MODULES

The Beginner's Guide

by Daniela Engert
ABOUT ME

• Diploma degree in electrical engineering
• For 40 years creating computers and software
• For 30 years developing hardware and software in the field of applied digital signal processing
• Novice member of the C++ committee

• Employed by IGM Prüftechnik
PRELUDE

The road towards modules
WHAT IS A PROGRAM?

The compilation model of C++ is inherited from C, and as such half a century old:

- The compiler processes just one **single unit** of program text individually in total **isolation** from all other program texts. This complete unit of program text is called a 'translation unit' (TU).
- The result of compiling a TU is a binary representation of **machine instructions** and associated **meta information** (e.g. symbols, linker instructions). This binary data is called an 'object file'.
- The linker takes all of the generated object files, interprets the meta information and puts all the pieces together into one final **program**. In the world of the C++ standard, the program is then to be executed by the 'abstract machine'. In reality, this is real hardware whose observable behaviour is supposed to be the same.

- If necessary, the preprocessor stitches **multiple** program text **fragments** together to form a complete TU ready for compilation. These code fragments are called 'source files' and 'header files'.
- A program text fragment is the **unit of reuse** in traditional C++.
THE MOTIVATION

Much less a problem in C, due to the nature of C++ and its entities — in particular templates — much larger fractions of program text need to move from source files out into header files before being stitched back before compilation.

```cpp
#include <iostream> // tenths of thousands lines of code hide here!

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

On msvc 19.24, the total number of lines in this program is 53330.
The invention of so called header-only libraries and their popularity emphasize this problem.

