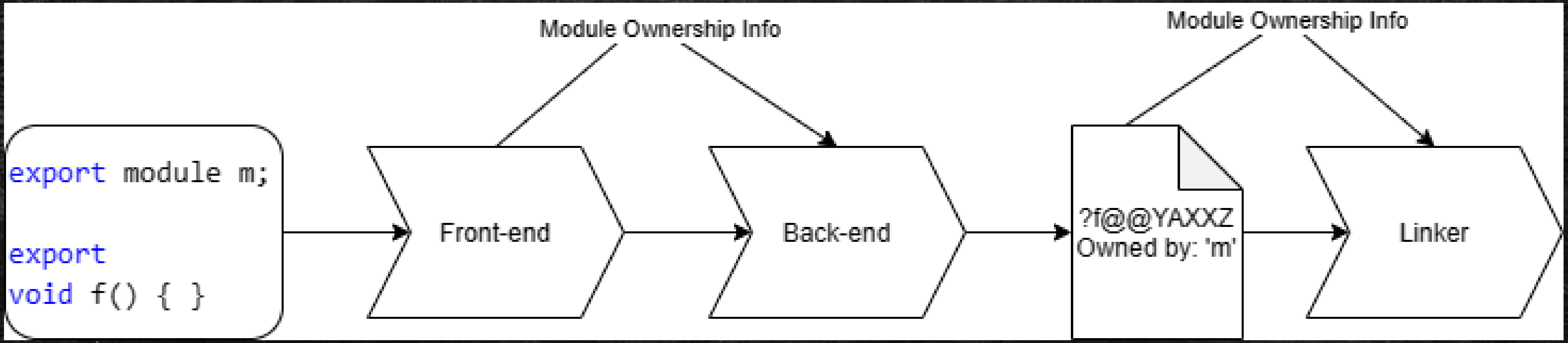
```
app3.obj : error LNK2019: unresolved external symbol _WINRT_GetRestrictedErrorInfo@4 referenced in function "public: __thiscall
winrt::hresult_error::hresult_error(struct winrt::hresult,struct winrt::take_ownership_from_abi_t)"
(??0hresult_error@winrt@@QAE@Uhresult@1@Utake_ownership_from_abi_t@1@@Z)
app3.obj : error LNK2019: unresolved external symbol _WINRT_RoGetActivationFactory@12 referenced in function "struct
winrt::Windows::Foundation::IActivationFactory __cdecl winrt::get_activation_factory<struct
winrt::Windows::Foundation::IActivationFactory>(struct winrt::param::hstring const &)"
(??$get_activation_factory@UIActivationFactory@Foundation@Windows@winrt@@@winrt@@YA?AUIActivationFactory@Foundation@Windows@0@AB
Uhstring@param@0@@Z)
app3.obj : error LNK2019: unresolved external symbol _WINRT_RoInitialize@4 referenced in function "void __cdecl
winrt::init_apartment(enum winrt::apartment_type)" (?init_apartment@winrt@@YAXW4apartment_type@1@@Z)
app3.obj : error LNK2019: unresolved external symbol _WINRT_RoOriginateLanguageException@12 referenced in function "private:
void __thiscall winrt::hresult_error::originate(struct winrt::hresult,void *)"
(?originate@hresult_error@winrt@@AAEXUhresult@2@PAX@Z)
app3.obj : error LNK2019: unresolved external symbol _WINRT_SetRestrictedErrorInfo@4 referenced in function "public: struct
winrt::hresult __thiscall winrt::hresult_error::to_abi(void)const " (?to_abi@hresult_error@winrt@@QBE?AUhresult@2@XZ)
app3.obj : error LNK2019: unresolved external symbol _WINRT_RoGetAgileReference@16 referenced in function "struct
winrt::Windows::Foundation::AsyncActionCompletedHandler __cdecl winrt::impl::make_agile_delegate<struct
winrt::Windows::Foundation::AsyncActionCompletedHandler>(struct winrt::Windows::Foundation::AsyncActionCompletedHandler const
&)"
(??$make_agile_delegate@UAsyncActionCompletedHandler@Foundation@Windows@winrt@@@impl@winrt@@YA?AUAsyncActionCompletedHandler@Fou
ndation@Windows@1@ABU2341@@Z)
app3.obj : error LNK2019: unresolved external symbol _WINRT_CoIncrementMTAUsage@4 referenced in function "struct
winrt::Windows::Foundation::IActivationFactory __cdecl winrt::get_activation_factory<struct
winrt::Windows::Foundation::IActivationFactory>(struct winrt::param::hstring const &)"
(??$get_activation_factory@UIActivationFactory@Foundation@Windows@winrt@@@winrt@@YA?AUIActivationFactory@Foundation@Windows@0@AB
Uhstring@param@0@@Z)
app3.obj : error LNK2019: unresolved external symbol _WINRT_RoTransformError@12 referenced in function
__catch$??$invoke@UAsyncActionCompletedHandler@Foundation@Windows@winrt@@U?$promise_base@Upromise_type@?$coroutine_traits@UIAsyn
cAction@Foundation@Windows@winrt@@$$V@experimental@std@@UIAsyncAction@Foundation@Windows@winrt@@X@impl@4@W4AsyncStatus@234@@impl
@winrt@@YA_NABUAsyncActionCompletedHandler@Foundation@Windows@1@ABU?$promise_base@Upromise_type@?$coroutine_traits@UIAsyncAction
@Foundation@Windows@winrt@@$$V@experimental@std@@UIAsyncAction@Foundation@Windows@winrt@@X@01@ABW4AsyncStatus@341@@Z$0
app3.obj : error LNK2019: unresolved external symbol _WINRT_WindowsCreateString@12 referenced in function "void * __cdecl
winrt::impl::create_string(wchar_t const *,unsigned int)" (?create_string@impl@winrt@@YAPAXPB_WI@Z)
app3.obj : error LNK2019: unresolved external symbol _WINRT_WindowsCreateStringReference@16 referenced in function "private: int
__thiscall winrt::param::hstring::create_string_reference(wchar_t const * const,unsigned int)"
(?create_string_reference@hstring@param@winrt@@AAEHQB_WI@Z)
```

