WG14 N2660

Title: **Improved Bounds Checking for Array Types**
Author: **Martin Uecker, University Medical Center Göttingen**
Date: **2021-02-13**

Introduction

Array types with static or dynamic bound can be used instead of pointers for safe programming because compilers can use length information encoded in the type to detect errors. In fact, a pointer to an array is nothing else than a bounded pointer type and existing compilers can already add run-time checks to detect out-of-bounds accesses. Still, there a missing features and loose ends which prevent programmers and tools from making optimal use of array types in C.

Example:

```
void foo(int n, double (*x)[n])
{
  (*x)[n] = 1; // invalid access can be detected at run-time
}
```

In the following simple proposals are listed, which – if adopted into C - would strengthen safe programming using array types. They address loose ends regarding arrays in general, function arguments declared as arrays, and flexible array members (FAM). Most of these proposal were proposed before (cf. [1,2] and N-documents referred to in the text). They are collected here to demonstrate that a much better and consistent integration of array types in C is still possible with only relatively minor changes to the language.

2. Function Arguments Declared as Arrays

2.1 Consistent Declarations [N2074]

Require the length expression for pointers arguments declared as array to be consistent across function declarations.

Example:

```
void foo(int n, char *x);
void foo(int n, char x[n]);
void foo(int n, char x[3]);
```

Rationale: Making inconsistent length specifiers a constraint violation would help diagnose bugs.

2.2 Bounds Checking

Treat the length specifier for a function parameter declared as an array as a real bound, i.e. it should be UB to access the array beyond this bound (at least as long as the pointer is used directly and is not cast to another pointer type before).

Example:

```
int foo(int n, char x[n])
{
   x[n] = 1; // becomes UB
}
```

Rationale: Currently, the array argument is adjusted to a pointer and the length information then becomes meaningless. On the other hand, if the bound is not respected this always hints at a programming error. Making accesses beyond the specified bound undefined would make it possible for compilers to reject this code either already at compile or to add run-time checks. Note that is is possible to get a similar effect by passing a pointer to an array as in the example shown in the introduction, but this would not help existing code and has less convenient syntax.

2.3 C++ Header Compatibility

In C++ accept at least first level VLA syntax in function prototypes declared with C calling convention. (This affects C++ but is listed here for completeness.)

Example:

```
extern „C“ void foo(int n, double a[n]); // should be legal
```

Rationale: VLAs are supported by some C++ compilers as an extension. There were previous attempts to introduce them into the C++ standard, but this did not progress to the unsolved C++ specific questions. Still, support for first-level VLAs in function prototypes would already improve interoperability between both languages. Compilers and tools could also use this information for analysis, but a simple implementation could simply ignore the length parameter.

2.4 Forward Declaration of Arguments

Allow forward declaration of arguments in a function prototype.

Example:

```
void foo(size_t len; char buf[static len], int len);
```

Rationale: With the recent removal of K&R function definitions, it is impossible to use a later parameter in a length specifier. The syntax for forward declarations already exists as an extension in GCC. An alternative that was proposed could be to allow referring to later arguments in length expressions. Although this might be more elegant, it also is more complicated, as it may require more complicated changes to parsers, could cause backwards compatibility issues, and requires dealing with mutual dependencies between parameters [1]:

```
int n = 3;
void foo(char x[n], int n); // would change meaning
void foo(char (*x)[sizeof(*y))], char (*y)[sizeof(*x)]);
     // mutual dependency
```

2.5 Prototypes in the Standard Library

Amend standard library function where a pointer argument could be declared as an array with length specifier
.

Example:

```
char* strncpy(size_t n; char dst[static n], const char* src);
```

Rationale: In line with the C2X charter [N2611], this would make the API self-documenting and allow tools to diagnose bound violations at compile-time or at run-time. One could consider extending this syntax also to void pointers.

2.6 Reduced Type Confusion I

Make applying `sizeof` (and possible also `typeof` and `_Generic`) to function parameters declared as arrays an obsolescent feature.

