The New C Standard (Excerpted material)

An Economic and Cultural Commentary

Derek M. Jones
derek@knosof.co.uk

6.2.1 Scopes of identifiers

scope of identifiers

An identifier can denote an object; 390

Commentary

This association is created by an explicit declaration.

Some objects do not have names— they are anonymous. An anonymous object can be created by a call to a memory-allocation function; an unnamed bit-field can be denoted by the identifier defined to have its containing union type.

Other Languages

Many languages use the term *variable* to denote what the C Standard calls an *object*. All languages provide a mechanism for declaring identifiers to denote objects (or variables).

Coding Guidelines

naming conventions

The use to which an identifier is put has been known to influence the choice of its name. This issue is fully discussed elsewhere.

a function; 391

Commentary

It is not possible to define an anonymous function within a strictly conforming program. Machine code can be copied into an object and executed on some implementations. Such copying and execution does not create something that is normally thought of as a function.

Coding Guidelines

Various coding guideline documents have recommended that names based on various parts of speech be used in the creating of identifier names denoting function. These recommendations are invariably biased towards the parts of speech found in English. (Not speaking any other human language, your author has not read documents written in other languages but suspects they are also biased toward their respective languages.)

identifier introduction

The general issue of identifier names is discussed elsewhere.

a tag or a member of a structure, union, or enumeration; 392

Commentary

It is possible to create anonymous structure and union types, and anonymous members within these types. It is not possible to create an anonymous member of an enumeration. Even if there are gaps in the values assigned to each enumeration constant, there are no implicit named members.

C++

The C++ Standard does not define the term *tag*. It uses the terms *enum-name* (7.2p1) for enumeration definitions and *class-name* (9p1) for classes.

Common Implementations

Some translators support anonymous structure and union members within a structure definition; there was even a C9X revision proposal, WG14/N498 (submitted by one Ken Thompson):

```
struct S {
          int mem1;
          union {
                 long umem1;
                 float umem2;
                };  /* No member name. */
         } s_o;

void f(void)
```

```
{
s_o.umem1=99; /* Translator deduces 'expected' member. */
}
```

There are several implementations that support this form of declaration. It is also supported in C++.

---

393 a typedef name;

Commentary

Character sequences such as **char**, **short** and **int** are types, not typedef names. They are also language keywords— a different lexical category from identifiers.

Common Implementations

Typedef names are specified as being identifiers, not keywords. However, most implementations treat them as a different syntactic terminal from identifiers. Translators usually performs a symbol table lookup just before translation phase 7, to see if an identifier is currently defined as a typedef name. This differentiation is made to simplify the parsing of C source and not visible to the user of a translator.

transla-tion phase 7

Coding Guidelines

The issue of naming conventions typedef names is discussed elsewhere.

typedef naming conventions

---

394 a label name;

label name

Commentary

A label name occurs in a different syntactic context to other kinds of identifiers, which is how a translator deduces it is a label.

label name space
labeled statements syntax

Other Languages

Fortran and Pascal require that label names consist of digits only.

Coding Guidelines

The issue of naming conventions for label names is discussed elsewhere.

label naming conventions

---

395 a macro name;

Commentary

An identifier can only exist as a macro name during translation phases 3 and 4.

macro definition lasts until
transla-tion phase 3
transla-tion phase 4

Coding Guidelines

The issue of typedef names is discussed elsewhere.

macro naming conventions

---

396 or a macro parameter.

identifier macro parameter

Commentary

An identifier can only exist as a macro parameter during translation phases 3 and 4.

Function parameters are objects, not a special kind of identifier.

transla-tion phase 3
transla-tion phase 4
parameter

C++

The C++ Standard does not list macro parameters as one of the entities that can be denoted by an identifier.

Coding Guidelines

The issue of typedef names is discussed elsewhere.

typedef naming conventions

---

397 The same identifier can denote different entities at different points in the program.

identifier denote different entities

Commentary

The word *entities* is just a way of naming the different kinds of things that an identifier can refer to in C. The concept of scope allows the same identifier to be defined in a different scope, in the same name space, to refer to another entity. The concept of name space allows the same identifier to be defined in a different name space, in the same scope, to refer to another entity.

scope 400
name space

C++

The C++ Standard does not explicitly state this possibility, although it does include wording (e.g., 3.3p4) that implies it is possible.

Other Languages

Languages that support some kind of scoping rules usually allow the same identifier to denote different entities at different points in a program.

Cobol has a single scope and requires that identifiers be defined in a declaration section. This severely limits the extent to which the same identifier can be used to denote different entities. However, members of the structure data type can use the same names in different types. Some dialects of Basic have a single scope.

Coding Guidelines

Some of the costs and benefits of using the same identifier to denote different entities include:

- costs: an increase in the number of developer miscomprehensions when reading the source. For instance, the same identifier defined as a file scope, static linkage, object in one source file and a file scope, extern linkage, object in another source file could mislead the unwary developer, who was not

