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.

6.9 External definitions

translation unit syntax
external declaration syntax

1810

```
translation-unit:
                external-declaration
                translation-unit external-declaration
external-declaration:
                function-definition
                declaration
```

Commentary

syntactically analyzed preprocessor directives syntax
translation unit known as

A *translation-unit* is the terminal production of C syntax. By the time a translator needs to perform syntax analysis the preprocessor directives, such as **#include** and **#define**, have been deleted (C syntax essentially has two terminal productions, the other being *preprocessing-file*). The term *translation unit* applies to the result of preprocessing a source file. This syntax requires that a translation unit contain at least one declaration (i.e., a source file that only contains preprocessing directives is not conforming). One of the reasons for this requirement is that some translators require the object file they create to contain at least one symbol.

The mechanism by which identifiers declared in separate translation units are made to refer to each other is linkage.

linkage

C++

The C++ syntax includes function definitions as part of declarations (3.5p1):

3.5p1
> *translation-unit:* *declaration-seq*$_{opt}$

While the C++ Standard differs from C in supporting an empty translation unit, this is not considered a significant difference.

Other Languages

Some languages (e.g., Cobol and the first Pascal Standard) do not support any external definitions because they require all of a programs source to be translated at the same time. While other languages (e.g., Ada) provide sophisticated separate compilation mechanisms. Some languages impose a structure, or ordering, on the components in a translation unit. For instance, Pascal requires constants to be declared first, followed by type declarations, then variable declarations and then function/procedure declarations.

Coding Guidelines

Although the interface issues are generally considered to be a significant source of faults in software there have been few empirical studies of these kinds of fault. Although a few studies have looked at source code level (e.g., incorrect number of parameters),[13] most have tended to investigate higher level algorithmic or design issues.[10]

prototypes cost/benefit

Prototypes

Use of function prototypes are strong recommended, if not mandated, by nearly every coding guideline document. Surprisingly there is little empirical evidence to support the claimed benefits over not using a prototype (because nobody has done the experiments). The one study that has been performed,[11] using experienced developers, showed that use of prototypes did increase productivity. The experiment was not typical of industry practice in that subjects were provided with a paper listing of the names of all types, constants, and functions that might be required. If a program is being written from scratch the cost of using prototypes is minimal and is probably recouped the first time a translator diagnoses a mismatch between the number of parameters in a function definitions and the number arguments given in a call to it. Use of prototypes are the only economically worthwhile option.

Cg 1810.1

An *external-declaration* that declares a function shall always contain a parameter type list (i.e., a function prototype).

However, the cost/benefit decision is not so clear-cut when modifying existing code that contains old-style definitions. When making a small modification to existing code, for instance adding a function call, the impact of changing the called function (to use a prototype in its definition) may extend throughout a programs source tree. Other source files, that are not directly touched by the original modification, may contain calls to the function whose definition has been changed (to use a prototype).

Should function definitions in existing code, that don't use prototypes, be changed to use prototypes on an individual basis, as small modifications are made to the source? Or should such changes to function definitions only be performed as part of a more global reengineering of the source code? The answer to these questions will depend on what phase of its life cycle a program is in, the current commercial environment, and technical factors such as the product development environment and the product testing procedures used.

Ordering of declarations

Within a source file external declarations usually occur in some kind of recognizable order. For instance, function definitions usually appear after all of the other kinds of declarations and definitions. There is a benefit in using a recognizable order for declarations and definitions, it is information that readers can use when searching for the declaration of a particular identifier. There are a number of factors that might be considered when ordering identifier declarations, including:

- Ordering by C language attributes associated with the declared identifier. For instance, one ordering might be: some preprocessor directives (see elsewhere for a discussion on the ordering of preprocessor directives), followed by typedef names, definitions of objects with external linkage, then objects with internal linkage, and so on.
- Ordering by algorithmic or implementation attributes associated with the declared identifier. For instance, the macro names, typedef names, and objects color, in a translation unit that handles printing, might all be declared together in one section of the source file.

preprocessor directives syntax

Both ordering have their own costs and benefits. For instance, ordering by C language attribute allows identifiers having that attribute to be quickly located within a source file. However, reading the declaration of an identifier may create the need to obtain information about identifiers associated with it (e.g., type declarations). An ordering that places declarations of identifiers sharing some form of algorithmic or implementation attributes together could reduce effort for searches based on these attributes.

