Proposal for C2y

WG14 N3433

Title: Alternative syntax for forward declaration of parameters

Author, affiliation: Christopher Bazley, Arm. (WG14 member in individual capacity – GPU expert.)

Date: 2025-01-18

Proposal category: New features

Target audience: Committee

Abstract: This paper proposes an alternative to the syntax for forward parameter declarations advocated by N3394 (and every previous paper on the same topic) and explains why it may be considered preferable.

Prior art: GCC, N2780, N3121, N3140, N3207, N3388, N3394.

Alternative syntax for forward declaration of parameters

Reply-to: Christopher Bazley (chris.bazley@arm.com)
Document No: N3433
Date: 2025-01-18

Summary of Changes

N3433

- Initial proposal

Introduction

This paper was not written with the intent of undermining the efforts of Martin Uecker and others to standardise some flavour of the GCC extension [1] that allows forward declarations of function parameters; its purpose is simply to offer the committee an alternative syntax. The arguments put forward by Martin for his preferred syntax are persuasive, but I do not believe they are unanswerable.

New syntax has far-reaching, unpredictable and irrevocable consequences because it sets new precedents and circumscribes future language extensions. I believe there is sufficient cause for concern that the committee should think very carefully about their decision.

The following quote from a famous programming language designer [2] seems pertinent:

> *It is not that syntax isn't important; it is immensely important because the syntax is quite literally what people see. A well-chosen syntax significantly helps programmers learn new concepts and avoids silly errors by making them harder to express than their correct alternatives. However,* ***the syntax of a language should be designed to follow the semantic notions of the language****, not the other way around. This implies that language discussions should focus on what can be expressed rather than how it is expressed.* ***An answer to the what often yields an answer to the how****, whereas a focus on syntax usually degenerates into an argument over personal taste.*

I believe that focusing on what can be expressed by forward parameter declarations, and equally importantly, *what cannot be expressed*, might lead some members of the committee to prefer the alternative syntax proposed by this paper. This alternative syntax is neither a subset nor a superset of that supported by GCC, but it is compatible for most common use cases.

Prior art

GCC extension

The sole example of a parameter forward declaration given in the GCC manual [1] does not show multiple parameter forward declarations:

```
struct entry
tester (int len; char data[len][len], int len)
{
  /* … */
}
```

The syntax and constraints of the extension are then described as follows:

> *You can write any number of such parameter forward declarations in the parameter list. They can be separated by commas or semicolons, but the last one must end with a semicolon, which is followed by the "real" parameter declarations. Each forward declaration must match a "real" declaration in parameter name and data type.*

Although parameter forward declarations have different semantics from "real" parameter declarations, both are said to belong to the same list. Semantically, there are two lists: forward declarations before the final semicolon and "real" declarations afterwards.

No explanation is given for the flexibility of the extension's syntax. Both forms are acceptable to GCC 14.2.0, but neither is acceptable to Clang 19.1.0.

All three of the following declarations are equivalent [3]:

```
void tester(int x, int y, int z;
            char data[z][y][x], int x, int y, int z);

void tester(int x; int y; int z;
            char data[z][y][x], int x, int y, int z);

void tester(int x; int y, int z;
            char data[z][y][x], int x, int y, int z);
```

Only by locating the last semicolon in each parameter type list can these declarations be parsed or understood. Any semicolon can be followed by "real" parameter declarations or by more parameter forward declarations. This ambiguity is uncharacteristic of C.

The number of semicolons within the brackets of a `for` statement can be used to get a sense of the inherent complexity of that construct. That can easily be distinguished from additional complexity resulting from the programmer's use of the language, such using the comma operator to evaluate three expressions after each execution of a loop's body instead of one.

In contrast, I get no sense of the inherent complexity of a function declaration that contains a variable number of semicolons: I can't tell whether it needs to have as many clauses as it has, or whether it could have additional clauses. Nor can I learn the semantics of those clauses from observation, because they don't fit any fixed template.

