Dependent Structure Types
Author: Martin Uecker

Charter Principles (n3280):

Do not leave features in an underdeveloped state
Enable secure programming

Context:

- Variably-modified (VM) types and VLAs were added in C99
- VM/VLAs were made optional in C11 due to - at that time - insufficient compiler support
- The use of the VLA notation in prototypes was part of the C23 charter (n2611)
- Various improvements for arrays including the [.n] syntax were proposed (n2660)
- VM types were made mandatory in C23 to facilitate bounds checking (n2778)
- Various improvements were made to C23 and C2Y to better support VM types (empty initialization n2900, _Countof n3369, _Generic n3348, composite type n3652)
- N3188 “Identifying array length state” had support in Strasbourg (10/3/5)

Introduction

One severe limitation of variably-modified types is that it is not possible to store them in structures or unions, which prevents their use in data structures. N3188 proposes to make the length state explicit which would allow such types in structures and also at file scope. The paper gained support in the Strasbourg meeting, but is relatively complex as it combines several features. Here, we propose wording for the minimal changes required to allow variably-modified types as members of structure and unions.

Example:

```
struct vec {
      int N;
      double (*data_ptr)[.N];
};

void vec_init(struct vec v)
{
      for (int i = 0; i < _Countof(*v.data_ptr); i++)
            (*v.data_ptr)[i] = 1.;
}

struct vec f()
{
      struct vec a = { 10, malloc(sizeof(access(a))) };
      if (!a.data_ptr) exit(1);
      vec_init(a);
      return a;
}
```

As shown in the next example, this can be simulated today with a simple macro that re-attaches the correct variably-modified type using a cast, which shows that this feaure can be implemented easily as a front-end only feature. The macro solution is not convenient, not perfectly safe, and also has the usual macro problems with double evaluation / expansion.

Example (macro): https://godbolt.org/z/h7618r8vf

```
struct vec {
      int N;
      double (*data_ptr)[/*.N*/];
};

#define access(x) (*(typeof((*(x).data_ptr)[0])(*)[(x).N])(x).data_ptr)

void vec_init(struct vec v)
{
      for (int i = 0; i < _Countof(access(v)); i++)
            (access(v))[i] = 1.;
}

struct vec f()
{
      struct vec a = { 10, malloc(sizeof(access(a))) };
      if (!a.data_ptr) exit(1);
      vec_init(a);
      return a;
}
```

Essentially, a compiler implementing the new syntax would insert code similar to the access macro. To make this proposal simpler to understand and to review, it is split up into two steps.

Change 1: VM-Types in Structures and Unions

We first allow variably-modified types for members of structures of unions. This is already supported by GCC so standardizes an existing extension. Note, we do **not** propose to allow VLAs as a structure member, because this would mean that offsets of structure members become variable causing various complications. While GCC supports this too (and some other languages such as Ada also allow this), this seems to create more problems than it solves and was also previously rejected by Clang. With Change 1 the following code would become standard conforming.

https://godbolt.org/z/h3K6vzTPq

```
void vec_init(int n, struct vec { double (*data_ptr)[n]; } x)
{
      for (int i = 0; i < _Countof(*x.data_ptr); i++)
            (*x.data_ptr)[i] = 1.;
}

void bar(int n)
{
      struct vec { double (*data_ptr)[n]; } a = { };
      if (!(a.data_ptr = malloc(sizeof(*a.data_ptr)))) exit(1);
      vec_init(n, a);
      free(a.data_ptr);
}
```

Change 2: New Syntax for Dependent Product Types

Second, we introduce a new syntax `[ . identifer ]` support dependent product types where the type of a member depends on another member.

Note that the specific syntax is not critical for this proposal. It was selected here because it had already found approval from WG14 in Strasbourg. It also does not require any changes to name lookup or disambiguation rules, does not break existing conforming code, does not collide with existing compiler extensions, is intuitive as it reuses the existing syntax of designators in initializers, and is simple to support in existing parsers.

