

The New C Standard (Excerpted material)

An Economic and Cultural Commentary

Derek M. Jones

derek@knosof.co.uk

6.7.5.3 Function declarators (including prototypes)

Constraints

function
declarator return
type

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

1592

Commentary

function
definition
return type

The wording given for function definitions is slightly different and the discussion given for that sentence is applicable here.

Other Languages

It is quite common for languages to support functions returning array types. Support for functions returning function types is much less common (e.g., Lisp, Algol 68).

parameter
storage-class

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

1593

Commentary

block scope
terminates
automatic
storage duration
register
storage-class

Parameters have block scope and automatic storage duration. Like any other object having block scope the developer might want to suggest that accesses to it be optimized. Use of any other storage-class specifier in a parameter declaration has no obvious semantics.

C++

7.1.1p2 *The **auto** or **register** specifiers can be applied only to names of objects declared in a block (6.3) or to function parameters (8.4).*

C source code developed using a C++ translator may contain the storage-class specifier **auto** applied to a parameter. However, usage of this keyword is rare (see Table ??) and in practice it is very unlikely to occur in this context.

The C++ Standard covers the other two cases with rather sloppy wording.

7.1.1p4 *There can be no **static** function declarations within a block, nor any **static** function parameters.*

7.1.1p5 *The **extern** specifier cannot be used in the declaration of class members or function parameters.*

An identifier list in a function declarator that is not part of a definition of that function shall be empty.

1594

Commentary

function
declarator¹⁶⁰⁸
empty list

An identifier list provides little useful information in a function declaration that is not also a definition. The issue of an empty identifier list is discussed elsewhere.

C++

The C++ Standard does not support the old style of C function declarations.

Example

```

1 extern int f_1(x, y); /* Constraint violation. */
2
3 int (*f_2(a, b))(x, y) /* The subject of an outstanding DR. */
4 { /* ... */ }
```

- 1595 After adjustment, the parameters in a parameter type list in a function declarator that is part of a definition of that function shall not have incomplete type.

parameter
adjustment
in definition

Commentary

Adjustment refers to array and function types, which are implicitly converted to a pointer to their first element (there is no such conversion for incomplete structure or union types). After adjustment, the types of parameters have the same requirements as the types of other block scope objects.

¹⁵⁹⁸ array type
adjust to pointer to

C translators need to know the number of bytes of storage needed by each parameter. Incomplete types do not have a known, at the point of declaration, size. Arguments having an array type are passed as the address of their first element, so their size is not needed.

size needed

C90

The C90 Standard did not explicitly specify that the check on the parameter type being incomplete occurred “after adjustment”.

C++

The C++ Standard allows a few exceptions to the general C requirement:

If the type of a parameter includes a type of the form “pointer to array of unknown bound of T” or “reference to array of unknown bound of T,” the program is ill-formed.⁸⁷

8.3.5p6

This excludes parameters of type “ptr-arr-seq T2” where T2 is “pointer to array of unknown bound of T” and where ptr-arr-seq means any sequence of “pointer to” and “array of” derived declarator types. This exclusion applies to the parameters of the function, and if a parameter is a pointer to function or pointer to member function then to its parameters also, etc.

Footnote 87

The parameter list (void) is equivalent to the empty parameter list. Except for this special case, void shall not be a parameter type (though types derived from void, such as void, can).*

8.3.5p2

The type of a parameter or the return type for a function declaration that is not a definition may be an incomplete class type.

8.3.5p6

```

1 void f(struct s1_tag ** p1) /* incomplete type, constraint violation */
2                               // defined behavior
3 {
4   struct s2_tag **loc; /* Size not needed, so permitted. */
5 }
```

Semantics

- 1596 If, in the declaration “T D1”, D1 has the form

D(parameter-type-list)

or

D(*identifier-list*_{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 function returning T*”.

Commentary

This defines the terms *derived-declarator-type-list* and *derived-declarator-type-list*. They are very rarely used outside of the standard, even by the Committee.

C++

The form supported by the C++ Standard is:

8.3.5p1 **D1** (*parameter-declaration-clause*) *cv-qualifier-seq*_{opt} *exception-specification*_{opt}

The term used for the identifier in the C++ Standard is:

8.3.5p1 “*derived-declarator-type-list function of (parameter-declaration-clause) cv-qualifier-seq*^{opt} *returning T*”;

The old-style function definition **D**(*identifier-list*_{opt}) is not supported in C++.

8.3.5p2 *If the parameter-declaration-list is empty, the function takes no arguments. The parameter list (void) is equivalent to the empty parameter list.*

The C syntax treats **D**() as an instance of an empty identifier-list, while the C++ syntax treats it as an empty *parameter-type-list*. Using a C++ translator to translate source containing this form of function declaration may result a diagnostic being generated when the declared function is called (if it specifies any arguments).