ODR: linker responsibility

- C++20 Onward:
  - For internal linkage names the process is simple: only search that symbol
  - The linker must build two sets:
    - References to external linkage symbols
    - Definitions of external linkage symbols
  - For external linkage names the “great hunt” begins…
  - If two definitions of the same external linkage name, issue an error
  - **The strong ownership model introduces extra bookkeeping**
    - **The linker remembers module provenance of declarations for symbols**
    - **When matching a module-owned external linkage symbol, only considers symbols owned by that module**

ODR: linker responsibility

C++20 and onward



Global module (old extern map)

Name	Definition
A	a.obj
B	null
C	c.obj
D	d.obj

Module ‘m1’

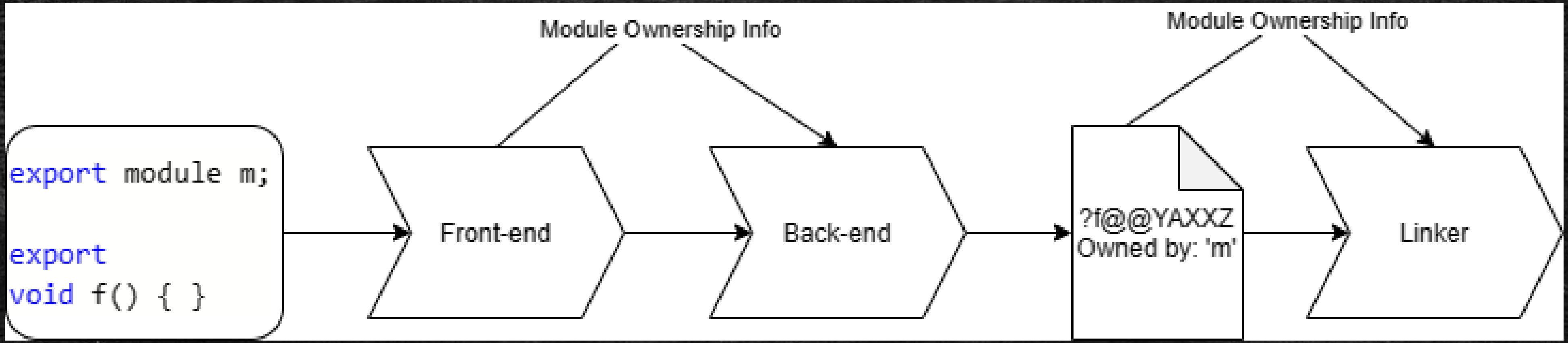
Name	Definition
B	m1.obj
F	m1.obj
C	m1.obj
H	m1.obj

Module ‘m2’

Name	Definition
A	m2.obj
G	m2.obj
H	m2.obj
D	m2.obj

ODR: linker responsibility

C++20 and onward



Global module (old extern map)