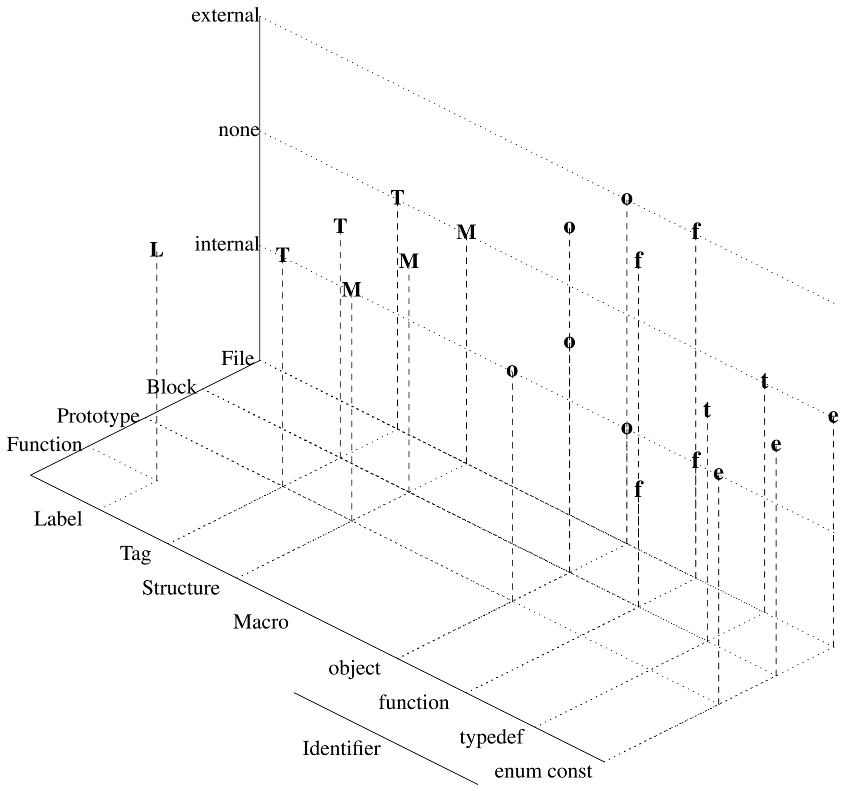

Figure 397.1: All combinations of linkage, scope, and name space that all possible kinds of identifiers, supported by C, can have. **M** refers to the members of a structure or union type. There is a separate name space for macro names and they have *no linkage*, but their scope has no formally specified name.

aware of the two uses of the same name, into making incorrect assumptions about an assignment to one of the objects. The confusability of one identifier with another identifier is one of the major issues of identifier naming and is discussed elsewhere, identifier confusability

- benefits: an increase in reader recall performance through consistent use of identifier names to denote the same set of semantic attributes. For instance, having the same identifier denoting a member in different structures can indicate that they all have the same usage (e.g., x_cord to hold the position on the x axis in a graph drawing program). identifier semantic associations member naming conventions

The issues involved in making cost/benefit trade-offs related to identifier spelling and the role played by an identifier and are discussed elsewhere. identifier syntax object role

Example

```
#define SAME(SAME) SAME /* Macro and macro parameter name spaces. */
/*
 * Additional lack of clarity could be obtained by replacing
 * any of the following identifiers, say id, by SAME(id).
 */

typedef struct SAME {          /* Tag name space. */
                    int SAME;  /* Unique name space of this structure definition. */
                    } SAME;    /* Ordinary identifier name space. */

SAME *(f(SAME SAME))(struct SAME SAME) /* Different scopes. */
{
SAME(SAME);                    /* A use (of the macro). */
if (SAME.SAME == sizeof(SAME)) /* More uses. */
   goto SAME;                  /* Label name space. */
else
   {                           /* A new scope. */
   enum SAME {
              SAME             /* Different name space. */
             } loc = SAME;     /* A use. */
   }

SAME:;                         /* Label name space. */
}
```

398 A member of an enumeration is called an *enumeration constant*. enumeration constant

Commentary

This defines the term *enumeration constant*.

C++

There is no such explicit definition in the C++ Standard (7.2p1 comes close), although the term *enumeration constant* is used.

Other Languages

The term *enumeration constant* is generic to most languages that contain enumerated types.

Coding Guidelines

It is important to use this terminology. Sometimes the terms *enumeration identifier* or *enumeration value* are used by developers. It is easy to confuse these terms with other kinds of identifiers, or values.

399 Macro names and macro parameters are not considered further here, because prior to the semantic phase of program translation any occurrences of macro names in the source file are replaced by the preprocessing token sequences that constitute their macro definitions.

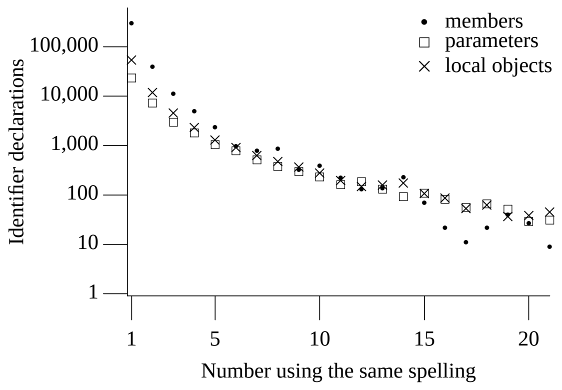

Figure 397.2: Number of declarations of an identifier with the same spelling in the same translation unit. Based on the translated form of this book's benchmark programs. Note that members of the same type are likely to be counted more than once (i.e., they are counted in every translation unit that declares them), while parameters and objects declared within function definitions are likely to be only counted once.

Commentary

macro definition lasts until
macro definition no significance after
preprocessing directives deleted
translation phase 7

Macro names are different from all other names in that they exist in a single scope that has no nesting. Macro definitions cease to exist after translation phase 4, as do preprocessing directives. The so-called *semantic phase* is translation phase 7.

Common Implementations

A few translators tag tokens with the macros they were expanded from, if any, as an aid to providing more informative diagnostic messages.

Coding Guidelines

Some of these coding guidelines apply to the source code that is visible to the developer. In such cases macro names, if they appear in the visible source, need to be considered in their unexpanded form.

preprocessor directives syntax

#undef

The concept of scope implies a coding structure that does not really exist during preprocessing (use of conditional inclusion does not create a new scope). The preprocessor views its input as an unstructured (apart from preprocessing directives) sequence of preprocessing tokens, some of which have special meaning. While it is possible to use **`#define`/`#undef`** pairs to simulate a scope, this usage is not common in practice.

---

visible identifier scope

For each different entity that an identifier designates, the identifier is *visible* (i.e., can be used) only within a region of program text called its *scope*. 400

Commentary

translation phase 7

lifetime of object

This defines the terms *visible* and *scope*. Visibility here means visible using the mechanisms defined in the standard for looking up identifiers (that have been previously declared). The concept of scope, in C, is based on the textual context in which an identifier occurs in translation phase 7— so-called *lexical scoping* (sometimes called *static scoping*, as opposed to *dynamic scoping*, which is based on when an object is created during program execution). The scope of a named object is closely associated with its lifetime in many cases.

C++

3.3p1 *In general, each particular name is valid only within some possibly discontiguous portion of program text called its scope.*

3.3p4

Local extern declarations (3.5) may introduce a name into the declarative region where the declaration appears and also introduce a (possibly not visible) name into an enclosing namespace; these restrictions apply to both regions.