GCC 14.2.0 doesn't accept an empty parameter forward declaration list [4]:

```
void invalid(; int x, int y, char data[y][x]);
```

This seems inconsistent with the validity of a `for` statement with an empty first clause [5]:

```
void foo(void)
{
    for  (; 0; ) {
    }
}
```

It also seems inconsistent with an empty first clause of a selection statement in C++ [6]:

```
void foo()
{
    if (; 0) {
    }
    switch (; 0) {
    }
}
```

An unrelated point of interest is that GCC 14.2.0 evaluates the size expression of a parameter forward declaration of variably modified type separately from the size expression of "real" declaration of the same parameter, and similar for operands of `typeof` [7]:

```
static int m;
void foo(char bar[m++], typeof(char [m+=5]) baz;
         char bar[m+=2], typeof(char [m+=10]) baz) {
    // increments m by 18
}
```

This is evidenced by the translated code:

```
foo:
        movw    r3, #:lower16:.LANCHOR0
        movt    r3, #:upper16:.LANCHOR0
        ldr     r2, [r3]
        adds    r2, r2, #18
        str     r2, [r3]
        bx      lr
```

N2780 Forward Declaration of Parameters (2021-07-11)

N2780 [8] was the original proposal to partially standardize the GCC extension, which justified its preferred choice of syntax as follows:

> *GCC supports comma and semicolon to separate multiple forward declarations. Here we propose to allow only the semicolon, because then it is known directly after parsing of each parameter whether it is a forward declaration or already the first real parameter declaration.*

It proposed the syntax:

> *parameter-type-list:*
> *parameter-forward-declaration* ***;*** *parameter-type-list*
> *parameter-list*
> *parameter-list* ***,*** ***…***
>
> *parameter-forward-declaration:*
> ***attribute-specifier-sequence***$_{opt}$ *declaration-specifiers declarator*

Where:

> *parameter-declaration:*
> *attribute-specifier-sequence*$_{opt}$ *declaration-specifiers declarator*
> *attribute-specifier-sequence*$_{opt}$ *declaration-specifiers abstract-declarator*$_{opt}$
>
> *declaration-specifiers:*
> *declaration-specifier* ***attribute-specifier-sequence***$_{opt}$
> *declaration-specifier declaration-specifiers*
>
> *declaration-specifier:*
> *storage-class-specifier*
> ***type-specifier-qualifier***
> *function-specifier*
>
> *declarator:*
> *pointer*$_{opt}$ *direct-declarator*
>
> *direct-declarator:*
> ***identifier attribute-specifier-sequence***$_{opt}$
> ***(*** *declarator* ***)***
> *array-declarator attribute-specifier-sequence*$_{opt}$
> *function-declarator attribute-specifier-sequence*$_{opt}$

Note that the rule for *parameter-forward-declaration* is identical to one of the alternatives for *parameter-declaration*. (**Bold text** indicates syntax elements that were later combined more directly by N3140 [10] into an alternative rule for *parameter-forward-declaration*.)

N2780 [8] also proposed a new constraint and altered semantics for function declarators:

> *An identifier declared in a parameter forward declaration shall also be declared in the parameter list.*
>
> *Parameter forward declarations may provide forward declarations of the identifiers of the parameters (for use in size expressions).*

However, the feature did not receive sufficient consensus to become part of C23.

N3121 Forward Declaration of Parameters v2 (2023-04-22)

N3121 [9] did not alter the proposed syntax.

It did propose changes to 6.2.2 Linkages of identifiers:

> *With the exception of parameter forward declarations and their respective parameter declarations, each declaration of an identifier with no linkage denotes a unique entity.*

And to 6.2.7 Compatible type and composite type:

> *The type of a parameter with a parameter forward declaration becomes the composite type at the parameter declaration.*

And to the semantics of function definitions:

> *On entry to the function, the size expressions of parameter declarations and forward parameter declarations 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.*

It also added an extra argument in favour of the proposed syntax, which was absent from the original paper:

> *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.*

N3140 Parameter Forward Declarations (2023-06-22)

N3140 [10] was co-authored by Jens Gustedt. It added the following constraint for function declarators:

> *The types specified in the parameter forward declaration and the corresponding parameter declaration shall be compatible.*