Name	Definition
A	a.obj
B	null
C	c.obj
D	d.obj

Module 'm1'

Name	Definition
B	m1.obj
F	m1.obj
C	m1.obj
H	m1.obj

Module 'm2'

Name	Definition
A	m2.obj
G	m2.obj
H	m2.obj
D	m2.obj

ODR: linker responsibility

```
main.obj : error LNK2019: unresolved external symbol "void
__cdecl g(void)" (?g@@YAXXZ) referenced in function _main

main.obj : error LNK2019: unresolved external symbol "void
__cdecl f(void)" (?f@@YAXXZ::<!m>) referenced in function _main

main.exe : fatal error LNK1120: 2 unresolved externals
```

Performance

Compiler Performance

- "You don't pay for what you don't use" –Bjarne Stroustrup
  - The old C++ textual inclusion model goes against this principle
- Headers vs header units vs named modules
- Spectacular improvement in compile-time performance

Headers vs header units vs named modules

Code sample credit to Bjarne Stroustrup: **Minimal module support for the standard library**
https://wg21.link/p2412r0

Tested with Visual Studio 16.11

	#include needed headers	import needed headers	import std	#include “all_std.h”	import “all_std.h”
Hello world (<iostream>)	1.63236s	0.31844s	0.09313s	6.19228s	0.66156s

```
int main()
{
    std::cout << "Hello, World!\n";
}
```

Headers vs header units vs named modules

Using only header units: **5.13x speedup**

	#include needed headers	import needed headers	import std	#include "all_std.h"	import "all_std.h"
Hello world (<iostream>)	1.63236s	0.31844s	0.09313s	6.19228s	0.66156s

```
int main()
{
    std::cout << "Hello, World!\n";
}
```

Headers vs header units vs named modules

Using named modules: **17.55x speedup**

	#include needed headers	import needed headers	import std	#include "all_std.h"	import "all_std.h"
Hello world (<iostream>)	1.63236s	0.31844s	0.09313s	6.19228s	0.66156s

```
int main()
{
    std::cout << "Hello, World!\n";
}
```

Headers vs header units vs named modules

66.51x speedup: **named modules** vs `#include`

7.11x speedup: **named modules** vs **header units**

	#include needed headers	import needed headers	import std	#include “all_std.h”	import “all_std.h”
Hello world (<iostream>)	1.63236s	0.31844s	0.09313s	6.19228s	0.66156s

```
int main()
{
    std::cout << "Hello, World!\n";
}
```

Header units overhead

- Import macros
- Import header include guards (think #pragma once)
- Importing declarations need to be merged with existing ones

You can mix ‘#include’ and ‘import’

- For gradual adoption, consider :

```
#include <vector> // indirectly from some header
import <vector>;  // indirectly from another header

std::vector<int> v;
```

Merging declarations in header units

- The compiler must merge the set of declarations from the textually included header file with the graph stored in the header unit
- The compiler must provably deduce that the existing declarations in the textual <vector> are the same as those in the header unit for the purposes of enforcing ODR guarantees

```
#include <vector>
import <vector>;

std::vector<int> v;
```