If a name is in scope and is not hidden it is said to be visible. 3.3.7p5

Other Languages

The concept of scope exists in most programming languages. In Cobol, and some dialects of Basic, all variables have the same scope. Languages which make use of dynamic scoping include APL, Perl, Snobol 4, and Lisp (later versions of Lisp use static scoping; support for static scoping was introduced in version 5 of Perl). While being very flexible, dynamic scoping is not very runtime efficient because the actual object referenced has to be found during program execution; not being known at translation time, it is not possible to fix its address prior to program execution.

```
#include <stdlib.h>

int total = 0;

void f_1(void)
{
/*
 * With dynamic scoping the current function call chain is important.
 * If f_1 is called via f_2 then the local definition of total in f_2
 * would be assigned to.  If f_1 was called directly from main, the
 * file scope declaration of total would be assigned to.
 */
total=1;
}

void f_2(void)
{
int total;

f_1();
}

int main(void)
{
if (rand() > 20)
   f_1();
else
   f_2();

return total;
}
```

In some languages the scope of an identifier may not be the same region of program text as the one it is visible in. For instance Pascal specifies that the scope of an identifier starts at the beginning of the block that contains its declaration, but it is only visible from the point at which it is declared.

```
int sum;

void pascal_scope_rules(void)
{
/*
 * In Pascal the scope of the local declarations start here.  The scope of the
 * nested declaration of sum starts here and hides the file scope declaration.
 * But the nested declaration of sum does not become visible until after its'
 * declaration.
 */
int add_up = sum; /* In Pascal there is no identifier called sum visible here. */
int sum = 3;
}
```

Common Implementations

Implementations used to make use of the fact that identifiers were only visible (and therefore needed to be in their symbol table) within a scope to reduce their internal storage requirements; freeing up the storage used for an identifier's symbol table entry when the processing of the scope that contained its declaration was complete. Many modern implementations are not driven by tight storage restrictions and may keep the information for reasons of optimization or improved diagnostic messages.

Some implementations *exported* external declarations that occurred in block scope to file scope. For instance:

```
void f1(void)
{
/*
 * Some translators make the following external linkage
 * declaration visible at file scope...
 */
extern int glob;
}

void f2(void)
{
/*
 * ... which means that the following reference to glob refers
 * to the one declared in function f1.
 */
glob++;
}

void f3(void)
{
/*
 * Linkage ensures that the following declaration always
 * refers to the same glob object declared in function f1.
 */
extern int glob;

glob++;
}
```

Coding Guidelines

alias analysis
object aliased

Objects and functions are different from other entities in that it is possible to refer to them, via pointers, when the identifiers that designate them are not visible. While such anonymous accesses (objects referenced in this way are said to be *aliased*) can be very useful, but they can also increase the effort needed, by developers, to comprehend code. The issue of object aliases is discussed elsewhere.

---

same identifier

Different entities designated by the same identifier either have different scopes, or are in different name spaces. 401

Commentary

The C Standard's meaning of same identifier is that the significant characters used to spell the identifier are identical. Differences in non-significant characters are not considered. Attempting to declare an identifier in the same scope and name space as another identifier with the same spelling is always a constraint violation.

C90

In all but one case, duplicate label names having the same identifier designate different entities in the same scope, or in the same name space, was a constraint violation in C90. Having the same identifier denote two different labels in the same function caused undefined behavior. The wording in C99 changed to make this case a constraint violation.

label name
unique

C++

The C++ Standard does not explicitly make this observation, although it does include wording (e.g., 3.6.1p3) that implies it is possible.

Coding Guidelines

The concept of name space is not widely appreciated by C developers. Given the guideline recommendation that identifier names be unique, independently of what name space they are in, there does not appear to be any reason for wanting to educate developers further on the concept of name space.

?? identifier
reusing names

---

402 There are four kinds of scopes: function, file, block, and function prototype.

scope
kinds of

Commentary

File scope is also commonly called *global scope*, not a term defined by the standard. Objects declared at file scope are sometimes called *global objects*, or simply *globals*.

Block scope is also commonly called *local scope*, not a term defined by the standard. Objects declared in block scope are sometimes called *local objects*, or simply *locals*. A block scope that is nested within another block scope is often called a *nested scope*.

The careful reader will have noticed a subtle difference between the previous two paragraphs. Objects were declared "at" file scope, while they were declared "in" block scope. Your author cannot find any good reason to break with this common developer usage and leaves it to English majors to rant against the irregular usage.

Identifiers defined as macro definitions are also said to have a scope.

macro
definition lasts
until

C++

The C++ Standard does not list the possible scopes in a single sentence. There are subclauses of 3.3 that discuss the five kinds of C++ scope: function, namespace, local, function prototype, and class. A C declaration at file scope is said to have *namespace scope* or *global scope* in C++. A C declaration with block scope is said to have *local scope* in C++. Class scope is what appears inside the curly braces in a structure/union declaration (or other types of declaration in C++).
Given the following declaration, at file scope:

```
struct S {
        int m; /* has file scope */
               // has class scope
        } v;   /* has file scope */
               // has namespace scope
```

Other Languages

File and block scope are concepts that occur in many other languages. Some languages only allow definitions within functions to occur in the outermost block (e.g., Pascal); this may nor may not be equated to function scope. Function prototype scope is unique to C (and C++).

Coding Guidelines

Many developers are aware of file and block scope, but not the other two kinds of scope. Is anything to be gained by educating developers about these other scopes? Probably not. There are few situations where they

404 scope
function
409 scope
function prototype

prototype 409 providing identifiers

are significant. These coding guidelines concentrate on the two most common cases— file and block scope (one issue involving function prototype scope is discussed elsewhere).

(A *function prototype* is a declaration of a function that declares the types of its parameters.) 403

Commentary

This defines the term *function prototype*; it appears in parentheses because it is not part of the main subject of discussion in this subclause. A function prototype can occur in both a function declaration and a function definition. Function prototype scope only applies to the parameters in a function prototype that is purely a declaration. The parameters in a function prototype that is also a definition have block scope.

scope 409 function prototype
block scope 408 terminates

---

scope function

A label name is the only kind of identifier that has *function scope*. 404

Commentary

labeled statements syntax
goto statement

Labels are part of a mechanism (the **goto** statement) that can be used to change the flow of program execution. The C language permits arbitrary jumps to the start of any statement within a function (that contains the jump). To support this functionality label name visibility needs to cut across block boundaries, being visible anywhere within the function that defines them.

