High Quality C Preprocessor MCPP

Kiyoshi Matsui
kmatsui@t3.rim.or.jp

March 19th, 2005

Abstract

There has been a long history of confusion about the specifications of C preprocessors. Although, after C90, preprocessor specifications tend to converge to the standard, so called standard-conformant preprocessors still often behave wrong. Moreover, almost every existing preprocessor is too reticent; it does not have adequate capability to check source code. MCPP is a free portable C preprocessor and provides a validation suite to make thorough tests and evaluation of C/C++ preprocessors. When this validation suite is applied to various preprocessors, MCPP achieves a prominent result; MCPP not only has the highest conformance but also provides a variety of accurate diagnostic messages. MCPP thus allows users to check almost all the preprocessing problems of source code.

1 Introduction

There has been a long history of confusion about the specifications of C preprocessors. Although, after C90 (C89) [2–5], preprocessor specifications tend to converge to the standard, so called standard-conformant preprocessors still often behave wrong. It can be said that preprocessing is a rather immature field compared to compiling.

Behind this, there lies a background that preprocessing specifications before C90 were very ambiguous. C90 gave the first overall definition of C preprocessing, going back to the principles of "what is preprocessing?". C90, however, has some compromising parts with the historical negative inheritance, which have not been cleared even by C99 [6–8]. Moreover, most of the existing preprocessors seem to have grafted each specifications of the Standard one by one without C preprocessing principles being made clear, thus prolonging the problems.

Against these backgrounds, not a few C programs have preprocessing-level problems, such as unnecessarily implementation-dependent code lacking of portability. One of the reasons for existence of the preprocessing phase in C is to provide greater portability, however, in fact, preprocessing itself has impaired it.

I have been developing a C preprocessor for a long time. My work so far has already been released as cpp V.2.0 and V.2.2 in August 1998 and in November 1998, respectively. During the course of updating the software to V.2.3, it was selected as one of the "Exploratory Software Projects" for year 2002 and for year 2003 by Information-Technology Promotion Agency (IPA), Japan. [1] V.2.3 and V.2.4 were released as the results of the project. Now, I release this V.2.5. My cpp is called MCPP (Matsui CPP) to distinguish it from other cpps.

I personally boast of MCPP as being number one C preprocessor now available in the world, not merely from self-praise, but because of its big feature that its behaviors have been completely verified using "Validation Suite", which I developed in parallel with MCPP.

Another feature is that it provides a lot of diagnostic messages that allows you to check almost

all the preprocessing problems in source programs and to increase source portability.

This document is organized as follows:

Section 2: Provides an overview of MCPP.

Section 3: Introduces briefly the basic specifications of C preprocessing.

Section 4: Introduces MCPP's accompanying Validation Suite and shows data to compare Standard conformance level and qualities with other preprocessors.

Section 5: Shows examples of bugs in compiler-system resident preprocessors.

Section 6: Describes source checking by MCPP of the real world programs.

Section 7: Discusses C preprocessing principles and how to implement them.

Section 8: Describes the current version of MCPP and future update plans.

2 MCPP Overview

MCPP has the following features:

1. Has the highest conformance to C Standards because MCPP aims at becoming a reference model of C and C++ preprocessors. MCPP provides run-time options to enable C99 and C++98 behaviors [9], needless to say C90.

2. Provides a validation suite that allows you to test C or C++ preprocessors themselves in great detail and comprehensively.

3. Provides a lot of diagnostic messages of more than one hundred types to pinpoint a problem in source code. They are divided into several classes. Messages of which class are displayed is controlled by run-time options.

4. Provides the #pragma directives to output various debugging information. The directives allow you to trace tokenization and macro expansion, to output a macro definition list and etc.

5. MCPP's multi-byte character processing can handle a variety of Japanese EUC-JP, shift-JIS and ISO-2022-JP, Chinese GB-2312, Taiwanese Big-5 and Korean KSX-1001 encodings as well as UTF-8. For the compiler-proper which cannot recognize shift-JIS or Big-5, MCPP can complement it.

6. Processing speed is not so slow; it can be used not only for debugging purpose but also for daily use. Since MCPP is so developed that it can operate in 16-bit environments, it can work properly in a system with a small amount of memory.

7. MCPP's source is portable. MCPP is so designed that it can generate a preprocessor to be used replacing a compiler system resident one on UNIX-like systems or DOS/Windows by modifying some settings in header files on compilation of MCPP. The portability MCPP source provides is so wide that it can be compiled not only with any C90, C99 or C++98 conformant compiler systems, but also with $K\&R^{1st}$ ones before C90.