Coding Guidelines

If the guideline recommendation specifying the use of prototypes is followed the identifier list form will not occur in new source code. However, it may occur in existing source code and the cost/benefit issues associated with changing this existing usage are discussed elsewhere.

function ??
declaration
use prototype

prototypes
cost/benefit

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

1597

Commentary

Rationale There was considerable debate about whether to maintain the current lexical ordering rules for variable length array parameters in function definitions. For example, the following old-style declaration

```
void f(double a[*][*], int n);

void f(a, n)
    int n;
    double a[n][n];
{
    // ...
}
```

cannot be expressed with a definition that has a parameter type list as in

```
void f(double a[n][n], int n) // error
{
    /* ... */
}
```

Previously, programmers did not need to concern themselves with the order in which formal parameters are specified, and one common programming style is to declare the most important parameters first. With Standard C's lexical ordering rules, the declaration of `a` would force `n` to be undefined or captured by an outside declaration. The possibility of allowing the scope of parameter `n` to extend to the beginning of the parameter-type-list was explored (relaxed lexical ordering), which would allow the size of parameter `a` to be defined in terms of parameter `n`, and could help convert a Fortran library routine into a C function. Such a change to the lexical ordering rules is not considered to be in the "Spirit of C", however. This is an unforeseen side effect of Standard C prototype syntax.

C++

The parameter-declaration-clause determines the arguments that can be specified, and their processing, when the function is called.

8.3.5p2

Other Languages

Many languages allow parameters to be defined to have types in the same way that they support the definition of objects. Some languages provide no explicit mechanism for defining the types of the parameters (e.g., Perl).

Coding Guidelines

Some coding guideline documents recommend that no identifiers be given for the parameters in a function prototype declaration. The rationale is that such identifiers could also match some earlier defined macro name. In practice, the effect of such a match is likely to be a syntax violation. Even if this does not occur the spelling of the identifier appearing in the prototype declaration is not significant. Whatever happens there will not be a quiet change in program behavior. The cost of a syntax violation (the identifier spelling will need to be changed) has to be balanced against the benefit of including an identifier in the parameter type list.

Some coding guideline documents recommend that any identifiers given for the parameters in a function prototype declaration have the same spelling as those given in the function definition. Such usage may provide readers with a reminder of information about what the parameter denotes. A comment could provide more specific information. Using identifiers in this way also provides visible information that might help detect changes to a functions interface (e.g., a change in the order of the parameters).

There does not appear to be compelling evidence for any of these options providing sufficient cost/benefit for a guideline recommendation to be worthwhile.

Example

```
1 #define abc xyz
2
3 void f_1(int abc);
4
5 int cost_weight_ratio(int cost, int weight);
6
7 int cost_weight_ratio(int weight, int cost)
8 {
9     return cost / weight;
10 }
```

array type
adjust to pointer
to

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. 1598

Commentary

This is a requirement on the implementation.

```

1 void f(      int a1[10],      /* equivalent to      int *      a1 */
2           const int a2[10],  /* equivalent to const int *      a2 */
3           int a3[const 10], /* equivalent to      int * const a3 */
4           const int a4[const 10]) /* equivalent to const int * const a4 */
5 { /* ... */ }
```

Occurrences of an object, not declared as a parameter, having an array type are implicitly converted to a pointer type in most contexts. The parameter declaration in:

```

1 void f(int a[const 10][const 20])
2 { /* ... */ }
```

is permitted by the syntax, but violates a constraint.

C90

Support for type qualifiers between [and], and the consequences of their use, is new in C99.

C++

This adjustment is performed in C++ (8.3.5p3) but the standard does not support the appearance of type qualifiers between [and].

Source containing type qualifiers between [and] will cause a C++ translator to generate a diagnostic.

Other Languages

Using qualifiers within the [and] of an array declaration may be unique to C.

Coding Guidelines

A qualifier appearing outside of [and] qualifies the element type, not the array type. For non-parameter declarations this distinction is not significant, the possible consequences are the same. However, the implicit conversion to pointer type, that occurs for parameters having an array type, means that the distinction is significant in this case. Experience shows that developers are not always aware of the consequences of this adjustment to parameters having an array type. The following are two of the consequences of using a qualifier in the incorrect belief that the array type, rather than the element type, will be qualified:

- The **volatile** qualifier— the final effect is very likely to be the intended effect (wanting to **volatile** qualify an object having a pointer type is much rarer than applying such a qualifier to the object it points at).
- The **const** qualifier— attempts to modify the pointed-to objects will cause a translator diagnostic to be issued and attempts to modify the parameter itself does not require a translator to issue a diagnostic.

Support for qualifiers appearing between [and] is new in C99 and there is insufficient experience in their use to know whether any guideline recommendation is cost effective.

