Doc. no.: P1373R0
Date: 2018-11-20
Programming Language C++
Audience: EWG
Reply to: Bjarne Stroustrup (bs@ms.com)

Syntax alternatives for modules

Bjarne Stroustrup

Abstract

This note examines eight approaches to the problem of the "module" keyword from the module proposal clashing with uses of "module" as an identifier in existing code. Ideally, we keep "module" and "import" as keywords, but if that is not feasible given existing code bases, we have alternatives. This note outlines eight alternative solutions and observes that all but two resolutions would probably be viable. It recommends this alternative if we can't use conventional keywords:

> In the global scope (only) the identifiers "module" and "import" are taken as keywords (representing their meaning for modules) when they appear as the first token of a declaration and not before a ::.

Much of this note was motivated by the -ext reflector thread on this topic and it uses quotes from that.

PS. This paper is posted just for the record. The issue was settled at the San Diego meeting in favor of a variant of the recommended solution implemented by Nathan Sidwell.

1. Introduction

A lot of people use "module" as an identifier:

> We have the data from ([c++std-ext-15855] Largest Open Source C++ Projects). Information as of January of 2016. Thank you Andrew Tomazos!!!
>
> This is based on tokens, i.e. not mentions in strings or comments.
>
> Out of 4,689,316,529 tokens there were the following numbers of occurrences of each identifier token:
>
> 398 resumable
> 316 await
> 825 synchronized
> 1,203 inspect
> 2,802 requires
> 3,963 concept
> 7,052 yield
> 22,771 when

27,522 import
178,469 module

There are about 1700 tokens average per source file. There are 2.5M source files.

That looks like using “module” would cause significant damage. In addition, Herb Sutter did some counting on Github:

“namespace module”: ~11,000 potential hits
“namespace import”: ~6,000 potential hits

I conclude that we must consider and evaluate alternatives. I think we have eight alternatives

- Stick with “module”, “import”, and “export” as keywords. The data above appears to make this – otherwise ideal – solution likely to cause serious problems.
- Use different – less often clashing – keywords; e.g., “moduledef”.
- Use combinations of keywords – “context sensitive keywords”; e.g., “module definition”.
- Bracketing the module name; e.g. “import<X>”.
- Use grammar rules to disambiguate uses of “module”, “import”, and “export”; e.g., “module X;” and “class module X”.
- Introduce a general mechanism in the language to say “this is a keyword” or “this is an identifier”; e.g., “@module”.
- Opt-in to new keywords; e.g., “__keyword(module)”.
- Use prefix underscores (and macros); e.g., “__Module”.

The first six solutions have well-received precedence in C++.

Not all “legacy” uses of an identifier are equally serious. For a start, consider:

- As a type name used in an ABI
- As a namespace name used in an ABI
- As a member name in a class used in an ABI
- As a function name
- As a global variable
- As a local variable
- As an alias

If we need more data, we’ll have to dig into such differences.

2. Best keywords

The current proposal uses ordinary keywords “module”, “import”, and “export”. In the abstract, that’s the ideal solution for users and tool builders. However, we have billions of lines of existing code to consider.

“export” has been reserved for over a decade, so that’s not a problem.

“import” is used about an order of magnitude less than “module”.

This is what the canonical first C++ program would look like:

```
import iostream;

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

I will use this example as not atypical of what many people's first impression of C++ will be. I am happy with this code and assume we all are.

It has been mentioned that we may use a workaround for the conventional keywords during a transition period only, say until C++23. I doubt that will work because important old code bases live “forever” independently of what the standards committee says. However, my favorite alternative to keywords “module” and “import” (§10) would serve for that because it would require no source code changes.

3. Odd keywords

In the past, we have resolved the problem with new keywords having previous uses by choosing keywords that were sufficiently mnemonic to be acceptable and sufficiently odd not to have been frequently used. I consider “decltype” and “constexpr” successful examples. Curiously, people keep complaining about “co_yield”. I wish I knew why. Possible reasons:

- People don’t like the coroutine proposal and use the keywords as a proxy
- People don’t like the underscore. We don’t usually use underscores in keyword (except for “function like” operations: e.g., “static_cast” and static_assert”).
- People don’t think “yield” is a frequently used name, but they are wrong: think finance and agriculture and/or count in existing code.

Nico Josuttis suggested two alternatives for “module”:

- cppmodule
- moduledef

yes, it's not perfect, but close because
- self intuitive
- pretty short
- probably not very conflicting.

Andrew Tomazos mentioned these ideas:

modu
modul
modname
modspace

I have mentioned “moduledef” in the past with “typedef” as a precedence and consider it the least bad alternative to “module”. However, “typedef” is itself a hack to avoid having "type" clash with early

1970s code. I have disliked it for decades. We don't write "classdef" and "namespacedef" exactly because I/we decided to grab the best keyword.

Putting "cpp" in front of a C++ keyword seems seriously weird to me. I mean, this is C++ and if you must, why not "c++module"?

I list my reaction to "cppmodule" to illustrate the mixture of logic and emotion that often is a part of a naming discussion. The choice of a name is often significant.

In addition to "module", we may have to deal with clashes with "import". I don't know how big a problem "import" is, but if we have to deal with "import", finding well-matched three "sufficiently odd and mnemonic" keywords for "module", "import", and "export" will be hard. I don't have a good candidate set.

Consider:

```
cppimport iostream;

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

4. Context sensitive keywords

We can use context to disambiguate "keywords" from identifiers.

