

The New C Standard (Excerpted material)

An Economic and Cultural Commentary

Derek M. Jones

derek@knosof.co.uk

6.7.5 Declarators

declarator
syntax

```

declarator:
    pointeropt direct-declarator
direct-declarator:
    identifier
    ( declarator )
    direct-declarator [ type-qualifier-listopt assignment-expressionopt ]
    direct-declarator [ static type-qualifier-listopt assignment-expression ]
    direct-declarator [ type-qualifier-list static assignment-expression ]
    direct-declarator [ type-qualifier-listopt * ]
    direct-declarator ( parameter-type-list )
    direct-declarator ( identifier-listopt )
pointer:
    * type-qualifier-listopt
    * type-qualifier-listopt pointer
type-qualifier-list:
    type-qualifier
    type-qualifier-list type-qualifier
parameter-type-list:
    parameter-list
    parameter-list , ...
parameter-list:
    parameter-declaration
    parameter-list , parameter-declaration
parameter-declaration:
    declaration-specifiers declarator
    declaration-specifiers abstract-declaratoropt
identifier-list:
    identifier
    identifier-list , identifier

```

Commentary

Parentheses can be used to change the way tokens in declarators are grouped in a similar way to the grouping of operands in an expression.

The declaration of function parameters does not have the flexibility available to declarations that are not parameters. For instance, it is not possible to write `void f(double a, b, c)` instead of `void f(double a, int b, int c)`.

C90

Support for the syntax:

```

direct-declarator [ type-qualifier-listopt assignment-expressionopt ]
direct-declarator [ static type-qualifier-listopt assignment-expression ]
direct-declarator [ type-qualifier-list static assignment-expression ]
direct-declarator [ type-qualifier-listopt * ]

```

is new in C99. Also the C90 Standard only supported the form:

```

direct-declarator [ constant-expressionopt ]

```

parameter
declaration
typedef name
in parentheses

C++

The syntax:

```

direct-declarator [ type-qualifier-listopt assignment-expressionopt ]
direct-declarator [ static type-qualifier-listopt assignment-expression ]
direct-declarator [ type-qualifier-list static assignment-expression ]
direct-declarator [ type-qualifier-listopt * ]
direct-declarator ( identifier-listopt )

```

is not supported in C++ (although the form *direct-declarator* [*constant-expression*_{opt}] is supported).

The C++ Standard also supports (among other constructions) the form:

8p4

```

direct-declarator      (      parameter-declaration-clause ) cv-qualifier-seqopt
exception-specificationopt

```

The C++ Standard also permits the comma before an ellipsis to be omitted, e.g., `int f(int a ...);`.

Other Languages

The extent to which the identifier being declared is visually separate from its associated type information varies between languages. At one extreme languages in the Pascal family completely separate the two (both Pascal and Ada require that a colon, `:`, appear between an identifier and its type information). C (and C++) are at the other extreme, integrating the identifier being declared into the syntax of the type declaration. Fortran might be considered intermediate, with the identifier being syntactically integrated with the type in some cases (e.g., array declarations `REAL A(10)`).

Common Implementations

The type qualifiers **near**, **far**, **pascal**, **tiny**, and **huge** (sometimes prefixed with one or more underscores) are ubiquitous extensions for implementations targeting the Intel x86 processor family (these qualifiers are also found in other implementations where the target processor supports a variety of pointer sizes). gcc supports a variety of kinds of declaration containing the keyword `__attribute__` that specify a variety of different kinds of semantic information about the identifier being declared. The HP C/iX translator^[1] supports a short pointer (32-bit) and long pointer (64-bit). The declaration of a type denoting a long pointer uses the punctuator `^`. For instance, `int ^long_ptr`.

Coding Guidelines

A very common mistake made by beginners is to treat the following two declarations of `arr` as being compatible:

```
1 int arr[];
```

and

```
1 int *arr;
```

being unaware that the duality between arrays and pointers only holds in expressions. However, this is a developer education rather than a coding guideline issue.

A more subtle mistake that even experienced developers make is to treat the star, `*`, token as belonging to the type information. For instance, in:

```

1 char * pc_1,
2   pc_2; /* Has type char, not char * */
3 char (* pc_3),
4   pc_4; /* Same type as pc_2.      */

```

EXAMPLE
array not pointer

it is only `p_1` and `p_3` that have pointer types. One possible reason for the categorizations mistake is the proximity of the type specifier and the `*` token. Developers do not seem to make the same mistake with array bounds declarators, where an identifier appears between the type and the `[]` tokens. The following guideline recommendation is a simple method of avoiding this kind of mistake.

