The New C Standard (Excerpted material)

An Economic and Cultural Commentary

Derek M. Jones
derek@knosof.co.uk

6.2.3 Name spaces of identifiers

name space

If more than one declaration of a particular identifier is visible at any point in a translation unit, the syntactic context disambiguates uses that refer to different entities. 438

Commentary

At a particular point in the source it is possible to have the same identifier declared and visible as a label, object/function/typedef/enumeration constant, enum/structure/union tag, and a potentially infinite number of member names. Syntactic context in C is much more localized than in human languages. For instance, in "There is a sewer near our home who makes terrific suites," the word *sewer* is not disambiguated until at least four words after it occurs. In C the token before an identifier (`->`, `.`, **`struct`**, **`union`**, and **`enum`**) disambiguates some name spaces. Identifiers used as labels require a little more context because the following `:` token can occur in several other contexts.

C++

This C statement is not always true in C++, where the name lookup rules can involve semantics as well as syntax; for instance, in some cases the **`struct`** can be omitted.

tag 441
name space

Common Implementations

The amount of sufficient syntactic context needed to determine uses that refer to typedef names and other kinds of identifiers is greater than most implementors of translators want to use (for efficiency reasons most existing parser generators only use a lookahead of one token). Translators handle this by looking up identifiers in a symbol table prior to passing them on to be syntactically processed. For instance, the sequence `int f(a, b)` could be declaring `f` to be an old-style function definition (where `a` and `b` are the identifier names) or a function prototype (where `a` and `b` are the types of the parameters). Without accessing the symbol table, a translator would not know which of these cases applied until the token after the closing right parenthesis was seen.

Coding Guidelines

syntactic context

belief maintenance

Syntactic context is one source of information available to developers when reading source code. Another source of information are their existing beliefs (e.g., obtained from reading other source code related to what they are currently reading). Note: There is a syntactic context where knowledge of the first token does not provide any information on the identity a following identifier; a declaration defines a new identifier and the spelling of this identifier is not known until it is seen.

semantic priming

Many studies have found that people read a word (e.g., doctor) more quickly and accurately when it is preceded by a related word (e.g., nurse) than when it is preceded by an unrelated word (e.g., toast). This effect is known as *semantic priming*. These word pairs may be related because they belong to the same category, or because they are commonly associated with each other (often occurring in the same sentence together). Does the syntactic context of specific tokens (e.g., `->`) or keywords (e.g., **`struct`**) result in semantic priming? For such priming to occur there would need to be stronger associations in the developer's mind between identifiers that can occur in these contexts, and those that cannot. There have been no studies investigating how developers store and access identifiers based on their C language properties. It is not known if, for instance, an identifier appearing immediately after the keyword **`struct`** will be read more quickly and accurately if its status as a tag is known to the developer.

It is possible for identifiers denoting different entities, but with the same spelling, to appear closely together in the source. For instance, an identifier with spelling `kooncliff` might be declared as both a structure tag and an object of type **`float`** and appear in an expression in both contexts. Is syntactic context the dominant factor in such cases, or will the cost of comprehension be measurably greater than if the identifiers had different spellings?[438.1]

[438.1] No faults can be introduced into the source through misuse of these identifiers because all such uses will result in a translator issuing a diagnostic.

A study by McKoon and Ratcliff[3] looked at contextual aspects of meaning when applied to the same noun. Subjects were read a paragraph and then asked to reply *true* or *false* to three test questions (see the following discussion). Some of the questions involved properties that were implicit in the paragraph. For instance, an interest in painting a tomato implicitly involves its color. The results showed that subjects verified the property of a noun more quickly (1.27 seconds vs. 1.39) and with a lower error rate (4.7% vs. 9.7%), in a context in which the property was relevant than in a context in which it was not relevant.

Paragraph 1

This still life would require great accuracy. The painter searched many days to find the color most suited to use in the painting of the ripe tomato.

Paragraph 2

The child psychologist watched the infant baby play with her toys. The little girl found a tomato to roll across the floor with her nose.

Test sentences

1) Tomatoes are red.

2) Balloons are heavy.

3) Tomatoes are round.

The results of this study show that the (category) context in which a name is used affects a subject's performance. However, until studies using C language constructs have been performed, we will not know the extent to which these context dependencies affect developer comprehension performance.

Are there situations where there might be a significant benefit in using identifiers with the same spelling, to denote different entities? Studies of word recall have found a frequency effect; the more concepts associated with a word, the fewer tip-of-the-tongue cases occur. Tag and typedef names denote very similar concepts, but they are both ways of denoting types.

word frequency
identifier
tip-of-the-tongue

```
1   typedef struct SOME_REC {int mem;} SOME_REC;
```

In the above case, any confusion on the developers' part will not result in unexpected affects (the name can appear either with or without the **`struct`** keyword— it is fail-safe).