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. 1599

Commentary

function
declarator
static

It would be a significant advantage on some systems for the translator to initiate, at the beginning of the function, prefetches or loads of the arrays that will be referenced through the parameters.

The use of the keyword **static** within the [and] of an array type is a guarantee from the developer to the translator. The translator can assume that at least the value of the parameter will not be NULL and that storage for at least the specified number of elements will be available.

Rules for forming the composite type of function types, one or more of which included the keyword **static**, were given by the response to DR #237.

The effect is as if all of the declarations had used static and the largest size value used by any of them. Each declaration imposes requirements on all calls to the function in the program; the only way to meet all of these requirements is to always provide pointers to as many objects as the largest such value requires.

DR #237

```

1 void WG14_DR_237(int x[static 10]);
2 void WG14_DR_237(int x[static 5]);
3
4 void WG14_DR_237(int x[1])
5 /*
6  * Composite type is void WG14_DR_237(int x[static 10])
7  */
8 { /* ... */ }
```

C90

Support for the keyword **static** in this context is new in C99.

C++

Support for the keyword **static** in this context is new in C99 and is not available in C++.

Other Languages

Most strongly typed languages require an exact correspondence between the number of elements in the parameter array declaration and the number of elements in the actual argument passed in a call.

Coding Guidelines

The information provided by constant expressions appearing within the [and] of the declaration of a parameter, having an array type, can be of use to static analysis tools. However, in practice because no semantics was associated with such usage in C90, such arrays were rarely declared. It remains to be seen how the semantics given to this usage in C99 will change the frequency of occurrence of parameters having an array type (i.e., will developers use this construct to provide information to translators that might enable them to generate higher-quality code, or to source code analysis tools to enable them to issue better quality diagnostics).

Example

```

void fadd(    double a[static restrict 10],
             const double b[static restrict 10])
{
    int i;

    for (i = 0; i < 10; i++) {
        if (a[i] < 0.0)
            return;

        a[i] += b[i];
    }
}
```

Rationale

```

    return;
}

```

This function definition specifies that the parameters *a* and *b* are restricted pointers. This is information that an optimizer can use, for example, to unroll the loop and reorder the loads and stores of the elements referenced through *a* and *b*.

function type
adjust to pointer
to

A declaration of a parameter as “function returning *type*” shall be adjusted to “pointer to function returning *type*”, as in 6.3.2.1.

1600

Commentary

This is a requirement on the implementation. Occurrences of an object, not declared as a parameter, having a function type are implicitly converted to a pointer type in most contexts.

Other Languages

Languages that support parameters having some form of function type usually have their own special rules for handling them. A few languages treat function types as first class citizens and they are treated the same as any other type.

Coding Guidelines

While developers might not be aware of this implicit conversion, their interpretation of the behavior for uses of the parameter is likely to match what actually occurs (experience suggests that the lack of detailed knowledge of behavior is not replaced by some misconception that alters developer expectations of behavior).

ellipsis
supplies no in-
formation

If the list terminates with an ellipsis (, ...), no information about the number or types of the parameters after the comma is supplied.¹²³⁾

1601

Commentary

That is, no information is supplied by the developer to the translator. The ellipsis notation is intended for use in passing a variable number of argument values (at given positions in the list of arguments) having different types. The origin of support for variable numbers of arguments was the desire to treat functions handling input/output in the same as any other function (i.e., the handling of I/O functions did not depend on special handling by a translator, such as what is needed in Pascal for the `read` and `write` functions).

Prior to the publication of the C Standard there existed a programming technique that relied on making use of information on an implementation’s argument passing conventions (invariable on a stack that either grew up or down from the storage location occupied by the last named parameter). Recognizing that developers sometimes need to define functions that were passed variable numbers of arguments the C Committee introduced the ellipsis notation, in function prototypes. The presence of an ellipsis gives notice to a translator that different argument types may be passed in calls to that function. Access to any of these arguments is obtained by encoding information on the expected ordering and type, via calls to library macros, within the body of the function.

C++

The C++ Standard does not make this observation.

Other Languages

The need to pass variable number of arguments, having different types, is a common requirement for languages that specify functions to handle I/O (some languages, e.g., Fortran, handle I/O as part of the language syntax). Many languages make special cases for some I/O functions, those that are commonly required to input or output a number of different values of different types. Having to call a function for each

value and the appropriate function used depending on the type of the value is generally thought onerous by language designers.

Coding Guidelines

Many coding guideline documents recommend against the use of ellipsis. The view being taken that use of this notation represents an open-ended charter for uncontrolled argument passing. What are the alternative and how would developers handle the lack of an ellipsis notation? The following are two possibilities:

- *Use file scope objects.* Any number of file scope objects having any available type could be declared to be visible to a function definition and the contexts in which it is called.
- *Use of unions and dummy parameters.* In practice, most functions are passed a small number of optional arguments. A function could be defined to take the maximum number of arguments. In those cases where a call did not need to pass values to all the arguments available to it, a dummy argument could be passed. The number of different argument types is also, usually, small. A union type could be used to represent them.