8. In addition to "Standard" mode, which conforms to C90, C99 and C++98 Standards, MCPP allows you to generate a preprocessor in various behavioral modes. You can generate the one based on the $K\&R^{1st}$ specifications too. The Standard mode MCPP has run-time options to specify the version of the Standards, moreover, it has an option of what I call "post-Standard" mode in which all the problems in C Standards are cleared. The pre-Standard mode one also has an option of the Reiser model cpp mode.

9. On UNIX-like systems, a configure script can be used to automatically generate a MCPP executable. If GNU C testsuite has been installed, most of the testcases of validation suite can be automatically executed by 'make check' command.

10. MCPP is an open source software. Under the BSD-style license, all of the sources, docu-

ments and the validation suite are provided open.

11. Sufficient documentation is provided both in Japanese and in English. The English versions was translated by Highwell inc.(Tokyo) [19] from the Japanese ones at "Exploratory Software Projects" and have been revised by the author.

    (a) INSTALL – Describes how to configure and make MCPP.

    (b) mcpp-summary.pdf – This summary document.

    (c) mcpp-manual.txt: Users Manual – Describes how to use MCPP, its specifications and meanings of diagnostic messages. Also suggests how to write portable source code.

    (d) mcpp-porting.txt: Porting Manual – Describes how to port MCPP to particular compiler systems.

    (e) cpp-test.txt: Validation Suite Manual – Also explain C Standards. It indicates contradictions and deficiencies in Standards themselves and proposes alternatives. It also shows the results of applying Validation Suite to several preprocessors.

3 Basic Specifications of Preprocess

Before entering into the subject, let me summarize the basic specifications of C/C++ preprocessing.

3.1 Procedure of Preprocess

The procedure of preprocessing was not at all described in $K\&R^{1st}$, hence had been the source of many confusions. C90 made clear the procedure by specifying the translation phases as follows:

1. Map source file characters to source character set, if necessary. Replace trigraphs.
2. Delete ¡backslash¿¡newline¿ sequences, splicing physical source lines to form logical source lines.
3. Decompose source file to preprocessing-tokens and white space sequences. Replace each comment by one space character. ¡newline¿s are retained.
4. Execute preprocessing directives, expand macro invocations. Process header file named by #include directive from phase 1 through phase 4, recursively.
5. Convert from source character set to execution character set, including escape sequences in string literals and character constants.
6. Concatenate adjacent string literals.
7. Convert preprocessing-tokens into tokens and compile.
8. Link.

After that, C99 added processing of `_Pragma()` operator in phase 4, also added and modified a few words. Nevertheless, the above outline was not changed.

C++98 inserted 'instantiation' phase after phase 7, and appended a so-called UCN specification, that is to convert source file character not in the basic source character set to universal character name (UCN) in phase 1, and convert it again to execution character set in phase 5.

Of these translation phases, from phase 1 through phase 4 are usually called preprocessing.

3.2 Diagnostics and Documentation

The definitions of diagnostics and document are virtually all the same among C90, C99 and C++98 except some difference of wording, and defined as follows:

Implementation shall issue diagnostic message, if a translation unit contains a violation of any

Table 1: Number of Test Items and Scores covered by Validation Suite V.1.5

		Number of Test Items	Highest Score
Standard conformance	K&R	31	166
	C90	140	432
	C99	20	98
	C++98	9	26
Quality issues	diagnostics	47	74
	others	18	164
total		265	960

syntax rule or constraint. It is implementation-defined how a diagnostic is identified.

Implementation shall document its choice on any implementation-defined behavior.

4 Results of Applying Validation Suite to Various Preprocessors

Another problem involved in preprocessor development is how to verify preprocessor's behavior and its quality. Wrong behavior or poor quality of compiler systems is, of course, out of question. However, in fact, many problems were detected in existing preprocessors when they were tested with Validation Suite. As a part of MCPP development, I developed Validation Suite and released it with MCPP. Validation Suite provides quite a lot of test items to measure various aspects of a preprocessor objectively and comprehensively as much as possible.

As shown in Table 1, Validation Suite V.1.5 contains as much as 265 test items, of which, 230 cover preprocessor behaviors and 35 documentation and quality evaluation. Score of each test item is weighted. The lowest score of each item is all 0. "Standard conformance" includes evaluation of diagnostic messages and documentation, as well as of behaviors. "K&R" means specifications common between $K\&R^{1st}$ and C90. "Standard conformance" for C99 and C++98 deals with new specifications that do no exist in C90. "Standard conformance" covers all the specifications of Standards.

"Quality: diagnostics" deals with diagnostic messages that are not required by C Standards. "Quality: others" deals with execution options, #pragmas, multi-byte character handling, processing speed, etc.

Table 2 shows the summary of results of applying Validation Suite V.1.5 to several compiler systems. The table shows compiler systems in a chronological order.

