High Quality C Preprocessor Mcpp

Kiyoshi Matsui
kmatsui@t3.rim.or.jp

April 7th, 2007

Abstract

There has been a long history of confusion about the specifications of C preprocessing. Although, after C90, preprocessor specifications tend to converge to the Standard, so called Standard-conforming preprocessors still sometimes behave wrong. Moreover, the existing preprocessors have not a little implementation-specific behaviors, and as a result preprocessing sometimes impairs portability, although one of the purpose of preprocessing is to provide portability. Furthermore, debug of preprocessing, which is difficult in compile phase, should be supported by preprocessor, but the existing preprocessors have little of that functionality. Under these problems lies the fact that in most compiler systems preprocessor has been an addition to compiler. This situation has not much changed since pre-C90 until now.

mcpp has been developed attempting to solve these problems. **mcpp** is an open source software. It is a 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 on-target diagnostic messages and #pragma directives to output debugging information. **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 have converged to the Standard, so called Standard-conforming preprocessors still sometimes 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 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 Standards one by one without C preprocessing principles being made clear, thus prolonging the problems. The subordinate situation of preprocessor to compiler makes another background of 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 often impaired it.

In addition, preprocessing causes debugging difficulties. Since preprocessing is a “pre”process of compiling, preprocessing directives and macros disappear in compile phase. Although the preprocessors themselves should assist debug of preprocess phase, no preprocessor does this.

I have been developing a C preprocessor for a long time. My work had already been released as cpp V.2.0 and V.2.2 in August 1998 and in Novem-

ber 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. After the project, V.2.5 and V.2.6 were released. My cpp is called **mcpp** (Matsui CPP) to distinguish it from other cpps.

mcpp is probably number one C preprocessor now available in the world. I say this 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 and #pragma directives for debugging information that allows you to check almost all the preprocessing problems in source programs and to increase source portability.

Also it is a portable preprocessor easily implementable for any compiler system, and hence can assure portability of preprocess phase independent on the compilers proper. Its source is structured on clearly defined preprocessing principles, and its specifications are clear.

This document is organized as follows:

Section 2: Provides an overview of **mcpp**.

Section 3: Introduces briefly the basic specifications of Standard 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 and problems 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/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[9,10] behaviors, needless to say C90.

2. Provides a validation suite that allows you to test and evaluate 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, ISO-2022-JP 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. It can work properly in a system with a small amount of memory.

7. **mcpp**'s source is portable. It can be compiled with any C90, C99 or C++98 conforming compiler systems. **mcpp** is so designed that it can generate a preprocessor to be used replacing a compiler-system-resident one (if possible) on UNIX-like systems or Windows by modifying some settings in header files on compilation of **mcpp**. **mcpp**'s source also allows to generate a compiler-independent preprocessor which behaves on its own independent of any compiler systems. Moreover, you can also compile **mcpp** as a subroutine to be called from some other main program.

8. In addition to "Standard" mode, which conforms to C90, C99 and C++98 Standards, **mcpp** has various behavioral modes, including the mode of $K\&R^{1st}$ specifications, the Reiser model cpp mode and what I call "post-Standard" mode in which all the problems in C Standards are cleared.

9. On UNIX-like systems, a configure script can be used to automatically generate a **mcpp** executable. If GCC 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, documents 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. After the project, the updates have been done by the author.

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

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

    (c) mcpp-manual.html: 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.html: Porting Manual – Describes how to port **mcpp** to particular compiler systems.

    (e) cpp-test.html: 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 Standard 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

Table 1: Number of Test Items and Scores covered by Validation Suite V.1.5.3

		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

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 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

One of the problems involved in preprocessor development is how to verify preprocessor's behavior and its quality. Though most compiler systems calls themselves as "Standard conforming", their verification data are not shown in many cases. 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. 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.1.5.3 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 not exist in C90. "Standard conformance" covers all the specifications of Standards.

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

There are some rooms for subjective evaluation in the quality items and the allocation of points, and there are problems in measuring the diverse items with one yardstick. Nevertheless, I think that this scale gives results fairly close to the actual usage impressions.