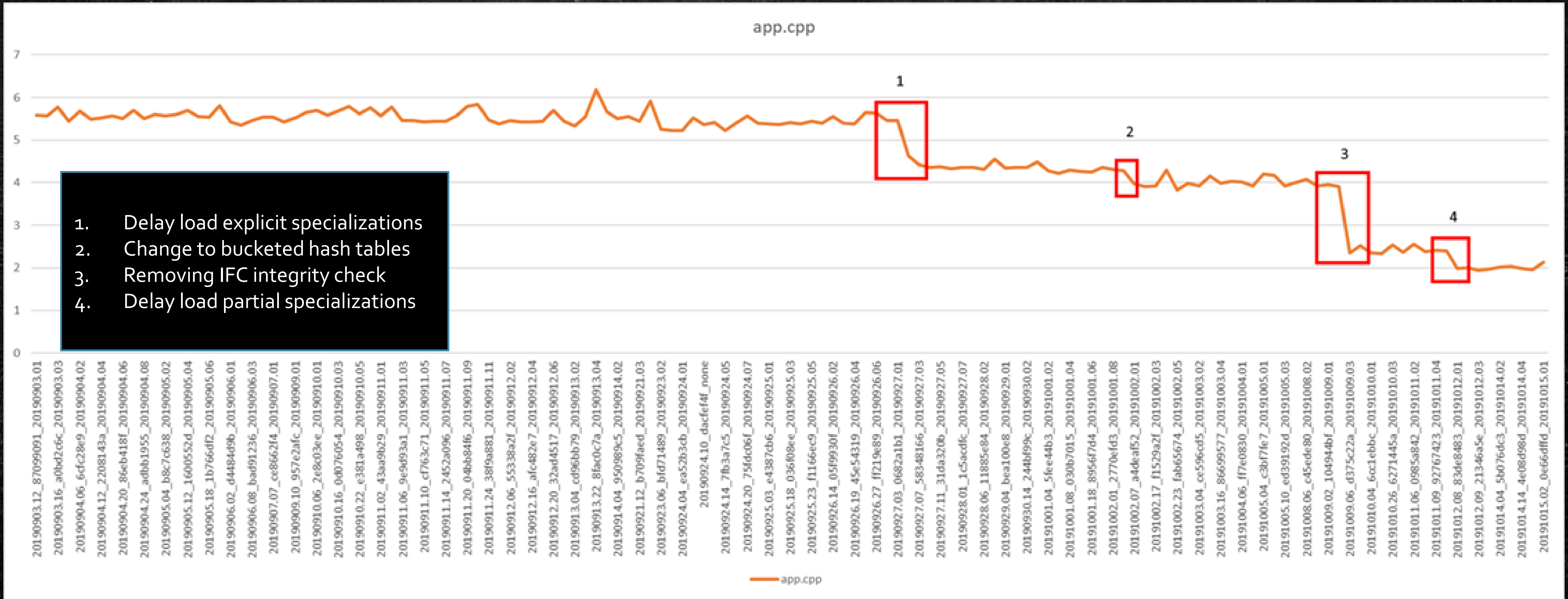
How does the compiler achieve performance?

- Use C++ semantics guarantees to drive materialization
- Use appropriate data structures
- Leverage serialization techniques
- Measure, measure, measure

A short story of MSVC modules performance

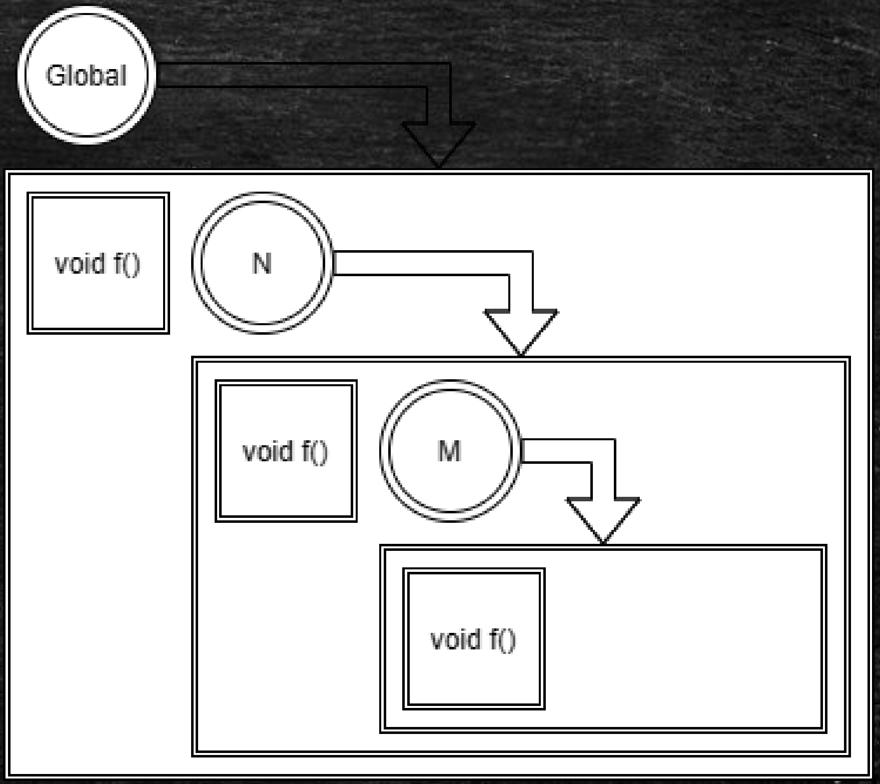
```
#include <cstdio>
import winrt;

using namespace winrt;
using namespace Windows::Foundation;
using namespace Windows::Web::Syndication;

IAsyncAction Sample() // Retrieve an RSS feed and print it.
{
    Uri uri(L"<REDACTED>");
    SyndicationClient client;
    SyndicationFeed feed = co_await client.RetrieveFeedAsync(uri);

    for (auto&& item : feed.Items())
    {
        hstring title = item.Title().Text();
        printf("%ls\n", title.c_str());
    }
}

int main()
{
    init_apartment();   // C++/WinRT startup for multi threaded app.
    Sample().get();     // Block on result.
}
```