```cpp
#include <fmt/format.h>

int main() {
    fmt::print("Hello, world!");
}
```

On msvc 19.24, the total number of lines in this program is 75198.
THE PROBLEM

The duplication of work — the same header files are possibly compiled multiple times and most of the compiled output is later-on thrown away again by the linker — while creating the final program is growing ever more unsustainable.

In case of multiple definitions of the same linker symbol, the linker will decide which one will ultimately end up in the final program. Unfortunately, the linker is totally agnostic of language semantics, has no clue if duplicate symbols implement the same thing, and which one to pick. This choice is up to the implementation without any guarantees.

The difficulties in figuring out the actual meaning of code leads to errors (e.g. violations of the 'one definition rule') and unsatisfactory tooling.
THE OBJECTIVE

Increase efficiency:

- Avoid duplication of work
- Minimize the total effort that a compiler has to put into the creation of a program

Increase effectiveness:

- Make reasoning about pieces of code much less dependent on the context
- Leave less room for accidental mistakes
- Open the path to much richer tooling
ENTER MODULES

The new kid on the block

Coming soon in C++20 — already available today
OVERVIEW

- A bit of history — how it came to be
- Modules 101 — basic concepts
- Modules level 2 — digging deeper
- Mo' Modules — the C++20 additions
- Pitfalls — beware!
- Implementation status — bumpy roads ahead ...
- From header to module — a reality check
- Transitioning to modules
THE HISTORY OF MODULES

how it came to be
TIMELINE

Modules have a history of more than 15 years now:

- **2004**: Daveed Vandevoorde reveals his ideas of modules and sketches a syntax (N1736)
- **2012**: WG21 SG2 (Modules) is formed
- **2012**: Doug Gregor reports about the efforts of implementing modules in clang
- **2014**: Gabriel Dos Reis and his co-authors show their vision of implementing Modules with actual language wording (N4214)
- **2017**: implementations in clang (5.0) and msvc (19.1) become usable
- **2018**: The Modules TS is finalized (N4720)
- **2018**: a competing proposal of syntax and additional features is proposed by Google — the so called ATOM proposal (P0947)
- **2019**: the Modules TS and ATOM merge (the first time that controlled fusion leads to a positive energy gain) (P1103)
- **2019**: the fused Modules proposal is merged into the C++20 committee draft (N4810)
MODULES 101

the basics
**OUR FIRST MODULE**

1. **Declares a module interface unit**
   - The name of this module is multiple parts possible, separated by a dot.
   - It must be a valid identifier.
   - It does not clash with other identifiers.
   - It can only be referred to in module declaration and import declaration.

2. **context sensitive**
   - Not exported = invisible outside the module.
   - Exported = visible outside the module by import.

3. **export module my.first_module;**
   - `int foo(int x) { return x; }`

4. **export int e = 42;**
   - `int bar() { return foo(e); }`

All exported entities have the same definition in all translation units!
export module my.first_module;

int foo(int);

export {
  int e = 42;

  int bar();
}

module my.first_module;

int foo(int x) {
  return x;
}

module my.first_module;

int bar() {
  return foo(e);
}
USING THE MODULE

import module → context sensitive

invisible

visible by import

1. import my.first_module;
2. int main() {
3.   foo(42); // sorry Dave!
4.   e = bar();
5. }

module name valid only in import declaration

does not compile, name 'foo' is not available for lookup

imports are cheap

imports of named modules exhibit only architected side effects

import order is irrelevant
global module fragment

- no declarations
- only preprocessor directives

module declaration without a name

global module default name 'universe'

module purview

# pragma once;

struct S {
  int value = 1;
}

module;
#include <vector>

export module my.first_module;
#include "mod.h"
export
std::vector<int> frob(S);

module;
#include <vector>

export module my.first_module;
#include "mod.h"

module:
1 #pragma once;
2 3 struct S {
4   int value = 1;
5  }

1 module;
2 #include <vector>
3
4 export module my.first_module;
5 #include "mod.h"
6
7 export std::vector<int> frob(S);

1 module;
2 #include <vector>
3
4 module my.first_module;
5 std::vector<int> frob(S s) {
6   return {s.value};
7 }
NAME ISOLATION

---

import my.first_module;
import your.best_stuff;

using namespace A;

int main() {
    return bar() + baz();
}

---

export module my.first_module;

int foo();

export namespace A {
    int bar() {
        return foo();
    }
}

} // namespace A

---

export module your.best_stuff;

int foo();

namespace A {
    export int baz() {
        return foo();
    }
}

} // namespace A

---

name '::foo' is attached to module 'my.first_module', i.e. '::foo@my.first_module', exported name '::A::bar' is attached to the global module namespace name '::A' is attached to the global module

name '::foo' is attached to module 'your.best_stuff', i.e. '::foo@your.best_stuff', exported name '::A::baz' is attached to the global module

no clash

same namespace '::A', exports its name and contents of this namespace part
import my.stuff;

int main(){
  return bar() + beast;
}

export module my.stuff;

import your.stuff;
export import other.stuff;

int foo();

export int bar() {
  return foo() + baz();
}

export module your.stuff;

import other.stuff

export
constexpr int foo(int);

export
constexpr int baz() {
  return foo(beast);
}

export module other.stuff;

export
constexpr int beast = 666;

export module other.stuff;

export
constexpr int beast = 666;
export module mod;

import other;

object file
mod.obj

BMI file
mod.bmi

discarded

none

predefined
commandline,

defaults,
commandline
All kinds of C++ entities can be exported that

- have a name
- have external linkage

Corollary: names with internal linkage or no linkage cannot be exported

Names of namespaces containing export declarations are implicitly exported as well

The export declaration

- must also introduce the name
- or be a using declaration eventually referring to an entity with external linkage
- includes all semantic properties known at this point

An export group must not contain declarations that cannot be exported, e.g. a static_assert or an anonymous namespace
SYNTAX GOTCHAS

Comments and empty lines before module declaration are ok.

```cpp
1 // ok
2
3 export module my.first_module;
```

-DMODULE="module"

```cpp
1 MODULE my.first_module;
```

-DMODULE="module ;"

```cpp
1 MODULE ;
```

-DMODULE="export module"

```cpp
1 MODULE my.first_module;
```

The module declaration must **not** be the result of macro expansion.
FIRST GUIDELINES

• think about taking advantage of the structured module naming scheme
• mirror the subparts of the module name in top-level namespaces
• think about mirroring this in the file system layout and file naming as well
• prefer a modularized standard library over standard library headers

• use #includes within the module purview very carefully — only if you really need to
• never #include standard library headers within the module purview!
MODULES LEVEL 2

digging deeper
[TAKE-OUTS]

- **Visibility of names**: invisible names from a foreign TU become visible by means of export and import.
- **Reachability of declarations**: the collected semantic properties associated with exported entities along the dependency chain of imported module interface units become available to the compiler.
- **module linkage**: like external linkage but applies to declarations in the module purview (mangling).
- **language linkage**: a way to re-attach names to the global module.