Other Languages

Few other languages define a special scope for labels. Most simply state that the labels are visible within the function that defines them, although a few (e.g., Algol 68) give labels block scope (this prevents jumps into nested blocks).

Languages that supported nested function definitions and require labels to be defined along with objects, often allow labels defined in outer function definitions to be referenced from functions defined within them.

Common Implementations

gcc supports taking the address of a label. It can be assigned to an object having a pointer type and passed as an argument in a function call. The label identifier still has function scope, but anonymous references to it may exist in other function scopes during program execution.

Coding Guidelines

This scope, which is specific to labels, is not generally known about by developers. Although they are aware that labels are visible throughout a function, developers tend not to equate this to the concept of a separate kind of scope. There does not appear to be benefit educating developers about its existence.

---

label declared implicitly

It can be used (in a **goto** statement) anywhere in the function in which it appears, and is declared implicitly by its syntactic appearance (followed by a **:** and a statement). 405

Commentary

goto statement

The C Standard provides no other mechanism for defining labels. Labels are the only entities that may, of necessity, be used before they are defined (a forward jump).

Other Languages

Some languages allow labels to be passed as arguments in calls to functions and even assigned to objects (whose contents can then be **goto**ed.)

Pascal uses the **label** keyword to define a list of labels. Once defined, these labels may subsequently label a statement or be referenced in a **goto** statement. Pascal does not support declarations in nested blocks; so labels, along with all locally declared objects and types, effectively have function scope.

Common Implementations

gcc allows a label to appear as the operand of the **&&** unary operator (an extension that returns the address of the label).

406 Every other identifier has scope determined by the placement of its declaration (in a declarator or type specifier).

identifier scope determined by declaration placement

Commentary

Every other identifier must also appear in a declarator before it is referenced. The C90 support for implicit declarations of functions, if none was currently visible, has been removed in C99. The textual placement of a declarator is the only mechanism available in C for controlling the visibility of the identifiers declared.

Other Languages

Languages that support a more formal approach to separate compilation have mechanisms for importing previously declared identifiers into a scope. Both Java and C++ support the **`private`** and **`public`** keywords, these can appear on a declarator to control the visibility of any identifiers it declares. Ada has a sophisticated separate compilation mechanism. Many other languages use very similar scoping mechanisms to those defined by the C Standard.

translation unit syntax

Coding Guidelines

Some coding guideline documents recommend that the scope of identifiers be minimized. However, such a recommendation is based on a misplaced rationale. It is not an identifier's scope that should be minimized, but its visibility. For instance, an identifier at file scope may, or may not, be visible outside of the translation unit that defines it. Some related issues are discussed elsewhere.

identifier definition close to usage

407 If the declarator or type specifier that declares the identifier appears outside of any block or list of parameters, the identifier has *file scope*, which terminates at the end of the translation unit.

file scope

Commentary

Structure and union members and any other identifiers declared within them, at file scope, also have file scope.

The return type on a function definition is outside of the outermost block associated with that definition. However, the parameters are treated as being inside it. Thus, any identifiers declared in the return type have file scope, while any declared in the parameter list have block scope.

408 block scope terminates

File scope may terminate at the end of the translation unit, but an identifier may have a linkage which causes it to be associated with declarations outside of that translation unit.

translation units communication between

Where the scope of an identifier begins is defined elsewhere.

416 tag scope begins
417 enumeration constant scope begins
418 identifier scope begins

C++

A name declared outside all named or unnamed namespaces (7.3), blocks (6.3), function declarations (8.3.5), function definitions (8.4) and classes (9) has global namespace scope (also called global scope). The potential scope of such a name begins at its point of declaration (3.3.1) and ends at the end of the translation unit that is its declarative region.

3.3.5p3

Other Languages

Nearly every other computer language supports some form of file scope declaration. A few languages, usually used in formal verification, do not allow objects to have file scope (allowing such usage can make it significantly more difficult to prove properties about a program).

In some languages individual identifiers do not exist independently at file scope— they must be part of a larger whole. For instance, in Fortran file scope objects are declared within **`common`** blocks (there can be multiple named common blocks, but only one unnamed common block). An entire common block (with all the identifiers it contains) needs to be declared within a source file, it is not possible to declare one single identifier (unless it is the only one declared by the common block). Other kinds of identifier groupings include *package* (Ada and Java), *namespace* (C++), *module* (Modula-2, Pascal), *cluster* (Clu), and *unit* (Borland Delphi). The underlying concept is the same, identifiers are exported and imported as a set.

Coding Guidelines

Some coding guidelines documents recommend that the number of objects declared at file scope be minimized. However, there have been no studies showing that alternative design/coding techniques would have a more worthwhile cost/benefit associated with their use.

The reasons for the declaration of an identifier to appear at file can depend on the kind of identifier being declared. Identifiers declared at file scope often appear in an order that depends on what they denote (e.g., macro, typedef, object, function, etc.). The issues associated with this usage are discussed elsewhere. The rest of this coding guideline subsection discusses the declaration of types, functions, and objects at file scope. Identifiers declared as types must appear at file scope if

external declaration syntax

- objects declared to have these types appear at file scope,
- objects declared to have these types appear within more than one function definition (having multiple, textual, declarations of the same type in different block scopes is likely to increase the cost of maintenance and C does not support the passing of types as parameters),
- other types, at files scope, reference these types in their own declaration.

Identifiers declared as functions must appear at file scope in those cases where two or more functions have a mutual calling relationship. This issue is discussed elsewhere.

function call recursive

Any program using file scope objects can be written in a form that does not use file scope objects. This can be achieved either by putting all statements in the function `main`, or by passing, what were file scope, objects as parameters. Are these solutions more cost effective than what they replaced? The following is a brief discussion of some of the issues.