Selecting an optimal declaration ordering requires information on the search patterns of future readers of the source. Given that this information is unlikely to be available, the most that the authors of declarations can do is ensure they use an ordering that future readers will recognize and be able to use when searching for identifier declarations.

Which declarations in which source file?

declarations in which source file

Technically any external declaration of an object or function could be defined in any of the sources files used to build a program (provided the necessary type definitions were visible). However, experience shows that when writing a new declaration, developers often attempt to put it in a source file containing other external declarations that are considered to be related to it. The set of external object and function contained in a translation unit are often considered to be members of a category. For instance, the contents of the library headers defined by the C Standard (e.g., string handling, time information, character handling, etc.).

This developer behavior of organizing object and function definitions into categories is a special case of general human categorization behavior. People use information about category membership in a number of ways. For instance, as an aid to recall, or to deduce properties or behavior of a category member based on their knowledge of other category members.

categorization developers organized knowledge

If the classification process is driven by individual choice, then it is necessary to ask to what extent the classification chosen by one developer is of benefit to another developer. At the time of this writing there

have been no studies of developer classification behavior that might be used to suggest an answer to this question (although the semantic processed involved in creating identifier spellings may be related).

A number of mathematical methods for *software clustering* (as the field has become known) have been proposed. These methods are generally legacy systems oriented in that they take the source of an existing program and attempt to find the subsystems from which it is composed.[15] They vary in the algorithms used and the measure of source code similarity used. Mitchell and Mancoridis[9] empirically compare various algorithms and similarity measures. Fasulo[5] provides an analysis of recent mathematical work on general clustering algorithms (i.e., not software specific).

application evolution

A common characteristic of programs that have an active community of users is that they continue to grow and evolve. For instance, an analysis, by Godfrey and Tu[6] of 96 releases of the source of the Linux kernel found that the number of uncommented lines of code closely fitted the equation $0.21 \times X^2 + 252 \times X + 90055$, where X is the number of days since release 1.0. At the other end of the scale are programs that are rarely used and whose source is relatively unchanging. The source files are changed tend to be those dealing with areas that are important to business.[7]

When adding new functionality to existing code developers are faced with making a cost/benefit analysis. Should they perform any restructuring that they feel is necessary (e.g., breaking up the contents of large source file into smaller source files, each representing some facet of the category represented by the original source), an investment cost that may not be repaid by a future benefit, or should they ignore future costs and minimize current costs (i.e., no restructuring)?

These issues are very complex and at the time of this writing it does not appear to be possible to give any simple guideline recommendation on which source file object and function declarations should appear in.

coupling

The coupling between two source files (i.e., the extent to which their objects and functions each other) has been found to affect the number of faults in a program.[3, 12] The commercial benefits, to hardware vendors, of minimizing the coupling between separate units is well documented.[1] However, the benefits may not unconditionally apply to software[4] and minimizing coupling may even increase costs in some cases.[2] Until the costs and benefits associated with coupling are better understood it is not possible to know if any guideline recommendation would be worthwhile.

By default, in C, external declarations of identifiers for objects and functions are visible outside of the translation unit that declares them (i.e., they have external linkage). Explicitly declaring identifiers to have internal linkage has the benefit of reducing the probability of name clashes with identifiers declared in other translation units. However, while the developer cost of typing the keyword **`static`** is negligible, the cost of working out which identifiers can be so declared may be nontrivial. Since the cost of a guideline recommending the use of internal linkage, where possible, may often be greater than the benefit, no such recommendation is made here.

