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.

5.1.1.2 Translation phases

translation phases of

The precedence among the syntax rules of translation is specified by the following phases.[5)] 115

Commentary

These phases were introduced by the C committee to answer translation ordering issues that differed among early C implementations.

If one or more source files is **#include**d, the phases are applied, in sequence, to each file. So it is not possible for constructs created prior to phase 4 (which handles **#include**) to span more than one source file. For instance, it is not possible to open a comment in one file and close it in another file. Constructs that occur after phase 4 can span multiple files. For instance, a string literal as the last token in one file can be concatenated to a string literal which is the first token in an immediately **#include**d file.

The following quote from the Rationale does not belong within any specific phrase of translation, so it is provided here. UCNs are discussed elsewhere.

universal character name syntax

UCN models of

Rationale

. . . , how to specify UCNs in the Standard. Both the C and C++ Committees studied this situation and the available solutions, and drafted three models:

A. Convert everything to UCNs in basic source characters as soon as possible, that is, in translation phase 1.

B. Use native encodings where possible, UCNs otherwise.

C. Convert everything to wide characters as soon as possible using an internal encoding that encompasses the entire source character set and all UCNs.

Furthermore, in any place where a program could tell which model was being used, the standard should try to label those corner cases as undefined behavior.

C++

C++ has nine translation phases. An extra phase has been inserted between what are called phases 7 and 8 in C. This additional phase is needed to handle templates, which are not supported in C. The C++ Standard specifies what the C Rationale calls model A.

C++ model A 116

Other Languages

Most languages do not contain a preprocessor, and do not need to go to the trouble of explicitly calling out phases of translation. The C Standard might not have had to do this either, had it not been for the differing interpretations of the base document made by some translators.

base document

Java has three lexical translation steps (the first two are needed to handle Unicode).

Coding Guidelines

Few developers have any detailed knowledge of the phases of translation. It can be argued that use of constructs whose understanding requires a detailed knowledge of the phases of translation should be avoided. The problem is how to quantify *detailed knowledge*. Coding guidelines apply in all phases of translation, unless stated otherwise.

The distinction between preprocessor and subsequent phases is a reasonably well-known and understood division. The processes used by developers for extracting information from source code is likely to be affected by their knowledge of how a translator operates. Thinking in terms of the full eight phases is often unnecessary and overly complicated. The following phases are more representative of how developers view the translation process:

1. Preprocessing.
2. Syntax, semantics, and machine code generation.

3. Linking.

Example

```
#include <stdio.h>

int main(void)
{
printf("\\" "101");
}
```

116 1. Physical source file multibyte characters are mapped, in an implementation-defined manner, to the source character set (introducing new-line characters for end-of-line indicators) if necessary.

translation phase 1

Commentary

This phase maps the bits held on some storage device onto members of the source character set (which is defined elsewhere). C requires that sequences of source file characters be grouped into units called lines (actually there are two kinds of lines). There is a lot of variability between hosts on how end-of-line is indicated. This specification requires that whatever method is used at the physical level, it be mapped to a single new-line indicator.

source character set
118 translation phase 2
end-of-line representation

The source file being translated may reside on host A, with the implementation doing the translation may be executing on host B, and the translated program may be intended to run on host C. All three hosts could be using different character set representations. During this phase of translation, we are only interested in host A and host B. The character set used by host C is of no consequence, to the translator, until translation phase 5.

133 translation phase 5

C90

In C90 the source file contains characters (the 8-bit kind), not multibyte characters.

C++

2.1p1

1. Physical source file characters are mapped, in an implementation-defined manner, to the basic source character set (introducing new-line characters for end-of-line indicators) if necessary. . . . Any source file character not in the basic source character set (2.2) is replaced by the universal-character-name that designates that character.

```
#define mkstr(s) #s

char *dollar = mkstr($); // The string "\u0024" is assigned
                         /* The string "$", if that character is supported */
```

C++ model A

Rationale

The C++ Committee defined its Standard in terms of model A, just because that was the clearest to specify (used the fewest hypothetical constructs) because the basic source character set is a well-defined finite set.

The situation is not the same for C given the already existing text for the standard, which allows multibyte characters to appear almost anywhere (the most notable exception being in identifiers), and given the more low-level (or *close to the metal*) nature of some uses of the language.

Therefore, the C committee agreed in general that model B, keeping UCNs and native characters until as late as possible, is more in the "spirit of C" and, while probably more difficult to specify, is more able to encompass the existing diversity. The advantage of model B is also that it might encompass more programs and users' intents than the two others, particularly if shift states are significant in the source text as is often the case in East Asia.

In any case, translation phase 1 begins with an implementation-defined mapping; and such mapping can choose to implement model A or C (but the implementation must document it). As a by-product, a strictly conforming program cannot rely on the specifics handled differently by the three models: examples of non-strict conformance include handling of shift states inside strings and calls like `fopen("\\ubeda\\file.txt","r")` and `#include "sys\udefault.h"`. Shift states are guaranteed to be handled correctly, however, as long as the implementation performs no mapping at the beginning of phase 1; and the two specific examples given above can be made much more portable by rewriting these as `fopen("\\" "ubeda\\file.txt", "r")` and `#include "sys/udefault.h"`.

Other Languages

Java 3.2

A translation of the Unicode escapes (3.3) in the raw stream of Unicode characters to the corresponding Unicode character . . .

ISO 646

Which means that characters other than those appearing in ISO/IEC 646 can appear in identifiers, strings and character constants, etc.

Common Implementations

This phase is where a translator interfaces with the host to read sequences of bytes from a source file. A source file is usually represented as a text file. There are a few hosts that have no concept of a text file. Some translators use relatively high-level system routines and rely on the host to return a line of characters; others perform block reads of binary data and perform their own character mappings. The choice will depend on the facilities offered by the host and the extent to which a translator wants to get involved in unpicking the format used to store characters in a source file. Some hosts treat text files (the usual method for storing source files) differently from binary files (lines are terminated and end-of-file may be indicated by a special character or trailing null characters).

There is no requirement that the file containing C source code have any particular form. Known forms include the following:

- *Stream of bytes.* Both text and binary files are treated as a linear sequence of bytes— the Unix model.
- *Text files have special end-of-line markers and end-of-file is indicated by a special character.* Binary files are treated as a sequence of bytes.
- *Fixed-length records.* These records can be either fixed-line length (a line cannot contain more than a given, usually 72 or 80, number of characters; dating back to when punch cards were the primary form of input to computers), or fixed-block length (i.e., lines do not extend over block boundaries and null characters are used to pad the last line in a block).

A translator that reads a block of characters at a time has to be responsible for knowing the representation of source files and may, or may not, have to perform some conversion to create an end-of-line indicator.[3]

Source files are usually represented in storage using the same set of byte values that are used by the translator to represent the source character set, so there is no actual mapping involved in many cases. The physical representation used to represent source files will be chosen by the tools used to create the source file, usually an editor.

The Unisys A Series[23] uses fixed-length records. Each record contains 72 characters and is padded on the right with spaces (no new-line character is stored). To represent logical lines that are longer than 72 characters, a backslash is placed in column 72 of the physical line, folding characters after the 71 onto the next physical line. A logical line that does end in a backslash character is represented in the physical line by two backslash characters.

The Digital Mars C[2] compiler performs special processing if the input file name ends in `.htm` or `.html`. In this case only those characters bracketed between the HTML tags <code> and </code> are considered significant. All other characters in the input file are ignored.

