Powerful C Preprocessor for Source Checking

Kiyoshi Matsui
kmatsui@t3.rim.or.jp

March 20th, 2004

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, however, 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 achieved 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

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 V2.2 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 2002" by Information-Technology Promotion Agency (IPA), Japan. [1] In 2003, it was continuously adopted to the same Project. In this document, 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 and its Validation Suite.

Section 3: Shows data to compare Standard conformance level and qualities with other preprocessors.

Section 4: Shows examples of bugs in compiler-system specific preprocessors.

Section 5: Describes MCPP's source checking and why such checking is required.

Section 6: Describes C preprocessing principles and how to implement them.

Section 7: Describes what is achieved through update to V.2.3 and V.2.4.

Section 8: Describes 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 [6, 7] and C++98 behaviors [8], needless to say C90 [2–5].

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 facility can handle a variety of Japanese, Chinese, Taiwanese and Korean encodings.

6. Processing speed is not so slow; it can be used not only for debugging purpose but also for daily processing. 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. Is portable. MCPP is so designed that it can generate a preprocessor to be used to replace a compiler system specific one on UNIX-like systems or DOS/Windows by modifying some settings in header files on compilation of MCPP. The portability MCPP 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 modes. These modes range from the one based on the $K\&R^{1st}$ specifications or the Reiser model to what I call "post-Standard" mode in which all the problems in C Standards are solved.

9. On UNIX-like systems, a configure script can be used to automatically generate a MCPP executable.

10. MCPP is an open source software.

11. Detailed documentation is provided:

    (a) README – Describes how to configure and make.

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

    (c) 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) 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 compiler systems.

3 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,

	Number of Test Items	Lowest Score	Highest Score
C90 Standard Conformance	173	-162	463
C99 Standard Conformance	20	0	98
C++98 Standard Conformance	9	0	26
Quality: Diagnostic Messages	47	0	74
Quality: Others	16	-40	113
Total	265	-202	774

Table 1: Number of Test Items and Scores covered by Validation Suite V.1.4

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.1.4 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. For preprocessors that properly implement specifications common between $K\&R^{1st}$ and C90, score 0 is given. For ones that even fail to do that, a negative score is given. For ones that properly implements new specifications of C90 onward, a positive score is given. "Standard Conformance Level" includes evaluation of diagnostic messages and documentation, as well as of behaviors. "Standard Conformance Level" for C99 and C++98 deals with new specifications that do no exist in C90. "Quality: Diagnostic Messages" deals with diagnostic messages that are not required by C Standards.

Table 2 shows the summary of results of applying Validation Suite V.1.4 to several compiler systems. The table shows compiler systems in a chronological order.

*1 Original version of DECUS cpp developed by Martin Minow (1985/06), which was slightly revised by the author and compiled by Linux/GNU C 3.2. [9]

*2 JRCPP, shareware for UNIX, OS/2 and MS-DOS, developed by J. Roskind and migrated to MS-DOS - OS/2 for trial usage (1990/03). This is a stand-alone preprocessor not corresponding for any particular compiler systems. [10]

*3 Japanese version for 1993 (1994/12). [11]

*4 GNU C 2.7.1/cpp (1995/12) ported to GO32, DOS extender, by DJ Delorie. Supported shift-JIS when ported to the Japanese version. [12]

*5 Revised version beta-13 of LSI C-86/cpp by Akira Kida (1996/02). [13]

*6 Cpp V.2.0 (1998/08), free software developed by the author. Was rewritten based on DECUS cpp. Is ported to various combinations of OSs and compiler systems, such as FreeBSD/GNU C 2.7, DJGPP V.1.12, WIN32/Borland C 4.0, MS-DOS/Turbo C2.0, LSI C-86 3.3, and OS-9/09/Microware C. Although Cpp V.2.0 allows generation of a preprocessor in various modes, the 32-bit system standard mode was used for this test.