```cpp
auto make() {
  // the semantic properties of struct S are reachable
  // from the point of its declaration
  struct S { int i = 0; };
  return S{};
}

static_assert(is_default_constructible_v<decltype(make())>);
```

```cpp
module mine;

extern "C++" int foo(); // external linkage, C++ language linkage, attached to global module
extern "C" int var;     // external linkage, C language linkage, attached to global module

int bar();  // module linkage, C++ language linkage, attached to module 'mine'
int jot;    // module linkage, C++ language linkage, attached to module 'mine'
```
MO’ MODULES
the C++20 additions
Refined types of module TUs specified in the C++20 draft:

- module interface partitions: must be re-exported through the primary module interface unit
- module implementation partitions: these do not implicitly import the primary interface unit
- the private module fragment: available only in case of single-file modules

These module unit types are currently mostly not available in implementations
Header modules are generated from header files without tampering them:

- **compiled** through compilation phases 1 ... 7 like a normal source file
- **all declarations** are implicitly **exported**
- all declarations are attached to the **global module**
- must not contain a module declaration
- do **not** have a module **name**
- **export** their **macro** definitions and deletions from the end of translation phase 4
- **taint** translation phase 4 of the **importing** translation unit beyond the point of the import declaration until the end of the **TU**

The import of an unnamed header file requires special syntax: the BMI of the compiled header module is nominated like an `#include` nominates a header file:

```
1  #include <vector>
2  #include "importable.h"
```

**manual**

```
1  import <vector>;
2  import "importable.h";
```

**no module name**
SANE PRECOMPILED HEADERS

Header modules have the same properties as precompiled headers:

- make all of their declarations visible in the importing TU like the equivalent #includes do
- affect macro definitions in the importing TU like the equivalent #includes do

The benefits of importing header modules over including header files:

- importing compiled declarations is faster — like precompiled headers
- limited isolation: the compilation context is not affected by the point of #include

But beware:

- header guards have no effect on the compilation of a header module!
LIMITATIONS OF HEADER MODULES

Not every header file can be compiled into a header module, only so called "importable headers" can

- the definition of an importable header and the resulting set of importable headers is implementation defined
- the C++ standard guarantees that all standard library headers — but not the wrapped C ones — are importable headers
- other header files can be assumed to be accepted as importable headers by an implementation if they behave reasonably well (e.g. no X-macros)

- Compiling importable headers into header modules requires special incantations of the compiler
- nominating header modules in import declarations with #include syntax require support from the build system to locate the BMIs
The C++ standard allows special support for header modules

- in case of C++ standard standard library headers — and only those —
- #include directives are automatically turned into import declarations nominating the same C++ standard library header

