The New C Standard (Excerpted material)

An Economic and Cultural Commentary

Derek M. Jones
derek@knosof.co.uk

Copyright ©2002-2008 Derek M. Jones. All rights reserved.

5.2.4.1 Translation limits

translation limits

The implementation shall be able to translate and execute at least one program that contains at least one instance of every one of the following limits:[13)] 276

Commentary

This is a requirement on the implementation (a single preprocessing translation unit containing all of the constructs given here, to the limits specified). The topic of a perverse implementation, one that can successfully translate a single program containing all of these limits but no other program, crops up from time to time. Although of theoretical interest, this discussion is of little practical interest, because writing a translator that only handled a single program would probably require more effort than writing one that handled programs in general.

The values for these limits were not obtained by measuring how often each construct appeared within existing source code. There is no claim that a program containing an instance of all such constructs is in any way representative of a typical program.

Rationale

Some of the limits chosen represent interesting compromises. The goal was to allow reasonably large portable programs to be written, without placing excessive burdens on reasonably small implementations, some of which might run on machines with only 64 K of memory. In C99, the minimum amount of memory for the target machine was raised to 512 K. In addition, the Committee recognized that smaller machines rarely serve as a host for a C compiler: programs for embedded systems or small machines are almost always developed using a cross compiler running on a personal computer or workstation. This allows for a great increase in some of the translation limits.

A program containing an instance of all such limits is one of the tests included in the commercially available C validation suites that used to be used by NIST and BSI.

C++

Annex Bp2

However, these quantities are only guidelines and do not determine conformance.

This wording appears in an informative annex, which itself has no formal status.

Other Languages

Many language definitions specify some minimum value for some constructs that implementations are required to support. The Modula-2 Standard contains what it called *limit-specification generators*. These are a set of Modula-2 programs, which when executed generate a set of Modula-2 programs that an implementation must be capable of translating and executing.

Common Implementations

Some implementations provide a translator option that allows the developer to control the amount of storage allocated to various internal data structures; for instance, the option `-xs1234` might specify a symbol table capable of holding 1,234 symbols. For large programs it can take several attempts before the various options are tuned (enabling the source to be translated within the available storage). Such implementation options may still be provided today, as part of a backwards compatibility mode.

Coding Guidelines

Most of the limit values are sufficiently generous that few of them are likely to be exceeded. But within these coding guidelines, we are not just interested in translator limitations, we are also interested in developer limitations. There may be readability, comprehensibility, or complexity issues associated with multiple occurrences of some constructs. A program that contains an excessive number of any particular construct could be poorly structured or simply a large program.

In the case of nested constructs, it is often claimed that developers have problems remembering the information if the nesting is too deep. The fact that developers experience problems remembering information on nested constructs suggests they are using short-term memory to hold this information. The capacity limits of short-term memory are only one of the issues involved in comprehending nested constructs. How developers organize information presented to them (from the source code), knowledge held in their long-term memories (about how a program works or memories of previous code readings), and the extent to which information from different nesting levels is related all need to be considered.

memory developer
coupling and cohesion

It is also important to consider the bigger picture of particular nested constructs. Are the alternatives any better? Some coding guideline documents specify a value for the maximum nesting level of particular constructs[4,5] (sometimes giving a rationale based on the famous 7±2 paper). These coding guidelines resist the attractions of providing a single, easy-to-calculate, maximum nesting limit. The issues are discussed in more detail within each nested construct.

Miller 7±2

---

277 — 127 nesting levels of blocks

limit block nesting

Commentary

Blocks are created by a number of different kinds of statements. The one most commonly thought of is a compound statement. There is no dependency on the kinds of statement that cause a block to be created. There could be any combination of **`if/while/for/switch`** or simply a compound statement with no associated header.

block
compound statement syntax

This limit might be reached in automatically generated code, but it would be considered an extreme case.

C90

15 nesting levels of compound statements, iteration control structures, and selection control structures

The number of constructs that could create a block increased between C90 and C99, including selection statements and their associated substatements, and iteration statements and their associated bodies. Although use of these constructs doubles the number of blocks created in C99, the limit on the nesting of blocks has increased by a factor of four. So, the conformance status of a program will not be adversely affected.

block selection statement
block selection sub-statement
block iteration statement
block loop body

C++

The following is a non-normative specification.

Nesting levels of compound statements, iteration control structures, and selection control structures [256]

Annex Bp2

Common Implementations

Nesting of blocks is part of the language syntax and is usually implemented with a table-driven syntax analyzer. Table-driven syntax analyzers maintain their own stack, often a predefined fixed size, of information. A very large number of nested blocks is likely to cause this parser table to overflow.

Coding Guidelines

In human-written code a significantly lower limit on the nesting of blocks is often recommended. Working purely on the basis of some form of line indentation, for every new block opened, more than five nested levels would lead to a visually difficult to follow, on a display device, source file. Blocks opened and closed within a macro definition would not affect the visual appearance of source, at the point of macro invocation. This kind of nesting would not be counted in the five recommendation.

---

278 — 63 nesting levels of conditional inclusion

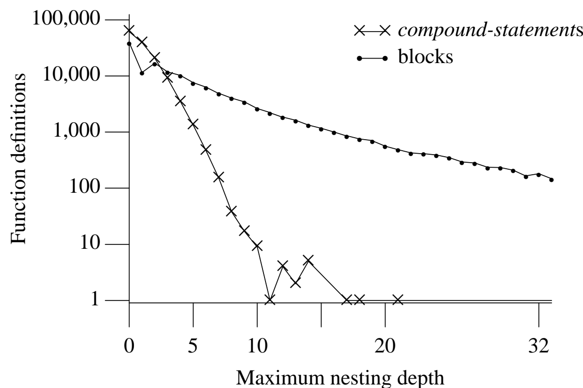

Figure 277.1: Number of functions containing blocks and `compound-statement`s nested to the given maximum nesting level. Based on the visible form of the `.c` files.

Commentary

conditional inclusion

Conditional inclusion is performed as part of preprocessing. As such, it is independent of the syntax processing performed by subsequent translation phases and is given its own limit.

The value of this limit is consistent with other limit values. It is something of a fortunate coincidence, because the same ratios applied in C90, where the following rationale did not apply. The value is half the limit value for nesting of blocks. This difference occurs because the C **`if`** statement is defined to create two blocks. Nesting **`if`** statements 64 deep would be sufficient to exceed the block limit, and 64 nested **`#if`** directives would exceed the above limit.

limit block nesting 277
block selection statement

C90

> *8 nesting levels of conditional inclusion*

C++

The following is a non-normative specification.

Annex Bp2

> *Nesting levels of conditional inclusion [256]*

Common Implementations

Several different techniques are used to write preprocessors. Given the relatively simple C preprocessor syntax, many have a *flat* view of the nesting of these constructs. They simply maintain a count of the current nesting level. A full parser/syntax-based approach faces the same type of problems found in handling the C syntax limits discussed here. But, such an approach is rarely seen in C preprocessor implementations.

This limit may be reached in automatically generated code.

Coding Guidelines

selection statement syntax

Conditional inclusion differs from selection statements in that it is not possible to prevent deep nesting by moving directives to separate functions (although they could be moved to a separate source file and accessed via a **`#include`** directive, or the design of the configuration control implied by the nested directives could be changed).

