The New C Standard (Excerpted material)

An Economic and Cultural Commentary

Derek M. Jones
derek@knosof.co.uk

6.7.7 Type definitions

typedef name syntax

1629

typedef-name:
  identifier

Commentary

name space ordinary identifiers

A typedef name exists in the same name space as ordinary identifiers. The information that differentiates an identifier as a *typedef-name*, from other kinds of identifiers is the visibility, or not, of a typedef definition of that identifier. For instance, given the declarations:

```
typedef int type_ident;
type_ident(D_1);          /* Function call or declaration of D_1?  */
type_ident * D_2;         /* Multiplication or declaration of D_2? */
```

it is not possible to decide, using syntax only, whether `type_ident(D_1);` is a function call or a declaration of `D_1` using redundant parentheses. In the declaration of `D_2` a parser does not know this is a declaration, rather than a syntax violation, until it has seen the sixth token after the `type_ident` at the start of the line (i.e., more than one token lookahead is required).

There are some contexts where the status of an identifier as a *typedef-name* can be deduced. For instance, the token sequence `; x y;` is either a declaration of `y` to have the type denoted by `x`, or it is a violation of syntax (because a definition of `x` as a typedef name is not visible).

Other Languages

Languages invariably use ordinary identifiers to indicate both objects and their equivalent (if supported) of typedef names.

Common Implementations

The syntax of most languages is such that it is possible to parse their source without the need for a symbol table holding information on prior declarations. It is also usually possible to parse them by looking ahead a single token in the input stream (tools such as `bison` support such language grammars).

The syntax of C declarations and the status of *typedef-name* as an identifier token creates a number of implementation difficulties. The parser either needs access to a symbol table (so that it knows which identifiers are defined as *typedef-name*s), or it needs to look ahead more than one token and be able to handle more than one parse of some token sequences. Most implementations use the symbol table approach (which in practice is more complicated than simply accessing a symbol table; it is also necessary to set or reset a flag based on the current syntactic context, because an identifier should only be looked up, to find out if it is currently defined as a *typedef-name*, in a subset of the contexts in which an identifier can occur).

Coding Guidelines

It is common developer practice to use the term *type name* (as well as the term *typedef name*) to refer to the identifier defined by a typedef declaration. There is no obvious benefit in attempting to change this common developer usage. The issue of naming conventions for typedef names is discussed elsewhere.

typedef naming conventions

The higher-level source code design issues associated with the use of typedef names are discussed below. Given some of the source reading techniques used by developers it is possible that a typedef name appearing in a declaration will be treated as one of the identifiers being declared. This issue is discussed elsewhere. This coding guideline subsection discusses the lower-level issues, such as processing of the visible source by readers and naming conventions.

reading kinds of
declaration syntax

The only time the visible form of declarations is likely to contain two identifiers adjacent to each other (separated only by white space) is when a typedef name is used (such adjacency can also occur through the use of macro names, but is rare). The two common cases are for the visible source lines, containing a declaration, to either start with a keyword (e.g., **`int`** or **`struct`**) or an identifier that is a member of an identifier list (e.g., a list of identifiers being declared).

Some of the issues involved in deciding whether to use a typedef name of a tag name, for structure and union types, is discussed elsewhere. Using a typedef name provides a number of possible benefits, including the following:

tag naming conventions

- Being able to change the type used in more than declaration by making a change to one declaration (the typedef name). In practice this cost saving is usually only a significant factor when the type category is not changed (e.g., an integer type is changed to an integer type, or a structure type is changed to another structure type). In this case the use of objects, declared using these typedef name, as operands in expressions does not need to be modified (changing, for instance, an array type to a structure type is likely to require changes to the use of the object in expressions).

type category

- The spelling of the type name may providing readers with semantic information about the type that would not be available if the sequence of tokens denoting the type had appeared in the source. The issue of providing semantic information via identifier spellings is discussed elsewhere.

identifier semantic associations

- The use of typedef names is sometimes recommended in coding guideline documents because it offers a mechanism for hiding type information. However, readers are likely to be able to deduce this (from looking at uses of an object), and are also likely to need to know an objects type category (which is probably the only significant information that type abstraction is intended to hide).

type category

Usage

A study by Neamtiu, Foster, and Hicks[2] of the release history of a number of large C programs, over 3-4 years (and a total of 43 updated releases), found that in 16% of releases one or more existing typedef names had the type they defined changed.[1]

Table 1629.1: Occurrences of types defined in a **`typedef`** definition (as a percentage of all types appearing in **`typedef`** definition). Based on the translated form of this book's benchmark programs.