Dev ??

A newly declared identifier denoting a tag may have the same spelling as a typedef name denoting the same type.

Macros provide a mechanism for hiding the syntactic context from readers of the visible source; for instance, the `offsetof` macro takes a tag name and a member name as arguments. This usage is rare and is not discussed further here.

Table 438.1: Identifiers appearing immediately to the right of the given token as a percentage of all instances of the given token. An identifier appearing to the left of a **`:`** could be a label or a **`case`** label. However, C syntax is designed to be parsed from left to right and the presence, or absence, of a **`case`** keyword indicates the entity denoted by an identifier. Based on the visible form of the `.c` files.

Token	.c file	.h file	Token	.c file	.h file
`goto` identifier	99.9	100.0	**`struct`** identifier	99.0	88.4
`#define` identifier	99.9	100.0	**`union`** identifier	65.5	75.8
`.` identifier	100.0	99.8	**`enum`** identifier	86.6	53.6
`->` identifier	100.0	95.5	**`case`** identifier	71.3	47.2

439 Thus, there are separate *name spaces* for various categories of identifiers, as follows:

Commentary

In C each name space is separate from other name spaces. As the name suggests this concept can be viewed as a space within which names exist. This specification was introduced in C90 so that existing code was not broken. There is also a name space for macro names.

macro one name space

The standard specifies various identifiers, having particular spellings, as being reserved for various purposes. The term *reserved name space* is sometimes used to denote identifiers belonging to this set. However, this term does not occur in the C Standard; it is an informal term used by the committee and some developers.

C++

C++ uses **`namespace`** as a keyword (there are 13 syntax rules associated with it and an associated keyword, **`using`**) and as such it is denotes a different concept from the C name space. C++ does contain some of the name space concepts present in C, and even uses the term namespace to describe them (which can be somewhat confusing). These are dealt with under the relevant sentences that follow.

Other Languages

Most languages have a single name space, which means that it is not necessary to talk about a name space as such. A few languages have a mechanism for separate translation that requires identifiers to be imported into a translation unit before they can be referenced. This mechanism is sometimes discussed in terms of being a name space.

Common Implementations

Some very early implementations of C did not always fully implement all of the name spaces defined in the C Standard. In particular early implementations treated the members of all structure and union types as belonging to the same name space[2] (i.e., once a member name was used in one structure type, it could not be used in another in the same scope).

Table 439.1: Occurrence of various kinds of declarations of identifiers as a percentage of all identifiers declared in all the given contexts. Based on the translated form of this book's benchmark programs.

Declaration Context	%	Declaration Context	%
block scope objects	23.7	file scope objects	4.4
macro definitions	19.3	macro parameters	4.3
function parameters	16.8	enumeration constants	2.1
struct/union members	9.6	**`typedef`** names	1.2
function declarations	8.6	tag names	1.0
function definitions	8.1	label names	0.9

---

label name space

— *label names* (disambiguated by the syntax of the label declaration and use); 440

Commentary

labeled statements syntax label name

Labels represent statement locations within a function body. Putting label names in their own name space represents a language design decision. It enables uses such as `if (panic) goto panic`.

C++

6.1p1 *Labels have their own name space and do not interfere with other identifiers.*

Other Languages

In Pascal and Ada labels must be declared before use. In Pascal they are numeric, not alphanumeric, character sequences. Some languages (e.g., Algol 60, Algol 68) do put label names in the same name space as ordinary identifiers.

Common Implementations

Some early implementations of C placed labels in the same name space as ordinary identifiers.

---

441 — the *tags* of structures, unions, and enumerations (disambiguated by following any[24)] of the keywords **struct**, **union**, or **enum**); tag name space

Commentary

The original definition of C[6] did not include support for typedef names. Tags were the mechanism by which structure, union, and enumerated types could be given a name that could be referred to later. Putting tags in a different name space removed the possibility of a tag name clashing with an ordinary identifier. It also has the advantage on large projects of reducing the number of possible naming conflicts, and allowing declarations such as the following to be used: tag declarations different scope

```
typedef struct listitem listitem;
```

C++

Tags in C++ exist in what is sometimes known as *one and a half name spaces*. Like C they can follow the keywords **struct**, **union**, or **enum**. Under certain conditions, the C++ Standard allows these keywords to be omitted.

