The New C Standard (Excerpted material)

An Economic and Cultural Commentary

Derek M. Jones
derek@knosof.co.uk

5.1.1.1 Program structure

program not translated at same time

A C program need not all be translated at the same time. 107

Commentary

translation unit syntax
translation unit linked
program image compatible
separate translation units
JIT

C's separate compilation model is one of independently translated source files that are merged together by a linker to form a program image. There is no concept of program library built into the language. Neither is there any requirement to perform cross-translation unit checking, although there are cross-translation unit compatibility rules for derived types.

There is no requirement that all source files making up a C program be translated prior to invoking the function `main`. An implementation could perform a JIT translation of each source file when an object or function in an untranslated source file is first referenced (a translator is required to issue a diagnostic if a translation unit contains any syntax and constraint violations).

linkage

Linkage is the property used to associate the same identifier, declared in different translation units, with the same object or function.

Other Languages

Some languages enforce strict dependency and type checks between separately translated source files. Others have a very laid-back approach. Some execution environments for the Basic language delay translation of a declaration or statement until it is reached in the flow of control during program execution. A few languages require that a program be completely translated at the same time (Cobol and the original Pascal standard).

Java defines a process called resolution which, " . . . is optional at the time of initial linkage."; and "An implementation may instead choose to resolve a symbolic reference only when it is actively used; . . . ".

Common Implementations

Most implementations translate individual source files into object code files, sometimes also called object modules. To create a program image, most implementations require all referenced identifiers to be defined and externally visible in one of these object files.

Coding Guidelines

The C model could be described as one of *it's up to you to build it correctly or the behavior is undefined.*. Having all of the source code of a program in a single file represents poor practice for all but the smallest of programs. The issue of how to divide up source code into different sources files, and how to select what definitions go in what files, is discussed elsewhere. There is also a guideline recommendation dealing with the uniqueness and visibility of declarations that appear at file scope.

external declaration syntax
identifier ??
declared in one file

Example

The following is an example of objects declared in different translation units with different types.

file_1.c

```
1 extern int glob;
```

file_2.c

```
1 float glob = 1.0;
```

Usage

A study by Linton and Quong[5] used an instrumented `make` program to investigate the characteristics of programs (written in a variety of languages, including C) built over a six-month period at Stanford University. The results (see Figure 107.1) showed that approximately 40% of programs consisted of three or fewer translation units.

source files
preprocessing files

The text of the program is kept in units called *source files*, (or *preprocessing files*) in this International Standard. 108

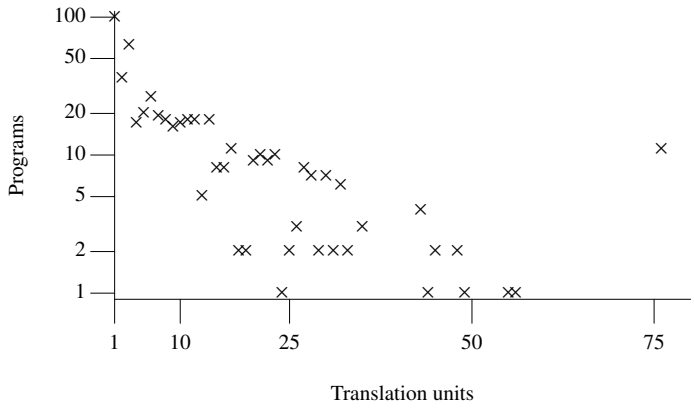

Figure 107.1: Number of programs built from a given number of translation units. Adapted from Linton.[5]

Commentary

This defines the terms *source files* and *preprocessing files*. The term *source files* is commonly used by developers, while the term *preprocessing files* is an invention of the Committee.

C90

The term *preprocessing files* is new in C99.

C++

The C++ Standard follows the wording in C90 and does not define the term *preprocessing files*.

Other Languages

The Java language specification[3] strongly recommends that certain naming conventions be followed for package names and class files. The names mimic the form of Web addresses and RFCs 920 and 1032 are cited.

Common Implementations

A well-established convention is to suffix source files that contain the object and function definitions with the `.c` extension. Header files usually being given a `.h` suffix. This convention is encoded in the `make` tool, which has default rules for processing file names that end in `.c`.