In both cases the body of the function needs some method of knowing which values to access (as it does when the ellipsis notation is used).

Is the cure worse than the problem? The ellipsis notation has the advantage of not generating new interface issues, which the use of file scope objects is likely to do. The advantage to declaring functions to take the maximum number of arguments (use of union types provides the subset of possible types that argument values may have) is that information about all possible arguments is known to readers of the function definition. The benefit of the availability of this information is hard to quantify. However, the cost (developer effort required to analyze the arguments in the call, working out which ones are dummy and which unions members are assigned to) is likely to be significant.

Recommending that developers not use the ellipsis notation may solve one perceived problem, but because of the cost of the alternatives does not appear to result in any overall benefit.

While there is existing code that does not use the macros in the `<stdarg.h>` header to access arguments, but makes use of information on stack layout to access arguments, such usage is rarely seen in newly written code. A guideline recommendation dealing with this issue is not considered worthwhile.

1602 The special case of an unnamed parameter of type `void` as the only item in the list specifies that the function has no parameters.

parameter
type void

Commentary

The base document had no special syntax for specifying a function declaration that took no parameters. The convention used to specify this case was an empty parameter list. However, an empty parameter list was also used to indicate another convention (which is codified in the standard), a function declaration that provided no information about the arguments it might take (although it might take one or more). Use of the keyword `void` provides a means of explicitly calling out the case of a function taking no parameters (had C90 specified that an empty parameter list denoted a function taking no parameters, almost every existing C program would have become non standards conforming).

¹⁶⁰⁸ function
declarator
empty list

C90

The C90 Standard was reworded to clarify the intent by the response to DR #157.

Other Languages

Strongly typed languages treat a function declared with no parameters as a function that does not take any arguments, and they sometimes (e.g., Pascal) require the empty parentheses to be omitted from the function call. Other languages vary in their handling of this case.

Example

```

1  typedef void Void;
2
3  extern int f(Void);
4
5  int f(Void)
6  { /* ... */ }
```

parameter
declaration
typedef name
in parentheses

If, in a parameter declaration, a single typedef name in parentheses is taken to be an abstract declarator that specifies a function with a single parameter, not as redundant parentheses around the identifier for a declarator, an identifier can be treated as a typedef name or as a parameter name, it shall be taken as a typedef name.

1603

Commentary

This is a requirement on the implementation. It is an edge case of the language definition. In:

```

1  typedef int T;
2  int f(int *(T));
```

`f` is declared as function returning `int`, taking a single parameter of type pointer to function returning `int` and taking a parameter of type `T`. Without the above rule it could also be interpreted as function returning `int`, taking a single parameter of type pointer to `int`, with redundant parentheses around the identifier `T`.

Wording changes to C90, made by the response to DR #009, were not made to the text of C99 (because the committee thought that forbidding implicit types would eliminate this problem, but an edge case still remained). The response to DR #249 agreed that these changes should have been made and have now been made.

In the following declaration of WG14_N852 the identifier `what` is treated as a typedef name and violates the constraint that a function not return a function type.

function¹⁵⁹²
declarator
return type

```

1  typedef int what;
2
3  int WG14_N852(int (what)(int)); /* Constraint violation. */
```

The case of empty parentheses is discussed elsewhere.

C90

The response to DR #009 proposed adding the requirement: “If, in a parameter declaration, an identifier can be treated as a typedef name or as a parameter name, it shall be taken as a typedef name.”

Other Languages

Most languages do not allow the identifier being declared to be surrounded by parentheses. Neither do they have the C style of declarators.

Coding Guidelines

Developers are unlikely to be familiar with this edge case of the language. However, the types involved (actual and incorrectly deduced) will be sufficiently different that a diagnostic is very likely to be produced for uses of an argument or parameter based on the incorrect type.

If the function declarator is not part of a definition of that function, parameters may have incomplete type and may use the `[*]` notation in their sequences of declarator specifiers to specify variable length array types.

1604

type name
empty parentheses

Commentary

Functions declared with a parameter having an incomplete structure or union type will only be callable after the type is completed. For instance:

```

1 void f_1(struct T); /* No prior declaration of T visible here, a new type that can never be completed. */
2 struct T;
3 void f_2(struct T); /* Prior declaration of T visible here, refers to existing type. */
4 /*
5  * Cannot define an argument to have type struct T at
6  * this point so the function is not yet callable.
7  */
8 struct T {
9     int mem;
10 };
11 /*
12  * Can now define an object to pass as an argument to f_2.
13  */

```

The following declarations of `f` are all compatible with each other (the operand of `sizeof` does not need to be evaluated in this context):

```

1 int f(int n, int a[*]);
2 int f(int n, int a[sizeof(int [][*])]);
3 int f(int n, int a[sizeof(int [n][n+1])]);

```

Use of incomplete types where the size is not needed is discussed elsewhere.

footnote
109

C90

Support for the `[*]` notation is new in C99.

