Cpplib Internals

Last revised Jan 2001
for GCC version 3.0

Neil Booth

Table of Contents

1 Cpplib—the core of the GNU C Preprocessor

The GNU C preprocessor in GCC 3.0 has been completely rewritten. It is now implemented as a library, cpplib, so it can be easily shared between a stand-alone preprocessor, and a preprocessor integrated with the C, C++ and Objective-C front ends. It is also available for use by other programs, though this is not recommended as its exposed interface has not yet reached a point of reasonable stability.

This library has been written to be re-entrant, so that it can be used to preprocess many files simultaneously if necessary. It has also been written with the preprocessing token as the fundamental unit; the preprocessor in previous versions of GCC would operate on text strings as the fundamental unit.

This brief manual documents some of the internals of cpplib, and a few tricky issues encountered. It also describes certain behaviour we would like to preserve, such as the format and spacing of its output.

Identifiers, macro expansion, hash nodes, lexing.

Conventions

cpplib has two interfaces—one is exposed internally only, and the other is for both internal and external use.

The convention is that functions and types that are exposed to multiple files internally are prefixed with '`_cpp_`', and are to be found in the file '`cpphash.h`'. Functions and types exposed to external clients are in '`cpplib.h`', and prefixed with '`cpp_`'. For historical reasons this is no longer quite true, but we should strive to stick to it.

We are striving to reduce the information exposed in '`cpplib.h`' to the bare minimum necessary, and then to keep it there. This makes clear exactly what external clients are entitled to assume, and allows us to change internals in the future without worrying whether library clients are perhaps relying on some kind of undocumented implementation-specific behaviour.

The Lexer

The lexer is contained in the file ‘cpplex.c’. We want to have a lexer that is single-pass, for efficiency reasons. We would also like the lexer to only step forwards through the input files, and not step back. This will make future changes to support different character sets, in particular state or shift-dependent ones, much easier.

This file also contains all information needed to spell a token, i.e. to output it either in a diagnostic or to a preprocessed output file. This information is not exported, but made available to clients through such functions as ‘cpp_spell_token’ and ‘cpp_token_len’.

The most painful aspect of lexing ISO-standard C and C++ is handling trigraphs and backlash-escaped newlines. Trigraphs are processed before any interpretation of the meaning of a character is made, and unfortunately there is a trigraph representation for a backslash, so it is possible for the trigraph ‘??/’ to introduce an escaped newline.

Escaped newlines are tedious because theoretically they can occur anywhere—between the ‘+’ and ‘=’ of the ‘+=’ token, within the characters of an identifier, and even between the ‘*’ and ‘/’ that terminates a comment. Moreover, you cannot be sure there is just one—there might be an arbitrarily long sequence of them.

So the routine ‘parse_identifier’, that lexes an identifier, cannot assume that it can scan forwards until the first non-identifier character and be done with it, because this could be the ‘\’ introducing an escaped newline, or the ‘?’ introducing the trigraph sequence that represents the ‘\’ of an escaped newline. Similarly for the routine that handles numbers, ‘parse_number’. If these routines stumble upon a ‘?’ or ‘\’, they call ‘skip_escaped_newlines’ to skip over any potential escaped newlines before checking whether they can finish.

Similarly code in the main body of ‘_cpp_lex_token’ cannot simply check for a ‘=’ after a ‘+’ character to determine whether it has a ‘+=’ token; it needs to be prepared for an escaped newline of some sort. These cases use the function ‘get_effective_char’, which returns the first character after any intervening newlines.

The lexer needs to keep track of the correct column position, including counting tabs as specified by the ‘-ftabstop=’ option. This should be done even within comments; C-style comments can appear in the middle of a line, and we want to report diagnostics in the correct position for text appearing after the end of the comment.

Some identifiers, such as ‘__VA_ARGS__’ and poisoned identifiers, may be invalid and require a diagnostic. However, if they appear in a macro expansion we don’t want to complain with each use of the macro. It is therefore best to catch them during the lexing stage, in ‘parse_identifier’. In both cases, whether a diagnostic is needed or not is dependent upon lexer state. For example, we don’t want to issue a diagnostic for re-poisoning a poisoned identifier, or for using ‘__VA_ARGS__’ in the expansion of a variable-argument macro. Therefore ‘parse_identifier’ makes use of flags to determine whether a diagnostic is appropriate. Since we change state on a per-token basis, and don’t lex whole lines at a time, this is not a problem.