Table 2 and figure 1 shows the summary of results of applying Validation Suite V.1.5.3 to several compiler systems. The table and the figure

Table 2: Validation Results of Each Preprocessor

Preprocessor	year/month	conformance					quality		overall
		K&R	C90	C99	C++ 98	total	diag- nostic	others	evalu- ation
DECUS cpp [1]	1985/01	150	240	0	0	390	15	78	483
mcpp 2.0 [2]	1998/08	166	430	58	10	664	68	125	857
Borland C 5.5 [3]	2000/08	164	366	20	6	556	18	72	646
GCC 2.95.3 [4]	2001/03	166	404	56	6	632	24	113	769
GCC 3.2 [5]	2002/08	166	419	86	20	691	32	117	840
ucpp 1.3 [6]	2003/01	166	384	88	9	647	25	88	760
Visual C 2003 [7]	2003/04	156	394	43	15	608	21	83	712
LCC-Win32 2003-08 [8]	2003/08	158	376	18	6	558	19	84	661
Wave 1.0.0 [9]	2004/01	140	338	53	18	549	21	79	649
mcpp 2.4 [10]	2004/02	166	432	98	22	718	74	134	926
GCC 3.4.3 [11]	2004/11	166	415	87	20	688	38	120	846
Visual C 2005 [12]	2005/09	160	399	65	17	641	20	77	738
LCC-Win32 2006-03 [13]	2006/03	156	374	22	6	558	22	85	665
GCC 4.1.1 [14]	2006/05	166	417	87	20	690	38	120	848
mcpp 2.6.3 [15]	2007/04	166	432	98	22	718	74	136	928
highest score		166	432	98	26	722	74	164	960

shows compiler systems in a chronological order.

[1] DECUS cpp: Original version written by Martin Minow,[11] which was slightly revised by the author and compiled with Linux/GCC.

[2] **mcpp** 2.0: Open source software developed by the author. Was rewritten based on DECUS cpp. Was ported to various compiler systems, such as FreeBSD/GCC 2.7, DJGPP V.1.12, WIN32/Borland C 4.0, etc. Although mcpp allows generation of a preprocessor of various specs, the standard mode of the executable compiled by GCC on Linux was used for this test.

[3] Borland C 5.5: Japanese version. Borland.[12]

[4] GCC 2.95.3: Bundled in VineLinux 3.2 or CygWIN 1.3.10.

[5] GCC 3.2: Compiled by the author on Linux.[13]

[6] ucpp 1.3: Portable open source software written by Thomas Pornin. A compiler-independent preprocessor.[14]

[7] Visual C++ 2003: Visual C++ .net 2003. Microsoft.

[8] LCC-Win32 2003-08: Developed by Jacob Navia. Dennis Ritchie's C90 preprocessor is used as its preprocessing part.[16]

[9] Wave 1.0.0: Open source software written by Hartmut Kaiser. Implemented using "Boost C++ preprocessor library" written by Paul Mensonides et. al. Tested about an executable for Windows.[17]

[10] **mcpp** 2.4: From V.2.0 onward, has been ported to Linux, FreeBSD / GCC 2.95-3.2, CygWIN 1.3.10, LCC-Win32 2003-08, Borland C 5.5 and Visual C++ 2003.

[11] GCC 3.4.3: Compiled by the author on Linux.

[12] Visual C 2005: Visual C++ 2005 Express Edition. Microsoft. [15]

[13] LCC-Win32 2006-03: LCC-Win32 2006/03 version.

As shown in the table, **mcpp** is 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, comprehensive documentation, useful execution options, #pragmas for debugging, handling of various multi-byte character encodings, and portability.

According to the table, the second best preprocessor to **mcpp** is GCC (GNU C) / cpp (cc1). GCC presents almost no problems as long as it processes C90 conforming sources. However, it 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

[14] GCC 4.1.1: Compiled by the author on Linux.