The human factors issues might be thought to be the same as those for the nesting of selection statements. However, developers generally do not visually indent nested conditional inclusion directives (see Figure **??**) a practice that is commonly used for selection (and other) statements.

Figure 278.1: Number of translation units containing conditional inclusion directives nested to the given maximum nesting level. Based on the visible form of the `.c` and `.h` files.

---

279 — 12 pointer, array, and function declarators (in any combinations) modifying an arithmetic, structure, union, or incomplete type in a declaration

limit
type complexity

Commentary

This limit is based on a particular encoding of type information within 32 bits, which was used in some early translators. Eight bits were used to encode the base type and up to 12 lots of 2 bits representing pointer-to, array-of, or function-of. Another consideration was that Fortran originally supported a maximum of seven subscripts in an array declaration. The committee recognized that Fortran to C translations needed to be able to support this number of nested array declarations.

Wording that appears elsewhere specifies that types defined via typedef names need to be included in the count.

279 limit type complexity

C++

The following is a non-normative specification.

> *Pointer, array, and function declarators (in any combinations) modifying an arithmetic, structure, union, or incomplete type in a declaration [256]*

Annex Bp2

Common Implementations

Some implementations continue to use the K&R technique. Many others use a dynamic data structure relevant to the type being defined and have no internal limits on the complexity supported.

Coding Guidelines

Data structures need to mimic the application domain being addressed. If a deep nesting of pointers, arrays, or function declarators is called for, there may be little benefit in arbitrarily splitting the declaration into smaller components, unless these subcomponents have semantic meaning within the application domain.

Example

```
1 char ************p_12;
2 short a_12[1][1][1][1][1][1][1][1][1][1][12];
3 void (* (* (* (* (* (* (* (* (* (* (* (*p)
4    (void))(void))(void))(void))(void))(void))(void))(void))(void))(void))(void))(void);
```

---

280 — 63 nesting levels of parenthesized declarators within a full declarator

limit
declarator
parentheses

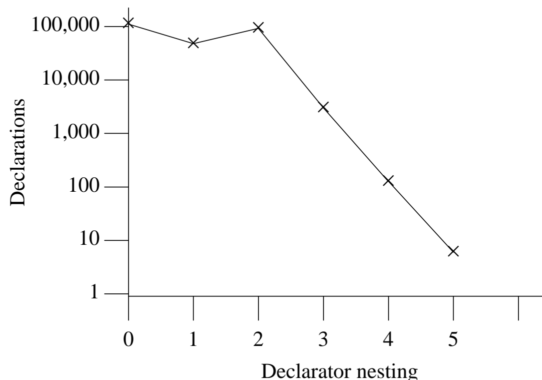

Figure 279.1: Number of full declarators containing a given number of modifiers. Based on the translated form of this book's benchmark programs.

Commentary

full declarator

The limit of 12 modifiers on a declaration is likely to be reached before this limit of 63 is reached on a full declarator (unless redundant `( )` are used, or some very rarely seen structure declarations). This limit is unlikely to be reached, even in automatically generated code.

```
1  struct {
2          (struct {
3                  (struct {
4                          ...
5                          } *p)[2];
6                  } *q)[1];
7          };
```

C90

31 nesting levels of parenthesized declarators within a full declarator

C++

The C++ Standard does not discuss declarator parentheses nesting limits.

Common Implementations

limit block nesting 277

Nesting of parentheses is part of the language syntax and is usually implemented with a table-driven syntax analyzer. The same implementation details as nesting of blocks applies. The nesting of parentheses can occur within a nested block, slightly increasing the chances of reaching an internal implementation limit.

---

parenthesized expression nesting levels

— 63 nesting levels of parenthesized expressions within a full expression 281

Commentary

While it is possible to keep within this limit in an expression containing one instance of every operator (C contains 47 unique operators), an expression containing more than one instance of two operators may need to exceed this limit— for instance, `(((((a0/x+a1)/x+a2)/x+a3)/x+a4)/x+a5)/x+....`

This limit is rarely reached except in automatically generated code. Even then it is rare.

C90

31 nesting levels of parenthesized expressions within a full expression

C++

The following is a non-normative specification.

Nesting levels of parenthesized expressions within a full expression [256]

Annex Bp2

Common Implementations

The same implementation details as nesting of declarators applies, with the difference that nesting of expressions is more common and likely to be deeper.

280 limit declarator parentheses

Coding Guidelines

Although some uses of parentheses may be technically redundant, they may be used to simplify the visual appearance of an expression, or to divide an expression into meaningful chunks. While minimizing the number of parentheses in an expression may be an interesting mathematical problem, minimization is not a desirable goal when writing source code. The top priority when considering the use of parentheses should always be comprehensibility of the resulting expression.

memory chunking

```
1   (((((a0 * x + a1) * x + a2) * x + a3) * x + a4) * x + a5) * x + a6
2
3   a * (b + (c << (d > (e == (f & (g ^ (h | (i && (j || (k_1 ? k_2 : (L = (m , n )))))))))));
```

The issue of expression complexity is discussed elsewhere.

expressions

---

282 — 63 significant initial characters in an internal identifier or a macro name (each universal character name or extended source character is considered a single character)

internal identifier significant characters

Commentary

An internal identifier is one whose name is never visible outside of the source file in which it is declared. Because it is not necessary to worry about external representation issues, it is possible to count one UCN as one character.

This limit may be reached in automatically generated code.

This minimum limit may be increased in a future revision of the standard.

significant characters future language directions

C90

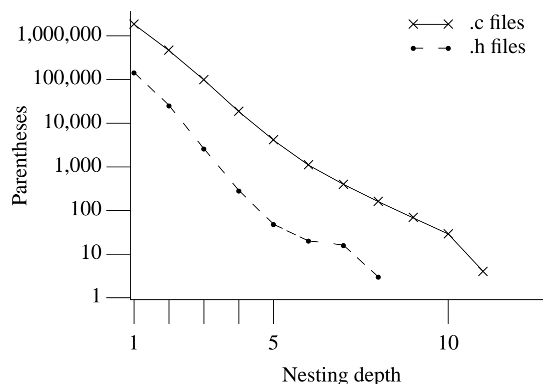

Figure 281.1: Nesting of all occurrences of parentheses. Based on the visible form of the `.c` and `.h` files.

31 significant initial characters in an internal identifier or a macro name

C++

2.10p1 *All characters are significant.*[20)]

C identifiers that differ after the last significant character will cause a diagnostic to be generated by a C++ translator.

The following is a non-normative specification.

Annex Bp2 *Number of initial characters in an internal identifier or a macro name [1024]*

Other Languages

Some languages are silent on the number of significant characters in an internal identifier; others specify the same limit as external identifiers.

Common Implementations

This is one area where translators are likely to use a fixed-size data structure (usually an array). Using a linked list of characters to represent an identifier name would be a significant overhead. Having a fixed-size data structure that grows once the available free space is filled is an alternative used by some implementations.

Coding Guidelines

identifier number of characters

The issue of identifier length is discussed elsewhere.

Usage

Very few identifiers approach the C99 translation limit (see Figure **??**).