> *The name lookup rules apply uniformly to all names (including* `typedef-names` *(7.1.3),* `namespace-names` *(7.3) and* `class-names` *(9.1)) wherever the grammar allows such names in the context discussed by a particular rule.* 3.4p1

In the following:

```
struct T {int i;};
struct S {int i;};
int T;

void f(T p); // Ill-formed, T is an int object
             /* Constraint violation */

void g(S p); // Well-formed, C++ allows the struct keyword to be omitted
             // There is only one S visible at this point
             /* Constraint violation */
```

C source code migrated to C++ will contain the **struct/union** keyword. C++ source code being migrated to C, which omits the *class-key*, will cause a diagnostic to be generated.

The C++ rules for tags and typedefs sharing the same identifier are different from C.

> *If the name in the* `elaborated-type-specifier` *is a simple identifier, and unless the* `elaborated-type-specifier` *has the following form:*
>
> `class-key identifier ;`
>
> *the identifier is looked up according to 3.4.1 but ignoring any non-type names that have been declared. If this name lookup finds a* `typedef-name`*, the* `elaborated-type-specifier` *is ill-formed.* 3.4.4p2

The following illustrates how a conforming C and C++ program can generate different results:

```
extern int T;

int size(void)
{
struct T {
```

```
          double mem;
          };

return sizeof(T);   /* sizeof(int) */
                    // sizeof(struct T)
}
```

The following example illustrates a case where conforming C source is ill-formed C++.

```
struct TAG {int i;};
typedef float TAG;

struct TAG x; /* does not affect the conformance status of the program */
              // Ill-formed
```

Other Languages

Tags are unique to C (and C++).

Coding Guidelines

tag naming conventions

A tag name always appears to the right of a keyword. Its status as a tag is clearly visible to the reader. The issue of tag naming conventions is discussed elsewhere.

Example

```
typedef struct X_REC {
                    int mem1;
                    } X_REC;
```

---

members name space

— the *members* of structures or unions; 442

Commentary

base document

In the base document, all members of structure and union types occupied the same name space (an idea that came from BCPL). Such member names were essentially treated as symbolic forms for offsets into objects.

```
struct {
        char mem_1;
        long mem_2;
       } x;
struct {
       char mem_3;
       char mem_4;
       } y;

void f(void)
{
/*
 * Under the original definition of C the following would zero a char
 * sized object in x that had the same offset as mem_4 had in y.
 */
x.mem_4 = 0;
}
```

This language specification prevented structures and unions from containing a particular member name once it had been used in a previous definition as a member name. The idea of member names representing offsets was a poor one and quickly changed. A consequence of this history is that structure and union definitions are thought of as creating a name space, not in terms of members existing in a scope (a usage common to many other languages, including C++).

C++

The following rules describe the scope of names declared in classes. 3.3.6p1

In C++ members exist in a scope, not a name space.

```
1  struct {
2          enum E_TAG { E1, E2}  /* C identifiers have file scope */
3                                // C++ identifiers have class scope
4                                m1;
5          } x;
6
7  enum E_TAG y; /* C conforming */
8                // C++ no identifier names E_TAG is visible here
```

Other Languages

Many languages treat the contents of what C calls a structure or union as a scope. The member-selection operator opening this scope to make the members visible (in C this operator makes the unique name space visible).

---

443 each structure or union has a separate name space for its members (disambiguated by the type of the expression used to access the member via the `.` or `->` operator); member namespace

Commentary

Although each structure or union type definition creates its own unique name space, no new scope is created. The declarations contained within these definitions have the scope that exists at the point where the structure or union is defined.

identifier
scope determined by declaration placement

C++

The name of a class member shall only be used as follows: 3.3.6p2

. . .

— *after the* `.` *operator applied to an expression of the type of its class (5.2.5) or a class derived from its class,*

— *after the* `->` *operator applied to a pointer to an object of its class (5.2.5) or a class derived from its class,*

```
1  struct {
2          enum {E1, E2} m;
3          } x;
4
5  x.m = E1; /* does not affect the conformance status of the program */
6            // ill-formed. X::E1 is conforming C++ but a syntax violation in C
```

Coding Guidelines

Given that one of two tokens immediately precedes a member name its status as a member is immediately obvious to readers of the source (the only time when this context may not be available is when a member name occurs as an argument in a macro invocation). Because of the immediate availability of this information there is no benefit in a naming convention intended to flag the status of an identifier as a member.

Members in different structure types may hold the same kind of information; for instance, a member named `next` might always point to the next element in a linked list, its type being a pointer to the structure type containing its definition. Members named `x_coord` and `y_coord` might occur in several structure types dealing with coordinate systems.

identifier ??
reusing names

The underlying rationale for the guideline recommendation dealing with reusing the same identifier name, confusion cased by different semantic associations, is only applicable if the semantic associations are different. If they are the same, then there is a benefit in reusing the same name when dealing with different members that are used for the same purpose.

Dev ??

A newly declared member may have the same spelling as a member in a different structure/union type provided they both share the same semantic associations; and if they both have an arithmetic type, their types are the same.

