Cpplib Internals

For GCC version 17.0.0 (pre-release)

(GCC)

Neil Booth

Copyright © 2000-2026 Free Software Foundation, Inc.

Permission is granted to make and distribute verbatim copies of this manual provided the copyright notice and this permission notice are preserved on all copies.

Permission is granted to copy and distribute modified versions of this manual under the conditions for verbatim copying, provided also that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.

Permission is granted to copy and distribute translations of this manual into another language, under the above conditions for modified versions.

Table of Contents

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 `internal.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 behavior.

The Lexer

Overview

The lexer is contained in the file `lex.cc`. It is a hand-coded lexer, and not implemented as a state machine. It can understand C, C++ and Objective-C source code, and has been extended to allow reasonably successful preprocessing of assembly language. The lexer does not make an initial pass to strip out trigraphs and escaped newlines, but handles them as they are encountered in a single pass of the input file. It returns preprocessing tokens individually, not a line at a time.

It is mostly transparent to users of the library, since the library's interface for obtaining the next token, `cpp_get_token`, takes care of lexing new tokens, handling directives, and expanding macros as necessary. However, the lexer does expose some functionality so that clients of the library can easily spell a given token, such as `cpp_spell_token` and `cpp_token_len`. These functions are useful when generating diagnostics, and for emitting the preprocessed output.

Lexing a token

Lexing of an individual token is handled by `_cpp_lex_direct` and its subroutines. In its current form the code is quite complicated, with read ahead characters and such-like, since it strives to not step back in the character stream in preparation for handling non-ASCII file encodings. The current plan is to convert any such files to UTF-8 before processing them. This complexity is therefore unnecessary and will be removed, so I'll not discuss it further here.

The job of `_cpp_lex_direct` is simply to lex a token. It is not responsible for issues like directive handling, returning lookahead tokens directly, multiple-include optimization, or conditional block skipping. It necessarily has a minor rôle to play in memory management of lexed lines. I discuss these issues in a separate section (see [Lexing a line], page 5).

The lexer places the token it lexes into storage pointed to by the variable `cur_token`, and then increments it. This variable is important for correct diagnostic positioning. Unless a specific line and column are passed to the diagnostic routines, they will examine the `line` and `col` values of the token just before the location that `cur_token` points to, and use that location to report the diagnostic.

The lexer does not consider whitespace to be a token in its own right. If whitespace (other than a new line) precedes a token, it sets the `PREV_WHITE` bit in the token's flags. Each token has its `line` and `col` variables set to the line and column of the first character of the token. This line number is the line number in the translation unit, and can be converted to a source (file, line) pair using the line map code.

The first token on a logical, i.e. unescaped, line has the flag `BOL` set for beginning-of-line. This flag is intended for internal use, both to distinguish a '`#`' that begins a directive from one that doesn't, and to generate a call-back to clients that want to be notified about the start of every non-directive line with tokens on it. Clients cannot reliably determine this for themselves: the first token might be a macro, and the tokens of a macro expansion do not have the `BOL` flag set. The macro expansion may even be empty, and the next token on the line certainly won't have the `BOL` flag set.

New lines are treated specially; exactly how the lexer handles them is context-dependent. The C standard mandates that directives are terminated by the first unescaped newline character, even if it appears in the middle of a macro expansion. Therefore, if the state variable `in_directive` is set, the lexer returns a `CPP_EOF` token, which is normally used to indicate end-of-file, to indicate end-of-directive. In a directive a `CPP_EOF` token never means end-of-file. Conveniently, if the caller was `collect_args`, it already handles `CPP_EOF` as if it were end-of-file, and reports an error about an unterminated macro argument list.

The C standard also specifies that a new line in the middle of the arguments to a macro is treated as whitespace. This white space is important in case the macro argument is stringized. The state variable `parsing_args` is nonzero when the preprocessor is collecting the arguments to a macro call. It is set to 1 when looking for the opening parenthesis to a function-like macro, and 2 when collecting the actual arguments up to the closing parenthesis, since these two cases need to be distinguished sometimes. One such time is here: the lexer sets the `PREV_WHITE` flag of a token if it meets a new line when `parsing_args` is set to 2. It doesn't set it if it meets a new line when `parsing_args` is 1, since then code like

```
#define foo() bar
foo
baz
```

would be output with an erroneous space before '`baz`':

```
foo
 baz
```

This is a good example of the subtlety of getting token spacing correct in the preprocessor; there are plenty of tests in the testsuite for corner cases like this.