The IBM ILE C development environment[6] associates a Coded Character Set Identifier (CCSID) with a source physical file. This identifier denotes the encoding used, the character set identifiers, and other information. Files that are **`#include`**d may have different CCSID values. A set of rules is defined for how the contents of these include files is mapped in relation to CCSID of the source files that **`#include`**d them. A **`#pragma`** preprocessing directive is provided to switch between CCSIDs within a single source file; for instance:

```
1  char EBCDIC_hello[] = "Hello World";
2
3  /* Switch to ASCII character set. */
4  #pragma convert(850)
5  char ASCII_hello[] = "Hello World";
6
7  /* Switch back. */
8  #pragma convert(0)
```

Example

If the source contains:

$??)

and the translator is operating in a locale where $ and the immediately following character represent a single multibyte character. Then the input stream consists of the multibyte characters: $? ?)

In another locale the input stream might consist of the multibyte characters: $? ?) with the ??) being treated as a trigraph sequence and replaced by].

WG14/N770

Table 116.1: Total number of characters and new-lines in the visible form of the `.c` and `.h` files.

	`.c` files	`.h` files
total characters	192,165,594	64,429,463
total new-lines	6,976,266	1,811,790
non-comment characters	144,568,262	43,485,916
non-comment new-lines	6,113,075	1,491,192

---

117 Trigraph sequences are replaced by corresponding single-character internal representations.

trigraph sequences phase 1

Commentary

The replacement of trigraphs by their corresponding single-character occurs before preprocessing tokens are created. This means that the replacement happens for all character sequences, not just those outside of string literals and character constants.

trigraph sequences mappings
string literal syntax
integer character constant

Other Languages

Many languages are designed with an Ascii character set in mind, or do not contain a sufficient number of punctuators and operators that all characters not in a commonly available subset need to be used. Pascal specifies what it calls lexical alternatives for some lexical tokens.

Common Implementations

Studies of translator performance have shown that a significant amount of time is consumed by lexing characters to form preprocessing tokens.[26] In order to improve performance for the average case (trigraphs are not frequently used), one vendor (Borland) wrote a special program to handle trigraphs. A source file that contained trigraphs first had to be processed by this program; the resulting output file was then fed into the program that implemented the rest of the translator.

Coding Guidelines

Because the replacement occurs in translation phase 1, trigraphs can have unexpected effects in string literals and character constants. Banning the use of trigraphs will not prevent a translator from replacing them if encountered in the source. Also, in string literal contexts the developers mind-set is probably not thinking of trigraphs, so such sequences are unlikely to be noticed anyway.

Sequences of ? characters may be needed within literals by the application. One solution is to replace the second of the ? characters by the escape sequence \?, unless a trigraph is what was intended.

Some guidelines suggest running translators in a nonstandard mode (some translators provide an option that causes trigraph sequences to be left unreplaced), if one exists, as a way of preventing trigraph replacement from occurring. Running a translator in a nonstandard mode is rarely a good idea; what of those developers who are aware of trigraphs and intentionally use them?

The use of trigraphs may overcome the problem of entering certain characters on keyboards. But visually they are not easily processed, or to be exact very few developers get sufficient practice reading trigraphs to be able to recognize them effortlessly. Digraphs were intended as a more readable alternative (the characters used are more effective memory prompts for recalling the actual character they represent; they are discussed elsewhere).

digraphs

Example

```
1  #include <stdio.h>
2
3  void f(void)
4  {
5  printf("??=");     /* Prints # */
6  printf("?" "?=");  /* Prints ??= */
7  printf("?\?=");    /* Prints ??= */
8  }
```

Usage

The visible form of the .c files contain 8 trigraphs (.h 0).

---

translation phase 2
physical source line
logical source line

2. Each instance of a backslash character (\) immediately followed by a new-line character is deleted, splicing physical source lines to form logical source lines. 118

Commentary

line splicing

This process is commonly known as *line splicing*. The preprocessor grammar requires that a directive exists on a single logical source line. The purpose of this rule is to allow multiple physical source lines to be spliced to form a single logical source line so that preprocessor directives can span more than one line. Prior to the introduction of string concatenation, in C90, this functionality was also used to create string literals that may have been longer than the physical line length, or could not be displayed easily by an editor.

Emailing source code is now common. Some email programs limit the number of characters on a line and will insert line breaks if this limit is exceeded. Human-written source might not form very long lines, but automatically generated source can sometimes contain very long identifier names.

C++

The first sentence of 2.1p2 is the same as C90.

The following sentence is not in the C Standard: 2.1p2

If, as a result, a character sequence that matches the syntax of a universal-character-name is produced, the behavior is undefined.

```
1  #include <stdio.h>
2
3  int \u1F\
4  5F;          // undefined behavior
5               /* defined behavior */
6  void f(void)
7  {
8  printf("\\u0123"); /* No UCNs. */
9  printf("\\u\
10 0123"); /* same as above, no UCNs */
11         // undefined, character sequence that matches a UCN created
12 }
```

Common Implementations

Some implementations use a fixed-length buffer to store logical source lines. This does not necessarily imply that there is a fixed limit on the maximum number of characters on a line. But encountering a line longer than the input buffer can complicate the generation of log files and displaying the input line with any associated diagnostics. Both quality-of-implementation issues are outside the scope of the standard.

Coding Guidelines

A white-space character is sometimes accidentally placed after a backslash. This can occur when source files are ported, unconverted between environments that use different end-of-line conventions; for instance, reading MS-DOS files under Linux. The effect is to prevent line splicing from occurring and invariably causes a translator diagnostic to be issued (often syntax-related). This is an instance of unintended behavior and no guideline recommendation is made.

The limit on the number of characters on a logical source line is very unlikely to be reached in practice and line splicing is rarely used outside of preprocessing directives. Existing source sometimes uses line splicing to create a string literal spanning more than one source code line. The reason for this usage often is originally based on having to use a translator that did not support string literal concatenation.

limit characters on line

Example

```
1  #include <stdio.h>
2
3  #define X          ??/
4            (1+1) /* ??/ -> \ */
5
6  extern int g\
7  l\
8  o\
9  b
10 ;
11
12 void f(void)
13 {
14 if (glob)
15    {
16    printf("Something so verbose we need\
17  to split it over more than one line\n");
18    printf ("Something equally verbose but at"
```

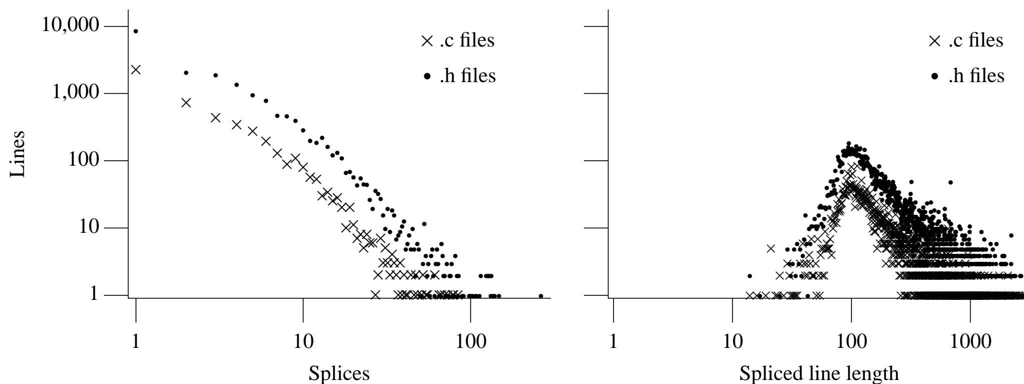

Figure 118.1: Number of physical lines spliced together to form one logical line and the number of logical lines, of a given length, after splicing. Based on the visible form of the `.c` and `.h` files.

```
19              " least we have some semblance of visual layout\n");
20     }
21
22  printf("\\u0123"); /* No UCNs. */
23  printf("\\
24  \u0123");          /* Same as above, no UCNs. */
25
26  printf("\\
27  \n");              /* No new-line output. */
28  }
```

Usage

In the visible form of the `.c` files 0.21% (`.h` 4.7%) of all physical lines are spliced. Of these line splices 33% (`.h` 7.8%) did not occur within preprocessing directives (mostly in string literals).