Example:

```
void foo(double x[4])
{
     double y[4];
     _Static_assert(sizeof x == sizeof y, “Fail!”);    // !
}
```

Rationale: `sizeof` applied to a pointer argument declared as an array returns the size of the pointer but not of the array. This is confusing and error-prone, especially when such operations are hidden in macros. Some compiler already warn about this.

2.7 Reduced Type Confusion II

Add a new keyword / syntax that causes the type of the parameter declared as an array to remain an array type, i.e. not be adjusted to a pointer [N1990].

Example:

```
void foo(double x[:4])
{
     double y[4];
     _Static_assert(sizeof x == sizeof y, “Fail!”);    // ok.
}
```

Rationale: It would then be possible to treat the argument as a proper array and all issues caused by the adjustment to a pointer is avoided (no loss of length information, bounds can be checked, etc.)

3 Arrays and VLAs

3.1 The lengthof Operator [N2529]

Add a `lengthof` operator that returns the length of an array type and yields a compile-time error when not applied to an array type. As an exception, it might make sense to allow the `lengthof` operator also on pointer arguments declared as arrays.

Example:

```
int N = 7;
double arr[N];
size_t len = lengthof arr;
```

Rationale: To obtain the length of an array it is possible to use a macro based on `sizeof` but this is error prone. There is currently no way to query the length of a parameter declared as an array type with length specifier which is adjusted to a pointer, so this would be a useful extension.

3.2 Mandatory VLAs

Make VLAs mandatory again.

Example::

```
void foo(int n, double (*x)[n])
{
  (*x)[n] = 1; // invalid access can be detected at run-time
}
```

Rationale: While VLAs and variably modified types are widely supported by compilers, they are an optional part of the language. This makes it difficult to use them in standard (or other) APIs and may also discourage programmers from using them in their own code. Making them mandatory sends a clear message that their use for safe programming is encouraged. Security concerns about their use only affect automatic VLAs on the stack (e.g. not other variably modified types) and can already be addressed using compile-time analysis.

3.3 VLAs in Aggregates [2, N1990]

Add a feature that makes it possible to store pointers to VLAs in an aggregate.

Example:

```
struct foo {
  int n;
  float (*x)[.n + 1];
} f;
```

Here, `*(f.x)` would then have type `float[x.n + 1]` where the length expression is evaluated whenever `f.x` is evaluated. The length expressions which are allowed should be restricted to avoid side effects (e.g. similar to const expressions 6.6p3).

Rationale: VLAs are not fully integrated into the type system as they can not be used in aggregate types. This prevents their use in data structures where they would be most useful. The suggested syntax reuses existing syntax for designators in initializers, which avoids problems with scoping and fits well with the fact that members of structures live in their own namespace.

3.4 Wide Pointer Types [1]

Add wide (fat) pointer types that also store the dimensions of the array.

Examples:

```
int N = 7:
double a[10][N][3];

double (*x)[:][:][:] = &a;      // assumed shape

(*x)[3][3][2] = 1.;             // access could be checked
double (*y)[10][7][3] = x;      // bounds could be checked

void f(double z[:][:][:]);
f(a);
```

Rationale: Pointers to VLAs are very powerful because the length is explicit but sometimes this is inconvenient. This extension is simple to implement by packing dimensions and pointer in a wide pointer object. Wide pointers then have a different and larger representation than regular pointer types but could still be converted to a void pointer or other array types of the same shape and element type.

3.5 Array Slices [3]

Support array slices and vector operations on them.

Example:

```
int a[10][5], b[10][5], c[10][5];

b[:][:] = c[:][:] = 3;
a[:][:] = b[:][:] + c[:][:];

int d[5][5];
d[0:5][:] = a[0:2:10][:] + b[0:5][:] + c[5:10][:];
```

Rationale: This would make many operations on arrays more convenient to describe and less error prone to use.

3.5 Wide Pointers II

Add a wide pointer type that can reference slices.

Examples:

```
double a[10] = { 0., 1., 2., 3., 4., 5., 6., 7., 8., 9. };

double (*x)[::] = &a[4:2:10];   // points to 4., 6., 8.
```

Rationale: A more sophisticated version of a wide pointer that allows strided access of sub-ranges can also be implemented easily and would be very useful for numerical programming.