And new semantics for function declarators:

> *It is implementation-defined whether forward declarations with types other than integer types or with other syntactic forms for the declaration are accepted. If other types or syntactic forms are accepted, the behavior is implementation-defined.*

The syntax originally proposed by N2780 [8], was amended to:

> *parameter-forward-declaration:*
> *attribute-specifier-sequence$_{opt}$ specifier-qualifier-list identifier attribute-specifier-sequence$_{opt}$*

Where:

> *specifier-qualifier-list:*
> *type-specifier-qualifier attribute-specifier-sequence$_{opt}$*
> *type-specifier-qualifier specifier-qualifier-list*

This alternative syntax allows only a subset of declarators allowed by Martin's syntax (to exclude derived types and non-integer types).

N3207 Forward Declaration of Parameters v3 (2023-12-14)

N3207 [11] was based on N3121 [9] rather than on N3140 [10]:

> *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.*

The proposed syntax was changed from the original proposal, N2780 [8], to:

> *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 ;*

Instead of defining a rule for *parameter-forward-declaration* that is identical to one of the alternatives for *parameter-declaration*, the existing rule *parameter-declaration* is now reused for forward parameter declarations.

Instead of recursing through *parameter-type-list* when multiple forward parameter declarations are present, *parameter-forward-declaration-list* is now a separate rule.

The wording of the proposed new constraint on function declarators was tightened from 'also' to 'exactly once' and updated to include a variant of the compatible-types constraint proposed in N3140 [10]:

> *An identifier declared by 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.*

GCC 14.2.0 doesn't enforce this constraint and I don't know how hard it might be to implement.

Examples of usage were added, including the following examples of valid usage:

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

The inclusion of the above examples, and the fact that they correspond to the syntax element *parameter-declaration* despite not being parameter declarations, is potentially confusing. It raises questions about the scope of the feature and whether there is implementation and usage experience for the whole feature, or only for forward parameter declarations.

It is not clear to me what advantage, other than aesthetic, forward struct declarations have over:

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

The above snippet is already accepted by GCC 14.2.0, but Clang 19.1.0 produces a diagnostic [12].

The non-parameter-declaration examples in N3207 [11] are not accepted by GCC 14.2.0 [13], which produces diagnostics:

```
<source>:1:8: error: parameter '({anonymous})' has just a forward
declaration
    1 | void c(struct bar { char buf[10]; }; struct bar *dst, const
struct bar *src);
      |        ^~~~~~~~~~~~~~~~~~~~~~~~~~
<source>:1:15: warning: 'struct bar' declared inside parameter list
will not be visible outside of this definition or declaration
    1 | void c(struct bar { char buf[10]; }; struct bar *dst, const
struct bar *src);
      |               ^~~
<source>:2:8: error: parameter '({anonymous})' has just a forward
declaration
    2 | void e(struct bar { int x; }; struct bar *dst, const struct
bar *src);
      |        ^~~~~~~~~~~~~~~~~~~~
<source>:2:15: warning: 'struct bar' declared inside parameter list
will not be visible outside of this definition or declaration
    2 | void e(struct bar { int x; }; struct bar *dst, const struct
bar *src);
      |               ^~~
```

Their inclusion presumably relates to the observation that:

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

GCC 14.2.0 does accept a similar declaration as the first clause of a `for` statement:

```
void foo(void)
{
    for (struct bar { char buf[10]; }; 0; )
    {
    }
}
```

But Clang 19.1.0 produces a diagnostic [14]:

```
<source>:3:17: error: non-variable declaration in 'for' loop
    3 |     for (struct bar { char buf[10]; }; 0; )
      |                 ^
```

It's unclear whether GCC's support for such declarations is an extension, or Clang's lack of support for them is a bug.

N3394 Forward Declaration of Parameters v4 (2024-11-23)

N3394 [15] is the latest proposal to partially standardize the GCC extension:

> *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.*

There were no further changes to the proposed syntax.