*1 DECUS cpp: Original version developed by Martin Minow, which was slightly revised by the author and compiled by Linux/GNU C 3.2. [10]

*2 Borland C 4.0: Japanese version for 1993. Tested on MS-DOS version [11]

*3 MCPP 2.0: Free software developed by the author. Was rewritten based on DECUS cpp. Was ported to various compiler systems, such as FreeBSD/GNU C 2.7, DJGPP V.1.12, WIN32/Borland C 4.0, MS-DOS/Turbo C 2.0, LSI C-86 3.3, and OS-9/09/Microware C. Although MCPP V.2.0 allows generation of a preprocessor in various modes, the 32-bit system standard mode was used for this test (compiled by GNU C 2.95.3 on Linux).

*4 Borland C 5.5: Japanese version. [12]

*5 GNU C 2.95.3: Bundled in VineLinux 2.6 and 3.1, FreeBSD 4.4 or CygWIN 1.13.

Table 2: Validation Results of Each Preprocessor

No.	Preprocessor	year/month	conformance					quality		overall
			K&R	C90	C99	C++ 98	total	diag	others	evaluation
1	DECUS cpp	1985/01	150	240	0	0	390	15	78	483
2	Borland C 4.0	1994/12	164	366	14	6	552	14	69	633
3	MCPP 2.0	1998/08	166	430	58	10	664	68	125	857
4	Borland C 5.5	2000/08	164	368	20	6	558	18	72	646
5	GNU C 2.95.3	2001/03	166	404	56	6	632	23	113	768
6	GNU C 3.2	2002/08	166	419	86	20	691	33	117	841
7	ucpp 1.3	2003/01	166	384	88	9	647	25	88	760
8	Visual C 2003	2003/04	156	394	43	15	610	20	83	711
9	LCC-Win32 3.2	2003/08	160	376	18	6	560	18	96	674
10	Wave 1.0.0	2004/01	140	338	53	18	549	20	79	648
11	GNU C 3.4.3	2004/11	166	415	87	20	688	39	120	847
12	MCPP 2.5	2005/03	166	432	98	22	718	74	134	926
	highest score		166	432	98	26	722	74	164	960

*6 GNU C 3.2: Compiled by the author under VineLinux 2.6 and FreeBSD 4.7. [13]

*7 ucpp 1.3: Portable free software developed by Thomas Pornin. A stand-alone preprocessor. [14]

*8 Visual C++ 2003: Visual C++ .net 2003. Microsoft. [15]

*9 LCC-Win32 3.2: Shareware, with source code available, developed by Jacob Navia et al. Dennis Ritchie's C90-conforming preprocessor is used as its preprocessing part. [16]

*10 Wave 1.0.0: Free software written by Hartmut Kaiser. Implemented using "Boost C++ preprocessor library" written by Paul Mensonides et. al. Tested about an executable on WIN32. [17]

*11 GNU C 3.4.3: Compiled on VineLinux 3.1.

*12 MCPP 2.5: From V.2.0 onward, MCPP has been ported to Linux/GNU C (2.95, 3.2, 3.3, 3.4), FreeBSD/GNU C (2.95, 3.2, 3.4), CygWIN 1.13, LCC-Win32 3.2, Borland C 5.5, Visual C++ .net 2003 and Plan 9 ed.4/pcc. [18]

As shown in the table, MCPP is by far the best in every aspect. Its conformance is perfect except it does not implement the C++98 queer specification to convert multi-byte character to UCN. It has more leads over other preprocessors on quality issues, such as abundant and accurate diagnostic messages, detailed documentation, useful execution options, #pragmas, handling of various multibyte character encodings, and portability.

According to the table, the second best preprocessor to MCPP is GNU C/cpp. GNU C/cpp presents almost no problems as long as it processes C90 conforming sources. However, GNU C/cpp still has the following problems, except for some unimplemented C99 and C++98 specifications, which will be implemented over time:

1. Diagnostic messages are insufficient. With the `-pedantic -Wall` option, many problems can be checked, but there still remain a lot of unchecked problems.
2. It provides little functionality to output debugging information.
3. Documentation is insufficient; there are many unclear or undocumented specifica-

