12. Reusable Incremental C Preprocessing, Part II

W.M. McKeeman & Shota Aki
Digital Equipment Corporation
110 Spitbrook Road
Nashua, NH 03062

Abstract

We extend the reusable preprocessor described in Part I of this paper to add a fine-grain incremental capability, achieved by saving lists of post-preprocessor (post-pp) tokens corresponding to blocks of source lines. When the source is unchanged and all macro definitions used in a block are unchanged, the saved post-pp tokens are reused instead of repeating the preprocessing. Changes in macro definitions are detected by journaling macro table activities, analogous to a common error recovery technique.

Introduction

A reusable C preprocessor was described in Part I of this paper [5]. Part II continues the description, treating fine-grain incremental preprocessing. This presentation also depends on details from two earlier papers and the C Standard [6, 4, 1].

The starting observation for fine-grain incremental preprocessing is that, for most C source, the preprocessor is irrelevant. Recall that the granularity of the output of the incremental scanner corresponds to a logical line (more than one source line when terminating new-lines are suppressed by a comment or \). A token increment (or tokinc) is the sequence of tokens in a logical line.

If a tokinc is not a preprocessing directive and contains no macro invocations, then the sequence of tokens output by the preprocessor (a ppinc) is the same as its input.¹

A second, more general, observation is that if a ppinc is clean with respect to the corresponding tokincs, is not a preprocessing directive, and the macro definition of identifiers in it have not changed (including those identifiers that were created during macro expansion), then the ppinc does not change either.

¹This contradicts a fine point in the C Standard which dictates that string literal concatenation precede conversion of preprocessing tokens to tokens [1, Section 2.1.1.2, phase 6&7], but as a practical matter concatenation is best left to semantic analysis. [5, page 56]
Because this complex condition is not entirely dependent on information in the corresponding tokinc(s), it has been called an action at a distance problem [3].

A third observation is that most C source seen by a preprocessor comes from header files. The .h to .c ratio is typically 10:1 or more.

The fundamental assumption behind all incremental processing is that changes are likely to be small with respect to the whole. This is typically true of program source during program development. Furthermore changes tend to cluster.

Our principal innovation, attaching journals of interface activity to the incremental structures for the purpose of resolving action-at-a-distance questions, has application to a broad class of algorithms. There is a supporting innovation to avoid gratuitous changes in the interface values which would otherwise cause unnecessary reprocessing.

The remainder of this paper describes a way to exploit these facts to skip the bulk of repetitive preprocessing. The principal objective is improved turn-around; the secondary objective is carrying clean/dirty information (information about what has changed) forward from the scanner to later translation processes so that they too can be incremental.

There is very little published information on preprocessors and apparently none at all on incremental preprocessing.

Preprocessor Tasks

Our preprocessor treats input in a few stylized ways as described in Part I of this paper. The functions of the preprocessor are separated into three relatively independent subcomponents: the pp-directive module, the pp-macro module and the pp-expression module. The additional logic for implementing incremental processing is isolated in the macro table and some skipping logic for non-directive source. While there is considerable implementation machinery managing the increments, it is nearly invisible from the traditional part of the preprocessor.

Relation between Scanner and Preprocessor

A ppinc is time-stamped and contains post-preprocessor tokens as well as a journal of #define, #undef, and macro lookup actions.

Figure 2 illustrates the relationship between tokincs and ppinscs. Input that is interpreted by the preprocessor has a null token sequence in the ppinscs. A multiline macro invocation results in a single ppinc.

Preprocessor Increment Abstraction

The ppinc abstraction is implemented (in C) in module ppinc.c with public interface ppinc.h. It supplies an interface similar to the scanner token abstraction except that it operates on ppincs[4]. It is not necessary to use (or even know
# define S(a) 
#a
main() {
puts(S(
  hello
world!)
));
}

tokincs

entry $S$

main() {
  lookup main
  lookup puts
  lookup $S$
  puts("hello world!");

}   

ppincs

journals

Figure 2: Relation between Token Increments and Preprocessor Increments

about the incremental interface if only sequential access to post-preprocessing tokens is needed.

A preprocessing increment is represented by type PpInc. Values of type PpInc are called pp increment handles or just ppincs. Objects of type PpInc can be declared, assigned, and passed as parameters. The functions are described in Table 1.