Only the last backslash on any physical source line shall be eligible for being part of such a splice. 119

Commentary

A series of backslash characters at the end of a line does not get consumed (assuming there are sufficient empty following lines). This is a requirement that causes no code to be written in the translator, as opposed to a requirement that needs code to be written to implement it.

C90

This fact was not explicitly specified in the C90 Standard.

C++

The C++ Standard uses the wording from C90.

Example

```
1   #include <stdio.h>
2
3   void f(void)
4   {
5   /*
6    * In the following the two backslash characters do not cause two
7    * line splices.  There is a single line splice.  This results in
8    * a single double-quote character, causing undefined behavior.
9    */
10  printf("\\
```

```
11
12  n");
13  /*
14   * The above is not equivalent to printf("n");
15   */
16
17  /*
18   * Below the backslash characters are on different lines.
19   */
20  printf("\
21  \
22  n");
23  }
```

footnote 5

120 5) Implementations shall behave as if these separate phases occur, even though many are typically folded together in practice.

Commentary

The phases of translation describe an intended effect, not an implementation strategy. It is not expected that implementations implement the different phases via different programs.

Common Implementations

translation technology

Many translators do split up the job of translation between a number of programs. Typically one program performs preprocessing (translation phases 1–4), while another performs syntax, semantic analysis, and code generation (the last operation is not directly mentioned in the standard). The usual method of communicating between the two programs is via intermediate files. If the original source file has the name `f.c` and translator options are used to save the output of various translation phases, the file holding the preprocessed output is normally given the name `f.i` and the file holding any generated assembler code is given the name `f.s`. Phase 8 is nearly always performed by a separate program (that can usually also handle languages other than C), a linker.

A compiler sold by Borland included a separate program to handle trigraphs (the programs handling other phases of translation did not include code to process trigraphs).

At least one program, `lcc`,[4] effectively only performs phase 7. It requires a third-party program to perform the earlier and later phases. The method of communication between phases is a file containing a sequence of characters that look remarkably like a preprocessed file (so `lcc` has to retokenize its input).

source file representation

121 Source files, translation units and translated translation units need not necessarily be stored as files, nor need there be any one-to-one correspondence between these entities and any external representation.

Commentary

The term *file* has a common usage within computing and the term *source file* could be interpreted to imply that source code had to be stored in such files. While source files are commonly represented using a text file within a host file system there is no requirement to use such a representation.

A translator may chose to internally maintain information about the effect of including a system header (e.g., an internal symbol table of declared identifiers) that is accessed when the corresponding **`#include`** is encountered. In such an implementation there is no external representation of the system header.

This sentence was added by the response to DR #308.

Other Languages

Languages which are defined by a written specification do not usually require that a particular external representation be used for source files. Languages defined by a particular implementation (e.g., PERL) require a source file representation that can be handled by that implementation.

Common Implementations

header precompiled

Some implementations support what are known as *precompiled headers*.[8,13] The contents of such headers have a form that has been partially processed through some phases of translation. The benefit of using precompiled headers is a, sometimes dramatic, improvement in rate of translation (figures of 20–70% have been reported).

IDE

Some software development environments (often called *IDE*s', Integrated Development Environments) hold the source code within some form of database. This database often includes version-control information, translator options, and other support information.

The description is conceptual only, and does not specify any particular implementation. 122

Commentary

as-if rule

The term *as-if rule* (or sometimes *as-if principle*) occurs frequently in discussions involving the C Standard. This term is not defined in the C Standard, but is mentioned in the Rationale:

Rationale

> The as if principle is invoked repeatedly in this Rationale. The C89 Committee found that describing various aspects of the C language, library, and environment in terms of concrete models best serves discussion and presentation. Every attempt has been made to craft the models so that implementors are constrained only insofar as they must bring about the same result, as if they had implemented the presentation model; often enough the clearest model would make for the worst implementation.
>
> A question sometimes asked regarding optimization is, "Is the rearrangement still conforming if the precomputed expression might raise a signal (such as division by zero)?" Fortunately for optimizers, the answer is "Yes," because any evaluation that raises a computational signal has fallen into an undefined behavior (§6.5), for which any action is allowable.

Essentially, a translator is free to do what it likes as long as the final program behaves, in terms of visible output and effects, as-if the semantics of the abstract machine were being followed. In some instances the standard calls out cases based on the as-if rule.

expression need not evaluate part of

This sentence was added by the response to DR #308.

C++

1.9p1

> *In particular, they need not copy or emulate the structure of the abstract machine. Rather, conforming implementations are required to emulate (only) the observable behavior of the abstract machine as explained below.5)*

Footnote 5

> *This provision is sometimes called the "as-if" rule, because an implementation is free to disregard any requirement of this International Standard as long as the result is as if the requirement had been obeyed, as far as can be determined from the observable behavior of the program. For instance, an actual implementation need not evaluate part of an expression if it can deduce that its value is not used and that no side effects affecting the observable behavior of the program are produced.*

---

source file end in new-line

A source file that is not empty shall end in a new-line character, which shall not be immediately preceded by a backslash character before any such splicing takes place. 123

Commentary

What should the behavior be if the last line of an included file did not end in a new-line? Should the characters at the start of the line following the **#include** directive be considered to be part of any preceding preprocessing token (from the last line of the included file)? Or perhaps source files should be treated as containing an implicit new-line at their end. This requirement simplifies the situation by rendering the behavior undefined.

125 source file partial preprocessing token

Lines are important in preprocessor directives, although they are not important after translation phase 4. Treating two apparently separate lines, in two different source files, as a single line opens the door to a great deal of confusion for little utility.

C90

The wording, " . . . before any such splicing takes place.", is new in C99.

Coding Guidelines

While undefined behavior will occur for this usage, instances of it occurring are so rare that it is not worth creating a coding guideline recommending against its use.

Example

```
1  /*
2   * If this source file is #include'd by another source file, might
3   * some implementation splice its first line onto the last line?
4   */
5  void f(void)
6  {
7  }\
```

---

124 3. The source file is decomposed into preprocessing tokens[6)] and sequences of white-space characters (including comments).

translation phase 3

Commentary

Preprocessing tokens are created before any macro substitutions take place. The C preprocessor is thus a token preprocessor, not a character preprocessor. The base document was not clear on this subject and some implementors interpreted it as defining a character preprocessor. The difference can be seen in:

EXAMPLE tokenization
base document

```
1  #define a(b) printf("b=%d\n", b);
2
3  a(var);
```

The C preprocessor expands the above to:

```
1  printf("b=%d\n", var);
```

while a character preprocessor would expand it to:

```
1  printf("var=%d\n", var);
```

Linguists used the term *lexical analysis* to describe the process of collecting characters to form a word before computers were invented. This term is used to describe the process of building preprocessing tokens and in C's case would normally be thought to include translation phases 1–3. The part of the translator that performs this role is usually called a *lexer*. As well as the term *lexing*, the term *tokenizing* is also used.