[15] **mcpp** 2.6.3: From V.2.4 onward, has been ported to GCC 3.3-4.1, MinGW/MSYS and Visual C++ 2005.[18]

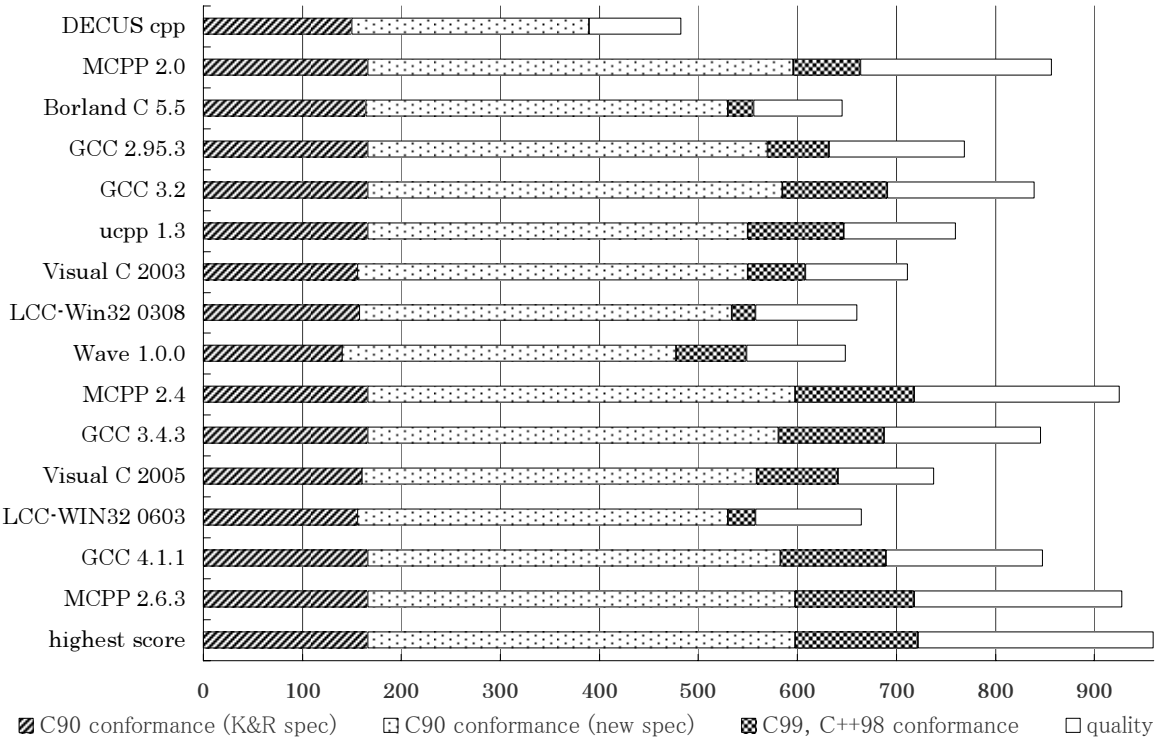

Figure 1: Validation Results of Each Preprocessor

a lot of unchecked problems.

2. It provides little functionality to output debugging information.

3. Documentation is insufficient; there are some unclear or undocumented specifications.

4. GCC uses its own specifications that are inconsistent with the Standards. Extended specifications should be implemented with #pragma.

Compared with GCC V.2/cpp, GCC V.3/cc1 has been much improved in these aspects, but is still insufficient.

mcpp is inferior to GCC/cc1 only in processing speed.

Other preprocessor has much more problems than GCC. 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.

```
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 > SHRT_MAX
```

Figure 2: Sample of Preprocessor Bugs

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.

5.1 Comment Generating Macro

Example-1 is a macro definition that is actually found in a Visual C Platform SDK 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/cl.exe processes this line as expected.

However, `//` is not a preprocessing-token. In addition, macro should be defined and expanded after source is 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 in this example.

2. Preprocessing procedure (translation phases) is implemented arbitrarily and lacks in logical consistency.

5.1.1 Mcpp's Diagnostic

mcpp issues some diagnostics while preprocessing `<windows.h>`, and on the macro of example-2 issues a diagnostic as shown in figure 3. (The lines are broken appropriately for printing.)

First, the source file name and the line number which contains the macro call in question, diagnostic message body, next, definition of the macro and its location, then, each includer's line which #include the source file, tracing back the nested includes one after the other. — It is clear what and where the problem is.