C++

The wording:

If the type of a parameter includes a type of the form “pointer to array of unknown bound of T” or “reference to array of unknown bound of T,” the program is ill-formed.⁸⁷⁾

8.3.5p6

does not contain an exception for the case of a function declaration.

Support for the `[*]` notation is new in C99 and is not specified in the C++ Standard.

Other Languages

Most languages require that structure types appearing in function declarations be complete. A number of languages provide some form of `[*]` notation to indicate parameters having a variable length array type.

Coding Guidelines

Functions declared with a parameter having an incomplete structure or union type might be regarded as redundant declarations. This issue is discussed elsewhere.

redundant
code

Example

```

1 void f_1(struct T_1 *p_1);
2
3 struct T_2;
4 void f_2(struct T_2 *p1);
5
6 void f_3(void)
7 {
8     struct T_1 *loc_1 = 0;
9     struct T_2 *loc_2 = 0;

```

```

10
11  f_1(loc_1); /* Constraint violation, different structure type. */
12  f_2(loc_2); /* Argument has same structure type as parameter. */
13  }

```

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. 1605

Commentary

parameter
storage-class ¹⁵⁹³

The only storage-class specifier that can occur on a parameter declaration is **register**. The interpretation of objects having this storage-class only applies to accesses to them, which can only occur in the body of the function definition.

Coding Guidelines

Whether or not function declarations are token identical to their definitions is not considered worthwhile addressing in a guideline recommendation.

An identifier list declares only the identifiers of the parameters of the function. 1606

Commentary

In a function declaration such a list provides no information to the translator, but it may provide useful commentary for readers of the source (prior to the availability of function prototypes). In a function definition this identifier list provides information to a translator on the number and names of the parameters.

C++

This form of function declarator is not available in C++.

An empty list in a function declarator that is part of a definition of that function specifies that the function has no parameters. 1607

Commentary

For a definition of a function, for instance `f`, there is no difference between the forms `f()` and `f(void)`. There is a difference for declarations, which is covered in the following C sentence.

Other Languages

An empty list is the notation commonly used in other languages to specify that a function has no parameters. Some languages also require that the parentheses be omitted.

Coding Guidelines

Differences in the costs and benefits of using either an empty list or requiring the use of the keyword **void** are not sufficient to warrant a guideline recommendation dealing with this issue.

The empty list in a function declarator that is not part of a definition of that function specifies that no information about the number or types of the parameters is supplied.¹²⁴⁾ 1608

Commentary

This specification differs from that of a function declarator that whose parameter list contains the keyword **void**.

parameter ¹⁶⁰²
type void

C++

The following applies to both declarations and definitions of functions:

If the parameter-declaration-clause is empty, the function takes no arguments.

A call made within the scope of a function declaration that specifies an empty parameter list, that contains arguments will cause a C++ translator to issue a diagnostic.

Common Implementations

Some translators remember the types of the arguments used in calls to functions declared using an empty list. Inconsistencies between the argument types in different calls being flagged as possibly coding defects.

Coding Guidelines

The guideline recommendation specifying the use of function prototypes is discussed elsewhere.

?? function
declaration
use prototype

1609 123) The macros defined in the `<stdarg.h>` header (7.15) may be used to access arguments that correspond to the ellipsis.

footnote
123

Commentary

These are the `va_*` macros specified in the library section.

1610 124) See “future language directions” (6.11.6).

footnote
124

1611 For two function types to be compatible, both shall specify compatible return types.¹²⁵⁾

function
compatible types

Commentary

This is a necessary conditions for two function declarations to be compatible. The second condition is specified in the following C sentence.

compati-
ble type
if

C++

The C++ Standard does not define the concept of compatible type, it requires types to be the same.

compati-
ble type
if

After all adjustments of types (during which typedefs (7.1.3) are replaced by their definitions), the types specified by all declarations referring to a given object or function shall be identical, . . .

3.5p10

All declarations for a function with a given parameter list shall agree exactly both in the type of the value returned and in the number and type of parameters; the presence or absence of the ellipsis is considered part of the function type.

8.3.5p3

If one return type is an enumerated type and the another return type is the compatible integer type. C would consider the functions compatible. C++ would not consider the types as agreeing exactly.

Other Languages

Compatibility of function types only becomes an issue when a language’s separate translation model allows more than one declaration of a function, or when pointers to functions are supported. In these cases the requirements specified are usually along the lines of those used by C.

Coding Guidelines

The rationale for the guideline recommendations on having a single textual declaration and including the header containing it in the source file that defines the function is to enable translators to check that the two declarations are compatible.