Usage

duplicate code
latent semantic analysis

On a large development project it is possible that more than one person will write some set of functions performing similar operations. This duplication of functionality occurs at a higher-level than copying and reusing sequences of statements (discussed elsewhere), it is a concept that is being duplicated. Marcus and Maletic[8] used latent semantic analysis to identify related source files (what they called *concept clones*). Source code identifiers and words in comments were used as input to the indexing process. An analysis of the Mozilla source code highlighted two different implementations of linked list functions and four files that contained their own implementations.

Constraints

external declaration not auto/register

The storage-class specifiers **`auto`** and **`register`** shall not appear in the declaration specifiers in an external declaration. 1811

Commentary

While it would have been possible to allow the storage-class specifier **`register`** to appear on some external declarations (e.g., those having internal linkage) the Committee did not.

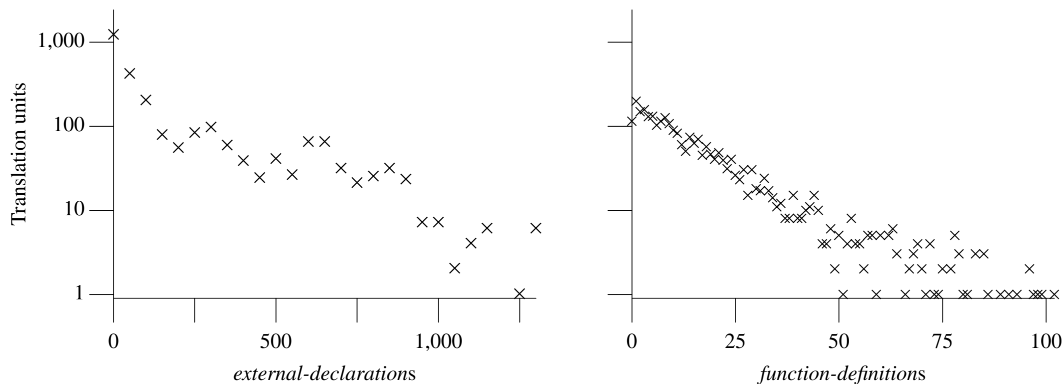

Figure 1810.1: Number of translation units containing a given number of `external-declaration`s and `function-definition`s declarations (rounded to the nearest fifty and excluding identifiers declared in any system headers that are **`#include`**d). Based on the translated form of this book's benchmark programs.

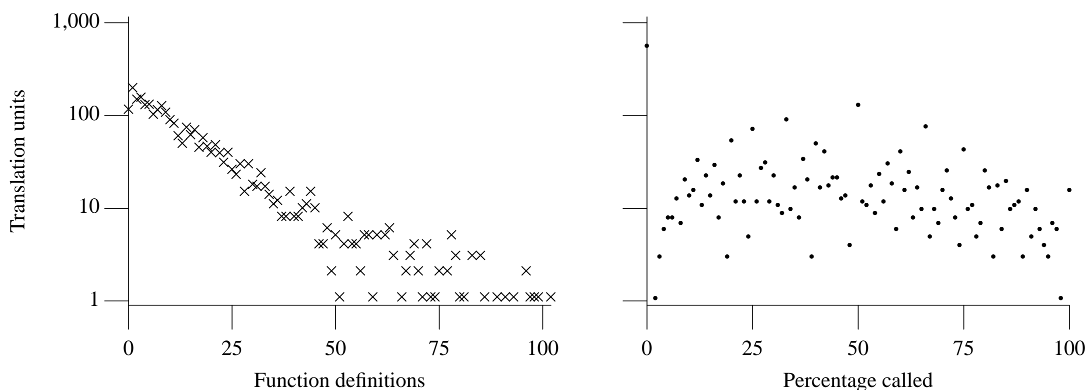

Figure 1810.2: Number of translation units containing a given number of function definitions and percentage of functions that are called within the translation unit that defines them. Based on the translated form of this book's benchmark programs.

C++

The C++ Standard specifies where these storage-class specifiers can be applied, not where they cannot:

> *The* **`auto`** *or* **`register`** *specifiers can be applied only to names of objects declared in a block (6.3) or to function parameters (8.4).* 7.1.1p2

A C++ translator is not required to issue a diagnostic if the storage-class specifiers **`auto`** and **`register`** appear in other scopes.