---

external identifier significant characters

— 31 significant initial characters in an external identifier (each universal character name specifying a short identifier of 0000FFFF or less is considered 6 characters, each universal character name specifying a short identifier of 00010000 or more is considered 10 characters, and each extended source character is considered the same number of characters as the corresponding universal character name, if any)[14)] 283

Commentary

program image

Information on externally visible identifiers needs to be stored in the files (usually object files) created by a translator. This information is compared against identifiers declared in other translation units when linking to build a program image. The predefined format of such files (not always within the control of the translator writer) may have limitations on what characters are acceptable in an identifier.

The values of 6 and 10 were chosen so that the encodings *u1234* and *U12345678* could be used.

C90

6 significant initial characters in an external identifier

C++

2.10p1 *All characters are significant.*[20)]

C identifiers that differ after the last significant character will cause a diagnostic to be generated by a C++ translator.

The following is a non-Normative specification.

Annex Bp2

Number of initial characters in an external identifier [1024]

Other Languages

The Fortran significant character limit of six was followed by many suppliers of linkers for a long time. The need for longer identifiers to support name mangling in C++ ensured that most modern linkers support many more significant characters in an external identifier.

Common Implementations

Historically, the number of significant characters in an external identifier was driven by the behavior of the host vendor-supplied linker. Only since the success of MS-DOS have developers become used to translator vendors supplying their own linker. Previously, most linkers tended to be supplied by the hardware vendor.

The mainframe world tended to be driven by the requirements of Fortran, which had six significant characters in an internal or external identifier. In this environment it was not always possible to replace the system linker by one supporting more significant characters. The importance of the mainframe environment waned in the 1990s. In modern environments it is very often possible to obtain alternative linkers.

Coding Guidelines

The number of significant characters should not affect the choice of a meaningful name. One coding technique is to continue to use the original (meaningful name) and to use macros to map to a different external name.

```
1 #define comms_inport_1 E1234
2 #define comms_inport_2 E1235
```

This approach suffers from the problem that there are two names associated with every object, not a good state of affairs from a program maintenance point of view. So, care needs to be taken that the alternative macro-derived names are not used directly.

C90 had a six character limit. Such a limit is very low and is an ideal that only a few, ultra-portable programs should still aspire to. However, it is possible that some C90 translators never migrate to the C99 limit (it being uneconomical to upgrade them).

The issue of identifier length is discussed more fully elsewhere.

identifier number of characters

---

284 13) Implementations should avoid imposing fixed translation limits whenever possible.

footnote 13

Commentary

This is only a suggestion to implementors, not a requirement. It implies that implementations ought to use dynamically allocated data structures, rather than fixed-size ones.

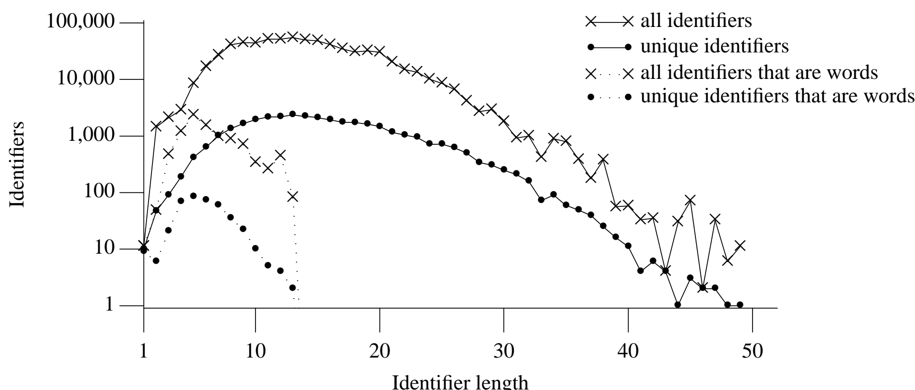

Figure 283.1: Number of identifiers, with external linkage, having a given length. Based on the translated form of this book's benchmark programs. Information on the length of all identifiers in the visible source is given elsewhere (see Figure **??**).

Common Implementations

Most implementations dynamically allocate data structures for most constructs. The developer's desire for translators that translate at reasonable rates means that there are trade-offs associated with the use of fixed-size data structures in some areas.

Coding Guidelines

The developer has no control over the design of an implementation. Although implementations do not go out of their way to make inefficient use of host resources, there is not always the commercial incentive, on some hosts, to improve the quality of a translator.

---

limit external identifiers

— 4095 external identifiers in one translation unit 285

Commentary

This limit may appear to be generous. But, it includes identifiers declared both by the developer and the implementation (when a system header is included). This limit may be reached in automatically generated code. The standard does not define a per program limit. This is mainly because some linkers are not provided by the translator vendor and are in many ways outside of these vendors' control.

C90

511 external identifiers in one translation unit

C++

The following is a non-normative specification.

Annex Bp2 *External identifiers in one translation unit [65536]*

Common Implementations

Most vendors include a large number of identifiers in their system headers. This is particularly true on workstations where the total number of identifiers declared in system headers can exceed 15,000 (see Table **??**). Developers have no control over the contents of these headers.

Usage

External declaration usage information is given elsewhere (see Figure **??**).

Table 285.1: Number of identifiers with external linkage (total 487), and total number of identifiers (total 810), implementations are required to declare in the standard headers.

Header	External Identifiers	Total Identifiers	Header	External Identifiers	Total Identifiers
<assert.h>	1	2	<signal.h>	2	12
<complex.h>	66	71	<stdarg.h>	3	5
<ctype.h>	15	15	<stdbool.h>	0	4
<errno.h>	1	4	<stddef.h>	0	5
<fenv.h>	11	24	<stdint.h>	0	38
<float.h>	0	31	<stdio.h>	49	65
<inttypes.h>	6	62	<stdlib.h>	36	37
<iso646.h>	0	11	<string.h>	22	24
<limits.h>	0	19	<tgmath.h>	0	60
<locale.h>	2	10	<time.h>	9	15
<math.h>	184	203	<wchar.h>	59	68
<setjmp.h>	2	3	<wctype.h>	18	22

---

identifiers number in block scope

— 511 identifiers with block scope declared in one block 286

Commentary

This limit may be reached in automatically generated code. In human-written code more than 10 identifiers declared in block scope is uncommon.

C90

127 identifiers with block scope declared in one block

C++

The following is a non-normative specification.

Identifiers with block scope declared in one block [1024]

Annex Bp2

Common Implementations

Most implementations take advantage of the scoping nature of blocks to create symbol table information when the declaration is encountered and to remove it (freeing up the storage used) when the block scope terminates. For implementations that operate in a single pass, generating machine code on a basic block basis, this can result in considerable storage savings. High-powered optimizing translators may still generate machine code in a single pass, but they usually build a tree representing all of the statements and expressions within each function. This means that information on block scope declarations cannot be freed up at the end of the block in which they occur.

Coding Guidelines

Having a large number of objects defined in the same block may be an indicator that a function definition has grown too large and needs to be split up, or an indicator that a structure type needs to be created. Although this is a design issue, there is a potential impact on comprehension effort. However, your author knows of no method of comparing the comprehension effort required for the various cases and so is silent on the subject.

Usage

The 53,630 function definitions in the translated form of this book's benchmark programs contained: definitions of 76 structure, union or enumeration types that included a tag; 6 **`typedef`** definitions; and definitions of 70 enumeration constants.