The ppinc iterators behave in the same manner as the token, tokinc, and pp iterators [4, 5]. When the start or end of the ppinc sequence is exceeded, a special handle value of notPPINC is returned by the iterators.

Functions PpIncTokIncDelete() and PpIncTokIncInsert() are the list manipulation functions to modify the ppinc sequence as the structure of the source changes (e.g., old lines deleted, new lines inserted, macro definitions changed). Preprocessor increments are hashed by tokincs (see the TokInc abstraction [4]) to facilitate fast deletion and insertion operations.

The ppinc abstraction provides several attribute functions. PpInc2Pp() returns the leading post-pp token of its associated ppinc. Thus the following invariant is always maintained:

PpFirst() == PpInc2Pp(PpIncFirst())

The function PpIncGfpPp() is used in the incremental mechanisms. Given a token, it will return the containing ppinc.

Function PpIncMda() returns the time stamp of the specified ppinc. The time stamp indicates the age of the post-pp tokens in the ppinc as well. Individual time stamps for each token could have been used but since a new list of post-pp tokens is created each time a change to a tokinc occurs, a single time stamp assigned to a ppinc is sufficient.
typedef struct {private layout} PpInc

- iterator functions
  w = PpIncFirst();  
  - start at head of current file
  w = PpIncNext(w);  
  - step to next increment
  w = PpIncPrev(w);  
  - step back to previous increment
  - list manipulation functions
  PpIncTokIncDelete(j);  
  - tokinc j has been deleted
  PpIncTokIncInsert(j, a);  
  - tokinc j inserted ahead of tokinc a
  - attribute functions
  t = PpInc2Pp(w);  
  - first post-pp token in ppinc w
  w = PpIncOffPp(t);  
  - ppinc associated with post-pp token t
  k = PpIncMade(w);  
  - time of construction of ppinc w

Table 1: Preprocessor Increment Interface

Incremental Preprocessing

Incremental preprocessing requires a solution for the action-at-a-distance problem and time-stamped input from a scanner.

Specifically, the state of the macro table changes as the input file is preprocessed, implying some sort of sequential processing of the whole input to ensure that table information is correct at the point of use. In fact our incremental preprocessor sequentially passes over all the ppincs, checking that they are not stale and carrying out action-at-a-distance checks. The tokens in a ppinc are not touched unless preprocessing must be repeated.

We chose to incrementally support preprocessing directives \#define, \#undef, and all non-directive text. The remainder of the input is either infrequent, inexpensive to repreprocess, or difficult to preprocess incrementally. The evaluation of preprocessor conditional expressions, in particular, is always repeated and always preceded by macro expansion.

Macro Table, \#define, and \#undef

The structure of the macro table is conventional. It is a list of macro definitions in order of source appearance. Each table entry is hashed by the String handle of the macro name for fast lookup[4]. The attributes of a macro table entry are type of macro (function versus object macro), the formal (if a function macro), and the logical line for the body of the macro. If there is no entry for its macro name in the macro table, directive \#define inserts a new definition. If there is a definition in the table but it is benign, the existing definition is left in place. If the redefinition is not benign (different in any way), a diagnostic is issued.
Directive `#undef` causes any macro definition for this name to be obscured (but does not delete the entry from the table).

An incremental preprocessor requires an unconventional feature in the macro table entries: a hot/cold status. *Hot* means the macro definition is active and a lookup on the name will find this entry. *Cold* means the definition is inactive and a lookup for use will skip over this entry. At the start of an incremental preprocessing run, all entries in the table are marked cold and unobscured.

When a `#define` ppline is encountered, its journal contains the result of the previous table lookup for the macro name.

\[ \text{loc} = \text{MacroDefLookup}(\text{name}) \]

The lookup is executed. Cold entries are visible; obscured entries are not visible. If the lookup gives the same result as before, the entry is marked hot, simulating the `#define` action. Because we wish to make only this simple check on table location, we never delete entries from the table, therefore avoiding accidental matches to a redefined macro. If the lookup fails to repeat the result, normal preprocessing of the `#define` is carried out. The old entry remains, but is clobbered so as never to be seen again.