A short story of MSVC modules performance



Delay loading: Name population

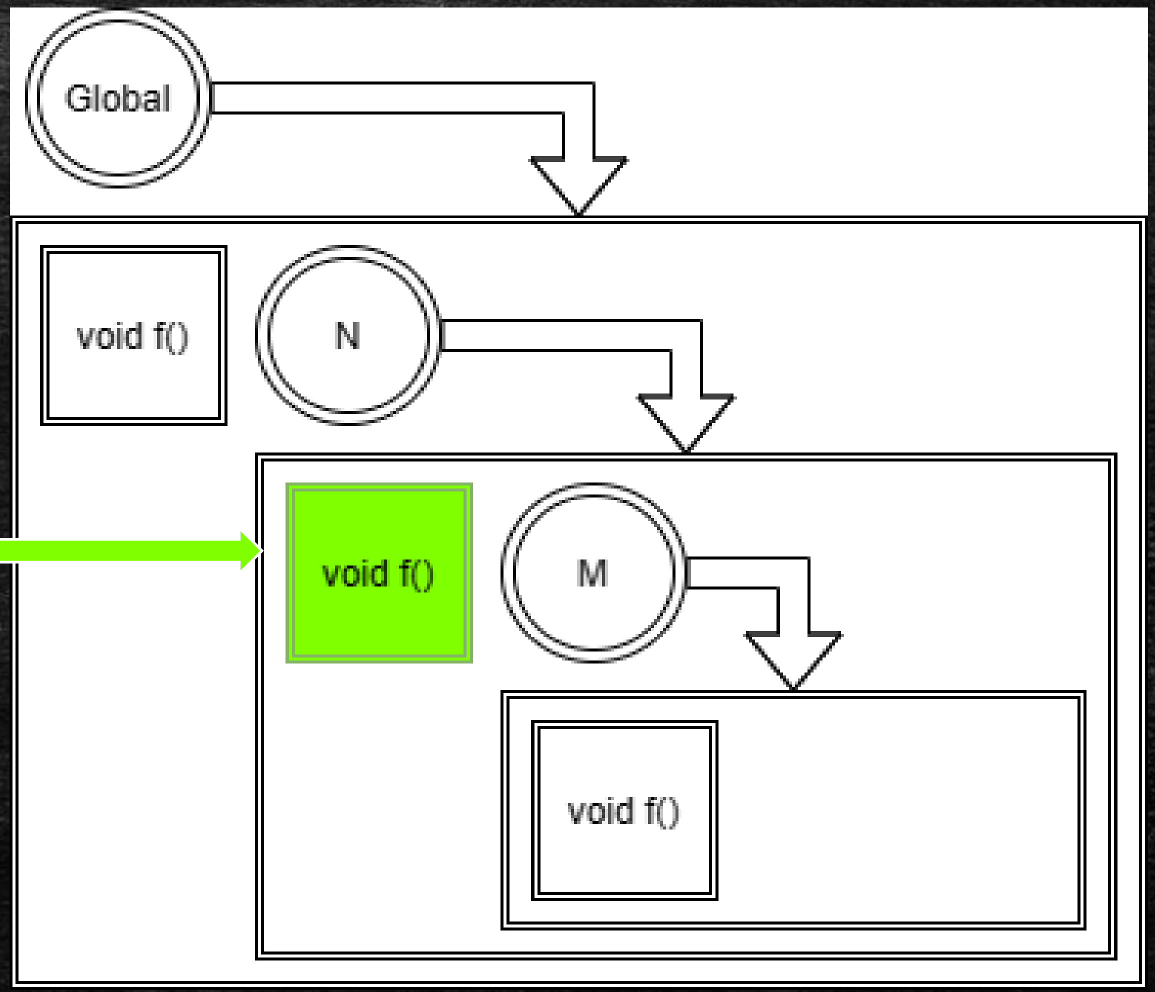
```
export module m;

export
void f();

export
namespace N {
    void f();

    namespace M {
        void f();
    }
}
```



- No other declaration is materialized
- Only names are populated in scopes
- Name lookup drives materialization