Type	Occurrences	Type	Occurrences
struct	58.00	**unsigned long**	1.47
enum	9.50	**int** *()	1.46
other-types	8.86	**enum** *()	1.46
struct *	6.97	**union**	1.38
unsigned int	2.68	**long**	1.29
int	2.46	**void** *()	1.18
unsigned char	2.21	**unsigned short**	1.07

Constraints

1630 If a typedef name specifies a variably modified type then it shall have block scope.

Commentary

Allowing a typedef name to occur at file scope appears to be useful; the identifier name provides a method of denoting the same type in declarations in different functions, or translation units. However, the expression denoting the number of elements in the array has to be evaluated during program execution. The committee decided that this evaluation would occur when the type declaration was encountered during program execution. This decision effectively prevents any interpretation being given tor such declarations at file scope.

1632 array size evaluated when declaration reached

C90

Support for variably modified types is new in C99.

C++

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

Coding Guidelines

The extent to which more than one instance of a variably modified type using the same size expression and element type will need to be defined in different functions is not known. A macro definition provides one mechanism for ensuring the types are the same. Variably modified types are new in C99 and experience with their use is needed before any guideline recommendations can be considered.

Semantics

In a declaration whose storage-class specifier is **typedef**, each declarator defines an identifier to be a typedef name that denotes the type specified for the identifier in the way described in 6.7.5. 1631

Commentary

The association between **typedef** and storage class is a syntactic one (they share the same declarator forms), not a semantic one.

Other Languages

Many languages that support developer defined type names define a syntax that is specific to that usage (which may simply involve the use of different keyword, for instance Pascal requires that type definitions be introduced using the keyword **type**).

Coding Guidelines

declaration visual layout

The issues involved in declarations that declare more than one identifier are discussed elsewhere.

Example

```
extern  int i, j[3];
typedef int I, J[3];
```

array size evaluated when declaration reached

Any array size expressions associated with variable length array declarators are evaluated each time the declaration of the typedef name is reached in the order of execution. 1632

Commentary

Rationale

Using a **typedef** to declare a variable length array object (see §6.7.5.2) could have two possible meanings. Either the size could be eagerly computed when the **typedef** is declared, or the size could be lazily computed when the object is declared. For example

```
{
    typedef int VLA[n];
    n++;
    VLA object;

    // ...
}
```

The question arises whether `n` should be evaluated at the time the type definition itself is encountered or each time the type definition is used for some object declaration. The Committee decided that if the evaluation were to take place each time the typedef name is used, then a single type definition could yield variable length array types involving many different dimension sizes. This possibility seemed to violate the spirit of type definitions. The decision was made to force evaluation of the expression at the time the type definition itself is encountered.

In other words, a typedef declaration does not act like a macro definition (i.e., with the expression evaluated for every invocation), but like an initialization (i.e., the expression is evaluated once).

C90

Support for variable length array declarators is new in C99.

C++

Support for variable length array declarators is new in C99 and is not specified in the C++ Standard.

Other Languages

Other languages either follow the C behavior (e.g., Algol 68), or evaluated on each instance of a typedef name usage (e.g., Ada).

Coding Guidelines

While support for this construct is new in C99 and at the time of this writing insufficient experience with its use is available to know whether any guideline recommendation is worthwhile, variable length array declarations can generate side effects, a known problem area. The guideline recommendation applicable to side effects in declarations is discussed elsewhere.