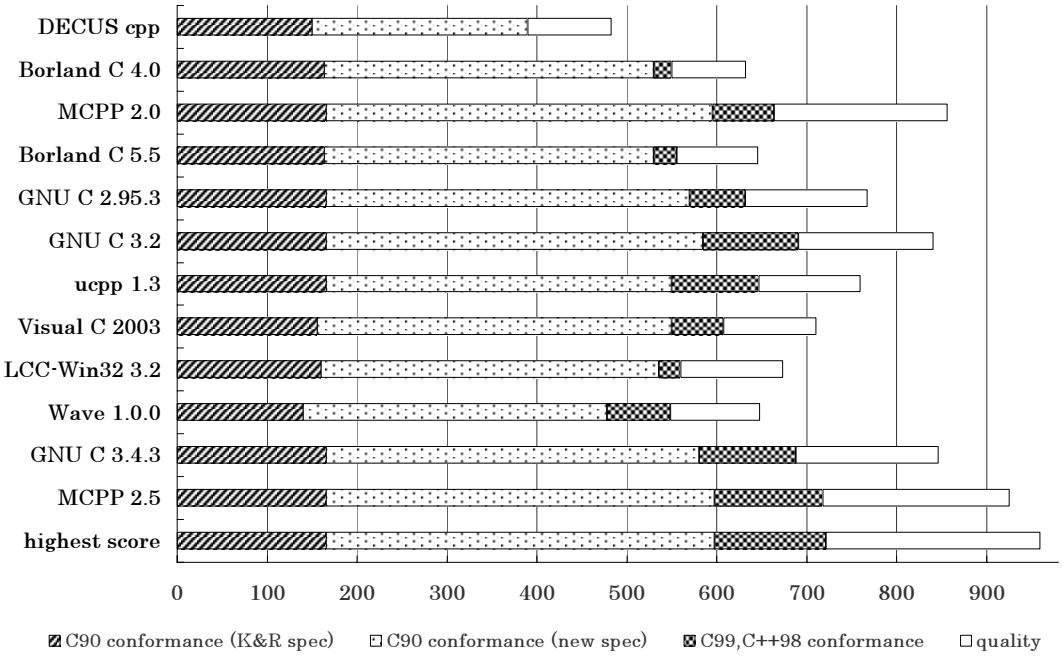

Figure 1: Validation Results of Each Preprocessor

tions. The problem is that the GNU C V.2/cpp has some traditional behaviors when `-traditional` option is not specified.

4. GNU C/cpp uses its own specifications that are inconsistent with C Standards. Extended specifications should be implemented with #pragma.

Compared with GNU C V.2/cpp, GNU C V.3/cpplib has been much improved in these aspects, but is still insufficient.

MCPP is inferior to GNU C/cpp only in processing speed.

Other preprocessor has much more problems than GNU C/cpp. The following problems are commonly found in many preprocessors.

1. As for the new specifications of C99 and C++98, most of the preprocessors implement only half of them.
2. Most preprocessors do not provide diagnostics sufficiently.
3. Most preprocessors provide few diagnostics on portability matters.
4. It is not uncommon to see off-target diagnostics issued.
5. Most preprocessors do not provide document sufficiently.
6. Most preprocessors cannot handle more than 1 or 2 multi-byte character encodings.

Moreover, at least 1 or 2 bugs are found in most preprocessors.

5 Examples of Preprocessor Bugs and Erroneous Specifications

Each preprocessor contains various bugs and erroneous specifications, only some of which this section cites. The samples are shown in figure 2.

```
example-1
    #define _VARIANT_BOOL   /##/

example-2
    _VARIANT_BOOL bool;


example-3
    #if     MACRO_0 && 10 / MACRO_0

example-4
    #if     MACRO_0 ? 10 / MACRO_0 : 0

example-5
    #if     1 / 0


example-6
    #include    <limits.h>
    #if     LONG_MAX + 1
```

Figure 2: Sample of Preprocessor Bugs

5.1 Comment Generating Macro

Example-1 is a macro definition that is actually found in the Visual C++ .net system header. This definition is used as shown in example-2. This code expects `_VARIANT_BOOL` to be expanded into `//`, commenting out that line. Actually, Visual C's cl.exe processes this line as expected.

However, `//` is not a preprocessing-token. In addition, macro definitions should be processed and expanded after sources are parsed into tokens and a comment is converted into one space. Therefore, it is irrational for a macro to generate comments. When this macro is expanded into `//`, the result is undefined since `//` is not a valid preprocessing-token.

This macro is, indeed, out of question, however, it is Visual C/cl.exe, which allows such an outrageous macro to be processed as a comment, should be blamed. This example reveals the following serious problems this preprocessor has:

1. Preprocessing is not token-based but character-based.
2. Preprocessing procedure (translation phases) is implemented arbitrarily and lacks in logical consistency.

5.2 Expressions That Should Be Skipped Causes an Error

The #if expressions in example-3 and 4 are correct expressions. These expressions are so carefully written that a division operation is carried out only when a denominator is not zero. However, some compiler systems perform a division when `MACRO_0` is zero and cause an error. Example-3 used to cause an error in many compiler systems, but now it is processed properly. Example-4 still causes an error in Visual C++, which shows that its preprocessor does not implement basic C specifications regarding evaluation of expressions properly.

On the other hand, Borland C 5.5 issues a warning to both example-3 and 4, which may not be definitely wrong. However, Borland C 5.5 issues the same warning to a division using a zero denominator shown in example-5. This means Borland C 5.5 cannot tell correct source code from wrong code. Turbo C issued the same error message to both correct expressions and incorrect ones that may cause a zero division error. Borland C simply degrades the error message to a warning. This could not be called non-conformant, but indicates a lack of careful consideration in and poor quality of diagnostic messages.