Cg 1547.1

The number of star, `*`, tokens appearing on each declarator of the same declarator list of a declaration shall be the same.

Redundant parentheses do not commonly appear in declarators and this issue is not discussed further.

Example

```

1  int (x); /* Redundant () */
2
3  int (((aa[1])[1])[1]); /* Harder to follow than without ()? */
4  int ab[1][1][1];
5
6  int (*ap1)[10]; /* Pointer to array of 10 ints. */
7  int *(ap2[10]); /* Array of pointers to int. */
8  int *ap3[10]; /* Which is this? */
9
10 typedef int * p_i;
11
12 p_i * p_1,
13      p_2;
```

Semantics

Each declarator declares one identifier, and asserts that when an operand of the same form as the declarator appears in an expression, it designates a function or object with the scope, storage duration, and type indicated by the declaration specifiers. 1548

Commentary

The form of an identifier in an expression is likely to be the same as that in the declarator. For instance, the declarator `* x` will have this form in an expression when the value pointed to by `x` is required and the declarator `y[2]` will have this form in an expression when an element of the array `y` is referred to. It is the declarator portion of a declaration that declares the identifier. There is a special kind of declarator, an *abstract-declarator*, which does not declare an identifier.

abstract
declarator
syntax

full declarator

A *full declarator* is a declarator that is not part of another declarator. 1549

Commentary

This defines the term *full declarator* (which mimics that for full expression). A declarator appearing as an operand of a cast operator, that is not part of an initializer, is a full declarator. The declarators for the parameters in a function declaration are part of another declarator.

full ex-
pression

C90

Although this term was used in the C90 Standard, in translation limits, it was not explicitly defined.

C++

This term, or an equivalent one, is not defined by the C++ Standard.

full declarator
sequence point

The end of a full declarator is a sequence point. 1550

Commentary

It is possible for a nonconstant expression to occur within a declarator (for instance, the expression denoting the number of array elements) and its may cause side effects. In:

side effect

```

1  extern int glob;
2
3  void f(char a_1[glob++], char a_2[glob++])
4  { /* ... */ }
```

`glob` is modified twice between two adjacent sequence points and the behavior is undefined. While in:

object
modified once
between sequence
points

```

1  int g(void)
2  {
3  static int val;
4  return ++val;
5  }
6
7  void f(char a_1[g()], char a_2[g()])
8  { /* ... */ }
```

the behavior is unspecified.

C90

The ability to use an expression causing side effects in an array declarator is new in C99. Without this construct there is no need to specify a sequence point at the end of a full declarator.

C++

The C++ Standard does not specify that the end of an declarator is a sequence point. This does not appear to result in any difference of behavior.

Other Languages

Those languages supporting some form of execution time array bounds selection invariably have the same behavior (i.e., an evaluation order is not defined).

Coding Guidelines

This ability for declarators to cause side effects is new in C99, although declarations could cause side effects in C90 through the use of an initializer. This issue of side effects is discussed elsewhere

object
modified once
between sequence
points

Example

```

1  extern int glob;
2
3  void f(void)
4  {
5  int arr_1[glob++],
6      arr_2[glob++];
7  }
```

1551 If in the nested sequence of declarators in a full declarator contains there is a declarator specifying a variable length array type, the type specified by the full declarator is said to be *variably modified*.

Commentary

This defines the term *variably modified*. The parameter types or return type of a function type are not usually considered to be *nested* within its full declarator.

The wording was changed by the response to DR #311.

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.

Furthermore, any type derived by declarator type derivation from a variably modified type is itself variably modified. 1552

Commentary

This wording ensures that in code such as the following example, the declaration of *y* is also variably modified.

```

1  int x;
2
3  typedef int vla[x];
4  vla y[3];

```

This sentence was added by the response to DR #311.

In the following subclauses, consider a declaration 1553

T D1

where *T* contains the declaration specifiers that specify a type *T* (such as `int`) and **D1** is a declarator that contains an identifier *ident*.

Commentary

This is the simplest, most basic, form of declaration (the various possible declarators, **D1**, supported in C, are described in the following C sentences).

The type specified for the identifier *ident* in the various forms of declarator is described inductively using this notation. 1554

Commentary

In the case of abstract declarators there is no identifier (although discussions involving these constructs sometimes refer to an *omitted* identifier).