?? full declarator all orderings give same type

Example

In the following the objects q and r will contain the same number of elements as the object p.

```
void f(int n)
{
START_AGAIN: ;

typedef int A[n];

A p;
n++;
A q;

   {
   int n = 99;
   A r; /* Uses the object n visible when the typedef was defined. */
   }

if ((n % 4) != 0)
   goto START_AGAIN;

if ((n % 5) != 0)
   f(n+2)
}
```

1633 A **typedef** declaration does not introduce a new type, only a synonym for the type so specified.

typedef is synonym

Commentary

Typedef names provide a mechanism for easily modifying (by changing a single definition) the types of a set of objects (those declared using a given typedef name) declared within the source code. The fact that a **typedef** only introduces a synonym for a type, not a new type, is one of the reasons C is considered to be a typed language but not a strongly typed language.

It is not possible to introduce a new name for a tag. For instance:

```
typedef oldtype newtype;                /* Supported usage.  */
typedef struct oldtype struct newtype;  /* Syntax violation. */
```

One difference between a typedef name and a tag is that the former may include type qualifier information. For instance, in:

```
typedef const struct T {
                        int mem;
                       } cT;
```

the typedef name cT is **const** qualified, while the tag T is not.

Other Languages

Some languages regard all typedef declarations as introducing new types. A few languages have different kinds of typedef declarations, one introducing a new type and the other introducing a synonym.

Common Implementations

Some static analysis tools support an option that allows typedef names to be treated as new (i.e., different) types (which can result in diagnostics being issued when mixing operands having types).

Coding Guidelines

Experienced users of strongly typed languages are aware of the advantages of having a typedef that creates a new type (rather than a synonym). They enable the translator to aid the developer (by detecting mismatches and issuing diagnostics) in ensuring that objects are not referenced in inappropriate contexts. Your author is not aware of any published studies that have investigated the costs and benefits of strong typing (there have been such studies comparing so called *strongly typed* languages against C, but it is not possible to separate out the effects of typing from other language features). Also there does not appear to have been any work that has attempted to introduce strong typing into C in a commercial environment.

Your authors experience (based on teaching Pascal and analyzing source code written in it) is that it takes time and practice for developers to learn how to use strong type names effectively. While the concepts behind individual type names may be quickly learned and applied, handling the design decisions behind the interaction between different type names requires a lot of experience. Even in languages such as Pascal and Ada, where implementations enforced strong typing, developers still required several years of experience to attain some degree of proficiency.

standard integer types

Given C's permissive type checking (e.g., translators are not required to perform much type checking on the standard integer types), only a subset of the possible type differences are required to generate a diagnostic to be generated. Given the lack of translator support for strong type checking and the amount of practice needed to become proficient in its use, there is no cost/benefit in recommending the use of typedef names for type checking purposes. The cost/benefit of using typedef names for other purposes, such as enabling the types of a set of objects to be changed by a single modification to the source, may be worthwhile.

Measurements of the translated form of this book's benchmark programs show that typedef names occur much more frequently than the tag names (by a factor of 1.7:1; although this ratio is switched in some programs, e.g., tag names outnumber typedef names by 1.5:1 in the Linux kernel). Why is this (and should the use of typedef names be recommended)? The following are some of the possible reasons:

- Developers new to C imitate what they see others doing and what they read in books.
- The presence of the **struct/union/enum** keyword is considered to be useful information, highlighting the kind of type that is being used (structure types in particular seem to be regarded as very different animals than other types). While this rationale goes against the design principle of hiding representation details, experience shows that uses of structure types are rarely changed to non-structure types.
- The usage is driven by the least effort principle being applied by a developer at the point where the structure type is defined. While the effort needed in subsequent references to the type may be less, had a typedef name had been used, the cost of subsequent uses is not included in the type definition decision process.

1634 That is, in the following declarations:

```
typedef T type_ident;
type_ident D;
```

type_ident is defined as a typedef name with the type specified by the declaration specifiers in **T** (known as *T*), and the identifier in **D** has the type "*derived-declarator-type-list T*" where the *derived-declarator-type-list* is specified by the declarators of **D**.

Commentary

Declarations of the form `type_ident D;` is the only situation where two identifier tokens are adjacent in the preprocessed source.

C++

This example and its associated definition of terms is not given in the C++ Standard.