Ville Voutilainen suggested:

```
module interface M;
module implementation M;
import module M;
```

Despite being a lousy typist, I find this quite attractive. One reason I find it attractive is that the context needed is very local. You can view these as simply keywords with a space in their spelling. I do, however, think that "definition" would be better than "implementation".

```
module definition M;
```

Consider a simple parsing strategy: You read an identifier and then if the terminating character is a space and the identifier is "import" check if it is followed by the identifier "module." Possibly

```
module import M;
```

would be better, more regular.

We can use plain "export" because "export" has been reserved for more than a decade.

It is likely that "import" and "export" will be far more frequent than "module" and that "import" will be part of the first "modern C++" program people see, so a nice and reasonably terse syntax is important.

```
import module iostream;
```

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

The first time someone sees this, the "module" might even help comprehension, but after a few dozen times, that will just be noise and a nuisance. The verbosity would last for decades.

5. Bracketing

Presenting the above alternatives, some people invariably comment that they'd like some sort of bracketing of the module name (undoubtedly inspired by experience with #include):

```
import "iostream"

import <iostream>
```

Comments include: "Then we wouldn't need the semicolon" and "Then import doesn't clash with other uses of 'import'". The last is not quite true (see §6 below) and the support for legacy headers blocks this line of development. We don't have many bracketing characters in C++ but we could consider “import{foo}”, “import[foo]”, and “import(foo)”. I don’t find those attractive.

Bracketing doesn’t address problems with “module”.

JF Bastien points out (without proposing it) that JavaScript relies on bracketing and more:

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/import

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/export

```
export { name1, name2, …, nameN };
export { variable1 as name1, variable2 as name2, …, nameN };
export let name1, name2, …, nameN; // also var, const
export let name1 = …, name2 = …, …, nameN; // also var, const
export function FunctionName(){...}
export class ClassName {...}

export default expression;
export default function (…) { … } // also class, function*
export default function name1(…) { … } // also class, function*
export { name1 as default, … };

export * from …;
export { name1, name2, …, nameN } from …;
export { import1 as name1, import2 as name2, …, nameN } from …;
export { default } from …;
```

I am not tempted.

6. Grammar-based resolution

Thinking back to the serious problems with C using “struct” to distinguish data structures from variable and function identifiers, I suggested that:

> I think the only problems are
>
> module;
>
> module X;
>
> module X:B;
>
> Of these
>
> module;
>
> module X:B;
>
> are easily handled context sensitively. That leaves
>
> module X;
>
> where in some old code "module" is a type name. This last case could be disambiguated by taking "module" as a type if it has previously been defined or alternatively as a module if "module;" has been seen.
>
> That would be a hack, of course, like the hack that allows us to write "struct stat stat;" and would prevent us from having global variables of a type called "module" in a TU defining a module. More technical details need sorting out for this to be feasible.

I suspect I was wrong about making "module X;" an object after a declaration of "module" has been seen: There would have to be a way to disambiguate in favor of it being a module definition. We already have ways of disambiguating in favor of an identifier, e.g., “class module X;”.

Nathan Sidwell commented:

> Time to come clean. I implemented -fno-module-keywords as an extension last week. At the outermost scope it causes '[export] module ...' and '[export] import ...' to be tentatively parsed as a module/import decl and if that fails, then as a c++17 declaration. Error recovery is pretty simple as one just looks to see if the first token is one of these two identifiers to give a meaningful message (which we want to do anyway, in case someone's compiling module code in non-module mode).
>
> Oh, that reminds me, there is another ambiguous case:
>
> ```
> template <typename T> struct import {};
>
> import<int> x;
> ```
>
> We'll tokenize '<int>' as a header-name, not <, int, >, and therefore fail, considering it an ill-formed import-declaration. This tokenizing only happens at the outermost scope. I don't know

if the existing code that uses these as identifiers has them as globally-visible template names.

Nathan and I came to our conclusions independently. The implementation techniques are similar to the one I suggested for context sensitive keywords.

We can distinguish

```
import <foo>     // legacy import
```

from

```
import<foo> x;  // variable
```

by looking ahead for the identifier.

See §9.

7. Keyword disambiguator

Francis Glasborow:

```
class module {};

module foo; // declares an object of type module

@module foo; // introduces a module called foo
```

Note that C++ has a continuing problem with supporting decades of legacy code, a problem not faced by most languages. It is our very success in creating long term robust code bases that causes us problems not faced elsewhere.

This is the idea of having a way of saying “this is a keyword” just as we have ways of saying “this is the name of a class” (e.g., “struct stat stat;”) and “this is a template.” This solution could then be used for all future keywords for which we feared clashes.

There are two variants of this idea:

- @ is required for all uses of "module" as a keyword
- @ can be added to "module" if/when we want to resolve a clash with an identifier

In practice, I think people would use @ everywhere to avoid making modules vulnerable to future type definitions.