```cpp
typedef int #include <vector>;
#include <string>;
```

- the support of this functionality is an optional feature of an implementation
- if available, it offers immediate benefits without changing the code base at all

😊 this is probably the 😊 feature of header modules 😊
PITFALLS

beware!
THINGS TO TAKE CARE OF

• mistaking module names as namespace names — modules do not establish a namespace
• calls from exported inline function bodies that call into functions with internal linkage that are not available in the importing TU
  ▪ exported inline functions
  ▪ inline member functions from exported classes
• incomplete treatment of module interface units as translation units
  ▪ missing compilation into an object file
  ▪ missing linking the object file into the final program
• precompiled headers or force-included headers (/FI or -include) may no longer be available due to syntactic restrictions

• semantic deviations because of compiler bugs (transitional)
IMPLEMENTATION STATUS

bumpy roads ahead ...
## LANGUAGE / LIBRARY FEATURES

<table>
<thead>
<tr>
<th>Feature</th>
<th>gcc (branch)</th>
<th>clang</th>
<th>msvc</th>
</tr>
</thead>
<tbody>
<tr>
<td>Syntax specification</td>
<td>C++20</td>
<td>&lt;= 8.0: Modules TS</td>
<td>&lt;= 19.23: Modules TS</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&gt;= 9.0: TS and C++20</td>
<td>&gt;= 19.24: C++20 △</td>
</tr>
<tr>
<td>Named modules</td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Module partitions</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Header modules</td>
<td>✗ (undocumented)</td>
<td>✗ (undocumented)</td>
<td>✗ (undocumented)</td>
</tr>
<tr>
<td>Private mod. fragment</td>
<td></td>
<td>✓ (in C++20 mode)</td>
<td></td>
</tr>
<tr>
<td>Name attachment</td>
<td>(✓)</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>#include → import</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>__cpp_modules</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>201810L △</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Modularized std library</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>
Build systems with support for modules are rare

- build2
  - supports clang, gcc, and msvc
  - predefines macro __cpp_modules
- more?
FROM HEADER TO MODULE

A reality check
Use a library from our in-house production codebase and check out what it takes to transform it into a named module.

Library "libGalil":

- wraps and augments a vendor provided C-library that implements low-level network communication with its 'Digital Motion Controller'
- adds higher level functions and error handling on top of it
CONVERTING AN EXISTING LIBRARY

- the OEM provides 2 headers, a link library and a DLL
- we add 4 more headers and 2 source files

The consumable artifacts after compiling the 2 source files are

- a static library
- a single header file with the C++ API

The actual interface is a single class plus some enums within a unique namespace.
The header "DmcDevice.h" exposing the API looks quite unsuspicious ...

```
#pragma once
#include <boost/asio/ts/net.hpp>
#include <boost/filesystem/path.hpp>
#include <boost/signals2.hpp>
#include <meerkat/semaphores.hpp>
#include <atomic>
#include <chrono>
#include <memory>
#include <string>
#include <string_view>
#include <system_error>
#include <vector>

namespace libGalil {

... // some enums used in member function parameters and return values
namespace detail {
... // 2 small classes used as non-static data members in the class below
}

class DmcDevice {
...
};
```
THE MODULE API

Turn this into the primary module interface unit "DmcDevice.cpp" of module "libGalil"

```cpp
module;  // the global module fragment starts here
#include <boost/asio/ts/net.hpp>
#include <boost/filesystem/path.hpp>
#include <boost/signals2.hpp>
...
#include <string>
#include <string_view>
#include <system_error>
#include <vector>
// the global module fragment ends here
export module libGalil;  // the module purview starts here
namespace libGalil {
  // entity 'namespace libGalil' implicitly exported
  export {
    // make enums visible outside of module
    ... // some enums used in member function parameters and return values
  }
  namespace detail {
    // not mentioned anywhere in the exported entities
    ...
    // totally hidden
  }
  export class DmcDevice {
    // make class name visible
    ...
    // and its contents reachable
  }
}
```
Compile the primary module interface just like any other translation unit. Depending on the compiler this might require compiler flags to nudge it to treat the source file as an module interface.

**MSVC 14.24 greeted me with this:**

```
DmcDevice.cpp(295,1): fatal error C1001: An internal error has occurred in the compiler.
DmcDevice.cpp(295,1): fatal error C1001: (compiler file 'mscl.cpp', line 1523)
DmcDevice.cpp(295,1): fatal error C1001: To work around this problem, try simplifying or changing the program near the locations listed above.
...
DmcDevice.cpp(295,1): fatal error C1001: INTERNAL COMPILER ERROR in 'C:\...\CL.exe'
Done building project "libGalil.vcxproj" -- FAILED.
```

At least it processed the source up to the last line, Clang 10 didn't even get that far.
As it turns out, there is some problem in a Boost library that both Boost.Filesystem and Boost.Signals2 depend on.

After modifying two member functions in a functionally identical or equivalent way I got rid of both Boost library includes, and at least MSVC compiles the module interface unit.
As before, transforming the source file "DmcDeviceImpl.cpp" into a module implementation unit is straight-forward:

