JTC1/SC22/WG14 - N3394

Forward Declaration of Parameters v4 (updates N3207)

Martin Uecker, Graz University of Technology

Date: 2024-11-23

Introduction

It is not possible to use a later parameter in a size specifier, which makes it impossible to annotate existing APIs where the length of a buffer is passed before the pointer. Parameter forward declarations as implemented in GCC as an extension solve this problem [1]. In C, identifiers generally require a declaration before use (the only exception are labels where this works because they carry no information). Originally proposed in N2780 for C23, this feature did not receive sufficient consensus. With the removal of K&R function definitions, this also became a problem for function definitions and was brought up again as open issue, but wording could not be stabilized in time for C23. Various people expressed interest in a revision of this paper for the next version of C.

Changes in v3: In addition to issues pointed out before on the reflector [2], this revision now also tries to address the issues raised in [3]. In particular, the syntax was revised to make it more symmetrical with regard to parameter declarations and to resolve the typename / identifier parsing ambiguity and, questions about storage classifiers.

Changes in v4: Updated numbering to latest working draft, replaced some cases of “parameter forward declaration” with “parameter declaration in a parameter forward declaration list”, and added “as a parameter” in 6.7.7p4 as suggested in reflector message SC22WG14.24531. As a semantic change, also suggested there, empty parameter forward declarations are now forbidden by a change to 6.7.1p2.

Example (protecting existing API using forward declaration):

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

int main()
{
     char buf[100];
     foo(buf, 101);
}
```

<source>:9:5: warning: '**foo**' accessing 101 bytes in a region of size 100

https://godbolt.org/z/q7oo81arW

Examples of other forward declarations in C:

```
extern int counter;    // object with external linkage
void foo(void);        // function prototype
struct foo;       // tag
```

Alternative 1:

It was proposed to allow referring to later arguments in size expressions as a more elegant solution. While this initially seemed appealing, it turned out that this would be more complicated to specify and implement: It requires more complicated changes to existing parsers, has backwards compatibility issues, and there are problems related to mutual dependencies between parameters [4]. Thus, this solution would require the invention of many new rules.

Example (backwards compatibility issue):

```
int a;
int foo(char buf[static a], size_t a) { // meaning would
change
```

Alternative 2:

N3188 proposes the use of [.n] as a new syntax which clearly is nicer by avoiding repeated declarations, but is more limiting and has currently no implementation experience. This would also require rules how this interacts with designated initializers. Note, that this is not necessarily an alternative, one could also have forward declarations as well as new syntax for special cases.

Syntax

GCC supports comma and semicolon to separate multiple forward declarations. Here we propose to allow only the semicolon, because then it is directly visible whether a declaration is a forward parameter declaration or a parameter declaration. The syntax is robust against typos, because confusing a semicolon with a comma would either cause an invalid re-declaration of the same parameter name or a forward declaration for a parameter that does not exist.

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

The syntax is similar to forward declarations used in `for` and also poposed for `if` (N3196) although there a declaration is not repeated:

```
    for (int i = 0; i < 10; i++)
        run(i);
```

```
if (int i = 3; n == i)
    run(n);
```

C++ Compatibility

The syntax is not used in C++. Function declarations using run-time size expressions in argument types are not supported in C++. Thus, the use of such new extension is useful only in function declarations which are already not compatible with C++. It is possible to hide both run-time size expressions and forward declaration behind a macro which is sadly already required for any kind of parameter declared as VLA in headers shared with C++. This technique also enables backwards compatibility with older compilers.

```
#define HIDE(x)
void foo(HIDE(size_t len;)
         char buf[HIDE(static len)],
         size_t len);
```

General Issues

The GCC extension is obscure and rarely used. Nevertheless, it is enabled by default and does not seem to cause problems. Apart from the removal of K&R function definitions, another reason it became more useful recently is that compilers started to use size expressions for improved compile-time and run-time bounds checking as shown in the initial example and this already sparked increased interest in adding such annotations to existing APIs. Because of such reasons, the extension was also requested by users of other compilers [5].

References:

[1] https://gcc.gnu.org/onlinedocs/gcc/Variable-Length.html

[2] [SC22WG14.20184] Parameter forward declarations (N2780)

[3] [SC22WG14.23585] Parameter forward declarations (N3121)

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

[5] https://github.com/llvm/llvm-project/issues/47617

Acknowledgments: Joseph Myers for reviewing previous versions of this proposal and Aaron Peter Bachmann and others for encouragement. All errors are mine.

(proposed wording on next page)

Proposed Wording

6.7 Declarations
6.7.1. Syntax

Constraints

2 **~~If~~ a A** declaration other than a static_assert or attribute declaration **that** does not include an init declarator list **or a parameter declaration in the parameter forward declaration list (see 6.7.7)**, **~~its declaration specifiers~~** shall include one of the following **in its declaration specifiers:**
— a struct or union specifier or enum specifier that includes a tag, with the declaration being of a form specified in 6.7.3.4 to declare that tag;
— an enum specifier that includes an enumerator list.

3 EXAMPLE ...

4 If an identifier has no linkage, there shall be no more than one declaration of the identifier (in a declarator or type specifier) with the same scope and in the same name space, except that:
— a typedef name may be redefined to denote the same type as it currently does, provided that type is not a variably modified type;
— tags may be redeclared as specified in 6.7.3.4.
--- parameters declared in a parameter forward declaration list are redeclared in the parameter list as specified in 6.7.7.4.

6.7.7 Declarators

1 Syntax