*7 Japanese version (2000/08). [14]

*8 GNU C 2.95.3 (2001/03) used under VineLinux 2.6, FreeBSD 4.4 or CygWIN 1.13. [15]

*9 GNU C 3.2R (2002/08) compiled by the author under VineLinux 2.6 and FreeBSD 4.7. [15]

*10 Portable free software (2003/01) developed by Thomas Pornin. A stand-alone preprocessor. [16]

*11 Microsoft (2003/04). [17]

*12 Shareware, with source code available, developed by Jacob Navia et al. (2003/08). Dennis Ritchie's C90-conforming preprocessor is used as its preprocessing part. [18]

OS	Compiler Systems	Preprocessor (Version)	(1)	(2)	Rem
Linux		DECUS cpp	230	287	1
MS-DOS		JRCPPCHK (V.1.00B)	400	451	2
WIN32	Borland C++ V.4.02J	cpp32	397	444	3
DJGPP V.1.12 M4	GNU C 2.7.1	cpp	445	545	4
MS-DOS	LSI C-86 V.3.30c	cpp (beta13)	341	397	5
FreeBSD, WIN32, etc.	GNU C, Borland C, etc.	MCPP (V.2.0)	495	651	6
WIN32	Borland C++ V.5.5	cpp32	397	451	7
Linux, FreeBSD	GNU C 2.95.3	cpp0	470	570	8
Linux, FreeBSD	GNU C 3.2R	cpp0	530	646	9
Linux, etc		ucpp (V.1.3)	483	562	10
WIN32	Visual C++ .net 2003	cl	452	517	11
WIN32	LCC-Win32 V.3.2	lcc	396	476	12
Linux, FreeBSD, etc.	GNU C, LCC-Win32, etc.	MCPP (V.2.4.1)	583	750	13
(1) Standard Conformance (2) Overall Evaluation					

Table 2: Validation Results of Each Preprocessor

*13 MCPP V.2.4.1 (2004/03). From V.2.0 onward, MCPP has been ported to Linux/GNU C (2.95.3, 3.2), FreeBSD/GNU C (2.95.4, 3.2), CygWin 1.13, LCC-Win32 3.2, Borland C 5.5, Visual C++ .net 2003 and Plan 9 ed.4/pcc. [19]

As shown in the table, MCPP is by far the best in every aspect – high conformance to C Standards, abundant and accurate diagnostic messages, detailed documentation, and its portability. Other preprocessors are even behind cpp V.2.0, which was developed five and half years ago. V.2.4 has been further updated and enhanced since then. You may think this result is natural because I tested my own preprocessor with my own validation suite, but seeing that Validation Suite offers such a variety of test items, you will agree that the evaluation is highly objective.

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 legal sources. However, GNU C/cpp still has the following problems, except for 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 poor; there are many unclear or undocumented specifications. 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.

4 Examples of Preprocessor Bugs and Erroneous Specifications

Each preprocessor contains various bugs and erroneous specifications, only some of which this section shows:

```
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
```

4.1 Comment Generating Macro

Example-1 is a macro definition that is actually found in the Visual C++ .net 2003 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 preprocessed 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.

4.2 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 Visual C++ 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 issues the same error message to both correct expressions and incorrect ones that may cause a zero division error. Borland C simply degrades an 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.

4.3 Overflow is Overlooked

The #if expression in example-6 causes an overflow. 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.

5 Why Is Source Code Check by Preprocessors Required?

A preprocessor is nothing but a small part of the entire C processing. What is the use of developing a preprocessor alone, even if it is ranked as number one in the world?

C preprocessing, as its name shows, is processing that is performed before the compiler-proper compiles source code. It tends to have been treated as an addition to the compiler-proper. However, the purpose of the preprocessing is to increase readability and maintainability of source code and to provide portability among various systems. In other words, it is to make it easy for programmers to handle source code. C preprocessing seriously affects coding and maintenance tasks. Therefore, C preprocessing, when used in an improper manner, would rather impair readability and portability. In that sense, source code checking by preprocessors is important.

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 were too reticent; since they passed questionable sources without issuing messages, problems remain unnoticed.

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. This results from inadequate validation of compiler systems themselves. Behind this, there lies a background that many C preprocessors have been developed based on C90 prior versions, and version upgrade has been made repeatedly, without C preprocessing principles being made clear. In order to develop a standard-conformant C preprocessor, it is necessary to define principles clearly and totally rewrite source programs based on these principles. In addition, it is essential to provide software to validate their conformity.

Moreover, as stipulated by the standard, all the compiler system-specific specifications must be documented. Accurate and detailed documentation is an integral part of compiler system development. However, many compiler systems fail to do so. Ambiguity about specifications forms a background to unportable source programs.

Furthermore, the standard's own contradictions and ambiguities, stemming from the historical background, makes this problem more complex. Although C preprocessors seem to be a technically matured field, many problems still remain unsolved.