Another place where state flags are used to change behaviour is whilst parsing header names. Normally, a ‘<’ would be lexed as a single token. After a #include directive, though, it should be lexed as a single token as far as the nearest ‘>’ character. Note that we don't allow the terminators of header names to be escaped; the first '`"`' or '`>`' terminates the header name.

Interpretation of some character sequences depends upon whether we are lexing C, C`++` or Objective-C, and on the revision of the standard in force. For example, '`::`' is a single token in C`++`, but two separate '`:`' tokens, and almost certainly a syntax error, in C. Such cases are handled in the main function '`_cpp_lex_token`', based upon the flags set in the '`cpp_options`' structure.

Note we have almost, but not quite, achieved the goal of not stepping backwards in the input stream. Currently '`skip_escaped_newlines`' does step back, though with care it should be possible to adjust it so that this does not happen. For example, one tricky issue is if we meet a trigraph, but the command line option '`-trigraphs`' is not in force but '`-Wtrigraphs`' is, we need to warn about it but then buffer it and continue to treat it as 3 separate characters.

Whitespace

The lexer has been written to treat each of '`\r`', '`\n`', '`\r\n`' and '`\n\r`' as a single new line indicator. This allows it to transparently preprocess MS-DOS, Macintosh and Unix files without their needing to pass through a special filter beforehand.

We also decided to treat a backslash, either '\' or the trigraph '`??/`', separated from one of the above newline indicators by non-comment whitespace only, as intending to escape the newline. It tends to be a typing mistake, and cannot reasonably be mistaken for anything else in any of the C-family grammars. Since handling it this way is not strictly conforming to the ISO standard, the library issues a warning wherever it encounters it.

Handling newlines like this is made simpler by doing it in one place only. The function '`handle_newline`' takes care of all newline characters, and '`skip_escaped_newlines`' takes care of arbitrarily long sequences of escaped newlines, deferring to '`handle_newline`' to handle the newlines themselves.

Another whitespace issue only concerns the stand-alone preprocessor: we want to guarantee that re-reading the preprocessed output results in an identical token stream. Without taking special measures, this might not be the case because of macro substitution. We could simply insert a space between adjacent tokens, but ideally we would like to keep this to a minimum, both for aesthetic reasons and because it causes problems for people who still try to abuse the preprocessor for things like Fortran source and Makefiles.

The token structure contains a flags byte, and two flags are of interest here: '`PREV_WHITE`' and '`AVOID_LPASTE`'. '`PREV_WHITE`' indicates that the token was preceded by whitespace; if this is the case we need not worry about it incorrectly pasting with its predecessor. The '`AVOID_LPASTE`' flag is set by the macro expansion routines, and indicates that paste avoidance by insertion of a space to the left of the token may be necessary. Recursively, the first token of a macro substitution, the first token after a macro substitution, the first token of a substituted argument, and the first token after a substituted argument are all flagged '`AVOID_LPASTE`' by the macro expander.

If a token flagged in this way does not have a '`PREV_WHITE`' flag, and the routine `cpp_avoid_paste` determines that it might be misinterpreted by the lexer if a space is not inserted between it and the immediately preceding token, then stand-alone CPP's output routines will insert a space between them. To avoid excessive spacing, `cpp_avoid_paste` tries hard to only request a space if one is likely to be necessary, but for reasons of efficiency it is slightly conservative and might recommend a space where one is not strictly needed.

Finally, the preprocessor takes great care to ensure it keeps track of both the position of a token in the source file, for diagnostic purposes, and where it should appear in the output file, because using CPP for other languages like assembler requires this. The two positions may differ for the following reasons:

- Escaped newlines are deleted, so lines spliced in this way are joined to form a single logical line.
- A macro expansion replaces the tokens that form its invocation, but any newlines appearing in the macro's arguments are interpreted as a single space, with the result that the macro's replacement appears in full on the same line that the macro name appeared in the source file. This is particularly important for stringification of arguments—newlines embedded in the arguments must appear in the string as spaces.

The source file location is maintained in the `lineno` member of the `cpp_buffer` structure, and the column number inferred from the current position in the buffer relative to the `line_base` buffer variable, which is updated with every newline whether escaped or not.

TODO: Finish this.

Hash Nodes

When cpplib encounters an “identifier”, it generates a hash code for it and stores it in the hash table. By “identifier” we mean tokens with type ‘`CPP_NAME`’; this includes identifiers in the usual C sense, as well as keywords, directive names, macro names and so on. For example, all of ‘`pragma`’, ‘`int`’, ‘`foo`’ and ‘`__GNUC__`’ are identifiers and hashed when lexed.