The term *same semantic associations* is somewhat ill-defined and subject to interpretation (based on the cultural background and education of the person performing the evaluation, an issue that is discussed in more detail elsewhere). While guideline recommendation wording should normally aim to be precise, it should also try to be concise. It is not clear that this guideline would be improved by containing more words.

semantic associations
enumerating

Members having arithmetic types could be interchanged in many situations without any diagnostic being issued. A member having type **`int`** in one structure and type **`float`** in another, for instance, may represent the same semantic concept; but the way they need to be handled is different. Inappropriately interchanging nonarithmetic types is much more likely to result in a diagnostic being generated.

A study by Neamtiu, Foster, and Hicks[5] 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 79% of releases one or more existing structure or union types had one or more fields added to them, while structure or union types had one or more fields deleted in 51% of releases and had one or more of their field names changed in 37% of releases. One or more existing fields had their types changed in 35% of releases.[4]

A study by Anquetil and Lethbridge[1] analyzed 2 million lines of Pascal (see Table 443.1 and Table 443.2). Members that shared the same name were found to be much more likely to share the same type than members having different names.

Table 443.1: Number of matches found when comparing between pairs of members contained in different Pascal records that were defined with the same type name. Adapted from Anquetil and Lethbridge.[1]

	Member Types the Same	Member Types Different	Total
Member names the same	73 (94.8%)	4 (5.2%)	77
Member names different	52 (11 %)	421 (89 %)	473

Table 443.2: Number of matches found when comparing between pairs of members contained in different Pascal records (that were defined with the any type name). Adapted from Anquetil and Lethbridge.[1]

	Member Types the Same	Member Types Different	Total
Member names the same	7,709 (33.7%)	15,174 (66.3%)	22,883
Member names different	158,828 (0.2%)	66,652,062 (99.8%)	66,710,890

Example

```
1  struct {
2          int m;
```

```
        } x;
struct {
         int m;
         struct {
                 int n;
                } n;
        } y;
```

the two members, named `m`, are each in a different name space. In the definition of `y` the two members, named `n`, are also in different name spaces.

444 — all other identifiers, called *ordinary identifiers* (declared in ordinary declarators or as enumeration constants).

ordinary identifiers
name space
ordinary identifiers

Commentary

This defines the term *ordinary identifiers*. These ordinary identifiers include objects, functions, typedef names and enumeration constants. Macro names and macro parameters are not included in this list. Keywords are part of the language syntax. Identifiers with the spelling of a keyword only stop being identifier preprocessing tokens and become keyword tokens in phase 7. Literals are not in any name space.

C++

The C++ Standard does not define the term *ordinary identifiers*, or another term similar to it.

Coding Guidelines

Naming conventions that might be adopted to distinguish between these ordinary identifiers, used for different purposes, are discussed in more detail in their respective sentences.

enumeration constant naming conventions
typedef naming conventions
macro naming conventions
function naming conventions

445 **Forward references:** enumeration specifiers (6.7.2.2), labeled statements (6.8.1), structure and union specifiers (6.7.2.1), structure and union members (6.5.2.3), tags (6.7.2.3), the **goto** statement (6.8.6.1).

446 23) As specified in 6.2.1, the later declaration might hide the prior declaration.

footnote 23

Example

```
extern int glob;

void f(void)
{
int glob; /* Hide prior declaration. */
   {
   extern int glob;
   }
}
```

447 24) There is only one name space for tags even though three are possible.

footnote 24

Commentary

The additional functionality, if three had been specified, would be to enable the same identifier spelling to be used after each of the three keywords. There is little to be gained from this and the possibility of much confusion.

C++

There is no separate name space for tags in C++. They exist in the same name space as object/function/typedef *ordinary identifiers*.

Common Implementations

Early translators allowed the **`struct`** and **`union`** keywords to be intermixed.

```
union Tag {
          int mem;
          };

int main(void)
{
struct Tag *ptr;   /* Acceptable in K&R C. */
}
```

References

1. N. Anquetil and T. Lethbridge. Assessing the relevance of identifier names in a legacy software system. In *Proceedings of CASCON'98*, pages 213–222, 1998.
2. B. W. Kernighan. Programming in C-A tutorial. Technical Report ???, Bell Laboratories, Aug. ???
3. G. McKoon and R. Ratcliff. Contextually relevant aspects of meaning. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, 14(2):331–343, 1988.
4. I. Neamtiu. Detailed break-down of general data provided in paper[5] kindly supplied by first author. Jan. 2008.
5. 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.
6. D. M. Ritchie, B. W. Kernighan, and M. E. Lesk. The C programming language. Technical Report 31, Bell Telephone Laboratories, Oct. 1975.