?? identifier
declared in one file
?? identifier
definition
shall #include

1612 Moreover, the parameter type lists, if both are present, shall agree in the number of parameters and in use of the ellipsis terminator;

Commentary

This condition applies if both function declarations use prototypes.

C++

A parameter type list is always present in C++, although it may be empty.

Other Languages

Most other languages require that the parameter type lists agree in the number of parameters.

Coding Guidelines

If the guideline recommendation specifying the use of prototypes is followed the parameter type lists will always be present.

function ??
declaration
use prototype

corresponding parameters shall have compatible types.

1613

Commentary

This requirement applies between two function declarations, not between the declaration of a function and a call to it.

C++

function call
arguments agree
with parameters

8.3.5p3 *All declarations for a function with a given parameter list shall agree exactly both in the type of the value returned and in the number and type of parameters; the presence or absence of the ellipsis is considered part of the function type.*

compati-
ble type
if

The C++ Standard does not define the concept of compatible type, it requires types to be the same. If one parameter type is an enumerated type and the corresponding parameter type is the corresponding compatible integer type. C would consider the functions to be compatible, but C++ would not consider the types as being the same.

Other Languages

Most languages require the parameter types to be compatible.

If one type has a parameter type list and the other type is specified by a function declarator that is not part of a function definition and that contains an empty identifier list, the parameter list shall not have an ellipsis terminator and the type of each parameter shall be compatible with the type that results from the application of the default argument promotions.

1614

Commentary

This C sentence deals with the case of a function prototype (which may or may not be a definition) and an old style function declaration that is not a definition. One possible situations where it can occur is where a function definition has been rewritten using a prototype, but there are still calls made to it from source where an old style declaration is visible. Function prototypes were introduced in C90 (based on the C++ specification). The committee wanted to ensure that developers could gradually introduce prototypes in to existing code. For instance, using prototypes for newly written functions. It was therefore necessary to deal with the case of source code containing so called *old style* and function prototype declarations for the same function.

Calls where the visible function declaration uses an old style declaration, have their arguments promoted using the default argument promotions. The types of the promoted arguments are required to be compatible with the parameter types in the function definition (which uses a prototype). This requirement on the parameter types in the function definition ensures that argument/parameter storage layout calculations (made by an implementation) are consistent.

default ar-
gument
promotions

The ellipsis terminator is a special case. Some translators are known to handle arguments passed to this parameter differently than when there is a declared type (the unknown nature of the arguments sometimes means that this special processing is a necessity). Given the possibility of this implementation technique the Committee decided not to require the behavior to be defined if the two kinds of declarations were used.

There can be no check on the number of parameters in the two declarations, since one of them does not have any. This issue is covered elsewhere.

arguments
same number as
parameters

C++

The C++ Standard does not support the C identifier list form of parameters. An empty parameter list is interpreted differently:

If the parameter-declaration-clause is empty, the function takes no arguments.

8.3.5p2

The two function declarations do not then agree:

After all adjustments of types (during which typedefs (7.1.3) are replaced by their definitions), the types specified by all declarations referring to a given object or function shall be identical, . . .

3.5p10

A C++ translator is likely to issue a diagnostic if two declarations of the same function do not agree (the object code file is likely to contain function signatures, which are based on the number and type of the parameters in the declarations).

Other Languages

Very few languages (Perl does) have to deal with the issue of developers being able to declare functions using two different sets of syntax rules.

Common Implementations

Implementations are not required to, and very few do, issue diagnostics if these requirements are not met. Whether programs fail to work as expected, if these requirements are not met, often depends on the characteristics of the processor. For those processors that have strict alignment requirements translators usually assign parameters at least the same alignment as those of the type `int` (ensuring that integer types with less rank are aligned on the storage boundaries of their promoted type). For processors that have more relaxed alignment requirements, or where optimisations are possible for the smaller integer types, parameters having a type whose rank less than that of the type `int` are sometime assigned a different storage location than if they had a type of greater rank. In this case the arguments, which will have been treated as having at least type `int`, will be at different storage locations in the function definitions stack frame.

alignment

Coding Guidelines

This case shows that gradually introducing function prototypes into existing source code can cause behavioral differences that did not previously exist. Most of the benefits of function prototype usage come from the checks that translators perform at the point of call. Defining a function using prototype notation and having an old style function declaration visible in a header offers little benefit (unless the function has internal linkage and is visible at all the places it is called). If an existing old style function definition is modified to use a function prototype in its definition, then it is effectively new code and any applicable guideline recommendations apply.

1615 If one type has a parameter type list and the other type is specified by a function definition that contains a (possibly empty) identifier list, both shall agree in the number of parameters, and the type of each prototype parameter shall be compatible with the type that results from the application of the default argument promotions to the type of the corresponding identifier.

