Modules, Componentization, and Transition

Gabriel Dos Reis Pavel Curtis
Microsoft

Abstract

We provide an analysis of constraints for a good, acceptable, and scalable module system for modern C++. This analysis is based on decades of practical experience with precompiled headers, and 40+ years of the include-file model, which has shown its limits. The paper also discusses several migration strategies. The end goal is to stimulate a technical discussion about the difficult choices we face in bringing C++'s compilation model into the era of semantics-aware developer tools, and of smart distributed and cloud build systems.

1 INTRODUCTION

"Modules" rank near the top of frequently requested features for C++, in particular C++17. By far, a common expectation is programmer productivity, not only by ameliorating the "inner loop" (edit-build-test), but also and primarily by bringing semantic structures that foster development tools like those enjoyed by programmers using programming languages featuring first-class "module" constructs (e.g. C#, Java, Ada, Python, etc.)

The current "module" proposal [1] being considered by the committee lists four fundamental goals:

1. Componentization
2. Isolation from macros
3. Scalable build
4. Semantics-aware developer tools

These goals are considered key to productivity.

Componentization is essential for any module system for C++ in order to deliver expected productivity at scale. Ideally, there should be a sufficiently simple and direct linguistic support for programmers to express

a) Component boundaries: what is consumable from outside vs. what is internal to a component
b) Dependencies on components: what other components are required

Furthermore, we suspect that without componentization, a module system for C++ that aims only for "scalable build" would essentially be a minor variation of "Precompiled Headers" (PCHs). Three decades

of experience with PCHs in heterogeneous environments and billions lines of code strongly suggest they are not the module system we have been looking for.

Macro isolation is another essential ingredient for programming at large. Macros make it hard, if not impossible, for tools to "understand" program source code. It is far too easy to underestimate how much drag they constitute on development tools and developer productivity. These problems have been extensively documented [2] [3] [4]. The take-away from previous attempts at limiting the reach of macros is that any modification to the preprocessor is bound to fragment the community: some would consider any improvement to the preprocessor goes too far, and others not far enough. Furthermore, for obvious compatibility reasons, any such improvement can only add to the existing mess.

A good module system for C++ should live up to the C++ tenet of zero-overhead abstraction: you don't pay for what you do not use. The current compilation model, based on header files, actively violates that principle since a preprocessor `#include` directive is essentially a textual copy-and-paste directive, ensuring that the whole contents of a source file is repeatedly processed, regardless of which entity is actually used. Ideally, an entity from a consumed component should not need (re)processing if it does not affect the semantics of a program.

Finally, modules should support the flourishing of tools that "understand" source code and development tools such as refactoring, dependency tracking, etc. A distributed build system can use semantics information from components to decide whether a source file need recompilation, or whether it can be shared across build instances.

In summary, goals (1), (2) and (4) can be seen as both leading directly to, and being necessary for (3).

2 COMPILATION MODEL

Processing a C++ source file is formally divided into 9 translation phases:

i. Translation phases 1 through 4 comprise what is traditionally referred to as *preprocessing*. These phases have no knowledge of the core C++ language rules, e.g. lexing, parsing, scoping, etc. No preprocessing construct survives beyond this stage.
ii. Translation phases 5 through 7 deal with lexical, grammatical, and static semantics rules. During these semantics phases, the preprocessor is no longer called upon in any observable way.
iii. Translation phase 8 is colloquially known as the "template instantiation phase", since it deals with producing and analyzing required template specializations.
iv. Finally, translation phase 9 is the linking phase where all translation units and instantiation units making up a program are combined and external symbol references are resolved.

This compilation model is ingrained in day-to-day programming tasks and tools. Ideally, an acceptable module system should not disturb this subdivision of phases, so that it does not do violence to developer tools.

It should be noted that after the preprocessing phase, **programmers are guaranteed that there are no traces of preprocessing constructs left that will come and bite them when they least expect it**. E.g., it is common for a developer investigating a bug, or for analysis tools, to request preprocessed source files,

e.g. with no preprocessor dependency. Ideally, an acceptable module system for C++ should maintain these expectations. For example, if a code fragment such as

```
import M;
```

revives prior preprocessing states (e.g. when `M` was compiled), then it would violate the spirit of, and the expectation that "the preprocessor is never revived by the lexer or parser."

This implies that the interpretation of a module 'import' directive, which is and must be a construct from the lexical/grammatical/semantic level, should not have an impact on the preprocessing phases (e.g., by introducing new macro definitions): the preprocessor should not know about the 'import' directive, and it cannot interpret the directive itself, so it similarly cannot change its macro-definition environment in response to the appearance of that directive. Said simply: importing a module should not introduce new preprocessor-macro definitions or change existing ones.