Common Implementations

Decomposing a source file into preprocessing tokens is straight-forward when starting from the first character. However, in order to provide a responsive interface to developers, integrated development environments often perform incremental lexical analysis[24] (e.g., only performing lexical analysis on those characters in the source that have changed, or characters that are affected by the change).

Coding Guidelines

The term *preprocessing token* is rarely used by developers. The term *token* is often used generically to apply to such entities in all phases of translation.

Usage

The visible form of the `.c` files contain 30,901,028 (`.h` 8,338,968) preprocessing tokens (new-line not included); 531,677 (`.h` 248,877) `/* */` comments, and 52,531 (`.h` 27,393) `//` comments.

preprocessing tokens white space separation

Usage information on white space is given elsewhere.

---

source file partial preprocessing token

A source file shall not end in a partial preprocessing token or in a partial comment. 125

Commentary

What is a partial preprocessing token? Presumably it is a sequence of characters that do not form a preprocessing token unless additional characters are appended. However, it is always possible for the individual characters of a multiple-character preprocessing token to be interpreted as some other preprocessing token (at worst the category “each non-white-space character that cannot be one of the above” applies).

preprocessing token syntax

For instance, the two characters `..` (where an additional period character is needed to create an ellipsis preprocessing token) represents two separate preprocessing tokens (e.g., two periods). The character sequence `%:%` represents the two preprocessing tokens `#` and `%` (rather than `##`, had a `:` followed).

The intent is to make it possible to be able perform low-level lexical processing on a per source file basis. That is, an **`#include`**d file can be lexically analyzed separately from the file from which it was included. This means that developers only need to look at a single source file to know what preprocessing tokens it contains. It can also simplify the implementation.

source file end in new-line 123

The requirement that source files end in a new-line character means that the behavior is undefined if a line (physical or logical) starts in one source file and is continued into another source file.

In this phase a comment is an indivisible unit. A source file cannot contain part of such a unit, only a whole comment. That is, it is not possible to start a comment in one source file and end it in another source file.

Coding Guidelines

Translators are not required to diagnose a comment that starts in one source file and ends in another. However, this usage is very rare and consequently a guideline recommendation is not cost effective.

---

comment replaced by space

Each comment is replaced by one space character. 126

Commentary

The C committee had to choose how many space characters a comment was converted into, including zero. The zero case had the disadvantage of causing some surprising effects, although some existing implementations had gone down this route. They finally decided that specifying more than a single space character was of dubious utility. Replacing a comment by one space character minimizes the interaction between it and the adjacent preprocessing tokens (space characters are not usually significant).

white space significant

Other Languages

Java supports the `/* */` form of comments. The specification does not say what they get converted into.

Common Implementations

base document

The base document replaced a comment by nothing (some implementations continue to support this functionality for compatibility with existing code[5,21]). This had the effect of treating:

```
1  int a/* comment */b;
```

as a declaration of the identifier ab. The C Committee introduced the ## operator to explicitly provide this functionality.

operator

Example

```
1  #define mkstr(a) #a
2
3  char *p = mkstr(x/* a comment*/y); /* p points at the string literal "x y" */
```

---

127 New-line characters are retained.

Commentary

New-line is a token in the preprocessor grammar. It is used to delimit the end of preprocessor directives.

Other Languages

New-line is important in several languages. Some older languages, and a few modern ones, have given meaning to new-line. Fortran (prior to Fortran 90) was not free format; the end of a line is the end of a statement or declaration, unless a line-continuation character appears in column 5 of the following line.

---

128 Whether each nonempty sequence of white-space characters other than new-line is retained or replaced by one space character is implementation-defined.

white-space sequence replaced by one

Commentary

In this phase the only remaining white-space characters that need to be considered are those that occur between preprocessing tokens. All other white-space characters will have been subsumed into preprocessing tokens. White-space characters only have significance now when preprocessing tokens are glued together, or as a possible constraint violation (i.e., vertical-tab or form-feed within a preprocessing directive).

white space between macro argument tokens

white-space within preprocessing directive

Other Languages

Some languages treat the amount of white space at the start of a line as being significant (make requires a horizontal tab at the beginning of a line in some contexts[22]). In Fortran (prior to Fortran 90) statements were preceded by six space characters (five if a line continuation was being used, or a comment). In Occam statement indentation is used to indicate the nesting of blocks.

Common Implementations

Most implementations replace multiple white-space characters by one space character. The existence, or not, of white-space separation can be indicated by a flag associated with each preprocessing token, preceded by space.

Integrated development environments vary in their handling of white-space. Some only allow multiple white-space characters, between tokens, at the start of a line, while others allow them in any context. White-space characters introduce complexity for tools' vendors[25] that is not visible to the developer.

Coding Guidelines

Sequences of more than one white-space character often occur at the start of a line. They also occur between tokens forming a declaration when developers are trying to achieve a particular visual layout. However, white-space can only make a difference to the behavior of a program, outside of the contents of a character constant or string literal, when they appear in conjunction with the stringize operator.

operator

Example

```
1  #define mkstr(a) #a
2
3  char *p = mkstr(2  []; /* p may point at the string "2  [", or "2 [" */
4  char *q = mkstr(2[);   /* q points at the string "2[" */
```

---

translation phase 4

4. Preprocessing directives are executed, macro invocations are expanded, and **_Pragma** unary operator expressions are executed. 129

Commentary

This phase is commonly referred to as *preprocessing*. The various special cases in previous translation phases do not occur often, so they tend to be overlooked.

Although the standard uses the phrase *executed*, the evaluation of preprocessor directives is not dynamic in the sense that any form of iteration, or recursion, takes place. There is a special rule to prevent recursion from occurring. The details of macro expansion and the **_Pragma** unary operator are discussed elsewhere.

macro being replaced found during rescan
macro replacement
_Pragma operator

C90

Support for the **_Pragma** unary operator is new in C99.

C++

Support for the **_Pragma** unary operator is new in C99 and is not available in C++.

Other Languages

PL/1 contained a sophisticated preprocessor that supported a subset of the expressions and statements of the full language. For the PL/1 preprocessor, *executed* really did mean executed.

Common Implementations

The output of this phase is sometimes written to a temporary source file to be read in by the program that implements the next phrase of translation.

If a character sequence that matches the syntax of a universal character name is produced by token concatenation (6.10.3.3), the behavior is undefined. 130

Commentary

The C Standard allows UCNs to be interpreted and converted into internal character form either in translation phase 1 or translation phase 5 (the C committee could not reach consensus on specifying only one of these). If an implementation chooses to convert UCNs in translation phase 1, it makes no sense to require them to perform another conversion in translation phase 4. This behavior is different from that for other forms of preprocessing tokens. For instance, the behavior of concatenating two integer constants is well defined, as is concatenating the two preprocessing tokens whose character sequences are 0x and 123 to create a hexadecimal constant.

universal character name syntax

The intent is that universal character names be used to create a readable representation of the source in the native language of the developer. Once this phase of translation has been reached, the sequence of characters in the source code needed to form that representation are not intended to be manipulated in smaller units than the universal character name (which may have already been converted to some other internal form).

C90

Support for universal character names is new in C99.

C++

translation phase 1 116

In C++ universal character names are only processed during translation phase 1. Character sequences created during subsequent phases of translation, which might be interpreted as a universal character name, are not interpreted as such by a translator.