5.3 Overflow is Overlooked

The constant expression in example-6 causes an overflow in C90. Most compiler systems do not issue a diagnostic message to this overflow. Only Borland C and Ucpp are quite inconsistent about this; they issue a warning to some cases, but not to most.

6 Why Is Source Code Check by Preprocessors Required?

Now, we will see source code checking by MCPP of the real world programs, taking examples of glibc

and others.

Not a few C programs have preprocessing-level problems; there are ones that are content with successful compilation in a particular compiler system and lack of portability, ones that are unnecessarily tricky, and ones that are still based on the specifications of a particular compiler system before C90. These sources will impair portability, readability and maintainability, and, what is worse, they will be likely to provide a hotbed of bugs. Although, in many cases, it is easy to rewrite such questionable sources into portable and clear ones, however, they are often left as they are.

One of the reasons for the existence of such sources is that preprocessing specifications before C90 were very ambiguous, which still leaves a trail even now when C99 Standard has been already established. Another reason is that the existing preprocessors are too reticent; since they pass questionable sources without issuing messages, problems remain unnoticed.

6.1 How Much Do Preprocessors Affect Sources?

By replacing a compiler system-resident preprocessor with MCPP, almost all the preprocessing problems in source programs, ranging from potential bugs and Standard violations to portability problems, can be identified.

Since MCPP V.2.0, I have reported the results of applying MCPP to FreeBSD 2.2.2R (May 1997) kernel and libc sources. Libc sources have almost no problems, but some kernel sources have some, although such sources account for only a small portion of the total number of source programs. Many of the problems were not originated in 4.4BSD-lite but written during porting to FreeBSD and enhancement.

When I applied MCPP V.2.3 then under development to preprocess Linux/glibc 2.1.3 (September 2000) sources, I found a lot of problems. These problems were frequently found in the programs that use traditional preprocessing specifications in UNIX-like systems and those that use GNU C/cpp's own or undocumented specifications. I think GNU C/cpp's default approval of such undesirable source programs without issuing a message not only preserves them but also produces new ones. It is more problematic that such illegal coding is not necessarily found in old sources only; it is sometimes found in newly developed sources. Sometimes, similar problems are found even in system headers.

On the other hand, there are some improvements; for example, nested comments, a Standard violation that was frequently found by the middle of 1990s on UNIX-like systems, are no longer found. This is because GNU C/cpp no longer allowed them. This indicates how much a preprocessor affects sources coding.

6.2 Sample Glibc Source Code Fragment

To see some preprocessing problems, let me take an example of a glibc 2.1.3 source code fragments used in VineLinux 2.1 (i386). The samples are shown in figure 3.

6.2.1 Multi-line String Literal

Example-7 shows this case. This traditional specification does not need to be used at all, but it is still used. Makefile sometimes generates this.

The preprocessing directive lines shown here require line splicing, so the code fragment should be written as shown in example-8.

Regardless of directive lines or not, a more general way of coding is to use string literal concatenation as shown in example-9. If this line were not a directive one, line splicing would be, of course, not required.

This way of coding is found in many source files, but, somehow, the old way of writing still remains in some.

```
example-7
    #define ELF_MACHINE_RUNTIME_TRAMPOLINE asm ("\
        .text
        .globl _dl_runtime_resolve
        etc. ...
    ");

example-8
    #define ELF_MACHINE_RUNTIME_TRAMPOLINE asm ("\
        .text\n\
        .globl _dl_runtime_resolve\n\
        etc. ...\n\
    ");

example-9
    #define ELF_MACHINE_RUNTIME_TRAMPOLINE asm ("\t"    \
        ".text\n\t"                                     \
        ".globl _dl_runtime_resolve\n\t"                \
        "etc. ...\n");


example-10
    #define HAVE_MREMAP defined(__linux__) && !defined(__arm__)

example-11
    #if HAVE_MREMAP

example-12
    defined(__linux__) && !defined(__arm__)

example-13
    defined(1) && !defined(__arm__)

example-14
    #if defined(__linux__) && !defined(__arm__)
    #define HAVE_MREMAP 1
    #endif


example-15
    #define CHAR_CLASS_TRANS SWAPU16

example-16
    #define SWAPU16(w) ((((w) >> 8) & 0xff) | (((w) & 0xff) << 8))

example-17
    #define CHAR_CLASS_TRANS(w) SWAPU16(w)
```

Figure 3: Code Fragments from glibc

6.2.2 ∗.S Files That Require Preprocessing

Some assembler sources have preprocessing directives, such as `#if`, and C comments embedded.