The lexer is 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.

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, for example, the routine that lexes a number, `parse_number`, cannot assume that it can scan forwards until the first non-number 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. If it encounters a '?' or '\', it calls `skip_escaped_newlines` to skip over any potential escaped newlines before checking whether the number has been finished.

Similarly code in the main body of `_cpp_lex_direct` 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. Such cases use the function `get_effective_char`, which returns the first character after any intervening escaped 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 C-style comments; they 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 the lexer's 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 state 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 behavior is whilst lexing 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 in C it is two separate ':' tokens and almost certainly a syntax error. Such cases are handled by `_cpp_lex_direct` based upon command-line flags stored in the `cpp_options` structure.

Once a token has been lexed, it leads an independent existence. The spelling of numbers, identifiers and strings is copied to permanent storage from the original input buffer, so a token remains valid and correct even if its source buffer is freed with `_cpp_pop_buffer`. The storage holding the spellings of such tokens remains until the client program calls cpp_destroy, probably at the end of the translation unit.

Lexing a line

When the preprocessor was changed to return pointers to tokens, one feature I wanted was some sort of guarantee regarding how long a returned pointer remains valid. This is important to the stand-alone preprocessor, the future direction of the C family front ends, and even to cpplib itself internally.

Occasionally the preprocessor wants to be able to peek ahead in the token stream. For example, after the name of a function-like macro, it wants to check the next token to see if it is an opening parenthesis. Another example is that, after reading the first few tokens of a #pragma directive and not recognizing it as a registered pragma, it wants to backtrack and allow the user-defined handler for unknown pragmas to access the full #pragma token stream. The stand-alone preprocessor wants to be able to test the current token with the previous one to see if a space needs to be inserted to preserve their separate tokenization upon re-lexing (paste avoidance), so it needs to be sure the pointer to the previous token is still valid. The recursive-descent C++ parser wants to be able to perform tentative parsing arbitrarily far ahead in the token stream, and then to be able to jump back to a prior position in that stream if necessary.

The rule I chose, which is fairly natural, is to arrange that the preprocessor lex all tokens on a line consecutively into a token buffer, which I call a *token run*, and when meeting an unescaped new line (newlines within comments do not count either), to start lexing back at the beginning of the run. Note that we do *not* lex a line of tokens at once; if we did that parse_identifier would not have state flags available to warn about invalid identifiers (see [Invalid identifiers], page 5).

In other words, accessing tokens that appeared earlier in the current line is valid, but since each logical line overwrites the tokens of the previous line, tokens from prior lines are unavailable. In particular, since a directive only occupies a single logical line, this means that the directive handlers like the #pragma handler can jump around in the directive's tokens if necessary.

Two issues remain: what about tokens that arise from macro expansions, and what happens when we have a long line that overflows the token run?

Since we promise clients that we preserve the validity of pointers that we have already returned for tokens that appeared earlier in the line, we cannot reallocate the run. Instead, on overflow it is expanded by chaining a new token run on to the end of the existing one.

The tokens forming a macro's replacement list are collected by the #define handler, and placed in storage that is only freed by cpp_destroy. So if a macro is expanded in the line of tokens, the pointers to the tokens of its expansion that are returned will always remain valid. However, macros are a little trickier than that, since they give rise to three sources of fresh tokens. They are the built-in macros like __LINE__, and the '#' and '##' operators for stringizing and token pasting. I handled this by allocating space for these tokens from the lexer's token run chain. This means they automatically receive the same lifetime guarantees as lexed tokens, and we don't need to concern ourselves with freeing them.

Lexing into a line of tokens solves some of the token memory management issues, but not all. The opening parenthesis after a function-like macro name might lie on a different line, and the front ends definitely want the ability to look ahead past the end of the current line. So cpplib only moves back to the start of the token run at the end of a line if the variable keep_tokens is zero. Line-buffering is quite natural for the preprocessor, and as a result the only time cpplib needs to increment this variable is whilst looking for the opening parenthesis to, and reading the arguments of, a function-like macro. In the near future cpplib will export an interface to increment and decrement this variable, so that clients can share full control over the lifetime of token pointers too.

The routine _cpp_lex_token handles moving to new token runs, calling _cpp_lex_direct to lex new tokens, or returning previously-lexed tokens if we stepped back in the token stream. It also checks each token for the BOL flag, which might indicate a directive that needs to be handled, or require a start-of-line call-back to be made. _cpp_lex_token also handles skipping over tokens in failed conditional blocks, and invalidates the control macro of the multiple-include optimization if a token was successfully lexed outside a directive. In other words, its callers do not need to concern themselves with such issues.