```
parameter-type-list:
      parameter-forward-declaration-list_opt parameter-type-list
      parameter-list
      parameter-list , …
parameter-forward-declaration-list:
      parameter-declaration ;
      parameter-forward-declaration-list parameter-declaration ;
parameter-list:
      parameter-declaration
      parameter-list , parameter-declaration
parameter-declaration:
      attribute-specifier-sequence_opt declaration-specifiers declarator
      attribute-specifier-sequence_opt declaration-specifiers abstract-declarator_opt
```

6.2.2 Linkages of identifiers

2 In the set of translation units and libraries that constitutes an entire program, each declaration of a particular identifier with external linkage denotes the same object or function. Within one translation unit, each declaration of an identifier with internal linkage denotes the same object or function. **With the exception of a parameter declaration in the parameter forward declaration list and the corresponding parameter declaration in the parameter list that declares the same identifier,** each declaration of an identifier with no linkage denotes a unique entity.

6.2.7 Compatible type and composite type

5 For an identifier with internal or external linkage declared in a scope in which a prior declaration of that identifier is visible 60), if the prior declaration specifies internal or external linkage, the type of the identifier at the later declaration becomes the composite type. **The type of a parameter with a parameter declaration in the parameter forward declaration list becomes the composite type at the parameter declaration in the parameter list.**

6.7.7.4 Function declarators

Constraints

2 The only storage-class specifier that shall occur in a parameter declaration is register.

4 An identifier declared as a parameter in a parameter forward declaration list shall be declared exactly once in the parameter list. Both declarations shall specify compatible types before adjustment and have the same storage-class specifiers.

Semantics

5 A parameter type list specifies the types of, and may declare identifiers for, the parameters of the function. **Parameter declarations in a parameter forward declaration list may provide forward declarations of the identifiers of the parameters (useful for size expressions).**

10 If, in a parameter declaration, an identifier can be treated either as a typedef name or as a parameter name, it shall be taken as a typedef name.

12 The storage class specifier in the declaration specifiers for a parameter declaration, if present, is ignored unless the declared parameter is one of the members of the parameter type list for a function definition. The optional attribute specifier sequence in a parameter declaration **parameter declarations in a parameter forward declaration list** appertains to the parameter.

6.9.2 Function definitions

10 On entry to the function, the size expressions **of parameter declarations (including parameter declarations in the parameter forward declaration list) of** ~~**of each**~~ variably modified ~~**parameter**~~ **type** are evaluated and the value of each argument expression is converted to the type of the corresponding parameter as if by assignment. (Array expressions and function designators as arguments were converted to pointers before the call.

Examples:

```
// valid examples

void transpose(int x; int y; double matrix[x][y], int x, int y);

void a(const struct buf *n;                 // forward declarations
       char dst[n->length],
       const struct buf *n);

void b(int n; char *x [[xzy::count(n)]], int n);

void c(struct bar { char buf[10]; }; struct bar *dst, const struct bar *src);

void d(double (*p)[3][*]; double (*p)[*][4]);
void d(double (*p)[3][4]);                          // compatible declaration of ‘a’

void e(struct bar { int x; }; struct bar *dst, const struct bar *src);

void f(int n; long (*in)[3 * n]; char buf[sizeof(*in)], long (*in)[3 * n]);

// invalid examples

void d(double (*p)[4][4]);        // incompatible declaration of ‘a’
void d(double (*p)[3][5]);        // incompatible declaration of ‘a’

void h(int x; const int x);       // incompatible types of ‘x’
void i(int x[3]; int x[4]);       // incompatible types  of ‘x’ before adjustment

void j(int x; int x; int x);      // invalid redeclaration of ‘x’, 6.7p3
void k(int x; int x, int x);      // invalid redeclaration of ‘x’, 6.7p3

void l(register int x; int x;)    // different storage-classifier
```

Additional Change

Because existing practice is to ignore the following constraint, it is suggested to remove it:

https://godbolt.org/z/YfGfMs7q3

12 ~~**The storage class specifier in the declaration specifiers for a parameter declaration, if present, is ignored unless the declared parameter is one of the members of the parameter type list for a function definition.**~~ The optional attribute specifier sequence in a parameter declaration **and in a parameter declarations in a parameter forward declaration list** appertains to the parameter.