5.2 Expressions That Should Be Skipped Causes an Error

The expressions in example-3 and 4 are correct ones. These expressions are so carefully written that a division is carried out only when a denominator is not zero. However, some preprocessors perform a division when `MACRO_0` is zero and cause an error. Example-3 used to cause an error in many preprocessors, but nowadays it is processed properly. Example-4 still causes an error in Visual C 2003, 2005, which shows that its preprocessor does not implement basic C specifications of expression evaluation correctly.

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 issues the same warning to a real division by zero shown in example-5. In other words, Borland C cannot tell correct source code from wrong code. Turbo C issued the same error message to both correct expressions and incorrect ones that cause a zero division error. Borland C simply degrades the error message to a warning. This could not be called non-conforming, but indicates a patchy work and poor quality of diagnostic messages.

```
c:/program files/microsoft platform sdk/include/oaidl.h:442:
error: Not a valid preprocessing token "//"
    in macro "_VARIANT_BOOL" defined as: #define _VARIANT_BOOL /##/
    /* c:/program files/microsoft platform sdk/include/wtypes.h:1073    */
    from c:/program files/microsoft platform sdk/include/oaidl.h: 442:
        _VARIANT_BOOL bool;
    from c:/program files/microsoft platform sdk/include/msxml.h: 274:
        #include "oaidl.h"
                                ...
```

Figure 3: A sample of diagnostic issued by mcpp

5.3 Overflow is Overlooked

The constant expression in example-6 causes an overflow in C90. Most preprocessors do not issue a diagnostic to this overflow. In other words, any message is not issued even if the value wraps round, and the sign and the comparison result are reversed. GCC and Borland C are 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 spoil portability, readability and maintainability, and, what is worse, they will be likely bug-prone. 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 2.2.2 (May 1997) kernel and libc sources. Libc sources had almost no problems, but some kernel sources had 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 (GNU LIBC) 2.1.3 (February 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 GCC/cpp's own or undocumented specifications. I think GCC/cpp's default passing of such undesirable sources without issuing a message not only preserves them but also produces new ones. It is more problematic that such coding is not necessarily found in old sources only; it is sometimes found in newly written 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 GCC/cpp no longer allowed them. This indicates how much a preprocessor affects sources coding.

Recently I checked Glibc 2.4 (March 2006) sources with **mcpp**, and found that most of the portability problems I had noticed in Glibc 2.1.3 were not resolved after the six years of updates. Although the very old style sources such as "multi-line string literal" have disappeared, those sources which depend on GCC's local specifications and undocumented behaviors did not decreased. On the contrary, they increased largely.

6.2 Sample Glibc Source Code Fragment

To see some preprocessing problems, let me take examples of Glibc 2.4 source code fragments.

6.2.1 ∗.S Files That Require Preprocessing

∗.S files are assembler sources with inserted preprocessing directives, such as `#if`, `#include`, and C comments. Some of them have also macros embedded.

Since assembler source is not consisted of C token sequence, it accompanies some risks to preprocess it by C preprocessor. To process an assembler source, a preprocessor must pass such characters as % or $ (which are not used in C except in string literal or in character constant) as they are, and retain existence or non-existence of spaces as they are.

Example-1 of figure 4 is part of a ∗.S file. `#ifdef SHARED` intends to be a directive of C. On the other hand, the latter part of each line starting with # are supposed to be comments. `# column.` is, however, syntactically indistinguishable from invalid directive, since the # is the first non-white-space-character of the line. `# + DW_EH_PE_sdata4` causes even syntax error in C.

The same source has a line as example-2. This is a macro call and is expected to be expanded as example-3, though I omit macro definitions here. The part `pthread_cond_wait@@GLIBC_2.3.2` is a sequence to be generated concatenating some parts by ## operator. There is, however, no C token nor pp-token containing @ (except string literal or character constant).

This source file is full of grammatical errors and undefined behaviors when processed by Standard C preprocessing, and even the lines free from errors are not at all assured to be preprocessed as expected.

To process assembler codes with C, it is recommended that the asm() function should be used whenever possible, to embed the assembler code 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. The assembler sources with macros embedded are usually unable to be dealt with asm(). This type of source is not a C source and essentially should be processed with an assembler macro processor.