Nevertheless, I have an optimistic view about this. Preprocessing is, for the most part, independent of the run-time environment, so, unlike other parts of a compiler system, it is relatively easy to make itself portable. Thus, it might be even possible for every compiler system to use the same high-quality and portable preprocessor.

5.1 How Much Do Preprocessors Affect Sources?

By replacing a compiler system-specific 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 cpp 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 occurred 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.

5.2 Sample Glibc Source Code Fragment

To see some preprocessing problems, let me take an example of a glibc 2.1.3 source code fragment used in VineLinux 2.1 (i386).

5.2.1 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.

5.2.2 $*$.S Files That Require Preprocessing

Some assembler sources have preprocessing directives, such as `#if`, and C comments embedded. Since assembler and C have its own syntax, the result of preprocessing assembler code with a C preprocessor is unknown. In addition, some source have some lines for assembler, such as #APP and #NO_APP, which are syntactically indistinguishable from invalid preprocessing directives.

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.

5.2.3 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.

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)
```

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.

5.2.4 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.

5.2.5 Undocumented Specifications on Environment Variable

This is a problem not of C sources but of 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.

6 Principle on MCPP Development

The goals I set when I developed MCPP are shown in Section 2, “MCPP Overview”. It took a huge amount of time to achieve them because they are rather idealistic, but I think I could have almost achieved them.

Needless to say that MCPP processes right sources properly, I made so much of its ability to issue an accurate diagnostic message to illegal sources or suspicious ones that I prepared a variety of messages.

I also made much of documentation; to avoid undocumented specifications, I tried to document all the specifications other than those in common with standard documents.

I have implemented MCPP in many combinations of operating systems and compiler systems, which, in turn, have improved portability of the MCPP source itself and have lead to a thorough check of MCPP behaviors.

Furthermore, concurrent development of the comprehensive Validation Suite contributed to creation of a bug-free preprocessor. The preprocessing part of LCC-Win32, for example, uses source code written by Ritchie, but is contains a lot of bugs, specifically in evaluation of #if expressions. However excellent Ritchie's sources may be, they cannot be free of bugs without a validation suite. GNU C/cpp has become almost bug free since it has been debugged by many peoples over times. With an adequate validate suite, it would have been bug free much earlier. For the preprocessors which are not so popular, it is safe to say that debugging without a validation suite is impossible.

6.1 Principle on Token-Based Processing

Let's take a look at a principle of MCPP internal implementation. What I made most of is "token-based" processing.

The root cause of many problems detected by MCPP lies in confusion about "token-based" processing, a C preprocessing principle. Since the principle had been ambiguous before C90, an idea of character-based text processing came in. After C90, many preprocessors overlooked or even allowed themselves to perform character-based text processing, thus prolonging the confusion. What is worse, C Standards themselves contain several half-hearted stipulations or contradictions, which were not revised even in C99, making this problem more complicated. 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.

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 and 5.5/cpp32, 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, including GNU C 2/cpp, do not issue a warning to an illegal token generated by macro expansion because they simply neglect checking a token generated by preprocessing. For a preprocessor to be token-based, it must check every token one by one, including generated ones.

What is more, I provide preprocessing in what I call "post-Standard" mode, in which strict token-based processing can be achieved by eliminating deficiencies from C Standards themselves and reorganizing them. If no problems were detected by MCPP in this mode, the source can be described as having high portability as long as preprocessing is concerned.

6.2 Function-Like Expansion of Function-Like Macros

In MCPP development, a macro expansion routine is one of the key issues I made most of. I tried to make MCPP program structure clear, without adhering to that of existing programs.

Expansion of a macro without an argument is rather straightforward. On the other hand, for expansion of argument macros, 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 for Validation Suite.

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 5.2.4 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 argument macros by naming them "function-like macros" and establishing a specification similar to that of a function call. 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 continues to sway, issuing a corrigendum and then revising it.