3.7 Zero Size Arrays

Define behavior for zero size arrays.

Examples:

```
int g[0];
size_t n = 0;
int f[n];
double (*p)[n];
```

Rationale: Corner cases in numerical code would be better support by allowing zero sized arrays. Some compilers already support these as an extension.

3.7 Longjmp

Specify that `longjmp` is not allowed to leak memory used by VLAs.

Example:

```
jmp_buf env;

void g(void) { longjmp(env, 1); }

void h(void)
{
  int n = 3;
  int a[n];       // may remain allocated
  g();
}

void f(void)
{
  if (0 == setjmp(env)) g();
}
```

Rationale: As the VLA could be defined in an intermediate function, the use of both VLAs and `longjmp` in the same code base appears dangerous. Compilers should either allocate VLAs on the stack (most do) or implement a proper stack unwinding scheme for `longjmp`.

4. Flexible Array Members

4.1 Incomplete Types

Specify that structures with a FAM (with unknown size) are incomplete types. Support old use cases which would become invalid as obsolescent features.

Examples:

```
struct foo { double a; char str[]; };    // FAM
sizeof(foo);      // FAM is ignored, deprecate!
struct foo x, y;
x = y;            // FAM is ignored, deprecate!
```

Rationale: The real size of a structure with FAM is unknown. This is the definition of an incomplete type. Such structures are currently treated as having a regular complete structure type and the FAM is silently ignored in most cases. This allows error prone use of `sizeof` and structure assignment.

4.3 Length Specifier

Adopt the syntax proposed in 3.3 for VLAs to allow specification of the length of a FAM. A structure with a FAM with a length specifier then has a known but variable size similar to a VLA and is a complete type of variable length.

Example:

```
struct foo2 { int len; char x[.len]; }; // FAM with bound

size_t len = 5;
size_t size = sizeof((struct foo2){ .len = len });
struct foo2* p = xmalloc(size);
p->n = len;
p->x[4] = ‘a’;    // VLA of size 5, access checked
```

Rationale: Computing the size of a structure with FAM is error prone as it requires explicit addition of the size of the array which is otherwise ignored. Also with size information accesses to FAM can be checked.

4.2 Initialization

Allow the initialization of a FAM or the length to complete the structure type.

Example (continued):

```
struct foo x = { 1.3, “bar” };  // initialization
char c = x.str[3];              // char array of size 4

struct foo2 y = { 3, “str” };   // compile-time error!
struct foo2 z = { .len = 10 };  // allocate enough space
z.str[5] = ‘a’;
```

Rationale: Initialization would be a safe way to allocate the correct size for automatic or global variables. This would work exactly as for conventional incomplete arrays that are completed by assignment. If a length specifier is provided space can be allocated for automatic variables and the initialization could be checked to not overflow the array.

4.3 Generalized Flexible Member

Generalize the FMA to any type which is incomplete or of variable size.

Example:

```
struct foo_priv;       // incomplete type
struct foo {
  int x;
  struct foo_priv y;   // allow!
};                     // struct foo also incomplete

struct foo *p = foo_alloc();
p->x = 1;              // access of x possible
foo_fun(&p->y);        // taking the address of y possible
sizeof(struct foo);    // error!
```

Rationale: Based on the changes above this would be a relatively straight-forward and consistent extension that would enable other interesting use cases and make flexible members are proper part of C's type system.

5. Conclusion

We summarized several idea on how to better integrate arrays into C's type system. This would strengthen safe programming with C as arrays can enable compile-time and run-time bounds checking. Support for numerical programming can also be improved. While most proposals are very simple, some would require some additional work. Still, all proposals build on existing concepts and can be implemented with relatively minor changes to the language.

6. Acknowledgement

Will Wray for providing background information and pointers.

7. References

[1] Ritchie DM. Variable-size arrays in C. The Journal of C Language Translation 1990;2:81-86.

[2] Cheng HH. Extending C with Arrays of Variable Length. Computer Standards and Interfaces 1995;17:375-406.

[3] https://www.cilkplus.org/tutorial-array-notation