Coding Guidelines

Support for universal character names is new and little experience has been gained in the kinds of mistakes developers make in its usage. It is still too early to know whether a guideline recommending "A character sequence that matches the syntax of a universal character name shall not be produced by token concatenation." is worthwhile.

Example

```
1   #define glue(a, b) a ## b
2
3   int glue(\u1, 234);
```

---

131 A **#include** preprocessing directive causes the named header or source file to be processed from phase 1 through phase 4, recursively.

Commentary

The **#include** preprocessing directives are processed in-depth first order. Once an **#include** has been fully processed, translation continues in the file that **#include**d it. There is a limit on how deeply **#include**s can be nested. Just prior to starting to process the **#include**, the macros `__FILE__` and `__LINE__` are set (and they are reset when processing resumes in the file containing the **#include**).

limit #include nesting

__FILE__ macro

__LINE__ macro

The effect of this processing is that phase 5 sees a continuous sequence of preprocessing tokens. These preprocessing tokens do not need to maintain any information about the source file that they originated from.

---

132 All preprocessing directives are then deleted.

preprocessing directives deleted

Commentary

This requirement was not explicitly specified in C90. In practice it is what all known implementations did. Also, macro definitions have no significance after translation phase 4.

macro definition no significance after

preprocessor directives syntax

Preprocessing directives have their own syntax, which does not connect to the syntax of the C language proper. The preprocessing directives are used to control the creation of preprocessing tokens. These are handed on to subsequent phases; they don't get past phase 4.

C++

This explicit requirement was added in C99 and is not stated in the C++ Standard.

Common Implementations

Many implementations write the output of the preprocessor to a file to be read back in by the next phase. Ensuring that declarations and statements retain the same line number as they had in the source file allows more specific diagnostic messages to be produced. To achieve this effect, some implementations convert preprocessing directives to blank lines, while others insert **#line** directives.

---

133 5. Each source character set member and escape sequence in character constants and string literals is converted to the corresponding member of the execution character set;

translation phase 5

Commentary

The execution character set is used by the host on which the translated program will execute. Up until this phase characters have been represented using the physical-to-source character set mapping that occurred in translation phase 1.

execution character set represented by

116 translation phase 1

The translation and execution environments may, or may not, use the same values for character set members. This conversion relies on implementation-defined behavior. In the case of escape sequences, an implementation is required to use the value specified (provided this value is representable in the type **char**).

character constant mapped

escape sequence octal digits

escape sequence hexadecimal digits

For instance, the equality '\07'==7 is true, independent of whether any member of the basic execution character set maps to the value seven. This issue is discussed in DR #040q8.

Common Implementations

Probably the most commonly used conversion uses the values specified in the Ascii character set. Some mainframe hosted implementations continue to use EBCDIC.

Coding Guidelines

footnote 141

Differences between the values of character set members in the translation and execution environments become visible if a relationship exists between two expressions, one appearing in a **#if** preprocessing directive and the other as a controlling expression. This issue is discussed elsewhere.

Example

In the following:

```
1  void f(void)
2  {
3  char ch = 'a';
4  char dollar = '$';
5  }
```

the integer character constant **'a'** is converted to its execution time value. It may have an Ascii value in the source character set and an EBCDIC value in the execution character set. The $ character will have been mapped to the source character set in translation phase 1. In the following example it can be mapped back to the $ character if the implementation so chooses.

```
1  #if ('a' == 97) && ('Z' == 90)
2  #define PREPROCESSOR_USES_ASCII 1
3  #else
4  #define PREPROCESSOR_USES_ASCII 0
5  #endif
6
7  _Bool ascii_execution_character_set(void)
8  {
9  return ('a' == 97) && ('Z' == 90);
10 }
```

---

correspond-ing member if no

if there is no corresponding member, it is converted to an implementation-defined member other than the null (wide) character.[7)] 134

Commentary

All source character set members and escape sequences, which have no corresponding execution character set member, may be converted to the same member, or they may be converted to different members. The behavior is required to be documented.

The null character is special in that it is used to terminate string literals. It is possible for a string literal to contain a null character through the use of an escape sequence, but such an occurrence has to be explicitly created by the developer. It is never added by the implementation.

C90

ISO 10646

The C90 Standard did not contain this statement. It was added in C99 to handle the fact that the UCN notation supports the specification of numeric values that may not represent any specified (by ISO 10646) character.

C++

2.2p3

The values of the members of the execution character sets are implementation-defined, and any additional members are locale-specific.

C++ handles implementation-defined character members during translation phase 1.

116 translation phase 1

Common Implementations

Most implementations simply convert escape sequences to their numerical value. There is no check that the value maps to a character in the execution character set. Characters in the source character set, which are not in the execution character set, are often mapped to the value used to represent that value in the translation environment.

Coding Guidelines

Why would source code contain an instance of a source character or escape sequence that did not have a corresponding member in the execution character set? The usage could be because of a fault in the program (and therefore outside the scope of these coding guidelines), or existing source is being ported to a new environment that does not support it.

Use of members of an extended character set is by its very nature dependent on particular environments. Porting source that contains such character usage does not fall within the scope of these guidelines.

extended character set

Example

```
1  char g_c = '$';
2  char my_address = "derek@99.C.ISO.Earth";
3
4  char e_c = '\007';
5
6  #if '@' == 64
7  char *char_set = "ASCII";
8  #else
9  char *char_set = "EBCDIC ;-)";
10 #endif
```

---

135 6. Adjacent string literal tokens are concatenated.

translation phase 6

Commentary

This concatenation only applies to string literals. It is not a general operation on objects having an array of **char**. String literal preprocessing tokens do not have a terminating null character. That is added in the next translation phase.

C90

6. Adjacent character string literal tokens are concatenated and adjacent wide string literal tokens are concatenated.

It was a constraint violation to concatenate the two types of string literals together in C90. Character and wide string literals are treated on the same footing in C99.

The introduction of the macros for I/O format specifiers in C99 created the potential need to support the concatenation of character string literals with wide string literals. These macros are required to expand to character string literals. A program that wanted to use them in a format specifier, containing wide character string literals, would be unable to do so without this change of specification.

Other Languages

Most languages that support string concatenation require that the appropriate operator be used to specify an operation to be performed. For instance, Ada uses the character **&**.

Coding Guidelines

There is the possibility, in an initializer or argument list, that a missing comma causes two unrelated string literals to be concatenated. However, this is a fault and considered to be outside the scope of these coding guidelines.

guidelines not faults

Example

```
1  #include <stdio.h>
2
3  /*
4   * Assign the same value, L"ab", to three different objects.
5   */
6  wchar_t *w_p1 = L"a" L"b",
7          *w_p2 = L"a"  "b",
8          *w_p3 = "a" L"b";
9
10 void f(void)
11 {
12 printf("\\" "066"); /* Output \066, escape sequences were processed earlier. */
13 }
```

Usage

In the visible form of the `.c` files 4.9% (`.h` 15.6%) of string literals are concatenated.

---

translation phase 7

7. White-space characters separating tokens are no longer significant. 136

Commentary

replacement list identical if

White-space is not part of the syntax of the C language. It is only significant in separating characters in the lexical grammar and in some contexts in the preprocessor. This statement could have equally occurred in translation phase 5.

Other Languages

Fortran spaces not significant

In Fortran white space is never significant.