---

287 — 4095 macro identifiers simultaneously defined in one preprocessing translation unit

limit
macro definitions

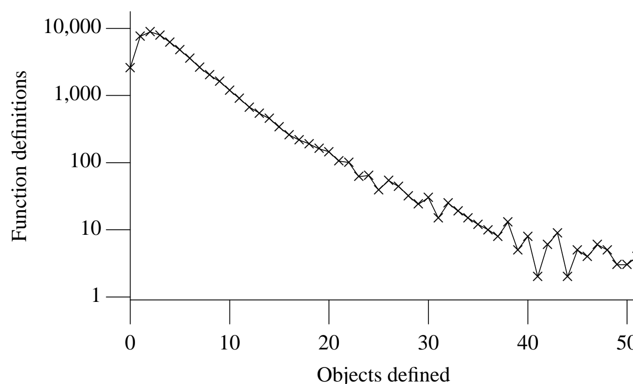

Figure 286.1: Number of function definitions containing a given number of definitions of identifiers as objects. Based on the translated form of this book's benchmark programs.

Commentary

This limit may appear to be generous. But, it includes macro identifiers declared both by the developer and the implementation (when a system header is included). The standard does not specify limits on the bodies of macro definitions. This is something that usually occupies much more storage than the identifier itself.

This limit may be reached in automatically generated code.

C90

1024 macro identifiers simultaneously defined in one translation unit

C++

The following is a non-normative specification.

Annex Bp2

Macro identifiers simultaneously defined in one translation unit [65536]

Common Implementations

footnote 3

Most vendors include a large number of identifiers in their system headers. This is particularly true on workstations where the total number of identifiers declared in system headers can exceed 15,000 (see Table **??**). Developers have no control over the contents of these headers. It would not be uncommon for the total number of macros in a translation unit to exceed this limit (assuming an appropriate number of system headers are included).

There are several public domain preprocessors that might be of use if this translator limit on number of macro identifiers is encountered. However, if the problem is caused by lack of storage on the host where the translation is performed, such a tool may not be of practical use. Using a different preprocessor, from the one provided as part of the implementation also introduces the problem of ensuring that any predefined, by one preprocessor, macro names are also defined with the same bodies when another preprocessor is used.

---

limit parameters in definition

— 127 parameters in one function definition **288**

Commentary

This limit is rarely reached except in automatically generated code, even then it is rare.

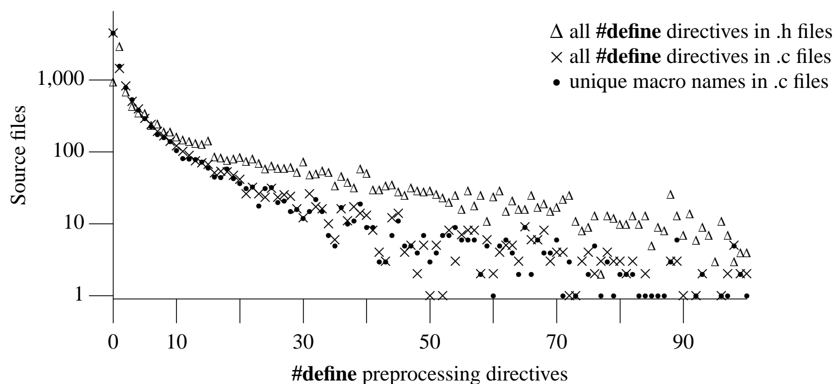

Figure 287.1: Number of source files containing a given number of identifiers defined as macro names in **`#define`** preprocessing directives. Unique macro name counts an identifier once, irrespective of the number of **`#define`** directives it appears in. Based on the visible form of the `.c` and `.h` files.

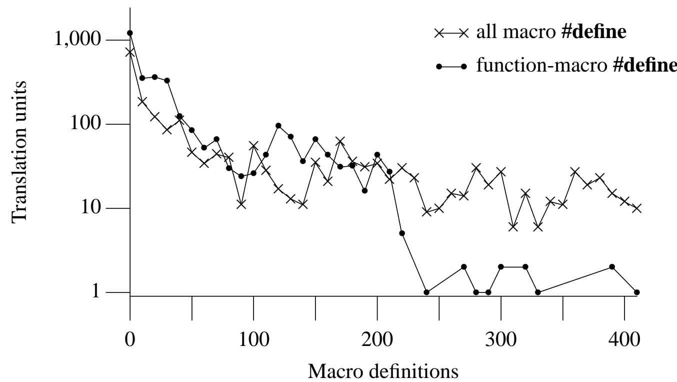

Figure 287.2: Number of translation units containing a given number of evaluations of `#define` preprocessing directives, excluding the contents of system headers, during translation of this book's benchmark programs (there were a total of 1,432,735 macros defined, of which 313,620 were function-like macros).

C90

> *31 parameters in one function definition*

C++

The following is a non-normative specification.

> *Parameters in one function definition [256]*

Annex Bp2

Common Implementations

Few hosted implementations place restrictions on the number of parameters in a function definition. Having one parameter on a stack is much the same as having 100. However, storage-limited execution environments (invariably freestanding) often limit the maximum number of parameters in a function definition.

The C binding for the GKS Standard[2] did manage to exceed the C90 limit, but this is uncommon.

Coding Guidelines

Some coding guideline documents recommend that use of file scope objects be minimized.[6] This has the consequence of increasing the number of parameters in function definitions. Other guideline documents recommend keeping the number of parameters below a certain limit to reduce the possibility of developers making mistakes (by passing arguments in the incorrect order). Possible alternatives include the following:

file scope

- *Relying on file scope objects.* Out of sight, out of mind— developers could easily forget to assign to these objects. Alternatively, once an object has file scope, any number of unexpected functions might also reference it, creating unintended dependencies.
- *Declaring a structure to hold the parameter values.* The arguments now need to be assigned to the members of the structure. The names of these members, if well chosen, could provide a useful reminder of the appropriate value to assign. The disadvantage is that there is no automatic checking when new parameters, in the form of new members, are added, potentially resulting in the new parameters being passed in existing invocations as uninitialized members.
- *Passing as much information as possible through parameters.*

There have been no empirically based studies whose results might be used as the basis for calculating which information-passing method has the optimal cost/benefit.

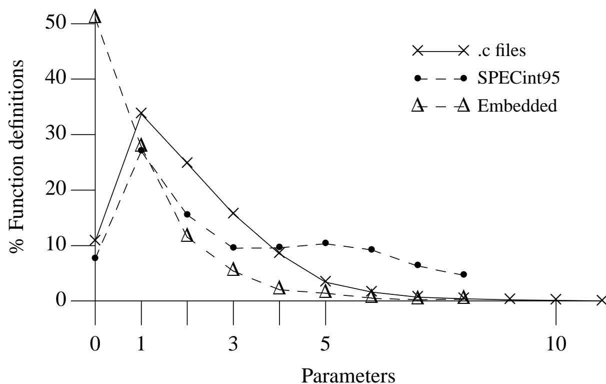

Figure 288.1: Percentage of function definitions appearing in the source of embedded applications (5,597 function definitions), the SPECint95 benchmark (2,713 function definitions), and the translated form of this book's benchmark programs (53,719 function definitions) declared to have a given number of parameters. The embedded and SPECint95 figures are from Engblom.[1]

---

function call number of arguments

— 127 arguments in one function call **289**

Commentary

limit parameters in definition 288

Functions declared using the ellipsis notation can be called with arguments that exceed this limit, while their definitions do not exceed the limit on the number of parameters.