The following are some of the advantages of defining objects at file scope (rather than passing the values they contain via parameters) include:

- Efficiency of execution. Accessing objects at file scope does not incur any parameter passing overheads. The execution-time efficiency and storage issues are likely to be a consideration in a freestanding environment, and unlikely to be an issue in a hosted environment.
- Minimizes the cost of adding new function definitions to existing source code. If the information needed by the new function is not available in a visible object, it will have to be passed as an argument. The function calling the new function now has a requirement to access this information, to pass it as a parameter. This requirement goes back through all call chains that go through the newly created function. If the objects that need to be accessed have file scope, there will be no need to add any new arguments to function calls and parameters to existing function definitions. In some cases using parameters, instead of file scope objects, can dramatically increase the number of arguments that need to be passed to many functions; it depends on how information flows through a program. If the flow is hierarchical (i.e., functions are called in a tree-like structure), it is straight-forward to pass information via parameters. If the data flow is not hierarchical, but like an undirected graph: with both `x` and `y` calling `a`, it is necessary for `p` to act as key holder for any information they need to share with `a`. The degree to which information flow and control flow follow each other will determine the ease, or complexity, of using parameters to access information. The further up the call chain the shared calling function, the greater the number of parameters that need to be passed through.

limit parameters in definition

The following are some of the disadvantages of defining objects at file scope:

- Storage is used for the duration of program execution.
- There is a single instance of object. Most functions are not recursive, so separate objects for nested invocations of a function are not usually necessary.

function call recursive

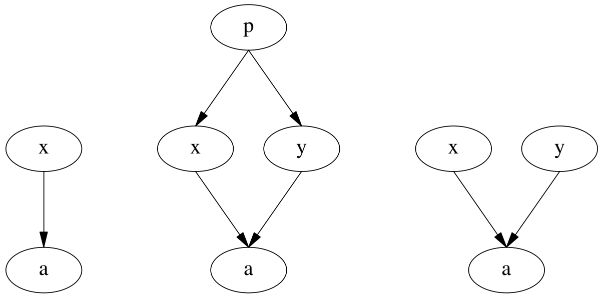

Figure 407.1: Some of the ways in which a function can be called— a single call from one other function; called from two or more functions, which in turn are all called by a single function; and called from two or more functions whose nearest shared calling function is not immediately above them.

- Reorganizing a program by moving function definitions into different translation units requires considering the objects they access. These may need to be given external linkage.
- Greater visibility of identifiers reduces the developer effort needed to access them, leading to a greater number of *temporary accesses* (the answer to the question, “how did all these unstructured accesses get into the source?” is, “one at a time”).
- Information flow between functions is implicit. Developers need to make a greater investment in comprehending which calls cause which objects to be modified. Experience shows that developers tend to overestimate the reliability of their knowledge of which functions access which file scope identifiers.
- When reading the source of a translation unit, it is usually necessary to remember all of the file scope objects defined within it (locality of reference suggests that file scope identifiers referenced are more likely to be those defined in the current, rather than other, translation units). Reducing the number of file scope objects reduces the amount of information that needs to be in developers long-term memory.
- The only checks made on references to file scope objects is that they are visible and that the necessary type requirements are met. For issues, such as information flow, objects required to have certain values at certain points are not checked (such checking is in the realm of formal checking against specifications). Passing information via parameters does not guarantee that mistakes will not be made; but the need to provide arguments acts as a reminder of the information accessed by a function and the possible consequences of the call.

What is the algorithm for deciding whether to use parameters or file scope objects? The answer to this question involves many complex issues that are still poorly understood.

The argument that many programs exhibit faults because of the unconstrained use of objects at file scope, therefore use of parameters must be given preference, is too narrowly focused. Because it is the least costly solution, in terms of developer effort, file scope objects are more likely to be used than parameter passing. Given that many programs do not have long lifespans their maintenance costs are small, or non-existent. The many small savings accrued over incremental changes to many small programs over a relatively short lifespan may be greater than the total costs incurred from the few programs that are maintained over longer periods of time. If these savings are not greater than the costs, what is the size of the loss? Is it less than the value $saving - cost$ for the case in which developers have to invest additional effort in passing information via parameters? Without reliable evidence gathered from commercial development projects, these coding guidelines are silent on the topic of use file scope versus use of parameters.

Although they don’t have file scope, the scope of macro names does terminate at the same point as identifiers that have file scope. Macro definitions are discussed elsewhere.

macro definition lasts until

The issue of how an identifier’s scope might interact with its spellings is discussed elsewhere.

scope naming conventions

Example

```
enum {E1, E2} /* File scope. */
              f(enum {E3, E4} x) /* Block scope. */
{ /* ... */ }

void g(c)
enum m {q, r} c; /* Block scope, in the list of parameter declarations. */
{ /* ... */ }
```

---

block scope terminates

If the declarator or type specifier that declares the identifier appears inside a block or within the list of parameter declarations in a function definition, the identifier has *block scope*, which terminates at the end of the associated block. 408

Commentary

tag 416 scope begins
enumeration 417 constant scope begins
identifier 418 scope begins

This defines the term *block scope* and specifies where it terminates. Where the scope of an identifier begins is defined elsewhere.

Structure and union members, and other identifiers declared within them (but not those defined as macro names), in block scope, also have block scope.

The reference to parameter declarations does not distinguish between those declared via a function prototype, or an old style function declaration. They both have block scope, denoted by the outermost pair of braces.

linkage

It is possible for an object or function declared in block scope to refer to the same object or function at file scope, or even in a different translation unit, through a process called *linkage*.

C++

3.3.2p1 *A name declared in a block (6.3) is local to that block. Its potential scope begins at its point of declaration (3.3.1) and ends at the end of its declarative region.*

3.3.2p2 *The potential scope of a function parameter name in a function definition (8.4) begins at its point of declaration. . . . , else it ends at the outermost block of the function definition.*

Other Languages

A few languages, for instance Cobol, do not have any equivalent to block scope. In Java local variables declared in blocks follow rules similar to C. However, there are situations where the scope of a declaration can reach back to before its textual declaration.

Gosling[1] *The scope of a member declared in or inherited by a class type (8.2) or interface (9.2) is the entire declaration of the class or interface type.*

```
class Test {
   test() { k = 2; }
   int k;
   }
```

Common Implementations

Parameters in a function definition having block scope, rather than function prototype scope causes headache for translator implementors. The traditional single pass, on-the-fly translation process does not know that the parameters have block scope until an opening curly bracket, a semicolon, or a type (for old style function definitions) is seen during syntax analysis.

implementation single pass

Coding Guidelines

When should identifiers be defined with block scope?

