

The New C Standard (Excerpted material)

An Economic and Cultural Commentary

Derek M. Jones

derek@knosof.co.uk

6.7.6 Type names

ab-
stract declarator
syntax

type-name:
 specifier-qualifier-list abstract-declarator_{opt}
abstract-declarator:
 pointer
 pointer_{opt} direct-abstract-declarator
direct-abstract-declarator:
 (*abstract-declarator*)
~~*direct-abstract-declarator_{opt} [assignment-expression_{opt}]*~~
~~*direct-abstract-declarator_{opt} [type-qualifier-list_{opt} assignment-expression_{opt}]*~~
~~*direct-abstract-declarator_{opt} [static type-qualifier-list_{opt} assignment-expression]*~~
~~*direct-abstract-declarator_{opt} [type-qualifier-list static assignment-expression]*~~
~~*direct-abstract-declarator_{opt} [*]*~~
direct-abstract-declarator_{opt} (parameter-type-list_{opt})

Commentary

An abstract declarator specifies a type without defining an associated identifier. The term *type-name* is slightly misleading since there is no name, the type is anonymous. The *direct-declarator* productions:

declarator
syntax

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

are not supported for abstract declarators (some committee members say this was intended, others that it was an accidental omission).

The wording was changed by the response to DR #289 and makes the syntax consistent with that for *direct-declarator*.

declarator
syntax

C90

Support for the form:

```
direct-abstract-declaratoropt [ * ]
```

is new in C99. In the form:

```
direct-abstract-declaratoropt [ assignment-expressionopt ]
```

C90 only permitted *constant-expression_{opt}* to appear between [and].

C++

The C++ Standard supports the C90 forms. It also includes the additional form (8.1p1):

```
direct-abstract-declaratoropt ( parameter-declaration-clause )
cv-qualifier-seqopt exception-specificationopt
```

Other Languages

A few languages (e.g., Algol 68) have a concept similar to that of abstract declarator (i.e., an unnamed type that can appear in certain contexts, such as casts).

Coding Guidelines

The term *type* is applied generically to all type declarations, whether they declare identifiers or not. Developers do not appear to make a distinction between declarators and abstract declarators.

More cognitive effort is needed to comprehend an abstract declarator than a declarator because of the additional task of locating the position in the token sequence where the identifier would have been, had it not been omitted. Whether there is sufficient benefit in providing an identifier (taking into account the costs of providing it in the first place) to make a guideline recommendation worthwhile is a complex question that your author does not yet feel capable of answering.

Semantics

1625 In several contexts, it is necessary to specify a type.

Commentary

These contexts are: a compound literal, the type in a cast operation, the operand of **sizeof**, and parameter types in a function prototype declaration that omits the identifiers.

1626 This is accomplished using a *type name*, which is syntactically a declaration for a function or an object of that type that omits the identifier.¹²⁶⁾

Commentary

This defines the term *type name* (and saying in words what is specified in the syntax).

C++

Restrictions on the use of type names in C++ are discussed elsewhere.

Coding Guidelines

A *type-name* is syntactically a declaration and it is possible to declare identifiers using it. For instance:

sizeof
constraints
cast
scalar or void
type
function
declarator return
type

```

1 void f_1(int p)
2 {
3   p=(enum {apple, orange, pear})sizeof(enum {brazil, cashu, almond});
4 }
5
6 struct T {int mem;} f_2(void)
7 {
8   struct T loc;
9   /* ... */
10  return loc;
11 }
```

Such uses are not common and might come as a surprise to some developers. The cost incurred by this usage is that readers of the source may have to spend additional time searching for the identifiers declared, because they do not appear in an expected location. There is no obvious worthwhile benefit (although there is a C++ compatibility benefit) in a guideline recommendation against this usage.

1627 EXAMPLE The constructions

```

(a)      int
(b)      int *
(c)      int *[3]
(d)      int (*)(3]
(e)      int (*)[*]
(f)      int *()
(g)      int *(void)
(h)      int (*const [])(unsigned int, ...)
```

EXAMPLE
abstract
declarators

name respectively the types (a) `int`, (b) pointer to `int`, (c) array of three pointers to `int`, (d) pointer to an array of three `ints`, (e) pointer to a variable length array of an unspecified number of `ints`, (f) function with no parameter specification returning a pointer to `int`, (g) pointer to function with no parameters returning an `int`, and (h) array of an unspecified number of constant pointers to functions, each with one parameter that has type `unsigned int` and an unspecified number of other parameters, returning an `int`.

Commentary

The following is one algorithm for locating where the omitted identifier occurs in an abstract declarator. Starting on the left and working right:

1. skip all keywords, identifiers, and any matched pairs of braces along with their contents (the latter are `struct/union/enum` declarations), then
2. skip all open parentheses, asterisks, and type qualifiers. The first unskipped token provides the context that enables the location of the omitted identifier to be deduced:
 - A `[` token is the start of an array specification that appears immediately after the omitted identifier.
 - A *type-specifier* is the start of the *declaration-specifier* of the first parameter of a parameter list. The omitted identifier occurs immediately before the last skipped open parenthesis.
 - A `)` token immediately after a `(` token is an empty parameter list. The omitted identifier occurs immediately before the last skipped open parenthesis.
 - A `)` token that is not immediately after a `(` token is the end of a parenthesized abstract declarator with no array or function specification. The omitted identifier occurs immediately before this `)` token.

C90

Support for variably length arrays is new in C99.

C++

Support for variably length arrays is new in C99 and is not specified in the C++ Standard.

126) As indicated by the syntax, empty parentheses in a type name are interpreted as “function with no parameter specification”, rather than redundant parentheses around the omitted identifier. 1628

Commentary

That is in the following declaration of `f`:

```

1  typedef char x;
2  void f(int (x), /* Function returning int having a single int parameter. */
3         int ()); /* Function returning int with no parameter information specified. */
4
5  void g(int (y) /* Parameter having type int and name y. */
6         );
```

The behavior of parentheses around an identifier is also described elsewhere.

Other Languages

This notation is used by many languages (many of which don't allow parentheses around identifiers).

Coding Guidelines

The guideline recommendation dealing with the use of prototypes in function declarators is applicable here.

footnote
126
type name
empty paren-
theses

parameter
declaration
typedef name
in parentheses

function ??
declaration
use prototype

References