If, in the declaration “**T D1**”, **D1** has the form 1555

identifier

then the type specified for *ident* is *T*.

Commentary

This is the most common form of declaration, declaring an identifier to have type *T*.

If, in the declaration “**T D1**”, **D1** has the form 1556

(*D*)

then *ident* has the type specified by the declaration “**T D**”.

Commentary

That is, the use of parentheses is purely a syntactic device that may affect the parsing of a sequence of tokens.

1557 Thus, a declarator in parentheses is identical to the unparenthesized declarator, but the binding of complicated declarators may be altered by parentheses.

Commentary

This specifies that this parenthesis around declarators have no semantics associated with them (they have associated semantics when they appear to the right of an identifier, they cause it to denote a function type). It is also possible to use typedefs to achieve the same effect. For instance:

```
1 int (*V_1)[10]; /* Pointer to array of int. */
2
3 typedef int A10_INT[10];
4 A10_INT *V_2; /* Pointer to array of int. */
```

Other Languages

Many languages do not support the bracketing of identifiers in declarations. It is only needed in C because the syntax of declarators uses tokens that can appear to the left or the right of the identifier whose type they specify. To specify some types parentheses are needed to alter the binding. For instance,

```
1 int *V[10]; /* Array of pointer to int. */
2 int (*V)[10]; /* Pointer to array of int. */
3
4 /*
5  * Type information is Pascal always reads left-to-right.
6  */
7 V : array[0..9] of ^ integer; /* Array of pointer to int. */
8 V : ^array[0..9] of integer; /* Pointer to array of int. */
```

Common Implementations

Parentheses are processed as part of the syntax. Their presence can alter the parsing (binding) of declarators. There are no implementation issues associated by reordering declarators through parenthesis, like there are for expressions.

parenthesized expression

Coding Guidelines

Experience shows that declarators of the form `*x[3]` are a source of developer miscomprehension. Some developers reading it left-to-right “a pointer to an array of . . .”, rather than the correct right-to-left order “array of pointer to . . .”. The interpretation of an objects type affects how it is used in expressions. Depending on how `x` is accessed the final operand type may become “a pointer to a pointer to . . .” under both interpretations of the type of `x` (indexing an object having an array type causes it to be converted to a pointer type in many contexts). A consequence of this conversion is that it is possible for `x` to be accessed, under both interpretations of its type, without a translator diagnostic (pointing out a type mismatch) being issued.

array converted to pointer

If the guideline recommendation specifying the use of parenthesis in expressions is applied to declarations, a reader of the source may notice a discrepancy between their incorrect interpretation of the type specified in an objects declaration and the type implied by how that object is used in an expression. However, using parenthesis to unambiguously define the intended syntactic binding of the components of a declarator is a more reliable method of ensuring that readers comprehend the intended type.

?? expression shall be parenthesized

Cg 1557.1

The declarator for an object declared to have type “array of pointer to . . .” shall parenthesize the array portion of the declarator.

Dev 1557.1

The array portion of a declarator need not be parenthesized if its element type is specified using a typedef name.

Example

```

1  #define INT_PTR int *
2  typedef int * int_ptr;
3  int * arr_1 [3];
4  int (*arr_2) [3]; /* Parenthesis must be used to declare this type. */
5  int *(arr_3 [3]);
6  int_ptr arr_4[3]; /* Unlikely to be interpreted as a pointer to an array. */
7  INT_PTR arr_5[3]; /* Not covered by previous deviation because the usage is recommended against elsewhere. */

```

Implementation limits

declarator
complexity lim-
its

As discussed in 5.2.4.1, an implementation may limit the number of pointer, array, and function declarators that modify an arithmetic, structure, union, or incomplete type, either directly or via one or more **typedefs**. 1558

Commentary

limit
type complexity

This wording differs from that in 5.2.4.1 in that it includes types defined via typedefs in the count.

C90

*The implementation shall allow the specification of types that have at least 12 pointer, array, and function declarators (in any valid combinations) modifying an arithmetic, a structure, a union, or an incomplete type, either directly or via one or more **typedefs**.*

Forward references: array declarators (6.7.5.2), type definitions (6.7.7).

1559

References

USA, 3 edition, Apr. 1999.

1. H. Packard. *HP C/iX Reference Manual*. Hewlett Packard, Inc,