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 it's 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 5.2.3, 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 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 Major Achievements of "Exploratory Software Project"

After releasing V.2.2, I could not spare my time to develop MCPP, so its development was stagnated. However, with adoption of MCPP to the "Exploratory Software Project for 2002" by Yutaka Niibe project manager as an opportunity, I spent less time in working and more time in developing MCPP.

7.1 V.2.3

The following subsections cover major achievements of MCPP V.2.3 and Validation Suite V.1.3 released in Feb. 2003.

7.1.1 Porting to GNU C 3.2

I replaced GNU C 3.2's own preprocessor with MCPP and recompiled GNU C 3.2 itself using GNU C 3.2. I applied the testsuite to both of the generated GNU C 3.2 to verify the results. I found that MCPP has almost enough compatibility with GNU C 3/cpp in practical use.

Thanks to MCPP, I was able to check GNU C 3.2 sources for preprocessing problems. I found that GNU C 3.2 source contains far less problems than other sources, such as glibc 2.1. This is because GNU C 3.2's preprocessor has been entirely rewritten from GNU C 2's. In the future, the revised preprocessor will continue to affect other software sources than GNU C.

Compared with GNU C 2, the GNU C 3 preprocessor was dramatically improved in many aspects; its preprocessor source code was entirely rewritten and better documented, and the testcases of its testsuite were dramatically increased in number. GNU C 3 now learns to issue a warning to a series of coding problems or a traditional way of writing, such as multi-line string literal shown in 5.2, as obsolete or deprecated. Now with increased documentation, GNU C 3 seems to become equal to MCPP, but it still lacks of warnings and cannot be described as very well documented.

7.1.2 Porting Validation Suite to GNU C/Testsuite

At V.1.3, I have created the edition of Validation Suite which is written so that it can be used as a testsuite for GNU C/cpp. I have confirmed that this edition can be applied to three preprocessors, GNU C 2.95 / cpp, GNU C 3.2/cpp and MCPP under Linux and FreeBSD. I think the edition can work with most GNU C 2.9x and 3.x versions.

The testcases of GNU C/cpp's testsuite were very unbalanced, so it is meaningful to provide more comprehensive ones like those of Validation Suite.

Besides, while the existing testcases of GNU C 3/testsuite expect only GNU C 3, and there are many testcases which cannot be applied even to GNU C 2/cpp, the testsuite edition of my Validation Suite has been written to test 3 preprocessors. The testcases of Validation Suite absorb the difference of spacing in preprocessing output and the difference of diagnostics by utilizing regular expression facilities of testsuite tools, such as DejaGnu and Tcl. Since these tools sometimes process regular expressions in a peculiar manner, and sometimes they even fail to do so, I had to make some contrivance to use them. With some contrivance, it was found that several compiler systems could be automatically tested.

However, in the testsuite, it is impossible to change runtime options for each compiler system. Actually, the C standard has several versions, such as C90, C95, C99 and etc., so it is necessary to use the `-std=` option to indicate which standard version a program conforms to. Unfortunately, GNU C older versions do not have this option, Therefore, my testsuite can be applied only to GNU C 2.9x or higher and MCPP V.2.3 or higher.

7.1.3 Creating English Documents

I asked “HighWell, Inc.” Limited Company, Tokyo, for translation of all the documents on MCPP V.2.3 and its Validation Suite into English. [20] Three bilingual translators translated these documents. I have reviewed all the translated documents and, for technical details, I revised their work.

7.2 V.2.4 prerelease

MCPP was continuously adopted to the “Exploratory Software Project for 2003” by Hiroshi Ichiji project manager.

In Nov. 2003, MCPP V.2.4-prerelease that contains the following updates was released:

7.2.1 Porting to Visual C++ .Net 2003

This time I took up Visual C++ .net 2003, the most popular commercially available C compiler system. I not only applied the Validation Suite to Visual C++, but also ported MCPP to it.

It was found that even the latest version of Visual C++ preprocessor has some fundamental defects, such as preprocessing procedure being implemented arbitrarily, and character-based processing still being mixed.

When I tried to port MCPP to Visual C++, it was impossible to directly replace the Visual C++ specific preprocessor with MCPP because it is integrated into the Visual C++ compiler. To cope with this situation, I had to write a makefile in such a way that a source program is first preprocessed by MCPP and then cl.exe is executed. In the Integrated Development Environment (IDE), MCPP cannot be used in a normal "project". To use MCPP in IDE, a makefile must be created first. Then, if you create a "makefile project" in IDE, it can recognize the makefile. This allows users to utilize almost every IDE functions, including source level debugging.

7.2.2 Creating Configure Script

On UNIX-like systems, I created a configure script to automate MCPP compilation.

One of the MCPP features is that MCPP can replace the preprocessor of a target compiler system. To achieve that, a lot of research on the implementation of the target compiler system is required, unlike general application programs. In fact, writing a configure.ac file itself requires some knowledge of the target compiler system.