When an `#undef` ppline is encountered, its journal also contains the result of the previous table lookup for the macro name.

\[ \text{loc} = \text{MacroUseLookup}(\text{name}) \]

Neither cold nor obscured entries are visible. If the lookup gives the same result as before (including "not there") preprocessing can be skipped. If a hot entry was found, it is marked obscured, implementing the `#undef` actions.

There are two kinds of macro definitions not in the source: those from compiler predefined macros and those from command line settings. These must be treated just like definitions occurring in the input.

This approach has the property of leaving invisible but space consuming garbage in the macro table. Over time, if there is a lot of programmer activity on the `#define` parts of the program, the storage allocated may become excessive. The garbage collection technique is straightforward: erase all contents of the ppines and the macro table and start over with a fresh slate. The user will only experience a performance glitch equivalent to one normal, non-incremental preprocessor run.

**Conditional Directives**

Expressions in `#if` and `#elif` are always evaluated. The *unexamined* branches of `#if` directives result in neither actions nor post-pp tokens. The nesting of preprocessor conditional directives must be tracked, even on the unexamined branches. While tracking through the tokens and not evaluating them, only the first token in each unexamined tokinc is examined. When the matching construct is finally reached, active examination begins again.
This situation reveals an interesting property of the widespread use of \#ifdef-\#endif bracketing to avoid double-including a header file. The entire header file is repeatedly examined by the preprocessor just so that its contents can be discarded. The programmer is often unaware of how much extra work the preprocessor is actually doing. As a consequence, some traditional compilers have added a check for this specific C construct and treat it as a special case. The check is: if the first thing in the file is \#ifdef followed by a \#define of the identifier tested in the \#ifdef, and the last thing in the file is the matching \#endif, and the file has not been touched since it was included, then the inclusion may be skipped.

An alternative language construct is the \#include directive with semantics of include-once. Such a directive has begun to appear in other languages and is a good candidate for inclusion in a future C Standard.

The include Directive

One need not do anything incremental for \#include. The incremental algorithm already discussed will make the processing of the contents of the included file incremental and therefore fast. On the other hand, because the vast majority of C presented to compilers comes from header files, precompiling headers is a common practice even for non-incremental compilers. We choose instead to rely on \_skiplists which are described below.

The \#include directive reverses the many-one mapping from tokens to ppincs. Now a single token is associated with a list of ppincs. If the \#include directive itself has been changed, the saved ppinc list is abandoned and preprocessing is repeated. Otherwise the ppincs corresponding to the header file are checked in the normal way.

Because the identity of the included file cannot reliably be known until the preprocessor evaluates the \#include directive, the preprocessor may have to call the scanner to get the needed tokens. As a practical matter, this means the incremental preprocessor must keep the scanner output ready for use and reuse over a series of runs. Our use of this technology in a vertically integrated development environment provided a convenient place to keep the required state.

Non-directive Increments

When a non-directive ppinc is not stale with respect to its corresponding tokens, the content of the ppinc is determined by the macro definitions which are in turn defined by the preceding tokens. A change could take place as a result of a change in polarity of an \#if, the explicit change of \#define or \#undef, or changing contents of an included header file. If every macro that was expanded during preprocessing of this ppinc has the same definition as before, and every identifier that was not defined as a macro name before is still not defined, the ppinc will be unchanged.
During non-incremental preprocessing, each identifier is looked up in the macro definition table. A journal of all the macro lookups is attached to each ppinc (repetitions are ignored). The lookups in the journal are played out. If all the resulting locations are found to be the same, the ppinc remains valid.

This algorithm does exactly as many macro lookups as would reprocessing the source. It never touches most of the tokens (assuming a small change in the source was made). The algorithm described so far is fast only if macro lookup is much quicker than the rest of preprocessing.

The whole macro table has a global status (a.k.a. Super Clean Bit) which is set clean at the start of preprocessing. It is set dirty when a new \texttt{define} or \texttt{undef} ppinc is inserted or deleted. The Super Clean Bit is used to avoid checking ppinc journals in non-directive ppincs. If the Super Clean Bit is set clean, then no changes to the macro table have occurred. The status of identifiers in a clean ppinc have therefore not changed so the ppinc may be checked without macro table lookups.