Possible benefits of declaring functions in block scope is to remove the overhead associated with having to **#include** the appropriate header (in translation environments having limited storage capacity this can be a worthwhile saving), and to allow the source of a function definition to be easily copied to other source files (the cut-and-paste model of software development). Possible benefits of defining objects in block scope is that storage only needs to be allocated when the block containing it is executed, and objects within each block can be thought about independently (i.e., developers do not need to remember information about objects defined in other blocks).

Experience shows that when writing programs, developers often start out giving most objects block scope. It is only when information needs to be shared between different functions that the possible use of file scope, as opposed to parameters, needs to be considered. This issue is discussed in the previous C sentence.

407 file scope

It is rare for a declaration of an identifier denoting a typedef, tag, or enumeration constant to occur in block scope. Data structures important enough to warrant such a declaration are normally referenced by more than one function. Such usage could be an indicator that a function is overly long and needs to be broken up into smaller functions.

identifiers number in block scope

The same coding guideline issues might also apply to macro definitions. That is, macro names that are only used within a single function definition be defined at the start of that function (along with other definitions local to that function). However, macros do not have block scope and existing practice is for macro definitions to be located at the start of source files (and this is where developers learned to expect to find them). Calculating whether there is a worthwhile cost/benefit in using **#define**/**#undef** pairs to emulate a block scope is likely to be difficult and these coding guidelines are silent on the issue.

macro definition emulate block scope

preprocessor directives syntax

Example

```
void DR_035(c)
enum m{q, r} /* m, q and r all have block scope, not file scope. */
             c;
{ /* ... */ }

void f(void)
{
/*
 * The following function declaration declares a type in its
 * return type.  This has block scope, effectively rendering
 * calls to this function as undefined behavior.
 */
extern struct {int i;} undefined_func(void);
}
```

409 If the declarator or type specifier that declares the identifier appears within the list of parameter declarations in a function prototype (not part of a function definition), the identifier has *function prototype scope*, which terminates at the end of the function declarator.

scope function prototype

Commentary

This type of scope was created by the C Committee for the C90 Standard (it was not in the original base document). This kind of scope is needed because the Committee decide to permit declarations of the form

base document

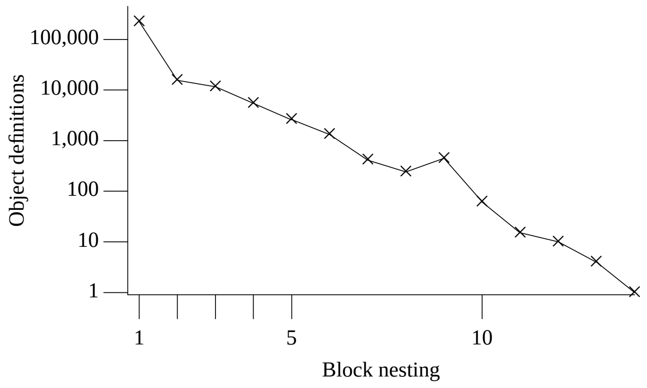

Figure 408.1: Number of object declarations appearing at various block nesting levels (level 1 is the outermost block). Based on the translated form of this book's benchmark programs.

prototype 409 providing identifiers

`extern void f(int x);`, where an identifier is specified for the parameter. Some developers believe that use of parameter names provides a benefit (this issue is discussed elsewhere).

A function declaration that defines a structure or union tag, or an anonymous structure or union type, in its function prototype scope renders the identifier it declares uncallable within the current translation unit (unless an explicit cast is applied to the function declarator at the point of call). This is because any calls to it cannot contain arguments whose types are compatible with the types of the parameters (which are declared using types that are only visible inside the function prototype scope). In the case of enumeration types declared in a function prototype scope, the identifier declared is callable (any arguments having an integer type will be converted to the enumerated types compatible integer type).

Other Languages

C (and C++) is unique in having this kind of scope. Other languages tend to simply state that the identifiers within the matching parentheses must be unique and do not allow new types to be defined between them, removing the need to create a scope to make use of the associated functionality to correctly process the parameter declarations.

Coding Guidelines

Tags, enumeration constants and parameter identifiers declared in function prototype scope are not visible outside of that scope and there are no mechanisms available for making them visible. Such usage serves no useful purpose since function prototype scope hides them completely. Anonymous structure and union types declared in function prototype scope do nothing but render the declared function unreferenceable.

prototype providing identifiers

The identifiers appearing in a prototype essentially fill the same role as comments, i.e., they may provide a reader of the source with useful information. The only reason that identifiers appear in such declarations may be because the author cut-and-pasted from the function definition. The presence of identifiers slightly increases the probability that a match against a defined macro will occur, but the consequences are either harmless or a syntax error. There does not appear to be a worthwhile cost/benefit in a guideline recommendation for or against this usage.

Example

```
/*
 * Declaring a new type in a function prototype scope renders that
 * function uncallable.  No other function can be compatible with it.
 */
extern void uncallable(struct s_tag{int m1;} p);

extern void call_if_compatible_int(enum {E1, E2} p);
```

410 If an identifier designates two different entities in the same name space, the scopes might overlap.

scope overlapping

Commentary

Or they will be disjoint, or the translation unit contains a constraint violation.

Note that if a tag denoting an incomplete type is visible, then another declaration of that identifier in a different scope in the tag name space introduces a new type; it does not complete the previously visible incomplete type.

tag declarations different scope

C90

This sentence does not appear in the C90 Standard, but the situation it describes could have occurred in C90.

C++

The scope of a declaration is the same as its potential scope unless the potential scope contains another declaration of the same name. 3.3p1

Other Languages

Languages that support at least two forms of scope (file and block) invariably allow identifiers declared in them to have overlapping scopes.

Coding Guidelines

Why would a developer want to use the same name to designate different entities in the same name space with overlapping scopes? Such usage can occur accidentally; for instance, when a block scope identifier has the same name as a file scope identifier that is beyond the developer's control (e.g., in a system header). Having the same identifier designate different entities, irrespective of name space and scope, is likely to be a potential cause of confusion to developers.