Coding Guidelines

file name abbreviations

Restrictions on the number of characters in a filename are usually more severe than for identifiers (MS-DOS 8.3, POSIX 14). These restrictions can lead to the use of abbreviations in the naming of files. An automated tool developed by Anquetil and Lethbridge[1] was able to extract abbreviations from file names with better than 85% accuracy. A comparison of automated file clustering,[2] against the clustering of files in a large application, by a human expert, showed nearly 90% accuracy for both precision (files grouped into subsystems to which they do not belong) and recall (files grouped into subsystems to which they do belong).

Development groups often adopt naming conventions for source file names. Source files associated with implementing particular functionality have related names, for instance:

1. Data manipulation: *db* (database), *str* (string), or *queue*.
2. Algorithms or processes performed: *mon* (monitor), *write*, *free*, *select*, *cnv* (conversion), or *chk* (checking).
3. Program control implemented: *svr* (server), or *mgr* (manager).
4. The time period during which processing occurs: *boot*, *ini* (initialization), *rt* (runtime), *pre* (before some other task), or *post* (after some other task).

5. I/O devices, services or external systems interacted with: *k2*, *sx2000*, (a particular product), *sw* (switch), *f* (fiber), *alarm*.
6. Features implemented: *abrvdial* (abbreviated dialing), *mtce* (maintenance), or *edit* (editor).
7. Names of other applications from where code has been reused.
8. Names of companies, departments, groups or individuals who developed the code.
9. Versions of the files or software (e.g., the number 2 or the word *new* may be added, or the name of target hardware), different versions of a product sold in different countries (e.g., *na* for North America, and *ma* for Malaysia).
10. Miscellaneous abbreviations, for instance: *utl* (utilities), or *lib* (library).

identifier selecting spelling

The standard has no concept of directory structure. The majority of hosts support a file system having a directory structure and larger, multisource file projects often store related source files within individual directories. In some cases the source file directory structure may be similar to the structure of the major components of the program, or the directory structure mirrors the layered structure of an application.[4] The issues involved in organizing names into the appropriate hierarchy are discussed later.

structure type sequentially allocated objects
enumeration set of named constants
abbreviating identifier
identifier introduction

Files are not the only entities having names that can be collected into related groups. The issues associated with naming conventions, the selection of appropriate names and the use of abbreviations. are discussed elsewhere.

source file inclusion
header name same as .c file

Source files are not the only kind of file discussed by the C Standard. The **#include** preprocessing directive causes the contents of a file to be included at that point. The standard specifies a minimum set of requirements for mapping these header files. The coding guideline issues associated with the names used for these headers is discussed elsewhere.

---

preprocessing translation unit known as

A source file together with all the headers and source files included via the preprocessing directive **#include** is known as a *preprocessing translation unit*. 109

Commentary

This defines the term *preprocessing translation unit*, which is not generally used outside of the C Standard Committee. A preprocessing translation unit contains all of the possible combinations of translation units that could appear after preprocessing. A preprocessing translation unit is parsed according to the syntax for preprocessing directives.

preprocessor directives syntax

C90

The term *preprocessing translation unit* is new in C99.

C++

Like C90, the C++ Standard does not define the term *preprocessing translation unit*.

Other Languages

Java defines the term *compilation unit*. Other terms used by languages include *module*, *program unit*, and *package*.

Coding Guidelines

Use of this term by developers is almost unknown. The term *source file* is usually taken to mean a single file, not including the contents of any files that may be **#include**d. Although a slightly long-winded term, *preprocessing translation unit* is the technically correct one. As such its use should be preferred in coding guideline documents.

---

translation unit known as

After preprocessing, a preprocessing translation unit is called a *translation unit*. 110

Commentary

This defines the term *translation unit*. A translation unit is the sequence of tokens that are the output of translation phase 4. The syntax for translation units is given elsewhere.

translation phase 4
translation unit syntax

C90

A source file together with all the headers and source files included via the preprocessing directive ***#include****, less any source lines skipped by any of the conditional inclusion preprocessing directives, is called a translation unit.*

This definition differs from C99 in that it does not specify whether macro definitions are part of a translation unit.

C++

The C++ Standard, 2p1, contains the same wording as C90.

Common Implementations

In many translators the task of turning a preprocessing translation unit into a translation unit is the job of a single program .

footnote 5

Coding Guidelines

Although the term *translation unit* is defined by the standard to refer to the sequence of tokens *after preprocessing*; the term is not commonly used by developers. The term *after preprocessing* is commonly used by developers to refer to what the standard calls a translation unit. There seems to be little to be gained in trying to change this common usage term.