It is recommended that the asm() function should be used whenever possible, as shown in example-9, to embed the assembler source part in a string literal, and that not ∗.S but ∗.c should be used as a file name. In this way, directive lines other than #include can be used in the middle of the lines of string literals.

Some assembler sources have a macro embedded, which cannot be dealt with asm(). This type of source is not a C source and essentially should be processed with an assembler macro processor. It is not desirable to use a C preprocessor for this purpose.

6.2.3 Macro Expanded to 'defined'

There is a macro definition shown in example-10 and the macro is used as shown in example-11.

However, the behavior is undefined in Standard C when a #if line have a 'defined' pp-token in a macro expansion result. Apart from it, this macro definition is first replaced as example-12.

Supposing that `__linux__` is defined as 1, and `__arm__` is not defined, it is finally expanded as shown in example-13.

`defined(1)` on a #if expression, of course, is a syntax error. The same thing would happen to GNU C/cpp, if `HAVE_MREMAP` were not on a #if line. However, on the #if line, GNU C/cpp stops macro expansion at example-12 and evaluates it as a #if expression. This specification lacks of consistency in that how to expand a macro differs between when the macro is on a #if line and when on other lines. It also lacks of portability. This code should be written as shown in example-14.

6.2.4 Object-Like Macros Expanded to Function-like Macros

Some object-like macros are defined to be expanded to function-like macro names. These macros are expanded as function-like macros. This happens because the token sequence immediately following the object-like macro invocations are involved in macro expansion. This way of expansion is a traditional specification before C90, which was approved by C90. In that sense, it can be described as providing greater portability. Let me take an example of an object-like macro shown in example-15.

`SWAPU16` is defined as shown in example-16.

What seems to be an object-like macro that is actually expanded as a function-like macro is inferior in readability at least. There is no reason to write in this way. This way of writing originates in an idea of editor-like text replacement, which is not desirable for C function-like macro. This macro should be written as a function-like macro from the beginning, as shown in example-17.

6.2.5 Undocumented Specifications on Environment Variable

This is a problem not of C source but of a Makefile. In GNU C 2/cpp, there is an undocumented specification that if an environment variable `SUNPRO_DEPENDENCIES` is defined and the -dM option is specified, macro definitions in source code are output to the file specified with the environment variable. One of the Makefiles follows this specification. Also, there is another similar environment variable named `DEPENDENCIES_OUTPUT`, which is documented. I wonder why these environment variables need to be used?

In addition to the above, there is more undesirable coding, most of which can be easily written in a clearer and more readable way. The source programs in question account for only a small portion of total number of the Glibc source files that extends to several thousands, however, if GNU C/cpp had issued a warning to such programs, they would have been rewritten already, or written in a quite different way from the beginning.

7 Principles of C Preprocessing and MCPP implementation

Behind the many preprocessing problems identified by MCPP and its Validation Suite, there lies a confusion about principles of C preprocessing. The principles and specifications of C preprocessing before C90 were very ambiguous. C90 gave the first overall definition of C preprocessing, going back to its principles. Most existing preprocessors, however, seem to have implemented each specifications of the Standard one by one without C preprocessing principles being made clear, thus prolonging the problems. Furthermore, C90's own contradictions and ambiguities stemming from the historical background, which have not been revised even by C99, makes the problem more complex.

We can reasonably extract some principles from C90 preprocessing specifications as follows:

1. Preprocessing is token-based in principle.
2. The syntax of macro call with arguments is similar to that of function call.
3. Processing of macros is one of the preprocessing tasks and have no priority over other processing.
4. Preprocessing is the phase of "pre"processing independent from the execution environment, and requires little implementation-defined parts.

Those are also the principles of MCPP implementation.

7.1 Principle on Token-Based Processing

C preprocessing is "token-based" in principle. Since the principle had been ambiguous before C90, an idea of character-based text processing came in. After C90, many preprocessors have overlooked or even allowed themselves to perform character-based text processing, still leaving the problem. What is worse, C90 itself contains some compromising parts with character-based processing, as in the specifications of # operator or header-name token. (For the discussion on this issue, see section 1.7.1 of cpp-test.txt of Validation Suite.)

MCPP has a program structure that strictly follows the token-based preprocessing principle, which is quite different from traditional character-based preprocessing. Other preprocessors seem to aim at token-based processing, but actually character-based processing got mixed in many cases. I think a certain percentage of preprocessor bugs is caused by this.

In Borland C 4.0,5.5 or Visual C++ .net 2003, for example, a token generated by macro expansion is sometimes merged with the proceeding or following one to become one token. This is an example of half-hearted token processing. Many preprocessors do not issue any warning to an illegal token generated by macro expansion because they simply neglect checking a token generated by preprocessing.