```cpp
#include "DmcDevice.h"
#include "gclibo.h"
#pragma comment(lib, "gclib")
#include "expected.h"
#include <boost/signals2/signal.hpp>
#include <boost/algorithm/string.hpp>
#include <boost/filesystem.hpp>
#include <boost/filesystem/fstream.hpp>
#include <boost/asio/ts/net.hpp>
#include <fmt/format.h>
#include <meerkat/ping.hpp>
#include <meerkat/makeIpEndpoint.hpp>
#include <algorithm>
#include <cassert>
#include <chrono>
#include <memory>
#include <system_error>

... lots of code
```
just declare the global module fragment and the module itself

```c
module;           // the global module fragment starts here
#pragma comment(lib, "gclib")

#include "expected.h"
#include <boost/signals2/signal.hpp>
#include <boost/algorithm/string.hpp>
#include <boost/filesystem.hpp>
#include <boost/filesystem/fstream.hpp>
#include <boost/asio/ts/net.hpp>
#include <fmt/format.h>
#include <meerkat/ping.hpp>
#include <meerkat/makeIpEndpoint.hpp>
#include <algorithm>
#include <cassert>
#include <chrono>
#include <memory>
#include <system_error>

module libGalil;   // the module purview starts here

... lots of code
```
BUILD THE LIBRARY

DmcDevice.cpp

DmcDevice.obj

DmcDevImpl1.obj

DmcDevImpl1.cpp

DmcDevImpl2.obj

DmcDevImpl2.cpp

libGalil.ifc

libGalil.lib

libGalil.lib
\begin{verbatim}
#include <libGalil/DmcDevice.h>

int main() {
    libGalil::DmcDevice("192.168.55.10");
}
\end{verbatim}

becomes

\begin{verbatim}
import libGalil;

int main() {
    libGalil::DmcDevice("192.168.55.10");
}
\end{verbatim}

The compile time was taken on a Intel Core i7-6700K @ 4 GHz using msvc 19.24.28117, average of 100 compiler invocations after preloading the filesystem caches.

The time shown is the additional time on top of compiling an empty main function.
TRANSITIONING TO MODULES

My Assessment
THE PROS

Immediate benefits:

- improved control over API surfaces, better isolation
- improved control over overload-sets
- improved control over argument dependent lookup
- lower probability of unconscious one-definition rule violations
- no macros intruding from other source code
- no macros escaping into other source code
- smaller compilation context
- better local reasoning about source code

Future benefits:

- new tooling opportunities
- improved compile times, faster development cycles
- increased developer satisfaction
THE CONS

Current roadblocks:

- limited compiler availability
- limited feature support
- limited compiler stability
- major implementation bugs
- mostly missing modularized standard library implementations
- severely limited support by build systems
- hardly any support in IDEs or code editors
- all but missing documentation

Long-term costs:

- longer build-dependency chains
MY RECOMMENDATIONS

• start experimenting with simple examples to get familiar with the new syntax and features
• stick with the feature set as described in the Modules TS (i.e. no module partitions, no header modules) before exploring the less-well supported C++20 module language features
• use the C++20 syntax from the beginning
• use a simple build tool like 'make' (or even simpler) to start with as little friction as possible
• or use a build system with support for modules (e.g. build2) to focus your efforts on learning modules rather than fighting with the build environment
• be prepared to run into problems with your compiler. There will be crashes, miscompiles, or even totally misleading compiler messages
• be resilient to frustration and persevere
• feel happy while becoming confident in using modules as a great language feature to structure your codebase and manage API surfaces
RESOURCES

Papers

• Modules in C++, 2004, Daveed Vandevoorde
• Modules, 2012, Doug Gregor
• A Module System for C++, 2014, Gabriel Dos Reis, Mark Hall, Gor Nishanov
• C++ Modules TS, 2018, Gabriel Dos Reis
• Another take on Modules, 2018, Richard Smith
• Merging Modules, 2019, Richard Smith
• C++20 Draft

Contact

• dani@ngrt.de
• danielae on Slack

Images: Bayeux Tapestry, 11th century, world heritage

source: Wikimedia Commons, public domain
Ceterum censeo ABI esse frangendam
as soon as a named entity is declared within a given scope, it may become subject
to name lookup
name lookup finds only names that are visible, i.e. the name is not hidden
the visibility of a particular named entity is not a static property but the result of

- the point and scope of its first declaration
- the point and scope from where it is looked-up
- the lookup rules
  - regular, unqualified lookup
  - qualified lookup
  - argument dependent lookup

without modules, total invisibility of entities with linkage is impossible
moving declarations from headers into modules makes them totally invisible
exporting names from a module and importing them controls the extent to which
names become visible in the importing translation unit
REACHABILITY OF DECLARATIONS

the reachability of a declaration is orthogonal to the visibility of the declared name

• each visible declaration is also reachable
• not all reachable declarations are also visible

the set of semantic properties associated with a reachable declaration depends on the point within a TU

• after the declaration
• after the definition

when a named entity is exported from a module, then

• the name becomes visible
• the declaration becomes reachable with the set of properties known at this point
• all declarations referred to from the exported declaration become reachable, too!
linkage determines the relationship between named entities within scopes of

- a single translation unit
- multiple translation units

non-modular C++ knows three kinds of linkage

- **no linkage**: entities at function block scope are **not** related to any other entities with the same name. They live in solitude within this scope
- **internal**: entities at namespace or class scope that are **not** related to any entities with the same name in other TUs. There may be multiple of them in the program
- **external**: entities that are **related** to entities with the same name in all other TUs. They are the same thing and there is only one incarnation in the final program

Modules add a fourth kind of linkage

- **module linkage**: effectively the same as external linkage, but confined to TUs of the same module
module linkage

- applies to names attached to named modules
- requires different mangling of linker symbols
- exhibits the same linker behavior as external linkage does to linker symbols from names attached to the global module

therefore, each named module opens a new, separate linker symbol domain

- external-linkage names attached to the global, unnamed module are decorated with no additional name part
- module-linkage names attached to a named module become decorated with an additional name part derived from the module name
C++ knows two kinds of language linkage that apply to function types, functions and variables with external linkage

- C++ language linkage — this is the default
- C language linkage

The language linkage affects (at least) the mangling of external-ish names in addition to that, declarations within a linkage specification in the purview of a module attach the declared names to the global module.
module partition contributing to interface
- must be exported through the primary module interface unit
- may be imported into other TUs of this module

primary module interface
- constitutes the full module interface

partition name must be unique within module

no module name