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 built-ins entered in the hash table during initialization. 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. Built-in macros are flagged as special, and instead contain an enum indicating which of the various built-in 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

Macro expansion is a tricky operation, fraught with nasty corner cases and situations that render what you thought was a nifty way to optimize the preprocessor's expansion algorithm wrong in quite subtle ways.

I strongly recommend you have a good grasp of how the C and C++ standards require macros to be expanded before diving into this section, let alone the code!. If you don't have a clear mental picture of how things like nested macro expansion, stringizing and token pasting are supposed to work, damage to your sanity can quickly result.

Internal representation of macros

The preprocessor stores macro expansions in tokenized form. This saves repeated lexing passes during expansion, at the cost of a small increase in memory consumption on average. The tokens are stored contiguously in memory, so a pointer to the first one and a token count is all you need to get the replacement list of a macro.

If the macro is a function-like macro the preprocessor also stores its parameters, in the form of an ordered list of pointers to the hash table entry of each parameter's identifier. Further, in the macro's stored expansion each occurrence of a parameter is replaced with a special token of type `CPP_MACRO_ARG`. Each such token holds the index of the parameter it represents in the parameter list, which allows rapid replacement of parameters with their arguments during expansion. Despite this optimization it is still necessary to store the original parameters to the macro, both for dumping with e.g., `-dD`, and to warn about non-trivial macro redefinitions when the parameter names have changed.

Macro expansion overview

The preprocessor maintains a *context stack*, implemented as a linked list of `cpp_context` structures, which together represent the macro expansion state at any one time. The `struct cpp_reader` member variable `context` points to the current top of this stack. The top normally holds the unexpanded replacement list of the innermost macro under expansion, except when cpplib is about to pre-expand an argument, in which case it holds that argument's unexpanded tokens.

When there are no macros under expansion, cpplib is in *base context*. All contexts other than the base context contain a contiguous list of tokens delimited by a starting and ending token. When not in base context, cpplib obtains the next token from the list of the top context. If there are no tokens left in the list, it pops that context off the stack, and subsequent ones if necessary, until an unexhausted context is found or it returns to base context. In base context, cpplib reads tokens directly from the lexer.

If it encounters an identifier that is both a macro and enabled for expansion, cpplib prepares to push a new context for that macro on the stack by calling the routine `enter_macro_context`. When this routine returns, the new context will contain the unexpanded tokens of the replacement list of that macro. In the case of function-like macros, `enter_macro_context` also replaces any parameters in the replacement list, stored as `CPP_MACRO_ARG` tokens, with the appropriate macro argument. If the standard requires that the parameter be replaced with its expanded argument, the argument will have been fully macro expanded first.

`enter_macro_context` also handles special macros like `__LINE__`. Although these macros expand to a single token which cannot contain any further macros, for reasons of token spacing (see [Token Spacing], page 15) and simplicity of implementation, cpplib handles these special macros by pushing a context containing just that one token.

The final thing that `enter_macro_context` does before returning is to mark the macro disabled for expansion (except for special macros like `__TIME__`). The macro is re-enabled when its context is later popped from the context stack, as described above. This strict ordering ensures that a macro is disabled whilst its expansion is being scanned, but that it is *not* disabled whilst any arguments to it are being expanded.

Scanning the replacement list for macros to expand

The C standard states that, after any parameters have been replaced with their possibly-expanded arguments, the replacement list is scanned for nested macros. Further, any identifiers in the replacement list that are not expanded during this scan are never again eligible for expansion in the future, if the reason they were not expanded is that the macro in question was disabled.

Clearly this latter condition can only apply to tokens resulting from argument pre-expansion. Other tokens never have an opportunity to be re-tested for expansion. It is possible for identifiers that are function-like macros to not expand initially but to expand during a later scan. This occurs when the identifier is the last token of an argument (and therefore originally followed by a comma or a closing parenthesis in its macro's argument list), and when it replaces its parameter in the macro's replacement list, the subsequent token happens to be an opening parenthesis (itself possibly the first token of an argument).