Common Implementations

Some implementations[14] take the complete program into account, during register allocation, rather than just one individual function definitions at a time. However, this is one case where developers may still be able to use their knowledge of program behavior to make better use of register resources (high-quality automatic register allocation is very dependent on the use of program execution traces). `gcc` supports the **`register`** storage class appearing in the declaration of objects at file scope. However, the register to be used needs to be explicitly specified, for instance in:

```
1   register int *ipc asm("a5");
```

the register `a5` is dedicated to holding the value of the object `ipc`.

definition one external

There shall be no more than one external definition for each identifier declared with internal linkage in a translation unit. 1812

Commentary

external definition 1817
tentative definition
extern identifier linkage same as prior declaration

This C sentence is referring to an *external definition*, as defined below. It is possible for there to be no explicit external definition, in which case a tentative one is implicitly created. It is possible for the same *external definition* to be declared more than once in the same translation unit.

Other Languages

Most languages only allow one definition of any kind of identifier. Fortran allows more than one and linkers are expected to pick a unique instance.

linkage

internal linkage exactly one external definition

Moreover, if an identifier declared with internal linkage is used in an expression (other than as a part of the operand of a **`sizeof`** operator whose result is an integer constant), there shall be exactly one external definition for the identifier in the translation unit. 1813

Commentary

external definition 1817

This C sentence is referring to an external definition, as defined below, not an identifier declared with external linkage. An identifier may be declared with internal linkage and a function type. A definition for this function is only required (apart from the specified special case) if the identifier is used in an expression. If the result of the **`sizeof`** operator is an integer constant its evaluation can be performed during translation and there is no need for a definition to exist.

C++

The C++ Standard does not specify any particular linkage:

> 3.2p3 *Every program shall contain exactly one definition of every non-inline function or object that is used in that program; no diagnostic required.*

The definition of the term used (3.2p2) also excludes operands of the **`sizeof`** operator.

Other Languages

Some languages (e.g., Ada and Pascal) require a definition if there is a declaration of an identifier.

Coding Guidelines

Function declarations, with internal linkage, that are not referenced within a translation unit are redundant. The issue of redundant code is discussed elsewhere.

redundant code

Semantics

As discussed in 5.1.1.1, the unit of program text after preprocessing is a translation unit, which consists of a sequence of external declarations. 1814

Commentary

translation unit known as
external declaration syntax 1810

Clause 5.1.1.1 defines the term *translation unit* as such. The use of the term *external declarations* here refers to the syntactic definition. These external declarations include declarations of identifiers that have internal linkage.

C++

The C++ Standard does not make this observation.

These are described as “external” because they appear outside any function (and hence have file scope). 1815

Commentary

A term that is commonly used to refer to such declarations, by developers working in a variety of computer languages, is *global*.

C++

The C++ Standard does not refer to them as "external" in the syntax.

Coding Guidelines

Many developers intermix the terms *external* and *global*. There is no obvious benefit to be had in changing this practice (if this were at all possible).

1816 As discussed in 6.7, a declaration that also causes storage to be reserved for an object or a function named by the identifier is a definition.

Commentary

The discussion of this C behavior occurs elsewhere.

definition identifier

1817 An *external definition* is an external declaration that is also a definition of a function (other than an inline definition) or an object.

external definition

Commentary

This defines the term *external definition*. The definition of an inline definition specifies that it does not provide an external definition.

inline definition not an external definition

C90

Support for inline definitions new in C99.

C++

The C++ Standard does not define the term *external definition*, or one equivalent to it.

Coding Guidelines

While many developers intermix the terms *external definition* and *external declaration*, the distinction between them is significant. Coding guideline documents need to ensure that they use correct terminology.