This limit is rarely reached except in automatically generated code, even then it is rare.

C90

31 arguments in one function call

C++

The following is a non-normative specification.

Annex Bp2

Arguments in one function call [256]

Common Implementations

Few hosted implementations place restrictions on the number of arguments passed in one function call. However, storage-limited execution environments (invariably freestanding) sometimes have limits on the number of bytes available on the function call stack.

---

limit macro parameters

— 127 parameters in one macro definition **290**

Commentary

macro function-like

Function-like macro definitions are sometimes used to provide an alternative to an actual function call. These limits ensure that such definitions can handle at least as many parameters as function definitions.

C90

31 parameters in one macro definition

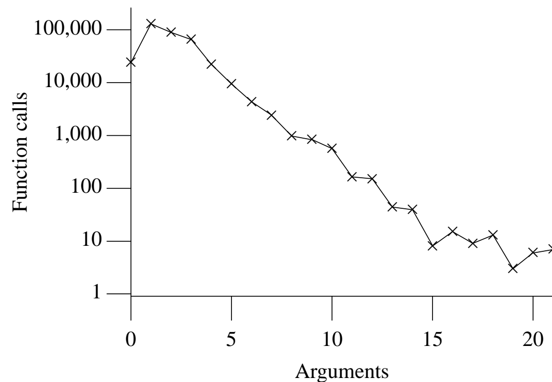

Figure 289.1: Number of function calls containing a given number of arguments. Based on the translated form of this book's benchmark programs.

C++

The following is a non-normative specification.

Parameters in one macro definition [256]

Annex Bp2

Common Implementations

A few implementations used fixed-size data structures for macro definitions. The extent to which these will be increased to support the new C99 limit is not known.

Coding Guidelines

In the case where the macro body is not syntactically a function body, a large number of parameters may be the most reliable method of ensuring that the intended objects are accessed. Because macro bodies are expanded at the point of reference, the objects visible at that point (not the point of definition) are accessed.

---

291 — 127 arguments in one macro invocation

limit arguments in macro invocation

Commentary

It is now possible, in C99, to define macros taking a variable number of arguments, using a similar principle to that used in function definitions. Although the arguments corresponding to the `...` notation are treated as a single parameter, inside the body of the macro definition, not individual ones.

... arguments macro

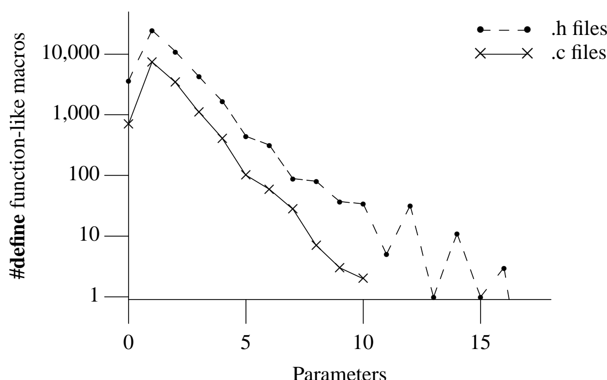

Figure 290.1: Number of function-like macro definitions having a given number of parameter declarations. Based on the visible form of the `.c` and `.h` files.

C90

31 arguments in one macro invocation

C++

The following is a non-normative specification.

Annex Bp2

Arguments in one macro invocation [256]

Common Implementations

Some implementations limit the size (e.g., the number of characters) of an argument (an early version of Microsoft C[3] had a 256-character limit).

---

limit characters on line

— 4095 characters in a logical source line **292**

Commentary

translation phase 2

A logical line is created from a physical line after any line splicing has taken place in translation phase 2. Line splicing is only really needed in macro definitions. This limit can really be thought of as applying to the number of characters in a macro definition.

macro replacement

Note that this limit does not apply to the result of any macro expansion. The C Standard defines a token-based preprocessor; characters and line length need not enter into the macro expansion process.

This limit may, rarely, be reached in automatically generated code.

C90

509 characters in a logical source line

C++

The following is a non-normative specification.

Annex Bp2

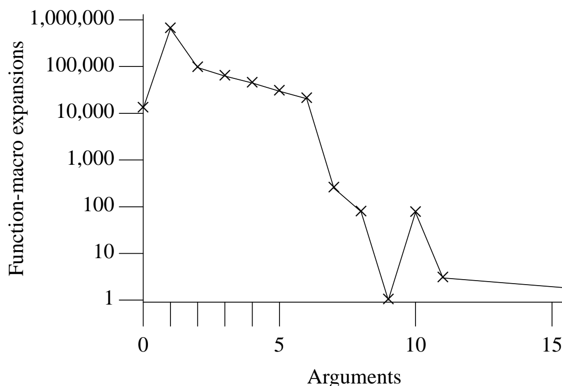

Figure 291.1: Number of function-like macro expansions containing a given number of arguments, excluding expansions that occurred while processing system headers, during translation of this book's benchmark programs.

Characters in a logical source line [65536]

Other Languages

Fortran (prior to Fortran 90) had a limit of 80 characters— the width of an IBM punched card.

Common Implementations

Some implementations use a fixed-length buffer for handling single logical source lines. Others use a fixed-length buffer and handle those situations where the length of a logical line exceeds the length of that buffer as a special case. The environmental limit on the minimum number of characters that may be supported on a physical line may affect translators written in C.

---

293 — 4095 characters in a character string literal or wide string literal (after concatenation)

limit string literal

Commentary

translation phase 6
292 limit characters on line

This limit applies after translation phase 6. If the limit on the number of characters in a logical line is taken into account then, allowing for the delimiting quote characters, the only way of reaching or exceeding this limit without exceeding any other limits is via concatenation. Longer strings, than this limit, can be created by copying character values into object storage. But, these would not be string literals.

C90

509 characters in a character string literal or wide string literal (after concatenation)

C++

The following is a non-normative specification.

Annex Bp2

Characters in a character string literal or wide string literal (after concatenation) [65536]

Common Implementations

Some implementations use a fixed-length buffer to hold the characters making up a preprocessing token (in this case a string literal). The characters forming the string literal are rarely held on a linked list. Other implementations use a string-handling package to look after the details of manipulating variable-length string literals and have no internal length restrictions.

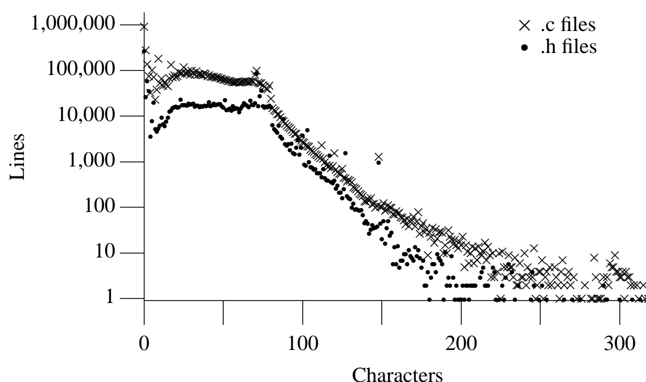

Figure 292.1: Number of physical lines containing a given number of characters. Based on the visible form of the `.c` and `.h` files.

Coding Guidelines

If a very long string literal is needed by the application, it makes sense to try to create it as a single entity. String literals containing more than a few hundred characters are rare enough not to be worth a coding guideline.