Nevertheless, if we also standardize the `counted_by` and similar attributes (which we should do to be able to annotate existing code where the type can not be changed and to be compatible with C++ in headers) it would – in principle - be nice to use the same syntax to refer to other members. N3656 discusses different options and favors a change to the name lookup rules as already implemented by the existing clang prototype by Apple. Outside of attributes, changing the name lookup rules would risk breaking existing standard conforming code and the exiting GCC extension, so this would need to be carefully considered. Nevertheless, there is not much risk should WG14 decide later to adopt such changes based on a future proposal. In this case, we could even remove the `[ .n ]` notation again before the finalization C2Y.

Future Work

A logical next step is to introduce the changes needed to support flexible-array members as previously discussed in N2660, N2905 and N3188. Another change would allow redeclaration of dependent structure types under specific conditions. Even further changes could support VM-types and VLAs at file scope as proposed in N3188. A future improvement could also be revise the general definition of variably-modified types to make a dependent structure not to be considered a variably-modified type, when it an identifier to another member, while making a function prototype where the size does not refer to an argument be variably-modified, i.e. the definition should consider whether the identifier is bound or free relative to the specified type.

Note that tagged unions with run-time checking could be implemented on top of a future flexible-array member version of this feature and one could consider further syntax improvements in this direction.

```
struct foo {
      enum { A, B, C } tag;
      union {
            char a[.tag == A];
            double b[.tag == B];
      };
};
```

https://godbolt.org/z/4GW4Mvdzx

The proposal has synergies with array notation for vectorization (N3617, N3618), polymorphic types (N3212), safer span or string types (N3210), and bounds-safe programming in general (N3211, N3395).

Acknowledgment: Jens Gustedt for pointing out some errors in a previous draft version.

Wording (relative to N3685)

Change 1 (variably-modified types in structure/unions):

6.7.3.2 Structure and union specifiers

Constraints

3 Each member of a structure or union shall have a known constant size[xxx] ~~(hence, a structure shall not contain an instance of itself, but can contain a pointer to an instance of itself)~~, except that the last member of a structure with more than one named member can have incomplete array type; such a structure (and any union containing, possibly recursively, a member that is such a structure) shall not be a member of a structure or an element of an array.

xxx) Hence, a structure **can** not contain an instance of itself, but can contain a pointer to an instance of itself. **A structure can not contain a VLA.**

Semantics

11 A ~~member of a structure or union can have any type of known constant size other than a variably modified type.136) . In addition, a~~ member can be declared to consist of a specified number of bits (including a sign bit, if any). Such a member is called a bit-field;137) its width is preceded by a colon.

~~136)A structure or union cannot contain a member with a variably modified type because member names are not ordinary identifiers as defined in 6.2.3~~

6.7.3.4 Tags

Constraints

1 Where two declarations that use the same tag declare the same type, they shall both use the same choice of struct, union, or enum. If two declarations of the same type have a member-declaration or enumerator-list, one shall not be nested within the other and both declarations shall fulfill all requirements of compatible types (6.2.7) with the additional requirement that corresponding members of structure or union types shall have the same (and not merely compatible) types **and no member shall have a variably modified type in either of the two declarations**.

6.7.7 Declarators
6.7.7.1 General
6.7.7.3 Array declarators

Constraints

2 If an identifier is declared as having a variably modified type, it shall ~~be an ordinary identifier (as defined in 6.2.3),~~ have no linkage, and have either block scope or function prototype scope. If an identifier is declared to be an object with static or thread storage duration, it shall not have a variable length array type.

