A Stand-Alone C Preprocessor

Kenneth Walker

Department of Computer Science, The University of Arizona

1. Basis for Preprocessor Features

This preprocessor is based on the proposed ANSI C standard, but does not strictly conform to the standard. The major enhancement to the standard is the implementation of *multi-line* macros, which may contain embedded preprocessor directives.

2. Standard Features

This preprocessor supports the standard preprocessor directives:

```
#include
#define
#undef
#if
#ifdef
#ifndef
#elif
#else
#endif
#line
#error
#pragma
#
```

It supports the standard macros:

```
__LINE__
__FILE__
__DATE__
__TIME__
```

See below for a discussion of __STDC__. In addition, the preprocessor supports the standard features of trigraph replacement, deletion of back-slash new-line pairs, macro replacement, token concatenation, stringizing, and concatenation of adjacent strings.

3. Multi-Line Macros

The multi-line macro facility introduces two new preprocessing directives: #begdef and #enddef. These directives bracket the body of the multi-line macro. The #begdef directive is modeled after the #define directive but does not contain the body of the macro. As with standard macros, there are both object-like and function-like multi-line macros. This distinction is determined by whether the macro is defined with an argument list or not. The two forms of #begdef are

#begdef *macro-name*
#begdef *macro-name* (*[parameter-list]*)

A multi-line macro may contain embedded preprocessor directives, including other multi-line macro definitions. #line directives are executed when they are encountered, but all other directives are stored in the macro and executed whenever the macro is expanded. As an example of using this facility, it is possible to write a macro which will generate different versions of a function based on arguments to the macro. The macro GenFree in the following example illustrates this. The context for this example is a program that maintains free lists of several structures. Each structure is assumed to contain a next pointer. The free routines for some structures may be called with NULL values, while others never will be.

```
#define True 1
#define False 0

#begdef GenFree(func, struct_name, checked)
void func(x)
struct struct_name *x;
{
#if checked
   if (x == NULL)
      return;
#endif

   x->next = struct_name ## _f_lst;
   struct_name ## _f_lst = x;
}
#enddef

struct item1 *item1_f_lst;
struct item2 *item2_f_lst;

GenFree(free_item1, item1, True)
GenFree(free_item2, item2, False)
```

If this source is in the file t.c, then the result of running the preprocessor on t.c is

```
#line 17 "t.c"

struct item1 *item1_f_lst;
struct item2 *item2_f_lst;

#line 4 "t.c"

void free_item1(x)
struct item1 *x;
{

   if (x == NULL)
      return;


   x->next = item1_f_lst;
   item1_f_lst = x;
}
```

```
#line 4 "t.c"

void free_item2(x)
struct item2 *x;
{
#line 12 "t.c"

   x->next = item2_f_lst;
   item2_f_lst = x;
}
```

4. Other Deviations from the Standard

The preprocessor fails to conform the the proposed ANSI standard on several other points. These points are minor and most programmers need not be concerned with them, but they are described here for completeness. The standard requires that a new-line within a macro invocation be treated as a *normal* white-space character. Presumably this means that such a new-line does not terminate a preprocessor directive. However, this preprocessor recognizes directives after back-slash new-line pairs and comments have been removed, but before macro invocations are recognized. For example, in the code fragment

```
#include file
(a)
```

file may not be a function-like macro, because the #include directive ends with the new-line following file.

Preprocessing directives are also recognized before stringizing operators. Therefore, in the macro

```
#begdef m(include)
#include "a.h"
#enddef
```

#include is a directive, not a stringized parameter. If it had occurred somewhere other that at the beginning of a line, it would have been a stringized parameter. Note that this is only relevant to multi-line macros. Another point only relevant to multi-line macros is the fact that stringizing and token concatenation ''belong'' to the outer most of nested definitions and are executed when that macro is expanded. So that

```
#begdef f1(a)
#define f2(b) a ## b
#enddef
f1(x)
f2(y)
```

produces xb, not xy.

White space is stripped from the beginning and end of macro arguments. As required by the standard, arguments are expanded in isolation before being substituted into the macro (except that stringizing and token concatenation are done on unexpanded arguments), where they are rescanned. Because a multi-line macro can contain macro defines and undefines, the expansion of the multi-line macro may have the side effect of changing macro definitions. The pre-expansion of arguments is done once before the macro body is scanned, so these side effects do not affect the pre-expansion. However, the side effects can affect arguments when they are rescanned in the body of the macro. Therefore, the expansion of arguments may be different in different parts of the multi-line macro.