It is important to note that when cpplib reads the last token of a given context, that context still remains on the stack. Only when looking for the *next* token do we pop it off the stack and drop to a lower context. This makes backing up by one token easy, but more importantly ensures that the macro corresponding to the current context is still disabled when we are considering the last token of its replacement list for expansion (or indeed expanding it). As an example, which illustrates many of the points above, consider

```
#define foo(x) bar x
foo(foo) (2)
```

which fully expands to '`bar foo (2)`'. During pre-expansion of the argument, '`foo`' does not expand even though the macro is enabled, since it has no following parenthesis [pre-expansion of an argument only uses tokens from that argument; it cannot take tokens from whatever follows the macro invocation]. This still leaves the argument token '`foo`' eligible for future expansion. Then, when re-scanning after argument replacement, the token '`foo`' is rejected for expansion, and marked ineligible for future expansion, since the macro is now disabled. It is disabled because the replacement list '`bar foo`' of the macro is still on the context stack.

If instead the algorithm looked for an opening parenthesis first and then tested whether the macro were disabled it would be subtly wrong. In the example above, the replacement list of '`foo`' would be popped in the process of finding the parenthesis, re-enabling '`foo`' and expanding it a second time.

Looking for a function-like macro's opening parenthesis

Function-like macros only expand when immediately followed by a parenthesis. To do this cpplib needs to temporarily disable macros and read the next token. Unfortunately, because of spacing issues (see [Token Spacing], page 15), there can be fake padding tokens in-between, and if the next real token is not a parenthesis cpplib needs to be able to back up that one token as well as retain the information in any intervening padding tokens.

Backing up more than one token when macros are involved is not permitted by cpplib, because in general it might involve issues like restoring popped contexts onto the context stack, which are too hard. Instead, searching for the parenthesis is handled by a special function, `funlike_invocation_p`, which remembers padding information as it reads tokens. If the next real token is not an opening parenthesis, it backs up that one token, and then pushes an extra context just containing the padding information if necessary.

Marking tokens ineligible for future expansion

As discussed above, cpplib needs a way of marking tokens as unexpandable. Since the tokens cpplib handles are read-only once they have been lexed, it instead makes a copy of the token and adds the flag `NO_EXPAND` to the copy.

For efficiency and to simplify memory management by avoiding having to remember to free these tokens, they are allocated as temporary tokens from the lexer's current token run (see [Lexing a line], page 5) using the function `_cpp_temp_token`. The tokens are then re-used once the current line of tokens has been read in.

This might sound unsafe. However, tokens runs are not re-used at the end of a line if it happens to be in the middle of a macro argument list, and cpplib only wants to back-up more than one lexer token in situations where no macro expansion is involved, so the optimization is safe.

Token Spacing

First, consider an issue that only concerns the stand-alone preprocessor: there needs to be a guarantee that re-reading its preprocessed output results in an identical token stream. Without taking special measures, this might not be the case because of macro substitution. For example:

```
#define PLUS +
#define EMPTY
#define f(x) =x=
+PLUS -EMPTY- PLUS+ f(=)
        ↦ + + - - + + = = =
not
        ↦ ++ -- ++ ===
```

One solution would be to simply insert a space between all adjacent tokens. However, we would like to keep space insertion 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.

For now, just notice that when tokens are added (or removed, as shown by the `EMPTY` example) from the original lexed token stream, we need to check for accidental token pasting. We call this *paste avoidance*. Token addition and removal can only occur because of macro expansion, but accidental pasting can occur in many places: both before and after each macro replacement, each argument replacement, and additionally each token created by the '`#`' and '`##`' operators.

Look at how the preprocessor gets whitespace output correct normally. The `cpp_token` structure contains a flags byte, and one of those flags is `PREV_WHITE`. This is flagged by the lexer, and indicates that the token was preceded by whitespace of some form other than a new line. The stand-alone preprocessor can use this flag to decide whether to insert a space between tokens in the output.

Now consider the result of the following macro expansion:

```
#define add(x, y, z) x + y +z;
sum = add (1,2, 3);
        ↦ sum = 1 + 2 +3;
```

The interesting thing here is that the tokens '`1`' and '`2`' are output with a preceding space, and '`3`' is output without a preceding space, but when lexed none of these tokens had that property. Careful consideration reveals that '`1`' gets its preceding whitespace from the space preceding '`add`' in the macro invocation, *not* replacement list. '`2`' gets its whitespace from the space preceding the parameter '`y`' in the macro replacement list, and '`3`' has no preceding space because parameter '`z`' has none in the replacement list.

Once lexed, tokens are effectively fixed and cannot be altered, since pointers to them might be held in many places, in particular by in-progress macro expansions. So instead of modifying the two tokens above, the preprocessor inserts a special token, which I call a *padding token*, into the token stream to indicate that spacing of the subsequent token is special. The preprocessor inserts padding tokens in front of every macro expansion and expanded macro argument. These point to a *source token* from which the subsequent real token should inherit its spacing. In the above example, the source tokens are '`add`' in the macro invocation, and '`y`' and '`z`' in the macro replacement list, respectively.