---

limit minimum object size

— 65535 bytes in an object (in a hosted environment only) 294

Commentary

Many CISC processors have an efficient 16-bit addressing mode to access objects. It is often possible to create and access objects that exceed this addressing range, but the implementation and execution time overhead can be much higher. In many ways this limit can be seen as giving permission for implementations to stay within the natural addressing structure of their target processor (should it be a 16-bit one).

The standard does not say anything about the storage duration of objects of this size; does it apply to all of them or at least one of them? There is no specification requiring that it be possible to define more than one of these objects, or for several smaller objects whose total size is 65,535 bytes to be supported.

This limit matches the corresponding minimum limits for `size_t` and `ptrdiff_t`. This limit means that for a translator where the type **`int`** is represented in 16 bits, the typedefs `size_t` and `ptrdiff_t` must have ranks at least equal to the type **`long`**.

Many freestanding environments don't even have 64 K bytes of memory in total. However, the standard does not specify a minimum object size that must be supported in these environments.

DR #266

Committee Discussion

Translation limits do not apply to objects whose size is determined at runtime.

C90

32767 bytes in an object (in a hosted environment only)

C++

The following is a non-normative specification.

Annex Bp2

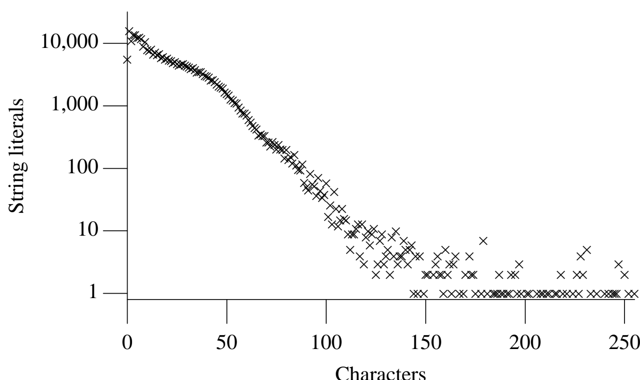

Figure 293.1: Number of character string literals containing a given number of characters (i.e., their length). Based on the visible form of the `.c` files.

Size of an object [262144]

Common Implementations

The maximum size of an object that can be defined may depend on its storage duration. On the basis that most function definitions only use a small amount of local data, some processor designers choose to ignore the relatively rare case of large amounts of local storage being required. The addressing modes needed to access large local objects, with a few instructions, may not be available (instead, relatively long sequences of instructions need to be used). These processor-based limitations can lead to translator vendors deciding not to go to the trouble of supporting very large objects having automatic storage duration.

If support for objects larger than 64 K is needed, it is most likely to be available via allocated storage. However, in some cases there may be limitations caused by host environment restrictions on the amount of storage that can be dynamically allocated in one contiguous storage area.

Linkers sometimes place restrictions on the maximum size of an object that can be statically allocated. This will affect objects with static storage duration.

Coding Guidelines

Although the standard may require that it be possible to define an object of the specified size, it is silent on the circumstances in which a program containing such a definition must be capable of executing. The standard does not provide any mechanism for verifying that a particular function invocation will successfully start to execute (i.e., not generate a stack overflow). In the case of static storage allocation, the program will either start to execute, or fail to start executing.

Use of dynamic allocation for objects does provide a degree of developer control of the situation where the allocation request fails. The disadvantage of such allocation methods is that it puts more responsibility for getting things right onto the developers' shoulders. Handling execution environment object storage limitations is a design and algorithmic issue that is outside the scope of these coding guidelines.

---

295 — 15 nesting levels for **#include**d files

limit
#include nesting

Commentary

This limit makes no distinction between system headers and developer-written header files. However, an implementation is required to support its own system headers whose contents are defined by the standard. If a particular implementation chooses to use nested **#include**s, then it is responsible for ensuring that these do not prevent a translator from meeting its obligations with regard to this limit.

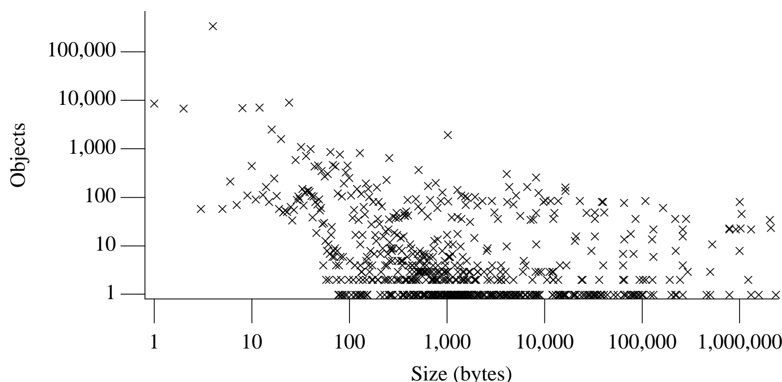

Figure 294.1: Number of objects requiring the specified amount of storage. Based on the translated form of this book's benchmark programs, using integer types whose sizes were: `sizeof(short) == 2`, `sizeof(int) == 4`, and `sizeof(long) == 4`; and alignment requirements that were a multiple of a types size.

Use of nested **#include**s requires that the source file containing each of the **#include** directive be kept open, while the included file is processed. Supporting 15 nesting levels invariably requires keeping at least 17 (the top-level source file and the file holding the generated code also need to be counted) files open simultaneously. Using the `fsetpos` library function to record the current file position, at the point the **#include** occurs, and closing the current file before opening the nested include is possible; however, your author has never heard of an implementation that uses it.

C90

8 nesting levels for #included files

C++

The following is a non-normative specification.

Annex Bp2 *Nesting levels for #included files [256]*

Common Implementations

This limit is not just about data structures within the translator. Most environments have limits on the number of files that a process can have open simultaneously. This limit may be soft in the sense that the developer can modify it, or hard in that the limit is built into the OS (requiring a kernel rebuild to change it).

Coding Guidelines

Developers have no control over the nesting used by system headers. These headers are essentially black boxes, so it is permissible to ignore any nesting that occurs within them, from the point of view of calculating the maximum **#include** nesting level.

Does the depth of nesting of **#include** files affect the cost of ownership of source code?

Headers that only contain information specific to a particular application area, or even data type, would seem to be following the principles of information-hiding. The nesting of headers might be mapped to the corresponding nesting of application data structures. A practical problem associated with information-hiding in headers is locating the header that contains a particular identifier declaration. Program development environments rarely provide tools for handling headers in a structured fashion.

In a traditional development environment the source code editor does not usually have any knowledge of the character sequences it is displaying, although some editors do support a tags facility (enabling a database of identifier tags and the files that reference them to be built). Syntax highlighting is also becoming common and C++ style class browsers are growing in popularity, but structured support for displaying header file contents is still uncommon.

Given the support tools that are likely to be available to a developer, a limit on the nesting of **#include** directives could provide a benefit by reducing developer effort when browsing the source of various header files. However, limiting the nesting to which **#include** directives can occur could increase configuration-management costs, or result in poorly structured source files. The cost/benefit issues are complex and these coding guidelines say nothing more on the issue.

---

limit case labels

— 1023 **case** labels for a **switch** statement (excluding those for any nested **switch** statements) 296

Commentary

The limit chosen in C90 had a rationale behind it. The value used for C99 has been increased in line with the percentage increases seen in other limits.