When the target compiler system is GNU C , the configure script can automates the entire process from compiling MCPP apropriately to replacing the GNU C specific preprocessor with MCPP. If the testsuite of GNU C were installed, 'make check' could even execute the testsuite edition of Validation Suite. I can do this because I know GNU C.

Unfortunately, I have no experience to do the same for other compiler systems on UNIXes. So, I decided to write a configure script in such a way that it provides several options and users can specify necessary information using these options. In addition, to replace compiler system specific preprocessors with MCPP, additional work of writing source code to MCPP is required.

A configure script cannot be applied to compiler systems on DOS/Windows, except for CygWIN. So, DIFfile and makefile are provided for each of these systems as before.

7.2.3 Other

MCPP options and #pragma directives have been added, and diagnostic messages regarding macro re-definition and expansion have been revised. MCPP source code has been tidied out by deleting old code.

Along with this revision, Japanese documents have been updated.

Up till now, the MCPP source files and documents did not show any license notice explicitly. From V.2.4 prerelease onward, these files include a BSD-style license.

7.3 V.2.4

In Feb. 2004, MCPP V.2.4 that contains the following updates was released:

7.3.1 Handling of Various Multi-Byte Character Encoding

Since V.2.0, MCPP has supported various multi-byte (two-byte) character encodings, such as Japanese EUC-JP and shift-JIS, Chinese GB 2312, Taiwanese Big5, and Korean KS C 5601 (KSX 1001). MCPP V.2.4 now provides a framework to handle more complex encodings. To begin with, MCPP V.2.4 has enabled ISO-2022-JP1 and UTF-8 handling.

32-bit or more system ported MCPP has implemented all these encodings at the same time. Now, environment variables and run-time options allow users to change the default encoding. MCPP V.2.4 has implemented the `#pragma setlocale` directive provided in Visual C++, which allows users to change encodings in one source file. Furthermore, if the compiler-proper cannot handle an encoding, MCPP can now compensate its inability.

One system used to have only one fixed multi-byte character encoding. At present, the more necessary it is to have multi-language enabled programs, a more variety of encodings compiler systems must be able to handle. It is important for software programmers to be able to handle various encodings easily through environment variables, run-time options and #pragma.

7.3.2 Ported to Plan 9

MCPP has been ported to Plan 9 edition 4/pcc. This system uses the compiler written by Ken Thompson and the preprocessor written by Dennis Ritchie. Ritchie's cpp can be replaced with MCPP. Of cource, MCPP is much superior in quality. Plan 9 uses UTF-8 as multi-byte character encoding. It was confirmed that MCPP could handle UTF-8 properly.

7.3.3 Updated Documents

Along with this version, I have updated both English and Japanese versions of MCPP documents. Just like a year ago, I asked Highwell to translate the updated documents and completed them by correcting the work.

7.3.4 Release to World

I packaged MCPP along with its English documents to send it to gnu.org and freebsd.org.

8 V.2.4.1

The "Exploratory Software Project" ended at Feb. 2004. After that, at Mar. 2004, MCPP V.2.4.1 was released. In this version, I revised the recursive macro expansion.

9 Update Plans for V.2.5

Following updates are planned for MCPP V.2.5.

1. Since MCPP provides a framework to facilitate addition of multi-byte character encodings, MCPP will support more encodings, including ISO-2022-$*$.

2. MCPP diagnostic messages will be stored in a separate file so that anyone can add diagnostic messages in various languages at any time.

3. GNU C 3/cpp source programs and its testsuite will be given more consideration.

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

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

10 Conclusion

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 have extended for less than two years, I am very proud to have finally completed a C preprocessor that I think ranks number ONE in the world. I am now ready to send out MCPP and its English documents into the world for international evaluation. 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", Niibe project manager created a cvs repository, a ftp site, and web pages in m17n.org. [19] The latest versions, including revisions under development, are stored here.

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

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

[10] J. Roskind, JRCPPCHK.
At present, the site address is unknown.

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

[12] DJ Delory, DJGPP 1.12 m4.
`http://www.vector.co.jp/soft/dos/prog/se016978.html`

[13] Akira Kida, LSI C-86 / cpp revised beta13.
This software was available in CD-ROM of *C Magazine* in the past.

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

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

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

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

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

[19] Kiyoshi Matsui, MCPP V.2.4.1.
`http://www.m17n.org/mcpp`

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