It is quite easy to get multiple padding tokens in a row, for example if a macro's first replacement token expands straight into another macro.

```
#define foo bar
#define bar baz
[foo]
        ↦ [baz]
```

Here, two padding tokens are generated with sources the '`foo`' token between the brackets, and the '`bar`' token from foo's replacement list, respectively. Clearly the first padding token is the one to use, so the output code should contain a rule that the first padding token in a sequence is the one that matters.

But what if a macro expansion is left? Adjusting the above example slightly:

```
#define foo bar
#define bar EMPTY baz
#define EMPTY
[foo] EMPTY;
        ↦ [ baz] ;
```

As shown, now there should be a space before '`baz`' and the semicolon in the output.

The rules we decided above fail for '`baz`': we generate three padding tokens, one per macro invocation, before the token '`baz`'. We would then have it take its spacing from the first of these, which carries source token '`foo`' with no leading space.

It is vital that cpplib get spacing correct in these examples since any of these macro expansions could be stringized, where spacing matters.

So, this demonstrates that not just entering macro and argument expansions, but leaving them requires special handling too. I made cpplib insert a padding token with a `NULL` source token when leaving macro expansions, as well as after each replaced argument in a macro's replacement list. It also inserts appropriate padding tokens on either side of tokens created by the '`#`' and '`##`' operators. I expanded the rule so that, if we see a padding token with a `NULL` source token, *and* that source token has no leading space, then we behave as if we have seen no padding tokens at all. A quick check shows this rule will then get the above example correct as well.

Now a relationship with paste avoidance is apparent: we have to be careful about paste avoidance in exactly the same locations we have padding tokens in order to get white space correct. This makes implementation of paste avoidance easy: wherever the stand-alone preprocessor is fixing up spacing because of padding tokens, and it turns out that no space is needed, it has to take the extra step to check that a space is not needed after all to avoid an accidental paste. The function `cpp_avoid_paste` advises whether a space is required between two consecutive tokens. To avoid excessive spacing, it tries hard to only require 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.

Line numbering

Just which line number anyway?

There are three reasonable requirements a cpplib client might have for the line number of a token passed to it:

- The source line it was lexed on.
- The line it is output on. This can be different to the line it was lexed on if, for example, there are intervening escaped newlines or C-style comments. For example:

```
foo /* A long
comment */ bar \
baz
⇒
foo bar baz
```

- If the token results from a macro expansion, the line of the macro name, or possibly the line of the closing parenthesis in the case of function-like macro expansion.

The `cpp_token` structure contains `line` and `col` members. The lexer fills these in with the line and column of the first character of the token. Consequently, but maybe unexpectedly, a token from the replacement list of a macro expansion carries the location of the token within the `#define` directive, because cpplib expands a macro by returning pointers to the tokens in its replacement list. The current implementation of cpplib assigns tokens created from built-in macros and the '`#`' and '`##`' operators the location of the most recently lexed token. This is a because they are allocated from the lexer's token runs, and because of the way the diagnostic routines infer the appropriate location to report.

The diagnostic routines in cpplib display the location of the most recently *lexed* token, unless they are passed a specific line and column to report. For diagnostics regarding tokens that arise from macro expansions, it might also be helpful for the user to see the original location in the macro definition that the token came from. Since that is exactly the information each token carries, such an enhancement could be made relatively easily in future.

The stand-alone preprocessor faces a similar problem when determining the correct line to output the token on: the position attached to a token is fairly useless if the token came from a macro expansion. All tokens on a logical line should be output on its first physical line, so the token's reported location is also wrong if it is part of a physical line other than the first.

To solve these issues, cpplib provides a callback that is generated whenever it lexes a preprocessing token that starts a new logical line other than a directive. It passes this token (which may be a `CPP_EOF` token indicating the end of the translation unit) to the callback routine, which can then use the line and column of this token to produce correct output.

Representation of line numbers

As mentioned above, cpplib stores with each token the line number that it was lexed on. In fact, this number is not the number of the line in the source file, but instead bears more resemblance to the number of the line in the translation unit.

The preprocessor maintains a monotonic increasing line count, which is incremented at every new line character (and also at the end of any buffer that does not end in a new line). Since a line number of zero is useful to indicate certain special states and conditions, this variable starts counting from one.