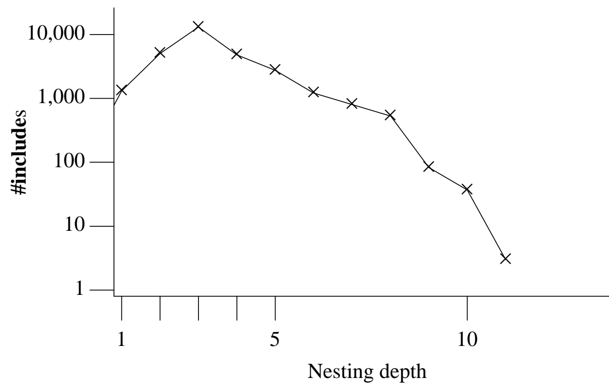

Figure 295.1: Number of **#include** preprocessor directives, that contain the quote-delimited form of header name (occurrences of the < > delimited form were not counted), having a given nesting depth. Based on the translated form of this book's benchmark programs.

C90

*257 case labels for a **switch** statement (excluding those for any nested **switch** statements)*

The intent here was to support **switch** statements that included 256 unsigned character values plus EOF (usually implemented as -1).

switch statement

C++

The following is a non-normative specification.

*Case labels for a **switch** statement (excluding those for any nested **switch** statements) [16384]*

Annex Bp2

Common Implementations

The number of **case** labels in a **switch** statement may affect the generated code. For instance, a processor instruction designed to indirectly jump based on an index into a jump table may have a 256 (bytes or addresses) limit on the size of jump table supported.

selection statement syntax

Coding Guidelines

The number of **case** labels in a **switch** statement is an indication of the complexity of the application, or the algorithm. Splitting a **switch** statement into two or more **switch** statements solely for the purpose of reducing the number of **case** labels within an individual **switch** statement has costs and benefits. The following are some of the issues:

- The size of a large **switch** statement, in the visible source, can affect the ease of navigation of the function containing it.
- In some environments (e.g., freestanding environments) there may be resource limitations (these may occur as a result of processor architecture or internal limits of the translator used); for instance, generating to case tables when there are large ranges of unused case values may make inefficient use of storage, or mapping to if/else pairs may cause a critical section of code to execute too slowly.
- The code complexity is increased (e.g., a conditional test that did not previously exist needs to be introduced).
- There may not be semantically meaningful disjoint sets of **case** labels that can form a natural division into separate **switch** statements.

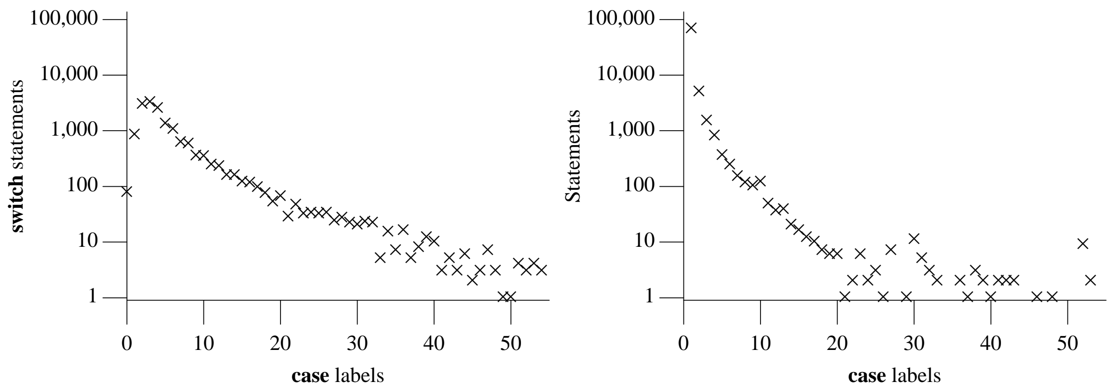

Figure 296.1: Number of **switch** statements containing the given number of **case** labels (left) and number of individual statements labeled by a given number of **case** labels (right). Based on the visible form of the `.c` files. Note that counts do not include occurrences of the **default** label.

limit members in struct/union

— 1023 members in a single structure or union 297

Commentary

This limit does not include the members of structure tags, or typedef names, that have been defined elsewhere and are referenced in a structure or union definition.

This limit may be reached in automatically generated code.

C90

127 members in a single structure or union

C++

The following is a non-normative specification.

Annex Bp2

Data members in a single class, structure or union [16384]

Coding Guidelines

The organization of the contents of data structures is generally determined by the application and algorithms used. While it may be difficult to imagine a structure definition containing a large number of members without some subset sharing a common characteristic, which enables them to be split into separate definitions; this does not mean that structures with large numbers of members cannot occur for a good reason. This issue is discussed in more detail elsewhere.

limit struct/union nesting 299

It is unlikely that a, human-written, union definition would ever contain a large number of members.

Usage

Measurements of classes,[7] in large Java programs, have found that the number of members follows the same pattern as that in C (see Figure 297.1).

limit enumeration constants

— 1023 enumeration constants in a single enumeration 298

Commentary

This C99 limit has a value comparable to the increased limits of other constructs sharing the same C90 limit.

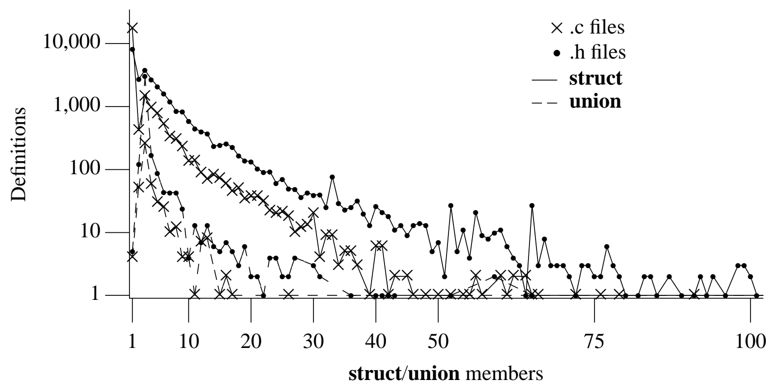

Figure 297.1: Number of structure and union type definitions containing the given number of members (members in any nested definitions are not included in the count of members of the outer definition). Based on the visible form of the `.c` and `.h` files.

C90

127 enumeration constants in a single enumeration

C++

The following is a non-normative specification.

Enumeration constants in a single enumeration [4096]

Annex Bp2

Common Implementations

The only known implementation restrictions on enumeration constants in an enumeration is the amount of available memory.

Coding Guidelines

Enumerators are unstructured in the sense that it is not possible to break down an enumerator into component parts to be included in some higher-level enumerator.

Would an application ever demand a large number of enumeration constants in a single enumerator? There are some cases where a large number do occur; for instance, specifying the tokens in the SQL/2 grammar requires 318 enumeration constants. Limiting the number of enumeration constants in an enumeration would not appear to offer any benefits, especially since there is no mechanism (like there is for structure members) for creating hierarchies of enumeration constants.

---

299 — 63 levels of nested structure or union definitions in a single struct-declaration-list

limit struct/union nesting

Commentary

The specification of definitions, rather than types, suggests that this limit does not include nested structures and unions that occur via the use of typedef names.