```
1 export module my.stuff : part;
2
3 export {
4    template <typename T>
5        struct S {
6            S(T x) { /* ... */ }  
7            operator int() const { /* ... */ }  
8        };
9
10    int func(auto x) { return S{x}; }  
11 }
```

```
1 export module my.stuff;
2
3 export import : part;
4
5 export
6 int foo();
```
module partition **not** contributing to interface

- **not** visible outside of module
- may be **imported** into other TUs of this module
- do **not implicitly** import the module interface
- may **replace** module-internal `#includes`

```
1 module my.stuff : forward;
2
3 template <typename T>
4 struct S;
```

```
1 module my.stuff : impl;
2
3 import : forward;
4
5 template <typename T>
6 struct S {
7     S(T x) { /* ... */ }
8     operator int() const { /* ... */ }
9 };
```
module interface partition

- must be the only, single translation unit of this module

private module fragment

- no exports
- definitions not reachable from outside of module

partition name must be 'private'
Add a shim header "DmcDevice.h" like this after renaming the 'old' header file:

```cpp
#include <libGalil/DmcDevice.h>

int main() {
    libGalil::DmcDevice("192.168.55.10");
}
```

Now, all users of the library will benefit from modularization as soon as their compiler is capable of C++ modules even without any code changes.

```cpp
#pragma once
#if __cpp_modules >= 201907
import libGalil;
#else
#include "DmcDevice.hh"
#endif
```