1818 If an identifier declared with external linkage is used in an expression (other than as part of the operand of a **`sizeof`** operator whose result is an integer constant), somewhere in the entire program there shall be exactly one external definition for the identifier;

external linkage exactly one external definition

Commentary

This requirement is the external linkage equivalent of the requirement given for identifiers having internal linkage. However, violation of this requirement causes undefined defined behavior, it is not a constraint violation. The reason for this difference in behavior is that the C committee wanted implementations to be able to use third-party (e.g., the host OS vendor) linkers provided as part of the translation host environment. Some of these linkers support more relaxed, Fortran style, linkage conventions, and it is not possible to guarantee that a diagnostic will be issued for a violation of this requirement.

1813 internal linkage exactly one external definition
linkers
linkage

C90

Support for the **`sizeof`** operator having a result that is not a constant expression is new in C99.

C++

The specification given in the C++ is discussed elsewhere.

1813 internal linkage exactly one external definition

Other Languages

Some languages (e.g., Ada) have sophisticated separate compilation mechanisms that require specialist linker support, while the designers of other languages have been willing to live within the limitations of linkers that their implementations are likely to make use of.

Common Implementations

Many linkers do not issue a diagnostic if more than one definition of the same identifier is contained within their input files. However, most linkers do issue a diagnostic if the program image they are asked to create includes a reference to an identifier for which there is no available definition.

Many linkers create identifiers for their own internal use. For instance some Unix linkers create the identifiers `etext`, `edata`, and `end`, to designate special addresses within a programs address space. These identifiers are used purely as symbols that represent an address, they occupy no storage. Some programs make use of the information provided by these addresses. Declarations, such as the following, can be used to provide an identifier that can be referenced in expressions (there is no definition of the object provided in the source of the program, it is provided by the linker).

```
1   extern const void end;
```

A fuller discussion of using such a declaration can be found in DR #012.

Coding Guidelines

identifier ?? declared in one file

Following the guideline recommendation dealing with the textual placement of all identifiers, having external linkage, in a single header prevents the creation of multiple declarations that may be incompatible with each other. However, every object or function referenced, in a program, requires a unique definition. One translation unit needs to contain an identifier's definition, while all the others only contain its declaration. There are a number of techniques for ensuring that declarations and their corresponding definition match. Two commonly seen techniques are:

1. using a macro, often called `EXTERN`, or `EXTERNAL`. For instance, consider a developer written header containing the following declaration:

   ```
   1   EXTERN int glob;
   ```

   macro object-like

   object reserve storage

   In every translation unit, except one, that includes this header, the identifier `EXTERN` is defined as an object-like macro that expands to the keyword **`extern`**. In one translation unit the macro name expands to nothing, causing that declaration to become a definition. Variations on this technique are used to handle explicit initialization (in fact if an explicit initializer is given the absence or presence of the **`extern`** storage-class specifier is irrelevant, a definition is always created),

definition identifier

2. placing a definition of the identifier, that is textually separate from the declaration in the header file, in one of the translation units. Use of this technique does violate the guideline recommendation specifying a single textual occurrence of a declaration (a definition is also a declaration). Recreating the problem of ensuring that both declarations are the same.

In the case of functions, their declarations and definitions have to be textually separate. The complexities of using preprocessor directives to enable one textual occurrence to be used both as the declaration and the definition has a much higher cost than benefit. A method of ensuring that the two textual occurrences are the same is needed. One solution, that can be applied to both object and function declarations, is to make use of a property of the C language. It is possible to have multiple declarations of the same identifier, in the same scope, with external linkage, provided their declarations are compatible (a slightly less restrictive requirement than being the same). Translators are required to issue a diagnostic if the types are not compatible, so automated checking is performed.

Cg 1818.1 A source file that contains a textual definition of an object or function, having external linkage, shall **`#include`** the header file that contains its textual declaration.

1819 otherwise, there shall be no more than one.[137)]

Commentary

An identifier declared in a program may also be defined, even if it is not referenced in the program. However, whether such an identifier is referenced or not, the requirement on there being at most one definition is the same.