7.2 Function-Like Expansion of Function-Like Macros

Expansion of a macro without an argument is rather straightforward. On the other hand, for expansion of macros with arguments, many specifications have been existed historically, thus leading to tremendous confusion about it. Although C90 seems to have put an end to this confusion, it still lingers. For details on this issue, refer to 1.7.6 of cpp-test.txt.

This confusion is due to two factors: Text-based thinking that originates in editor-like text replacement, and the traditional specification on macro expansion, that is, if a replacement list forms the first half part of another argument macro invocation, the succeeding token sequence are involved in rescanning during macro expansion. The example shown in 6.2.4 is one of the

least serious cases. This results from the fact that C preprocessor's traditional implementation happens to have such a deficiency. Is not it a bug specification, although unintentional, which introduced various abnormal macros?

C90 tried put an end to this confusion about how to expand macros with arguments by naming them "function-like macros" and establishing a specification similar to that of a function call. In other words, C90 first intended that function-like macro and function are interchangeable each other. C90 articulated that a macro in an argument is first expanded and then a parameter in a replacement list is substituted with the corresponding argument and that macro expansion in an argument must be completed within the argument. (Before C90, it seems that, in many cases, a parameter is first substituted with an argument and then is expanded during rescanning.)

On the other hand, however, C90 approved the bug specification that succeeding token sequence are involved in rescanning, which violates the function-like processing principle, resulting in prolonged confusion. At the same time, C90 added the stipulation that a macro with the same name should not be re-replaced during rescanning to prevent infinite recursion in macro expansion. However, because of its approval of involvement of succeeding token sequence, the range in which such re-replacement is inhibited still remains unclear. Thus, C Standards continue to sway, issuing a corrigendum and then revising it.

7.3 Separation of Macro Expansion from the Other Processing

Many C preprocessors seem to have a traditional program structure in which a replacement list and its succeeding text are read successively during macro rescanning. Each time they replace a macro invocation with its replacement list, they repeat rescanning for the next macro with its start point shifting gradually.

This traditional program structure illustrates the historical background of C preprocessors: they were derived from macro processors. For some preprocessors, including GNU C 2/cpp, a macro rescanning routine is de facto main routine of a preprocessor. The main routine rescans text with its start point shifting gradually until it reaches the end of an input file, during the course of which, a routine to process preprocessing directives is called. This is an old macro processor structure with a disadvantage that macro expansion and other processing are likely to got mixed. As shown in 6.2.3, how to expand a macro differs between when the macro is on a #if line and when on other lines. This is one of the typical examples of this mixture. (GNU C 2/cpp internally treats `defined` on a #if line as a special macro.)

MCPP provides a macro expansion routine in Standard and post-Standard modes that is quite different from traditional routines. The macro expansion routine is dedicated to macro expansion and performs no other tasks. Likewise, other routines ask the routine for all macro expansion and only receive the result. The macro expansion routine has a recursive structure, not of a repeating one, with a simple mechanism to prevent re-replacement of a macro with the same name. Expansion of a function-like macro strictly follows the function-like processing principle, and rescanning is basically completed within a macro invocation. This is all that the macro expansion routine does in post-Standard mode. In Standard mode, the macro expansion routine provides a mechanism to deal with the irregular specification in C Standards so that it can exceptionally handle succeeding token sequence only when necessary. This makes a program structure more clear but also makes it easy to detect an abnormal macro to issue a warning.

7.4 Portable C Preprocessor

Although one of the reasons for existence of the preprocessing phase in C is to provide greater portability, preprocessing itself has often impaired it, because in most compiler-systems the preprocessor has been an addition to the compiler and has had unnecessarily implementation-specific

behaviors. In contrast, C90 specified preprocessing as a phase mostly independent from the execution environment, hence guaranteed rather great portability.

What is more, thanks to C90, most parts of a preprocessor itself can be written portable, unlike other components of a C compiler-system. Thus, it might be even possible for every compiler-system to use the same high quality and portable preprocessor. A portable preprocessor for portable source has been ready to appear since C90. Development of MCPP began motivated by this situation. Though, many existing compilers have absorbed preprocessor into themselves, an independent preprocessor has a merit of decreasing compiler-dependent behaviors and increasing portability of preprocessing as an independent phase.

The above principles were embodied in the C90 preprocessing stipulations. At the same time, the above contradictions also existed, which were left to later Standard to solve. C99, however, solved none of these basic problems, while it added some new features. What is worse, there are a few areas where simple-and-clearness of the specifications were impaired by the appended features. C++98 has more problems than C99. (For these problems, refer to cpp-test.txt.)

After all, it can be said that, in the history of C preprocessing, C90 was the one and only attempt to clarify the basics of the language, though not satisfactory enough. Today, the specifications began to diffuse again, and clarification stepping into the basics is expected. I think that the direction should be to complete the principles which C90 did only halfway.