Example

```
1  #define mkstr(x) #x
2
3  char *a_space_plus_b = mkstr(a +b);
4  char *a_plus_b = mkstr(a+b);
```

---

preprocessing token converted to token

Each preprocessing token is converted into a token. 137

Commentary

#if identifier replaced by 0

These are the tokens seen by the C language parser (the same conversion also occurs within **#if** preprocessing directives). It is possible for a preprocessing token to not be convertible to a token. For instance, in:

```
1  float f = 1.2.3.4.5;
```

`1.2.3.4.5` is a valid preprocessing token; it is a pp-number. However, it is not a valid token.

pp-number syntax

Preprocessing tokens that are skipped as part of conditional compilation need never be converted to tokens (because they never make it out of translation phase 4).

```
1  #if 0
2  float f = 1.2.3.4.5; /* Never converted. */
3  #endif
```

A preprocessing token that cannot be converted to a token is likely to cause a diagnostic to be issued. At the very least, there will be a syntax violation. It is a quality-of-implementation issue as to whether the translator issues a diagnostic for the failed conversion.

Other Languages

Few other language contain a preprocessor. Any problems associated with creating a token are directly caused by the sequence of input characters.

---

138 The resulting tokens are syntactically and semantically analyzed and translated as a translation unit.

syntactically analyzed

Commentary

This is what the bulk of the standard's language clause is concerned with.

Common Implementations

This is the phase where the executable code usually gets generated.

Luo, Chen, and Yu[14] found that an equation of the form L^x, where L is the number of bytes in the source file (i.e., header file contents are not included) and x some constant, provided a good estimate for the elapsed time needed to translate a source file (an x value of approximately 0.66 was found for GCC 3.4.2) and to link it (an x value of approximately 0.1 was found for GCC).

source files

Coding Guidelines

Coding guidelines are not just about how the semantics phase of translation processes its input. Previous phases of translation are also important, particularly preprocessing. The visible source, the input to translation phase 1, is probably the most important topic for coding guidelines.

---

139 8. All external object and function references are resolved.

translation phase 8

Commentary

This resolution is normally carried out by a program commonly known as a linker. Although the standard says nothing about what such resolution means its common usage (in linker terminology) is that references to declarations are made to refer to their corresponding definitions.

C++

The C translation phase 8 is numbered as translation phase 9 in C++ (in C++, translation phase 8 specifies the instantiation of templates).

Common Implementations

The code generated for a single translation unit invariably contains unresolved references to external objects and functions. The tool used to resolve these references, a linker, may be provided by the implementation vendor or it may have been supplied as part of the host environment.

Resolving references that involve addresses larger than the length of a machine code instruction can lead to inefficiencies and wasted space. For instance, generating a 32-bit address using instructions that have a maximum length of 16 bits requires at least three instructions (some bits are needed to specify what the instruction does; for instance, load constant). In practice most addresses do not require a full 32-bit value, but this information is not available until link-time, while translators have been forced to make worst-case assumptions. Many processors solve this problem by having variable-length instructions.

The simplifications derived from the principles behind RISC mean that their instructions have a fixed length. It is possible to handle 32 bit addresses using a 32-bit instruction width by appropriate design conventions (e.g., the call instruction encoded in 2 bits, leaving 30 bits for addresses and requiring that functions start on a 4-byte address boundary). The move to a 64-bit address space reintroduces problems seen in the 16/32-bit era two decades earlier. However, processor performance and storage capabilities have moved on and link-time optimizations are now practical. In particular the linker knows the values of addresses and can generate the minimum number of instructions needed.[20]

The quest for higher-quality machine code has led research groups to look at link-time optimizations.[27] Having all of the components of a program available opens up opportunities for optimizations that are not available when translating a single translation unit. A study by Muth, Debray, Watterson, and De Bosschere[15] found that on average 18% of a programs instructions could have their operands and result determined at link time. This does not imply that 18% of a program's instructions could be removed; knowing information provides opportunities for optimization, like replacing an indirect call with a direct one.

Levine[12] discusses the principles behind linking object files to create a program image and loading that program image, into storage, prior to executing it.

Coding Guidelines

identifier declared in one file ??

The problem of ensuring that the same identifier, declared in different translation units, always resolves to a definition having the same type is discussed elsewhere.

---

140 Library components are linked to satisfy external references to functions and objects not defined in the current translation.

Commentary

translation units preserved

The term *library components* is a broad one and can include previously translated translation units. The implementation is responsible for linking in any of its provided C Standard library functions, if needed.

The wording of this requirement can be read to imply that all external references must be satisfied. This would require definitions to exist for objects and functions, even if they were never referenced during program execution (but were referenced in the translated source code). This is the behavior enforced by most linkers.

undefined behavior indicated by

Since there is no behavior defined for the case where a reference cannot be resolved, it is implicitly undefined behavior.

Other Languages

translation unit syntax

This linking phase is common to many implementations of most programming languages that support some form of separate compilation. In a few cases the linking process can occur on an as-needed basis during program execution, as for instance in Java.

Common Implementations

linkers

Most linkers assume that all of the definitions in the developer-supplied list of object files, given to the linker, are needed, but that definitions from the implementation system libraries only need be included if they are referenced from the developer-written code.

Optimizing linkers build a function call tree, starting from the function `main`. This enables them to exclude functions, supplied by the developer, from the program image that are never referenced during execution.

The two commonly used linking mechanisms are static and dynamic linking. Static linking creates a program image that contains all of the function and object definitions needed to execute the program. A statically linked program image has no need for any other library functions at execution time. This method of linking is suitable when distributing programs to host environments where very little can be assumed about the availability of support libraries. If many program images are stored on a host, this method has the disadvantage of keeping a copy of library functions that are common to a lot of programs. Also if any libraries are updated, all programs will need to be relinked.

Dynamic linking does not copy library functions into the program image. Instead a call to a dynamic linking system function is inserted. When a dynamically linked function is first called, the call is routed

via the dynamic link loader, which resolves the reference to the desired function and patches the executing program so that subsequent calls jump directly to that routine. Provided there is support from the host OS, all running programs can access a single copy of the library functions in memory. On hosts with many programs running concurrently, this can have a large impact on memory performance (reduced swapping). Any library updates will be picked up automatically. The program image lets the dynamic loader decide which version of a library to call.

An implementation developed by Neamtiu, Hicks, Stoyle and Oriol[16] supports what they call *dynamic software updating*, whereby an executing program can be patched at a fine level of granularity, e.g., a modified version of a function (the parameter and return types have to be compatible) can replace the one currently in the executing program image.

Coding Guidelines

The standard does not define the order in which separately translated translation units and implementation supplied libraries are scanned to resolve external references. Many implementations scan files in the order in which they are given to the linker.

Most implementations define additional library functions. The names of such functions rarely follow the conventions, defined in the C Standard, for implementation-defined names. There is the possibility that one of these names will match that of an externally visible, developer-written function definition. Placing system libraries last on the list to be processed helps to ensure that any developer-provided definitions, whose names match those in system libraries, are linked in preference to those in the system library. Of course there may then be the problem of the system library referencing the incorrect definition.