Some of these coding guidelines apply to the sequence of tokens input to translation phase 7 (semantic analysis). There is rarely any difference between what goes into phase 5 and what goes into phase 7, and the term *after preprocessing* is commonly used by developers. For simplicity all phrases of translation after preprocessing are lumped together as a single whole. The guidelines in this book apply either to the visible source, before preprocessing, or after preprocessing.

translation phase 7

---

111 Previously translated translation units may be preserved individually or in libraries.

translation units preserved

Commentary

The standard does not specify what information is preserved in these translated translation units. It could be a high-level representation, even some tokenized form, of the original source. It is most commonly relocatable machine code and an associated symbol table.

The standard says nothing about the properties of libraries, except what is stated here.

Other Languages

Some languages specify the information available from, or the properties of, their separately translated translation units. Many languages, for instance Fortran, do not even specify as much as the C Standard.

Java defines several requirements for how packages are to be preserved and suggests several possibilities; for instance, class files in a hierarchical file system with a specified naming convention.

Common Implementations

In the Microsoft Windows environment, translated files are usually given the suffix `.obj` and libraries the suffix `.lib` or `.dll`. In a Unix environment, the suffix `.o` is used for object files and the suffix `.a` for library files, or `.so` for dynamically linked libraries.

Coding Guidelines

Coding guidelines, on the whole, do not apply to the translated output. Use of tools, such as `make`, for ensuring consistency between libraries and the translated translation unit they were built from, and the source code that they were built from, are outside the scope of this book.

translation units communication between

The separate translation units of a program communicate by (for example) calls to functions whose identifiers have external linkage, manipulation of objects whose identifiers have external linkage, or manipulation of data files. 112

Commentary

translation phase 8

translation unit syntax

compatible separate translation units

Translation phase 8 is responsible for ensuring that all references to external objects and functions refer to the same entity. The technical details of how an object or function referenced in one translation unit accesses the appropriate definition in another translation unit is a level of detail that the standard leaves to the implementation. The issues of the types of these objects agreeing with each other, or not, is discussed elsewhere.

Data read from a binary file is always guaranteed to compare equal to the data that was earlier written to the same file, during the same execution of a program. Separate translation units can also communicate by manipulating objects through pointers to those objects. These objects are not restricted to having external linkage. Similarly, functions can also be called via pointers to them. Visible identifiers denoting object or function definitions are not necessary.

Common Implementations

program startup

Information on the source file in which a particular function or object was defined is not usually available to the executing program. However, hosts that support dynamic linking provide a mechanism for implementations to locate functions that are referenced during program execution (most implementations require objects to have storage allocated to them during program startup).

Coding Guidelines

declarations in which source file

identifier ?? declared in one file

The issue of deciding which translation unit should contain which definition is discussed elsewhere, as is the issue of keeping identifiers declared in different translation units synchronized with each other.

translation unit linked

Translation units may be separately translated and then later linked to produce an executable program. 113

Commentary

program 107 not translated at same time

This is all there is to the C model of separate compilation. The C Standard places no requirements on the linking process, other than producing a program image. How the translation units making up a complete program are identified is not specified by the standard. The input to translation phase 8 requires, under a hosted implementation, at least a translation unit that contains a function called `main` to create a program image.

hosted environment startup

Common Implementations

Most translators have an option that specifies whether the source file being translated should be linked to produce a program image (translation phase 8), or the output from the translator should be written to an object file (with no linking performed). In a Unix environment, the convention is for the default name of the file containing the executable program to be `a.out`.

Forward references: linkages of identifiers (6.2.2), external definitions (6.9), preprocessing directives (6.10). 114

References

1. N. Anquetil and T. Lethbridge. Extracting concepts from file names: A new file clustering criterion. In *Proceedings of the 1998 International Conference on Software Engineering*, pages 84–93. IEEE Computer Society Press/ACM Press, 1998.

2. N. Anquetil and T. C. Lethbridge. Recovering software architecture from the names of source files. *Journal of Software Maintenance: Research and Practice*, 11:201–221, 1999.

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

4. B. Laguë, C. Ledue, A. L. Bob, E. Merlo, and M. Dagenais. An analysis framework for understanding layered software architectures. In 6^{th} *International Workshop on Program Comprehension*, 1998.

5. M. A. Linton and R. W. Quong. A macroscopic profile of program compilation and linking. *IEEE Transactions on Software Engineering*, 15(4):427–436, Apr. 1989.