Commentary

In this C sentence the two types are a function declaration that uses a prototype and a function definition that uses an old style declaration. In both cases the developer has specified the number and type of two sets of parameters. Both declarations are required to agree in the number of parameters (if the number and type of each parameter agrees there cannot be an ellipsis terminator in the function prototype).

default argument promotions

A function defined using an identifier list will be translated on the basis that the arguments, in calls to it, have been promoted according to the default argument promotions. Calls to functions where the visible declaration is a function prototype will be translated on the basis that the definition expects the arguments to be converted at the point of call (to the type of the corresponding parameter). This C sentence describes those cases where the two different ways of handling arguments results in the same behavior. In all other cases the behavior is undefined.

C++

The C++ Standard does not support the C identifier list form of parameters.

8.3.5p2 *If the parameter-declaration-clause is empty, the function takes no arguments.*

The C++ Standard requires that a function declaration always be visible at the point of call (5.2.2p2). Issues involving argument promotion do not occur (at least for constructs supported in C).

```

1 void f(int, char);
2 void f(char, int);
3 char a, b;
4 f(a,b);           // illegal: Which function is called? Both fit
5                  // equally well (equally badly).
```

Common Implementations

Implementations are not required to, and very few do, issue diagnostics if these requirements are not met.

Coding Guidelines

Adding prototype declarations to an existing program may help to detect calls made using arguments that are not compatible with the corresponding functions parameters, but they can also change the behavior of correct calls if they are not properly declared. Adhering to the guideline recommendation specifying that the header containing the function prototype declaration be **#included** in the source file that defines the function is not guaranteed to cause a translator to issue a diagnostic if the above C sentence requirements are not met. The following guideline recommendation addresses this case.

identifier ??
definition
shall #include

Cg 1615.1

If a program contains a function that is declared using both a prototype and an old style declaration, then the type of each parameter in the prototype shall be compatible with the type of corresponding parameter in the old style declaration after the application of the default argument promotions to those parameter types.

Example

```

_____ file_1.c _____
1 int f(p_1)
2 signed char p_1; /* Expecting argument to have been promoted to int. */
3 {
4 return p_1+1;
5 }
```

```

1 int f(signed char p_1);
2
3 int g(void)
4 {
5 return f('0'); /* Argument converted to type signed char. */
6 }

```

1616 (In the determination of type compatibility and of a composite type, each parameter declared with function or array type is taken as having the adjusted type and each parameter declared with qualified type is taken as having the unqualified version of its declared type.)

parameter
qualifier in
composite type

Commentary

This specifies the relative ordering of requirements on adjusting types, creating composite types and determining type compatibility. While the composite type of a parameter is always its unqualified type, the wording of the response to DR #040 question 1 explains how composite types are to be treated.

parameter
type
adjusted
function
composite type
compatible
type
if
composite
type

The type of a parameter is independent of the composite type of the function, . . .

In the body of a function the type of a parameter is the type that appears in the function definition, not any composite type. In the following example DR_040_a and DR_040_b have the same composite types, but the parameter types are not the same in the bodies of their respective function definitions.

DR #040 question 1

```

1 void DR_040_a(const int c_p);
2 void DR_040_a(    int  p)
3 {
4 p=1;                /* Not a constraint violation. */
5 }
6
7 void DR_040_b(    int  p);
8 void DR_040_b(const int c_p)
9 {
10 c_p=1;             /* A constraint violation. */
11 }

```

Calls to functions will make use of information contained in the composite type. The fact that any parameter types, in the composite type, will be unqualified is not significant because it is the unqualified parameter type that is used when processing the corresponding arguments.

prior declaration
visible

argument
type may be
assigned

C90

The C90 wording:

(For each parameter declared with function or array type, its type for these comparisons is the one that results from conversion to a pointer type, as in 6.7.1. For each parameter declared with qualified type, its type for these comparisons is the unqualified version of its declared type.)

was changed by the response to DR #013 question 1 (also see DR #017q15 and DR #040q1).

C++

The C++ Standard does not define the term *composite type*. Neither does it define the concept of compatible type, it requires types to be the same.

composite
type
compatible
type
if

The C++ Standard transforms the parameters' types and then:

If a storage-class-specifier modifies a parameter type, the specifier is deleted. [Example: register char becomes char* —end example] Such storage-class-specifiers affect only the definition of the parameter within the body of the function; they do not affect the function type. The resulting list of transformed parameter types is the function's parameter type list.*

It is this parameter type list that is used to check whether two declarations are the same.

Coding Guidelines

Adhering to the guideline recommendation specifying the use of function prototypes does not guarantee that the composite type will always contain the same parameter type information as in the original declarations.

It is possible that while reading the source of a function definition a developer will make use of information, that exists in their memory, that is based on the functions declaration in a header, rather than the declaration at the start of the definition. The consequences of this usage, in those cases where the parameter types differ in qualification, do not appear to be sufficiently costly (in unintended behavior occurring) to warrant a guideline recommendation.