Linking is often an invisible part of building the program image (diagnostics are sometimes issued for multiply defined identifiers). Checking which references have been resolved to which definitions is often very difficult. In the case of an incorrect function definition being used, a set of tests that exercise all called functions is likely to highlight any incorrect usage. In the case of incorrect objects, things might appear to work as expected. A guideline recommendation dealing with the incorrect definition being used to build the program image sounds worthwhile. However, this usage is likely to be unintended (a fault) and these coding guidelines are not intended to recommend against the use of constructs that are obviously faults.

guidelines not faults

---

141 All such translator output is collected into a program image which contains information needed for execution in its execution environment.

program image

Commentary

This wording implies a single location, perhaps a file (containing all of the information needed) or a directory (potentially containing more than one file with all of the necessary information). Several requirements can influence how a program image is built, including the following:

- *Developer convenience.* Like all computer users, developers want things to be done as quickly as possible. During the coding and debugging of a program, its image is likely to be built many times per hour. Having a translator that builds this image quickly is usually seen as a high priority by developers.
- *Support for symbolic debugging.* Here information on the names of functions, source line number to machine code offset correspondence, the object identifier corresponding to storage locations, and other kinds of source-related mappings needs to be available in the program image. Requiring that it be possible to map machine code back to source code can have a significant impact on the optimizations that can be performed. Research on how to optimize and include symbolic debugging information has been going on for nearly 20 years. Modern approaches[29] are starting to provide quality optimizations while still being able to map code and data locations.
- *Speed of program execution.* Users of applications want them to execute as quickly as possible. The organization of a program image can have a significant impact on execution performance.

- *Hiding information.* Information on the internal workings of a program may be of interest to several people. The owner of the program image may want to make it difficult to obtain this information from the program image.[28]
- *Distrust of executable programs.* Executing an unknown program carries several risks: It may contain a virus, it may contain a trojan that attempts to obtain confidential information, it may consume large amounts of system resources or a variety of other undesirable actions. The author of a program may want to provide some guarantees about a program, or some mechanism for checking its integrity. There has been some recent research on translated programs including function specification and invariant information about themselves, so-called proof carrying programs.[17] Necula's proposed method includes a safety policy and operating in the presence of untrusted agents. On the whole the commonly used approach is for the host environment to ring fence an executing program as best it can, although researchers have started to look at checking program images[30] before loading them, particularly into a trusted environment.

Other Languages

Most language specifications are silent on the issue of building program images. The Java specification explicitly specifies a possible mechanism supporting a distributed, across separate files, program image.

Common Implementations

Some program images contain all of the necessary machine code and data; they don't reference other files containing additional definitions— static linking. While other program images copy the object code derived from developer-written code in a single file, which also contains references to system-supplied definitions that need to be supplied during execution (held in other files)— dynamic linking.

The file used to hold a program image usually has some property to indicate its status as an executable program. Under MS-DOS the name the file extension is used (`.exe` or `.com`). Under Unix the list of possible file attributes, held by the file system, includes a flag to indicate an executable program.

The information in a program image may be laid out so that it can be copied directly into memory, as is. For instance, a MS-DOS `.com` file is completely copied into memory, starting at address 0x100. A Unix ELF (Executable and Linking Format) file contains segments that are copied to memory, as is (additional housekeeping information is also held in the file). Arranging for a program image to contain an exact representation of what needs to be held in memory has the advantage of simplifying swapping (assuming that the host performs this relatively high-level memory-management function) of unused code. It can simply be read back in from the program image; it does not have to be copied from memory to a swap partition and read back in.

On some hosts the layout of the program image, on a storage device, is different from its execution-time layout in storage. In this case the information in the program image is used to construct the execution-time layout when the program is first invoked. Such image formats include COFF (Common Object File Format) under Unix, MS-DOS `.exe` files and the IBM 360 object format (based on fixed-length records of 80 characters).

The ANDF (Architecture Neutral Distribution Format) project, http://www.tendra.org, aimed to produce a program image that could be installed on a variety of different hosts. The image contained a processed form of the original source code. An installer, running on a particular host, took this intermediate program image and a platform-specific program image (containing the appropriate machine code) from it, thus installing the program on that host. Installers were produced for a number of different host processors. The technology has not thrived commercially and much of the source code has now been made publicly available.

The performance of an application during certain parts of its execution is sometimes more important than during other parts. Program startup is often one such instance. Users of applications often want them to start as quickly as possible. How the program image is organized can affect the time taken for a program to start executing. One study by Lee, Crowley, Baer, Anderson, and Bershad[9] found that program startup

only required a subset of the information held in the program image. They also found that in some cases applications had scattered this information over the entire program image, requiring significantly more information to be read than was required for startup. They showed that by reordering how the program image was structured it was possible to reduce application startup latency by more than 58% (see Table 141.1).

Table 141.1: *Total* is the number of code pages in the application; *Touched* the number of code pages touched during startup; *Utilization* the average fraction of functions used during startup in each code page. Adapted from Lee.[9]

Application	Total	Touched (%)	% Utilization
acrobat	404	246 (60)	28
netscape	388	388 (100)	26
photoshop	594	479 (80)	28
powerpoint	766	164 (21)	32
word	743	300 (40)	47

Program images contain sequences of machine code instructions that are executed by the host processor. In some application domains the instructions making up a program can account for a large percentage of memory usage. Several techniques are available to reduce this overhead: Compression techniques can be applied to the sequence of instructions, or function abstraction can be applied.

instruction compression

A variety of instruction compression schemes have been proposed. The optimal choice can depend on the processor instruction format (RISC processors with fixed-length instructions, or DSP processors using very wide instruction words) and whether decompression is supported in hardware or software.[10] Compression ratios in excess of 50% are commonly seen. Performance penalties on decompression range from a few percent to 1,000% (for software decompression). At the time of this writing IBM's PowerPC 405 is the only commercially available processor that makes use of hardware compression (the CodePack algorithm). A good discussion of the issues and experimental results can be found in[11]

On early Acorn RISC Unix machines, the bottleneck in loading a program image was the disk drive. Analysis showed that it was quicker to load a smaller image and decompress it, on the fly, than to fetch the extra disk blocks with no decompression (a simple compression program was included with each system). The encoding of the ARM machine code created opportunities for compression. For instance, the first 4 bits of each opcode are a conditional. Most instructions are always executed and these four bits contained the value 0xE; a value of 0xF meant *never* and rarely occurred in the image.

Another approach to reducing the memory footprint of a program is to use compression on a function-by-function basis. Kirovski[7] reported an average memory reduction of 40% with a runtime performance overhead of 10% using a scheme that held uncompressed functions in a cache; a call to a compressed function causes it to be uncompressed and placed in the cache, possibly removing the uncompressed code for a function already in the cache.

The ordering of functions within the program image can also affect performance by affecting what is loaded into any instruction cache. In one study[19] it was found that in an Oracle database running an online transaction-processing application (with an instruction footprint of about 1 M byte) 25% to 30% of the execution time was caused by instruction stalls. By reordering functions in memory (dynamically, during program execution, based on runtime behavior), it was possible to improve performance by 6% to 10%[19].

basic block

function abstraction

Function abstraction (sometimes called *procedure abstraction*) is the process of creating functions from sequences of machine instructions. A call to this translator-created function replaces the sequence of instructions that it duplicates. On its own this process would slow program execution and make the program image larger. However, if the same sequence of instructions occurs at other places in the program, they can also be replaced by a call. Provided the sequence being replaced is larger than the substituted call instruction, the program image will shrink in size. Zastre[31] provides a discussion of the issues.

For a collection of embedded applications, Cooper and McIntosh[1] were able to reduce the number of instructions in the program image by 4.88%, with a corresponding increase of executed instructions of 6.47%.

Using profile information, they were able to reduce the dynamic overhead to 0.86%, with a corresponding decrease in the static instruction count of 3.89%.