An existing constraint on declarations (6.7.1) is modified:

> *~~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.*

This proposed modification appears to relate to the non-parameter-declaration examples added in N3207 [11] but it doesn't make sense to require every *parameter-declaration* in a *parameter-forward-declaration-list* to include a `struct`, `union` or `enum` specifier in its *declaration-specifiers*. Perhaps the intent was to only require such a specifier when the *parameter-declaration* has the *abstract-declarator*$_{opt}$ ("empty parameter forward declaration") form.

Summarized history of proposed syntaxes

1. Restricted to exclude abstract parameter declarations.
2. Restricted to exclude abstract parameter declarations, derived types and non-integer types.
3. Same syntax as a parameter declaration.

Precedents for use of semicolons

Semicolons within round brackets

Traditionally, the only part of C's syntax that used semicolons between round brackets was the `for` statement. The complexity of understanding and parsing `for` loops was alleviated by the fact that they **always come in three parts**, regardless of whether all parts had been provided by the programmer. From a terminology standpoint, K&R (2nd ed.) referred to 'parts' and 'expressions' of a `for` statement but not 'clauses.'

In C23, the only syntax element that uses semicolons between round brackets is the `for` statement (6.8.6.1):

> *for (expression$_{opt}$; expression$_{opt}$; expression$_{opt}$) secondary-block*
>
> *for (declaration expression$_{opt}$; expression$_{opt}$) secondary-block*

Where:

> *declaration:*
> *declaration-specifiers init-declarator-list$_{opt}$;*
> *attribute-specifier-sequence declaration-specifiers init-declarator-list ;*
> *static_assert-declaration*
> *attribute-declaration*
>
> *static_assert-declaration:*
> *static_assert (constant-expression , string-literal) ;*
> *static_assert (constant-expression) ;*
>
> *attribute-declaration:*
> *attribute-specifier-sequence ;*

(When the first clause of a `for` statement is parsed as *declaration expression$_{opt}$*, it yields two clauses separated by a semicolon. Thus, a `for` statement still has exactly three clauses.)

The syntax of the `for` statement is also described by the ISO C standard (6.8.6.4) as:

> *for (clause-1 ; expression-2 ; expression-3) statement*

When *clause-1* is a *declaration*, it can declare multiple objects derived from a type specified by its *declaration-specifiers* [16]:

```
int main(void)
{
    for (int a, *b = &a; ; ) {
    }
    return 0;
}
```

N3388 [17] extended the syntax of `switch` and `if` statements in C2Y:

> *selection-statement:*
> `if (` *selection-header* `)` *secondary-block*
> `if (` *selection-header* `)` *secondary-block* `else` *secondary-block*
> `switch (` *selection-header* `)` *secondary-block*
>
> *selection-header:*
> *expression*
> *declaration expression*
> *simple-declaration*
>
> *simple-declaration:*
> *attribute-specifier-sequence*$_{opt}$ *declaration-specifiers declarator* = *initializer*

(When a *selection-header* is parsed as *declaration expression*, it yields two clauses separated by a semicolon; no semicolon is present in the other two cases. Thus, `if` and `switch` have exactly one or two clauses.)

When *selection-header* incorporates a *declaration*, it can declare multiple objects derived from the type specified by its declaration-specifiers [18]:

```
int main(void)
{
    if (int a, *b = &a; 0) {
    }
    switch (int a, *b = &a; 0) {
    }
    return 0;
}
```

Even after the changes in N3388 have been integrated into ISO C, **none of these iteration or selection statements allow an unlimited number of semicolons** such as GCC allows for forward parameter declarations.

Semicolons outside of round brackets

Semicolons terminate a *declaration* or (through an *unlabeled-statement*) *expression-statement* or *jump-statement*. These can be combined into a *block-item-list* through *block-item*.

However, it's debatable whether the semicolon is used as a list item separator in this context because an *unlabeled-statement* can also be a *primary-block*, and primary blocks (except for `do` loops) are not terminated by a semicolon [19]:

```
int main(void)
{
    {
        int a, *b = &a;
    }
    int c;
    {
        double c, d[10];
    }
    return 0;
}
```

The only unarguable use of semicolon as a list item separator in ISO C is its role terminating each *member-declaration* in a *member-declaration-list* [20]:

```
struct foo {
  int a, *b;
  double c, d[10];
};
```

Each *member-declaration* can declare multiple members derived from the type specified by its *specifier-qualifier-list*, just as each *declaration* in the previous example can declare multiple objects derived from the type specified by its *declaration-specifiers*.