Delay loading: Name population

```
import m;

int main() {
    N::f();
}
```



This is what we refer to as **on demand materialization**

Delay loading: template specializations

Delay loading: template specializations

```
template <typename>
struct S;

template <>
struct S<char> { };

template <>
struct S<int> { };

template <>
struct S<float> { };

template <>
struct S<double> { };

S<short> s;
```

Specialization map of 'S'

Argument	Specialization Type
char	Explicit
int	Explicit
float	Explicit
double	Explicit
short	Implicit

Delay loading: template specializations

- For correct C++ semantics the definition of a specialization is not needed unless it is odr-used
  - The compiler is already good about knowing when to instantiate a template
  - We only need to materialize specific explicit specializations when this instantiation happens

Delay loading: template specializations

```
export
template <typename>
struct S;

template <>
struct S<char> { };

template <>
struct S<int> { };

template <>
struct S<float> { };

template <>
struct S<double> { };

S<short> s;
```

Specialization map of ‘S’ when ‘S’ is in a module

Argument	Specialization Type
char	Reserved (ID: 1)
int	Reserved (ID: 2)
float	Reserved (ID: 3)
double	Reserved (ID: 4)
short	Implicit

Delay loading: template specializations

```
import m;

int main() {
    S<char> s1;
    S<short> s2;
}
```

Specialization map of 'S' when 'S' is in a module

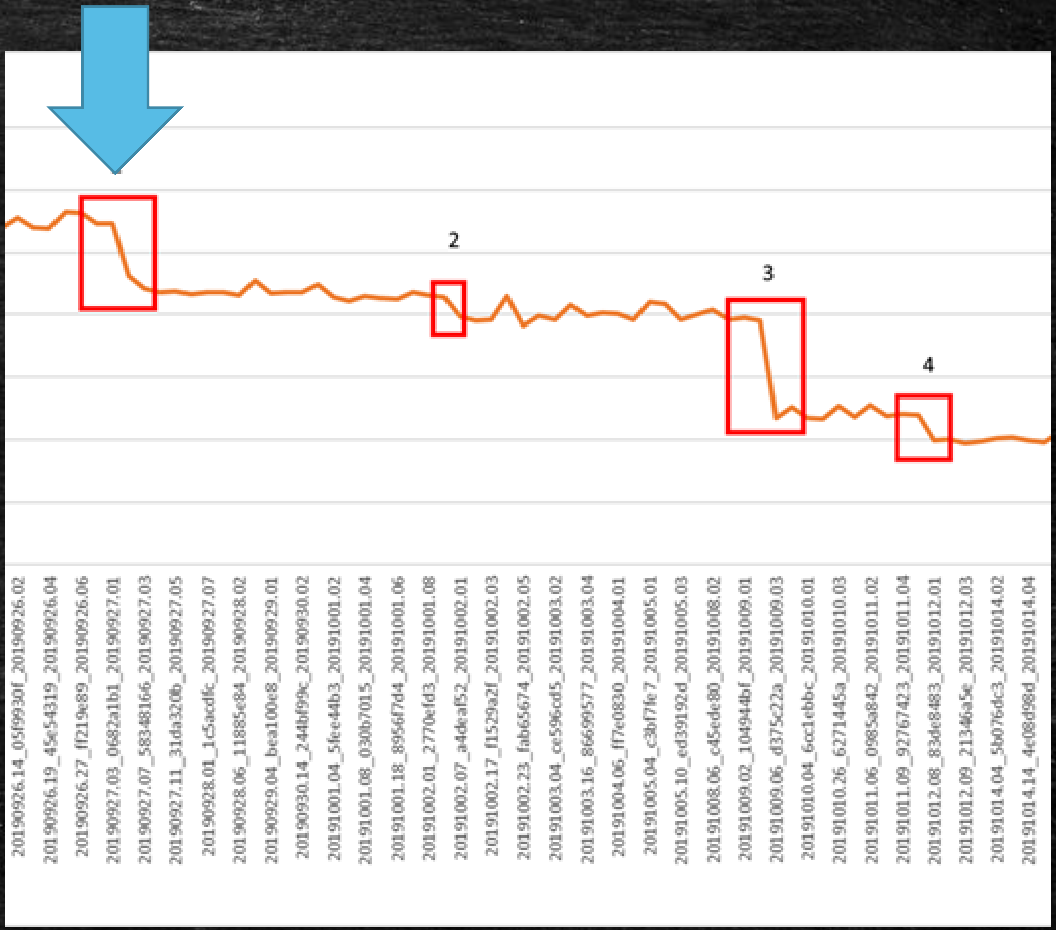
Argument	**Specialization Type**
char	Resolved (ID: 1)
int	Reserved (ID: 2)
float	Reserved (ID: 3)
double	Reserved (ID: 4)
short	Implicit