Example

```
1 void f_1(int p_a[3]);
2 void f_1(int *p_a );    /* Compatible with previous f_1 */
```

function ??
declaration
use prototype

EXAMPLE 1 The declaration

```
int f(void), *fip(), (*pfi)();
```

declares a function **f** with no parameters returning an **int**, a function **fip** with no parameter specification returning a pointer to an **int**, and a pointer **pfi** to a function with no parameter specification returning an **int**. It is especially useful to compare the last two. The binding of ***fip()** is ***(fip())**, so that the declaration suggests, and the same construction in an expression requires, the calling of a function **fip**, and then using indirection through the pointer result to yield an **int**. In the declarator **(*pfi)()**, the extra parentheses are necessary to indicate that indirection through a pointer to a function yields a function designator, which is then used to call the function; it returns an **int**.

If the declaration occurs outside of any function, the identifiers have file scope and external linkage. If the declaration occurs inside a function, the identifiers of the functions **f** and **fip** have block scope and either internal or external linkage (depending on what file scope declarations for these identifiers are visible), and the identifier of the pointer **pfi** has block scope and no linkage.

Commentary

The algorithm for reading declarations involving both function and pointer types follows a right then left rule similar to that used for reading declarations involving array and pointer types. However, it is not possible to declare a function of functions, so only one function type on the right is *consumed*.

C++

Function declared with an empty parameter type list are considered to take no arguments in C++.

EXAMPLE 2 The declaration

```
int (*apfi[3])(int *x, int *y);
```

declares an array **apfi** of three pointers to functions returning **int**. Each of these functions has two parameters that are pointers to **int**. The identifiers **x** and **y** are declared for descriptive purposes only and go out of scope at the end of the declaration of **apfi**.

1617

1618

EXAMPLE
array of pointers

Commentary

The parentheses are necessary because `int *apfi[3](int *x, int *y);` declares `apfi` to be an array of three functions returning pointer to **int** (which is a constraint violation).

array element
not function type

1619 EXAMPLE 3 The declaration

```
int (*fpfi(int (*)(long), int))(int, ...);
```

declares a function `fpfi` that returns a pointer to a function returning an **int**. The function `fpfi` has two parameters: a pointer to a function returning an **int** (with one parameter of type **long int**), and an **int**. The pointer returned by `fpfi` points to a function that has one **int** parameter and accepts zero or more additional arguments of any type.

Commentary

The declaration

```
int (* (*fpfppi(int (*)(long), int))(int, ...))(void);
```

declares a function `fpfppi` that returns a pointer to a function returning a pointer to a function returning an **int** involves parenthesizing part of the existing declaration and adding information on the parameters (in this case `(void)`).

1620 125) If both function types are “old style”, parameter types are not compared.

footnote
125

Commentary

For *old style* function types there is no parameter information to compare (technically there is information available in one case, when one is a definition; however, the standard considers this information to be local to the function body).

C++

The C++ Standard does not support *old style* function types.

Other Languages

Some languages do not specify whether declarations of the same function should be compared for compatibility.

1621 EXAMPLE 4 The following prototype has a variably modified parameter.

```
void addscalar(int n, int m,
              double a[n][n*m+300], double x);

int main()
{
    double b[4][308];
    addscalar(4, 2, b, 2.17);
    return 0;
}

void addscalar(int n, int m,
              double a[n][n*m+300], double x)
{
    for (int i = 0; i < n; i++)
        for (int j = 0, k = n*m+300; j < k; j++)
            // a is a pointer to a VLA with n*m+300 elements
            a[i][j] += x;
}
```

C90

Support for variably modified types is new in C99.

C++

Support for variably modified types is new in C99 and is not specified in the C++ Standard.

Coding Guidelines

The expression `n*m+300` occurs in a number of places in the source. Replacing this expression with a symbolic name will reduce the probability of future changes to one use of this expression not being reflected in other uses.

EXAMPLE 5 The following are all compatible function prototype declarators.

1622

```
double maximum(int n, int m, double a[n][m]);
double maximum(int n, int m, double a[*][*]);
double maximum(int n, int m, double a[ ][*]);
double maximum(int n, int m, double a[ ][m]);
```

as are:

```
void f(double (* restrict a)[5]);
void f(double a[restrict][5]);
void f(double a[restrict 3][5]);
void f(double a[restrict static 3][5]);
```

(Note that the last declaration also specifies that the argument corresponding to `a` in any call to `f` must be a non-null pointer to the first of at least three arrays of 5 doubles, which the others do not.)

Commentary

The conversion of parameters having array type to pointer type allows the **restrict** type qualifier to occur in this context.

Forward references: function definitions (6.9.1), type names (6.7.6).

1623

EXAMPLE
compatible func-
tion prototypes

array
converted
to pointer

References