Doc. no.: P0955r0
Date: 2018-02-11
Programming Language C++
Audience: EWG
Reply to: Bjarne Stroustrup (bs@ms.com)

Modules and macros

Bjarne Stroustrup

Abstract

My main reasons for disliking macros (that is C/C++-style macros) are

- They don't obey scope and type rules
- They make what the human programmer sees different from what the compiler proper sees
- They seriously constrain tool building

That last reason is why C and C++ tools have lagged those of other languages, often by decades, and still lag in feature sets, quality, and cost. This hindrance is unfortunately easily underestimated and misunderstood.

These problems could easily compromise the value of modules in C++. I therefore argue that we should treat macros and modules as orthogonal (for some definition of orthogonal) and aim to limit the negative impact of macros by discouraging their use. Modules offer a historic opportunity to improve code hygiene and compile times for C++ (bringing C++ into the 21st century). A module design that did not offer modularity would most likely squander that opportunity forever.

1. Modularity

What do I mean by modularity?

Order independence:

```
import X;
import Y;
```

should be the same as

```
import Y;
import X;
```

In other words, nothing can implicitly "leak" from one module into another. That's one key problem with **#include** files. Just about anything in an **#include** can affect any subsequent **#include**. In particular,

- macros leak out of an **#include**d file, potentially changing the syntax of subsequent code

- type names leak out of an **#include**d file, potentially changing the meaning of declarations in subsequent code
- function declarations leak out of an **#include**d file, potentially changing the resolution of overloads in subsequent code

Such problems are the ones we need to solve for reasons of code hygiene. If that ``subsequent code'' is ours we want those effects (that's presumably why we **#include**d, though nasty surprises are not uncommon), but not if the ``subsequent code'' is another unrelated **#include**.

The Modules TS elegantly avoid such problems by guaranteeing that

- nothing from an **import**ed module is used unless it is mentioned in the **import**ing code
- nothing from the **import**ing code can modify the meaning of code in the **import**ed module

That is,

- **import**ing a module simply makes code available for composition (and not modification)
- the definition of a module interface is determined exclusively in the one place where it is defined

In addition, the Modules TS offers a necessary related modularity property:

- Nothing is implicitly **export**ed from a module

Had this not been the case, code doing **import M** would be exposed to code that was used in the implementation of **M**. Thus, modularity would be compromised by exposing an **import**ing module to essentially arbitrary code. This of what is dragged in by **#include <windows.h>**. This property is essential for limiting dependencies on a large system (such as an operating system or a graphics system) to specific, well-specified sub-sets presented as modules.

2. Compile-time improvements

Most people assume that having modules will improve compile times significantly (factors rather than just percents). After all, many languages with modules see orders of magnitude better compile-time performance than C++. However, such speedups are not trivial to achieve and not free in terms of what language features you can use while getting them. As an aside, I can point out that back in 1976, I modularized a Simula program, just to find that compiling a program using precompiled modules was significantly slower than compiling the whole source at once. Others have had the same surprise with many different languages and module systems up to today.

Modularity offers such massive improvements, but only if modules are truly independent and only if importing a module is a very cheap operation. Where I have seen slow module systems, the problem has been that importing a module was an expensive operation. The reason could be that importing a module could essentially amount to recompilation of that module's source code or that importing involved entering the imported constructs into the scope of into the importing scope (whether they were used or not) or both.

3. Requirements

For a module system to provide modularity, it must

- Provide the code hygiene improvements outlined in §1
- Provide the significant compile-time improvements outlined in §2
- Allow for existing code to be compiled unchanged and mixed with code compiled using modules

The last requirement is necessary to allow gradual conversion of existing systems. This is offered by the Modules TS [N4720]: we can use **#include** as ever both in the implementation of modules and side-by-side with **import**s.

4. Representing modules

The obvious representation of a C++ program is as a typed DAG (sometimes called an AST, though it is not abstract, not about syntax, and not a tree), such as the IPR [GDR,2011]. Since this has been done, we know that this can lead to a compact representation that is fast and reasonably easy to use. We can assume that roughly equivalent approaches yield similar benefits. Here, I use the term used in the modules TS: “ASG” (“Abstract Semantic Graph”). The IPR is a concrete example of an ASG.

The problem is macros. An IPR represent a program as the compiler sees it; that is, post-preprocessing.

Macros used internally to a module implementation is not a serious problem. We can do

```
module;
#include "nasty.h"
// … module implementation …
```

To our hearts content with only the usual problems with macros.