C90

15 levels of nested structure or union definitions in a single struct-declaration-list

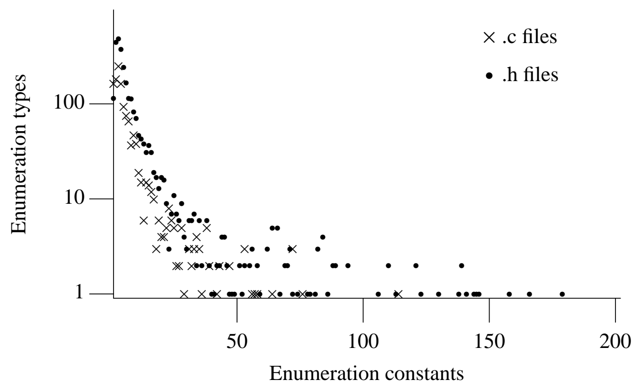

Figure 298.1: Number of enumeration types containing a given number of enumeration constants. Based on the visible form of the `.c` and `.h` files (also see Figure **??**).

C++

The following is a non-normative specification.

Annex Bp2

Levels of nested class, structure, or union definitions in a single `struct-declaration-list` *[256]*

Other Languages

Fortran 90 and Common Lisp do not support the lexical nesting of record declarations.

Common Implementations

Nesting of definitions is part of the language syntax and usually implemented using a table-driven syntax analyzer. Most nested structure and union definitions occur at file scope; that is, they are not nested within other constructs. However, even if a declaration occurs in block scope, it is likely that any internal table limits will not be exceeded by a definition nested to 63 levels. This limit is only half that of nested block levels, even if creating a new level of structure nesting consumes 2 to 3 times as many table entries as a new level of nested block, there is likely to be sufficient table entries remaining to handle a deeply nested structure definition.

Coding Guidelines

Textual nesting of structure and union definitions is not necessary. C contains a mechanism (typedef names or tags) that removes the need for any textual nesting within structure or union definitions. Is there some optimal level of nesting, or can developers simply create whatever declarations suit them at the time?

The following are some of the benefits of using textually nested definitions:

- Having a definition nested within the structure of which it is a subcomponent highlights, to readers, the close association between the two types.
- Nested definitions may reduce reader effort (e.g., source code scanning) in locating a needed definition (e.g., the type of a member will be visible in the source next to where the member appears).
- If a single instance of a definition is needed, there is no need to create a tag name for it.

Some of the costs of using textually nested definitions include:

- Given the typical source code indentation strategies used for nested definitions, it is likely that deeply nested definitions will cause the same layout problems as deeply nested blocks (this issue is discussed elsewhere).

limit block nesting 277

- There is a reader expectation that a reference to a tag name, either refers to a definition nested within the current definition or that is not nested within any other definition. Such an expectation increases search costs (because the search is performed visually rather than via, say, an editor search command).
- C++ compatibility— a structure/union definition is treated as a scope in C++, which means that names defined within it are not visible outside of it. Any references to tags nested within other definitions will cause a C++ translator to issue a diagnostic and the definition will have to be unnested before these references will be acceptable to a C++ translator.

scope kinds of

There are also costs and benefits associated with nested definitions, whether such definitions are created through textual occurrence in the source or the use of tag or typedef names. The alternative to using nested definitions is to have a single level of definition (i.e., all structure or union members have a scalar or array type).

The issues involved in deciding the extent to which members having a shared semantic association should be contained in a single structure definition (i.e., used whenever that member is required, sometimes creating a nested structure) or duplicated in more than one structure definition (which does not require a nested structure to be created) include:

- References (e.g., as an operand in an expression) to members nested within other definitions requires a sequence of member-selection operators (it may be possible to visually hide these behind a macro invocation). The visible source needed for accessing deeply nested members may impact the layout of the containing expression (e.g., a line break may be needed).

member selection

- As source code evolves, the information represented by a member that was once unique to one structure definition may need to be represented in other structure definitions. Creating a new type containing the related members may have many benefits (moving a member to such a definition changes what was a nonnested member into a nested member). However, the cost of adding a nesting level to an existing type definition can be high. The editing effort will be proportional to the number of occurrences of the moved members in the existing code, which requires the appropriate additional member-selection operation to be added.
- Type definitions are not always based on the categorization attributes of the application or algorithm. How a source code manipulates that information also needs to be considered. For instance, if some function needs to access particular information and takes as an argument a structure object that might reasonably contain a member holding that information, it might be decided to define the member holding that information in that structure type rather than another one.

categorization

- The advantages of organizing information according to shared attributes is discussed in the introduction.

developers organized knowledge

At the time of this writing algorithmic methods for optimally, or even approximately optimal, selecting structure type hierarchies are not known. Some of the considerations such an algorithm would need to take into account: the effort needed by readers to recall which structure type included which member, the effort needed to modify the types as the source evolved over time, and the structuring requirements caused by the need to pass arguments or create pointers to members.

Example

```
1 struct R {
2         struct {
3                int mem_1;
4                } mem_2;
5         };
6 struct S {
7         /*
```

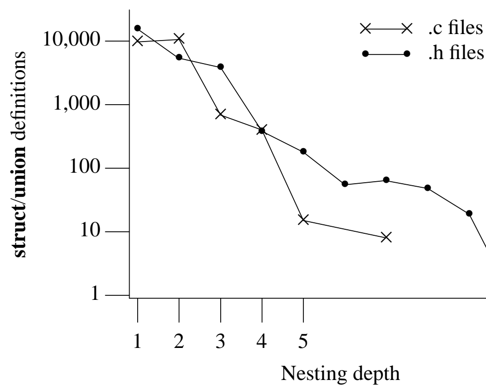

Figure 299.1: Number of structure and union type definitions containing the given number nested members that are textually structure and union type definitions (i.e., definitions using `{ }` not typedef names). Based on the visible form of the `.c` and `.h` files.

```
8              * A tag is only needed if the structure type is referred to.
9              */
10            struct T {
11                     int mem_1;
12                     } mem_2;
13            };
14
15   struct U {
16            int mem_1;
17            };
18   struct V {
19            struct U mem_2;
20            };
```

References

1. J. Engblom. Why SpecInt95 should not be used to benchmark embedded systems tools. *ACM SIGPLAN Notices*, 34(7):96–103, July 1999.
2. ISO. *Implementation of ISO/IEC 8651-4:1995 Information technology —Computer graphics —Graphics kernel system (GKS) language bindings —Part 4: C*. ISO, 1995.
3. Microsoft. *Microsoft QuickC Compiler*. Microsoft Corporation, 1987.
4. MISRA. *Guidelines for the Use of the C Language in Vehicle Based Software*. Motor Industry Research Association, Nuneaton CV10 0TU, UK, 1998.
5. MISRA. *MISRA-C:2004 Guidelines for the Use of the C Language in Vehicle Based Software*. Motor Industry Research Association, Nuneaton CV10 0TU, UK, 2004.
6. U. M. of Defence. *Defence Standard 00-55. Requirements for safety related software in defence equipment. Part 2: Guidance*. UK Ministry of Defence, Aug. 1997.
7. R. Wheeldon and S. Counsell. Power law distributions in class relationships. In *Third International Workshop on Source Code Analysis and Manipulation (SCAM 2003)*, pages 45–54, Sept. 2003.