Many optimization techniques will result in the same sequence of source statements being translated into different sequences of machine instructions. Context can be an important issue in optimization. Performing function abstraction prior to some optimizations can increase the number of duplicate sequences of instructions.[18]

Coding Guidelines

Use of static linking may be thought to ensure that program images continue to produce the same behavior when a host's libraries are updated. Experience shows that changes to these libraries can cause changes of behavior, particularly when low-level device drivers are involved.

Use of dynamic linking ensures that programs can take advantage of any bug fixed or performance improvements associated with updated libraries. Use of dynamically linked libraries also helps to ensure that all programs use the same libraries (reducing the likelihood of different programs failing to cooperate because of execution-time incompatibilities, for instance through use of a shared memory interface). If a required library is not available in the host environment, either it will not be possible to execute the program image or it will fail during execution.

The issue of static versus dynamic linking is a configuration-management issue and is outside the scope of these coding guidelines.

Forward references: universal character names (6.4.3), lexical elements (6.4), preprocessing directives (6.10), trigraph sequences (5.2.1.1), external definitions (6.9). 142

---

footnote 6

6) As described in 6.4, the process of dividing a source file's characters into preprocessing tokens is context-dependent. 143

Commentary

In only a few cases does a lexer, for C language, need information on the context in which it is encountering characters. The issue of whether an identifier is defined as a typedef or not is not applicable during the creation of preprocessing tokens; both have the same lexical syntax. However, the syntax used by most translators makes a distinction between identifiers and typedef names, so a symbol table lookup is needed when the preprocessing token is converted to a token (or at some other time, prior to translation phase 7).

Other Languages

In most languages no knowledge of context is needed to create tokens from a character sequence. It has become accepted practice in language design that it be possible to implement a lexer using a finite state automata. There are several tools that automatically produce lexers from a specification; for example, `flex` from the Open Software Foundation.

Fortran spaces not significant

In Fortran white space is not significant and in some cases it is necessary to scan ahead of the current input character to decide the lexical form of a given sequence of characters.

For example, see the handling of < within a **`#include`** preprocessing directive. 144

Commentary

header name recognized within #include

This issue is discussed elsewhere.

---

footnote 7

7) An implementation need not convert all non-corresponding source characters to the same execution character. 145

Commentary

An implementation may choose to convert all source characters without a corresponding member in the execution character set to a single execution character whose interpretation implies, for instance, that there was no corresponding member. Your author knows of no implementation that takes this approach.

corresponding member if no 134

C++

The C++ Standard specifies that the conversion is implementation-defined (2.1p1, 2.13.2p5) and does not explicitly specify this special case.

References

1. K. D. Cooper and N. McIntosh. Enhanced code compression for embedded RISC processors. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 139–149, 1999.
2. Digitalmars. Digitalmars C compiler. www.digitalmars.com, 2003.
3. C. W. Fraser and D. R. Hanson. A retargetable compiler for ANSI C. *SIGPLAN Notices*, 26(10):29–43, Oct. 1991.
4. C. W. Fraser and D. R. Hanson. *A Retargetable C Compiler: Design and Implementation*. Benjamin/Cummings Pub. Co., Redwood City, CA, USA, 1995.
5. HP. *DEC C Language Reference Manual*. Compaq Computer Corporation, aa-rh9na-te edition, July 1999.
6. IBM. *WebSphere Development Studio ILE C/C++ Programmer's Guide*. IBM Canada Ltd, Ontario, Canada, sc09-27 12-02 edition, May 2001.
7. D. Kirovski, J. Kin, and W. H. Mangione-Smith. Procedure based program compression. In *Proceedings of the* 30th *Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'97)*, pages 204–213. Association for Computing Machinery, 1997.
8. B. Koehler and R. N. Horspool. CCC: A caching compiler for C. *Software–Practice and Experience*, 27(2):155–165, 1997.
9. D. Lee, J.-L. Baer, B. Bershad, and T. Anderson. Reducing startup latency in web and desktop applications. Technical Report TR-99-03-01, University of Washington, Department of Computer Science and Engineering, Mar. 1999.
10. C. Lefurgy, E. Piccininni, and T. Mudge. Reducing code size with run-time decompression. In *Proceedings of the Sixth International Symposium on High-Performance Computer Architecture*, pages 218–227, Toulouse, France, Jan. 8–12, 2000. IEEE Computer Society TCCA.
11. C. R. Lefurgy. *Efficient Execution of Compressed Programs*. PhD thesis, University of Michigan, 2000.
12. J. R. Levine. *Linkers & Loaders*. Morgan Kaufmann Publishers, 2000.
13. A. Litman. An implementation of precompiled headers. *Software–Practice and Experience*, 23(3):341–350, 1993.
14. G. Luo, T. Chen, and H. Yu. Toward a progress indicator for program compilation. *Software–Practice and Experience*, 37(???):909–933, Apr. 2007.
15. R. Muth, S. Debray, S. Watterson, and K. de Bosschere. alto: A link-time optimizer for the DEC Alpha. Technical Report TR98-14, The Department of Computer Science, University of Arizona, Wednesday, Dec. 9 1998.
16. I. Neamtiu, M. Hicks, G. Stoyle, and M. Oriol. Practical dynamic software updating for C. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 72–83, June 2006.
17. G. C. Necula. *Compiling with Proofs*. PhD thesis, Carnegie Mellon University, Oct. 1998. Available as Technical Report CMU-CS-98-154.
18. J. Runeson. Code compression through procedure abstraction before register allocation. Thesis (m.s.), Uppsala University, Box 311, S-751 05 Uppsala, Sweden, 2000.
19. D. J. Scales. Efficient dynamic procedure placement. Technical Report Research Report 98/5, Compaq Western Research Laboratory, 1998.
20. A. Srivastava and D. W. Wall. Link-time optimization of address calculation on a 64-bit architecture. Technical Report Research Report 954/1, Western Research Laboratory - Compaq, Feb. 1994.
21. Sun. *C User's Guide*. Sun Microsystems, Inc, Palo Alto, CA, USA, revision a edition, May 2000.
22. S. Talbott. *Managing Projects with Make*. O'Reilly & Associates, Inc, 1989.
23. Unisys Corporation. *C Programming Reference Manual, Volume 1: Basic Implementation*. Unisys Corporation, 8600 2268-203 edition, 1998.
24. T. A. Wagner and S. L. Graham. General incremental lexical analysis. http://harmonia.cs.berkeley.edu/harmonia/papers/twagner-lexing.pdf, 1997.
25. T. A. Wagner and S. L. Graham. Modeling user-provided whitespace and comments in an incremental SDE. *Software–Practice and Experience*, ???(1):???, 1998.
26. W. M. Waite. The cost of lexical analysis. *Software–Practice and Experience*, 16(5):473–488, 1986.
27. D. W. Wall. Global register allocation at link time. Technical Report 86/3, Western Research Lab, Oct. 1986.
28. C. Wang, J. Hill, J. Knight, and J. Davidson. Software tamper resistance: Obstructing static analysis of programs. Technical Report CS-2000-12, University of Virginia, Dec. 2000.
29. L.-C. Wu, R. Mirani, H. Patil, B. Olsen, and W.-M. W. Hwu. A new framework for debugging globally optimized code. In *Proceedings of the SIGPLAN'99 Conference on Programming Language Design and Implementation*, 1999.
30. Z. Xu, T. Reps, and B. P. Miller. Typestate checking of machine code. In *European Symposium On Programming 2001*, 2001.
31. M. J. Zastre. Compacting object code via parameterized procedure abstraction. Thesis (m.s.), University of Victoria, 1993.