When two identifiers in the same name space have overlapping scopes, it is possible for a small change to the source to result in a completely unexpected change in behavior. For instance, if the identifier definition in the inner scope is deleted, all references that previously referred to that, inner scoped, identifier will now refer to the identifier in the outer scope. Deleting the definition of an identifier is usually intended to be accompanied by deletions of all references to it; the omission of a deletion usually generates a diagnostic when the source is translated (because of a type mismatch). However, if there is an alternative definition, with a compatible type, for an access to refer to, translators are unlikely to issue a diagnostic.

Dev ?? Identifiers denoting objects, functions, or types need only be compared against other identifiers having, or returning, a scalar type.

Example

```
extern int total_valu;
extern struct T {
                 int m1;
                } data_fields;

void f(void)
{
float total_valu = 0.0;
int data_fields = 0;
}
```

```
void g(void)
{
int X;

   {
   int Y = X; /* Scope of second X not yet opened. */
   int X;
   }
}
```

Once an outer identifier has been hidden by the declaration in an inner scope, it is not possible to refer, by name, to the outer object.

---

scope inner
scope outer

If so, the scope of one entity (the *inner scope*) will be a strict subset of the scope of the other entity (the *outer scope*). 411

Commentary

This defines the terms *inner scope* and *outer scope*. The term *nested scope* is also commonly used to describe a scope that exists within another scope (often a block scope nested inside another block scope).

file scope accessing hidden names

This C statement only describes the behavior of identifiers. It is possible to refer to objects in disjoint scopes by using pointers to them. It is also possible, through use of linkage, for an identifier denoting an object at file scope to be visible within a nested block scope even though there is another declaration of the same identifier in an intervening scope.

```
extern int glob;

void f1(void)
{
int glob = 0;

   {
   extern int glob; /* Refers to same object as file scope glob. */

   glob++;
   }
/* Visible glob still has value 0 here. */
}
```

C++

The C observation can be inferred from the C++ wording.

3.3p1 *In that case, the potential scope of the declaration in the inner (contained) declarative region is excluded from the scope of the declaration in the outer (containing) declarative region.*

Other Languages

This terminology is common to most programming languages.

Within the inner scope, the identifier designates the entity declared in the inner scope; 412

Commentary

identifier scope begins 418

The scopes of identifiers, in C, do not begin at the opening curly brace of a compound block (for block scope). It is possible for an identifier to denote different entities within the same compound block.

```
typedef int I;

void f(void)
```

```
{
I I; /* The inner scope begins at the second I. */
}
```

Coding Guidelines

Developers are unlikely to spend much time thinking about where the scope of an identifier starts. A cost effective simplification is to think of an identifier's scope being the complete block that contains its declaration. One of the rationales for the guideline recommendation dealing with reusing identifier names is to allow developers to continue to use this simplification.

?? identifier reusing names

---

413 the entity declared in the outer scope is *hidden* (and not visible) within the inner scope.

outer scope identifier hidden

Commentary

This defines the term *hidden*. The entity is hidden in the sense that the identifier is not visible. If the entity is an object, its storage location will still be accessible through any previous assignments of its address to an object of the appropriate pointer type. If such identifiers are file scope objects, they may also be accessible via, developer-written, functions that access them (possibly returning, or modifying, their value).

There is only one mechanism for directly accessing the outer identifier via its name, and that only applies to objects having file scope.

411 file scope accessing hidden names

C++

The C rules are a subset of those for C++ (3.3p1), which include other constructs. For instance, the scope resolution operator, `::`, allows a file scope identifier to be accessed, but it does not introduce that identifier into the current scope.

Other Languages

The hiding of identifiers by inner, nested declarations is common to all block-structured languages. Some languages, such as Lisp, base the terms *inner scope* and *outer scope* on a time-of-creation basis rather than lexically textual occurrence in the source code.

Coding Guidelines

The discussion of the impact of scope on identifier spellings is applicable here.

scope naming conventions

Example

It is still possible to access an outer scope identifier, denoting an object, via a pointer to it:

```
void f(void)
{
int i,
    *pi = &i;

    {
    int i = 3;

    i += *pi; /* Assign the value of the outer object i, to the inner object i. */
    }
}
```

Usage

In the translated form of this book's benchmark programs there were 1,945 identifier definitions (out of 270,394 identifiers defined in block scope) where an identifier declared in an inner scope hid an identifier declared in an outer block scope.

---

414 Unless explicitly stated otherwise, where this International Standard uses the term "identifier" to refer to some entity (as opposed to the syntactic construct), it refers to the entity in the relevant name space whose declaration is visible at the point the identifier occurs.

Commentary

prior declaration not

The C Standard is a definition of a computer language in a stylized form of English. It is not intended as a tutorial. As such, it is brief and to the point. This wording ensures that uses of the term *identifier* are not misconstrued. For instance, later wording in the standard should not treat the declaration of glob as being a prior declaration in the context of what is being discussed.

```
struct glob {
            int mem;
            };

extern int glob;
```

C90

There is no such statement in the C90 Standard.

C++

There is no such statement in the C++ Standard (which does contain uses of *identifier* that refer to its syntactic form).

Coding Guidelines

Coding guidelines are likely to be read by inexperienced developers. Particular guidelines may be read in isolation, relative to other guidelines. Being brief is not always a positive attribute for them to have. A few additional words clarifying the status of the identifier referred to can help confirm the intent and reduce possible, unintended ambiguities.

---

scope same

Two identifiers have the *same scope* if and only if their scopes terminate at the same point. 415

Commentary

incomplete type completed by

block iteration statement
block loop body
block selection statement
block selection substatement

This defines the term *same scope* (it is used in the description of incomplete types). The scope of every identifier, except labels, starts at a different point than the scope of any other identifier in the source file. Scopes end at well-defined boundaries; the closing **}** of a block, the closing **)** of a prototype, or on reaching the end of the source file, the end of an *iteration-statement*, loop body, the end of a *selection-statement*, or the end of a substatement associated with a selection statement.

C90

block iteration statement
block loop body
block selection statement
block selection substatement

Although the wording of this sentence is the same in C90 and C99, there are more blocks available to have their scopes terminated in C99. The issues caused by this difference are discussed in the relevant sentences for iteration-statement, loop body, a *selection-statement* a substatement associated with a selection statement.