C++

The C++ Standard does not permit more than one definition in any translation unit (3.2p1). However, if a non-inline function or an object is not used in a program it does not prohibit more than one definition in the set of translation units making up that program.

Source developed using a C++ translator may contain multiple definitions of objects that are not referred.

Common Implementations

Some translators optimize away (by not writing any information about them to the object file) objects with internal linkage that are not referenced within the translation unit that contains their definition. Many linkers attempt to only include the definitions of objects and functions, in a program image, that are referenced from within that program.

Coding Guidelines

An identifier that is defined and not referenced is redundant code. This issue is discussed elsewhere.

redundant code

1820 137) Thus, if an identifier declared with external linkage is not used in an expression, there need be no external definition for it.

footnote 137

Commentary

This permission is needed to support the C model of separate translation. A header file containing many declarations may be included in the source files used to build a program. However, because a definition for an identifier declared in a header is only required if the identifier is referenced, developers do not have to be concerned about what is declared in the headers they include.

C++

The C++ Standard does not make this observation.

Other Languages

Whether or not it is possible to declare an identifier without also providing a definition depends on the separate translation model used by a language.

References

1. C. Y. Baldwin and K. B. Clark. *Design Rules. Volume 1. The Power of Modularity*. The MIT Press, 2000.
2. R. D. Banker and S. A. Slaughter. The moderating effects of structure on volatility and complexity in software enhancement. *Information Systems Research*, 11(3):219–240, Sept. 2000.
3. V. R. Basili, L. C. Briand, and S. Morasca. Defining and validating measures for object-based high-level design. Technical Report IESE-Report No. 018.98/E, Fraunhofer Institute for Experimental Software Engineering, 1998.
4. A. Brooks, J. Daly, J. Miller, M. Roper, and M. Wood. Replication of experimental results in software engineering. Technical Report ISERN-96-10, Department of Computer Science, University of Strathclyde, Livingstone Tower, Richmond Street, Glasgow G1 1XH, UK, 1996.
5. D. Fasulo. An analysis of recent work on clustering algorithms. Technical Report Technical Reptort 01-03-02, Dept. of Computer Science and Engineering, University of Washington, 1999.
6. M. W. Godfrey and Q. Tu. Evolution in open source software: A case study. In *Proceedings of the International Conference on Software Maintenance (ICSM'00)*, pages 131–142, Oct. 2000.
7. C. F. Kemerer and S. Slaughter. Determinants of software maintenance profiles: An empirical investigation. *Software Maintenance: Research and Practice*, 9(4):235–251, 1997.
8. A. Marcus and J. I. Maletic. Identification of high-level concept clones in source code. In *Proceedings of the* 16th *IEEE International Conference on Automated Software Engineering (ASE 2001)*, pages 107–114, Nov. 2001.
9. B. S. Mitchell and S. Mancoridis. Comparing the decompositions produced by software clustering algorithms using similarity measures. In *Proceedings of the 2001 International Conference on Software Maintenance (ICSM'01)*, pages 744–753. IEEE, Apr. 2001.
10. D. E. Perry and W. M. Evangelist. An empirical study of software interface faults. In *International Symposium on New Directions in Computing*, pages 32–38. IEEE, Aug. 1985.
11. L. Prechelt and W. F. Tichy. A controlled experiment measuring the effect of procedure argument type checking on programmer productivity. Technical Report Technical Report CMU/SEI-96-TR-014, Software Engineering Institute, Carnegie Mellon University, 1996.
12. R. W. Selby and V. R. Basili. Analysing error-prone system structure. *IEEE Transactions on Software Engineering*, 17(2):141–152, Feb. 1991.
13. T. A. Thayer, M. Lipow, and E. C. Nelson. *Software Reliability*. North-Holland Publishing Company, 1978.
14. D. W. Wall. Global register allocation at link time. Technical Report 86/3, Western Research Lab, Oct. 1986.
15. T. A. Wiggerts. Using clustering algorithms in legacy systems remodularizations. In 4th *Working Conference on Reverse Engineering (WCRE '97)*, pages 33–43. IEEE, Oct. 1997.