6.2.2 Variadic Macro of GCC Spec

GCC has variadic macro of its own specification like example-1 of figure 5 since before C99. In addition, GCC 3 implemented another peculiar syntax like example-2. GCC 2.95.3 and later implements C99 spec one, too, which is exemplified as example-3.

Both of the GCC specs are very queer ones, and do not correspond to C99 spec one-to-one. While C99 spec requires at least one real argument for variable arguments, GCC specs permit absence of argument, and the ## is not token concatenation operator here, but it has a special meaning to remove the immediately preceding comma in case real argument for variable argument is absent. Old GCC spec even uses "named variable argument" as `args...` which is not a C pp-token.

In Glibc sources, all the variadic macros used are the old GCC spec ones, and no C99 spec nor even GCC 3 spec one is found. Moreover, there are extremely many macro calls which lack real argument for variable argument and hence cause re-

```
example-1
        .byte   8               # Return address register
                                # column.
    #ifdef SHARED
        .uleb128 7              # Augmentation value length.
        .byte   0x9b            # Personality: DW_EH_PE_pcrel
                                # + DW_EH_PE_sdata4

example-2
    versioned_symbol (libpthread, __pthread_cond_wait, pthread_cond_wait,
                      GLIBC_2_3_2)

example-3
    .symver __pthread_cond_wait, pthread_cond_wait@@GLIBC_2.3.2
```

Figure 4: Glibc source sample 1: ∗.S file

```
example-1
    #define libc_hidden_proto(name, attrs...)   hidden_proto (name, ##attrs)

example-2
    #define libc_hidden_proto(name, ...)    hidden_proto (name, ## __VA_ARGS__)

example-3
    #define libc_hidden_proto(name, ...)    hidden_proto (name, __VA_ARGS__)
```

Figure 5: Glibc source sample 2: vadiadic macro of GCC-spec

moval of preceding comma. In short, the old spec has been used all the time until now.

As a variadic macro specification, C99 one is the most reasonable and portable. When a macro call needs no real argument for variable argument, it is recommended to use some harmless argument such as `0` or `NULL`.

6.2.3 Macro Expanded to 'defined'

There is a macro definition shown in example-1 of figure 6 and the macro is used as shown in example-2. 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 invocation is first replaced as example-3. Supposing that `_STATBUF_ST_BLKSIZE` is defined as example-4, it finally expands as shown in example-5. When `_STATBUF_ST_BLKSIZE` is not defined, it results as example-6. Both are #if expression syntax errors, of course.

The same thing would happen to GCC, if `_STATBUF_ST_BLKSIZE` were not on a #if line. However, on the #if line, GCC stops macro expansion at example-3 and evaluates it as a #if expression. This behavior 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 macro should be defined as shown in example-7. Or the #if line of example-2 should be written as example-8 without using the macro `_STATBUF_ST_BLKSIZE`.

By the way, **mcpp** issues an error on example-2, and in addition, if you sandwich the line with #pragma directives as example-9, it outputs macro expansion process, and you can grasp what is wrong.

6.2.4 Object-Like Macros Expanded as 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,

```
example-1
    #define _G_HAVE_ST_BLKSIZE defined (_STATBUF_ST_BLKSIZE)

example-2
    #if _G_HAVE_ST_BLKSIZE

example-3
    defined (_STATBUF_ST_BLKSIZE)

example-4
    #define   _STATBUF_ST_BLKSIZE

example-5
    defined ()

example-6
    defined (0)

example-7
    #if defined (_STATBUF_ST_BLKSIZE)
    #define _G_HAVE_ST_BLKSIZE 1
    #endif

example-8
    #if defined (_STATBUF_ST_BLKSIZE)

example-9
    #pragma MCPP debug expand token
    #if _G_HAVE_ST_BLKSIZE
    #pragma MCPP end_debug
```

Figure 6: Glibc source sample 3: a macro expanded to 'defined'

```
example-1
    #define _IO_close close_not_cancel

example-2
    #define close_not_cancel(fd) __close_nocancel (fd)

example-3
    #define _IO_close(fd)   close_not_cancel(fd)
```

Figure 7: Glibc source sample 4: object-like macro expanded to name of a function-like macro

which was approved by C90. In that sense, it can be described as providing greater portability. For example, an object-like macro is defined as example-1 of figure 7, while `close_not_cancel` is defined as shown in example-2.