This variable therefore uniquely enumerates each line in the translation unit. With some simple infrastructure, it is straight forward to map from this to the original source file and line number pair, saving space whenever line number information needs to be saved. The code the implements this mapping lies in the files `line-map.cc` and `line-map.h`.

Command-line macros and assertions are implemented by pushing a buffer containing the right hand side of an equivalent `#define` or `#assert` directive. Some built-in macros are handled similarly. Since these are all processed before the first line of the main input file, it will typically have an assigned line closer to twenty than to one.

The Multiple-Include Optimization

Header files are often of the form

```
#ifndef FOO
#define FOO
...
#endif
```

to prevent the compiler from processing them more than once. The preprocessor notices such header files, so that if the header file appears in a subsequent `#include` directive and `FOO` is defined, then it is ignored and it doesn't preprocess or even re-open the file a second time. This is referred to as the *multiple include optimization*.

Under what circumstances is such an optimization valid? If the file were included a second time, it can only be optimized away if that inclusion would result in no tokens to return, and no relevant directives to process. Therefore the current implementation imposes requirements and makes some allowances as follows:

1. There must be no tokens outside the controlling `#if`-`#endif` pair, but whitespace and comments are permitted.
2. There must be no directives outside the controlling directive pair, but the *null directive* (a line containing nothing other than a single '`#`' and possibly whitespace) is permitted.
3. The opening directive must be of the form

   ```
   #ifndef FOO
   ```

   or

   ```
   #if !defined FOO     [equivalently, #if !defined(FOO)]
   ```

4. In the second form above, the tokens forming the `#if` expression must have come directly from the source file—no macro expansion must have been involved. This is because macro definitions can change, and tracking whether or not a relevant change has been made is not worth the implementation cost.
5. There can be no `#else` or `#elif` directives at the outer conditional block level, because they would probably contain something of interest to a subsequent pass.

First, when pushing a new file on the buffer stack, `_stack_include_file` sets the controlling macro `mi_cmacro` to `NULL`, and sets `mi_valid` to `true`. This indicates that the preprocessor has not yet encountered anything that would invalidate the multiple-include optimization. As described in the next few paragraphs, these two variables having these values effectively indicates top-of-file.

When about to return a token that is not part of a directive, `_cpp_lex_token` sets `mi_valid` to `false`. This enforces the constraint that tokens outside the controlling conditional block invalidate the optimization.

The `do_if`, when appropriate, and `do_ifndef` directive handlers pass the controlling macro to the function `push_conditional`. cpplib maintains a stack of nested conditional blocks, and after processing every opening conditional this function pushes an `if_stack` structure onto the stack. In this structure it records the controlling macro for the block, provided there is one and we're at top-of-file (as described above). If an `#elif` or `#else` directive is encountered, the controlling macro for that block is cleared to `NULL`. Otherwise, it survives until the `#endif` closing the block, upon which `do_endif` sets `mi_valid` to true and stores the controlling macro in `mi_cmacro`.

`_cpp_handle_directive` clears `mi_valid` when processing any directive other than an opening conditional and the null directive. With this, and requiring top-of-file to record a controlling macro, and no `#else` or `#elif` for it to survive and be copied to `mi_cmacro` by `do_endif`, we have enforced the absence of directives outside the main conditional block for the optimization to be on.

Note that whilst we are inside the conditional block, `mi_valid` is likely to be reset to `false`, but this does not matter since the closing `#endif` restores it to `true` if appropriate.

Finally, since `_cpp_lex_direct` pops the file off the buffer stack at `EOF` without returning a token, if the `#endif` directive was not followed by any tokens, `mi_valid` is `true` and `_cpp_pop_file_buffer` remembers the controlling macro associated with the file. Subsequent calls to `stack_include_file` result in no buffer being pushed if the controlling macro is defined, effecting the optimization.

A quick word on how we handle the

```
#if !defined FOO
```

case. `_cpp_parse_expr` and `parse_defined` take steps to see whether the three stages '`!`', '`defined-expression`' and '`end-of-directive`' occur in order in a `#if` expression. If so, they return the guard macro to `do_if` in the variable `mi_ind_cmacro`, and otherwise set it to `NULL`. `enter_macro_context` sets `mi_valid` to false, so if a macro was expanded whilst parsing any part of the expression, then the top-of-file test in `push_conditional` fails and the optimization is turned off.

File Handling

Fairly obviously, the file handling code of cpplib resides in the file `files.cc`. 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 optimization, 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.

Concept Index