To be standard conforming, the macro __STDC__ must be predefined to 1. This indicates the compiler is standard conforming. This preprocessor is used both with compilers that are standard conforming and with those that are not. For this reason, the macro is left undefined. It may be defined to 1 through a command line option when the preprocessor is used with a standard conforming compiler. There is an additional predefined macro __RCRS__ which indicates the number of levels of recursion allowed for macros. Its definition must be an integer. It is given an initial value of 1, indicating that only one call to a given macro may be in effect at a time, that is, that recursion is

not allowed. If it is undefined, unlimited recursion is allowed. In the standard, recursion is never allowed, but in multi-line macros recursion can be terminated with conditional compilation so it makes sense to allow it. For example, the following macro repeats a piece of text a specified number of times.

```
#undef __RCRS__

#begdef rep(text, n)
#if n > 0
text
rep(text, (n-1))
#endif
#enddef
```

Note that n is an expression not a number. If the macro is called with n of 4, it will terminate with the conditional

```
#if ((((4-1)-1)-1)-1) > 0
```

While the maximum level of recursion is exceeded for a macro, further expansion of the macro is inhibited; the name is treated as if it were undefined. This is consistent with the standard.

Arithmetic in conditional directives is always done using long ints. However, to be standard conforming, a preprocessor must at least distinguish between intermediate results that are long and those that are unsigned long. The size of a long int is determined by the C compiler used to compile the preprocessor.

Include files are put in-line before processing, not afterwards. Therefore, macro invocations and conditional inclusion may extend across include files. This is not allowed in the standard.

#line directives are always executed when they are encountered, even in branches of conditional inclusion which are not selected.

The standard does not allow vertical tabs within a preprocessing directive. This preprocessor does not enforce this. The standard does not allow tokens between an #else or #end and the following new-line. This preprocessor ignores such tokens, so it can be used with existing system include files which violate the standard.

5. "Implementation Defined" Behavior

The standard designates some aspects of the preprocessor as being implementation defined. One example of this is the interpretation of the #pragma directive. This implementation ignores all pragmas.

Whether the spelling of white space is retained is implementation defined. This is not significant if the output of a preprocessor goes directly into a compiler. However, the output of this stand-alone preprocessor goes into a file. Comments are normally replaced by a single space character and other white space is kept as is, except that white space may be added or deleted around any generated #line directives. There is a command-line option for retaining comments. When a directive is deleted from the text after being executed, the following white space up through the first new-line is also deleted.

The interpretation of character constants as integers is implementation defined. This is relevant to expression evaluation for conditional inclusion. When possible, this interpretation is based on the compiler used to compile the preprocessor. An example of this is whether an octal character constant with the high order bit set is a positive or negative number (determined by whether char is signed or unsigned).

The interpretation of wide characters is implementation defined. This preprocessor treats wide characters the same as normal characters. Only the first byte of a multi-byte character is used during expression evaluation; the rest are ignored.

The standard specifies that character strings be converted to internal form before concatenation. This insures that the concatenation of strings such as "\12" with "1" results in the two character string "\0121" and not the one character string "\121". Problems only arise when the first string ends with a one or two digit octal constant or ends with a hexadecimal constant and the second string start with a digit from the corresponding base. When these situations arise, the preprocessor converts the character at the end of the first string to a canonical three digit octal form; overflow from hexadecimal constants is ignored. The only other situation where the representation of characters is changed is when the target compiler does not support one or more of the standard escape sequences (for example,

'\a'). In this case, the escape sequence is converted to the three digit octal form appropriate to the target compiler.

The syntax of include file names and the search for include files is implementation defined. This preprocessor uses Unix path conventions. This is sufficient for most systems, but may have to be changed for others. For include file names that start with '/', this preprocessor uses the name as the complete path name of the include file. For other names enclosed in quotes, the search starts in the location (directory in Unix terminology) of the file containing the #include directive, proceeds to the location of the file which included the file containing the #include directive, and so on back to the location of the primary source file. If the include file is not found in any of those locations, the search continues through locations specified on the command line, then finishes with the location(s) of system include files. Note that this preprocessor is meant to be used with a variety of C compilers. The location of system include files varies among these compilers; this location is determined when the preprocessor itself is compiled. The search for file names enclosed in angle brackets only includes locations on the command line and the location(s) of system include files.

When the file name on an include directive is the result of macro expansion and it is the form enclosed in angle brackets, the mapping of tokens into the name is implementation defined. This implementation converts each sequence of white space into a single space character and uses the exact spelling of other tokens for constructing the file name.