\section*{Tokincs, ppincs, and Skiplists}

The tokincs from a source file are connected by the navigation functions (named \texttt{first}, \texttt{next}, \texttt{prev} ...) The ppincs are connected by similarly named functions. In our implementation there is a list structure behind this facade. Each ppinc is also connected to its starting tokinc. It is possible, and even common, for an .h file to be included many times in one .c file. Typically the C programmer protects against this eventuality with bracketing \texttt{ifdef} directives so that the main body of each .h file is skipped after the first successful \texttt{include}. A consequence is that two different ppincs may point to the same tokinc, and either or both ppincs may be clean with respect to that tokinc. Because of this, a simple clean/dirty bit is not enough information and time-stamps must be used for both kinds of increments.

In our earlier paper on incremental scanning, the problem of reestablishing the navigation links after a change turned out to look much like file differencing. All changes can be expressed as a sequence of inserts and deletes. Before preprocessing is attempted, the tokincs, new and old, have been relinked. In walking the ppincs in file order, repreprocessing must take place when some ppinc no longer points to any tokinc—indicating the old tokinc has been deleted and therefore this ppinc is stale. Crossing to the tokincs from the previous (and therefore known to be valid) ppinc, and stepping down the tokincs reveals the next tokinc that must correspond to the start of a ppinc. Now one must find out if any existing ppinc points to this tokinc. If so, every intervening ppinc is stale, is discarded, and the links reestablished. If not, a new ppinc is created and the process is repeated.

Traditional C preprocessors keep the association of post-preprocessing text and its file origins by placing the equivalent of \texttt{#line} directives in the preprocessor output. In the incremental preprocessor, direct association between
ppins and tokines provides the locator information. This same information is also used to implement \_LINE\_.

Performance measures on the solution described so far showed that incremental preprocessing, although very fast, was still the bottleneck in fast turnaround. The problem was checking all those ppins, especially the ones from included header files.

We therefore implemented skiplists [2]. The implementation required a second set of navigation links, initially providing for skipping down the ‘next’ lists in chunks of some large number of ppins per step. The first set of links is unchanged by this superposition of a second access method. There is a journal for each super-chunk which merges all the journals of the skipped ppins. To do a skip, the merged journal is played out. If every check is passed, all the ppins are valid and can be skipped in one step. If any check fails, we drop back to the finer granularity and do the ppins one at a time for a while. The super-chunks are always broken whenever there is a situation that requires active preprocessing, such as \#include, \#if, or end of an included file.

Obviously, large steps in the header files were almost always rewarded by successful skips. We allowed change activity to dictate the location of breaks as the skiplists evolved. This made the skips get shorter and shorter but also avoided doing the journal checks twice, which a failing skip test always required. The essential point was to keep the header file processing costs to a minimum.

Diagnostics and Error Recovery

When the preprocessor detects an error, it can issue a diagnostic in the usual way. Source locators are somewhat complex because the problem might well originate in a command line directive, or even tokens that were manufactured by \# or \#
.

In an incremental preprocessor, there is good reason to report one error and stop. The ppins downstream of the error are probably OK and forging ahead after an error has a good chance of preventing their reuse. Our decision was to not proceed beyond any fatal error detected during scanning or preprocessing. They are rare anyway, and our turnaround was good, but the “Voice of the Customer” was not enthusiastic about our decision.

Summary

It is feasible to separate the preprocessor from a C compiler so that it can be used for other reasons. This is functionally equivalent to the traditional -E flag except that it does not require source text output and can use tokenized input if it is available.

A preprocessor can be made incremental at approximately the granularity of source lines. The incremental preprocessor is fast, and enables incremental parsing. The amount of state that must be kept in active memory is several
times larger than the original source. To skip a lot of processing, you must keep a lot of reusable results around. The size of the saved state determines the limits of this technology.

Combining these capabilities provides a useful component for analyzing C source.

References


William McKeeman is a Senior Consulting Engineer for Digital. He has co-authored several books and has published papers in the areas of compilers, programming language design, and programming methodology. His current technical interests are studying and improving compile speed and responsiveness and the application of Software Engineering techniques to small programming projects. He can be reached at mckeeman@tle.dec.com.

Shota Aki is a Principal Software Engineer at Digital. He was a principal developer of the VAX APL interpreter. His current interests are in the areas of CASE, Analysis and Design Methods, Compilers, and Tools. He can be reached at aki@tle.dec.com.