Member declarations cannot have initialisers.

Semicolons in English grammar

I do not think the committee should entirely disregard natural language precedents for usage of commas and semicolons.

In English grammar, according to the University of Adelaide [21]:

> *The semicolon both separates two independent clauses, and links them at the same time.*

This sounds very much like its usage in `if` and `switch` statements.

Also, according to the University of Wisconsin – Madison Writer's Handbook [22] one should:

> *Use a semicolon between items in a list or series if any of the items contain commas.*

This could also describe a function declaration, if we admit the existence of two items: a list of forward parameter declarations and a separate list of "real" parameter declarations:

```
void tester (int x, int y;
             char data[y][x], int x, int y);
```

Summary of precedents

- No precedent for allowing an unlimited number of semicolons between brackets.
- No precedent for restricting the number of declarators to one in a declaration terminated by a semicolon.
- The only precedent for disallowing initialisers in a declaration terminated by a semicolon is `struct` or `union` member declarations (which are not bracketed).
- The only precedent for semicolon as a list item separator is `struct` or `union` member declarations (which are not bracketed).

Likely programmer errors

Programmers familiar with existing precedents are likely to make wrong assumptions about forward parameter declarations.

They may assume that they can only forward-declare one parameter, since semicolons between brackets have hitherto always delineated a fixed number of fields.

They may assume that they can provide an initializer for a declaration [23]:

```
int foo(int x = 0; char data[x], int x) {
}
```

They may assume that they can declare multiple objects derived from the specified type [24]:

```
int foo(int x, *y; char data[x], int x) {
}
```

They may assume that they can declare objects with function scope that are not parameters [25]:

```
int foo(int n; char data[]) {
    n = 0;
}
```

The root cause of all these errors is the same thing that makes forward parameter declarations deceptively cosy and familiar — their resemblance to declarations in other contexts. Abuse of the forward parameter declaration list to declare `struct` types (as proposed by N3394 [15]) seems to stem from the same confusion.

Some blame can be attributed to ANSI's syntax for function declarations, which diverged needlessly from C's otherwise flexible and expressive declaration syntax. If the following were a valid function declaration, there could have been no question of reusing semicolons in such an ambiguous way:

```
struct foo {int x, y, z;};
void foo(int x, y, z;);
```

Short of inventing new syntax different from the GCC extension, there is no perfect solution. However, there is an important mitigation that the committee could adopt: standardise the GCC extension in a form that reduces the resemblance between parameter forward declarations and declarations in other contexts.

Proposed mitigation

My proposed mitigation is threefold:

- Use semicolon only to separate any parameter forward declarations from "real" parameter declarations. GCC already supports this.
- Use comma to separate parameter forward declarations. GCC already supports this.
- Allow a semicolon in the absence of parameter forward declarations. GCC does not support this, which makes its syntax inconsistent with `for` loops and selection statements (as previously noted).

I believe these changes make complex use cases easier to understand and align the extended parameter list syntax with the rest of the language. They also simplify the syntax description.

The limitations and peculiarities of C's parameter declaration syntax are already well-known. By making clear that parameter forward declarations **are** parameter declarations, a lot of the ambiguities about what they can express, and how, simply vanish.