6. Command Syntax with Standard Options

The preprocessor is invoked with the command

pp *{ option }* { file }**

option ::= –C |
–P |
–D *identifier [= [macro–body]]* |
–U *identifier* |
–I *path* |
–o *file*

If no file is specified or '–' is specified for a file, the preprocessor reads from standard input. Normally the preprocessor replaces each comment with a space; the –C option cause the preprocessor to retain the spelling of comments. The preprocessor normally outputs #line directives; the –P option suppresses the output of #line directives (however, white space is still adjusted as if #line directives were output).

The –D option specifies the definition of an object-like macro that is to be put in effect before preprocessing starts. *identifier* is the name of the macro. If nothing else is given, the macro is given the value 1. If only an equal sign is given, the macro is defined, but null. If a macro body is given, white space is stripped from the front and the end.

The –U option undefines predefined macros before preprocessing starts. The macros __STDC__ and __RCRS__ may be undefined. In addition, there may be predefined macros that are specific to the C compiler the preprocessor is being used with. All such macros may be the subject of the –U option. These macros are determined when the preprocessor is compiled.

The –I option specifies a directory to look in when searching for include files. These directories are searched in the order given on the command line. The overall search order for include files is explained above.

By default, the output of the preprocessor goes to standard output. The –o option directs the output to the specified file.

Other options may be defined when the preprocessor is used with specific compilers. These are typically needed so the preprocessor knows what macros to predefine. For example, a compiler on an IBM PC may predefine different macros depending on the memory model specified for compilation. These additional options are determined when the preprocessor is compiled and closely mimic the corresponding options of the compiler.

Appendix A: Porting

The preprocessor is written is C and is organized to be part of the Icon source distribution. When porting the preprocessor, it is necessary to supply a define.h file as described in the appropriate Icon porting document. The preprocessor also uses other shared .h files. These include config.h, cpuconf.h, and proto.h. Porting may require adding conditional code or making other changes to these files. Additional changes may be needed in common routines, in particular, time.c.

Three files for the preprocessor itself may need changes. These files contain code that relates to setting up initial defined constants and locating system include files. These features must match those of the compiler the preprocessor will be used with. System dependent code in pmain.c determines what options beyond the standard ones this version of the preprocessor will recognize.

Functions in p_init.c are responsible for establishing predefined constants. This may require processing non-standard command line options, environment variables, configuration files, etc. Functions in files.c are responsible for locating system include files and dealing with variations in path syntax. Locating system include files may require processing non-standard command line options, environment variables, configuration files, etc.

Appendix B: Unix

No additional options are supported for this system.

The macro unix is always predefined to 1. Any of the following that are defined when the preprocessor is compiled are also predefined to 1.

```
i286
i386
mc68000
mc68010
mc68020
sparc
sun
vax
```

System include files are located in /usr/include/.

Appendix C: MS-DOS, Microsoft C 5.0

Additional options supported for this system are -AS, -AC, -AM, -AL, -AH, -Za, and -J. These options may be supplied through the CL environment variable or on the command line. If supplied through the environment variable, they may be introduced with / rather than -. Other Microsoft C options which effect preprocessing are also recognized from the environment variable. These include -D, -U, -u, and -I options.

The macros MSDOS and M_I86 are always predefined to 1. The following table shows the options and the macro names they cause to be predefined to 1.

-AS	M_I86SM
-AC	M_I86CM
-AM	M_I86MM
-AL	M_I86LM
-AH	M_I86LM, M_I86HM
-Za	NO_EXT_KEYS
-J	_CHAR_UNSIGNED

The search path found in the environment variable, INCLUDE, (if defined) is used to to locate system include files.

Appendix D: MS-DOS, Turbo C 2.0

Additional options supported for this system are -mt, -ms, -mm, -mc, -ml, -mh, and -p. These options may either be supplied on the command line or in the file turboc.cfg. The file used is the first one in the search path with this name. Other Turbo C options which effect preprocessing are also recognized from this file. These include -I (see below), -D and -U options.

The macro __MSDOS__ is always predefined to 1. The macro __TURBOC__ is always predefined to the value it had when the preprocessor was compiled. The following table shows the options and the macro names they cause to be predefined to 1.

-mt	__TINY__
-ms	__SMALL__
-mm	__MEDIUM__
-mc	__COMPACT__
-ml	__LARGE__
-mh	__HUGE__
-p	__PASCAL__

If -p is not given, __CDECL__ is predefined to 1 instead of __PASCAL__.

The -I options in the file turboc.cfg are used to locate system include files.