1635 A typedef name shares the same name space as other identifiers declared in ordinary declarators.

Commentary

This issue is discussed elsewhere.

name space
ordinary identifiers

1636 EXAMPLE 1 After

```
typedef int MILES, KLICKSP();
typedef struct { double hi, lo; } range;
```

the constructions

```
MILES distance;
extern KLICKSP *metricp;
range x;
range z, *zp;
```

are all valid declarations. The type of **distance** is **int**, that of **metricp** is "pointer to function with no parameter specification returning **int**", and that of **x** and **z** is the specified structure; **zp** is a pointer to such a structure. The object **distance** has a type compatible with any other **int** object.

Coding Guidelines

The use of uppercase in typedef names is discussed elsewhere.

typedef
naming conventions

1637 EXAMPLE 2 After the declarations

```
typedef struct s1 { int x; } t1, *tp1;
typedef struct s2 { int x; } t2, *tp2;
```

type **t1** and the type pointed to by **tp1** are compatible. Type **t1** is also compatible with type **struct s1**, but not compatible with the types **struct s2**, **t2**, the type pointed to by **tp2**, or **int**.

Coding Guidelines

The issue of different structure types declaring members having the same identifier spelling is discussed elsewhere.

member
naming conventions

1638 EXAMPLE 3 The following obscure constructions

```
typedef signed int t;
typedef int plain;
struct tag {
        unsigned t:4;
        const t:5;
        plain r:5;
};
```

declare a typedef name **t** with type **signed int**, a typedef name **plain** with type **int**, and a structure with three bit-field members, one named **t** that contains values in the range [0, 15], an unnamed const-qualified bit-field which (if it could be accessed) would contain values in either the range [-15, +15] or [-16, +15], and one named **r** that contains values in one of the ranges [0, 31], [-15, +15], or [-16, +15]. (The choice of range is implementation-defined.) The first two bit-field declarations differ in that **unsigned** is a type specifier (which forces **t** to be the name of a structure member), while **const** is a type qualifier (which modifies **t** which is still visible as a typedef name). If these declarations are followed in an inner scope by

```
t f(t (t));
long t;
```

then a function **f** is declared with type "function returning **signed int** with one unnamed parameter with type pointer to function returning **signed int** with one unnamed parameter with type **signed int**", and an identifier **t** with type **long int**.

Coding Guidelines

identifier ?? reusing names

The guideline recommendation dealing with the reuse of identifiers is applicable here.

EXAMPLE 4 1639
On the other hand, typedef names can be used to improve code readability. All three of the following declarations of the **signal** function specify exactly the same type, the first without making use of any typedef names.

```
typedef void fv(int), (*pfv)(int);

void (*signal(int, void (*)(int)))(int);
fv *signal(int, fv *);
pfv signal(int, pfv);
```

Commentary

EXAMPLE abstract declarators

The algorithm for locating the omitted identifier in abstract declarators can be used to locate the identifier declared by the declarator in complex declarations.

EXAMPLE 5 If a typedef name denotes a variable length array type, the length of the array is fixed at the time the typedef name is defined, not each time it is used: 1640

```
void copyt(int n)
{
        typedef int B[n];   // B is n ints, n evaluated now
        n += 1;
        B a;                // a is n ints, n without += 1
        int b[n];           // a and b are different sizes
        for (int i = 1; i < n; i++)
                a[i-1] = b[i];
}
```

Other Languages

Some languages (e.g., Ada) support parameterized typedefs that allow information, such as array size, to be explicitly specified when the type is instantiated (i.e., when an object is declared using it).

Coding Guidelines

At the time of this writing there is insufficient experience available with how variable length array types are used to know whether a guideline recommendation dealing with modifications to the values of objects used in the declaration of typedef names, such as n in the above example, is worthwhile.

References

1. I. Neamtiu. Detailed break-down of general data provided in paper[2] kindly supplied by first author. Jan. 2008.
2. I. Neamtiu, J. S. Foster, and M. Hicks. Understanding source code evolution using abstract syntax tree matching. In *Proceedings of the 2005 International Workshop on Mining Software Repositories*, pages 1–5, May 2005.