Note: The data structure which drives the resolution is the Russian Coat Check Algorithm described by Sean Parent in CppCon2019

Delay loading

Using the right data structures

- Measuring compiler performance revealed some things
  - std::map allocates memory in a way that causes cache misses when matching keys
- Let's try using bucketed hash tables…

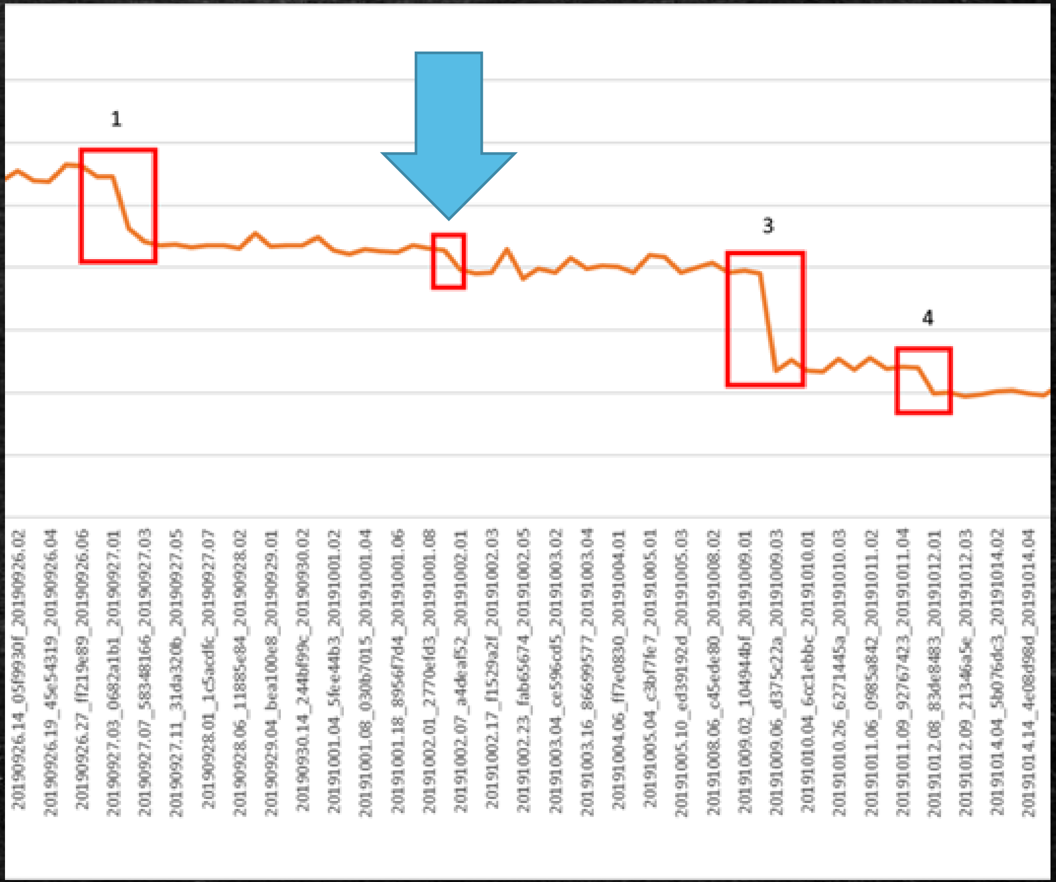


Leverage serialization techniques

- The compiler memory maps the IFC for faster random access
- For verifying the integrity of the IFC the compiler stores a content hash in the IFC and verifies it on import
  - This process implies we need page in the entire IFC on import before doing any real work

Leverage serialization techniques

- What if we turn it off…
- The hash computation takes a tiny fraction of time. **Not paging in the entire IFC is the biggest win.**