10 EXAMPLE 4 All valid declarations of variably modified (VM) types are either at block scope or function prototype scope. Array objects declared with the thread_local, static, or extern storage-class specifier cannot have a variable length array (VLA) type. However, an object declared with the static storage-class specifier can have a VM type (that is, a pointer to a VLA type). ~~Finally, only ordinary identifiers can be declared with a VM type and identifiers with VM type cannot, therefore, be members of structures or unions.~~

```
extern int n;
int A[n];                       // invalid: file scope VLA
extern int (*p2)[n];            // invalid: file scope VM
int B[100];                     // valid: file scope but not VM
void fvla(int m, int C[m][m]);  // valid: VLA with prototype scope
void fvla(int m, int C[m][m])   // valid: adjusted to auto pointer to VLA
{
      typedef int VLA[m][m];    // valid: block scope typedef VLA
      struct tag {
            int (*y)[nm];       // invalid: y not ordinary identifier member and VM
            int z[nm];          // invalid: z not ordinary identifier member and VLA
      };
      int D[m];                 // valid: auto VLA
      static int E[m];          // invalid: static block scope VLA
      extern int F[m];          // invalid: F has linkage and is VLA
      int (*s)[m];              // valid: auto pointer to VLA
      extern int (*r)[m];       // invalid: r has linkage and points to VLA
      static int (*q)[m] = &B;  // valid: q is a static block pointer to VLA
}
```

Change 2 (dependent structure/union types):

6.5.3.4 Structure and union members

Semantics

6 When an expression is evaluated that refers to a member with a variably-modified type declared using the [. *identifer*] notation, the type of the expression is changed so that all unspecified sizes specified in this manner are replaced by a value that is obtained by accessing the designated member.

6.7.3.2 Structure and union specifiers
Constraints

4 If the [. identifer] notation is used in the declaration of a member, the identifier shall be the name of a member of the same structure or union type and this member shall have an integer type. For a structure or union type specified at file scope, if a member has a variably-modified type the type shall not be specified with *array length expressions* of non-constant size and not with the [*] notation.

Semantics

11 A member of a structure or union can have any type of known constant size ~~**other than a variably modified type.**~~[136)]

12 In addition, a member can be declared to consist of a specified number of bits (including a sign bit, if any). Such a member is called a bit-field;137) its width is preceded by a colon.

136)~~**A structure or union cannot contain a member with a variably modified type because member names are not ordinary identifiers as defined in 6.2.3**~~ **A VLA does not have known constant size and is therefor not allowed. If the [. identifer] notation is used in the sequence of declarators in a member declaration, then this refers to an array of unspecified size (see 6.7.7.3) and the type of the member is a variably modified type.**

6.7.7 Declarators
6.7.7.1 General
Syntax

array-declarator:
direct-declarator [type-qualifier-list$_{opt}$ assignment-expression$_{opt}$]
direct-declarator [static type-qualifier-list$_{opt}$ assignment-expression]
direct-declarator [type-qualifier-list static assignment-expression]
direct-declarator [type-qualifier-list$_{opt}$ *]
direct-declarator [. identifier]

6.7.7.3 Array declarators

Constraints

1 In addition to optional type qualifiers and the keyword static, the [and] can delimit an expression, or ***, or an designator using the . identifier notation.** If they delimit an expression, called the array length expression, the expression shall have an integer type. If the expression is a constant expression, it shall have a value greater than zero. The element type shall not be an incomplete or function type. The optional type qualifiers and the keyword static shall appear only in a declaration of a function parameter with an array type, and then only in the outermost array type derivation. **A designator shall appear only in the declaration of a member of a structure or union.**

2 If an identifier is declared as having a variably modified type, it shall ~~**be an ordinary identifier (as defined in 6.2.3),**~~ have no linkage~~**, and have either block scope or function prototype scope**~~. If an identifier is declared to be an object with static or thread storage duration, it shall not have a variable length array type.