MCPP is a C preprocessor which is constructed on the principles of “token-based processing”, “function-like expansion of function-like macro”, “separation of macro expansion routine from other processing” and “portable preprocessing”. In its conformant mode, MCPP obeys the Standard's irregularities using some modifications on these principles. In addition, MCPP provides preprocessing in what I call “post-Standard” mode, in which these principles are obeyed thoroughly by eliminating deficiencies from Standards themselves and reorganizing them. If no problems were detected in this mode, the source can be said as having high portability as long as preprocessing is concerned.

8 Current Version and Update Plans

8.1 V.2.5

MCPP V.2.5 is an update to V.2.4.1 which was released in March 2004. The updated points are as follows:

1. Absorbed the 'post-Standard' mode into an execution option of the 'Standard' mode. Absorbed the 'old-preprocessor' setting into an execution option of the 'pre-Standard' mode.
2. Revised again and made perfect the recursive macro expansion.
3. Modified 'old-preprocessor' specifications to follow Reiser cpp.
4. Changed some execution options and #pragma directive names.
5. Added portings to GNU C V.3.3 and V.3.4.
6. Made a few small improvements.
7. Changed the point allotment of Validation Suite.

8.2 Update Plans for V.2.6

Updating of MCPP is getting behind from the previous plan. At present, following updates are planned for MCPP V.2.6.

1. MCPP diagnostic messages will be stored in a separate file so that anyone can add diagnostic messages in various languages at any time.
2. GNU C 3/cpp source programs and its testsuite will be given more consideration.

3. An option to automatically rewrite unportable source programs to portable ones will be implemented.

4. For better search, two versions, texinfo and html, will be created for documents.

9 Conclusion

I have developed a C preprocessor MCPP in parallel with an exhaustive validation suite for C preprocessing, aiming at the highest conformance and the highest quality. As a result, I have succeeded to show superiority of MCPP over other C preprocessors. Also, I have discussed the implementation method of C preprocessor and asserted that it is vital for an excellent preprocessor to construct program on the ground of clear principles.

It was in 1992 when I began to develop MCPP based on DECUS cpp. After ten years, MCPP was adopted to one of the "Exploratory Software Projects", which gave me a chance to send it out into the world. With the finishes that extended for nearly two years, I completed a C preprocessor that I think ranks number one in the world. Now, I will send out MCPP with its English documents into the world for international evaluation. Moreover, I was estimated as one of the highest class programmers by the achievement of "Exploratory Software Projects". I am also satisfied with myself, who have done a good job as a middle-aged amateur programmer.

The older versions of MCPP and its Validation Suite are available at `http://download.vector.co.jp/pack/`.

During "2002 Exploratory Software Projects", cvs repository, a ftp site, and web pages are created in m17n.org. [18] MCPP V.2.3 and later are stored here.

I have continued updating of MCPP after the project, and will keep on it. Many C programmers comments and feedback, as well as participation in MCPP development are welcome!

Related Materials and URL

[1] Information-Technology Promotion Agency (IPA), Japan,
"Exploratory Software Projects".
`http://www.ipa.go.jp/jinzai/esp/`

[2] ISO/IEC. *ISO/IEC 9899:1990(E) Programming Languages – C*. 1990.

[3] ISO/IEC.
ibid. Technical Corrigendum 1. 1994.

[4] ISO/IEC.
ibid. Amendment 1: C integrity. 1995.

[5] ISO/IEC.
ibid. Technical Corrigendum 2. 1996.

[6] ISO/IEC. *ISO/IEC 9899:1999(E) Programming Languages – C*. 1999.

[7] ISO/IEC.
ibid. Technical Corrigendum 1. 2001.

[8] ISO/IEC.
ibid. Technical Corrigendum 2. 2004.

[9] ISO/IEC. *ISO/IEC 14882:1998(E) Programming Languages – C++*. 1998.

[10] Martin Minow, DECUS cpp.
`http://sources.isc.org/devel/lang/cpp-1.0.txt`

[11] Borland International Inc.,
Borland C++ V.4.0. 1994.

[12] Borland Software Corp.,
Borland C++ Compiler 5.5
`http://www.borland.co.jp/cppbuilder/freecompiler/`

[13] Free Software Foundation, GCC.
`http://gcc.gnu.org/`

[14] Thomas Pornin., ucpp.
`http://pornin.nerim.net/ucpp/`

[15] Microsoft Corporation,
Visual C++ .net 2003

[16] Jacob Navia., LCC-Win32.
http://www.q-software-
solutions.com/lccwin32/

[17] Hartmut Kaiser, Wave V.1.0.0.
http://sourceforge.net/
projects/spirit/

[18] Kiyoshi Matsui, MCPP V.2.5.
http://www.m17n.org/mcpp

[19] Highwell, Inc. Limited Company.
http://www.highwell.net/