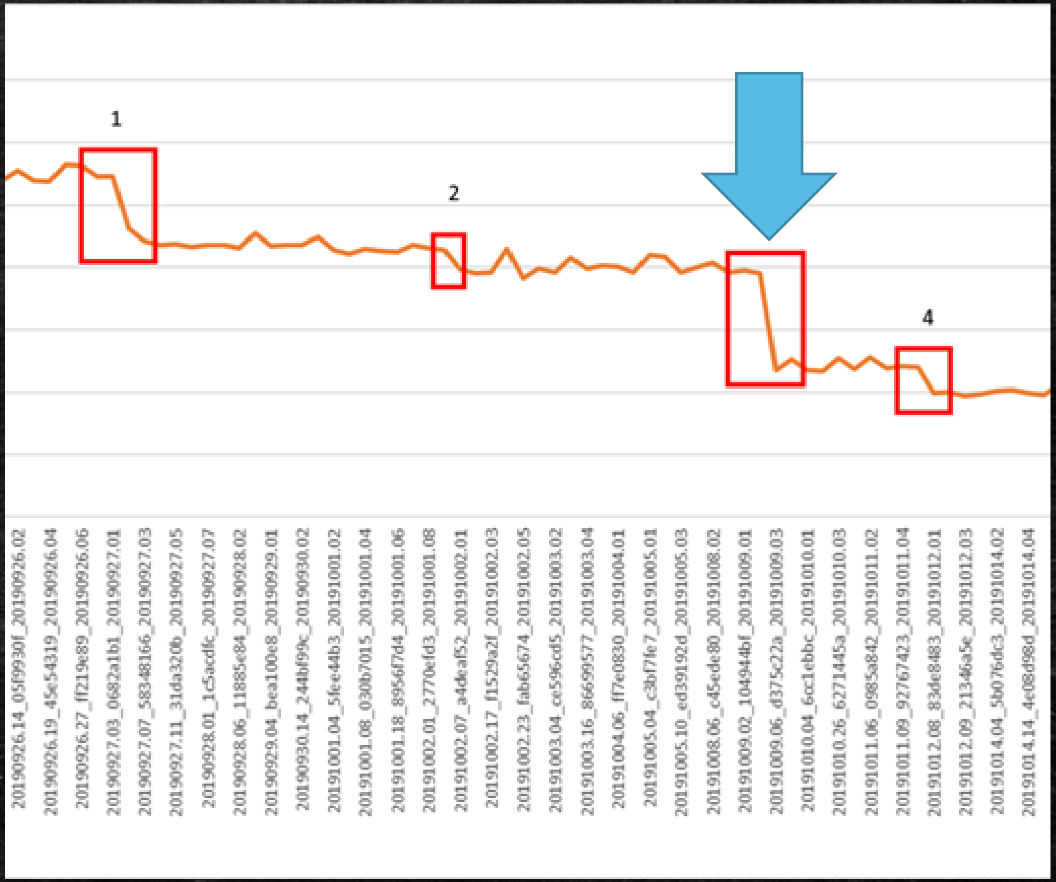


Leverage serialization techniques

- Similar products, e.g. C# ecosystem, opted for a similar strategy due to performance concerns
- The compiler still has an option to validate the contents of the IFC on import: **`/validateIfcChecksum`**
  - Just 1st level verification, not complete security solution
- We still recommend having an option to validate the IFC integrity

Measure, measure, measure

- The mantra of any software engineer
- The final throughput gain was had by finding that partial specializations of class templates dominated the compile time
- Let’s delay them…



IDE Experience

IDE: Behind the scenes

- Sharing a common build module interface (BMI) ensures both compilers agree on C++ semantics
  - Once the IDE understands one BMI if another compiler emits that same BMI you get cross-platform IntelliSense for free
  - Performance wins in IntelliSense engine by reusing work already done by batch compiler
- The two compilers need to agree on BMI versioning
- Dependency computation
  - Compilers need to be involved, see: Format for describing dependencies of source files, By Kitware (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p1689r3.html)

Moving off Precompiled Headers

Implementor Advice: Move off PCHs!!!

- Start with named modules
  - Take the opportunity to practice more code hygiene – you will reap the benefits in performance
  - re-usable across projects
  - Stored in our portable IFC format for analysis purposes
  - provide stronger initialization guarantees, as per the standard
  - MSVC PCHs are notoriously large, IFCs are generally an order of magnitude smaller
- If your project is stuck on header files or in a hurry, try header units
  - Get *some* of the benefits of named modules

More Lessons learned

Adopt modern practice and be flexible

- Don’t use token-stream representation of a translation unit
  - For the first iteration, MSVC used tokens as its primary currency exchange format; switching to an ASG turned out to be a major win in both semantics correctness, and performance
- Don’t use concrete syntax trees
  - They are complex, volatile (change every time WG21 meets)
  - Still require repeated semantics processing
- Don’t be inflexible about existing compiler frameworks
  - Header unit and modules switches went through many iterations as we received feedback
  - Consider going beyond the bare minimum standard conformance: e.g. strong module ownership

Q&A