3 Living With History

If we can't modify the preprocessor, if we can't modify the language to bring fundamental preprocessor constructs (such as macro expansion) to the semantics processing phase, then how do we handle existing massive lines of code? Does a good and acceptable module system for C++ have to be "all-or-nothing"? Is there any transition story? Is it top-down? Is it bottom up?

Well, first not all existing code are immutable. Some will evolve, others won't for various reasons. For the codes that can evolve (e.g., because they are under active development), the transition must be toolable. That is, the module system should lend itself to an automatic or semi-automatic "modularization" process for scalable adoption.

For the codes that cannot be changed or that cannot evolve, we must be willing to consider extra-linguistic solutions, possibly ones that are part of the build setup, if we want to bring *some* of the benefits of modules to those unchanging codes, or just accept that they will forever be consumed via preprocessing constructs.

So, how bad is it?

4 A Tale of Two Implementations

To this date, there are two ongoing, experimental implementations of module systems for C++. One is in Clang, and the other in Microsoft Visual C++ Dev 14 Update 1. What follows is a brief introduction and analysis of both systems.

4.1 Clang Modules

The authoritative reference on "modules in Clang" at the time of this writing is the documentation available at [5]. They were originally developed for Objective-C (and Clang's implementation of C), later extended to C++. The basic idea is to express how a header file may be thought of as a module. Presently, there is no C++ source-level construct for defining a module or using modules in Clang. Rather, the Clang team has consistently expressed their desire to track WG21's work on the subject. Clang's module

semantics is achieved by altering the sequence of standard translation phases when the preprocessing directive #include is encountered in the input source code, by automatically mapping certain header files to "modules". The altered behavior is accomplished by authoring a sort of "configuration file" (not a C++ source file) named `module.modulemap` placed alongside the set of header files intended to be treated as modules. Here is a simplified example from [5]:

```
module std {
	module assert {
		textual header "assert.h"
		header "bits/assert-decls.h"
		export *
	}

	module complex {
		header "complex.h"
		export *
	}
	// …
}
```

This states that a certain set of header files are to be treated specially by the compiler when an `#include` preprocessor directive nominates them. In particular, an inclusion of "`assert.h`" will always result in textual copy-and-paste (i.e. the standard copy-and-paste semantics) of the contents of `assert.h`. On the other hand, given a directive including "`complex.h`", the compiler will first look for an existing pre-compiled version of that header file. If present, its compiled contents will be made available to the translation unit requesting the inclusion. In case of no existing precompiled version of "`complex.h`", the compiler starts a fresh instance of itself, compiles "`complex.h`", caches the result for later reuse, and continues processing as if that header had already been compiled. Unlike the "`assert.h`" case (where a textual inclusion is always performed), the "`complex.h`" follows different translation rules. In particular, macros defined in such "modular" header files are subject to the following rules [5]:

- Each definition and undefinition of a macro is considered to be a distinct entity.
- Such entities are *visible* if they are from the current submodule or translation unit, or if they were exported from a submodule that has been imported.
- A `#define X` or `#undef X` directive *overrides* all definitions of `X` that are visible at the point of the directive.
- A `#define` or `#undef` directive is *active* if it is visible and no visible directive overrides it.
- A set of macro directives is consistent if it consists of only `#undef` directives, or if all #define directives in the set define the macro name to the same sequence of tokens (following the usual rules for macro redefinitions).
- If a macro name is used and the set of active directives is not consistent, the program is ill-formed. Otherwise, the (unique) meaning of the macro name is used.

These rules are illustrated in the reference documentation [5] with the following example: consider that

- `<stdio.h>` defines a macro `getc` (and exports its `#define`)
- `<cstdio>` imports the `<stdio.h>` module and undefines the macro (and exports its `#undef`)

then the `#undef` overrides the `#define`, and a source file that imports both modules *in any order* will not see `getc` defined as macro.

Module.modulemap essentially provides a form of *macro visibility* management.

4.2 VISUAL C++ 2015 UPDATES

Starting with Update 1 of its 2015 release, Visual C++ features an ongoing implementation of the module proposal presented in [1]. That design leaves the standard phases of translation untouched, therefore allowing existing development tools and workflow to continue working along common expectations. On the other hand, it introduces into the core language direct support for:

- Expressing component boundaries through module declarations and "export declarations"
- Expressing component dependencies through "import directives"

A module declaration identifies, in the source file, the module to which a given translation unit belongs. An export declaration states that a given entity is accessible from the outside of a module. The set of export declarations in a module form the interface of that module. An import statement makes available to the current translation unit all the entities exported by the imported module. Modules are isolated from macros defined in imported modules. That is, a macro defined in an imported module does not affect the importing translation unit; conversely, a macro defined in a translation unit does not affect the module it imports or their interfaces. Furthermore, that proposal has no provision for exporting macros from modules.