If we take this direction, I would hope (expect) that we would be able to use @ for older keywords so we could write @if and @class. Otherwise, new features requiring keywords would be seen as second-class citizens and it would be a glaring irregularity in the language.

```
@import iostream;

int main()
{
```

```
        std::cout << "Hello, world!\n";
}
```

A less ambitious scheme would be to just distinguish "module" and "import". For example, using the less noisy dot:

```
.import iostream;

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

Now, would we also require or allow a dot before "export"?

I really don't like @the idea @of having @to add @a distinguishing mark @to keywords; that emphasizes grammar over contents. Just think how that might look @if you applied @the idea @to English.

8. Opt-in to new keywords

John Spicer mentioned that we might let users opt in to use the new keywords. For example:

```
__keyword(module)

__keyword(import)
```

That would imply that all uses of "import" as an identifier would have to occur lexically before the first use of "import" as a module directive. Alternatively, we would need a directive to make a keyword no longer a keyword, e.g., "__unkeyword(module)". Either way, this would force programmers to rework their code before they could use it in modules. I consider that unmanageable.

Also, simple, common uses would be ugly:

```
__keyword(import)

import iostream;

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

Alternatively, the "__keyword" mechanism could be used for each use of the keyword:

```
__keyword(import) iostream;

int main()
{
```

```
        std::cout << "Hello, world!\n";
}
```

Herb Sutter points out that there is precedence for something like this in C++/CLI with an "__identifier" operation to suppress a keyword.

I see this as a more verbose variant of the @ notation from §7.

Alternatively, an identifier could be made into a keyword (possibly in the global scope only) from the command line or a keyword suppressed from the command line. That would make the choice of meaning for "import" in a per-TU basis. That's not very flexible and would segregate module code from code using "module" or "import" as identifiers. It would also make the source code unreadable in isolation (without knowing the compiler options. I don't see that as a viable transition strategy.

9. Underscore and macro magic

Someone suggested the C approach of a keyword with a leading underscore plus a capital (e.g. _Bool, _Atomic, and _Complex) plus a macro in a header file (e.g., bool, atomic, and complex). I consider that the worst of both world.

Prefix underscore keywords are ugly, widely disliked, out of character with the rest of the language, exposing the history of the language, and arguably making newer features second class citizens.

Underscore keywords are commonly used together with macros to hide the ugliness

```
#define module __Module
```

That macro would need to be standardized avoid different organizations using different "nice names" for "__Module". Once standardized, the "nice names" (e.g., "module") become de facto keywords because a user must avoid using "module" for anything else in case some #included a header that (directly or indirectly) contains a definition (standard or not).

Thus the "__Module" solution is one of the two alternatives that I consider completely unacceptable.

```
__Module iostream;

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

10. My preferred solution

I'm flexible, but of course I have a favorite approach, a grammar-based approach (§6), a variant of Nathan's proposal [Sidwell,2018]:

> In the global scope (only) the identifiers "module" and "import" are taken as keywords (representing their meaning for modules) when they appear as the first token of a declaration and not before a ::.

That's it.

```
import foo;       // a module import directive
```

If you want to use "import" as the name of a type in the global scope, precede it with something:

```
class import foo;          // a variable of class import

const import foo = 7;      // a constant of class import

extern import foo;         // a variable of class import

::import foo;              // a variable of class import
```

In non-global scopes, there are no module constructs so we can use "import" and "module" as identifiers as ever.

```
int f(double module);

void f() { import x = 7; }

class X {
        int import;
};

void ff(X* p)
{
        auto g = p->import;
}

namespace import {
        module x;
}

import::x = {7};           // variable in namespace import

namespace {
        module z;
}

::module z;       // variable of type found in the unnamed namespace
```

Finally, consider this beauty (from Gabriel Dos Reis):

```
module;
class module { };
export module M;   // module declaration or exported variable declaration?
```

I wouldn't encourage such use of "module" and "import" as identifiers in new code, but it seems to take care of most problems with existing code.

I thought about the problem with a template called “import” (§6) and came to the conclusion that having a rule for that was most likely more trouble than it was worth:

```
import<int> f(); // error: bad import directive

:: import<int> f();          // OK, assuming import is global

N::import<int> f();          // OK, assuming import is in namespace N
```

I will of course reconsider if someone brings forward a significant number of real-world examples.

There are undoubtedly problems with this solution. It is still just a direction/suggestion, but if we like it, all could be ready for Kona.

11. Conclusions

We should seriously consider *not* having “module” and “import” as keywords. If we decide that we cannot use “module” and “import” as keywords, choose a solution based on context (what I have labeled context sensitive keywords” and “name resolution”).

A plea: whatever you do, don’t delay accepting modules for lack of consensus over which of the solutions listed here is best. Apart from keyword opt-in and the macro magic, I think that they are all minimally acceptable, so we would just be arguing which would be marginally better.

Acknowledgements

Thanks to all who took part in the “Context-sensitive module keyword” reflector debate and/or sent me comments.

References

Nathan Sidwell: Modules: Context Sensitive Keyword. P0924r0.