Sequence points

According to Annex C, there is a sequence point:

> *Between the evaluation of a full expression and the next full expression to be evaluated. The following are full expressions:* ***a full declarator for a variably modified type****; an initializer that is not part of a compound literal (6.7.11); the expression in an expression statement (6.8.4);* ***the controlling expression of a selection statement*** *(if or switch) (6.8.5); the controlling expression of a while or do statement (6.8.6);* ***each of the (optional) expressions of a for statement*** *(6.8.6.4); the (optional) expression in a return statement (6.8.7.5).*

That suggests the output of the following program (which uses a GCC extension) might be predictable [26]:

```
#include <stdio.h>

int n = 0;
int p[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};

int main(void)
{
    struct bar {
        int x[n += 3];
        int y[p[n]];
    };
    struct bar b;
    printf("%zu", sizeof(b.y)); // 12
    return 0;
}
```

The question is of limited interest because 6.7.3.2 p11 in the ISO C standard says:

> *A member of a structure or union can have any complete object type other than a variably modified type.*

However, the output of the following program might also be predictable [27]:

```
#include <stdio.h>

int n = 0;
int p[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};

void foo(int (*x)[n += 3], int (*y)[p[n]])
{
        printf("%zu", sizeof(*y)); // 12
}

int main(void)
{
    foo(NULL, NULL);
    return 0;
}
```

6.9.2p11 does not specify an order of evaluation for full expressions that are part of a list of parameter declarations. An order could be imposed between forward and “real” parameter declarations but a holistic approach to evaluation of expressions in declarators might be better.

Proposed wording changes

The proposed wording is a diff from the N3299 working draft [28]. Some of the text is taken from N3394 [15]. Green text is new text, while ~~red~~ text is deleted text.

A new syntax rule for *parameter-forward-declaration-list* isn't strictly necessary, but there is precedent for redundant rules where they serve to clarify the language's semantics (e.g., an instance of *typedef-name* is simply an *identifier*).

6.2.2 Linkages of identifiers

1 An identifier declared in different scopes or in the same scope more than once can be made to refer to the same object or function by a process called *linkage*. [20)] There are three kinds of linkage: external, internal, and none.

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. ~~Each~~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

4 If any of the original types satisfies all requirements of the composite type, it is unspecified whether the composite type is one of these types or a different type that satisfies the requirements.[47)]

5 For an identifier with internal or external linkage declared in a scope in which a prior declaration of that identifier is visible,[48)] 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 Declarators

6.7.7.1 General

Syntax

function-declarator:
direct-declarator (*parameter-type-list*$_{opt}$)
direct-declarator (*parameter-forward-declaration-list*$_{opt}$; *parameter-type-list*$_{opt}$)

parameter-forward-declaration-list:
parameter-list

6.7.7.4 Function declarators

Constraints

1 A function declarator shall not specify a return type that is a function type or an array type.

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

3 After adjustment, the parameters in a parameter forward declaration list or parameter type list in a function declarator that is part of a definition of that function shall not have incomplete type.

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

Semantics

5 A parameter type list specifies the types of, and can declare identifiers for, the parameters of the function.

6 Parameter declarations in a parameter forward declaration list provide forward declarations of the identifiers of the parameters (useful for size expressions).

~~6~~7 A declaration of a parameter as "array of type" shall be adjusted to "qualified pointer to type", where the type qualifiers (if any) are those specified within the `[` and `]` of the array type derivation. If the keyword `static` also appears within the `[` and `]` of the array type derivation, then for each call to the function, the value of the corresponding actual argument shall provide access to the first element of an array with at least as many elements as specified by the size expression.

21 EXAMPLE 6 The following declarations illustrate valid use of parameter forward declaration lists:

```
void a(int x, int y; double matrix[y][x], int x, int y);
void b(double (*p)[3][*]; double (*p)[*][4]);
void c(double (*p)[3][4]); // compatible redeclaration
void d(int n, long (*in)[3 * n];
       char buf[sizeof(*in)], long (*in)[3 * n], int n);
void e(int x; int x);
void f(signed int x; int x);
void g( ; int x);
void h( ; );
void i( ; ...);
```

22 EXAMPLE 7 The following declarations illustrate invalid use of parameter forward declaration lists:

```
void k(int x; const int x); // incompatible types of 'x'
void l(int x[3]; int x[4]); // incompatible types before adjustment
void m(int *x; int x[]); // incompatible types before adjustment
void n(int x, int x; int x); // invalid redeclaration of 'x'
void o(int x; int x, int x); // invalid redeclaration of 'x'
void p(register int x; int x); // different storage class
void q(int x; int y); // 'x' is not in the parameter type list
void r(int x; int x; int x); // invalid syntax
```

6.9.2 Function definitions

11 On entry to the function, the size expressions of each variably modified parameter and typeof operators used in declarations of parameters 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.)

