Document Number: p1218r0
Date: 2018-10-05
To: SC22/WG21 EWG
Reply to: Nathan Sidwell
nathan@acm.org / nathans@fb.com
Re: N4720 Merging Modules

Redefinitions in Legacy Imports

Nathan Sidwell

The merged modules draft introduces legacy header units, which can introduce multiple definitions of an entity into a compilation. When are such definitions well formed, when are they ill-formed? Should any cases be implementation-defined?

1 Background

Legacy header units share some redefinition semantics from the header-files from whence they came. But some module-specific situations exist where new semantics have been defined.

1.1 Direct Visible Redefinition

Consider:

```
// legacy-a.h
struct X {}; // #1:1

// legacy-b.h
import "legacy-a.h";
struct X {}; // #1:2
```

The definition at #1:2 is ill-formed, because the definition #1:1 is visible. This matches the include-file behaviour, were the import declaration be replaced by an include directive.

1.2 Reachable Sibling Definitions

Consider:

```
// legacy-a.h
struct X {}; // #2:1

// legacy-b.h
struct X {}; // #2:2
```

```
// elsewhere
import "legacy-a.h";
import "legacy-b.h";
X x; // #2:3
```

The declaration at #2:3 causes lookup of `::X`. This finds definitions #2:1 and #2:2, which are the same (they both occur in the global module). This is well-formed.

1.3 Indirect Non-visible Redefinition

Consider:

```
// legacy-a.h
struct X {}; // #3:1

// module foo
export module foo;
import "legacy-a.h"; // #3:2

// legacy-b.h
import foo;
struct X {}; // #3:3
```

This situation occurs when a component used by (non-modularized) `legacy-b.h` has been modularized, but uses a legacy header.

The definition #3:1 is not visible at #3:3, should that latter definition be well-formed?

2 Discussion

The semantics described in Sections 1.1 & 1.2 are correct. The third example is not so clear, falling somewhere between the other two cases.

The two legacy headers are clearly related, they define the same entity. But for unknown reasons they decline to cooperate in that definition. Some corners of C library implementations might have this property.

Depending on implementation strategy, implementing this case could be by:

- Skip over the second definition's tokens, assuming the ODR, or
- Hide the earlier definition during parsing, and discard or check the result

Such an implementation could be extended to allow the first case to be well-formed. It is unlike the second case, where the merging happens during, or immediately after deserializing the two definitions.

This case can only occur in partially modularized code bases. Were module `foo` a legacy header unit, it would be equivalent to the first example and ill-formed.

This case occurs, as Clang modules deals with the situation, but I do not know how common it might be. It is not clear how easy it might be to work around in the source base.

2.1 Default Function Arguments

Default function arguments may be added by a redeclaration. This presents some additional subtleties to the above examples. The merged proposal covers these cases, there being no changes to [dcl.fct.default].

In the first example's organization, `"legacy-b.h"` could add an earlier default argument to a declaration first created in `"legacy-a.h"`. Following the *as-if header* paradigm, that would be well formed. If it added a duplicate default argument, it would be ill-formed.

In the second example, the two legacy headers could declare the same function with different numbers of default arguments. The ODR places no restrictions on the default arguments of a function declared in different translation units.[1]

One option would be to keep the declarations distinct, and add default arguments as necessary during overload resolution. Clearly this can give rise to ambiguities – how is the overload resolution algorithm to pick which set of default arguments to apply?[2] This does not seem helpful semantics. Better semantics are:

- The duplicated default arguments should be ODR-same, NDR – a QOI issue.

And either

- Non-duplicated default arguments become visible for any use of either declaration.
- There can be no non-duplicated arguments.

In other words, the intersection of the default argument sets must be the same, and uses observe the union. Whether the union may be larger than the intersection is a decision to be made.

Again, the third example falls between these two cases, and for consistency should follow the same solution as the class redefinition case.

2.2 Pre-import Definitions

The examples may be perturbed by placing the definition before the import. However, after the Bellevue'18 meeting, it was decided that non-preamble [explicit] legacy imports were ill-formed, in

1 It does require the *definition* of multiply-defined functions to have the same token sequence.[basic.def.odr]/12.1
2 It is also going to be tricky to implement in certain implementations.

order to simplify generating preprocessed output. That simplifies analysis of the direct case – it continues to be ill-formed. The indirect situation remains the interesting case:

```
// legacy-a.h
struct X {}; // #4:1

// module foo
export module foo;
import "legacy-a.h"; // #4:2

// legacy-b.h
struct X {}; // #4:3
import foo;  // #4:4
X x;         // #4:5
```

The definition #4:5 causes lookup of `::X`, and hence the lazy loading of `"legacy-a.h"`'s definition. The merging here is semantically very similar to the well-formed indirect case, as it occurs at the same point (just after lazy loading) and not during parsing of a structure definition. However, allowing this case, but disallowing the Section 1.3 case will result in source code fragility with respect to reordering.

3 Proposal

The indirect non-visible redefinition case of Section 1.3 is ill-formed.

For default function arguments, duplicated ones must be ODR-same. There must no non-duplicated ones. Again, both cases are NDR.

The pre-definition indirect redefinition case discussed in Section 2.2 is ill-formed.