The problem is macros that must be consistent across uses of modules

file 1:

```
#define Foo 1
// .. define module M …
```

file 2:

```
#define Foo 2
import M;
// … use M and Foo …
```

For a simple, realistic example, you can think of **Foo** as **NDBUG**. To maintain modularity, we cannot have **Foo** in **file 2** affect the implementation of **Foo**, so the code in **file 2** and in **M** see different values of **Foo**. This can easily lead to disaster. For example, a struct used in the implementation of **M** and in the code in **file 2** (e.g., **std::vector**) may have different implementations depending on the value of **Foo**.

I see three ways of handling such “non-encapsulated” macros:

1. Ban them. This is infeasible for **NDEBUG**, but we could make such inconsistent use UB. It already is in other contexts.

2. Keep a pre-processed copy a module for each set of macros used in a module by defined outside it. Use “the right copy” depending on the macro definitions at the call point.
3. Add nodes representing macros to the IPR.

Approach [1] is my favorite. It is simple and forward-looking. It does not complicate our module representations or slow down compilations (unless we want to check whether macros are used consistently, which would be novel and probably relatively cheap).

Approach [2] would most likely require the module be a simple textual **.cpp** source file plus a cache mechanism. We have decades experience with that general kind of scheme in the form of preprocessed headers. Module definitions would be large and then number of copies for different macro values could become significant. Compiling from the textural form is expensive. I don’t see this approach delivering compile-time improvements. On the contrary, unless we spend a lot of effort, it will slow down builds.

Approach [3] would complicate the IPR and would break down for macros that “messes” too much with syntax, making this approach degenerate into approach [2] and having to repeatedly compile from “token soup.”

Consider examples of what I am thinking of

```
void f(int x
#ifdef FOO
, double d
#endif
);
```

and

```
#fdef BAR
{ preamble(x);
#endif
// …
#ifdef BAR
postamble(x,y); }
#endif
```

Yes, I have seen such, and worse, in real code. Naturally, some of you would quite reasonably say, “so don’t do that!” but a tool builder must build for the worst case. The mere possibility of such horrors drags down the sophistication and performance of our tools (incl. compilers) to the lowest level.

Could we compromise by allowing only “well behaved macros?” Doing so would antagonize people who like complicated macros and we’d have a hard time defining “well-behaved”. I suspect any useful definition of “well-behaved macro” would be roughly equivalent to can be implemented using non-macro C++ facilities.” If so, focusing on providing tools for converting macros is a much more promising approach (for an experiment, see [Kumar,2011]).

Approaches [2] and [3] both offers the possibility of an exponential explosion of alternatives/copies as the number of macros increase.

5. Exporting macros

There has been much talk about exporting macros. This would imply that the rules for defining and using macros and modules would be intertwined. That would be a major mistake. Macros are fundamentally different from proper language rules. Consider

```
import M;
// … code here …
```

Can a macro exported from M unconditionally affect the following code? If so, importing would no longer be an essentially free operation? If not, how would the compiler know when to look into the module to find a macro? Unless syntax-modifying macros were banned, it would have to be during before syntax checking, but that implies that we always have to look into the imported module.

Assuming that we would not want to export all macros defined within a module, the preprocessor would have to be modified to understand about exported macros (macro names are not known after current pre-processing). Modifying the preprocessor has traditionally been fraught with problems, including C compatibility problems. The problem of what to do with macros from the scope surrounding a module definition would re-emerge in the form of: Can a module re-export a macro? Again, that would require a pre-processor change as well as a change to the module semantics.

Consider again order independence:

```
import X;
import Y;
```

should be the same as

```
import Y;
import X;
```

But if modules could export macros, then this can no longer be true. For starters, module X might export

```
#define Y Foo
```

And module Y might export

```
#define X Bar
```

Thus, exporting macros from modules destroys modularity.

6. Conclusion

To get modularity, don’t introduce special rules for macros except explicitly deeming the use of a macro with different values in the implementation of a module and in the context where the module is imported UB.

7. References

- [N4720] Modules TS. N4720 for the Working Draft.

- [Kumar,2011] A. Kumar, A. Sutton, and B Stroustrup: Rejuvenating Macros as C++11 Declarations. Proc. 28th IEEE International Conference on Software Maintenance. September 2012. [GDR,2011] Gabriel Dos Reis and Bjarne Stroustrup: A Principled, Complete, and Efficient Representation of C++. Journal of Mathematics in Computer Science Volume 5, Issue 3 (2011), Page 335-356 doi:10.1007/s11786-011-0094-1.