What seems to be an object-like macro that is actually expanded as a function-like macro is inferior in readability. There is no reason to write in this way at least in this example. 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-3.

6.2.5 Sources Depending on Trivials of GCC's Behavior

Besides, among the scripts or tools used in making Glibc, I found some codes which unnecessarily depend on the trivial details of GCC behavior. Some of them even expect number of spaces on the line top of preprocessed output exactly as GCC emits, and some other expects exact number of spaces between pp-tokens preprocessed as GCC does.

In addition to the above, there are some more undesirable codings, 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 GCC by default had issued a warning to such programs, they would have been written in a quite different way from the beginning.

In the world of open source software, most of the preprocessing portability problems were trails of pre-C90 traditional UNIX codings, until mid-nineties. Nowadays, such extremely old style codings have decreased largely, instead, most of the problems are those which depend on GCC's local specifications.

In Glibc, such GCC-dependent sources have not decreased since year 2000 till 2006, they have rather increased. In a large scale software, many source files are interrelated each other, and change of their interfaces become difficult as time passes, therefore even newly written sources follow the existing interfaces. On the other hand, change of GCC behavior breaks many sources, and the possible influence becomes greater with time, therefore GCC becomes difficult to change its behavior. I believe that both of GCC and Glibc need to tidy up their old local specifications and old interfaces drastically in the near future.

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 quite 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.

Some principles may be reasonably extracted from C90 preprocessing specifications:

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 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 2.7 of cpp-test.html.)

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 character-based processing got mixed occasionally.

In Borland C 5.5 or Visual C 2003, 2005, 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 handling. Also 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 Macro

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 2.7.6 of cpp-test.html.

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 macro" 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 the text succeeding the macro call 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. In some pre-

processors, including GCC 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 examples of this mixture. (GCC 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 spoiled 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, it is not a good program structure where a preprocessor and a compiler interdepend complexly. 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 lost by the appended features. C++98 has more problems than C99. (For these problems, refer to cpp-test.html.)

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 required. 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 conforming 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 irregularities 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.2.6

mcpp V.2.6 was released in July 2006, and was an update to V.2.5 which had been released in March 2005. The updated points are as follows:

1. Made compiler-independent-build really independent of any compiler systems.
2. Integrated Standard modes and pre-Standard modes into one executable, made all the behavioral modes to be specified by the invocation options.
3. Added portings to GCC 4.0 and Visual C++ 2005.

Afterwards, V.2.6.1, V.2.6.2 and V.2.6.3 were released by April 2007. In these versions, porting to MinGW/MSYS was done, compatibility to GCC of GCC-specific-build was increased, most of the text file documents were converted to html, subroutine-build to use **mcpp** as a subroutine from other main program was created, and several bugs were fixed..

8.2 Update Plans for V.2.7

Updating of **mcpp** is far behind from the previous plan. At present, following updates are planned for **mcpp** V.2.7.

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. An option to automatically rewrite unportable source programs to portable ones will be implemented.
3. A series of testcases for testing **mcpp**'s own specifications will be created in Validation Suite.

9 Conclusion

I have been developing a C preprocessor **mcpp** aiming at the highest conformance, the highest quality and usefulness for source checking. Since validation system is indispensable for developing a language processing system, and no adequate one existed for preprocessor, I developed an exhaustive validation suite for preprocessor in parallel to **mcpp**. As a result, I have succeeded to show superiority of **mcpp** over other preprocessors. Besides, I showed advantage of a preprocessor independent from compiler systems, in decreasing compiler-dependent behaviors and assuring portability of preprocessing. Also, I 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. **mcpp** with its English documents became ready 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.

I am still continuing to update **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 URLs

[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] ISO/IEC. *ISO/IEC 14882:2003(E) Programming Languages – C++.* 2003.

[11] Martin Minow, DECUS cpp.
`http://www.isc.org/index.pl?/sources/devel/`

[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,
`http://msdn.microsoft.com/vstudio/express/visualc/default.aspx`

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

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

[18] Kiyoshi Matsui, mcpp.
`http://mcpp.sourceforge.net/`

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