Each node in the hash table contain various information about the identifier it represents. For example, its length and type. At any one time, each identifier falls into exactly one of three categories:

- Macros

  These have been declared to be macros, either on the command line or with `#define`. A few, such as ‘`__TIME__`’ are builtins entered in the hash table during initialisation. The hash node for a normal macro points to a structure with more information about the macro, such as whether it is function-like, how many arguments it takes, and its expansion. Builtin macros are flagged as special, and instead contain an enum indicating which of the various builtin macros it is.

- Assertions

  Assertions are in a separate namespace to macros. To enforce this, cpp actually prepends a `#` character before hashing and entering it in the hash table. An assertion’s node points to a chain of answers to that assertion.

- Void

  Everything else falls into this category—an identifier that is not currently a macro, or a macro that has since been undefined with `#undef`.

  When preprocessing C++, this category also includes the named operators, such as ‘`xor`’. In expressions these behave like the operators they represent, but in contexts where the spelling of a token matters they are spelt differently. This spelling distinction is relevant when they are operands of the stringizing and pasting macro operators `#` and `##`. Named operator hash nodes are flagged, both to catch the spelling distinction and to prevent them from being defined as macros.

The same identifiers share the same hash node. Since each identifier token, after lexing, contains a pointer to its hash node, this is used to provide rapid lookup of various information. For example, when parsing a `#define` statement, CPP flags each argument’s identifier hash node with the index of that argument. This makes duplicated argument checking an O(1) operation for each argument. Similarly, for each identifier in the macro’s expansion, lookup to see if it is an argument, and which argument it is, is also an O(1) operation. Further, each directive name, such as ‘`endif`’, has an associated directive enum stored in its hash node, so that directive lookup is also O(1).

Macro Expansion Algorithm

File Handling

Fairly obviously, the file handling code of cpplib resides in the file ‘`cppfiles.c`’. It takes care of the details of file searching, opening, reading and caching, for both the main source file and all the headers it recursively includes.

The basic strategy is to minimize the number of system calls. On many systems, the basic `open ()` and `fstat ()` system calls can be quite expensive. For every `#include`-d file, we need to try all the directories in the search path until we find a match. Some projects, such as glibc, pass twenty or thirty include paths on the command line, so this can rapidly become time consuming.

For a header file we have not encountered before we have little choice but to do this. However, it is often the case that the same headers are repeatedly included, and in these cases we try to avoid repeating the filesystem queries whilst searching for the correct file.

For each file we try to open, we store the constructed path in a splay tree. This path first undergoes simplification by the function `_cpp_simplify_pathname`. For example, ‘`/usr/include/bits/../foo.h`’ is simplified to ‘`/usr/include/foo.h`’ before we enter it in the splay tree and try to `open ()` the file. CPP will then find subsequent uses of ‘`foo.h`’, even as ‘`/usr/include/foo.h`’, in the splay tree and save system calls.

Further, it is likely the file contents have also been cached, saving a `read ()` system call. We don’t bother caching the contents of header files that are re-inclusion protected, and whose re-inclusion macro is defined when we leave the header file for the first time. If the host supports it, we try to map suitably large files into memory, rather than reading them in directly.

The include paths are internally stored on a null-terminated singly-linked list, starting with the `"header.h"` directory search chain, which then links into the `<header.h>` directory chain.

Files included with the `<foo.h>` syntax start the lookup directly in the second half of this chain. However, files included with the `"foo.h"` syntax start at the beginning of the chain, but with one extra directory prepended. This is the directory of the current file; the one containing the `#include` directive. Prepending this directory on a per-file basis is handled by the function `search_from`.

Note that a header included with a directory component, such as `#include "mydir/foo.h"` and opened as ‘`/usr/local/include/mydir/foo.h`’, will have the complete path minus the basename ‘`foo.h`’ as the current directory.

Enough information is stored in the splay tree that CPP can immediately tell whether it can skip the header file because of the multiple include optimisation, whether the file didn’t exist or couldn’t be opened for some reason, or whether the header was flagged not to be re-used, as it is with the obsolete `#import` directive.

For the benefit of MS-DOS filesystems with an 8.3 filename limitation, CPP offers the ability to treat various include file names as aliases for the real header files with shorter names. The map from one to the other is found in a special file called ‘`header.gcc`’, stored in the command line (or system) include directories to which the mapping applies. This may be higher up the directory tree than the full path to the file minus the base name.

Index