C++

The C++ Standard uses the term *same scope* (in the sense "in the same scope", but does not provide a definition for it. Possible interpretations include using the common English usage of the word *same* or interpreting the following wording

3.3p1 *In general, each particular name is valid only within some possibly discontiguous portion of program text called its scope. To determine the scope of a declaration, it is sometimes convenient to refer to the potential scope of a declaration. The scope of a declaration is the same as its potential scope unless the potential scope contains another declaration of the same name. In that case, the potential scope of the declaration in the inner (contained) declarative region is excluded from the scope of the declaration in the outer (containing) declarative region.*

to imply that the scope of the first declaration of a is not the same as the scope of b in the following:

```
{
int a;
int b;
   {
   int a;
   }
}
```

Other Languages

A few languages define the concept of same scope, often based on where identifiers are declared not where their scope ends.

Example

In the following example the identifiers X and Y have the same scope; the identifier Z is in a different scope.

```
void f(void)
{
int X;
int Y;

   {
   int X; /* A different object named X. */
   int Z;

   } /* Scope of X and Z ended at the } */
}    /* Scope of X and Y ended at the } */

void g(void)
{
for (int index = 0; index < 10; index++)
   {
   int index;
   /* Scope of second index ends here. */ }
/* Scope of first one ends here.    */ }
```

---

416 Structure, union, and enumeration tags have scope that begins just after the appearance of the tag in a type specifier that declares the tag.

tag scope begins

Commentary

The scope of some identifiers does not start until their declarator is complete. Applying such a rule to tags would prevent self-referencing structure and union types (i.e., from having members that pointed at the type currently being defined).

418 identifier scope begins

```
struct S1_TAG {
              struct S1_TAG *next;
              };
```

However, just because a tag is in scope and visible does not mean it can be referenced in all of the contexts that a tag can appear in. A tag name may be visible, but denoting what is known as an *incomplete type*.

incom-plete type completed by

```
struct S2_TAG {
              char mem1[sizeof(struct S2_TAG)]; /* Constraint violation. */
              };

/*
 * Here E_TAG becomes visible once the opening { is reached.
 * However, the decision on the size of the integer type chosen may
 * require information on all of the enumeration constant values and
 * its type is incomplete until the closing }.
 */
enum E_TAG {E1 = sizeof(enum E_TAG)};
```

C++

The C++ Standard defines the term *point of declaration* (3.3.1p1). The C++ point of declaration of the identifier that C refers to as a tag is the same (3.3.1p5). The scope of this identifier starts at the same place in C and C++ (3.3.2p1, 3.3.5p3).

Coding Guidelines

incomplete type completed by

The scope of a tag is important because of how it relates to completing a previous incomplete structure or union type declaration. This issue is discussed in more detail elsewhere.

---

enumeration constant scope begins

Each enumeration constant has scope that begins just after the appearance of its defining enumerator in an enumerator list. 417

Commentary

enumeration specifier syntax

The definition of an enumerator constant can include a constant expression that specifies its value. The scope of the enumeration constant begins when the following comma or closing brace is encountered. This enables subsequent enumerators to refer to ones previously defined in the same enumerated type definition.

C++

3.3.1p3 *The point of declaration for an enumerator is immediately after its* `enumerator-definition`*. [Example:*

```
const int x = 12;

{ enum { x = x }; }
```

Here, the enumerator `x` *is initialized with the value of the constant* `x`*, namely 12.]*

In C, the first declaration of `x` is not a constant expression. Replacing it by a definition of an enumeration of the same name would have an equivalent, conforming effect in C.

Coding Guidelines

identifier reusing names ??

The example below illustrates an extreme case of the confusion that can result from reusing identifier names. The guideline recommendation dealing with reusing identifier names is applicable here.

Example

```
enum {E = 99};

void f(void)
{
enum { E = E + 1  /* Original E still in scope here. */
                , /* After comma, new E in scope here, with value 100. */
       F = E};    /* Value 100 assigned to F. */
}
```

---

identifier scope begins

Any other identifier has scope that begins just after the completion of its declarator. 418

Commentary

The other identifiers are objects, typedefs, and functions. For objects' definitions, the declarator does not include any initializer that may be present. A declarator may begin the scope of an identifier, but subsequent declarators in the same scope for the same identifier may also appear (and sometimes be necessary) in some cases.

declarator syntax

incom-plete type completed by

C++

The C++ Standard defines the potential scope of an identifier having either local (3.3.2p1) or global (3.3.5p3) scope to begin at its *point of declaration* (3.3.1p1). However, there is no such specification for identifiers having function prototype scope (which means that in the following declaration the second occurrence of p1 might not be considered to be in scope).

```
void f(int p1, int p2[sizeof(p1)]);
```

No difference is flagged here because it is not thought likely that C++ implementation will behave different from C implementations in this case.

Other Languages

Some languages define the scope of an identifier to start at the beginning of the block containing its declaration. This design choice can make it difficult for a translator to operate in a single pass. An identifier from an outer scope may be referenced in the declaration of an object; a subsequent declaration in the same block of an identifier with the same name as the referenced outer scope one invalidates the reference to the previous object declaration and requiring the language translator to change the associated reference.

In some cases the scope of an identifier in Java can extend to before its point of declaration.

408 block scope terminates

Coding Guidelines

Cases such as the one given in the next example, where two entities share the same name and are visible in different portions of the same block are covered by the guideline recommendation dealing with reusing identifier names.

?? identifier reusing names

Example

```
void f(void)
{
enum {c, b, a};
typedef int I;

   {
   I I; /*
         * The identifier object does not become visible until the ;
         * is reached, by which time the typedefed I has done its job.
         */

   int a[a]; /*
              * Declarator completed after the closing ], number of
              * elements refers to the enumeration constant a.
              */

   struct T {struct T *m;} x = /* declarator complete here. */
                               {&x};
   }
}
```

419 **Forward references:** declarations (6.7), function calls (6.5.2.2), function definitions (6.9.1), identifiers (6.4.2), name spaces of identifiers (6.2.3), macro replacement (6.10.3), source file inclusion (6.10.2), statements (6.8).

References

1. J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison–Wesley, 1996.