Semantics

3 If, in the declaration "T D1", D1 has one of the forms:

D [type-qualifier-list$_{opt}$ assignment-expression$_{opt}$] attribute-specifier-sequence$_{opt}$
D [static type-qualifier-list$_{opt}$ assignment-expression] attribute-specifier-sequence$_{opt}$
D [type-qualifier-list static assignment-expression] attribute-specifier-sequence$_{opt}$
D [type-qualifier-list$_{opt}$ *] attribute-specifier-sequence$_{opt}$
D [. identifier] attribute-specifier-sequence$_{opt}$

and the type specified for ident in the declaration "T D" is "derived-declarator-type-list T", then the type specified for ident is "derived-declarator-type-list array of T".163)164) The optional attribute specifier sequence appertains to the array. (See 6.7.7.4 for the meaning of the optional type qualifiers and the keyword static **and 6.7.3.2. for the meaning of the [. identifier] notation**)

4 If the array length expression is not present, the array type is an incomplete type. If there is * instead of an array length expression, the array type is a variable length array type of unspecified array length, which can only be used as part of the nested sequence of declarators or abstract declarators for a parameter **or member** declaration or as part of such a sequence in a type name of a generic association, not including anything inside an array length expression in one of those declarators,165) such arrays are nonetheless complete types. If the array length expression is an

integer constant expression and the element type has a known constant size, the array type has a known constant size; otherwise, the array type ~~has~~ is a variable length array type.

5 If the array length expression is not an integer constant expression: if it occurs in a declaration at function prototype scope **or for a member of a structure or union** or in a type name of a generic association (as described above), it is treated as if it were replaced by *; otherwise, each time it is evaluated it shall have a value greater than zero. The size of each instance of a variable length array type does not change during its lifetime. Where an array length expression is part of the operand of the typeof or sizeof operators and changing the value of the array length expression would not affect the result of the operator, it is unspecified whether or not the array length expression is evaluated. Where an array length expression is part of the operand with a _Countof operator and changing the value of the array length expression would not affect the result of the operator, the array length expression is not evaluated. Where an array length expression is part of the operand of an alignof operator, that expression is not evaluated.

10 EXAMPLE 4 ~~**All valid declarations of variably modified (VM) types are either at block scope or function prototype scope.**~~ Array objects declared with the thread_local, static, or extern storage-class specifier cannot have a variable length array (VLA) type. However, an object declared with the static storage-class specifier can have a VM type (that is, a pointer to a VLA type). ~~**Finally, only ordinary identifiers can be declared with a VM type and identifiers with VM type cannot, therefore, be members of structures or unions.**~~

```
extern int n;
int A[n];                       // invalid: file scope A has linkage and is VLA
extern int (*p2)[n];            // invalid: file scope p2 has linkage and is VM
typedef int (*td)[n][n];        // invalid: typedef name with VM type not at block scope
int B[100];                     // valid: file scope but not VM
void fvla(int m, int C[m][m]);  // valid: VLA with prototype scope
constexpr int MAXLEN = 10;
struct svm {
      int k;
      char (*buf)[.k];          // valid: member and VM
      char (*p)[n];             // invalid: VM member at file scope not using designator
      int z[.k];                // invalid: member and VLA
      char q[MAXLEN];           // valid: not VLA
};
void fvla(int m, int C[m][m])   // valid: adjusted to auto pointer to VLA
{
      typedef int VLA[m][m];    // valid: block scope typedef VLA
      struct tag {
            int (*y)[nm];       // invalid: y not ordinary identifier member and VM
            int z[nm];          // invalid: z not ordinary identifier member and VLA
      };
      int D[m];                 // valid: auto VLA
      static int E[m];          // invalid: static block scope VLA
      extern int F[m];          // invalid: F has linkage and is VLA
      int (*s)[m];              // valid: auto pointer to VLA
      int (*t)[.m];             // invalid: designator but not in structure or union
      extern int (*r)[m];       // invalid: r has linkage and points to VLA
      static int (*q)[m] = &B;  // valid: q is a static block pointer to VLA
}
```

6.7.9 Type definitions

Constraints

2 If a typedef name specifies a variably modified type then it shall have block scope.