12 Size expressions of variably modified types used in a parameter forward declaration list are evaluated separately from size expressions of variably modified types used in a parameter type list.

~~12~~13 After all parameters have been assigned, the compound statement of the function body is executed.

Alternative proposed wording change

6.9.2 Function definitions

11 On entry to the function, the size expressions of each variably modified parameter and typeof operators used in declarations of parameters 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.)

12 Size expressions of variably modified types used in a parameter forward declaration list are evaluated before size expressions of variably modified types used in a parameter type list.

~~12~~13 After all parameters have been assigned, the compound statement of the function body is executed.

C.1 Known Sequence Points

1 The following are the sequence points described in 5.2.2.4:

— Between the evaluation of size expressions of variably modified types used in a parameter forward declaration list and the evaluation of size expressions of variably modified types used in a parameter type list (6.9.2).

Rebuttal of arguments for the syntax in N3394

N3394 [15] makes two main arguments for its proposed syntax, which I will try to address here.

Visibility of forward parameter 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.*

Does that matter? In the proposed grammar, both *parameter-forward-declaration-list* and *parameter-list* are composed of instances of *parameter-declaration*. An instance of *parameter-forward-declaration-list* probably should be parsed the same as a normal list of parameters; if it isn't, then there is a possibility of unwanted divergence.

Let's imagine someone reading a program does mistake a forward parameter declaration for a "real" parameter declaration. They are required to have compatible types before adjustment and declare the same identifier, which is more stringent than the requirement for function prototype declarators to be compatible. Programmers seem to trust parameter declarations in function declarators.

The real issue is the number and order of arguments passed by a caller, not their types or identifiers. I find it hard to believe that someone writing code to call a function would not read to the end of its prototype, given that an expression must be specified for every parameter. It follows that the main risk is when modifying code, not when writing it.

Passing too many arguments is a constraint violation, but passing parameters in the wrong order is often impossible to diagnose:

```
void tester(int x, int y, int z; int z, int y, int x);
tester(x, y, z); // whoops!
```

Neighbouring parameter declarations of the same type are already faulted by some linters for this reason. I do not believe the above example is representative of real functions.

It's easy to format code to reduce the likelihood of mistaking forward declarations for "real" ones:

```
void tester(int x, int y, int z;
            char data[z][y][x], int x, int y, int z);
```

When reading the above code, I can tell for certain that I have reached the end of a forward parameter declaration list upon encountering the semicolon. Before that, I must assume that all declarations are forward parameter declarations, but that's no different from my presumption at the beginning of any parameter list.

It's also easy to format code to increase the likelihood of forward declarations being mistaken for "real" ones even if the syntax proposed by N3394 [15] is used:

```
void tester(int x;
            int y; int z; char data[z][y][x], int x, int y, int z);
```

When reading the above code, I can tell at the end of the declaration of `x` that it was a forward parameter declaration, but I cannot assume that the following parameter will be a "real" parameter declaration because the semicolon **doesn't delineate semantically different constructs**.

If the only meaning of the semicolon is "the previous declaration was a forward parameter declaration", it should be possible to interleave forward declarations and "real" declarations!

```
void tester(int x; char data[x], int x,
            int y; int z; char data[y][z], int y, int z);
```

Robust against typos

> *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.*

Consider the following declaration in this paper's preferred syntax:

```
void tester(int x, int y, int z;
            char data[z][y][x], int x, int y, int z);
```

It might be mistakenly written as follows, using a semicolon instead of a comma in the forward parameter declaration list:

```
void tester(int x, int y; int z;
            char data[z][y][x], int x, int y, int z);
```

This would result in a syntax error.

It might be mistakenly written as follows, using a semicolon instead of a comma in the parameter type list:

```
void tester(int x, int y, int z;
            char data[z][y][x]; int x, int y, int z);
```

This would also result in a syntax error.