4.2.1 Producing and Consuming Module Interfaces

Defining, compiling, and consuming a module is simple. For example, assuming a C++ source file `src.ixx` contains the following

```
import Calendar.Months;          // for Chrono::Month
import Calendar.Days;            // for Chrono::Day
module Calendar.Dates;           // module Calendar.Dates follows
namespace Chrono {
       export struct Date {
         Date(Day, Month, int);
          // …
       };
}
```

When this module unit is compiled invoking the compiler as

```
cl.exe –module src.ixx
```

the compiler processes the source file producing an object file `src.obj` (as usual) and a file named `Calendar.Dates.ifc`. The latter, called an IFC file, is a binary file describing the full semantic graph of the interface of `Caldendar.Dates`. That is, the content of that file represents the result of compiling all declarations exported from the module `Calendar.Dates`. Every (compiled) module has an IFC file. It is the only compilation artifact necessary to consume a module. When the compiler sees an import declaration, it locates the corresponding IFC file, loads its contents, and makes the interface available for consumption. Note that the IFC file need not be stored in a file named after the module. An IFC can be embedded in any binary file, including in a "static library archive" hence providing a coherent "self-descriptive" component with a well-defined boundary and good isolation. IFCs can be grouped together

with other build artifacts for package delivery purposes; however, "packaging" and "package delivery" are outside the scope of modules proper and of the current module proposal.

An IFC is completely toolable. That is, it can be inspected for discovery of exported entities; in particular, it does not require a C++ compiler once produced. Similarly, an IFC can be edited without requiring a C++ compiler. This is obviously useful for "smart" build systems and other development tools, including IDEs, debuggers, runtime systems, etc. Note that an IFC is not part of an executable program: its contents are largely logical and contain no executable code. This leads to various scenarios for transition and for coping with "legacy" codes.

4.2.2 Bridging Old and New

An existing header file `header.h` can be made consumable as a module (i.e. via an import declaration, complete with macro isolation benefits) as follows

i. Make a C++ source file (say src.cxx) comprising of just `#include "header.h"`
ii. Compile that source with the command line

```
cl.exe –module:export src.cxx –module:name MyModule
```

This has the effect of producing an IFC for a module named `MyModule` whose interface is the set of all namespace-scope declarations with external linkage.

This technique can be used to produce IFCs allowing consumption of "legacy" components (that will never evolve to modules) from codes using modules. It is amenable, with further tooling, to automatic or semi-automatic migration to modularization for codes that desire to evolve.

For legacy codes that produce interfaces via macros, there is an additional compiler switch to retain a macro state at the end of the "module translation unit" and emit it in a wrapper header file for the consumer to use. For example

```
cl.exe –module:export src.cxx –module:name MyModule –module:wrapper new-
header.h –module:wrapperMacros macro-file
```

produces the IFC file `MyModule.ifc` along with a wrapper header file `new-header.h`. This wrapper header file essentially contains an import declaration nominating `MyModule`, followed by a series of preprocessing directives defining all macros listed in the file macro-file that are active at the end of the translation unit `src.cxx`. This way, the consumer can just include "`new-header.h`": it will benefit from module semantics and a certain level of macro isolation as specified. Furthermore, this solution retains the traditional separation of preprocessor constructs and core language constructs.

5 CONCLUSION

Componentization, macro isolation, semantics-aware development tools are essential ingredients, just as important as achieving scalable build throughput for modern C++. In search of transitional scenarios for large scale adoption, it is essential not to introduce new constructs in the language that will further compromise macro isolation. This document outlined such a scenario for the module proposal [1] as being

implemented in the Visual C++ compiler. This approach maintains the standard phases of translation and the principle that macros are introduced by preprocessor constructs, not language feature, all while introducing no new preprocessing directives.

6 REFERENCES

[1] G. Dos Reis, M. Hall and G. Nishanov, "A Module System for C++ (Revision 4)," 2015.

[2] B. Stroustrup, "The Design and Evolution of C++," Addison-Wesley, 1994.

[3] B. Stroustrup, "#scope: A simple scoping mechanism for the C/C++ preprocessor," 2004.

[4] T. Plum, "The "scope" extension for the C/C++ preprocessor," 2004.

[5] Clang Modules, "Clang 3.8 Documentation," 03 October 2015. [Online]. Available: http://clang.llvm.org/docs/Modules.html.