It might be mistakenly written as follows, using a comma instead of a semicolon to terminate the forward parameter declaration list:

```
void tester(int x, int y, int z,
            char data[z][y][x], int x, int y, int z);
```

This would result in a constraint violation because of 6.7.1p4:

> *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:*

In summary, the syntax proposed by this paper is no less robust against typos than the syntax proposed by N3394 [15]. Accidental use of a semicolon instead of a comma can always be diagnosed as a syntax error without recourse to constraints or semantics. In contrast, the syntax proposed by N3394 does not allow such misuse to be diagnosed as a syntax error.

Acknowledgements

I would like to acknowledge the work of Martin Uecker, Jens Gustedt, Joseph Myers and others towards standardization of this feature in a different form.

References

[1] Using the GNU Compiler Collection (GCC): Extensions to the C Language Family: Arrays of Variable Length
https://gcc.gnu.org/onlinedocs/gcc/Variable-Length.html#Arrays-of-Variable-Length

[2] The Design and Evolution of C++, Bjarne Stroustrup, 1994.

[3] Compiler Explorer
https://godbolt.org/z/rGqqE4ns9

[4] Compiler Explorer
https://godbolt.org/z/W8GWYbjeo

[5] Compiler Explorer
https://godbolt.org/z/vKzqa4E1Y

[6] Compiler Explorer
https://godbolt.org/z/6WTTxj34x

[7] Compiler Explorer
https://godbolt.org/z/qTjzxaaKh

[8] N2780 Forward Declaration of Parameters, Martin Uecker, 2021-07-11.
https://www.open-std.org/jtc1/sc22/wg14/www/docs/n2780.pdf

[9] N3121 Forward Declaration of Parameters v2 (updates N2780), Martin Uecker, 2023-04-22
https://www.open-std.org/jtc1/sc22/wg14/www/docs/n3121.pdf

[10] N3140 Parameter Forward Declarations, Martin Uecker, Jens Gustedt, 2023-06-22
https://www.open-std.org/jtc1/sc22/wg14/www/docs/n3140.pdf

[11] N3207 Forward Declaration of Parameters v3 (updates N3121), Martin Uecker, 2023-12-14
https://www.open-std.org/jtc1/sc22/wg14/www/docs/n3207.pdf

[12] Compiler Explorer
https://godbolt.org/z/caYdx1Ejo

[13] Compiler Explorer
https://godbolt.org/z/n5TEfTMM6

[14] Compiler Explorer
https://godbolt.org/z/EMfa11jEa

[15] N3394 Forward Declaration of Parameters v4 (updates N3207), Martin Uecker, 2024-11-23
https://www.open-std.org/jtc1/sc22/wg14/www/docs/n3394.pdf

[16] Compiler Explorer
https://godbolt.org/z/zdY5rMPYx

[17] N3388 `if` declarations, v4, Alex Celeste, 2024-11-11
https://www.open-std.org/jtc1/sc22/wg14/www/docs/n3388.htm

[18] Compiler Explorer
https://godbolt.org/z/Kv4TMqPjz

[19] Compiler Explorer
https://godbolt.org/z/T171nMM5x

[20] Compiler Explorer
https://godbolt.org/z/arhEKb44d

[21] The University of Adelaide – How to Use Semicolons
https://www.adelaide.edu.au/writingcentre/ua/media/56/learningguide-semicolons.pdf

[22] University of Wisconsin – Madison Writer's Handbook
https://writing.wisc.edu/handbook/semicolons/

[23] Compiler Explorer
https://godbolt.org/z/snfs1frs6

[24] Compiler Explorer
https://godbolt.org/z/f6a6hj8K3

[25] Compiler Explorer
https://godbolt.org/z/q63a4G5xP

[26] Compiler Explorer
https://godbolt.org/z/rqafhf8qP

[27] Compiler Explorer
https://godbolt.org/z/nPosxhbT4

[28] N3299 Working Draft C2y Post C23-Publication, Meneide, 2024-07-28
https://www.open-std.org/jtc1/sc22/wg14/www/docs/n3299.pdf